The Cedar Manual

Version 4.2

Release as [Indigo]<Cedar>Documentation>Manual.df
Came from [Indigo]<CedarDocs>Manual>Manual.df
Last edited By Jim Horning on June 8, 1983 6:45 pm

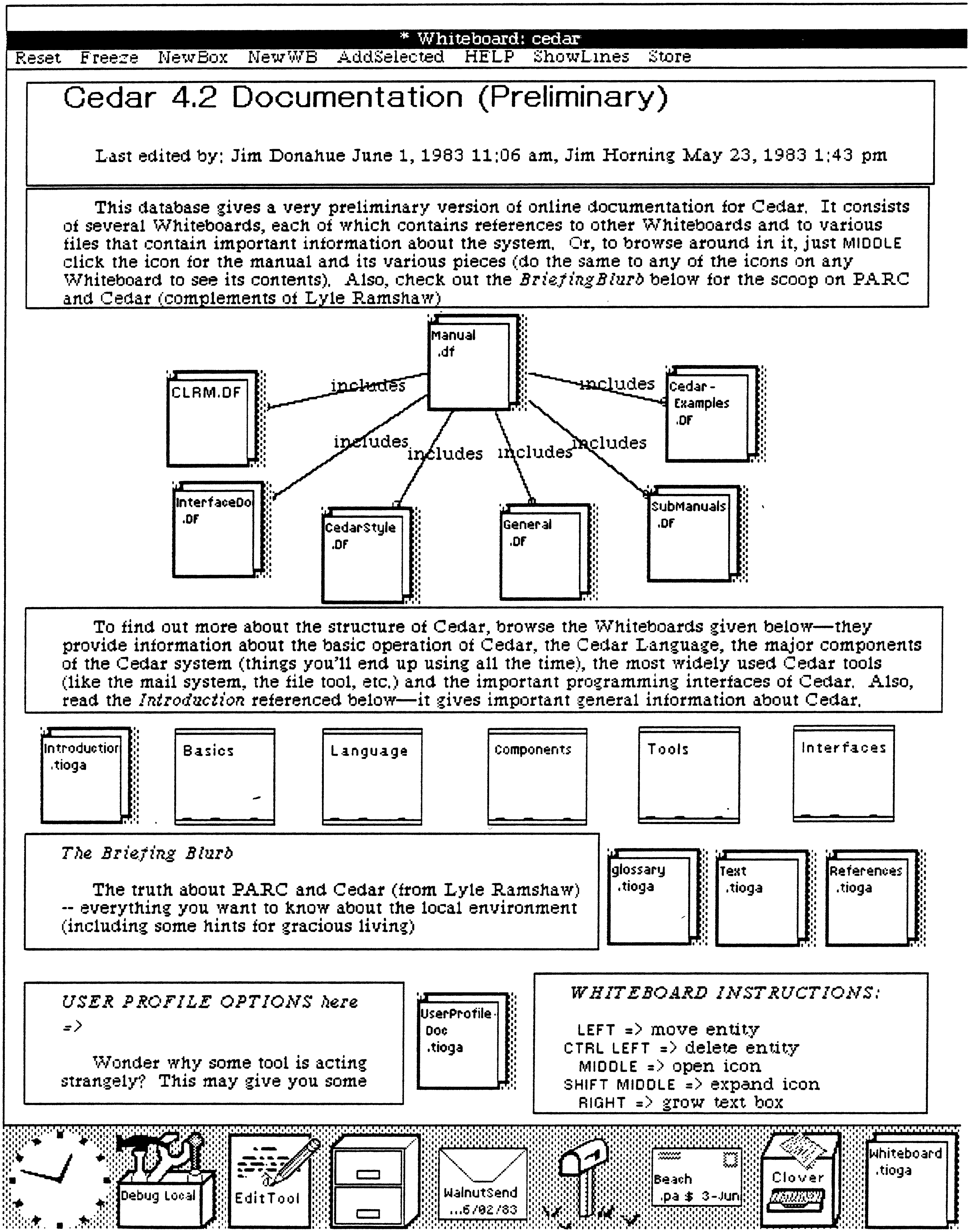
Abstract: Cedar is a new computing environment developed by CSL and ISL for use on D-machines—Dorados, Dolphins, and Dandelions. This collection of documents describes Release 4.2 of Cedar, June 1983. It consists of a number of sections that have been separately written, filed, updated, and checked. They may not be entirely consistent—the authors would appreciate being informed of inconsistencies, either internal to the documentation, or between the documentation and the system. Because various part of the system are still under intensive development, any hardcopy documentation is probably out of date. Each of the major components begins with a title sheet that lists the file name of its working ("came from") version, which may be more up-to-date. In the following table of contents, items marked with * are short summaries; you should obtain loose copies for quick reference.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

DRAFT - For Internal Xerox Use Only - DRAFT

Contents



Cedar 4.2 Documentation (Preliminary)

Last edited by: Jim Donahue June 1, 1983 11:06 am, Jim Horning May 23, 1983 1:43 pm

This database gives a very preliminary version of online documentation for Cedar. It consists of several Whiteboards, each of which contains references to other Whiteboards and to various files that contain important information about the system. Or, to browse around in it, just MIDDLE click the icon for the manual and its various pieces (do the same to any of the icons on any Whiteboard to see its contents). Also, check out the *BriefingBlurb* below for the scoop on PARC and Cedar (complements of Lyle Ramshaw)

To find out more about the structure of Cedar, browse the Whiteboards given below—they provide information about the basic operation of Cedar, the Cedar Language, the major components of the Cedar system (things you'll end up using all the time), the most widely used Cedar tools (like the mail system, the file tool, etc.) and the important programming interfaces of Cedar. Also, read the *Introduction* referenced below—it gives important general information about Cedar.

The Briefing Blurb

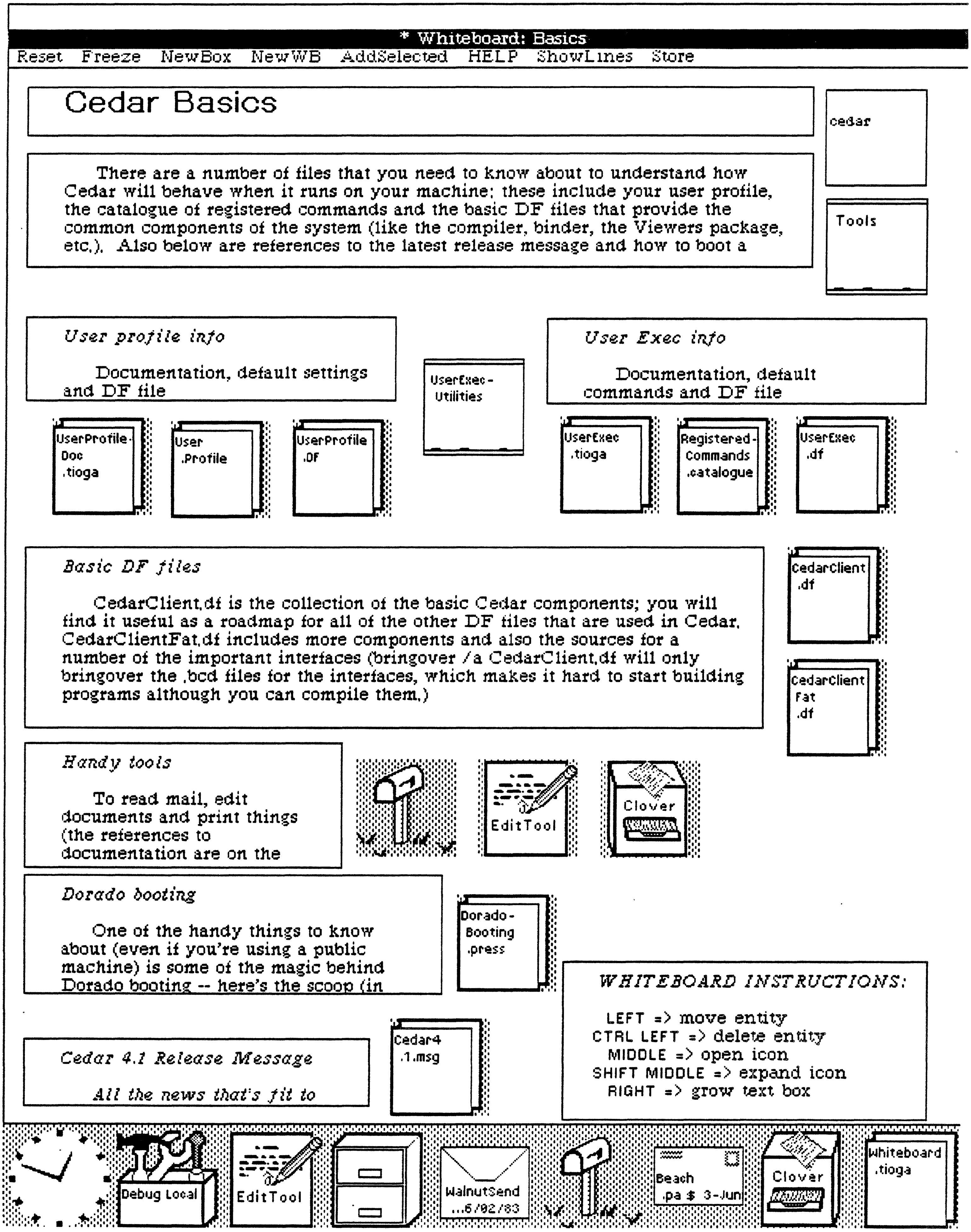
The truth about PARC and Cedar (from Lyle Ramshaw) -- everything you want to know about the local environment (including some hints for gracious living)

USER PROFILE OPTIONS here =>

Wonder why some tool is acting strangely? This may give you some

WHITEBOARD INSTRUCTIONS:

LEFT => move entity
CTRL LEFT => delete entity
MIDDLE => open icon
SHIFT MIDDLE => expand icon
RIGHT => grow text box



Cedar Basics

There are a number of files that you need to know about to understand how Cedar will behave when it runs on your machine; these include your user profile, the catalogue of registered commands and the basic DF files that provide the common components of the system (like the compiler, binder, the Viewers package, etc.). Also below are references to the latest release message and how to boot a

User profile info

Documentation, default settings and DF file

User Exec info

Documentation, default commands and DF file

Basic DF files

CedarClient.df is the collection of the basic Cedar components; you will find it useful as a roadmap for all of the other DF files that are used in Cedar. CedarClientFat.df includes more components and also the sources for a number of the important interfaces (bringover /a CedarClient.df will only bringover the .bcd files for the interfaces, which makes it hard to start building programs although you can compile them.)

Handy tools

To read mail, edit documents and print things (the references to documentation are on the

Dorado booting

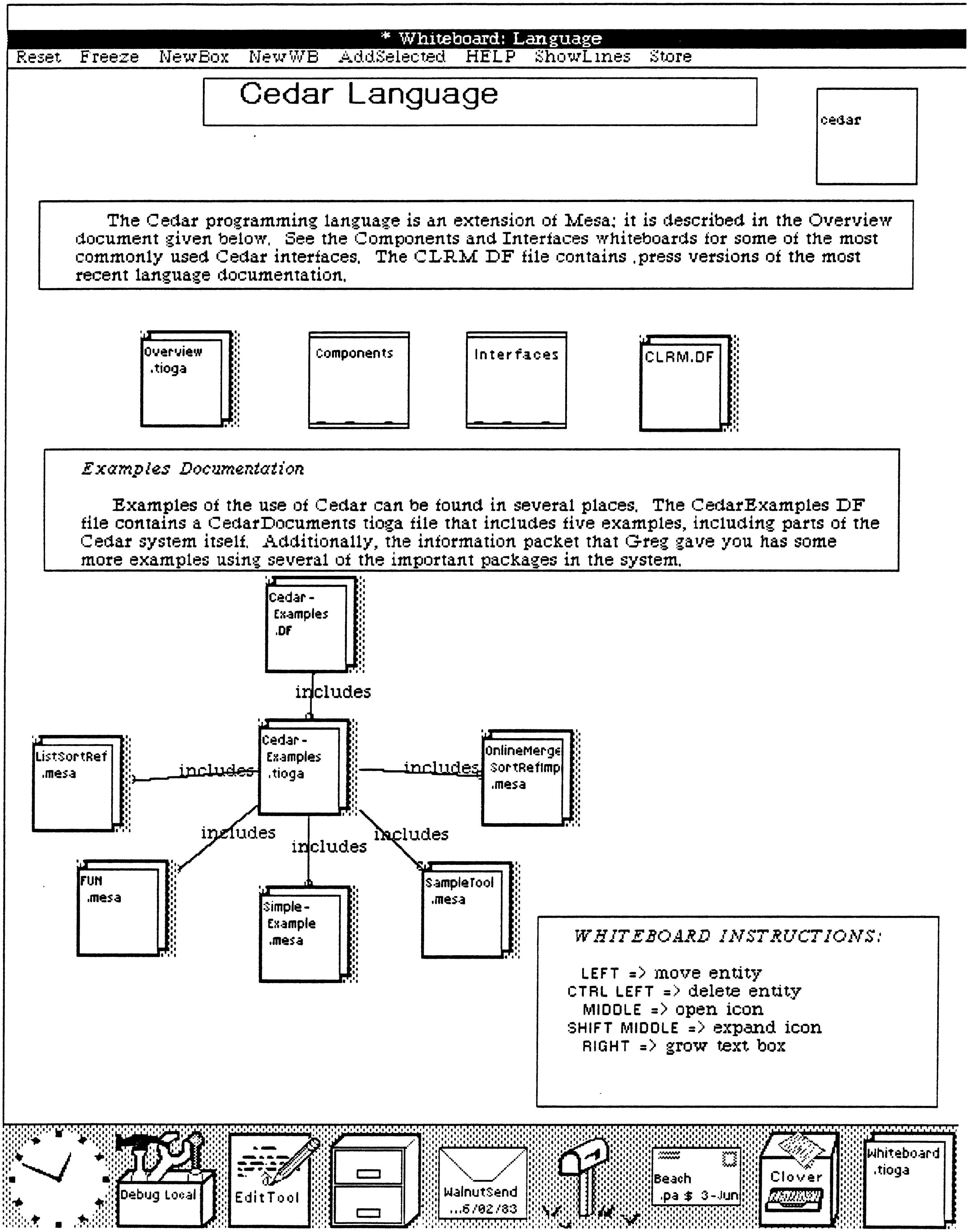
One of the handy things to know about (even if you're using a public machine) is some of the magic behind Dorado booting -- here's the scoop (in

Cedar 4.1 Release Message

All the news that's fit to

WHITEBOARD INSTRUCTIONS:

LEFT => move entity
CTRL LEFT => delete entity
MIDDLE => open icon
SHIFT MIDDLE => expand icon
RIGHT => grow text box



* Whiteboard: Language

Reset Freeze NewBox NewWB AddSelected HELP ShowLines Store

Cedar Language

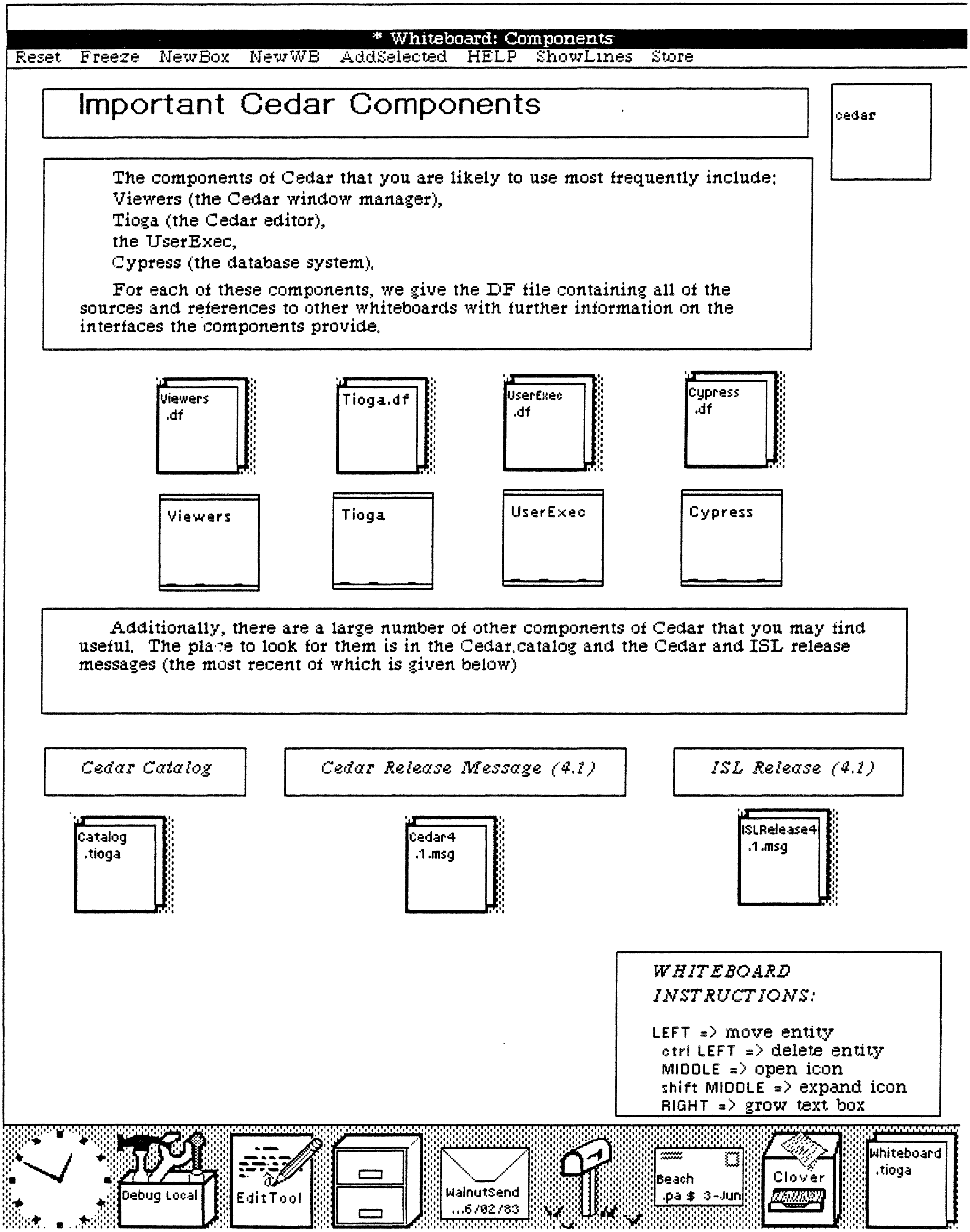
The Cedar programming language is an extension of Mesa; it is described in the Overview document given below. See the Components and Interfaces whiteboards for some of the most commonly used Cedar interfaces. The CLRM DF file contains .press versions of the most recent language documentation.

Examples Documentation

Examples of the use of Cedar can be found in several places. The CedarExamples DF file contains a CedarDocuments tioga file that includes five examples, including parts of the Cedar system itself. Additionally, the information packet that Greg gave you has some more examples using several of the important packages in the system.

WHITEBOARD INSTRUCTIONS:

LEFT => move entity
CTRL LEFT => delete entity
MIDDLE => open icon
SHIFT MIDDLE => expand icon
RIGHT => grow text box



Important Cedar Components

The components of Cedar that you are likely to use most frequently include:
Viewers (the Cedar window manager),
Tioga (the Cedar editor),
the UserExec,
Cypress (the database system).

For each of these components, we give the DF file containing all of the sources and references to other whiteboards with further information on the interfaces the components provide.

Additionally, there are a large number of other components of Cedar that you may find useful. The pla:e to look for them is in the Cedar.catalog and the Cedar and ISL release messages (the most recent of which is given below)

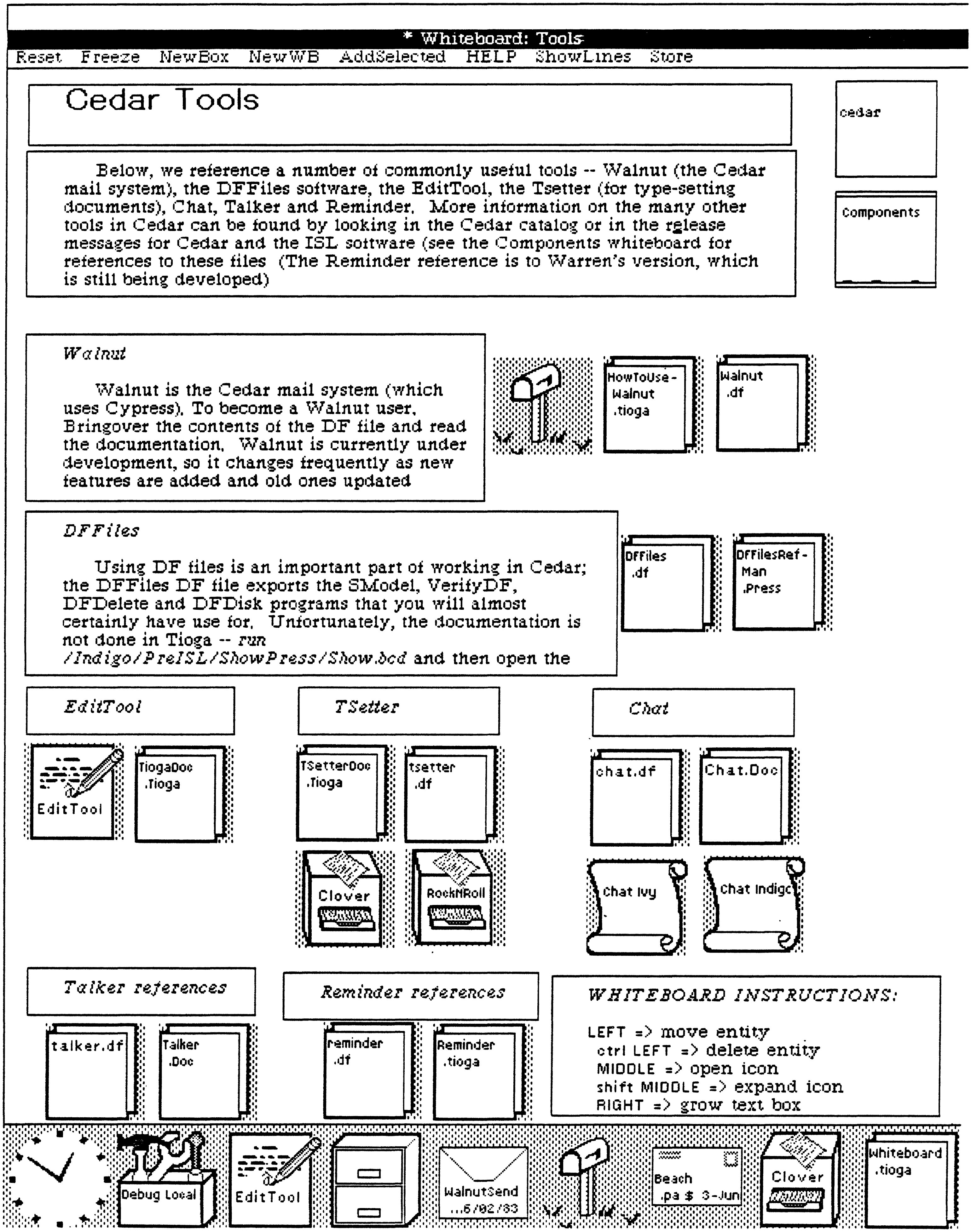
Cedar Catalog

Cedar Release Message (4.1)

ISL Release (4.1)

WHITEBOARD INSTRUCTIONS:

LEFT => move entity
ctrl LEFT => delete entity
MIDDLE => open icon
shift MIDDLE => expand icon
RIGHT => grow text box



Cedar Tools

Below, we reference a number of commonly useful tools -- Walnut (the Cedar mail system), the DFFiles software, the EditTool, the Tsetter (for type-setting documents), Chat, Talker and Reminder. More information on the many other tools in Cedar can be found by looking in the Cedar catalog or in the release messages for Cedar and the ISL software (see the Components whiteboard for references to these files (The Reminder reference is to Warren's version, which is still being developed)

Walnut

Walnut is the Cedar mail system (which uses Cypress). To become a Walnut user, Bringover the contents of the DF file and read the documentation. Walnut is currently under development, so it changes frequently as new features are added and old ones updated

DFFiles

Using DF files is an important part of working in Cedar; the DFFiles DF file exports the SModel, VerifyDF, DFDelete and DFDisk programs that you will almost certainly have use for. Unfortunately, the documentation is not done in Tioga -- *run /Indigo/PreISL/ShowPress/Show.bcd* and then open the

EditTool

TSetter

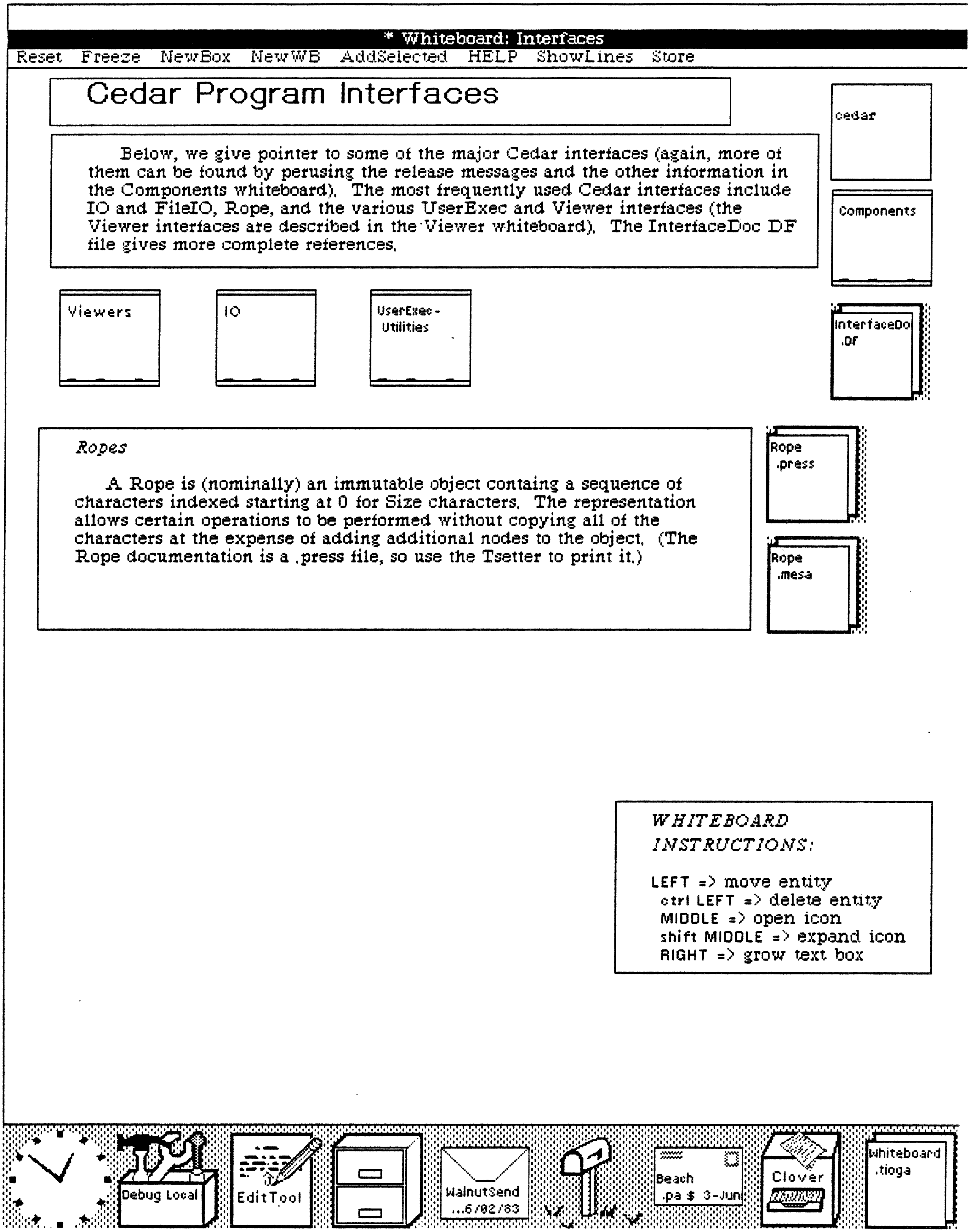
Chat

Talker references

Reminder references

WHITEBOARD INSTRUCTIONS:

LEFT => move entity
ctrl LEFT => delete entity
MIDDLE => open icon
shift MIDDLE => expand icon
RIGHT => grow text box



Cedar Program Interfaces

Below, we give pointer to some of the major Cedar interfaces (again, more of them can be found by perusing the release messages and the other information in the Components whiteboard). The most frequently used Cedar interfaces include IO and FileIO, Rope, and the various UserExec and Viewer interfaces (the Viewer interfaces are described in the Viewer whiteboard). The InterfaceDoc DF file gives more complete references.

Ropes

A Rope is (nominally) an immutable object containg a sequence of characters indexed starting at 0 for Size characters. The representation allows certain operations to be performed without copying all of the characters at the expense of adding additional nodes to the object. (The Rope documentation is a .press file, so use the Tsetter to print it.)

WHITEBOARD INSTRUCTIONS:

LEFT => move entity
ctrl LEFT => delete entity
MIDDLE => open icon
shift MIDDLE => expand icon
RIGHT => grow text box

Introduction to Cedar

Version 4.2

Release as	[Indigo]<Cedar>Documentation>Introduction.tioga, .press
Came from	[Indigo]<CedarDocs>Manual>Introduction.tioga, .press
By	Jim Morris, Mark Brown, *et al.*
Last edited	By Scott McGregor on October 12, 1982 9:57 am
	By Jim Horning on December 20, 1982 6:12 pm
	By Ed Taft on June 1, 1983 11:33 am
	By Warren Teitelman on June 2, 1983 11:18 am

Abstract: This memo is a sort of operators' manual for acquiring and using Cedar. It explains the minimum you need to to know about most things, depending upon other documents for the full story.

This memo is probably out of date if it is in hardcopy form. It is intended to document Release 4.2 of Cedar, June 1983, but some sections still reflect earlier releases.

[If you are reading this document on-line, try using the Tioga Levels and Lines menus (if you can) to initially browse the top few levels of its structure before reading it straight through.]

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

DRAFT - For Internal Xerox Use Only - DRAFT

Introduction to Cedar: Contents

0. Introduction

Cedar has a small, close-knit user community and it is changing very fast. Much useful information is not written down or appears in informal messages. To learn about using the system you must take some lessons from someone; get them to show you how to start a D-machine, how to use the mouse, etc. To work effectively you must keep in touch with what is going on. If you are using Cedar put your name on the CedarUsers mailing list. (See the section 1.6 for instructions on how to do it.) If you would like to participate in or listen to design discussions, put your name on CedarDiscussion. If you are just generally interested in what is going on try CedarInterest. Questions and bug reports should be sent to CedarSupport—currently John Maxwell. Questions about a particular section this memo can be addressed to the person(s) listed at the beginning of that section. Questions about specific packages and problems should be addressed to the maintainers listed in the Catalog, with copies to CedarSupport; but, if you're not sure whom to ask, the following people have a general knowledge of how to use the system: Atkinson, Brown, Morris, Levin, Maxwell, McGregor, Paxton, Rovner, Satterthwaite, Schmidt.

Typographical conventions employed herein:

Names of keys appear in capital letters in a small, alternate font; e.g., SHIFT, ESC, RETURN
↵ is sometimes used for RETURN.

Things that are typed or displayed on the screen appear in an alternate font; e.g., **compile foo**

When we describe fancy interactions in which a system completes commands, what you actually type is underlined; e.g., OthelloDorado.eb_

1. The Cedar World

To use Cedar you must first find a Dorado or Dolphin.

Cedar will usually be found in its *idle* state, displaying the words "Type Key" in the cursor. After pressing a key on the keyboard, you will be asked to supply your name and password.

To awaken a slumbering Dorado press its boot button three times and wait for something to appear on the screen (a minute or so).

To start Cedar on a wakened Dorado hold down the C (for Cedar) key and boot the machine by pressing the boot button three times.

To turn on a Dolphin press the start button on its maintenance panel and wait for something to appear on the screen (a couple of minutes).

To start Cedar on an awakened Dolphin hold down the P (for Pilot) key and boot the machine by pressing the boot button once.

If Cedar is properly installed on the machine the cursor will turn into a cedar tree, and you will be prompted for a name and password or just a password; if you don't know what to type, get help. If the version number at the top of the screen is larger than 4.2, check for a newer version of this memo. If it is smaller, get the new release using the instructions in section 1.7.2. If the screen says **Othello**. . ., typing **Rollback** or **Boot Client** may get you to Cedar. Otherwise, you have no Cedar world; consult section 1.7 on how to get your disk set up.

1.0 Credentials Roy Levin

Cedar expects its user to be an individual registered with Grapevine. Whenever you enter the Cedar world, you will be asked to supply your Grapevine RName and password. Once you have been authenticated, Cedar will remember your credentials until you either (1) push the boot button, (2) boot a non-Cedar partition (e.g., an Alto partition), or (3) push the "Idle" button in the extreme upper right corner of the screen. If you are not registered with Grapevine, contact your local support staff.

The precise form in which Cedar asks for your credentials depends upon the way in which the credentials were originally installed. Public machines and some personal machines (at their owners' option) have "unprotected" disks, meaning that Cedar will permit any individual recognized by Grapevine to log in. Other personal machines, however, have "protected" disks, meaning that Cedar will only allow a specific individual to log in. To change from a protected disk to an unprotected one or vice versa, use Othello's "Install Credentials" command.

1.1 Screen Management and Input Scott McGregor

The basis for screen management is the *Viewer*. In general, a viewer manifests itself as a rectangular area on the screen. Some viewers simply display text, others are virtual *buttons* that invoke procedures when clicked. We use the verb *click* to describe the acts of positioning the mouse-controlled cursor over a viewer then depressing and releasing a mouse button, usually the left one. *Middle click* means to depress the middle button, *right click* means to depress the right button, etc. In the past mouse buttons were imagined to be red, yellow, and blue scanning from left to right, so you will occasionally see that terminology.

Across the top of the screen is a small *message area* where various comments about the system's status and behavior will appear. If the message is especially important, the message window will flash to call the your attention to it. The large middle part of the screen is divided into two columns for displaying tools and documents. Most tools and documents will initially appear as *icons*—small pictures at the bottom of the screen. You can *open* an icon to see its contents by middle clicking it; holding down the SHIFT key while clicking makes it consume the whole column. Left clicking an icon *selects* it, making it the recipient of type-in from the keyboard. Icon keyboard commands include:

C	Move the icon to the color display (assuming you have the hardware).
DEL	Delete the icon.
L	Move the icon to the left column.
M	Move the icon to another column.
O	Open the icon (like middle clicking).
SHIFT-O	Open the icon full size (like SHIFT middle clicking).
R	Move the icon to the right column.

Open viewers display a menu of commands across the top when you move the cursor into the caption (the black band at the top containing the name of the viewer). The commands are as follows:

Destroy	Make the viewer disappear.
Adjust	Change the size of the viewer or size of the column (see below).
Top	Move the viewer to the top of the column.
<--	Move the viewer to the left column.
-->	Move the viewer to the right column.
Grow	Close all other viewers in the column.
Close	Make the viewer iconic.

Middle clicking in the caption menu always invokes the Grow command, and Right clicking always invokes Close. This allows you to invoke these frequently used operations without having to position the mouse as accurately.

The height of a viewer in a column is computed from a set of hints, some determined by programs and some indicated by the user. The program which created the viewer can specify a desired height (such as in the EditTool and Watch viewers) or the program can request that the viewer receive a "fair share" of the available space (as in Tioga text viewers). The user may override these program hints by clicking the *Adjust* caption menu command, entering a mode where a new height hint may be specified with the mouse. Moving the cursor out of the original column changes the mode to allow the user to specify a new column height and width. At any time while in adjust mode, simultaneously depressing two mouse buttons cancels the adjust command.

Some menu items and button will be displayed with a strikeout bar through the text. These are known as *guarded* commands, implying that command, if inadvertantly triggered, might cause loss of your current state. To invoke a guarded command, click once to remove the guard and then again to trigger (within a few seconds or the guard will reappear).

In order to type something into a viewer, you must first establish the *input focus* by clicking somewhere inside it. (If the viewer is a typescript viewer, i.e. one in which you and the system alternately insert characters, as opposed to a Tioga document, you should make sure that you click the mouse in the white space below the last character.) Left clicking in a text area positions the blinking *caret* that indicates where your typed characters will appear. A sequence of text characters may be selected by left clicking the first character and right clicking the last; they will appear video reversed, i.e., white on black. Those characters then become the *current selection* which various buttons (e.g., Open) treat as an input parameter. Whatever you type replaces the current selection when it is video reversed. There are many other things to learn about selection described in the Tioga manual [G1]. Tioga is generally similar to Laurel [G2].

There are some buttons at the right end of the message area. You can boot any of your system volumes (e.g., Alto, Client, Othello) by clicking Boot to bring up a set of guarded buttons. The New button creates a new text viewer that you can type new text into; typing a file name followed by LF will load a file into it. The Open button creates a new text viewer for the file named by the current selection. Clicking Idle causes the screen to turn black and display a dancing "Type Key" cursor. This is the preferred way of ending a session without actually leaving the Cedar world; various volatile structures will be made secure on the disk. If you subsequently hit a key, you will be asked to login, after which the screen will be restored to its state at the time that Idle was invoked. The **Fly** button will display a set of nine desktops, any one of which you can examine by left clicking. All of your icons are transferred to the new desktop.

The checkpoint facility allows you to save the effect of a long set-up computation such as the one that occurs after an installation of a release. Arm and click the **Checkpoint** button at the top of the screen and wait for a while (about four minutes on a Dolphin). Don't worry that the cursor won't move with the mouse. Then, whenever you start your Cedar world or click the **Rollback** button, you will find yourself in this state. It is important to understand that rolling back only restores the state of the virtual memory. It does not undo changes made to files or the directory. Thus you should create checkpoints only when the system is in a quiescent state with no open files: otherwise, strange things might happen after a rollback. If you want to overwrite a bcd (either by compiling or file transfer) that was loaded prior to a checkpoint you should boot the Client volume and create a new checkpoint. Don't make checkpoints on public Dorados unless you know how to restore the standard one when you are done.

1.2 User Exec **Warren Teitelman**

Note: The following is summarized and extracted from the documentation of Cedar UserExec, [G14], which also appears as as a separate section of the Cedar Manual. For more complete discussion, and lots of examples, refer to this documentation, which can also be found on your disk as the file UserExec.tioga.

General Comments

The Cedar executive is called the UserExec. It is an amalgam of the Alto Exec, a Cedar Language interpreter, and a debugger—backed up by optional automatic error correction facilities similar to InterLisp's DWIM. For example, the UserExec can be used to load and run bcds, list, copy, rename, and delete files, evaluate Cedar Language expressions, catch breaks and signals, and display the state of a process that has been stopped by a break or signal.

The user interacts with a particular UserExec (you can have several around, even executing, at the same time) through a special viewer called a WorkArea. Each WorkArea has a name, typically a single letter, which is displayed as part of its caption. Each WorkArea also has a *mode*, which is either Executive or Interpreter, also displayed as part of its caption. To "talk" to a particular UserExec, simply establish the input focus by clicking in the bottom portion of the corresponding viewer, and then start typing.

Since the WorkArea is a viewer, all of the viewer facilities are available for manipulating the WorkArea (see section 1.1). In particular, it can be grown, adjusted, scrolled, moved, closed (the UserExec will continue running and/or outputting characters, you just won't see them until you reopen the viewer), opened, split, or even destroyed (which will abort any operation that is executing the next time it tries to do input or output). Furthermore, since the viewer views a Tioga typescript, all of the Tioga editing and selection mechanisms can be used with respect to the command line that you are composing: you can edit this line to your hearts content, and when you terminate the command, it will look exactly like you typed in the edited line.

New UserExecs and their corresponding WorkAreas can be created via the **New** menu button which appears in the first line of each UserExec menu. This works even when a particular WorkArea seems to have "died". Existing UserExecs can be destroyed via the Destroy Viewer menu button. If you destroy your only WorkArea, an Exec button is posted at the top of the screen which you can use to create new WorkAreas.

Events

Each user interaction with a UserExec is called an *event.* At the start of each event, the user is prompted by & followed by the event number. The user then types in a command line consisting of the name of a registered command, followed by its arguments, if any, and terminated by ↵, ?, CTRL-X, or ESC. The UserExec then performs the indicated operation, prints the result, and prompts the user for the next event. Typing DEL during the input of a command will abort the input, causing you to be reprompted. Clicking the **Stop** menu button (or typing CTRL-DEL) during the exeuction of an event will cause it to abort the current operation (but sometimes it takes a little while).

Terminating a command line with ? signifies a request for additional information. Specifically, ? by itself prints a list of commands currently registered, ? after a registered command prints a description of the command, and ? after a Cedar expression prints the type of (the value of) the expression. For example:

```
    &4 walnut?
Walnut        Creates a viewer for sending or retrieving mail.
&5 ← 3.2?
is of type REAL
&6 ← Rope.Cat?
is of type PROC [r1, r2, r3, r4, r5, r6: ROPE ← NIL] RETURNS [ROPE];
  — returns the concatenation of up to six ropes (limit based on eval stack depth)
  — BoundsFault occurs if the result gets too large
```

Terminating a command with CTRL-X means to "expand" the command, but not to execute it, i.e. e.g. perform * expansion. Terminating a command with ESC means to "complete" the command, as far as possible, but not to execute it.

Registered Commands

The following are some of the more useful registered commands. More commands can be discovered through the use of ?.

@ takes a file name as argument. Treat the contents of the named file as a command file, i.e. interpret the text as a sequence of commands. If {file} has no extension, look for a file of the form {file}.**commands** or {file}.cm.

← Treat the remainder of the input line as a mesa expression to be evaluated. Evaluate the expression and print its value. If the expression is terminated with ?, print the type of its value, rather than the value. If the expression is terminated with !, print the value showing the referents of all REFs and POINTERs to an unlimited depth.

Note: many users prefer to do interpretation of expressions in Interpreter WorkAreas or Action WorkAreas (see below), in which case the ← is automatically provided at the beginning of each command line, and to use Executive WorkAreas for "executive" type of operations such as running programs and manipulating files.

Bind Bind a list of configurations. See discussion of **Compile** menu button below.

Bringover Retrieve files using a specified df file (see 1.3.2)

ChangeAreaMode change an Executive WorkArea to an Interpreter WorkArea and vice versa.

Compile Compile a list of modules.

Copy Copy contents of one or more files to another. Syntax is **Copy new ← old1 old2 ... oldn.**

Date Type today's date and time.

Delete Delete a list of files.

Fetch Copies remote file(s) to corresponding local file(s).

Help	Provide more complete explanation of UserExec in a separate viewer.
List	Print size and creation date for the indicated files.
ListByDates	Print size and creation date for the indicated files, sorted by create date.
Login	Supply user name and password.
Rename	Rename a file. Syntax is Rename new ← old.
Run	Load and Start the named programs.
SModel	Store files on remote servers using a specified df file. (see section 1.3.2)
TSetter	Create a typesetter viewer for specified server, and use it to print named documents
User	Type the name of the logged-in user.
Walnut	For sending or retrieving mail.

For convenience, a number of commonly used registered commands can also be invoked via menu buttons that appear in the first line of the menu of each WorkArea. If it isn't obvious what these menu buttons do, consult [G14].

Interpreter WorkAreas

When typing to an Interpreter WorkArea, the user is always prompted with &nn ←, where nn is the event number. What the user types following the ← is treated as an expression to be evaluated in the current context and default global context, if any, for the WorkArea. The value of the expression will be assigned to the variable whose name precedes the ←, i.e. &nn. This value can be referenced in later expressions. As mentioned earlier, if ? is typed following an expression, the type of the expression, plus other explanatory information, is printed.

The following is taken from an actual session with an Interpreter WorkArea. The italicized text at the right is added commentary not printed by UserExec.

```
    &2 ← Rope.Cat["Ce", "dar"]↵  Note that Rope is the interface, not the implementation.
"Cedar"

    &3 ← LIST[1, 3.2, &]↵        & evaluates to the previous result (&2 in this case.)
(↑1, ↑3.2, "Cedar")

    &4 ← &3?                     What type of list did the interpreter produce?
is of type LORA: TYPE = LIST OF REF ANY

    &5 ← &3.first↑?
is of type INT

    &6 ← &3.rest.first↑?
is of type REAL

    &7 ← List↵
default global context changed to: ListImpl
{globalFrame: ListImpl}

    &8 ← Appendd[Reverse[&3],&3]]↵         Note that Reverse is now interpreted as List.Reverse.
Appendd -> Append ? Yes                   Spelling correction.
("Cedar", ↑3.2, ↑1, ↑1, ↑3.2, "Cedar")
```

For more detailed information about exactly what subset of Cedar language expressions the interpreter can handle see section 2.3.

Action WorkAreas

Actions occur when a program raises a signal or error that is not caught or encounters a breakpoint. Whenever an action occurs, the corresponding process is stopped so that it can be examined, and control transfers to a different WorkArea called an Action WorkArea, or ActionArea for short. An ActionArea is an Interpreter WorkArea whose default context is the context of the action. The user can then walk the stack and evaluate expressions. The user can also choose to ignore the action for the time being and type some other command in a different WorkArea. If the user does not wish to pursue the cause of the action at all, the simplest way to make it "go away" is to click **Abort**. For complete discussion of Action Areas and ActionArea Commands, see [G14].

Using the History facility

Each WorkArea has associated with it a history of all of the events that have taken place in that WorkArea. The user can examine this history via the **History** registered command, reexecute a particular event or events using the **Redo** command, or substitute new parameters (text strings) into a particular event or events and then reexecute them via the **Use** command.

Confirmation

Occasionally the UserExec will attempt to correct an error: e.g. a misspelled file name, an invalid selector, syntax error, etc. In this situation, two new menu buttons, Yes and No, will be posted in the menu for the corresponding WorkArea. Depending on the settings in the user's profile (see UserProfile.doc), some errors will be corrected automatically, but in other cases, confirmation will be requested. When/if the user is asked to confirm (depending on the settings in the user's profile, some errors may be corrected automatically), the user can confirm using these buttons, or by typing Y or N. If the user has typed ahead before the need for confirmation was detected, the typeahead will be retained, and the user must confirm using the **Yes** and **No** buttons.

1.3 Files

All of the material you are working on, including programs, is stored in *files*. Each different document you handle will be stored on its own file. The file system is somewhat complicated by the fact that it spans a network and developed in an evolutionary fashion.

1.3.1 Local and Remote Files *Ed Taft*

A file on your local disk is identified by its name, which is a string of letters (upper and lower case can be used interchangeably), digits, and any of the punctuation characters +- . $. By convention, a simple file name has two parts, which are called the *main name* and the *extension*: they are separated by a period. For example, "Introduction.tioga" is a file name, with main name "Introduction" and extension "tioga". File names cannot include blanks, or any punctuation characters except the ones just mentioned.

It is important to name your files in some systematic way, using the main name to identify it, and the extension to tell what kind of file it is. Unless there is a good reason to do otherwise, it is best to use one of the standard extensions given below.

Here is a list of extensions commonly encountered:

.bcd	Cedar object program
.config	system configuration, input to binder
.cm	command file for the User Exec or other programs
.doc	Tioga document (old convention)
.df	list of dated files for use in moving files between machines
.mesa	Cedar or Mesa source code

.model	system model
.press	Press-format file, suitable for printing
.tioga	Tioga document

The system doesn't care whether you capitalize letters in file names or not (i.e., ALPHA, alpha, and aLpHa refer to the same file), but it is a good idea to use capitalization to make names more readable. This is especially useful when a name consists of more than one word, since blanks are not allowed in file names: e.g., TripReport or MasterList. A file name with the form X$ is taken to be an older, backup version of X. Many subsystems will save the previous version under such a name.

File servers are large repositories for files. A file server's disk typically has hundreds of times the capacity of your local disk. Besides providing back-up for your local disk, they are the only reasonable places to put files you wish others to see or want to access yourself from different machines. The only reasonable way to do business is to keep your personal files backed up on a remote server. You should not rest easily unless the latest versions of all your important files are on a remote server somewhere.

In general, the name of file in network has the form

[Server]<directory>subDirectories>name.extension!version

This form is sometimes called the *full path name.* The server is the name of the machine. Indigo, Ivy, and MAXC are the local servers; the first two are instances of IFS—the interim file system. The directory is the name of a project or person. Each user who has an account on a file server has his own directory, named by his user name. Files within a directory may be organized into *sub-directories* (except on MAXC). For example, the file named

<Jones>Memos>ActivityReport.bravo!3

belongs in directory Jones, sub-directory Memos. You can have as many sub-directories as you wish within your own directory. You can even have sub-directories within sub-directories, to as many levels as you wish, subject to an overall limit of 99 characters in each file name. Subdirectories are entirely a naming convention based upon the use of the character >; there are no special operations for dealing with subdirectories. The name and extension serve the same purposes as on the local machine. When you put a file onto a file server, if there is already a file with the same name, the new file is added, with a version number one bigger than the old one. When you reference a file without specifying a version you get the one with the largest version number. As you can see, it is almost never necessary for you to specify a version number explicitly. (On MAXC, the character ; is used instead of ! to prefix a version number.) Each file in the system carries a *create time*: the time when the content of the file was created. This attribute serves as a server-independent version stamp.

You can name a group of files by using file name *patterns* containing the magic character * which stands for any string of characters. For example, the pattern *.memo stands for all the files which have the extension "memo", and the pattern *.BWL* stands for all the files which have "BWL" as the first three characters of the extension.

1.3.2 DF Files *Eric Schmidt*

As soon as you find yourself dealing with multiple files, you should devise *DF files* to help you back up and retrieve them. A DF file is a human-readable file that describes a list of files with their create dates. The simplest way to create a DF file is to list all the files of interest in a file and apply the command SModel to it. For example, to create the DF file describing all the files mentioned in a previous version of this memo we created the file init.df with the following content:

```
Directory [indigo]<Cedar>init>
  Init.df
  AltoSupport.cm
  Cedar.cm
  D0.cm
  D0Release2.5.1.cm
  Dorado.cm
  DoradoRelease2.5.1.cm
  RunPilot.bcd
Directory [indigo]<Cedar>documentation>
  GettingStartedInCedar.memo
  GettingStartedInCedar.press
```

We then typed

```
SModel init
```

to the User Exec. This transfered all the files to the remote directories, including an updated version of init.df that contained the version numbers and create times of the files:

```
Directory [indigo]<Cedar>init>
  Init.df
  AltoSupport.cm!1                  22-Mar-82  9:47:24 PST
  Cedar.cm!1                        18-Mar-82 14:57:23 PST
  D0.cm!2                           22-Mar-82 10:38:37 PST
  D0Release2.5.1.cm!3               22-Mar-82 10:55:02 PST
  Dorado.cm!3                       22-Mar-82 12:33:35 PST
  DoradoRelease2.5.1.cm!1           22-Mar-82 10:11:30 PST
  RunPilot.bcd!1                    2-Feb-82 23:37:34 PST
Directory [indigo]<Cedar>documentation>
  GettingStartedInCedar.memo!1      22-Mar-82 14:02:23 PST
  GettingStartedInCedar.press!1     22-Mar-82 14:03:11 PST
```

Once you have a DF file, you can use the BringOver and SModel programs to manage the movement of your system.

```
BringOver /a init
```

makes sure the local disk contains the proper versions of the files in init.df by retrieving new versions automatically (/a) if needed. You should use full path names to specify a DF file if you want the remote version to control things; e.g.

```
BringOver /a [Indigo]<Cedar>init>init
```

After making changes to files, you can use SModel to write out the new versions; it compares the create times on the local files with those in the DF file and transfers files when they differ. You can nest DF files if your package requires another package. All of this is a simplification; see [G3] for more information. If you are providing a component of a Cedar release you must read [G4] to learn the proper way to use DF files for a Cedar release. You can learn a lot by looking at the DF files on [Indigo]<Cedar>Top>.

1.3.3 The File Tool *Larry Stewart*

The File Tool—its iconic form looks like a file cabinet—is used for listing files on various machines and transferring them. Its topmost section provides various fields for you to type in things and a few mode buttons; click the field name and the cursor will move to its entry. The * notation can be used in the first three fields to designate groups of files.

Directory designates a remote server and subdirectories; e.g., [Indigo]<Cedar>Top>. The * notation can be used to designate multiple places.

Filename(s) is a sequence of files to be transferred or listed; subdirectories may be included; e.g., Docs>Intro.press. Remote file names are derived by concatenating Directory with these names. The * notation can be used to designate multiple files. Entering @X will cause the contents of file X to be used.

Local refers to local file names. It can be used if you wish to rename a single file as it is being transferred. On a retrieve it names the destination file; on a store it names the source. The * notation can be used to designate multiple files. Entering @X will cause the contents of file X to be used.

DF File is the name of a DF file that fetch may be done through. See **DFGet**.

Connect Name and **Password** are needed for access to certain directories. You may omit the password when connecting to a directory belonging to a project of which you are a member.

Update is a button that sets the mode of operation so that a file will be moved only if a version of it already exists at the destination and has an earlier create date than the file to be moved.

Update>a does the same thing except the file will be moved even if it doesn't exist at the destination; most people prefer it.

ExportsOnly applies to fetches done through DF files.

Verify sets the mode so that you must confirm each transfer by clicking buttons that will be presented to you in the middle section

The second section holds a set of buttons that represent commands.

Retrieve fetches the remote files given by the combination of **Directory** and **Filename(s)**. If no local name is given the short form of the remote one is used.

Store moves files to a remote machine. If no **Local** entry is present it uses the Filename(s).

Local-List displays information about the local files.

Remote-List displays information (version, size, creation time) about the remote file.

List-Options brings up a set of buttons that change the things printed out by the List commands. **Apply** causes the new settings to take effect; **Abort** resets to the old settings.

Close shuts down the remote connection

DFGet retrieves the files listed in **Filename(s)** using a DF file to discover their full path names. **Directory** is prepended to **DF File** to define the full path name of the DF file itself.

DFGetBoth is the same as **DFGet** but fetches the .mesa and .bcd for each name listed in **Filename(s)**.

Local-Delete deletes the local files; it must be armed. *Warning:* Unlike certain systems, once you have deleted a file, you cannot get it back. Proceed with caution.

Remote-Delete deletes the remote file; it must be armed.

The third section is an output typescript. The **Stop** button in the top-most menu can be used to abort transfers and lists in an orderly way.

1.3.4 Chat and File Space Management *Larry Stewart, Ed Taft*

Sooner or latter you will run out of disk space on your IFS or MAXC directory. File space management activities not supported by the File Tool or DF files can be carried out by connecting to a file server with Chat. Chat uses a viewer to simulate a teletype computer terminal, and thereby enables you to talk directly to executive programs running in various server machines.

To initiate a conversation with the executive in a server type **Chat server -l** to the User Exec. If all goes well, you will see a message from the server's executive and @ at the left margin prompting you for type-in. If Chat has trouble getting connected, it will tell you its problem after trying for a few seconds. This usually means that the server is broken; you might try again in a few minutes. To redirect an existing Chat viewer to a sever type the server name into a window, select the name, and click **Login**. If you click **Connect** rather than Login, you can login by hand: type

```
@Login (user) name (password) password
```

Whatever the server executive is doing, you can force it to stop by typing CTRL-C. On MAXC, you may have to type CTRL-C several times in quick succession to get it to stop.

When you are finished talking to the server Executive, type

```
@Logout
```

(or **Quit** if the server is an IFS). Then click **Disconnect**. If the file server is an IFS, you will be logged out automatically if you don't type anything for three minutes. This is because IFS can service only a small number of users (currently nine) at once; the automatic logout is intended to prevent IFS from being tied up by users who aren't doing anything useful. Simply closing a chat viewer does not shut down the connection unless you right click the **Close** button.

You type commands to IFS and to MAXC in more-or-less the same way (except for those commands that have different names on the two systems); however, the responses from IFS and MAXC are usually somewhat different. You may type ? at any point to obtain a brief explanation of what you are expected to type in next. MAXC normally does not display the remainder of abbreviated commands; however, you can force it to do so by terminating fields you type in with ESC rather than space.

To delete all old versions of files (i.e., all but the highest-numbered version of each file), on IFS type

```
@Delete *,
@@Keep (# of versions) 1
@@Confirm (all deletes automatically)
@@
```

on MAXC type

```
@Delver
Delete oldest? Yes
Delete 2nd newest? Yes
File(s):_
```

It is a good idea to do this fairly frequently, since old versions of files can pile up and waste a lot of space. To find out how much space you are using on the file server, type

```
@DskStat
```

One IFS or MAXC page is equivalent to about four D-machine pages. You will notice that you also have a *disk limit* which is the maximum number of pages you are permitted to use on the file server at one time. If you exceed your disk limit, the server won't let you store any more files until you first delete some existing ones to get you below your disk limit. To get your limit changed, consult your local support staff.

You can direct your attention to some other directory by typing

```
@Connect (to directory) OtherDir (password) password
```

You may omit the password when connecting back to your own directory, or when connecting to a directory belonging to a project of which you are a member.

MAXC provides facilities for *archiving* files onto magnetic tape, where the cost of storing them is negligible. You can get an archived file back within one day. To archive one or several files, type

```
@Archive File file1 file2 . . .
```

(Note that the command name consists of the two words "Archive File"; after that you should type the names of the files you want to archive.) The files will be archived onto tape within a day or two.

After this has been done, they will be deleted from the disk automatically, and you will get a message notifying you that the archiving has been done.

MAXC keeps track of your archived files in an *archive directory* which you can list exactly like your regular MAXC directory, using the Interrogate command rather than the Directory command; for example,

```
@Interrogate *.bravo
```

If the listing is of just one file, MAXC will ask you whether or not you want it retrieved from the tape. If you say Yes, the file will appear on your MAXC directory within a day, and you will get a message to that effect.

Because MAXC's disk capacity is fairly small relative to the number of users who have MAXC accounts, the disk occasionally becomes full and it becomes necessary for a *forced archive* to be performed in order to make some space available. In a forced archive, all files that haven't been referenced (retrieved, printed, or whatever) in the past 90 days are written onto tape and deleted. You will be notified when any of your files are archived for this reason, and the procedure for getting them back is the same as given above.

1.4 User Profile **Warren Teitelman**

A number of components of Cedar permit the user to tailor Cedar's behavior along certain predefined dimensions via a mechanism called the *user profile.* Whenever you boot or rollback, your user profile is consulted to obtain the value for these parameters. This operation is performed by consulting a file whose name is <YourName>.profile, e.g., MBrown.Profile, or if no such file exists, User.Profile. The entries in this profile are of the form

```
Key: Value
```

where, for any given key, the value is expected to be either TRUE/FALSE, a number, or a token (a sequence of characters delimited by SP, CR, TAB, COMMA, COLON, or SEMICOLON, or an arbitrary sequence of charcters delimited by quotes), or a sequence of tokens. Comments can appear at any point in the profile, and are ignored.

More information may be found by examining UserProfile.doc, which lists all the currently available options.

1.5 Walnut **Rick Cattell**

The program for reading and sending electronic mail in Cedar is called Walnut. It uses the database management system as a repository for the messages. The documentation for Walnut can be found in the file [Indigo]<WalnutDoc>HowToUseWalnut.press; a copy of this documentation appears as a later chapter of this manual.

1.6 Maintain **Andrew Birrell**

Various administrative tasks associated with mail, authentication and other uses of Grapevine can be performed with the Maintain command. Bringover and RunAndCall Maintain. Its interface is layered according to the complexity of the operations various people need to perform. Many users will need only the level called "normal". This allows you to inspect distribution lists, add or remove yourself from lists, and change your password. When using Maintain, you must always specify names in full. Thus, you must say "CSL↑.pa", not "CSL↑", and "Birrell.pa", not "Birrell".

To look at a distribution list (a "group" in Grapevine terminology), fill in the text field labelled "Group" and click the "Members" button in the first line labelled "Type". The "Summary" button in that line will show you the access controls associated with that group (which control who may add or remove members). To add yourself to a group, fill in the "Group" field and click "Self" in the line labelled "Add". Similarly, you can remove yourself with the line labelled "Remove". Not all groups allow you to add or remove yourself. If you're not allowed to change the group, you should send a message to the owner of the group asking for the change. For example, you would send a message to "Owner-CSL↑.pa" to ask about "CSL↑.pa".

For the sake of security, it is a good idea to change your password occasionally (say, once a year). To do this, make sure your name is in the text field labelled "Individual", fill in your new password in the line below, labelled "Argument", then click "Password" in the line labelled "Set". Passwords should be at least six characters and unpronouncable. If you have an account on MAXC you will need to change the password there via Chat, type

```
@Change Password (of directory) name (old password) xxx (new password) yyy
```

1.7 Setting up your disk Eric Schmidt, Ed Taft

1.7.1 Getting to Othello

Othello is a general Pilot utility for setting up disks. There are variety of paths to Othello. On an arbitrary machine in an arbitrary state hold down BS, RETURN, and ' while booting (on a Dorado, push the boot button three times in quick succession). This places you in the Network Executive. The type-in conventions are simple: ? lists the possible commands, BS backspaces, DEL cancels the current line. To start up a program from the NetExec, simply type the name of that program followed by RETURN or ESC. In fact, you need only type enough of the name to distinguish it from all the others; we shall underline only that portion your need to type before the RETURN. You are currently on your way to Othello; type

```
>MesaNetExec
```

placing you in the Mesa Network Executive which has similar typing conventions. From there type

```
>OthelloDorado.pb_
```

or

```
>OthelloD0.pb_
```
(For historical reasons Dolphins are sometimes called D0's.)

depending upon whether you are using a Dorado or Dolphin. If you are in the Cedar world already, you can simply boot Othello with the Boot button at the top of the screen.

When Othello starts, it will ask you to log in. You must supply your Grapevine registered name and correct password before Othello will permit you to do anything else.

Now that you are in Othello you can use the standard command files described below for initializing disks and getting releases. If you want to do non-standard things with Othello see section 4.8 of [G5].

Here are a few fine points about starting Othello:

When starting with a new disk which you plan to erase and format, you should say

```
>Switches: n_
```

to the Mesa Network Executive before starting Othello. This prevents Othello from attempting to put the existing file system on-line when it starts up. It also permits you to log in as yourself even if someone else's credentials are installed on the disk; however, all Othello will allow you to do is to erase the disk—you cannot examine the existing contents of someone else's disk by this means.

On a Dolphin, if you ever want to boot your Cedar system from Othello rather than the boot button, you must call for the Cedar microcode by typing

```
>SetVersions for germ and microcode
Germ: D0.eg
Microcode: CedarD0.eb
```

to the Mesa Network Executive before starting Othello.

1.7.2 Getting a New Release

The next section describes how to start with a brand new, unformatted disk. This section assumes you already have a Cedar world set up, are happy with the distribution of space among your volumes, but would like to upgrade your system to the latest release. (If you want to change the volume structure read pp. 40-41 of [G5].) If you have just set up your disk, there is no need to perform these operations.

First, you should perform some disk clean-up. There are two possible levels of clean-up: deleting all the old BCDs and symbols files or erasing the volumes. You should at least do the former if there is any chance that the new release introduces new versions of things. No end of confusion will result if old versions of BCD files get mixed in with the new things. Erasing a volume takes longer and requires that you evacuate and recover personal files, but it promotes compact files, clean directory structures, and other healthful things. The utility DFDisk [G3] is useful in figuring out what you need to save. To erase your Cedar Client volume, get into Othello and type

```
>Erase
Logical Volume Name: Client
Are you sure? [y or n]: y
```

Whether or not you erase, you can get latest release's boot files by typing

```
>@
Command file: [Indigo]<Cedar>top>DoradoRelease.cm
```

or

```
Command file: [Indigo]<Cedar>top>D0Release.cm
```

New microcode, germ, and boot files will be fetched. On a Dolphin you will be put back into the Alto world; boot while holding down P. On a Dorado you will be put directly into the Cedar world.

Getting a new release takes under two minutes, so only the most impatient people will want shortcuts for updating a single item like the microcode. They should read the command file to see how to do it.

1.7.3 Initializing a Disk

You should only do this step on your personal disk or machine; don't do this to a public machine's disk! This initial setup should work no matter whether your disk is blank, smashed, or already contains a working version of Cedar. Besides taking time, this initial setting up discards the record of bad pages each disk has, so you do not want to reformat your disk gratuitously. If you have a functioning Cedar world and just want to get a new release or clean up your disk, go back to subsection 1.7.2.

To initialize a Dorado that has a new, unformatted disk, type to Othello

```
>@
Command file: [Indigo]<Cedar>top>FormatNewPrivateDorado4.cm
```

It is assumed that you have an Alto world on partition 5 of your disk. This command file will take the other four partitions of the Dorado's disk (1, 2, 3, and 4) and create four Pilot logical volumes: the Client volume for Cedar, the Debugger volume for CoCedar, the Othello volume for general utility, and the Booter volume for checkpoints. When the cedar tree cursor appears, go back to section 1.0.

If for some reason, you want to recreate standard logical volumes on a disk that has already been formatted with partitions 1, 2, 3, and 4 dedicated to Cedar, use the command file [Indigo]<Cedar>Top>MakeDoradoDisk4.cm. This command file is identical to FormatNewPrivateDorado4.cm except that the physical volume is not reformatted and your list of bad disk pages is kept.

For Dolphin users, some ground rules:

(1) Your Dolphin should have 768K words of real memory, indicated by 3072 appearing on the maintenance panel while you are in the Alto world. The hardware maintenance staff will add memory to your Dolphin upon request. Make sure you have the latest memory controller upgrade, too.

(2) You should dedicate 3/4 of your disk space to Cedar/Pilot. This assumes that you are using your Alto partition for just Bravo, SIL, and other nostalgia items. If you don't now have a small Alto disk partition, here is how to convert: FTP all personal files from your disk to some safe place, like a file server; CopyDisk from [Indigo]<BasicDisks>Mesa6-14.bfs to BFS1; finally, FTP your personal files back.

To initialize a Dolphin Cedar world, type to Othello:

```
>@
Command file: [Indigo]<Cedar>top>FormatNewPrivateD0.cm
```

After about fifteen minutes, you should have an initialized Dolphin with 3/4 of its disk space dedicated to Cedar. You will have three volumes on your machine: the Client volume for Cedar programs, the Othello volume for general utility, and the Booter volume for checkpoints. Note that there is no Debugger on your disk. Use BugBane for common bugs and teledebugging for cases BugBane can't handle. A person with a Dorado will be happy to help you teledebug.

After all this you will find yourself back in the Alto world. Boot the machine while holding down P, and you should be in the Cedar world, looking at the cedar tree. Go and read section 1.0.

If for some reason, you want to recreate standard logical volumes on a disk that has already been formatted with 3/4 of the disk dedicated to Cedar, use the command file [Indigo]<Cedar>Top>MakeD0Disk.cm. This command file is identical to FormatNewPrivateD0.cm except that the physical volume is not reformatted and your list of bad disk pages is kept.

If you want to devote your entire Dorado disk to Cedar (and eliminate the Alto partition entirely) then use the command file [Indigo]<Cedar>top>FormatNewPrivateDorado5.cm. This will destroy everything already on the disk, so be sure you have saved anything that you want preserved.

If you wish to configure your disk in other than the standard way (e.g., to use all of a disk for Cedar on a Dolphin), consult a wizard.

1.7.4 Other Othello commands

If you find that you use Othello a lot, you may want to set things up so that the default action upon booting the machine is to start Othello rather than to boot or roll back your Cedar Client world. This enables you to get to Othello directly rather than via the long excursion through the MesaNetExec. The procedure for setting things up this way is:

```
>Set Physical volume boot files
Logical volume name: Othello
Set physical volume boot file from this logical volume? Yes
Set physical volume pilot microcode from this logical volume? Yes
Set physical volume germ from this logical volume? Yes
Are you sure? Yes
```

```
>Set Debugger pointers
for debuggee logical volume: Othello
for debugger logical volume: Debugger
Are you sure? Yes
```

From Othello, you can roll back your Client world by saying "RollBack Client"; you can boot your Client world by saying "Boot Client" followed by two CRs; and you can go directly to the debugger either by typing CTRL-SWAT (the SWAT key is the unmarked one next to the right SHIFT key) or by saying "Boot Debugger" and specifying switches of "w".

1.8 General Failure Modes

You may have the misfortune to encounter a bug in the Cedar system that causes it to crash. There are various ways to recover from crashes.

If you seem to be stuck and the maintenance panel lights (on a Dorado they actually appear on the screen) say:

910: This is displayed while booting or world-swapping and does not indicate a failure.

912: Version mismatch between the germ and boot file. Consult an expert.

915: Cedar tried to transfer control to a world-swap debugger, but there isn't one on your local disk. See section 2.3.

920: This is displayed while booting or world-swapping and does not indicate a failure.

921: An unrecoverable disk error occurred while booting or world-swapping. Try again; but if the problem persists you may need to rebuild your file system.

922: An Etherboot of Othello or some other program timed out; try again.

923: Something about your germ or boot file is wrong, try getting a new release of Cedar (per section 1.7.3)

933: Something about your machine has changed since the Cedar checkpoint and/or Debugger image was installed on the disk. Most likely either the disk pack was moved to another machine or the machine's Ethernet address was changed. Boot the Debugger, boot the Client, and remake the checkpoint.

937: Unable to get the time from the Ethernet, most likely due to Ethernet or time server failure, but possibly due to a hardware problem in your machine. If repeated failures occur, consult a wizard.

957: This is a symptom of a hardware problem on Dolphins. Notify the hardware maintainers.

960: Wait for a while, possibly even 20 minutes. Your disk is being garbage-collected. Be patient; booting merely starts the process over again.

Anything less than 900 typically indicates a microcode or hardware problem. Maintenance panel codes less than 900 usually occur only on Dolphins. See [G6] for a list of Dolphin maintenance panel codes. On a Dorado, a hardware or microcode-detected error usually halts the machine; the usual manifestation is that the screen turns completely black or gray with diagonal stripes. Maintenance panel codes above 900 but not in the above list are usually but not always due to hardware problems also.

If the maintenance panel says 990 (or, on a Dorado, no numbers are visible), but there is no response to keyboard or mouse input, you need to get control somehow. In general, there are several levels of fall-back to try. Starting from the least drastic:

0. Go to a world-swap debugger by typing CTRL-SWAT; the SWAT key is the unmarked one next to the right SHIFT key. You can look around and then resume without losing anything. If your machine doesn't have a world-swap debugger installed, this may produce a maintenance panel code of 915, as described above.

1. Type CTRL-LEFTSHIFT-SWAT to get to a world-swap debugger "delicately". As with the previous

step, this can produce a maintenance panel code of 915.

2. Click a Stop button or type CTRL-DEL in a viewer. This will sometimes stop a run-away program and get the system behind the viewer to listen.

3. Perform a rollback (or boot if no checkpoint file exists) using either the Rollback button or the physical boot button. This loses whatever is in virtual memory and open files.

4a. On Dorados, boot by a three button boot, holding down C. This fetches new microcode from the net.

4b. On Dolphins, press the start button on the maintenance panel and then perform step 2. The main difference between a keyboard boot and a maintenance panel boot is that the former requires the disk to be spun up and the machine already to be in a fairly good state, whereas the latter will start the machine from an arbitrary state.

If the above methods fail to get your Cedar World up, there are problems with your file system.

5. There are various scavenging procedures available under Othello, described in [G5]: Check Drive, Scavenge, Physical Volume Scavenge, and DEScavenger. Get an expert to help you with these. These might recover the situation without much information loss.

6. Erase your client volume and get a new release, losing all non-backed up files. See section 1.7.3.

7. Initialize your disk and get a new release. See section 1.7.2.

2. Programming in Cedar

2.1 Running programs

You might try running the following example program:

```
-- Test.mesa, last modified by Jim Morris July 8, 1982 12:31 pm

DIRECTORY
   IO USING [GetInt, PutF, CreateViewerStreams, int, STREAM],
   UserExec USING [CommandProc, RegisterCommand]
   ;

Test: CEDAR PROGRAM IMPORTS IO, UserExec = BEGIN
in, out: IO.STREAM;

Compute: UserExec.CommandProc = BEGIN
   i, j: INT;
   out.PutF["Type me a couple of numbers: "];
   i ← in.GetInt[];
   j ← in.GetInt[];
   out.PutF["The sum of %g and %g is %g.\n", IO.int[i], IO.int[j], IO.int[i+j]];
   END;
[in, out] ← IO.CreateViewerStreams["Compute.Log"];
UserExec.RegisterCommand["Compute", Compute];
END.
```

To create this file click **New**, copy this text into the new file, and store the file as test.mesa (using the **Files** sub-menu). You then compile and run it with the following interaction:

```
&1 compile test
Loading Compiler.bcd. . .
Compiling: test . . . . . . no errors
End of compilation
&2 Run test
&3 Compute
```

Now click anywhere inside the new viewer named Compute.Log, and type in two numbers, followed by a RETURN.

2.2 System Models   Eric Schmidt, Ed Satterthwaite

As soon as you start dealing with a system of programs you should consider using a system model to describe and control them [G7]. There are two benefits: the modeller will figure out what you need to re-compile automatically, and it will replace modules in a running system so that you needn't always restart your program after fixing a bug. The following is a trivial system model whose only component is the test program from above.

```
-- Trivial.Model, 24-Jun-82 17:53:12 PDT

OPEN @BasicCedar.model;

Trivial: PROC [IOImpl: IO,
   UserExecImpl: UserExec,
   RopeImpl: Rope] RETURNS [] [
   Bringover: TYPE == @Bringover.bcd;   -- detour to avoid parser bug
   Main: CONTROL == @Test.Mesa
```

```
    ]
```

The parameters of the procedure Trivial are implementations for the three interfaces IO, UserExec, and Rope. Those interfaces are types declared in the file BasicCedar.model. When you run this model the implementations will be supplied from already loaded programs.

To start the modeller type

run model

This will bring up the Modeller viewer. Type **trivial** in for ModelName and click **StartModel** to tell the modeller to read in and analyze the model. A series of messages to appear in the modeller's lower window, concluding with a line of dashes. Click **Begin** to start the program. This might cause Test.mesa to be compiled; you must give it permission to compile (remember to click inside the viewer first) by typing Y. The program will be loaded, the Compute.Log viewer will appear, and the command Compute can be invoked from the User Exec, as before.

To change the program to behave differently edit Test.mesa to change the "+" to a "-" and "sum" to "difference". Save the file and click **Continue** in the Modeller menu. This will cause Test.mesa to be recompiled and reloaded. Invoke "Compute" from the User Exec again and try the new program. The modeller may be shut down in an orderly way by clicking **StopModel**.

Currently, BugBane (see 2.3) may get confused about which version of the module you are talking about. You should use the **ResetCache** button after each module replacement.

2.3 BugBane **Russ Atkinson**

BugBane provides the Cedar debugging facilities, which include a basic interpreter, primitives for controlling programs by setting breakpoints, proceeding from breakpoints, and other such services. These facilities are available to the user through the UserExec as described in section 1.2.

Interpreter

The interpreter has access to all names defined in the global frames of loaded programs (including types) plus all names in a special name space local to the interpreter. These special names all begin with &.

The interpreter handles a subset of Cedar expressions. The following summary of the subset language is from the BugBane documentation [G8]:

constants	fixed, REAL, Rope.ROPE, CHAR, BOOL, enumerated
simple variables	evaluated according to search rules below
x.y	x is a RECORD, REF or POINTER TO RECORD, global frame
x[y]	x is a SEQUENCE or ARRAY, REF or POINTER TO SEQUENCE or ARRAY
P[args]	P is a PROCEDURE taking given arguments
RT[args]	a RECORD constructor where RT is a RECORD type
LIST[exprs]	evaluates a list of expressions, producing a LIST OF REF ANY
X ← Y	X and Y are expressions

The interpreter handles expressions containing arithmetic and logical operators (such as + and OR), and conditional expressions (IF but not SELECT.) Sometimes it is necessary to write parentheses around an expression to prevent the interpreter from getting confused. In general, you must prefix a procedure name with the name of its interface or implementation module; e.g., **Rope.Cat**. However, if you evaluate an unadorned interface or implementation module name, e.g., **List**, unprefixed names on later lines are interpreted relative to that module.

In looking up a name, the interpreter

Checks to see if the name begins with &, and if so it binds to the named &-variable.

Otherwise, it searches the global default context, if any. The global default context can be set by interpreting an unadorned module as described above, or via the UserExec command SetContext.

Otherwise, it searches the current local context, if any. The local context is the sequence of local frames and associated global frames in the call stack of the stopped process. See discussion of Action Areas under section 1.2.

If this fails, it searches the space of all interface names exported by loaded program modules. If multiple instances of the same program module have been loaded, only the most recent one will be seen. Also, the association of interfaces and programs works most of the time, but may fail.

If this fails, it tries to match the name with the name of a loaded program module (subject to the three restrictions just mentioned). This search will not find individual components such as variables contained in these global frames; such components must be qualified by the module name.

Since the name lookup process can take a long time, it runs for only a certain time, and then says that the name is undefined. If you know that a name in question would be found if it searched further, you should follow the name with !. This will cause the lookup process to try with all its might and all your time. However, you can always tell the lookup process to stop by typing CTRL-DEL or by clicking the Stop button in the Work Area menu.

CoPilot

CoPilot is the backstop debugger for Cedar. See [G5] for a complete description. In the very near future, CoPilot will go away and be replaced by remote debugging in which the debugger is a full Cedar system. If you find yourself in a situation which seems to require knowledge of CoPilot, you probably should find a wizard.

3. References

In general, the following directories are worth browsing:

[Indigo]<Cedar>Documentation>* is the general repository for documentation.

[Indigo]<Cedar>Top>*.df will list pointers to things in the release

[Indigo]<APilot>* for Pilot stuff. There is a list mapping short file names to path names on Documentation>APilotFiles.txt

[Indigo]<Cedar>Documentation>Cedar3.5Xref.press, .txt is an inverted listing giving the DF file for every file in the release; it it helpful for finding things.

There are subdirectories of these directories depending on the package or subsystem involved. Use * liberally when in doubt.

3.1 General References

In the following assume the file is on [Indigo]<Cedar>Documentation> unless a full path name is explicitly given

[G1] Paxton, W., *The Tioga Editor.* In the Cedar Manual and TiogaDoc.press, .tioga.
The manual for the text editor and manuscript preparation system.

[G2] Brotz, D., *Laurel Manual*, CSL-81-6.

[G3] Schmidt, Eric, *The DF Files Reference Manual*, DFFilesRefMan.press.

[G4] Levin, Roy, Cedar Releases: Policies and Procedures, ReleaseProcedures.press/bravo

[G5] SDD, *Pilot User's Handbook.* On [Iris]<Pilot>Doc> PilotUsersHandbook.press.
You don't need all of it and it's a long document to print; borrow or obtain a hardcopy and look at pages 47-92 for Cascade documentation if you deal with CoPilot extensively.

[G6] Fiala, Ed, [Indigo]<D0Docs>MPCodes.*.

[G7] Schmidt, E. and Lampson, B., *Cedar System Modelling Reference Manual*, ModelRefMan.Press.

[G8] Atkinson, BugBane, [Indigo]<Cedar>BugBane>BugBane.doc, BBV.doc, BugBane.shorts, and BugBane.wish.

[G9] Horning, J. (ed.), *The Cedar Catalog*, In the Cedar Manual and Catalog.tioga/press.
A description of programs available for use and study.

[G10] Levin, *The Release Messages*, CedarRelease*.msg.
These messages describe the properties of each new release. They often contain vital pieces of information about known bugs and how to avoid them. Obviously, the information in older messages may be out of date.

[G11] Ornstein, *Dorado User Rules*, posted by sign-up sheets opposite CSL coffee room.

[G12] Ramshaw, *The Alto/Dolphin/Dorado Briefing Blurb*, [MAXC]<AltoDocs>BriefingBlurb.press. Describes (almost) everything there was to know about the basic CSL/ISL computer environment in 1981.

[G13] Lampson, B., Taft, E., *Alto Users Handbook*, November, 1978. The basic reference for using the Alto system.

[G14] Teitelman, W., Cedar UserExec, UserExec.tioga, .tioga.press

3.2 Cedar Language References

[L1] Mitchell, Maybury, and Sweet, *Mesa 5.0 Manual*, CSL-79-3, April, 1979. The Mesa documentation is fragmented across this and the next two references, so you need to be familiar with all three.

[L2] SDD, *Mesa 6.0 Compiler Update*, [Ivy]<Mesa>Doc>Compiler60.press.

[L3] Satterthwaite *et al.*, *Cedar Mesa 6T5*, [Indigo]<CedarDocs>Lang>Cedar6T5.press. This is a detailed description of the Cedar language, assuming one knows Mesa. Some changes since this document are enumerated in a shorter document, Documentation>Cedar7T11.press.

[L4] Horning, J., *Cedar Language Overview*, Overview.tioga, .press; Lampson, B., *Cedar Lanuguage Reference Manual, Grammar, and Summary*, CLRM.press, CLRMGram.press, CLRMSafeGram.press, CLRMSumm.press; Mitchell, J., *Annotated Cedar Examples*, CedarExamples.tioga, .press.

[L5] Mitchell, J. *Stylizing Cedar*, cedarstyle.doc, .press, *Cedar Style Sheet*, stylesheet.sil, .press.

Raison d'Etre

The purpose of this document is to help immigrants adapt to the local computing community. By "the local community", I mean primarily the Computer Science Lab, the Imaging Sciences Lab, and the Integrated Design Lab of the Xerox Palo Alto Research Center, better known by the acronyms CSL, ISL, and IDL respectively. Immigrants to other computing communities within Xerox may also find this document of interest, but I make no guarantees. I shall assume herein that said immigrants know quite a bit about computer science in general. Hence, I shall concentrate upon discussing the idiosyncratic characteristics of the local hardware environment, software environment, social environment, linguistic environment, and the like.

You will doubtless read many documents while you are at Xerox. A common convention observed in many manuals and memos is that fine points or items of complex technical content peripheral to the main discussion appear in small type, like this paragraph. You will soon discover that you cannot resist reading this fine print and that, despite its diminutive stature, it draws your eyes like a magnet. This document has such passages as well, just so that you can begin to enjoy ferreting out the diamonds hidden in the mountain of coal.

There is a great deal of useful information available on-line at Xerox in the form of documents and source programs. Reading them is often very helpful, but finding them can be a nuisance. Throughout this document, references to on-line material are indicated by <*n*>, where *n* is a citation number in the bibliography at the end of this document. Standard citations to the open literature appear as [*n*].

If you are fortunate enough to be reading this document from within Tioga (the Cedar editor), you should pause at this point to try out the "Def" command. If you were to select the three characters "<n>" in the preceding paragraph and then click the "Def" command with the middle mouse button, you would then find yourself looking at the place in this document where "<n>" is defined, that is, where it appears followed by a colon. You could then get back to this section of the document by clicking the "PrevPlace" command with any mouse button. The "Def" command is almost as good as an automatic indexing facility. On another topic, you might try clicking the "FirstLevelOnly" button (click "Levels" first if you can't find the "FirstLevelOnly" button), and then clicking "MoreLevels" a few times. Try scrolling a bit too. The Tioga "levels" commands are almost as good as an automatic table of contents.

Reading a document from front to back can be mighty boring. Fortunately, this document is so disorganized that it is not at all clear that it really has a front and a back in any normal sense. You might as well just browse through and read the parts that look interesting. To help out the browsers in my reading community, I have more or less abandoned the custom of being careful to define my terms before I use them. Instead, all the relevant terms, acronyms, and the like have been collected in a separate Glossary. Some information is contained *only* in the Glossary, so you may want to skim through it later (or now, for that matter). The "Def" command in Tioga is particularly helpful when browsing the Glossary from within Cedar; try selecting the word "Tioga", and then clicking the "Def" button in the Glossary viewer, for example. While writing the Glossary, I assumed that you have a basic knowledge of computer science, and a modicum of common sense: don't expect to find terms like "computer" and "network" in the Glossary.

The Briefing Blurb:

Exploring the Ethernet with Mouse and Keyboard

1983 Edition

By Lyle Ramshaw of PARC/CSL

An immigration document in the tradition of Roy Levin's *A Field Guide to Alto-Land.*

June 7, 1983

Filed on: [Indigo]<Cedar>Documentation>BriefingBlurb.tioga, BriefingBlurb.press

Abstract: This document is a general introduction to the computing environment at PARC slanted towards the needs and interests of newcomers to the Computer Science Laboratory. If you are looking at this document on-line from within the editor named Tioga, you might want to use the level-clipping functions to see the overall structure rather than simply plowing straight through. Click the "Levels" button in the top menu, then click "FirstLevelOnly" in the new menu that appears. That will show you the major section headings. Click "MoreLevels" to see the subsections, or click "AllLevels" to read the details.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

For Internal Use Only

Naming Things

At the outset, you should know something about the names of the creatures that you will find here. The prevailing local philosophy about naming systems is perhaps somewhat different from the trend elsewhere. We do have our share of alphabet soup, that is, systems and languages that are named by acronyms of varying degrees of cuteness and artificiality; consider, for example: PARC, FTP, MAXC, IFS. But we are trying to avoid making this situation any worse. To this worthy end, names for hardware and software systems are frequently taken from the *Sunset Western Garden Book* [1]; Grapevine servers are named after wines; Dorados are named after capital ships; Pilot releases are named after California rivers. As this convention about names does not meet with universal approval, it seems inappropriate to offer a justification of the underlying philosophy without offering equal time to the opposition. You will doubtless provoke a far more interesting discussion if you advance your own views on naming to almost anyone wandering in the corridors.

While we are on the general topic of the names of things, we should discuss for a moment the local customs for constructing single identifiers out of multiple word phrases. Suppose that you would like to name a variable in your program "name several words long". In some environments, a special character that isn't a letter but that acts something like a letter is used as a word separator within identifiers; this leads to names such as

"name!several!words!long" or "name_several_words_long".

No such character is in common use locally, however. Instead, shifting between upper and lower case is used to show the word boundaries, leading to the name

"NameSeveralWordsLong".

Some people, including Don Knuth, think that identifiers with mixed case look terribly ugly. I refuse to get sucked into expressing my opinion in this document; once again, I exhort you to espouse your views in the corridors.

There are several fine points that I should mention as well. As a general rule, case is significant for identifiers in the local programming languages, but case is not significant in file names or in Grapevine R-names. Thus, the Cedar identifiers "REF", "Ref", and "ref" are quite distinct, but the file names "BriefingBlurb.tioga" and "briefingblurb.tioga" are equivalent, as are the R-names "Ramshaw.PA" and "ramshaw.pa". In Mesa and Cedar, there is a further convention that the case of the first letter of an identifier is used to distinguish fancy objects, such as procedures and types, from simple ones, such as integers and reals. Thus, the identifier name "ProcWithFiveWordName" begins with an upper case "P", but the name "integerWithFiveWordName" begins with a lower case "i". The latter form looks very strange to most people when they first see it. When you first tasted an olive, you probably didn't like it. Now, you probably do. Give these capitalization conventions the same chance that you would an olive.

These capitalization conventions don't work too well when acronyms and normal words appear together in one identifier. Suppose, for example, that I wanted to introduce an identifier named "FTP version number". Logic would demand "FTPVersionNumber", but this doesn't look quite right: many people would be probably write "FTPversionNumber" instead. Of course, since a version number is probably an integer, it should really be "fTPVersionNumber" Ugh. Perhaps case is being used for too many purposes?

Local Hardware

Most of the offices and some of the alcoves around PARC have personal computers in them of one flavor or another. The first of these was the Alto. There are more than a thousand Altos in existence now, spread throughout Xerox, the four universities in the University Grant program (U. of Rochester, CMU, MIT, and Stanford), and other places. In recent years, most of the local Altos have been replaced by various flavors of D-machines: Dorados, Dolphins, and Dandelions. Both D-machines and Altos come equipped with bitmap displays, mice, and Ethernet interfaces. Let's discuss these components first, and then turn our attention to the various personal computers that contain them.

Bitmap Displays

First, let's talk about displays. Different displays use different representations of images. A character display represents its image as a sequence of character codes. This is a very compact representation, but not a very flexible one; text is all you can get, and probably in only a limited selection of fonts. A vector display represents its image as a list of vector coordinates. This works very well for certain varieties of line drawings, but not so well for filled areas or text. A bitmap display, on the other hand, produces an image by taking a large matrix of zeros and ones, and putting white where the zeros are and black where the ones are (or vice versa). The great advantage of bitmap displays are their flexibility: you can specify a tremendous number of images by giving even a relatively small array of bits. Cursors and icons are two large classes of prominent examples. Of course, you do have to supply enough memory to hold all those bits. Altos and D-machines store their bitmaps in main storage. An alternative would be to provide a special chunk of memory on the side where the display's image sits: such a memory is often called a *frame buffer*.

The primary display of the Alto is a bitmap that is 608 pixels wide by 808 pixels high. Such a display is almost large enough to do a reasonable job of rendering a single 8.5" by 11" page of text. The CRT on a D-machine has the long axis horizontal instead of vertical, giving a bitmap display that is 1024 pixels wide by 808 high. It had to be 808 high so that D-machines could emulate Altos, of course. The extra space allows you to have something else on the screen as well as the somewhat scrunched page of text that you are editing.

Ere I leave you with a mistaken impression, let me note in passing that bitmap displays are not the final solution to all of the world's problems. Raster displays that can produce various levels of gray as well as black and white can depict images free of the "jaggies" and other artifacts that are inherent in bitmap displays [2]. And, for some purposes, color is well worth its substantial expense.

Mice

But now on to mice. A mouse has two obvious properties—it rolls and it clicks. Inside the machine, the mouse position and the display cursor position are completely unrelated; but most software arranges for the cursor to "track" the mouse's movements. The three mouse buttons go by various names; "left", "middle", and "right" is one set of names. The mouse buttons are also called "red", "yellow", and "blue" respectively, even though physically they are nearly always black. These colorful names were proposed at an earlier time when some of the mice had their buttons running horizontally instead of vertically. Using colors (even imaginary ones!) worked better than switching back and forth between the nomenclatures "top-middle-bottom" and "left-middle-right".

Mice also come in two basic flavors: mechanical and optical. Our current mechanical mice roll on three balls: two small ones, and one large one. Motion of the large ball is sensed by two little wipers inside the mouse, one sensing side to side rolling while the other senses forward and backward rolling. The motion of each wiper drives a commutator, and little feelers slide along the commutator, producing the electrical signals that the listening computer can decode. Building one of these little gadgets is not quite as hard as building a Swiss watch, but it's in the same league. The optical mice are a more recent innovation. An optical mouse lives on a special pad, covered with little white dots on a black background. A lens in the mouse images a portion of the pad onto the surface of a custom integrated circuit. This IC has sixteen light-sensitive regions, some of which notice that they are being shined on by the image of a white dot on the pad. As the mouse slides along the pad on its Teflon-coated underbelly, the images of the white dots move across the IC; it is subtly constructed so as to observe this phenomenon, and take appropriate electrical action. For more details on this interesting application of a custom chip, you might enjoy checking out Dick Lyon's blue-and-white report on the subject [3].

The Ethernet

Two's company, three's a network. A collection of machines within reasonable proximity is hooked together by an Ethernet; if that doesn't sound familiar, I know of some blue-and-whites that you might like to browse [4,5]. Ethernets are connected to each other by Gateways and phone lines, which for most purposes allow us to ignore the topology of the resulting network. The resulting network as a whole is called an *Internet.* Occasionally, it's nice to know where things *really* are, and that's when a map <6> is helpful.

Ethernets come in two flavors: old and new. The old one runs at 3 MBits/sec, and should now be referred to as the "Experimental Ethernet". The unqualified name "Ethernet" should be reserved for the new one, the standardized version used in OSD products; it runs at 10 MBits/sec.

We all know how uncommunicative computers can be when left to their own devices. That's why we invent careful protocols for them to use in talking to each other. There are two entire worlds of protocols that are spoken on our various Ethernets as well: old and new. The old ones are called PUP-based (PARC Universal Packet) [7]. The new ones are known by the acronym NS (Network Systems) [8, 9]. I'm sure that the NS protocols must be documented, but I don't know where; sorry. Each protocol world includes a hierarchy of protocols for various purposes such as transporting files, or sending and receiving mail.

In addition to connecting up all of the personal computers, the network also includes a number of machines generically called *servers.* Normally, servers have special purpose, expensive hardware attached to them, such as large-capacity disks, or printers. Their purpose in life is to make that hardware available to the local community. We tend to identify servers by function, so we talk about print servers, file servers, name lookup servers, mailbox servers, tape servers, and so on. Many of the protocols for use of the Ethernet were developed precisely so that personal computers could communicate effectively with servers.

The Alto

The innards of the Alto are wonderfully described in a clear and informative blue-and-white report [10]; I seriously recommend that you read it. In the very unlikely event that you need to know still more about the Alto, you might try looking in the Alto hardware manual <11>. But for our purposes, suffice it to say that the Alto is a 16-bit minicomputer whose primary claim to fame is that it comes equipped with a bitmap display, a mouse, and an Ethernet interface.

D-Machines

The D-machines are a family of personal computers, each member of which has a name starting with the letter "D". As long as you don't look too closely, D-machines look a lot alike. In particular, they are all 16-bit computers with a microprogrammed processor that handles most of the I/O as well running the user's programs. And they all generally come equipped with a hard disk, a bitmap display, a keyboard, a mouse, and an Ethernet interface. There are differences of course: in size, in speed, and in flexibility.

The Dolphin (formerly called the D0)

The Dolphin was one of the early D-machines, and there are still some of them around. Dolphins are housed in the same sized chassis as Altos. You can tell that they aren't Altos because they have wide screen terminals, and because they don't have a slot on top for a removable disk pack. Instead, they use a 28MByte Winchester disk drive made by Shugart. Dolphins can talk to both 3 MBit and 10 MBit Ethernets.

The Dandelion

The Dandelion is the D-machine processor that is used in the Star products. It comes in a box about half the width of an Alto chassis, and roughly the same height and depth. Dandelions are less flexible than Dolphins, since the microprocessor is shared among the various I/O devices and the emulator in a fairly rigid round-robin fashion (associated with the terms "clicks" and "rounds"). As a consequence, it isn't very easy to hang a new I/O device off of a Dandelion. On the other hand, Dandelions are both faster and cheaper than Dolphins. Dandelions talk only to 10 MBit Ethernets.

The Dorado

Building large software systems is a demanding chore. It doesn't help any when the hardware upon which your programming environment is based doesn't have enough horsepower to support you properly—that is, in the manner to which you would like to become accustomed. After some years of trying to shoehorn large programs into Altos, CSL twisted the arms of its hardware folk and talked them into building the Dorado, the current high-performance model in the D-machine line. The processor, the instruction fetch unit, and the memory system of the Dorado have been written up in papers for your enjoyment [12]. Dorados come equipped with an 80 MByte removable-pack disk drive at present; new models may start showing up soon with a 315 MByte Winchester drive instead. Dorados talk only to 3 MBit Ethernets at present.

A Dorado is roughly three to five times faster than an Alto when emulating an Alto, that is, running BCPL. A Dorado runs compute-bound Mesa software roughly eight to ten times as fast as an Alto. Because of the raw power of a Dorado, it is usually the computer of choice for substantial programming projects. The primary difficulty about Dorados is that there aren't enough of them (and the related fact that they are rather tricky to build). Some people have their own, but others must share a pool of public machines. Now, even though the Dorado disk drives have removable packs, it really isn't very convenient to start your session of a public Dorado by mounting your own pack. The biggest difficulty is that you must be at the processor to change the disk pack, and the processor is a long way away. Subsidiary difficulties are that you must power down a Dorado in order to change the disk pack, and that T-80 disk packs are difficult to label effectively. As a result, when you borrow a Dorado, you generally also want to borrow at least some of the space on that Dorado's local disk. In order for this sharing to work out well, certain social taboos and customs concerning the use of such local disks have emerged, under the general rubric of "living cleanly". More on this topic anon.

In a return to the ways of the past, the Dorado processors are rack mounted in a remote, heavily air-conditioned machine room. It was initially intended that the Dorado, like the Alto, would live in your office. To prevent its noise output from driving you crazy, a very massive case was designed, complete with many pounds of sound-deadening material. But experience indicated that Dorados ran too hot when inside of these cabinets, and the concept of having Dorado processors in offices was abandoned. With progress in general and VLSI in particular, there is hope that some successor to the Dorado will once again come out of the machine room and into your office.

The Dicentra

The Dicentra is the newest D-machine. Essentially, it consists of the processor of the Dandelion with the tasking stuff striped out squeezed onto one Multibus card. It communicates with its memory and with I/O devices over the Multibus. Dicentras will talk to any Ethernet, or any I/O device for that matter, for which you can supply a Multibus interface card; that's one of the Dicentra's strengths. The initial application of the Dicentra is as a processor for low cost Internet gateways. The Dicentra and the Dandelion are named after wildflowers partially because they are outgrowths of an initial design of Butler Lampson's called the Wildflower.

The Daffodil

The Daffodil is a D-machine that doesn't exist yet, but someday may. If so, it will be cheap to build, since it will use custom integrated circuits. The Daffodil is product-related. Thus, please don't talk about it too widely. I mention it hear only so that you will know what it is that Chuck Thacker is talking about.

The Dragon

The Dragon is a high-performance processor based on custom integrated circuits that is being designed in CSL; confusingly enough, though, the Dragon is not really a D-machine. For example, the Dragon word size is 32 bits rather than 16. The underpinnings of Cedar will be adjusted as necessary so that Cedar will run on a Dragon; but this will take some doing.

A few comments about Booting

All of the local processors come equipped with a hidden button called the "boot button" that is used to reinitialize the processor's state. The Alto had just one boot button, hidden behind the keyboard; pushing it booted the Alto. On Dolphins, the situation is only slightly more complex: there are two boot buttons, one at the back of the keyboard, and the other on the processor chassis itself. They perform roughly the same function, but the one on the chassis is a little more potent. On Dorados, there is a lot more going on. There are really two computers involved, the main Dorado processor and a separate microcomputer called the *baseboard*. It is the baseboard computer's job to monitor the power supplies and temperature and to stage-manage the complex process of powering up and down the main processor, including the correct initialization of all of its RAM's. The boot button on a Dorado is actually a way of communicating with this baseboard computer. You encode your request to the baseboard computer by pushing the boot button repeatedly: each number of pushes means something different. For details, see Ed Taft's memo on the subject <13>. If the baseboard computer of the Dorado has gone west for some reason (as occasionally happens), your only hope is to push the *real* boot button, a little white button located on the processor chassis itself, far, far away. Just as the boot button on the keyboard is essentially a one-bit input device for the baseboard computer, the baseboard computer also has a one-bit output device: a green light located on the processor chassis. Various patterns of flashing of this light mean various things, as detailed in <13>.

There is one more bit of folklore about booting that I can't resist mentioning—every once in a while, I have to throw in some subtle tidbit to keep the wizards who read this from getting bored. Our subject this time is the "long push boot". Suppose that you have been working on your Dorado for a while, and you walk away to go to the bathroom. When you return and reach toward your keyboard, you get a static shock. You are only mildly annoyed at this until you notice that the cursor is no longer tracking the mouse, and the machine doesn't seem to hear any of your keystrokes. The screen looks OK, but the Dorado is ignoring all input. What has probably happened is that the microprocessor in your terminal has been knocked out by the static shock. Yes, Virginia! In addition to the Dorado itself, and the baseboard computer, there is also a microprocessor in your terminal (located in the display housing), which observes your input actions and sends them on to the main processor under a protocol referred to as "the seven-wire interface". What you want to do now is to reboot the terminal microprocessor without disturbing the state of the Dorado at all—after all, you were in the process of editing something, and you are now in danger of loosing those edits. What you should do is to depress the boot button and hold it down for quite a while (more than 2.5 seconds); and then release it. This is known as a "long push boot", and it does just what you want under these conditions: it reboots your terminal without affecting anything higher up.

MAXC: a blast from the past

Before we leave the topic of hardware completely, I should pause to mention the existence of MAXC (the name is said to be an acronym for Multiple Access Xerox Computer). Over the years, the folk in CSL built two MAXC's. Each was a good-sized microprogrammed computer that spent its days emulating a PDP-10: running TENEX, and timesharing away with the best of them. One of the MAXC's still survives, the one initially known as MAXC2, and it serves us now primarily as the Internet's interface to the Arpanet. Vestiges of MAXC1 still survive as souvenirs in some people's offices. Most of the stuff going between the Internet and the Arpanet is electronic mail; our mail systems understand about Arpanet recipients, so there is no need to talk to MAXC directly just to send Arpanet mail. There are a few other computing tasks that MAXC can perform and that no one has yet had the energy to supply in some other way, such as archiving files onto magnetic tape. But most folks should be able to spend their time here quite happily without ever talking directly to MAXC.

Local Programming Environments

Various programming environments have grown up around the various pieces of hardware mentioned above. You can get a software merit badge simply by writing one non-trivial program in each envirnoment.

Programming on MAXC

Since we were discussing MAXC just a moment ago, let's get it out of the way first. From a software point of view, MAXC is a PDP-10. Thus, it is programmed either in the assembler Macro-10 or else in one of a variety of higher level languages [14]. Fortunately, there aren't all that many new programs that have to be written to run on MAXC any more.

BCPL

The first high-level programming language used on the Alto was BCPL, and quite a bit of program writing was done in that environment over the years. By now, however, essentially no new programming is being done in BCPL. The language itself will be around for some time to come, since there are BCPL programs that perform valuable services for us: the print server programs Press and Spruce and the file server program IFS are three important examples.

Of the better-known computer languages, BCPL is closest to C. The fundamental data type in BCPL is a sixteen-bit word. There are facilities in the language for building structured data objects including records and pointers. But there is no type-checking in the language at all. For example, if *foo* is a pointer to a record of type *node* that includes a field named *next*, that field is referenced in BCPL by writing

"foo>>node.next",

which means "treat *foo* as a pointer to a *node*, and extract the *next* field". In a strongly typed language, you wouldn't have to mention that *foo* was a pointer to a *node*, since the compiler would be keeping track of the fact that *foo* was so declared. The BCPL compiler, however, thinks of *foo* as a sixteen bit value, just like any other sixteen bit value. For example, it would be legal in BCPL to write

"(foo+7)>>node.next", or "foo>>otherNode.next".

Some of the strictness of the Mesa approach to type-checking and version matching discussed below may be a reaction to BCPL's free-wheeling ways of handling these issues. Further details about the BCPL language and environment can be found elsewhere <14, 15, 16>.

The debugger in the BCPL environment was named "Swat". This name is preserved in the local dialect as the name of the bottom of the three unmarked keys at the right edge of the keyboard. Various debuggers may be invoked in various environments by depressing this key, perhaps in conjunction with the left-hand shift key or the control key. (The right hand shift key won't do; it is too close to the swat key itself for comfort!)

Mesa

Mesa is a strongly typed. PASCAL-like implementation language designed and built locally. It first ran on Altos. Herein, I shall call that system Alto/Mesa. Dolphins and Dorados (but not Dandelions) can run Alto/Mesa by impersonating an Alto at some level. More recent instances of Mesa now run on all of our D-machines under the Pilot operating system. In passing, I should observe that Pilot is an operating system written in Mesa by folk in SDD. It is a heavier-weight operating system than the Alto OS, providing its clients with multiprocessing, virtual memory, and mapped files.

Alto/Mesa programs do not use the Alto OS at all, mostly because Mesa and BCPL have rather different philosophies about the run-time world in which they exist. So the first thing that a Mesa program does when running on an Alto is to junta away almost all of the OS, and set about building a separate Mesa world. It is a considerable nuisance for Mesa and BCPL programs to communicate, since their underlying instruction sets are completely different. So, most of the important OS facilities, such as the file system, had to be re-implemented directly in Mesa. Mesa's memory management strategies replace the revolutionary tactics of "junta" and "counter-junta" with the relative anarchy of segment swapping.

A fair amount of software was written in Alto/Mesa, but little new programming is being done in that environment; that is, Alto/Mesa isn't quite at dead as BCPL, but it is getting there. Perhaps the crown jewels of Alto/Mesa are the systems Laurel, Grapevine, Mockingbird, and Griffin. You will be hearing more about the former two in the section on electronic mail; to find out more about the latter two, check out their entries in the Glossary.

The Pilot version of Mesa is the home to lots of active programming in several locations. First, it is the system in which the Star product was and is being implemented by OSD. The programmers in OSD have developed a set of tools for programming in Mesa variously called the "Tools Environment" or "Tajo". This body of software may soon be marketed under the name "the Mesa Development Envirnoment". In addition, Pilot Mesa is the current base of the Cedar project in CSL and ISL. More on Cedar later.

Although Mesa programs look a lot like PASCAL programs when viewed in the small, Mesa provides and enforces a modularization concept that allows large programs to be built up out of smaller pieces. These smaller pieces are compiled separately, and yet the strong type checking of Mesa is enforced even between different modules. The basic idea is to structure a system by determining certain abstract collections of facilities that some portions of the system will supply to other portions. Such an abstraction is called an "interface", and it is codified for the compiler's benefit in a Mesa source file called an "interface module". An interface module defines certain types, and specifies a collection of procedures that act on values of those types. Only the procedure headers go into the interface module, not the procedure bodies (except for INLINE's, sad to say). This makes sense, since all the interface module has to do is to give the compiler enough information so that it can type-check programs that use the abstraction.

Having specified the interface, some lucky hacker then has the job of implementing it—that is, of writing the procedure bodies that actually do the work. These procedure bodies go into a different type of module called an "implementation module". An implementation module is said to "export" the interface that it is implementing; it may also "import" other interfaces that it needs to do its job, interfaces that some other program will implement.

In simple systems, each interface is exported by exactly one module. In such a system, there isn't much question about who should be supplying which services to whom. In fact, in these simple cases, the *binding*, that is, the resolution of imports and exports, can be done on the fly by the loader. But in more complex cases, there might be several different modules in the system that can supply the same service under somewhat different conditions, or with somewhat different performance. Then, the job of describing exactly which modules are to supply which services to which other modules can become rather subtle. A whole language was devised to describe these subtle cases, called C/Mesa. The Binder is the program that reads a C/Mesa description, called a *config*, and builds a runnable system by filling imports request from exports according to the recipe.

The Mesa language is described by a manual [18]. It lies somewhere between a tutorial and a reference manual. Some people find some portions of it rather obscure; in particular, the discussion of interfaces and implementations in Chapter 7 is often cited as confusing. To make matters a little worse, that manual documents Mesa version 5.0; the current Alto/Mesa is version 6.0, and Pilot mesa has advanced even further. From the point of view of the Mesa language itself, the most important changes that have occurred since version 5.0 are the introduction of sequences and zones in version 6.0; they are documented for your reading pleasure <19>. You may also be interested in Jim Morris's comments on how programs should be structured in Mesa [20].

Smalltalk

Smalltalk was developed by the folk who now call themselves the Software Concepts Group (formerly known as the Learning Research Group). The Smalltalk language is the purest local embodiment of "object-oriented" programming:

A computing world is composed of "objects".

The only way to manipulate an object is to be polite, and ask it to manipulate itself. One asks by sending the object a message. All computing gets done by objects sending messages to other objects.

Every object is an "instance" of some "class".

The class definition specifies the behavior of all of its instances—that is, it specifies their behavior in response to the recipt of various messages.

Genealogists will recognize that ideas from both Simula and Lisp made their way into Smalltalk, together with traces of many other languages.

For some years now, the folk in SCG have been working at trying to get the Smalltalk language and system out into the great wide world. The first public event that came out of this effort was the August 1981 issue of Byte magazine; it was devoted to Smalltalk-80, including a colorful cover drawing of the now famous Smalltalk balloon. In addition, the SCG folk are writing several books about Smalltalk, and they are planning to license the system itself to various outside vendors. The first of the books, entitled *Smalltalk-80: The Language and Its Implementation*, emerged from the presses at Addison-Wesley just recently [21]. Future books will include *Smalltalk-80: The Interactive Programming Environment*, and *Smalltalk-80: Bits of History, Words of Advice.*

Interlisp-D

LISP is the standard language of the Artificial Intelligence community. Pure LISP is basically a computational incarnation of the lambda calculus; but the LISP dialects in common use are richer and bigger languages than pure LISP. Interlisp is one dialect of LISP, an outgrowth of an earlier language called BBN-LISP; for more historical details, read the first few pages of the Interlisp Reference Manual [22]. One of the biggest strengths of Interlisp is the large body of software that has developed to assist people programming in Interlisp. Consider the many features of Interlisp: an interpreter, a compatible compiler, sophisticated debugging facilities, a structure-based editor, a DWIM (Do What I Mean) error correction facility, a programmer's assistant, the CLISP package for Algol-like syntax, the Masterscope static program analysis database, and the Transor LISP-to-LISP translator, to name a few.

Interlisp itself has been implemented several times. Interlisp-10 is the widely-used version that runs on PDP-10's. Interlisp-D is an implementation of Interlisp on the D-machines [23], produced by folk at PARC. In the process of building Interlisp-D, the boundary between Interlisp and the underlying virtual machine was moved downward somewhat, to minimize the dependencies of Interlisp on its software environment; that is, functions that were considered primitive in Interlisp-10 were implemented in Lisp itself in Interlisp-D. But the principal innovations of Interlisp-D are the extensions that give the Interlisp user access to the personal machine computing environment: network facilities and high-level graphics facilities (including a window package) among them.

By the way, Interlisp has the honor of being the first system (to my knowledge) to use the prefix "Inter-". This prefix has become quite the rage of late: Internet, Interpress, Interscript—you get the general idea.

Cedar

Back in 1978, folk in CSL began to consider the question of what programming environment we would use on the emerging D-machines. A working group was formed to consider the programming environments that then existed (Lisp, Mesa, and Smalltalk) and to form a catalog of programming environment capabilities, ranked by both by value and by cost. A somewhat cleaned-up version of the report of that working group is available as a blue-and-white for your perusal [24]. After pondering the alternatives for a while, CSL chose to build a new programming environment, based on the Mesa language, that would be the basis for most of our programming during the next few years. That new environment is named "Cedar".

Cedar documentation is in a constant state of flux; indeed, it might be said that Cedar as a whole, not only its documentation, is in a constant state of flux. Much of the documentation for the current release is accessible through a ".df" file named Manual.df <25>. Hardcopies of this packet of stuff, entitled "The Cedar Manual", are produced from time to time, and distributed to Cedar programmers.

The programming language underlying Cedar is essentially Mesa with garbage collection added. Now, adding garbage collection actually changes things quite a bit. First of all, it changes programming style in large systems tremendously. Without garbage collection, you have to enforce some set of conventions about who owns the storage. When I call you and pass you a string argument, we must agree whether I am just letting you look at my string, or I am actually turning over ownership of the string to you. If we don't see eye to eye on this point, either we will end up both owning the string (and you will aggravate me by changing *my* string!) or else neither of us will own it (and its storage will never be reclaimed—a storage leak). Once garbage collection is available, most of these problems go away: God, in the person of the garbage collector, owns all of the storage; it gets reclaimed when it is no longer needed, and not before. But there is a price to be paid for this convenience. The garbage collector takes time to do its work. In addition, all programmers must follow certain rules about using pointers so as not to confuse the garbage collector about what is garbage and what is not.

Thus, programs in the programming language underlying Cedar look a lot like Mesa programs, but they aren't really Mesa programs at all, on a deeper level. To avoid confusion, we decided to use the name "Cedar" to describe the Cedar programming language, as well as the environment built on top of it. Cedar is really two programming langauges: a restricted subset called the *safe language*, and the unrestricted full language. Programmers who stick to the safe language can rest secure in the confidence that nothing that they can write could possibly confuse the garbage collector. Their bugs will not risk bringing down the entire environment around them in a rubble of bits. Those who choose to veer outside of the safe language had better know what they are doing.

Those who want to know more about Cedar are once again encouraged to dredge up a copy of the Cedar Manual <25>. It includes documentation on how Cedar differs from Mesa, annotated examples of Cedar programs, manuals for many of Cedar's component parts, a Cedar catalog, and lots of other good stuff. By the way, the most authoritative source for what the current Cedar compiler will do on funny inputs can be found in a document called the Cedar Language Reference Manual, also known by the acronym CLRM. This is logically part of the Cedar Manual, but it is currently bound separately, and only available in draft form. The CLRM suggests a particular design philosophy for building a polymorphic language that is a superset of the current Cedar, since that is the direction in which the authors of the CLRM, Butler Lampson and Ed Satterthwaite, would like to nudge the Cedar language.

Local Software

This section is a once-over-lightly introduction to some of the major software systems that are available in the Alto and Cedar worlds. First, let me mumble some general words about how such subsystems are documented. The most commonly used Alto subsystems are documented in a tome called the Alto User's Handbook [26]. The less commonly used ones are documented in a catalog entitled "Alto Subsystems" <27>. In addition, Suzan Jerome wrote a Bravo primer aimed at non-programmers [28]. In Cedar, the current best sources are the Cedar Manual mentioned above <25>, and a brand new public database, sitting on Alpine, containing whiteboards of Cedar documentation. Unfortunately, I won't hear about the latter until Dealer tomorrow, so that I can't tell you any more about it at the moment; I'm sorry, but that's life in a rapidly changing world. Wow! I've seen the whiteboards stuff now, and it's flashy! Maybe this is the last version of the Briefing Blurb that I'll ever have to write.

Filing

When programming in the Alto world, or in current Cedar, you are dealing with two different types of file systems: local and remote. The local file system sits on your machine's hard disk. Remote file systems are located on file servers, machines with big disks that are willing to store files for you. Local file systems have several unpleasant characteristics in comparison with the remote systems: they are small, and they aren't very reliable. Both of these problems have consequences.

Because local file systems are small, it isn't in general practical to store more than one version of a file on the local disk. Thus, in our current local file systems, writing a "new version" of a file really means writing on top of the old one. Nearly everyone who isn't accustomed to this (particularly PDP-10 hackers) gets burned by it at least once. There is one important exception to this general rule of "no old versions", however: our text editors maintain one backup copy of each file being edited as a separate file, whose name ends with a dollar sign. That is, the backup copy of "foo.tioga" is stored in the file "foo.tioga$", and similarly for Bravo. Note that our remote file servers do maintain multiple versions of files. Letting old versions of things accumulate is one easy way to overflow your disk usage allocation on a remote server.

No disk is completely reliable. Our remote file servers have automatic backup facilities that protect us from catastrophic disk failures. But the local file systems have no such automatic protection. Since this protection isn't provided automatically, it behooves you to adjust your behavior appropriately: make sure that, on a regular basis, backup copies of the information on your local disk are put in some safe place, such as on a remote file server where suitable precautions are constantly being taken by wizards to protect against disk failure. Doing this is one facet of what is meant by the phrase *Living Cleanly*, which deserves its own section.

Living Cleanly (also known as "Keeping your bags packed")

The phrases "living cleanly" and "keeping your bags packed" refer to a particular style of use of your local file system. In order to understand the cosmic issues involved, we should pause to discuss the ways in which local and remote file systems have been used over the years.

Back in the Alto days, personal files were usually stored on one's Alto disk pack, while project-related and other public files were stored on remote servers. Careful folk would occasionally store backup copies of their personal files on remote servers as well, in case of a head crash. But, as a general rule, one thought of one's Alto pack as the repository of one's electronic state. This made sharing Altos quite convenient, since you could turn any physical Alto into "your Alto" just by spinning up your disk pack.

In the glorious world of the Cedar future, all of your personal files as well as all public files will live on file servers in the network. The disk attached to your personal computer will, from time to time, contain copies of some of this network information, for performance reasons; but you won't have to do anything to achieve this, and you won't have to worry about how it is done. From the user's point of view, all files will act as if they were remote at all times. Indeed, except in a few funny cases, there won't even be any notion of "local file"; "file" will mean "remote file".

At the moment, we are sitting in an unpleasant trasitional phase somewhere between these two styles of usage of the local disk: we are attempting to simulate the latter state by means of manual methods and social pressure. We want you to think of your data as really living out on the file servers. That is the proper permanent home for your personal files as well as for public files. You will have to bring copies of these files, both private and public, to your local disk in order to work on them. But, at the end of each editing session, you should store the new versions of files that you have created back out to their permanent remote homes. None of this happens automatically at present; you have to make it happen manually by using various file shuffling tools, such as the "DF files" discussed below. Using these tools is a hassle, and learning how to use them can be confusing. But, there are four important benefits to be reaped from adopting a clean living life-style.

First, you are taking a step towards the glorious future.

Secondly, you are protecting yourself against failures of the local disk. A clean liver only holds information on her local disk for the duration of an editing session. This puts a reasonable bound on the amount of information that she can lose because of a disk crash.

Thirdly, there are various reasons why erasing your local disk is a good idea when updating to a new release of the Cedar system; sometimes, in fact, it is required. Since clean living folk don't keep long term state on their local disks, this doesn't bother them in the slightest.

Finally, and perhaps most importantly, clean living is the key to sharing disk space on machines without removable disks. When you use a public Dolphin or Dorado, you are forced to share its disk space with the other members of the community. This sharing is predicated on a policy of clean living: when your session is over, you must store away all of your files on remote file servers. The person who uses the machine next may need to free up some disk space; if so, she is perfectly entitled to delete your files without qualm or pause. And you won't mind a bit, it says here, because you have been living cleanly.

The above paragraph is the "letter of the law" regarding the sharing of public disk space. People who want to be well regarded should also pay some attention to the "spirit of the law": sharing things is always more pleasant when everyone acts with a modicum of politeness and care. Don't delete the previous user's files if she was called away by some disaster and didn't have a chance to clean up. Try not to delete the standard system files, such as the Compiler, that sit in the local file system, since whoever follows you will be justifiably aggravated by their absence. Even more important, if you do exotic things such as bringing over non-standard versions of system files, try to put everything back to normal when you leave ere you cause whoever follows you to become hopelessly confused.

Local file systems

The local file system in the Alto world is called either the "Alto file system" or the "BFS", the latter being an acronym for Basic File System. The biggest that a BFS can be is 22,736 pages. This is substantially bigger than the entire disk on an Alto. However, Dolphins and Dorados have much bigger local disks. Hence, when a Dolphin or Dorado is emulating an Alto, its local disk is split up into separate worlds called *partitions*, each containing a maximum-sized BFS. Dolphin disks can hold two full partitions, while Dorado disks can hold five. What partition you are currently accessing is determined by the contents of some registers that the disk microcode uses. There is a command called "partition" in the Executive and the NetExec that allows you to change the current partition.

When operating in the Pilot world, a disk pack is called a physical volume, and it is divided into worlds called logical volumes. (Pilot, you will recall, is the new operating system written in SDD.) The area of the disk devoted to Pilot volumes must be disjoint from the area devoted to Alto-style partitions. Most Dolphins that run Cedar are set up with a half-sized Alto partition, and the other three-quarters of the disk devoted to Pilot; most Dorados that run Cedar have one full-sized Alto partition, and the other four-fifths of the disk devoted to Pilot.

In current Cedar, many programs still restrict you to working with files in the local file system, which is maintained by Pilot in the appropriate logical volume. The editor Tioga, for example, will let you read remote files specified by a full path name, but it won't let you edit them; only local files may be modified. In subsequent Cedar's, there will be a new local file system and directory package, the Nucleus and FS respectively, to go along with the new virtual memory manager (also part of the Nucleus). These wonders will make it somewhat easier to ignore the existence of the local file system, except for its beneficial effects on performance; that is, they will make clean living more nearly automatic.

All of our local file systems use a representation for files that drastically reduces the possibility of a hardware or software error destroying the disk's contents. The basic idea is that you must tell the disk not only the address of the sector you want to read or write, but also what you think that sector holds. This is implemented by dividing every sector into 3 parts: a header, a label, and a data field. Each field may be independently read, written, or compared with memory during a single pass over the sector. The Alto file system stuffs a unique identification of the disk block, consisting of a file serial number and the page number within the file, into the label field. Now, when the software goes to write a sector, it typically asks the hardware to compare the label contents against data in memory, and to abort the writing of the data field if the compare fails. This makes it pretty difficult, though not impossible, to write in the wrong place. Furthermore, it distributes the structural information needed to reconstruct the file system over the whole disk, instead of localizing it in one place, the directory data structures, where a local disaster might wipe it out. Each local file system also has a utility program called a Scavenger that rebuilds the directory information by looking at all of the disk labels.

Remote file systems

The most important local file servers are IFS's, an acronym for Interim File System (one of the crown jewels of the BCPL programming environment). Like I always say, "temporary" means "until it breaks", and "permanent" means "until we change our minds". Indigo and Ivy are two prominent local IFS's; Indigo stores mostly project files, while Ivy stores mostly personal files. MAXC also serves as a file server for some specialized applications. Juniper was CSL's first attempt to build a distributed transactional file server; it was one of the first large programs written in Mesa. Alpine is a new effort to build such a beast in the context of Cedar, in support of distributed databases and other such wonderful things. Some Walnut users have been storing their mail databases on Alpine for a month or more.

There is no coherent logic to the placement of "general interest" files and directories, nor even to the division between Maxc, Indigo, and Ivy. Browse through the glossary at the end of this document to get a rough idea of what's around. If something was made available to the universities in the University Grant program, then it is probably on Maxc (or archived off of Maxc), since Maxc is the machine that the university folk can access.

IFS supplies a general sub-directory structure which the Maxc file system lacks, and as a result there are lots of place to look for a file on an IFS. For example, on Maxc you might look for

[Maxc]<AltoDocs>MyFavoritePackage.press

while on IFS you would probably look for

[Indigo]<Packages>Doc>MyFavoritePackage.press, or

[Indigo]<Packages>MyFavoritePackage>Documentation.press,

or perhaps some other permutation. This requires a bit of creativity and a little practice. However, if you get in the habit of using "*"s in file name specifications, you will find all sorts of things you might not otherwise locate. Note that a "*" in a request to an IFS will expand into all possible sequences of characters, *including* right angle brackets and periods. Thus, for example, a request for

<Packages>*press

refers to all files on all subdirectories of the Packages directory that end with the characters "press". A "*" won't match a left angle bracket, by the way. Thus, if you ask for "*.press", you are referring to all Press files on the current directory. If you ask for "<*.press", you are referring to all of the Press files on the entire IFS (expect such a search to take a long time!).

Warning: Once you have gotten used to the IFS conventions about "*"s in file names, you will find the TENEX rules quite restrictive and unnatural. On TENEX, asterisks can be used for only two purposes: either to wildcard the entire prefix of the filename or to wildcard the entire extension. If you want to refer to all of the files on a TENEX directory, you must say "*.*", not just "*"; if you want to refer to all of the files whose names start with an "H", you are simply out of luck. This lack of "forward compatibility" (the opposite of backward compatibility?) has tripped up many a searcher.

There is a movement afoot in the Cedar world to simplify our file naming conventions by replacing the various flavors of brackets with a UNIX-like slash. Thus, in some Cedar systems, such as the FileTool, the documentation file mentioned above could be referred to as

/Indigo/Packages/MyFavoritePackage/Documentation.press.

File Properties

The "size" of a file is its length measured in disk pages; the "length" of a file is its length measured in bytes. The "create date" of a file is the date and time at which the information in that particular version .of the file was "created", that is, the date when this that sequence of bytes came into being. Copying a file from one file system to another does not change the create date, since the information in the file, the sequence of bytes, is not affected. The create date is almost always what you want to know about a file. Some of our systems also maintain a "write date" or a "read date", but they are less well defined, and not as interesting.

Editing and Typesetting

In the outside world, document production systems are usually de-coupled from text editors. One normally takes the text that one wants to include in a document, wraps it in mysterious commands understood by a document processor, feeds it to that processor, and puzzles over the resulting jumble of characters on the page. In short, one programs in the document processor's language using conventional programming tools—an editor, a compiler, and sometimes even a debugger. Programmers tend to think this is neat; after all, one can do anything with a sufficiently powerful programming language. (Remember, Turing machines supply a sufficiently powerful programming language too.) However, document processors of this sort frequently define bizarre and semantically complex languages, and one soon discovers that all of the time goes into the edit/compile/debug cycle, not careful prose composition.

Bravo is the editor and typesetter in the Alto world, and it represented a modest step away from the programming paradigm for document production. A single program provided both the usual editing functions *and* a reasonable collection of formatting tools. You can't program Bravo as you would a document "compiler", but you can get very tolerable results in far less time. The secret is in the philosophy: what you see on the screen is what you get on paper. You use the editing and formatting commands to produce on the screen the page layout you want. Then, you tell Bravo to ship it to a print server and presto! You have a hardcopy version of what you saw on the screen. Sounds simple, right?

Of course, it isn't quite that easy in practice. There are dozens of subtle points having to do with fonts, margins, tabs, headings, and on and on. Bravo was a success because most of these issues are resolved more or less by fiat—someone prepared a collection of configuration parameters and a set of forms that accommodated most document production. Many of the configuration options aren't even documented, so it is hard to get enough rope to hang yourself. The net effect is that one spent more time composing and less time compiling.

In Bravo's wake, several new editors of unformatted text appeared: the Laurel editor, and the editor in the Tools Environment are prominent examples. The Laurel editor is particularly noteworthy in that it pioneered the development of a modeless (or at least less modal) user interface for an editor. The Star product editor and Tioga are more recent local editors in the full Bravo tradition: they can handle formatting and multiple fonts. Tioga is the editor within Cedar, and its user interface is very close to the widely beloved Laurel modeless interface—try going back to Bravo after using Tioga for a while, and see how horrible it feels to have to remember to type "i" and "ESC" all the time. Tioga shows formatted text on the screen. To get a hardcopy of that text, the current path involves running a companion program called the TSetter, which will compose your pages for printing and send them to a print server. Tioga's documentation is particularly convenient, since it usually available in iconic form at the bottom of the Cedar screen <29>.

Dealing with editor bugs

All text editors have bugs. Furthermore, you are often most likely to tickle one of the remaining bugs in an editor when you are working furiously on a hard problem, and hence, have been editing for a long time without saving the intermediate results. As fate would have it, these are exactly the times when it is most damaging and most upsetting to lose your work. There is nothing quite like the sinking feeling you get when a large number of your precious keystrokes gurgle away down the drain. Both Bravo and Tioga have mechanisms that can, in some cases, save you from the horrible fate of having to do all those hours of editing over again. Bravo attempts to safeguard you by keeping track of everything that you have done during the editing session in a log file; in case of disaster, this log can be replayed to recapture most of the effects of the session. If you have a disaster when editing in Bravo, be careful NOT to respond by running Bravo again to assess the damage. By running Bravo again in the normal way, you will instantly sacrifice all chance of benefiting from the log mechanism, since the log allows replay only of the most recent session. What you want to do instead is run the program "BravoBug" ("Bravo/R" is not an adequate substitute). It wouldn't be a bad idea to ask a wizard for help also. While you are looking for a wizard, try and think of some good answer to the question "Why are you using Bravo, anyway?", which said wizard will almost certainly ask.

The most common—perhaps I should really say "the least rare"—source of editing disasters in Tioga is problems with monitor locks. Unfortunately, this class of problem usually makes further progress in any part of Cedar impossible, since Tioga is so basic to the Cedar system. If you can get to the CoCedar debugger, you might be able to save your edits by calling the procedure

```
← ViewerOpsImpl.SaveAllEdits[ ]
```

Rumor has it that Cedar versions from 4.2 on will allow you to invoke this procedure by hitting a special collection of keys in Cedar itself, even after Tioga has become wedged. A further rumor has supplied more details: holding down both the left and the right shift keys and the Swat key for more than 1 second will invoke SaveAllEdits[ ]. While the saving is taking place, the cursor will become a black box.

Printing

In general, our printers are built by taking a Xerox copier and adding electronics and a scanning laser that produce a light image to be copied. There are many different types of such printers, and there are multiple instances of each printer type as well. There are also many different programs that would like to produce printed output. The Press print file format was our first answer to the problem of allowing every printing client to use every printer. Press files are the Esperanto of printing. Most print servers demand that the documents that you send to them be in Press format. This means you have to convert whatever you have in hand (often text) to Press format before a server will deign to print it.

Press file format <30> is hairy, and some print servers don't support the full generality of Press. Generally, however, such servers will simply ignore what they can't figure out, so you can safely send them any Press file you have.

A Press file can ask that text be printed in one of an extensive collection of standard fonts. Unfortunately, you must become a wizard in order to print with your own new font. You can't use a new font unless it is added to the font dictionary on your printer, and adding fonts to dictionaries is a delicate operation: a sad state of affairs. If the Press file that you send to a printer asks for a font that the printer doesn't have, it will attempt a reasonable substitution, and, in the case of Spruce, tell you about the substitution on the break page of your listing. If you have chronic font difficulties of this sort, contact a wizard.

There is a new print file format under development, called Interpress. The print servers that are part of the Star product speak a dialect of Interpress. A print file in Interpress format is called a *master*. Our local plans for printing Interpress masters involve converting them first into a printer-dependent print file in so-called PD format (with conventional extension ".pd"). From there, a relatively simple driver program on each printer should be able to produce the final output.

The rest of this section was contributed by Julian Orr of ISL:

PARC has a variety of printers available for your hardcopy needs. We have high volume printers for quantities of text, listings, and documentation; we have slower printers with generally higher quality for more complex files; and we have very slow printers for extremely high quality. All of our current printers except Platemaker offer 384 spots per inch and share a common font dictionary. We use two different software systems for printing Press files, both running on Altos: one is called Spruce, and the other is called (confusingly) Press. Spruce offers speed and spooling, but it can only image characters and rules, and not too many of them. This makes it limited in graphics applications. Furthermore, Spruce is limited to the particular sizes of fonts that it has stored in its font dictionary: it does not know how to build new sizes by converting from splines. Press is slower, but can handle arbitrary bitmaps, and can produce odd-sized fonts from splines.

ISL is developing Interpress printing capabilities. Printing ".pd" files is now an option on most Press printers (that is, on printers running the program Press as opposed to Spruce). Just ship your ".pd" file to the printer in the standard way: it is smart enough to figure out whether what you have sent it is in PD or Press format, and it will invoke PDPrint or Press as appropriate. Documentation on these two printing programs is available, by the way <31, 32>. PD printing should not be undertaken without consultation with a wizard.

Dover printers run Spruce for high volume printing, producing a page per second. CSL's Dover, named Clover, is found in room 2106; ISL's Dover, named Menlo, is in room 2305. Samples of the Dover font dictionary may be found next to Clover and Menlo. Instructions for modifying the queue and generally running these Spruce printers are to be found next to their Alto terminals.

Lilac is our color Press printer and may be found in 2106 with Clover. It is a three color, composite-black machine; it generally produces good quality output, but is occasionally temperamental. Anyone interested in color printing or the state of Lilac should join the distribution list LilacLovers↑.pa.

In the ISL maze area, room 2301, we have an assortment of black and white Press printers, answering variously to the names of RockNRoll, Quoth, and Stinger. The printers are two Ravens (Raven is a Xerox product), one Hornet, and one Gnat (the latter two are prototypes). The print quality is normally excellent. Instructions for interpreting status displays are posted locally. To be informed of which printer is functioning and where, join the list ISLPrint↑.pa. There should be three printers up for most of the summer. Periodically one or another of these or Lilac are pre-empted for debugging.

Our best quality printer is Platemaker, which is normally operated at 880 spots per inch, but can be run up to 2200 spi; it is not normally useful to go beyond 1600. Platemaker uses a laser to write on photographic paper or film. Color images can be done in individual separations, which are then merged using the Chromalin process. The Platemaker printing process is used for final prints of fine images or for printing masters for publication. If you wish to have something printed, speak to Julian Orr, Eric Larson, or Gary Starkweather.

Sending and Receiving Mail

We rely very heavily on an electronic mail system. We use it for mail and also for the type of announcement that might, in other environments, be posted on a physical or electronic bulletin board. In our environment, a physical bulletin board is pretty useless, since people spend too much of their days staring at their terminals and too little wandering the halls. Electronic bulletin boards might work satisfactorily. But a bulletin board, being a shared file to which many people have write access, is a rather tricky thing in a distributed environment. It probably presupposes a distributed transactional file server, for example. Mumble. For whatever reason, the fact remains that we don't have an electronic bulletin board facility at the moment. As a result, announcements of impending meetings, "for sale" notices, and the like are all sent as messages directed at expansive distribution lists. If you don't check your messages once a day or so, you will soon find yourself out of touch (and saddled with a mailbox full of obsolete junk mail). And conversely, if you don't make moves to get on the right distribution lists early, you may miss lots of interesting mail. This business of using the message system for rapid distribution of announcements can get out of hand. One occasionally receives notices of the form: "meeting X will start in 2 minutes—all interested parties please attend".

Grapevine is the distributed transport mechanism that delivers the local mail [33]. When talking to Grapevine, individuals are referred to by a two-part name called an "R-name", which consists of a prefix and a registry separated by a dot; for example, "Ramshaw.pa" means Ramshaw of Palo Alto. In addition to delivering the mail, Grapevine also maintains a distributed database of distribution lists. A distribution list is also referred to by an R-name, whose prefix conventionally ends in the character up-arrow, as in "CSL↑.pa". Distribution lists are actually special cases of a construct called a Grapevine "group". Groups can be used for such purposes as controlling access to IFS directories. There is a program named Maintain that allows you to query and update the state of the distribution list database. In fact, there are two versions of Maintain: the documented one with the unfortunate teletype-style user interface is used from within Laurel or the Mesa Development Environment <34>; the undocumented one with the futuristic menu interface is used from within Cedar. Some distribution lists are set up so that you may add or remove yourself using Maintain. If you try to add yourself to Foo↑.pa and Maintain won't let you, the proper recourse is to send a message to the distribution list Owners-Foo↑.pa, asking that you please be added to Foo↑.

At the moment, Grapevine pretty much has a monopoly on delivering the mail. But there are several different programs that give users access to Grapevine's facilities from different environments. From an Alto, one uses Laurel, which is mentioned elsewhere as a pioneer of modeless editor interfaces. Even if you aren't a Laurel user, I recommend that you read Chapter 6 of the Laurel Manual [35], which is an enlightening and entertaining essay on proper manners in the use of the mail system. In the Mesa Development Environment, the program Hardy provides services analogous to Laurel's. From within Cedar, most folk use Walnut, whose documentation appears as part of the Cedar Manual <25>. Walnut represents a step towards the future in some respects, since Walnut uses Cypress, the Cedar database management system, to store your mail in a database. Access to Grapevine from within Cedar can also be had without the database frills through a program called Peanut, which stores your messages in a structured Tioga document instead of in a database. Finally, in case travel should take you away from your multi-function personal workstation, there are servers on the Internet known by the name "Lily" to whom you can connect from any random teletype in order to peruse the mail sitting in your Grapevine mailbox.

Packaging Systems and Controlling Versions

In the BCPL world, the primary facility for packaging up a group of related files and either handing them out or storing them for later recall was the *dump file.* A dump file, given conventional extension ".dm", is simply the concatenation of the dumpees, together with enough header information to allow the dumpees to be pulled apart again. Dump files have fallen out of favor.

In the Alto/Mesa world, and more strongly, in the Cedar world, a collection of software called "DF files" has grown up that attacks the problem of describing and packaging systems, detailing their interdependencies, and controlling the versions of things. You can find out a lot about DF files by reading Eric Schmidt's dissertation [36]. You can find the answers to detailed questions about the behavior of the various programs that deal with DF files by reading the reference manual for DF files <37>. All that I will try to do here is to give you some idea of what DF files are good for, and how, in a general sense, they are used. One way or another, all Cedar programmers must make their peace with DF files: they perform valuable functions, and they have no current competition.

In the simplest case, a DF file just consists of a list of the names of a related set of files. At this level, a DF file is something like a dump file: given the DF file, you can get at each of the files that it describes. Of course, you want to be sure that you get the right versions of the described files, so just having the DF file list their names isn't quite enough. If there were an Internet-wide notion of version number that made sense, we could get around this problem by specifying the version number along with the file name. But there isn't. The closest thing to a Internet-wide unique identification stamp that we have is the create date of the file. Thus, what a DF file really contains is a list of file names and associated create dates.

The first program that you will meet that deals with DF files is Bringover. Bringover's job is to retrieve to your local file system the set of files described by a particular DF file. This would be something of a challenge unless the DF file included some hint to Bringover concerning where in the great, wide Internet the correct versions of these files might be found. So DF files do indeed include such hints: in particular, they include specifications of remote directories on which to look. These directories are just hints, in the sense that Bringover will always verify by checking the create date that it is getting you the correct version of the specified file. If Bringover can't find the correct version on the specified directory, it will issue a sprightly error message. Bringover has lots of bells and whistles. For example, you can point it at either a local or a remote DF file; you can ask it to retrieve just a selected subset of files to your local disk, rather than the entire set described by the DF file; or you can individually consider the files one by one, deciding which you would like to retrieve and which you wouldn't.

Suppose that I am working on a collection of files, such as the sources for this Briefing Blurb. I have made a DF file that describes them, and I can use Bringover to retrieve them from their remote and permanent home to my local file system, where I can edit them. The next thing that I need is a service that is symmetric to Bringover: after doing my editing, I want to put the new versions back on the remote file server, along with a new DF file that describes the new versions. This function is performed by SModel. I run SModel, and point it at the old DF file. SModel considers each file in turn, and looks to see if I have edited it; that is, it looks to see if the create date of that file in my local file system is now different than the create date stored in the old DF file. If so, SModel deduces that I have edited the file. It stores the new version that I have made out onto the remote directory. After doing this for each file in turn, SModel writes a new version of the DF file itself, filling in all of the create dates correctly to describe the new version of the entire ensemble. If the DF file describes itself, as most DF files do, SModel is smart enough to make sure that the new version of the DF file is stored out to the remote server as well. SModel also has lots of bells and whistles, but let's not go into them.

If that were the whole story, mere mortals could figure out DF files without straining their brains. But there's more. So far, we have only discussed DF files as descriptions of ensembles of files. In fact, these ensembles are often components of large programs. And this has consequences.

First, there are two distinctly different reasons that you might have for retrieving a program: you might want to change it, or you might just want to run it. In the latter case, you don't need to bring over all of the sources; all that you need is the runnable ".bcd". We could handle this by having two DF files: one for the users and the other for the maintainers. But that would be a disaster: the two DF files would never agree! Instead, each DF file distinguishes between files that it "exports" and the rest. The exported files are the ones that users need, while maintainers are assumed to need the entire ensemble. You can warn Bringover that you are a user rather than a maintainer by giving it the "/p" switch (which stands for Public-only).

Secondly, some programs are going to depend upon other programs: that is, the programs themselves will be "users" ("clients" is a better word here). This suggests that one DF file should be able to contain another DF file. In fact, there should be several different kinds of containment, corresponding to such phrases as:

"They who maintain me also maintain the stuff described by the following DF file."

"They who maintain me are also users (but not maintainers) of the stuff described . . ."

"They who use me are also users of the stuff described . . ."

You get the point? For more details on the ways that these things are done ("includes" and "imports"), check out the reference manual.

In case you still aren't convinced that things are complicated, it is now time to mention the fact that DF files are used for yet another purpose: they describe components of the Cedar release. During the Cedar release process, all of the new versions of Cedar components, which are sitting out on development directories, must be checked for consistency, and then moved en masse to the official release directory. And an entire new set of DF files must be produced, describing the released version of the system (as opposed to the development version). This means, among other things, that some DF files must specify two different remote directories: the development directory and the release directory. In addition, there is a third DF file program, called VerifyDF, whose purpose is to perform certain consistency and completeness checks on a DF file. By insisting that all component implementers have successfully run VerifyDF on their components, the Cedar Release Master ensures that the release process has at least a fighting chance of succeeding <38>.

In fact, there are several other programs related to DF files that are sometimes useful. DFDisk, for example, looks at all of the files on your local disk and classifies them according to where they may be found on remote servers. This is a convenient way to determine what local files need to be backed up before erasing the local file system for some reason. For more on DFDisk, an introduction to DFDelete, and more, see the DF files reference manual <37>.

DF files grew up over time in response to a mixed bag of needs. As they became more popular, features were added one by one to make them more useful in these varying contexts. The resulting system as a whole is rather hard to grok, but I hope that this introduction has given you a leg up on the problem, at least.

Some Tidbits of Lore

About CSL

CSL has a weekly meeting on Wednesday afternoons called Dealer, starting at 1:15. The name comes from the concept of "dealer's choice"—the dealer sets the ground rules and topic(s) for discussion. When someone says she will "give a Dealer on X", she means that she will discuss X at some future weekly meeting, taking about 15 minutes to do so (plus whatever discussion is generated). Generally, such discussions are informal, and presentations of half-baked ideas are encouraged. The topic under discussion may be long-range, ill-formed, controversial, or all of the above. Comments from the audience are encouraged, indeed, provoked. More formal presentations occur at the Computer Forum on Thursday afternoons; the Forum is not specifically a CSL function, and it is open to all Xerox employees, and sometimes also to outsiders. Dealers are also used for announcements that are not appropriate for distribution by electronic mail. Members of CSL are expected to make a serious effort to attend Dealer.

On occasions of great festivity, Dealer is replaced by a picnic on the hill (that is, Coyote Hill, across Coyote Hill Road), with Mother Xerox picking up the tab.

The CSL Archives (not to be confused with TENEX archiving) are a collection of file cabinets and 3-ring binders that provide a continuing record of CSL technical activities. The archives are our primary line of defense in legal matters pertaining to our projects. They also make interesting reading for anyone curious about the history of any particular project.

There is also an institution known as the CSL Notebook, which exists to make all of the potentially interesting documentary output of CSL folk easily accessible to all CSL folk. Trip reports, design documents, immigration manuals (like this one): they should all be submitted to the CSL Notebook <39>. If you thought that it was worth writing down, it is pretty likely that there are other folk in CSL who would consider it worth reading, and submitting it to the CSL Notebook is one easy way to get it read. (I believe that likely looking submissions to the CSL Notebook are considered for entry into the CSL Archives as well.)

About ISL

ISL also has a weekly meeting, on Tuesdays starting at 11:00 am. This meeting has no catchy name at the moment.

Some Code Phrases

You may occasionally hear the following incomprehensible phrases used in discussions, sometimes accompanied by laughter. To keep you from feeling left out, we offer the following translations:

"Committing error 33"

(1) Predicating one research effort upon the success of another. (2) Allowing your own research effort to be placed on the critical path of some other project (be it a research effort or not). Known elsewhere as Forgie's principle.

"You can tell the pioneers by the arrows in their backs."

Essentially self-explanatory. Usually applied to the bold souls who attempt to use brand-new software systems, or to use older software systems in clever, novel, and therefore unanticipated ways ... with predictable consequences. Also heard with "asses" replacing "backs".

"We're having a printing discussion."

Refers to a protracted, low-level, time-consuming, generally pointless discussion of something peripherally interesting to all. Historically, printing discussions were of far greater importance than they are now. You can see why when you consider that printing was once done by carrying magnetic tapes from Maxc to a Nova that ran an XGP.

Fontology

The body of knowledge dealing with the construction and use of new fonts. It has been said that fontology recapitulates file-ogeny.

"What you see is what you get."

Used specifically in reference to the treatment of visual images by various systems, e.g., a Bravo screen display should be as close as possible to the hardcopy version of the same text. Also known is some circles by the acronym "WYSIWYG", pronuonced "whiz-ee-wig".

"Moving right along", "Pop!", or *"Hey guys, up-level!"*

Each of these phrases means that the conversation has degenerated in some respect, often by becoming enmeshed in nitty-gritty details. Feel free to shout out one or more of these phrases if you feel that a printing discussion has been going on long enough. If two participants in a large meeting begin discussing details that are of interest to them but not of interest to the group as a whole, shout *"Off-line!"* instead.

"Life is hard"

Two possible interpretations: (1) "While your suggestion may have some merit, I will behave as though I hadn't heard it." (2) "While your suggestion has obvious merit, equally obvious circumstances prevent it from being seriously considered." The charm of this phrase lies precisely in this subtle but important ambiguity.

"What's a spline?"

"You have just used a term that I've heard for a year and a half, and I feel I should know, but don't. My curiosity has finally overcome my guilt." Moral: don't hesitate to ask questions, even if they seem obvious.

Hints for Gracious Living

There are a couple of areas where life at PARC can be made more pleasant if everyone is polite and thoughtful enough to go to some effort to help out. Here are a few words to the wise:

Coffee

Both ISL and CSL have coffee alcoves where tea, cocoa, and several kinds of coffee are available. All coffee drinkers (not just the secretaries or some other such barbarism) help out by making coffee. If you are about to consume enough coffee that you would leave less than a full cup in the pot, it is your responsibility to make a fresh pot, following the posted instructions. There are lots of coffee fanatics around, and they get irritated beyond all reason if the coffee situation isn't working out smoothly. For those coffees for which beans are freshly ground, the local custom is to pipeline grinding and brewing. That is, you are expected to grind a cup of beans while brewing a pot of coffee from the previous load of ground beans. This speeds up the brewing process for everyone, since a load of ground beans is—at least, had better be—always ready when the coffee pot runs out.

Sharing Office Space

Be warned as well that some lab members are unbelievably picky about the state of their offices. The convention is that any Alto in an empty office is fair game to be borrowed. Private Dolphins and Dorados may be borrowed only by prior arrangement with their owners, because of the problems of sharing disk space. If you use someone's office for any reason, take care to put everything back *exactly* the way it was. Don't spill crumbs around, or leave your half-empty cocoa cup on the desk, or forget to put the machine back in the state that you found it, or whatever. Of course, lots of people wouldn't mind even if you were less than fanatically careful. But some people do mind, and there is no point in irritating people unnecessarily.

Sharing printers

When you pick up your output from a printer, it is considered antisocial merely to lift your pages off the top of the output hopper, and leave the rest there. Take a moment to sort the output into the labelled bins. Sorting output is the responsibility of everyone who prints, just as making coffee is the responsibility of everyone who drinks (coffee). Check carefully to make sure that you catch every break page: short outputs have a way of going unnoticed, and hence being missorted, especially when they come out right next to a long output in the stack. The rule for determining which bin is to use the first letter that appears in the name on the break page. Thus, "Ramshaw, Lyle" should be sorted under "R", while "Lyle Ramshaw" should be sorted under "L". A trickier question is what to do with output for "Noname", or the like. Following the rule would suggest filing such output under "N", but that doesn't seem very helpful, since the originator probably won't find it. Check the contents and file it in the right box if you happen to recognize whose output it is; otherwise, either leave it on top of the printer or stick it back in the output hopper.

The phone system

When the Voice Project has had its way, our phone system will be a marvelous assemblage of computers talking to computers, and this section of the Briefing Blurb will have to be expanded to tell you all about it. At the moment, however, we are simply customers of Pacific Telephone, so there isn't too much to say. First, a little preaching.

If you make a significant number of personal long-distance phone calls from Xerox phones, it is your responsibility to arrange to reimburse Xerox for them. This may not be that easy, either, since phone bills take quite a while (six weeks or so) to percolate through the bureaucracy upstairs, and the said bureaucracy also has a lot of trouble figuring out where to send the phone bills of new people, and people who move around a lot. Just because it is easy to steal phone service from Xerox doesn't make it morally right; if you think you aren't being paid enough, you should start agitating for a raise. If enough suspicious calls are made without restitution, PARC (being a bureaucracy) will impose some bureaucratic "solution" on all of us.

So as not to end on a sour note, let's discuss how the phone system works, anyway. The offices within PARC have four-digit extensions within the 494 exchange, a system known as Centrex; to dial another office, those four digits suffice. Dialing a single 9 as the first digit gives you an outside line, and you are now a normal customer of Ma Bell: see a phone book for more details (Oh, come now, surely you know about phone books!). Dialing a single 8 gives you different sounding dial tone, and puts you onto the IntelNet (not to be confused with the InterNet). The IntelNet is a Xerox-wide company phone system, complete with its own phone book, and its own phone numbers. If you are calling someone in some remote part of Xerox, you can save Mother Xerox some bread by using the IntelNet instead of going straight out over Ma Bell's lines. On the other hand, you may not get as good a circuit to talk over—although this situation is frequently said to be improving. Furthermore, through the wonders of modern electronics, you can dial any long-distance number over the IntelNet. Just use the normal area code and Ma Bell number: the circuitry is smart enough to take you as far as possible towards your destination along IntelNet wires, and then switch you over to Ma Bell lines for the rest of the trip. Using the IntelNet doesn't start to save money until the call is going a fair distance; therefore, the IntelNet doesn't let you call outside numbers in area codes 408, 415, and 916—better to just dial 9.

One more thing: after you have dialed a number on the IntelNet, you will hear a funny little beeping. At that point, you are being asked to key in a four-digit number to which the call should be billed. You should use the four-digit extension number for your normal office phone under most circumstances. Calls made by dialing 9 instead of 8 are always charged to the phone from which they are placed.

The first three rings (roughly speaking) of an incoming call occur only in your office. The next roughly three rings happen both at your office phone and at a receptionist's phone, centrally located in the laboratory. During normal business hours, the receptionist's phones are staffed; thus, someone will at least take a message for you, and leave it on a little slip of paper in your physical message box. If the second three rings go by without either of those two phones answering, the call is then forwarded to the guards desk downstairs (I believe).

If you are expecting a call but won't be near your normal phone, a call forwarding facility exists: dial 106 and then the number to which you want your calls to be forwarded. Later on (*try* not to forget), you dial 107 on your normal phone to cancel the forwarding. When I forward my phone, I turn the phone around physically, so that the touch-pad faces the wall. This helps me to remember to cancel the forwarding again later, at which point I turn the phone back the normal way. There is also a way to transfer incoming calls to a different Xerox number: Depress the switch hook once, and dial the destination number; when the destination answers, you will be talking to the destination but the original caller won't be able to hear your conversation; depressing the switch hook again puts all three of you on the line; then you can hang up when you please. If the destination doesn't answer, depressing the switch hook once again will flush the annoying ringing or busy signal.

References

Reference numbers in [square brackets] are for conventional hardcopy documents. Many of them are Xerox reports published in blue and white covers; the CSL blue-and-whites are available on bookshelves in the CSL Alcove. Reference numbers in <angle brackets> are for on-line documents. The path name for such files is given herein in the form

[FileServer]<Directory>SubDirectory>FileName.Extension

for backward compatibility with earlier systems. Recently, the simpler alternative form

/FileServer/Directory/SubDirectory/FileName.Extension

has begun to come into local currency, but some systems still demand brackets rather than slashes.

<*n*>: The generic form for a reference to an on-line document.

[*n*]: The generic form for a reference to a hardcopy document.

[1]: **Sunset New Western Garden Book.** Lane Publishing Company, Menlo Park, CA, 1979. The definitive document on Western gardening for non-botanists; 1200 plant identification drawings; comprehensive Western plant encyclopedia; zoned for all Western climates; plant selection guide in color.

[2]: John E. Warnock. **The Display of Characters Using Gray Level Sample Arrays.** blue-and-white report CSL-80-6.

[3]: Richard F. Lyon. **The Optical Mouse, and an Architectural Methodology for Smart Digital Sensors.** blue-and-white report VLSI-81-1.

[4]: **The Ethernet Local Network: Three Reports.** blue-and-white report CSL-80-2.

[5]: John F. Shoch, Yogen K. Dalal, Ronald C. Crane, and David D. Redell. **Evolution of the Ethernet Local Computer Network.** blue-and-white report OPD-T8102.

<6>: [Maxc]<AltoDocs>NetTopology.press. Contains a picture of the entire internetwork configuration in seven pages. It is out of date. All such documents are always out of date. A copy is posted on the wall opposite the Alcove in CSL.

[7]: David R. Boggs, John F. Shoch, Edward A. Taft, and Robert M. Metcalfe. **Pup: An Internetwork Architecture.** blue-and-white report CSL-79-10.

[8]: **Internet Transport Protocols.** Xerox System Integration Standard report XSIS 028112, December 1981.

[9]: **Courier: The Remote Procedure Call Protocol.** Xerox System Integration Standard report XSIS 038112, December 1981.

[10]: C..P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. **Alto: A personal computer.** blue-and-white report CSL-79-11.

<11>: [Maxc]<AltoDocs>AltoHardware.press. Everything that you need to know to write your own Alto microcode.

[12]: **The Dorado: A High-Performance Personal Computer; Three Papers.** blue-and-white report CSL-81-1.

<13>: [Indigo]<DoradoDocs>DoradoBooting.press. Describes how to boot a Dorado, and how to configure it to boot in various ways.

[14]: Myer, T. H.. and Barnaby, J. R. **TENEX Executive Language Manual for Users.** Available from Arpa Network Information Center as NIC 16874, but in the relatively unlikely event that you need one, borrow one from a Tenex wizard.

<15>: [Maxc]<AltoDocs>BCPL.press. The reference manual for the BCPL programming language.

<16>: [Maxc]<AltoDocs>OS.press. The programmer's reference manual for the Alto Operating System, including detailed information on the services provided and the interface requirements.

<17>: [Maxc]<AltoDocs>Packages.press. A catalogue giving documentation for the various BCPL packages that other hacker's have made available.

[18]: James G. Mitchell, William Maybury, and Richard Sweet. **Mesa Language Manual, Version 5.0.** blue-and-white report CSL-79-3. A cross between a tutorial and a reference manual, though much closer to the latter than the former.

<19>: [Ivy]<Mesa>Doc>Compiler60.press. Describes the changes in the Mesa language and the compiler that occurred in moving from Mesa 5.0 to Mesa 6.0.

[20]: Morris, J. H. **The Elements of Mesa Style.** Xerox PARC Internal Report, June 1976. Somewhat out of date (since Mesa has changed under it), but a readable introduction to some useful program structuring techniques in Mesa.

[21]: Adele Goldberg and David Robson. **Smalltalk-80: The Language and Its Implementation.** book published by Addison-Wesley, 1983.

[22]: Warren Teitelman. **Interlisp Reference Manual.** Published in a blue and white cover, although not officially a blue-and-white. October, 1978.

[23]: The Interlisp-D Group. **Papers on Interlisp-D.** blue-and-white report CIS-5 (also given the number SSL-80-4), Revised version, July 1981.

[24]: L. Peter Deutsch and Edward A. Taft, editors. **Requirements for an Experimental Programming Environment.** blue-and-white report CSL-80-10.

<25>: [Indigo]<Cedar>Documentation>Manual.df. Hardcopies are entitled **The Cedar Manual.**

[26]: **Alto User's Handbook.** Internal report, published in a black cover. The version of September, 1979 is identical to the version of November, 1978 except for the date on the cover and title page. Includes sections on Bravo, Laurel, FTP, Draw, Markup, and Neptune

<27>: [Maxc]<AltoDocs>SubSystems.press. Documentation on individual Alto subsystems, collected in a single file. Individual systems are documented on [Maxc]<AltoDocs>systemname.TTY, and these files are sometimes more recent than SubSystems.press.

[28]: Jerome, Suzan. **Bravo Course Outline.** Internal report, published in a red cover. Oriented to non-programmers.

<29>: [Indigo]<Tioga>Documentation>TiogaDoc.tioga, or TiogaDoc.press. How to use the Tioga editor.

<30>: [Maxc]<PrintingDocs>PressFormat.press. Describes the Press print file format.

<31>: [Maxc]<PrintingDocs>PressOps.press. Describes the Press printing program.

<32>: [Maxc]<PrintingDocs>PDPrintOps.press. Describes the PDPrint printing program.

[33]: Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. **Grapevine: an Exercise in Distributed Computing.** blue-and-white report CSL-82-4.

<34>: [Ivy]<Laurel>Maintain.press. Documentation for the teletype version of Maintain, the version that is used from within Laurel or Tajo.

[35]: Douglas K. Brotz. **The Laurel Manual.** blue-and-white report CSL-81-6.

[36]: Eric Emerson Schmidt. **Controlling Large Software Development in a Distributed Environment.** blue-and-white report CSL-82-7.

<37>: [Indigo]<Cedar>Documentation>DFFilesRefMan.press. The reference manual for the use of DF files.

<38>: [Indigo]<Cedar>Documentation>ReleaseProcedures.press. Describes the policies and procedures that individuals who contribute to Cedar releases need to understand and observe.

<39>: [Indigo]<CSL-Notebook>Docs>HowToUseCSLNotebook.press.

A Glossary of Terms, Subsystems, Directories, and Files

(and acronyms, protocols, and other trivia)

Try reading me in Tioga, using the "Def" command to get around!

abstract machine: A set of low-level functions and capabilities, provided by some combination of hardware and software, that forms the underpinnings of a system sitting above. For example, the **Interlisp-D** system, which runs on various machines, consists of a lot of machine-independent stuff sitting on top of a small amount of machine-dependent code. The goals of the machine-dependent part were specified by describing an abstract machine that it must implement. As another example, part of the **Cedar** project has been the specification of a "Cedar computer" as an abstract machine.

Alpine: A transactional file server being built within **CSL** for use by database systems and other distributed computing applications. Alpine is being built on top of **Cedar**, and will help support the **FS** file system.

Alto: (On its way to being archaic.) A small personal computer with a **bitmap** display and **mouse**, designed at **PARC**; the precursor to **D-machines**. See the **blue-and-white** report titled "Alto: A personal Computer", number CSL-79-11.

Alto world: An environment created by running an **Alto emulator** on a **D-machine**.

AltoFontGuide.Press: A file, available on **[Indigo]<Fonts>**, that tells all about the existing families of display-screen raster **fonts**, and describes how they are organized on different subdirectories of **[Indigo]<AltoFonts>**. Note that the name "AltoFonts" is an anachronism, and should really be changed to "DisplayScreenFonts" or the like; the same rasters that were drawn for use on **Altos** work just fine on today's **D-machines**.

AM: Acronym for the **Cedar** abstract machine.

ARPA: Acronym for the Advanced Research Projects Agency of the United States Department of Defense. They support, among other things, a network linking research computers: our ARPANET address is PARC-MAXC.

atom: (or ATOM:) Unique identifiers implemented over a global naming space. Two occurrences of the same atom will evaluate to the identical value, rather than just to equivalent values. Atoms have always been part of **Interlisp**; they were added to **Mesa** on the way to **Cedar**. In **Cedar**, an atom literal is written with a prefixed dollar sign, as in "$foo". Each atom has a list of <name, value> pairs associated with it, called its property list.

bank: A unit of measurement of primary storage in an **Alto world**, equal to 64K 16-bits words, that is, 128K bytes. An **Alto** II has four banks, while **Dorados** have at least eight.

bar: A generally thin, generally rectangular, generally invisible region of the screen in which certain generally display-related actions occur, e.g., the scroll bar, the line-select bar.

baseboard: A microcomputer that lives on the lowest printed-circuit board of a **Dorado**. The baseboard listens to the terminal's boot button, and to various thermometers. Its job is to supervise the rather complex booting sequence necessary for bringing a Dorado up from a cold start. The baseboard announces its state to the outside world by flashing a number (encoded in unary) on a little green light on the Dorado chassis. Signs near each bank of Dorados explain what various numbers of flashes mean.

Bayhill: Another name for **Building 96**, occupied by part of SDD. The **Bayhill** building is located on Hillview just before it runs into Arastradero.

BCD: A compiled object program module in **Mesa** or **Cedar**; an acronym for Binary Configuration

Description.

BCPL: A free-wheeling and typeless system programming language used as the environment for much early Alto programming. Also, the compiler for that language.

BFS: An acronym for Basic File System; the contents of a disk or **partition** used by an **Alto world**. Also a standard software package for low-level management of an Alto file system.

Binder: BCD's **export** services to their **clients**, and, in turn, **import** various services from other BCD's. The process of resolving these inter-module references is called *binding*, and the Binder is the program that does it. Actually, the loader can handle many of the easy cases of binding on the fly, as part of the loading process; but for complex stuff, you need the Binder. The Binder accepts compiled modules (with extension ".bcd") and binding instrutions in the form of a configuration description (with extension ".config"); it produces another ".bcd" as output.

BITBLT: (pronounced "bit-blit"). A complex instruction used for moving and possibly modifying a rectangular **bitmap**. The "BLT" part is an acronym for BLock Transfer.

bitmap: Generally refers to a representation of a graphical entity as a sequence of bits directly representing image intensity at the points of a raster. The display hardware and microcode on an **Alto** or **D-machine** process what is essentially a bitmap of the image to be displayed. At PARC, bitmaps are normally stored word-aligned, and in row-major order.

blue-and-white: A report that has been cleared for distribution outside Xerox, and published in a blue and white cover. Such reports have identifying numbers formed by concatenating the laboratory acronym, the year, and a small integer. One of my favorites is the Laurel Manual, by Douglas K. Brotz, number CSL-81-6; I especially recommend Chapter 6. **CSL** blue-and-whites are stored on bookshelves in the CSL Alcove. A list giving the titles and numbers of all of the blue-and-whites is available from the PARC Library.

boot: Short for "bootstrap", which is in turn short for "bootstrap load". Refers to the process of loading and starting a program on a machine whose main memory has undefined contents.

boot button: The small button behind the keyboard used (sometimes in conjunction with the keyboard) to **boot** some program into execution. On **Dolphin's** or **Dorado's**, there are other more potent boot buttons on the chassis, in addition to the boot button behind the keyboard.

boot file: A file that contains a bootable program. Used to start **Cedar**, as well as various games and other useful programs available from the **NetExec** in the **Alto world.**

boot server: A computer on the network that provides a retrieval service for certain stand-alone programs (which are encapsulated as **boot files**). See **NetExec.**

bootlights: (archaic) A screen pattern resembling a city skyline. Occurs occasionally in the **Alto world** when some erroneous unanticipated condition arises, e.g., getting a parity error in a **BCPL** program on a disk that doesn't have **Swat.**

Bravo: (archaic) An integrated text editor and document formatting program that runs on the **Alto**; a vital program that nevertheless is no longer maintained or supported.

BravoBug: (archaic) A program used when Bravo crashes to **replay** the editing actions up to the point of the crash.

BravoX: A successor to **Bravo** written in **Butte** with somewhat greater functionality and a somewhat richer interface. Warning!: BravoX source files are stored in a weird and wonderful format that almost NO programs other than BravoX can handle. Also, BravoX runs, at the moment, only on Alto II's and (perhaps?) Dolphins.

break page: A header page that divides one printed file from another in the output of a **Spruce** printer. If Spruce encountered any difficulties during the printing run, it will inform you of them on the break page.

Bringover: A program that retrieves files from remote file servers to one's local disk; Bringover reads ".df" files in order to figure out what versions of what files should be retrieved, and where in the great wide electronic world they might be found. Use of Bringover (confusing as it may be at the outset) is to be recommended over use of either **FTP** (in the **Alto world**) or the **FileTool** (in **Cedar**), since the version control and system-description features of ".df" files are very valuable.

bug: A computing term for a non-feature, something that is not as intended. Sometimes used in a different sense to refer to the act of pointing at something with the mouse, and then clicking a mouse button; but this usage is frowned upon by 100% of our Usage Panel (namely me: I recommend using the verb "click" instead in this context, since I think that "bug" is already an overloaded word).

bug award: Refers to a occasional custom within **CSL** and **ISL** wherein those brave souls responsible for ferreting out the cruelest and most intricate bugs in critically important systems are rewarded for their efforts by being presented with a cute little bug-shaped sticker that they can then display on their office nameplate or elsewhere. A bug award is the moral equivalent of a gold star. If the sticker consists of a background from which a bug has been excised, then the award is an "inverse bug award", and serves to praise its recipient for producing code that is notably free of insect infestations.

BugBane: A package that implements the basic primitives necessary for high-level debugging in the **Cedar** world; the **UserExec** is a **client** of BugBane, and, in turn, provides debugging services to **users** of Cedar.

Building 32: A part of **OSD**, located on Hanover Street, north of Page Mill. Once called **PARC-place**, when it was occupied by parts of PARC.

Building 34: A part of **PARC**, located on Hillview, just across Coyote Hill from the **Building 35**, the home of the **ICL**.

Building 35: The main building of **PARC**, located at the intersection of Coyote Hill and Hillview. The site of the cafeteria.

Building 37: A part of **PARC**, located on Hanover Street, north of Page Mill, and just south of **Building 32**. The site of the **CSL Electronic Model Shop**.

Building 96: A part of **OSD**, located where Hillview runs into Arastradero; also called the **Bayhill** building. Current home of some parts of SDD.

Butte: A compiler for **BCPL** that outputs **Mesa**-style byte codes instead of Nova assembly code; also, the byte codes themselves, and the microcode that implements them.

button: A small area on the screen that reacts when clicked with the mouse. In **Viewers**, buttons are rectangular areas labelled with a word or phrase; they are organised into **menus**.

byte code: **Lisp**, **Mesa**, **Cedar**, **Smalltalk**, and **Butte** at **PARC** compile into directly executable languages that are stack oriented, and whose op codes are usually one byte long. Such an instruction is called a **byte code**. These **byte codes** are in turn interpreted by special microcode on each of our various machines.

Cabernet: A particular mail server that is part of the **Grapevine** distributed transport mechanism, located in the **CSL** machine room.

caret: A blinking pointer, indicating where keyboard characters will appear when typed.

catch phrase: A chunk of **Mesa** or **Cedar** code that is prepared to handle a certain type of exceptional condition. The best of way to think of a catch phrase is as the body of a procedure variable that is dynamically bound. Such procedures variables are called **signals**. If you suspect that an exceptional condition might arise, and you think that you knows what to do if it does, you specify this response as a catch phrase; that is, you bind a procedure value to the signal, which is a procedure variable. If any procedure that you call notices that the condition has in fact arisen, it will notify the world by "raising the signal", which should be thought of as a procedure call to the catch phrase that you specified. (This method of explaining signals is a minor facet of the religion esposed in the **CLRM**.)

Cedar: A large project in CSL to build a programming environment for CSL's future applications. Also the name of that environment. Also the name of the programming language upon which it is built. The Cedar language is a variant of **Mesa** augmented by garbage collection, **atoms**, and run-time types. The design of the **Cedar** environment was strongly influenced by the programming environment and services available in **Interlisp** and **Smalltalk**. For a discussion of the goals of Cedar, see the **blue-and-white** report titled "Requirements for an Experimental Programming Environment", number CSL-80-10.

CedarGraphics: A subroutine package of graphic primitives, built within **ISL**, that forms an important part of **Cedar**. Its design was heavily influenced by the results of experimental systems written in **JaM.**

Chardonnay: A **Grapevine** server.

Chat: A program that provides teletype-like "interactive" access to a remote computer on the network. Most programming environments include this capability in some form; both Alto and Cedar include programs actually named "Chat". Chat is mainly used to communicate with **Maxc** and **IFS** servers.

Checkpoint: A method used in **Cedar** to preserve the state of your computing world. Taking a Checkpoint involves preserving a shapshot of the current state of the virtual memory, but not of the file system. If, after taking a Checkpoint, something bad happens and your Cedar system gets **wedged**, the command **RollBack** will return you to the earlier clean state of your virtual memory; but changes to the file system made between the Checkpoint and the subsequent RollBack, such as storing edited versions of files, will not be undone.

Cheshire: A subsidiary of Xerox. They make a machine that binds stacks of paper into booklets by melting glue and letting it be absorbed by the edges of the paper. There are Cheshire binders in **CSL** and in the PARC **TIC**.

Chromalin: The trade name of a fancy color printing process used with the **PlateMaker** for creating high-resolution color prints from **Press** files or **PD** files.

Chipmunk: A **D-machine Mesa** program for interactively creating and editing integrated circuit designs. Chipmunk makes use of a color display in addition to the normal black-and-white one. It is a successor to **Icarus.**

Cholla: A **Laurel**-based IC fabrication line control program, which is used in **ICL**.

CIFS: An acronym for Cedar Interim File System. CIFS is currently used within **Cedar** to manage a portion of the local disk as a cache containing readonly copies of remote files. This function and others will someday be provided by FS. CIFS was the first CSL system to allow the components of a hierarchical file name to be separated with simple slashes instead of with square brackets and angle brackets; the clumsier brackets are being used in this document (sigh) for compatibility with the past.

CIS: An acronym for Cognitive and Instructional Sciences Group. A part of PARC, and the home of many of the builders of **Interlisp-D**.

Clearinghouse: The analog of the **Grapevine** registration database in the **NS** world. That is, a machine running **Star** talks to the local Clearinghouse in order to find out how to talk to a particular **file server** or **print server**.

click: A manipulation of a mouse button. Pushing and releasing a mouse button several times in quick succession is sometimes called a "double-click", "triple-click", etc. as appropriate. The phrases "click-hold" and "double-click-hold" are also sometimes heard.

client: A program (as opposed to a person) that avails itself of the services of another program or system. **Laurel** is a **client** of **Grapevine.** See **user.**

Clover: A **Dover** located in CSL.

CloverFonts.Press: A file, available on **[Indigo]<Fonts>**, that lists by family name, face, size, and rotation all of the fonts in **Clover**'s font dictionary. That is, this file lists the fonts that you can print with; for the fonts that you can see on your screen, see **AltoFontGuide.Press** instead. To see the characters of the fonts in all their glory, check out the book located on top of Clover called CloverCharacters.Press.

CLRM: Acronym for the Cedar Language Reference Manual. This document isn't exactly easy bedtime reading, but it is the most authoritative description currently available of the behavior of Cedar programs in interesting and subtle cases. The CLRM also attempts to convert you to a particular religion regarding the proper design of a polymorphic language within the Algol tradition. To get the good dope about current Cedar without spending the time necessary to undergo religious conversion, skip immediately to Chapters 3 and 4 of the CLRM.

CoCedar: A **world-swap** debugger for **Cedar.**

color display: A CRT display with red, green, and blue phosphors. **Griffin** and **Chipmunk** both use the color display, and the color display is also available to users of Cedar with a minimum of hassle through the good auspices of the Cedar **ColorDevice.** The public **Dorados** with color displays are listed at the sign-up sheets.

ColorDevice: A component of **Cedar** that provides low-level support for a **color display.**

Com.cm: A file used by the Alto **Executive** to store the current command being executed. See **Rem.cm.**

Commander: A "light-weight" command interpreter, providing the minimum of functionality needed by **Cedar** implementers while they are developing a new release of the system. Most users of Cedar can instead enjoy the more plentiful features of the **UserExec.**

component: Among many other things, a chunk of software that is distributed as part of a **Cedar release.**

config: A source file that tells the **Binder** how to assemble modules into a complete system.

CoPilot: A **world-swap** debugger for **Pilot.**

CopyDisk: A stand-alone program used to transfer an Alto **BFS**, that is, the entire contents of an Alto disk or **partition.** May be used between computers or on a single computer with multiple disk drives.

create date: When said of a file, the date and time that the information contained in this particular version of this particular file was created. Create dates are generally stored accurate to the nearest second. This makes them sufficiently unique that the pair <file name, file version's create date> can serve as a unique identifier for a particular pile of bits.

credentials: Proof that you are who you say you are; usually your **Grapevine R-name** and the corresponding password.

CSL: Acronym for Computer Science Laboratory, a part of **PARC**, located on the second floor of **Building 35.**

CSL Notebook: A mechanism for distributing, indexing, and generally sharing the documentary output of folk in **CSL.**

cursor: A small picture on the display that tracks the motions of the **mouse.**

Cypress: A database package based upon an entity-value-relationship model of data, and written in **Cedar. Walnut, Hickory,** and **Squirrel** are clients of Cypress.

D-machine: A generic name, referring to any of the current machines within Xerox that implement the **PrincOps** architecture: **Dandelions, Dicentras, Dolphins,** and **Dorados** are the primary **D-machines.**

D0: ("D-zero", not "DO") An obsolete name for the **Dolphin**, a **D-machine.**

Daffodil: An inexpensive **D-machine** using custom VLSI, being designed by local folk. Since the Daffodil may become the hardware base of future OSD products, certain details concerning the Daffodil project are rather sensitive.

Daisy: A **Dover** located in the **Bayhill** building.

Dandelion: The name of the processor that is in the **Star** products; an example of a **D-machine.**

dead: Either not currently operational (said of a piece of hardware), or operational but not currently undergoing continued development and support (said of bodies of software).

Dealer: The name of **CSL**'s weekly meeting, occurring on Wednesday afternoons from 1:15 until 2:45 (or so); also used to refer to the person speaking at that meeting. Giving such a presentation is referred to as "giving a Dealer" or sometimes "Dealing". See also **weekly meeting.**

DDS: (archaic) Acronym for Descriptive Directory System. An Alto **subsystem** providing sophisticated manipulation of the Alto file directory system. See also **Neptune.**

DF files: · A collection of programs for describing the files needed to build a complicated system, for automatically retrieving these files from remote **file servers** to the local disk (**Bringover**), and for storing them back later (**SModel**). Unlike the more grand and glorious **system models** to come, DF files primarily addresses the problems engendered by our current feudal collection of file systems. The letters "DF" are an acronym for Description Files, which suggests that the phrase "DF files" is redundant.

Dicentra: A recent and inexpensive **D-machine.** The Dicentra essentially consists of the **Dandelion**'s CPU squeezed onto one **Multibus** card, and communicating with memory and with I/O device controllers over the Multibus.

dirtball: A small, perhaps struggling outsider; not in the major or even the minor leagues. For example, "Xerox is not a **dirtball** company".

distribution list: A list of **R-names** to which mail can be addressed. In some cases, **Maintain** can be used to add oneself to interesting DL's, such as "MesaFolklore↑.pa". If Maintain responds that you aren't allowed to do that, the correct recourse is to send a polite message to "Owners-MesaFolklore↑.pa", asking that they please add you to their list. For more details about distribution lists, try either the **Laurel** manual or the document [Ivy]<Laurel>Maintain.Press, which describes the **Alto** and **Tajo** versions of Maintain.

DiskDescriptor: A file that contains the disk allocation information used by an Alto file system.

DL: Acronym for Distribution List.

DLS: Acronym for Data Line Scanner: an Alto equipped with lots of modems plus other hardware and microcode to allow dialing into and out of the **Internet.**

DMT.boot: Acronym for Display Memory Tester. A memory diagnostic for the **Alto world.** DMT is automatically booted from the network by the **Alto Executive** after the Alto has been idle for about 20 minutes. DMT accepts various commands: try pushing the "S" key, and also try typing shift-swat. Designing cursors for DMT is a popular sport: send your suggestion as a list of 16 octal numbers to David Boggs (Boggs.PA), along with a suggested title line and an indication of whether you want to be credited by name.

Dolphin: A **D-machine**; once called the **D0.** More flexible than a **Dandelion**, but also slower and more expensive.

Dorado: A high-performance **D-machine**, designed by **CSL** and coveted by all and sundry. See the **blue-and-white** report titled "The Dorado: A High-Performance Personal Computer", number CSL-81-1.

Dover: Generic name for a type of 384 bpi laser-scan printer built on the Xerox 7000 xerographic engine and connected to an Alto by means of a **Orbit** interface. Successor to **EARS. Dovers** are normally driven by the program **Spruce.**

Dragon: Generic name of a new, custom-chip processor being designed by a team in CSL; it is hoped that the **Dragon** will satisfy our ambitions to have "a **Dorado** in a shoe box".

Draw: (archaic) An Alto **subsystem** that permits interactive construction of pictures composed of lines, curves, and text. Draw users may be interested to note that a program ReDraw exists that converts Draw source files into **Press** files that will print without the **jaggies** on a **Dover.** Users of **Alto emulators** on **D-machines** must use DDraw and ReDDraw instead.

Dumper.boot: (archaic) A file used for desperation debugging in an **Alto world.** Dumps (most of) the current core image to **Swatee** for subsequent inspection by a debugger.

DWIM: Acronym for Do What I Mean: a facility intended to help the programmer by making LISP do what you mean, rather than what you say.

EARS: (archaic) Acronym for Ether Alto Research SLOT. An obsolete prototype laser-scan printer built on the Xerox 7000 xerographic engine and equipped with a hardware character generator. (Interesting to some as an example of a third level acronym: the S in EARS stands for SLOT, and the L in SLOT stands for LASER, and LASER itself is an acronym!)

EditTool: A menu-oriented command interface to the **Tioga** editor, providing complete access to Tioga's functionality, including the commands that you can't type (either because they can't be typed, or because you have forgotten how to type them).

EFTP: A venerable **PUP**-based protocol now mostly used to transfer print files to **print servers.**

Electronic Model Shop: An arm of **CSL** located on Hanover street in **Building** 37; this group of folks do small-scale production runs of computer equipment for **CSL.** Frequently called the **Garage.**

EmPress: An Alto **subsystem** used to convert text files to **Press** format and ship them to a **Press print server.**

emulator: A technique in which one computer is programmed to imitate another. Fast imitations are called emulators, while sufficiently slow ones are called simulators.

EOS: Acronym for Electro-Optical Systems; an organization located in Pasadena that was formerly a part of Xerox. The defense contracting portion of EOS was recently sold by Xerox for 40 megabucks. The portion of EOS that built Scientific Information Systems is now SIS; they are the ones who are marketing **D-machines** running **Interlisp** and **Smalltalk** to the outside world.

Ernestine: A particular **Lily** server located in Building 35.

Ethernet: The communication line connecting many computers (with compatible interfaces) together. Strictly speaking, an Ethernet is a single, continuous piece of co-axial cable, but the term is sometimes applied to the entire network accessible through the cooperation of **Gateways** (which is more correctly called an **InterNet**). Ethernets come in two flavors: the original Ethernet, now called the Experimental Ethernet, was built within PARC and runs at 3 MBits/sec. The Ethernet that has been proposed as a communication standard is a re-engineering that runs at 10 MBits/sec. PARC currently has Ethernets of both these flavors running around, as well as a special 1.5MBits/sec Ethernet used by the **EtherPhones.** See the **blue-and-white** report titled "The Ethernet Local Network: Three Reports", number CSL-80-2.

EtherPhone: A box of magic widgets that can replace your office telephone, giving you much greater functionality by taking advantage of the power of computing in general, and of your personal multi-function workstation in particular. An EtherPhone has a microphone, a speaker, digital-to-analog and analog-to-digital converters, a connection to Ma Bell, an Ethernet interface, and several microprocessors to tie them all together. The EtherPhone is a recent product of the **Voice** Project within CSL. The existence of the EtherPhone should make it easy to write lots of exciting experimental systems (any volunteers to write a CedarVoice interface?).

Executive: A distinguished Alto **subsystem** that provides simple commands to inspect and manipulate the file system directory, and to initiate other subsystems.

export: A **Mesa** or **Cedar** program that provides (either some or all of) the services described in an **interface** is said to export that interface.

file extension: The portion of a file name that appears following a period (possibly null). By convention, a number of extensions are reserved to indicate the type of data in the file, though not all subsystems are consistent in their use of extensions. Some commonly encountered extensions are:

~	an Alto Executive command (not really an extension)
.al:	screen font rasters in the original format
.bcd:	Mesa object program module
.bcpl:	BCPL source program module
.bfs:	an entire Alto file system gathered into a file
.boot:	program invokable by booting
.br:	BCPL object program module
.bravo:	text file containing Bravo formatting information
.cm:	Executive command file
.config:	Mesa source that describes how to combine modules
.df:	description of a system for use with DF files software
.dl:	distribution list (in a file as opposed to in Grapevine's database)
.dm:	(archaic) dump file, i.e., several logical files stored as one
.errors:	Swat error message file
.icons:	file containing displayable Icon images
.image:	executable Alto/Mesa program
.jam:	JaM interpretable code
.ks:	screen font rasters in a fancy format
.laurel:	special flavor of .bcd that can be run within Laurel
.log:	history of certain program actions
.mail:	Laurel mail file
.mail-dmsTOC:	Laurel table-of-contents file
.mesa:	Mesa source program module
.pd:	file in PD (=printer dependent) print file format, usually produced from an Interpress master
.press:	print file in Press format
.profile:	records a user's preferred values of various user interface parameters in Cedar
.run:	executable Alto program, that is, a subsystem
.sil:	SIL source file for a drawing
.st:	Smalltalk source program text
.strike:	screen font rasters in a compact and efficient but limited format
.style:	Tioga document style rules for formatting
.symbols:	Mesa symbol table (for debugging)
.syms:	BCPL symbol table (for debugging)
.tex:	TEX source text
.tfm:	font metric information
.tip:	TIP interaction description
.tioga:	Tioga text document

file name: See **file extension** and **path name** for information about the local conventions for file names.

file server: A computer on the network that provides a file storage and retrieval service. **MAXC**, **IFS**, and **Alpine** are three different types of file servers.

FileTool: A program in Cedar that allows the user to store and retrieve files from and to remote file servers. Use of the FileTool to retrieve portions of large systems to one's local file system is fraught with peril, since it is quite important that one retrieve consistent versions of things if the large system is to work, and the FileTool doesn't include any scheme of version control. Cautious programmers use **Bringover** and "**.df**" files from the beginning; everyone uses **Bringover** and "**.df**" files eventually.

FLG: (pronounced "flug") In LISP programs, a switch that customizes a program's behavior to an individual user's working habits.

fog index: A measure of prose obscurity. Units are years of education required in order to understand the measured prose.

font: An assortment of characters all of one size and style; more precisely, a mapping from a set of character code numbers to a consistent collection of graphic images.

Fonts.widths: A file containing character-width information for a large number of fonts. Used by some programs that do text formatting while producing **Press** files. The standard source is **[Indigo]<Fonts>Fonts.Widths.** Other programs appeal to separate ".tfm" files, one for each font, as their source of information about character metrics.

foo: The first meta-syntactic variable. The second is "bar". There is a tie for third between "fum" and "baz". The words "foo" and "bar" are cognates, both derived from "fubar", an acronym popular in the U.S. Navy and used by early computer programmers employed by the Navy, possibly as a technical term describing the state of a system.

Football: A two-person game in **Cedar**.

format: An attribute of a **node** in a **Tioga** document. Examples might be "long quotation", or "item in a bulletted list". The effect of the various formats is defined by the **style**.

FS: A file directory system that will emerge in **Cedar** along with the **Nucleus**; FS will replace **CIFS** and the Common Software Directory (a part of Pilot).

FTP: Acronym for File Transfer Protocol (or Program). An **Alto world** program that provides a convenient user interface to the file transfer protocol, enabling the transfer of files between co-operating computers on the **Internet**.

Garage: A nickname for the **Electronic Model Shop**, a part of **CSL**.

Gateway: A computer serving as a forwarding link between separate **Ethernets**. Gateways may also perform certain server functions, such as **name lookup**.

germ: A small part of **Pilot** that runs first; the germ handles bootstrap loading, inloading and outloading memory images during **worldswaps**, **teledebugging**, and the like.

Grapevine: The distributed electronic message transport system; it has a set of protocols all its own, and provides various server functions such as authentication. See the **blue-and-white** report titled "Grapevine: an Exercise in Distributed Computing", number CSL-82-4.

Griffin: A **Mesa** illustration program, a successor to **Draw**. Excellent on filled areas, and handles color. **Griffin** was the source of many of the pretty pictures hanging near **Lilac**.

group: (when referring to **Grapevine**) A set of **R-names**. The standard interpretation of a group is a **distribution list**. For example, CSL↑.PA is the group of all people in **CSL**, in case they all should get copies of a message. Groups can also be used for other purposes, such as access control. The R-names that constitute a group are called its *members*. In addition, a group has *friends* and *owners:* a *friend* is someone who may add or delete herself from the group, while an *owner* may add or delete anyone from the group.

Hardy: A **Tool** that provides the functionality of **Laurel**, that is, mail sending and receiving, within **Tajo**; a client of **Grapevine.**

Hickory: A reminder and calendar system based on the **Cypress** database in **Cedar.**

Hornet: Generic name for a family of 300 bpi laser-scanned printers, built on top of 2600 copiers.

Ibis: An IFS server in SDD/Palo Alto.

Icarus: (archiac) An Alto-based program for creating and editing integrated circuit designs graphically and interactively.

icon: A small image representing some concept. Used extensively in **Star** and **Cedar.**

Idun: An IFS server in SDD/Palo Alto: the home **file server** of the **Pilot** group.

ICL: Acronym for Integrated Circuit Laboratory, a part of **PARC**, located in **Building 34.**

IDL: Acronym for Integrated Design Laboratory, an incipient part of **PARC.** Once formed, IDL (to be pronounced "ideal" rather than "idle") will be located somewhere in **Building 35.**

IFS: Acronym for Interim File System. An Alto-based **file server.** Many IFS servers exist on various **Ethernets**, including **Ivy**, **Indigo**, **Ibis**, **Iris**, **Idun**, **Igor**, **Phylum**, and Erie.

IFU: Acronym for Instruction Fetch Unit; many computers have them.

Igor: An IFS server in SDD/Palo Alto: the home **file server** of the **Mesa** group. This name should be pronounced "Eye-gore", as in the movie *Young Frankenstein.*

Imager: A new implementation of the **CedarGraphics** package that is under development.

implementation module: A **Mesa** or **Cedar** module that actually provides a set of services, as opposed to an **interface module**, which simply specifies exactly what those services are to be.

Indigo: An **IFS** server in **PARC**, used by **CSL** and **ISL** to store project software files.

[Indigo]<AltoDocs>: A directory on which documentation for various Alto subsystems are stored (generally with extension **.press**).

[Indigo]<AltoFonts>: A directory on which screen fonts for the Alto are stored (extensions **.al**, **.strike**, or **.ks**). Subdirectories are used on this directory to distinguish various families of display screen fonts that have accumulated over the years.

[Indigo]<BasicDisks>: A directory on which the standard starting configurations for Alto disks are stored, as files with extension "**.bfs**". The normal way to initialize a new **Alto world** is to use **CopyDisk** to retrieve one of these disk images.

[Indigo]<Cedar>: A directory containing the **Cedar** source code and documentation.

[Indigo]<Cedar>Documentation>: A directory containing the on-line documentation for the latest version of **Cedar.**

[Indigo]<Cedar>Top>: A directory containing top level **.df** files for **components** of the current **Cedar release.**

[Indigo]<Fonts>: A directory containing various documents of printing interest, including **Fonts.widths.** You might be interested in **CloverFonts.Press**, or **AltoFontGuide.Press.**

[Indigo]<ISL>: A directory of packages released by **ISL** for use within **Cedar.** Contains mainly interactive graphics software and document formatting tools.

[Indigo]<PreCedar>: A development directory for **[Indigo]<Cedar>**; that is, this is where **components** of a new release of **Cedar** are stored while they are being developed. One of the jobs of the release process is to move things from <PreCedar> to <Cedar>.

[Indigo]<PreISL>: The analogous development directory for **[Indigo]<ISL>.**

input focus: Suppose that the user types a key, while operating in an environment that supports multiprogramming—lots of things going on at once, each in their own window.. Which program was the keystroke intended for? Different systems have different conventions on this important point. In **Tajo**, the window in which the **cursor** is currently located gets the keystroke. But in several other systems, including **Cedar**, there is a concept called the "input focus" that is passed around among the running programs; whatever program has the input focus gets the keystrokes. Left-clicking a mouse button inside of a window often has the side effect of giving that window the input focus.

Inscript: A mechanism for keeping track of user input to a program in a general way (key strokes, mouse clicks, and the like), used within **Cedar.**

install: A term applied to the Alto Operating System and a number of **subsystems** (notably **Bravo**), referring to a procedure whereby certain configuration options are established. Frequently, what is really going on is that the program being installed is salting away somewhere the current hard disk addresses of the pages of important files, so that later access to those files can avoid the tedious operations of looking up the file in a directory and chaining through disk headers to get to the right place within the file.

Intelnet: The Xerox corporate phone system, accessible by starting your dialing with the digit 8. Not to be confused with the **Internet.**

interface: A formal contract between pieces of a system describing a collection of services to be provided. A provider of these services is said to "implement the interface"; a consumer of them is called a "client of the interface".

interface module: In **Mesa** and **Cedar**, interfaces are written down as a special kind of source file, starting with the word "DEFINITIONS" instead of "PROGRAM". This explicit encoding of an interface is called an interface module.

Interlisp: A dialect of Lisp with a large library of facilities, as witnessed by Interlisp's famous 15-pound reference manual (would that Cedar were so well documented!).

Interlisp-D: An implementation of Interlisp on **D-machines**, done by a group within PARC. It provides network facilities and high-level graphics primitives. See the **blue-and-white** report entitled "Papers on Interlisp-D", number CIS-5 (SSL-80-4) Revised.

Internet: Many Ethernets connected by Gateways form an Internet.

InterPress: A print file format standard that is currently under development: a second cut at the same issues addressed by **Press** format.

InterScript: A standard format for the interchange of editable documents that is currently under development.

Iris: An IFS server in SDD/Palo Alto, which serves as the official source of released **Pilots.**

ISL Acronym for Imaging Sciences Laboratory, a part of **PARC** located on the second floor of **Building 35.**

Ivy: An IFS server in **PARC**, used by **CSL** and **ISL** mainly to store personal files.

jaggies: The annoying sharp corners visible when smooth curves are imaged on a raster device without sufficient resolution.

JaM: Acronym for John (Warnock) and Martin (Newell). An interactive language, similar to the language Forth, with a simple, stack-oriented execution model; equipped with lots for graphic operations as primitives; implemented in **Mesa.**

JaMGraphics: A component of an **ISL** release which provides **JaM** commands for all the **CedarGraphics** features. Creating JaM pictures with JaMGraphics can be very addictive.

Jedi: A **Hornet** at **PARC.**

Juniper: (archaic) An Alto-based distributed file system, built within **CSL.**

Juno: A constraint-based system for interactive graphics in **Cedar.**

junta: A technique for eliminating layers of the Alto Operating System that are not required by a particular subsystem.

Kanji: A **Dover** in **Building 34.**

Klamath: A forthcoming version of **Pilot** and other **Mesa** system software.

Lampson: A unit of speech rate. 1 Lampson is defined to be Butler's maximum sustained speed. For practical applications, the milliLampson is a more appropriate unit.

Larch: A family of specification languages.

Laurel: An Alto-based, display-oriented program that provides access to the facilities of **Grapevine** for sending and receiving mail. Succeeded by **Walnut** in the **Cedar** enviroment.

Leaf: A page-level file access protocol supported by some **IFS**'s.

level: There is a tree structure imposed upon the **nodes** that make up a **Tioga** document, and the Tioga editor can be informed to suppress the display of all nodes deeper than a certain level. In combination with **scrolling**, the levels commands in Tioga provide a convenient way to navigate in a well-structured document.

level i system: (for i ∈ [1..3]). A terminology for classifying (software) systems according to their intended user community:

1 implementers only

2 implementers and friendly users

3 naive users

Librarian: A **Tajo** program for check-in/check-out of the modules of a large **Mesa** system, used in **SDD**; also, a server for this program.

Lilac: A **Puffin** located in **CSL**, right next to **Clover**.

Lily: A program that provides teletype-style access to the mail sitting in one's **Grapevine** mailbox. Lily is designed to help out those folks who, because of travel or whatever, are unable to use their personal computers and either **Laurel**, **Hardy**, or **Walnut**. Also, a server that runs this program.

logical volume: A portion of a physical volume that is being used to support a **Pilot** environment: the **Pilot** equivalent of a **partition**.

look: An attribute of a character or string of characters in various editors, including **Bravo** and **Tioga**. "Bold" and "italic" are examples of Bravo's typographic looks, while "emphasis" and "quotation" are examples of the functional looks espoused by Tioga. The meaning of looks in Tioga, like the meaning of **formats**, is defined by the **style**.

Loops: A layer of software on top of **Interlisp** that turns it into an **object-oriented** environment tailored for building rule-based expert systems.

Lotus: Internal development name for the 1075 Xerox copier.

Lupine: The translator used to generate **RPC** stubs so that **Cedar** modules can call procedures located on remote machines.

Maggie: A tape server; that is, a machine on the **Internet** with tape drives that it will let a requesting machine use.

Magic: Acronym for Multiple Analyses of the Geometry of Integrated Circuits. A system for dealing with VLSI designs: printing them, converting them among formats, examining them with various programs.

Maintain: A, program for updating **Grapevine** registration information. There are two versions of Maintain. One, with a widely reviled teletype-style user inteface, is available within **Laurel**, or as a **Tool** in **Tajo**. It is documented in the file [Ivy]<Laurel>Maintain.Press. The other, with a nifty buttons-style interface, is available in **Cedar**. It is not yet documented.

MakeConfig: A program that reads **Mesa configs** and **bcds** and produces a collection of commands that will compile and bind the many modules of a system in the correct manner to build a consistent system.

Marion: A **Librarian** server in SDD/Palo Alto.

Markup: A dead Alto **subsystem** for editing **Press** files.

MAXC: Acronym for Multi-Access Xerox Computer (pronounced "Max"). A locally produced computer that is functionally similar to the DEC PDP-10. At one time, there were two MAXC's, named Maxc1 and Maxc2, but Maxc1 has gone away forever. From now on, "Maxc1", "Maxc2", and "Maxc" are all names for the same machine, which used to be called Maxc2.

[Maxc]<Alto>: A directory on which standard Alto (**BCPL**) programs and subsystems are stored. Only object code files (extension **.br**) and runnable files (extension **.run**) are stored here; source files and documentation are stored on **[Maxc]<AltoSource>** and **[Maxc]<AltoDocs>**, respectively.

[Maxc]<AltoDocs>: A directory on which documentation for Alto programs is stored. Common extensions are **.press** (for files directly printable by **Press** or **Spruce**), and **.tty** (plain text).

[Maxc]<AltoSource>: A directory on which source versions of standard Alto programs are stored.

[Maxc]<Forms>: (archaic) A directory containing files that are usable as templates (in **Bravo**) for various kinds of documents (e.g., memos, letters, reports).

[Maxc]<Printing>: A directory containing Alto printing and graphics programs.

[Maxc]<PrintingDocs>: A directory containing documentation related to printing and graphics facilities such as **Press** files and font file formats.

[Maxc]<SubSys>: A directory containing standard TENEX **subsystems**.

Menlo: A **Dover** located in **ISL**.

menu: A collection of text strings, buttons, or icons on a display screen generally used to represent a set of possible actions.

Mesa: A PASCAL-like, strongly typed, system programming language developed by **CSL** and **SDD**.

Mesa Development Environment: The package of software used by **SDD** to develop other software in **Mesa**; combines the **Tajo** user interface with the compiler, **binder**, packager, and other system software running on top of **Pilot**. The name "Mesa Development Environment" is often used when the plans to market this body of software running on **Dandelions** are being discussed.

MesaNetExec: A **Mesa** implementation of the **NetExec**; valuable because it knows how to load **Othello**.

MetaFont: A font-designing language built by Don Knuth at Stanford, and used to generate fonts for use with **TEX**. **Metafont** is available as MF.Sav on **Maxc**.

Microswitch keyboard: Microswitch is a company that makes keyboards. The standard **Alto** keyboard, also in use at **PARC** on **D-machines**, is made by Microswitch.

MIG: An acronym for Master Image Generator: a high-resolution laser-scanning printer, based on a photographic process. The MIG-1 can run up to 2000 bpi, while the slightly different MIG-3 runs at about 800 bpi. Also called the **Platemaker**.

Mockingbird: A music system that runs on a **Dorado** with an attached audio synthesizer and its keyboard. The goal of **Mockingbird** is to relieve the serious composer of some of the clerical burden of writing out scores for music as it being composed. For more details, see the **blue-and-white** report "Mockingbird: A Composer's Amanuensis", number CSL-83-2.

mode: A special state through which certain user interfaces must pass in order to perform certain functions. For example, in order to insert characters into a document in **Bravo**, one must type the "I" key, which invokes the "Insert" command. The effect of this command is to put Bravo into "insert mode", in which typing the "I" key has a quite different effect (to whit, it inserts an "I" into the document). One must then hit another special key, "ESC", in order to leave "insert mode". Modes are locally viewed as generally evil.

modeless: Describes a user interface that is free of **modes**. In such an interface, pressing a particular key always has essentially the same effect. **Laurel** was the first local system with an approximately modeless editor interface; the **Tioga** editing interface is very similar.

mouse: A type of pointing device with which many personal computers come equipped. The switches on the mouse are called "buttons" to distinguish them from the "keys" on the keyboard.

mouse-ahead: Analogous to typeahead, except refers to mouse clicks rather than to key strokes. Can become very confusing to non-**wizards**, as there is no analog of the backspace key for mouse clicks, that is, no way to cancel unwanted mouse clicks.

Multibus: An Intel standard specifying the physical and electrical characteristics of a bus by which various boards in small computers can communicate. Many useful boards that plug into a Multibus are available, such as Ethernet cards and disk controller cards. The **Dicentra** is a **D-machine** that uses the Multibus.

name lookup: In the context of network communications, the process of mapping a string of characters to a **network address**. Also, the protocol that defines the mechanism for performing such a mapping.

name lookup server: A computer that implements the **name lookup** protocol.

Nebula: A time server on the **Internet** that is equipped with an antenna to listen to time broadcasts made by a synchronous satellite, and hence has excellent long-term reliability. There is a display showing Nebula's opinion of the time in the same room as Clover: just the thing for setting your digital watch.

Neptune: An Alto **subsystem** providing more sophisticated manipulation of the file directory system than is available with the **Executive**. See also **DDS**.

NetExec.boot: A mini-**Executive** usable on an Alto without a spinning disk and obtainable directly over the **Ethernet** (from a **boot server**). The NetExec makes available a number of useful stand-alone programs, including **CopyDisk**, **Scavenger**, **FTP**, a number of diagnostics, and lots of neat games.

network address: A pair of numbers <network number, host number> that uniquely identifies any computer in an **Internet**.

node: A chunk of text in a **Tioga** document: each heading and paragraph in a document froms a node, and the nodes are hierarchically structured. Node-structured documents are easier to browse, using the **levels** commands in Tioga. Note: you can't have two nodes on the same line.

NS: An acronym for Network Services: the protocols for using the **Ethernet** in the **Star** world. NS packets are analogous to PUP's, and the NS protocols include analogs to such higher-level protocols as **FTP**.

Nucleus: A new virtual memory and file system base that is being built for **Cedar**, to replace portions of **Pilot**; it will emerge in Cedar 5.0.

Nursery: A large room in **CSL**, across from the Commons; so named because it was to be where new printers would be nursed to life, and also where fresh blood (summer interns and the like) would be housed. Does this mean that Bob Taylor thinks of graduate students as infants? I don't think so; course, I could be wrong... The funny windows were intended to make it convenient to hold demonstrations in the **Nursery** with some of the audience on the outside, looking in.

object-oriented: Describes a philosophy about how programs should be structured that finds its purest expression in the **Smalltalk** system. An object is a little pile of private data together with a collection of procedures by which other folks are allowed to ask the object to do something. Other folks must not play with the data directly, but instead are required to interact with the object only by calling its procedures (or, in **Smalltalk** parlance, sending it messages.) Think about complex numbers as a trivial example: A non-object-oriented programmer would probably represent a complex number as a record containing two real numbers. An object-oriented programmer would be tempted to represent a complex number as a record containing public fields and private fields. The values of the public fields would be procedures, with field names such as: AddToMe, MyXCoord, MyYCoord, NegateMe, MyMagnitude, and the like. The private fields in the standard implementation of complex number would be simply two reals, named X and Y. The advantage of the object-oriented approach is that someone else can come along later and implement a new flavor of complex number that uses polar coordinates in the private fields, and previous programs that dealt with complex numbers will not have to be changed.

OIS: An acronym for Office Information Systems: a name for a concept, a type of product, and (perhaps) a market, not a particular organization.

OPD: An acronym for Office Products Division, located mostly in Dallas. They make and sell 820's and the like; see **products**.

Orbit: A high performance Alto-based image generator designed to merge source rasters into a raster output stream for a **SLOT** printer (e.g., **Dover**). So named because it ORs bits into buffers.

OS: Acronym for Operating System. Generally used to refer to the Alto Operating System, which is stored in the file **Sys.boot**. Rarely used locally to refer to the operating system of the same name that runs on IBM 360/370 computers.

OSD: An acronym for Office Systems Division, of which **SDD** is a part; they deal with the higher end of the office market, in contrast to **OPD**.

Othello: A network-bootable **Pilot** utility, good for initializing logical volumes and the like.

page (on a disk): A unit of length: an Alto or Pilot page is 512 bytes, while an **IFS** page is 2048 bytes.

PARC: Acronym for Palo Alto Research Center.

partition: A chunk of a large local disk that is being used to emulate the largest system disk that the **Alto OS** allows. A **Dorado** has five **partitions**, while a **Dolphin** has two. **Partitions** are numbered starting at 1; the phrase "partition 0" refers to the current default partition. The current **partition** in use is determined by the contents of some registers that belong to the disk microcode. You can change these registers with the "partition.~" command available in the Executive and in the NetExec. A (14-sector) **partition** has 22,736 Alto **pages** (11.6 MBtyes). It took a little adroit shoehorning to fit two full partitions onto a Dolphin's disk: it turns out that a Shugart 4000 has just one too few cylinders to squeeze in two full partitions. So we have to ask the heads to seek off the end of the advertised disk (on the inside, it happens), and put one more cylinder in there! Ah, the joys of hardware hacking...

PasMesa: A program that more or less compiles Pascal source into **Mesa** source, and hence assists in importing Pascal programs into our environment; developed in **CSL**.

path name: The complete name of a file, including the **file server** and **directory** or subdirectory on which it is stored—everything you need to know to get the file. In the old style of writing (**Alto** and **IFS**), a **path name** consists of a machine name in square brackets followed by a directory name in angle brackets, optionally followed by one or more subdirectory names separated with right angle brackets, followed by the file name itself, as in

[Indigo]<Cedar>Documentation>BriefingBlurb.press.

Starting with **CIFS** in **Cedar**, a simple slash may be used instead of the various flavors of brackets, as in

/Indigo/Cedar/Documentation/BriefingBlurb.press.

PD files: A Printer Dependent print file format. The format and semantics of PD files are simpler than those of **Press** files. Software exists to turn **InterPress** masters into PD files, and also to print PD files on various marking engines, including **Lilac**, **Stinger**, and the **Platemaker**.

Peanut: A mail program in **Cedar** that fetches your messages into a structured **Tioga** document, rather than storing them in the **Cypress** database as does **Walnut**.

Penguin: Generic name for a type of 384 bpi laser-scan printer built on the Xerox 5400 xerographic engine, and connected to an Alto by means of an **Orbit** interface. **Penguins** have better **solid-area development** than **Dovers**, and can also print two-sided. They are normally driven with **Spruce.**

Phylum: An **IFS** in PARC.

physical volume: The name for a disk pack in **Pilot.**

PIE: Acronym for Personal Information Environment. Implemented in Smalltalk, PIE uses a description language to support the interactive development of programs, and to support the office-related tasks of document preparation, electronic mail, and database management. For more information, browse [Ivy]<PIE>.

Pilot: An operating system that runs on **D-machines**, and was produced in **SDD** for use by **Star** and future products. Using Pilot instead of the Alto OS gives you the advantages of multiprocessing and virtual memory. Pilot is the current base for **Cedar**, although parts of Pilot will soon be replaced by the **Nucleus.**

pixel: A contraction of the phrase "picture element", referred to the tiny, usually square cells out of which a raster image is built up.

plaid screen: Occurs when certain kinds of memory smashes overwrite the display bitmap area or control blocks. The term "salt & pepper" refers to a different pattern of similar origin.

Platemaker: Another name for the **MIG.**

PolyCedar: A name for the polymorphic language in the Algolic tradition that is the subject of the religious material in the **CLRM**. A possible future project in **CSL** to design and implement such a language.

Poplar: An interactive programming language system implemented in **Mesa**, an experimental system in the direction of programming by relatively inexperienced users. Useful for text manipulation applications.

Poseidon: A **Tool** that provides the functionality of **Neptune** in the **Tajo** environment.

Press: A file format used to encode documents to be transmitted to a printer. Files in this format are conventionally given the file extension **.press**. Also, a printing server program, written in **BCPL**, that can print curves and raster images as well as characters and rules.

PressEdit: A subsystem that recombines **Press** files on a page-by-page basis; it can also merge illustrations into documents, although requesting this is a somewhat arcane and delicate operation.

primary selection: A chunk of text that has been distinguished, usually by mouse clicks, as an argument to a future editing operation. The current primary selection is indicated in **Tioga** by a solid underline, or by video reversal.

PrincOps: The Xerox **Mesa** Processor Principles of Operation, essentially a description of a particular **abstract machine. D-machines** implement the **PrincOps** architecture by means of hardware and microcode, and **Pilot** was constructed to run on **PrincOps** machines.

print server: A computer that provides printing services, usually for files formatted in a particular way. The term also refers to the software that converts such files into a representation that can be processed by a specific printer hardware interface. **Spruce** and **Press** are examples of print server programs that accept the **.press** print file format.

proc: (or PROC:) An abbreviated form of the common and important word "procedure".

products: The following is a list of the most commonly encountered Xerox product numbers and their distinguishing characteristics:

800	typewriter-based, word-processing terminal
820	personal computer product
860	display-based, word-processing terminal

1000	new series of copiers being advertised with Marathon theme
1100	a Dolphin, sold outside to run Smalltalk and Interlisp
1108	a Dandelion, sold outside to run Interlisp
1132	a Dorado, sold outside to run Smalltalk and Interlisp
2600	desktop copier
3100	3 sec/page copier, good solid black-area development
4500	1 sec/page copier, 2-sided copying
5400	1 sec/page copier, good resolution
5700	1 sec/page laser-scan printer
6500	20 sec/page copier, color copying
7000	1 sec/page copier
8000's	the parts of **Star** have numbers in this range
9200	offset-quality, .5 sec/page copier
9700	offset-quality, .5 sec/page, laser-scan printer

PSD: Acronym for Printing Systems Division, a part of Xerox.

public interface: An interface that offers to provide services to all comers. Private interfaces, in contrast, specify the services that various modules in a single program will supply to each other.

Puffin: Generic name for a type of 384 bpi laser-scan color printer built on the Xerox 6500 xerographic engine, and normally driven by **Press**.

PUP: Acronym for PARC Universal Packet. The structure used to transmit blocks of information (packets) on the **Ethernet**. Also, one such unit of information: a datagram. Bob Metcalfe once remarked that this name was chosen since all prior PARC communication protocols were "real dogs". See the **blue-and-white** report entitled "Pup: An Internetwork Architecture", number CSL-79-10.

Quake: A **Dover** on the first floor of **Building 35**.

Quantum: Brand name of certain disk drives.

Quoth: A **Raven** in **ISL** (as in "Quoth the raven . . .").

R-name: A complete name from **Grapevine**'s point of view: **R-names** have two parts, a prefix and a registry, separated by a dot as in "Anderson.PA". **R-names** that designate distribution lists have prefixes that end in an up-arrow, as in "CSL↑.PA".

Raven: A 300 bpi laser-scan printer based on the 8044, with good **solid-area development**. Upgraded in **ISL** to 384 bpi and used as a **Press** printer.

registry: A concept used by **Grapevine** to partition the space of names. "PA" and "WBST" are examples of registries.

release: A consistent set of versions of all of the files in a large software system. **Cedar** releases occur whenever major enhancements in functionality become available or when sufficently numerous or important errors (see **show-stopper**) have been corrected.

release master: The person in charge of coordinating a **Cedar** release, with the help of special software (the ReleaseTool) based on **DF files**.

religious: Used locally to refer to a debate about which people have strong feelings, but for which there is no easy technical resolution; when discussing religious issues, positions are advanced based on belief rather than on understanding. For example, the question of whether or not **windows** in a user interface should be allowed to overlap and partially obscure each other, as pieces of paper do in the real world, is often the subject of religious debate. More experience in user interface design, or sufficient advances in the cognitive psychology of user interfaces, may someday make this question less religious.

Rem.cm: A file used by the Alto **Executive** to store commands to be interpreted after the current one has completed. See **Com.cm.**

replay: Refers to a Bravo facility that permits recovery after a crash. See **BravoBug.**

Reticle Generator: A version of the **MIG** that prints directly on masks for integrated circuits.

reverse engineering: Designing something by taking measurements from an existing sample that someone else designed.

Rigging: A **component** of **Cedar** that implements the various flavors of strings, including **Ropes.**

RockAndRoll: Another **Raven** printer in **ISL.**

Rockhopper: A **Penguin** in the **Bayhill** building.

RollBack: The way to return to a clean **Cedar** world saved by a **checkpoint.**

Rope: An immutable string of characters (a rope is a "thick" string). Ropes are the standard way to pass strings around within **Cedar**; other types of strings, including REF TEXT and REF READONLY TEXT, are available for places where performance is a big issue.

RPC: Acronym for Remote Procedure Call, a technique for calling a procedure from one machine to be executed in another machine over a network. Also, a package of software supporting Remote Procedure Calls within **Cedar.** RPC is the standard way for Cedar programs to communicate over the network: **Tank, Football, Alpine,** and **Etherphones** all communicate by means of RPC. For more details about the concept of RPC, as well as fascinating references to life in the South Pacific, read Bruce Nelson's thesis, which is available as the **blue-and-white** number CSL-81-9.

Rubicon: The release of **Pilot** upon which **Cedar** is currently based.

rule: A printing term describing a rectangle whose sides are parallel to the coordinate axes; usually thin enough in one dimension or the other to be thought of as a (horizontal or vertical) line.

Scavenger.boot: An Alto program available through the **NetExec** that checks for damaged file structures in a **BFS** and tries to repair them.

SCG: Acronym for Software Concepts Group, a part of PARC. The builders of **Smalltalk.**

scroll: Refers to a method of repositioning text on a display as though as though one were moving a **window** over a long, continuous sheet of paper.

scroll bar: A **bar,** usually located along the left edge of a **window,** with the property that **clicking** in this bar causes **scrolling** (or perhaps **thumbing**) to happen.

SDD: Acronym for System Development Division: the technical (as opposed to marketing) portion of **OSD.**

secondary selections: A chunk of text distinguished, usually by mouse clicks, as the second argument to a future editing operation. The current secondary selection is indicated in **Tioga** by a gray underline, or by a gray background.

Semillon: A Grapevine server in **Building 35.**

server: A computer dedicated to performing some collection of service functions for the communal good (e.g., a **print server**).

seven-wire interface: Yes, Virginia, hardware people use the concept of **interface** as well as software folk. The seven-wire interface describes how the microprocessor located in the terminal of a **D-machine** (in the base of the CRT, to be specific) communicates with the parent computer.

show-stopper: A **bug** serious enough to prevent further progress.

Shugart: A manufacturer of disk drives.

Sierra: A recent release of the **Mesa Development Environment**, based upon **Trinity** Pilot.

signal: A mechanism for handling exceptional conditions that arise in **Mesa** or **Cedar** programs. See **catch phrase**.

SIL: Acronym for Simple ILlustrator. An illustrator program used for logic design and drawing in general. A weird but efficient user interface: solid performance.

SIS: Acronym for Scientific Information Systems; the name of that part of **EOS** that is still a part of Xerox.

SLOT: Acronym for Scanning Laser Output Transducer.

Smalltalk: An integrated programming system based on **object-oriented** style and message passing, invented and developed by **SCG**. Described in great detail in a recently issued book(!).

SModel: A program that stores files back to remote file servers from one's local disk; SModel reads ".df" files in order to figure out what files have been changed, and which of these should be stored, and where in the great wide electronic world to store them. Use of SModel (confusing as it may be at the outset) is to be recommended over use of either **FTP** (in the **Alto world**) or the **FileTool** (in **Cedar**), since the version control and system-description features of ".df" files are very valuable.

solid-area development: The ability of a printer to produce large areas of black. Requests for large black areas on printers like **Dovers**, which don't have this ability, will result in a fringe of dark gray around a sea of light gray.

SophtSpheroid: A small, round, white object usually found on diamonds. Consider joining a Xerox softball team for more information on this indelicate topic.

Spruce: A program that takes **Press** files consisting of text and **rules**, converts them to a form acceptable by an **Orbit** interface, and prints them. A print server.

Spy: A program to investigate another program's performance when running in **Cedar**.

Squirrel: A personal database program based on the **Cypress** database in **Cedar**.

Star: An **OIS** product of Xerox, developed within **SDD**. Also referred to by various product numbers in the 8000's. The primary professional workstation of **Star** is the 8010. The 8000 architecture was created in **CSL**.

Stinger: A **Hornet** located in ISL, running **Press**.

STP: The **Pilot interface** to the **FTP** file transfer protocol.

style: A collection of little programs in a language very like **JaM** that define the meanings of the various **looks** and **formats** of the text in a formatted **Tioga** document. Different style rules exist for how things should look on the screen and for how they should look when printed on paper (implemented by the **TSetter**).

subdirectory: File directories on an IFS can be divided into a hierarchical collection of subdirectories. The subdirectory names are listed from the root of the tree down to the leaves, and are separated by the single character ">" (see path

subsystem: A program running under a specific operating system. Normally used to refer to Alto programs that run under the Alto **OS**, but also used to refer to PDP-10 programs that run under TENEX.

Swat: A debugger used primarily for **BCPL** programs. Also, the key used in conjunction with the "control" or "shift" keys to invoke this debugger, as well as various other debuggers. The Swat key is the lowest of the three unmarked keys at the right edge of the keyboard. Used as a verb to refer to the act of striking these keys or entering the debugger.

Swatee: A file used by debugging programs (both **Swat** and the Alto/Mesa debugger) to hold the core image of the program being debugged. Also used as a scratch file by many Alto subsystems. Not to be deleted under any circumstances.

Sys.boot: An Alto disk file containing the executable representation of the Alto Operating System.

SysDir: The Alto file directory. Roughly speaking, this file contains the mapping from file names to starting disk locations.

SysFont.al: An Alto screen font used by the **Executive** and (generally) as a default by other programs. The safest way to change your SysFont is with the Delete.~ and Copy.~ commands of the Alto Executive. Simply FTP'ing a new font on top of SysFont will cause exotic behavior during the CounterJunta when FTP is finished.

system models: A part of the **Cedar** project, aiming at giving programmers help is describing the structure of large systems: getting consistent versions of files, replacing single modules within a running system, and recompiling and rebinding just what has been changed, all in the right order.

Tajo: The user interface portion of the **Mesa Development Environment**. Each facility in the **Tajo** environment is called a **Tool**, and **Tajo** itself is sometimes called the **Tools Environment**.

Tank: An *n*-player video arcade game in **Cedar**. Get a tank game going and then close the tank **viewer** and check out the wonderful **icon** that results.

teledebug: Debugging one machine from another other the **Internet**. The prefix "tele-" is used in general for doing things remotely.

Telnet: A **PUP**-based protocol used to establish full-duplex, teletype-like communication with a remote computer. (The term is borrowed from a similar protocol used on the Arpa network.) **Chat** speaks this protocol.

Tenex: An operating system for the DEC PDP-10 computer, which also runs on **MAXC**.

TEX: A document compiler written by Don Knuth at Stanford; there are one and a half implementations of **TEX** at **PARC**: one in Sail that runs on **Maxc**, the half in **Cedar** (waiting on progress on the **Imager**). **TEX** can handle mathematical formulas, but doesn't let you see anything like what you get.

Thyme: An electrical-level dircuit simulator, used for evaluating the correctness and performance of small pieces of the designs of integrated circuits.

thumbing: A technique of positioning a file (usually text) to an arbitrary position for viewing on a display. The name is intended to suggest the "thumb-index" with which some dictionaries are equipped, which performs somewhat the same function: gets you to roughly the right place quickly.

TIC: Acronym for Technical Information Center; the fancy name for what is more generally known as the PARC library.

Tioga: The document editor in **Cedar**, which was built by folk in **ISL**. Tioga formatting uses the concepts of **level**, **node**, **look**, **format**, and **style**; for more details, read TiogaDoc.tioga. Documents formatted with Tioga can be printed with the **TSetter**.

TiogaDoc.tioga: Documentation for the Tioga editor. At one point, the official home of this file was the directory [Indigo]<Tioga>Documentation>.

TIP: A system for interpreting keyboard and mouse actions and turning them into sequences of commands. You may customize your Tioga user interface by layering your own TIP table on top of the standard Tioga TIP table.

Tool: A facility available in the **Tajo** environment, or the program that makes that facility available. For example, one speaks of the "File Tool", which can perform file transfers for you.

Tools Environment: Former name for **Tajo**.

transaction: A collection of reads and writes of shared data that is guaranteed to be atomic: either all of the writes happen (the **transaction** *commits*) or none of them do (it *aborts*). Furthermore, the reads will see consistent data in that either all of the writes made by some other **transaction** will be visible, or none of them will.

Trident: The brand name of a type of disk drive that is quite common around here. There are T-80's (that is, 80MByte Trident drives) and T-300's. Tridents are manufactured by Century Data Systems, a subsidiary of Xerox.

Trinity: The version of **Pilot** and other **Mesa** system software between **Rubicon** (the current base of **Cedar**) and **Klamath.**

TSetter: The typesetting program for **Tioga** documents; converts foo.**tioga** into foo.**press**, and optionally sends the latter to your favorite **print server**.

typeahead: An ability to type characters to a program before that program has asked for them. Useful for wizards; essential when using slow machines. See also **mouse-ahead.**

typescript: A file used to back-up information (usually text) appearing in a region of the display.

Twinkle: A **Gateway** in **Building 35** of PARC.

uncaught signal: An exceptional condition (perhaps an error indication) that no current program other than the **Mesa** or **Cedar** debugger has expressed a willingness to deal with. The debugger is willing to deal with anything, of course: it deals with these exceptional events by halting the offending process and then informing the user. In the language of the **CLRM**, an uncaught signal should be thought of as an invocation of a dynamically bound procedure that turns out not to have been bound at all; see **catch phrase.**

user: A person (rather than a program) who avails herself of the services of some program or system. At the moment, the author is a **user** of **Tioga**. See **client.**

user.cm: A file in the **Alto world** containing a number of logically distinct sections that each define certain configuration parameters (e.g., the location of a preferred **print server** for a particular file format). Programs that interpret such parameters are often organized to read user.cm only at **installation** time (e.g:, **Bravo**).

UserExec: The command interpreter for **Cedar.**

viewer: The name for a window in the **Viewers** window package.

ViewerDoc.tioga: Documentation for the **Viewers** window package. You might try looking for this file on the directory [Indigo]<Cedar>Documentation>.

Viewers: A screen management and window package for **Cedar** providing **buttons**, **menus**, and **windows.**

ViewRec: A software package in **Cedar** that produces convenient user interfaces to fairly arbitrary programs automatically.

Viking: A **Dover** on the first floor of **Building 35.**

VLSI: Acronym for Very Large Scale Integration of electronic circuits on chips.

VM: Acronym for Virtual Memory.

Voice: A small but mighty project in CSL to tame the telephone and otherwise make full use of voice communications in our personal information systems. The Voice Project recently produced the **EtherPhone.**

Walnut: A mail system for **Cedar.** Walnut uses the **Cypress** database to store and organize messages, and it calls upon **Grapevine** to transport them.

Watch: A **Cedar** performance monitoring tool displaying computing activity.

WaterLily: A **Mesa** program that does source compares: compares two text files and reports the differences. Available in Alto/**Mesa**, **Tajo**, and **Cedar.**

wedged: Describes the state of a program when there is no response to input from either the keyboard or the mouse. May affect the whole system (*my system is wedged*) or just some part thereof.

weekly meeting: The (boring) name of ISL's weekly meeting, held on Tuesdays starting at 11:00 am. See also **Dealer.**

whiteboards: A package in **Cedar** for arranging and accessing information graphically.

Winchester: Originally, this was the name of a project within IBM. But the name leaked out, and it is now used industry-wide to refer to a particular rigid disk technology. In a Winchester disk drive, the heads and platters come all hermetically sealed; that is, Winchester drives do not use removable disk packs.

window: A display region, usually rectangular, used to view (a portion of) an image that generally exceeds the bounds of the region.

wizard: One who knows a programming system inside and out.

Wonder: A **Dover** on the third floor of **Building 35.**

world-swap: The process of writing out the complete state of a machine's processer and memory onto a disk file, and of swapping in a different state. Some debuggers work by means of **world-swaps**, which swap between the debugger and the program being debugged. Note that, the more memory you have, the slower a world-swap will be.

XGP: (archaic) Acronym for Xerox Graphics Printer. An obsolete, CRT scanned, 200 bpi, continuous paper, xerographic printer.

XM: Acronym for Extended Memory: an option on Alto II's that allows the memory size to be increased from one to four **banks**.

Yoda: A **Dover** in **Building 35.**

Zinfandel: An Alto mail server that is part of the **Grapevine** distributed transport mechanism.

The Tioga Editor

Release as [Indigo]<Cedar>Documentation>TiogaDoc.Tioga and TiogaDoc.Press
Last edited by Bill Paxton on March 14, 1983
by Rick Beach on June 7, 1983 1:37 pm

Abstract: This is an overview of Tioga as available for Cedar 4.2 in June, 1983. Tioga is a system to help you prepare documents. Its two main components are an editor and a typesetter. The editor lets you create the text of a document. The typesetter composes the text into pages for printing. Tioga is already capable of dealing with simple technical papers and is also well suited to more mundane tasks such as writing programs and memos. In future versions, it will be suitable for complex technical documents and books and will support tables, math formulas, and figures containing synthetic graphics or scanned images.

If you are looking at this document on-line, you might want to use the level-clipping function to see the overall structure rather than simply plowing straight through. Hit the "Levels" button in the top menu, then hit "FirstLevelOnly" in the new menu that appears. That will show you the major section headings. Hit "MoreLevels" to see the subsections, or hit "AllLevels" to read the details.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Overview

Some editors represent a document as a list of paragraphs. In Tioga, a document is a tree structure rather than a list so that you can explicitly represent its hierarchical structure. In discussing the document tree we use Computer Science terminology in which a branch is recursively defined to be a node having zero or more children branches. The root node of the document tree is not displayed — although it can be modified by a few special commands — so the document basically appears to be a list of top-level branches.

Each node in the tree contains text. The characters of the text can have *looks* which control various aspects of their appearance such as font and size. Appearance is also influenced by the *format* of the node which determines things such as vertical and horizontal spacing. The document contains names of looks and formats, but not the specific interpretation of them. The interpretations are instead collected in a *style* which can be shared by many documents. For example, in the style for this document there are definitions of formats for titles, headings, and standard paragraphs, and there are definitions of looks for emphasis and for small caps. Rather than copying the specific details for the formats and looks, the document refers to them by name so it is easy to change the definitions in the style and modify the appearance uniformly throughout the document. The details of the style language will be described in a forthcoming memo.

User Categories

The Tioga user-interface is "layered" so that beginning users can protect themselves from the confusion that results from mistakenly giving a command. In ways described below, you can tell the system that you are either a beginner, an intermediate, or an advanced user. If you do nothing, the default is beginner.

As far as the Tioga user-interface is concerned, the user category determines which keyboard commands are currently enabled. As a beginner, you get the commands that use the special keys at the left and right of the keyboard, plus CTRL-A and CTRL-W for backspace character and word, respectively. As an intermediate user, you add a large number of commands that use print keys in combination with the various shift keys. As an advanced user, you add the keyboard commands for manipulating the document tree structure. Any category of user can get at any of the commands by using the Edit Tool. The user category mechanism is meant to let you protect yourself, not to limit you. You are free to change your own category at any time you feel like it. For example, to declare yourself to be an intermediate user, edit your user profile to say "UserCategory: intermediate".

Input Devices

Mouse

The mouse has three buttons named LEFT, MIDDLE, and RIGHT corresponding to their physical layout.

Keyboard

Used for commands as well as text input. Here are the names for the special keys.

LOOK (looks shift).	top right blank key.
NEXT.	middle right blank key.
REPEAT.	key labelled ESC.

DELETE.	key labelled DEL.
LOAD FILE.	key labelled LF.

The bottom right blank key is an alternative CTRL key. You can use either CTRL key and either SHIFT key interchangeably.

Scrolling and Thumbing

To move a viewer to look at a different part of the document, move the cursor into the left margin until it becomes a double arrow pointing both up and down. The part of the margin that becomes gray is called the "scroll bar". The dark gray part shows the relative size and location of the currently visible portion of the document. Left-click to move the text adjacent to the arrow to the top of the viewer. Right-click to move the text from the top of the viewer down to the arrow. These operations are called "scrolling" the document.

If you hold down MIDDLE in the scroll bar, the cursor becomes a right-pointing arrow. If you move the cursor to x% down from the top of the viewer and let up, the viewer will change to start about x% of the way through the document. This operation is called "thumbing".

Changing levels when you scroll

In a document such as this one that uses levels to reflect its logical structure, it is often useful to start with a view of the first level only and then progressively scroll the document up and show more levels as you zoom in on a particular topic. This process is slowed when you have to move the mouse from the scroll bar up to the levels menu and back, so we have made it possible to scroll and change levels in a single operation. This works by using the (left) SHIFT and CTRL keys as modifiers of the left-click that causes scrolling up.

Scroll up and show more levels – shift-left-click (hold the left SHIFT down when left-clicking to scroll)

Scroll up and show all levels – control-left-click (hold CTRL down when left-clicking to scroll)

Scroll up and show minimum levels – shift-control-left-click (hold both the left SHIFT and CTRL down when left-clicking to scroll).

Selections

The details of making selections will be covered later on. For now, you simply need to know that there is a single primary selection on the screen. The viewer containing the selection is referred to as the "selected viewer". Many of the following commands deal with the selection or the selected viewer.

Menus

Top level menu

Save – writes a new version of the file

Old version of the file is renamed to have a "$" at the end of the extension so that you can retrieve it if necessary.

If the version of the file on the disk has a more recent create date than the document you are about to save, the system will tell you. This is to warn you about situations in which you load a file and while it is in a viewer transfer a more recent version to your disk.

Get – loads the file named by the selection into a viewer

If the click is done with the left button, the file is loaded into the "clicked" viewer and replaces the previous contents. If it is done with the middle button, a new viewer is created below the clicked one. Finally, if it is done with the right button, the clicked viewer is closed and a new viewer appears in its place.

If you use a remote file name, such as /Cedar/Documentation/TiogaDoc.Tioga, the system will fetch the file and make the viewer "ReadOnly". This lets you browse remote files and copy information from them. In future releases, Tioga will support editing of remote files.

File Extensions

If the selected name does not include an explicit extension (i.e., there is no period in the selected text), Tioga will search for the file using a set of standard extensions. The default extensions are mesa, tioga, df, cm, config, and style. You can specify your own list of extensions with a user profile entry for "SourceFileExtensions".

If the selected name is of the form <alpha>.<beta> and such a file exists, it is opened. Otherwise, if <beta> is one of the standard set of extensions, you will be informed that the file doesn't exist. However, if <beta> is not a standard extension, the system tries to open the file as if you had simply selected <alpha>. If this succeeds, it searches in the file for a definition of <beta>. This convention is intended for use with programs that have many instances of <Interface>.<Item> in which <Interface>.mesa is a file containing a definition for <Item>.

GetImpl – like *Get* but loads the file that implements the selected interface name

If the selected name does not include an explicit extension (i.e., there is no period in the selected text), Tioga will use a set of standard implementation extensions. Currently, mesa is the only default implementation extension. You can specify your own list of implementation extensions with a user profile entry for "ImplFileExtensions".

If the selected name is of the form <Interface>.<Item> and an implementation for the item is currently loaded, the system will find the name of the file holding the implementation (our thanks to the Cedar runtime model for providing this information). Otherwise, the system tries to open the file as if you had simply selected <Interface>, and, if this succeeds, searches in the file for a definition of <Item>.

PrevFile – like *Get* but reloads the file that was previously in this viewer

Reset – discards edits by reloading the filed version of the document

Store – like *Save* but writes to the file named by the current selection

Clear – creates an empty viewer

If the click is done with the left button, the "clicked" viewer is cleared. If it is done with the middle button, a new empty viewer is created below the clicked one. Finally, if it is done with the right button, the clicked viewer is closed and a new empty viewer appears in its place.

The empty viewer will say "No Name" at the top in the place that would normally hold the file name. Naturally the most common thing to do with a "No Name" viewer is to load a file. If you type a file name into a "No Name" viewer and then hit LF, it is as if you selected the name and hit Get but no confirmation is required. CTRL-LF provides a similar function for GetImpl.

A Comment Regarding "Unsaved" Documents

It is not uncommon to forget to save the contents of a viewer before destroying it or loading something else into it. However this is not a disaster since Tioga holds onto "unsaved" documents so you can reload them with edits preserved. The number of such documents that Tioga will remember is set by a profile entry (UnsavedDocumentsCacheSize); the default is four. Whenever there are unsaved documents, Tioga creates a special viewer listing their names.

Unsaved documents are put into the cache by Destroy, Clear, Get, GetImpl, and PrevFile. They are *not* put in by Reset. If the cache is already full when a new entry arrives, the oldest entry is discarded. Whenever a file is needed for Get, GetImpl, or PrevFile, the cache is checked to see if an unsaved version is available. "No Name" documents are not put into the cache.

Time – inserts the current time at the caret

Split – creates a new viewer looking at the same document

The selection is highlighted in one viewer only, however edits will be reflected in all viewers for the document. Note that the split viewers can be independently closed, opened, or moved on the screen.

Places, Levels – show/remove submenus

Places

The Places menu contains commands that cause the viewer to begin displaying at a new place in the document. The first three of the commands search for instances of the current selection. For these commands, the button used in clicking the menu item determines how the search is carried out – left-click to search towards the end of the document, right-click to search towards the start of the document, or middle-click to search first towards the end and, if that fails, then from the start of the document. If the selection is visible in the viewer, the search starts there. Otherwise, it starts from the top of the viewer.

Find

Find another instance of the selected text.

In this command and the following two, capitalization matters in the search (e.g., hitting Find with "the" selected will not select "The").

Word

Find an instance of the selected text that is a "word" – i.e., doesn't have adjacent letters or digits.

Def

Find a "definition" of the selected text – i.e., an instance of the selected text that is immediately followed by a colon and doesn't have an immediately prior alphanumeric.

Position

This is useful with compiler error messages that give locations as character counts. The command scrolls to the selected character number and then selects it – e.g., if "183" is selected, scroll to and select character number 183 in the document.

Normalize

If the document in this viewer does not contain the selection, scrolls to the start of the document. Otherwise scrolls to make the selection visible – left-click to scroll to the start of the selection, right-click to scroll to the end, or middle-click to scroll to the caret.

PrevPlace

Go back to the place that was previously visible in this viewer discounting manual scrolling that may have taken place since.

Reselect

Restore the most recent selection in this viewer and scroll to it.

Levels

These commands let you control how deep the display goes in the document tree structure.

FirstLevelOnly – show only the top level nodes

MoreLevels – show one more level than currently

FewerLevels – show one fewer level than currently

AllLevels – show all levels of the tree

Selections

Primary selections

The primary selection is the one that's around most of the time and is the usual site for edits. It is displayed with a solid underline or with video reverse. Make a primary selection with the mouse in ways described below. During certain editing operations such as Copy or Move there is a "secondary" selection which is displayed as a gray underline or background.

Insertion point

The insertion point goes with primary selection. It is shown by a blinking "caret" at one end or the other of the selection – the end closer to the cursor when the selection was made or the one most recently extended.

Making selections

The selection hierarchy

The selection hierarchy consists of the following levels at which a selection can exist: point, character, word, node, branch, and document.

Point selection

The primary selection has a blinking caret at one end. Some operations, such as delete and type-in, reduce the selection to just the caret. This is called a point selection.

Character selection

Left-click with the cursor over the desired character. You can hold the left button down and move to the correct place before letting the button up.

You can cancel the new selection by hitting DEL while the mouse button is still down. The system will restore the previous selection.

Word selection

A "word" is defined as a sequence of letters and digits or a sequence of identical characters that are not letters or digits. Use the MIDDLE mouse button to make word selections. As with character selection, you can hold MIDDLE down and move to the correct word before letting up.

Node selection

Left-double-click to select a node.

Branch selection

A branch is a node and any children branches it might have. Middle-double-click to select a branch.

Document selection

CTRL-D extends the selection to include the entire document. "D" stands for for "Document".

Selection extension

Extend an existing selection by pointing at a new endpoint and right-clicking. The system will extend the end of the selection closer to the cursor when RIGHT goes down. You can hold RIGHT down and move the endpoint to a new position.

If you just right-click, the selection is extended at the same level in the selection hierarchy. For example, a selection at the word level will be extended a word at a time. Double-right-click to extend at a lower level in selection hierarchy. Triple-right-click to extend at a higher level in selection hierarchy. Thus, if you have a node-level selection and wish to extend it to a word position within a node, double-right-click to reduce the level to words and then do the extension. If necessary, you can then double-right-click again to reduce to character level.

Editing by making selections

Delete selections

A *delete selection* is deleted as soon as the selection is completed. It is shown as video reverse. Hold down CTRL to make a delete selection. The selection is complete when you let up on both the mouse button and the CTRL key.

Pending-delete selections

"Pending-delete" selections are automatically deleted by subsequent insertions. They are shown with video reverse rather than solid underline. Whenever you extend a selection it is automatically made pending-delete. Notice that you don't have to actually change the selection when you "extend" it. For example, you can middle-click to select a word and then immediately right-click to make it pending-delete. Or you can combine these actions by "rolling" from the LEFT or MIDDLE mouse button to the RIGHT button to make a char or word selection pending-delete. For example, the sequence MIDDLE-down, RIGHT-down, MIDDLE-up, RIGHT-up will produce a pending-delete word selection.

Copy and Move

Source selections are made with the SHIFT key held down and are shown with a gray underline.

Copy source to primary

If you hold down the SHIFT key and select, the source selection will be copied. If the primary selection is currently pending-delete, it will be replaced by the copy of the source. Otherwise, the copy will be inserted at the caret.

Move source to primary

If you hold down the SHIFT key and the CTRL key, the source selection will be moved. As before, if the primary selection is pending-delete, it will be replaced by the source. Otherwise, the source will be moved to the caret.

Destination selections for Copy and Move

The operations described above work by copying or moving a source selection to the primary selection. However, often the primary selection is itself the thing you want to copy or move. The following two commands take care of these situations.

Copy primary

To copy the primary selection, hit CTRL-S, and with the control key still held down, select a destination. The copy takes place as soon as you let the keys up. The primary selection is made not-pending-delete as soon as you hit CTRL-S to indicate that it will be copied rather than moved.

Move primary

To move the primary selection, hit CTRL-Z, and with the control key still held down, select a destination. The move takes place as soon as you let the keys up. The primary selection is made pending-delete as soon as you hit CTRL-Z to indicate that it will be moved rather than copied.

Transpose selections

We now have covered commands to copy or move either the primary or the source selections. A final option is to transpose the primary and the source. To do this, hit CTRL-X, and with the control key still held down, select a source. The transpose takes place as soon as you let the keys and mouse buttons up.

Miscellaneous

Cancelling a selection

Hit DEL before finishing the selection and the previous selection will be restored. This works during either primary, source, destination, or transpose selections.

Placeholders

A "placeholder" is all the text between a matching pair of placeholder brackets, ▶ ◀. Hit NEXT to find and select the next placeholder beyond the current selection. Hit SHIFT-NEXT to find the previous one. Recall that NEXT is the blank key to the right of RETURN.

If there isn't another placeholder, NEXT will move the selection to the end of the document. If the selection is already at the end, NEXT will try to find the next nested text viewer. Similarly, SHIFT-NEXT will move the selection to the start of the document if it doesn't find a previous placeholder and will try to find the previous nested text viewer if it is already at the start. This allows you to use NEXT both when filling in placeholders within a document and when filling in text viewers in tools.

Select visible – expand selection to blanks

CTRL-V expands the selection to include the "visible" characters on the left and right ends.

This is useful for selecting things like full file names. For example, in the following you could make a character selection anywhere in the name and then extend the selection with CTRL-V.

Filed on: [Indigo]<Tioga>Documentation>TiogaDoc.Tioga

Select matching brackets

CTRL-] extends the selection to the left and right to find a matching pair of [..]'s. Similarly for CTRL-}, CTRL-), and CTRL->. Note that you hit CTRL with the right bracket to extend the selection. In addition, you can hit CTRL with the left bracket to insert matching brackets around the selection.

Fine point: Unfortunately, our keyboards don't have both left and right quote keys. However most of the fonts, including TimesRoman and Helvetica, do provide a left and right single quote. The keyboard key inserts a right single quote (code 047); the left single quote (code 140) can be inserted using the MakeOctalCharacter command in the Edit Tool or with the Insert Matching Single Quotes command (CTRL-'). The Edit Tool also has a command which extends the selection to find a matching pair of left and right single quotes. The situation for double quotes is even less uniform. Some fonts, such as Classic, have a left double quote (code 264) in addition to a right double quote (code 042). However, most fonts have only the 042 double quote, so the Tioga commands for inserting and matching double quotes use that code exclusively.

Editing

Text input

Typed-in characters are inserted at the caret.

To insert the current time use CTRL-T – "T" for *Time.*

When you insert a carriage return by hitting RETURN, the system will automatically copy the blank characters (tabs and spaces) from the start of the previous line. To suppress this, type SHIFT-RETURN and only the carriage return will be inserted.

To insert control characters or characters with a specific octal code, use CTRL-K or CTRL-O.. The former will change the character before the caret to a control character, while the latter will convert the three digits before the caret to the corresponding octal character. The inverse operations are also available as CTRL-SHIFT-K and CTRL-SHIFT-O.

Abbreviation Expansion

CTRL-E – "E" for *Expand abbreviation*

When you hit CTRL-E, the caret is moved to the right of the selection if necessary and the keyname to the left of the caret is then replaced by the expansion text (according to the definition which is linked to the style in a manner described below). If the keyname had looks, they are added to the expansion. If the keyname was all caps, the expansion is made all caps too. If the keyname had an initial cap, the first character of the expansion is made uppercase (useful at the start of sentences, for example). If the definition node has a non-null format, the format of the caret node is changed to be the same as the definition node. If the expansion contains a placeholder, the first placeholder is selected. Otherwise the entire expansion is selected.

Definitions for abbreviations come from Tioga documents which are automatically read by the system when needed. The name of the appropriate abbreviations file is determined by the style that is in effect at the caret when the expansion takes place. For example, if the style is "Report", the abbreviations will come from the file "Report.Abbreviations". You can override this by explicitly naming the abbreviations file along with the keyname. For example, "Mesa.proc" will expand the abbreviation for "proc" from the file "Mesa.Abbreviations" independent of the style in effect at the caret. (Fine point: Since the system interprets a period before the keyname to mean that you're specifying a particular abbreviations file, you cannot type a vanilla abbreviation after a period.)

Each definition in an abbreviations file consists of a separate branch. The top node of the branch holds the keyname followed by an equals sign and then the text expansion. The rest of the branch, if any, is copied after the caret node as part of expanding the abbreviation. Any text following the keyname is moved to the end of the last child node in the branch. The definition may also include a list of operations to be performed after the expansion has been inserted. These operations have the same format as those in EditTool and are placed in parentheses after the keyname and before the equals sign.

Delete Character or Word

BackSpace: CTRL-A, CTRL-H, or BS. Deletes the character to the left of the caret.

Fine point: If the caret is at the start of a node, this does a Join command (q.v.).

BackWord: CTRL-W, or CTRL-BS. Deletes the word to the left of the caret.

DeleteNextChar: CTRL-SHIFT-A, CTRL-SHIFT-H, or SHIFT-BS. Deletes the character to the right of the caret.

DeleteNextWord: CTRL-SHIFT-W, or CTRL-SHIFT-BS. Deletes the word to the right of the caret.

Delete

As mentioned above, you can delete something by selecting it with CTRL held down. In addition, you can delete the current selection by hitting DEL.

Paste

Paste: CTRL-P. The most recently deleted text is copied to the caret. You can also use the Edit Tool to save the current selection to be pasted later.

Copy, Move, Replace, and Transpose

These operations are all carried out by making selections. They are described in detail in the previous section.

Insert matching brackets

CTRL-[adds a matching pair of [..]'s to the ends of the selection. Similar CTRL commands exist for {, (, <, -, ', and ". CTRL-B inserts matching placeholder brackets ▶◀.

Note: CTRL-' inserts a left single quote (code 140) and a right single quote (code 047); CTRL-" inserts the same character (code 042) at each end of the selection.

Case

All lower: CTRL-C. Makes the selection all lower case.

All caps: CTRL-SHIFT-C. Makes the selection all upper case.

Initial caps: CTRL-double C. Capitalizes each word in the selection.

First cap: CTRL-SHIFT-double C. Capitalizes the first word of the selection

Repeat

Hitting ESC will repeat the most recent non-empty edit sequence starting with the current selection. Edit sequences are separated by user-made selections. For example, if you select a word, delete it, type a new one, and then select something else, the edit sequence is delete followed by text entry. If you hit Repeat, the system will do a delete and retype the new word.

Auto-repeat is done by ESC-select: if you hold down the ESC key while making a selection, a Repeat will automatically be done as soon as the selection is completed. This is useful when you're doing a large number of repeats.

Undo

Hitting SHIFT-ESC undoes the most recent edit sequence and restores the selection to its prior state. If you want to undo more than just the most recent sequence, use the Edit History tool which is described later.

Tree structure editing

As explained in the introduction, Tioga documents consist of a tree of nodes. The following commands let you break, join, and nest nodes in the tree.

Break: CTRL-RETURN – break node at insertion point to create a new node.

Join: CTRL-J – join node at insertion point with previous node.

Nest: CTRL-N – move selected nodes to deeper nesting level in tree.

UnNest: CTRL-SHIFT-N – move selected nodes to shallower nesting level in tree.

Break & Nest: CTRL-I – simultaneously insert a new node and nest it.

Break & UnNest: CTRL-SHIFT-I – simultaneously insert and unnest.

Looks

Characters have looks which are named by the lower case letters "a" to "z". Looks are interpreted by the style to change the appearance of the text. For example, look "e" might stand for *"emphasis"* and might result in italic face in one style and bold face in another. Each character has a set of looks — thus it may have several looks simultaneously, but each look occurs only once. You can use the Edit Tool to read or change the set of looks for selected characters. The following keyboard commands are also available for dealing with looks.

Selection Looks

You can change the selection looks with the following commands. (Recall that the LOOK shift is the top blank key to the right of BS.)

LOOK-char to add to selection looks.

LOOK-SHIFT-char to remove from selection looks.

LOOK-space to remove all selection looks.

Caret Looks

Caret looks determine the looks of typed-in text. The caret picks up the looks of the adjacent selected text whenever a selection is made. Changing the selection looks also changes the caret looks. However, if you wish to change the looks of the caret without changing the selection looks, left-click the character key twice in quick succession.

LOOK-char-char to add to caret looks.

LOOK-SHIFT-char-char to remove from caret looks.

LOOK-space-space to remove all caret looks.

Using Selections To Copy Looks

To copy the looks of some existing text to the primary selection, hit CTRL-Q, and then with the CTRL key still held down, select the text with the looks you want to copy. The source looks replace any looks the selection previously had.

Automatic Mesa formatting

The CTRL-M command scans the selection for Mesa keywords, comments, and procedure names and gives them looks k, c, and n respectively. (As a convenience during typein, the entire caret node is reformatted if the selection is a single character or less.) With the standard Cedar style, the keywords will then be displayed in small caps, the comments will be italic, and the procedure names will be boldface. We expect to provide more extensive reformatting capabilities in the future.

Formats

Just as characters have looks, nodes have formats. The "format" is the name of a rule in the style that tells how to modify various parameters when displaying the node. For example, a style for documents might contain formats for titles, headings, quotations, standard paragraphs, etc. The Edit Tool has facilities for reading and changing the format of a node, or you can use the commands described below. (By convention, the null format name is equivalent to "default".)

Setting and inserting caret node format

CTRL-← will delete the word to the left of the caret and make it the format of the caret node.

Fine point: In most cases you will give this command immediately after typing the format name, so the caret will naturally be in the correct place. However, to handle cases in which you select the name before hitting CTRL-←, the caret will automatically be forced to the right of the selection at the start of this command.

CTRL-SHIFT-← inserts the format name.

This gives you a simple way to find out the format of a selected node.

Using selections to copy formats

To copy the format of some exisiting node to the selection nodes, hit CTRL-F, and then with the CTRL key still held down, select the node with the format you want to copy.

The Edit Tool

The Edit Tool provides a variety of operations on Tioga documents.

The text fields in the Edit Tool follow the convention that clicking the field name with LEFT causes the contents of the field to be selected pending-delete while clicking with RIGHT causes the field to be cleared and selected.

Search and Substitute

Search

To do a search, enter the text you're looking for in the "Target" field, select where you want the search to start, and left-click "Search" to search forward, right-click to search backwards, or middle-click to search first forward then backwards. The system searches from the current selection and updates the selected viewer if the search succeeds. (Fine point: in both searches and substitutes, the match is limited to a single node – we do not yet have mechanisms for doing matches across node boundaries.)

The multiple choices below the "Replacement" field control what is matched in searches and replaced in substitutes. You can select or deselect an item by clicking it with the mouse. White text on black background means that the item is selected; black text on white means it is not selected.

Text – if this option is selected, match characters of target text when searching.

Looks – if selected, match looks of target text when searching.

Format – match format of target node.

Style – match style of target node.

Comment – match comment property of target node.

For example, if you pick the Looks option and deselect the Text option, you can search for any text that has a particular set of looks. If you pick only Text, the matching will ignore the looks of the target. If you pick Text and Looks, the matching text must match both the characters and the looks of the target text.

The other options deal with node properties. If you pick Format, the matching will be limited to nodes with the same format as the target node. Similarly, if you pick Style, the matching will only look at nodes whose style is the same as the target node's. Finally, if you pick Comment, the match will consider only nodes with the same value of the Comment property (TRUE or FALSE) as the target.

The first two rows of boxes below the multiple choices give you control over how matching is performed. Left-click with the cursor over a box to change the choice next to it. The various choices are as follows:

1. Case of matching text

   Match Case – matching text must have same case as target text.

   Ignore Case – matching text does not have to have same case as target text.

2. Interpretation of target text

   Match Literally – don't treat target text as a pattern.

   Match as Pattern – do treat target as pattern. (Patterns are described below.)

3. Context of target text

   Match Anywhere – ignore context of matching text.

   Match Words Only – matching text must not have adjacent letters or digits.

Match Entire Nodes Only — matching text must span entire node.

4. Matching target looks

    Subset as Looks Test — looks of matching text must include target looks.

    Equal as Looks Test — looks of matching text must be identical to target looks.

Substitute

To do a substitution, enter the new text in the "Replacement" field, enter the text to be replaced in the "Target" field, and hit "Substitute" in the menu at the top of the Edit Tool. The Text/Looks/Format/... options guide the search in the usual manner and also control what is replaced.

If you pick Text and Looks, the matching text will be replaced just as if you had selected it with pending delete and made a source secondary selection of the replacement text.

If you pick only the Looks option, the matching text will have the target looks removed and the replacement looks added.

If you pick only Text, the replacement text will have the looks of the replaced text added to it. (Fine point: if the looks of the replaced text are not uniform, the looks of the first character will be used throughout.)

If you pick Format, the matching node will get the format of the replacement node. Similarly, picking Style causes the matching node to get the style of the replacement node, and picking Comment causes the matching node to get the same value of the Comment property as the replacement node.

The final three boxes in the Search&Substitute section give you further control over this operation.

1. First character capitalization of the replacement text

    First cap like replaced — if the replaced text starts with a capital letter, force the first letter of the replacement to be a capital too.

    Don't change caps — leave the replacement capitalization alone.

2. What is done to the matching text

    Do Replace — do the usual substitute or replace.

    Do Operations — instead of doing a replace, select the matching text and then do the operations currently in the "Operations" field of the Edit Tool.

3. Where the substitutions will take place

    Within Selection Only — substitute is limited to current selection.

    After Selection Only — substitute after selection to end of document.

    In Entire Document — substitute in the entire selected document.

Case-by-Case Substitutes

In addition to doing global substitutes, you can decide on a case-by-case basis whether or not to replace the matching text by the new text. Use the search commands to find the first matching text. Then if you hit "Yes", the system will do a "Replace" followed by a search forward. If you hit "No", it will skip the "Replace" and simply do another search. Thus to selectively substitute, start with a search, then do "Yes" for the cases you want to change and "No" for the others. The "Replace" command simply does for the current selection what a substitute would do for a match. Finally, the "Count" command is available to tell you how many substitutions would take place without actually changing the document.

In order to do a search or substitute, you will typically need to fill in the Target or Replacement fields in the Edit Tool. Naturally, this changes the selection, and before you can do the operation, you must restore the selection to the place where you actually want it to take place. The system helps you with this by saving the primary selection if it is not in the Edit Tool when you left-click either the Target or the Replacement button. The commands along the top of the Edit Tool – Search, Substitute, Yes, No, Replace, and Count – restore the saved selection if the primary selection is in the Edit Tool when they are clicked. The net effect is that if you start out with the selection in the right place, you can left-click the Target or Replacement buttons, fill in the needed information, and then directly click one of the commands without needing to reselect since the system will do it for you.

Patterns for Search and Substitute

When you specify that the target be matched as a pattern rather than literally, the following symbols in the target text are interpreted specially. A short summary of these symbols appears at the bottom of the Search&Substitute section of the Edit Tool.

Characters

#	Match any single character.
*	Match shortest possible sequence of characters.
**	Match longest possible sequence of characters.
'	Match the next character in the pattern exactly.
~	Match any character except the next one in the pattern.

Alphanumeric characters

@	Match any single alphanumeric character (letter or digit).
&	Match shortest possible sequence of alphanumeric characters.
&&	Match longest possible sequence of alphanumeric characters.

Non-alphanumeric characters

~@	Match any single non-alphanumeric character.
~&	Match shortest possible sequence of non-alphanumeric characters.
~&~&	Match longest possible sequence of non-alphanumeric characters.

Blank characters

%	Match any single blank character.
$	Match shortest possible sequence of blank characters.
$$	Match longest possible sequence of blank characters.

Non-blank characters

~%	Match any single non-blank character.
~$	Match shortest possible sequence of non-blank characters.
~$~$	Match longest possible sequence of non-blank characters.

Miscellaneous

|	Match start or end of node.
{	Mark start of resulting selection.
}	Mark end of resulting selection.
<	Mark start of named subpattern.
>	Mark end of named subpattern.

The named subpatterns are of use in substitutes that reorder or duplicate parts of the matching text. The full syntax for a named subpattern is <name:subpattern>. The special case of "match any sequence of characters" is provided as a default – i.e., <name> is equivalent to <name:*>. As far as the matching is concerned, the occurrence of <name:subpattern> is the same as if the subpattern had appeared without a name, but it has the side-effect of remembering the subsection it matched. The "Replacement" field can contain <name>'s corresponding to named subpatterns in the target. The replacement text is constructed by replacing the <name>'s with the section of replaced text that matched the subpattern. The replacement is automatically considered to be a pattern whenever the target is – you don't need to do anything special to get the <name>'s in the replacement interpreted as subpatterns.

For example, if the target is

Target: WHILE <pred>[<arg>] DO

and the replacement is

Replacement: WHILE <arg> IS <pred> DO

then "WHILE blue[moon] DO" will be converted to "WHILE moon IS blue DO".

Looks, Formats, Styles, and Properties

Looks

The Looks commands let you read and modify the looks of the caret or the selection. The looks are shown as a series of letters in the "Looks characters" field. The box lets you pick whether you want the looks for the selection or the looks for the caret.

Get – fills the "Looks characters" field with the letters for the caret/selection looks.

Set – reads the "Looks characters" field and sets the looks of the caret/selection.

Clear – removes all looks from the caret/selection.

Add – adds the specified looks to the caret/selection.

Sub – removes the specified looks from the caret/selection.

Formats

The Format commands let you read and modify the format for the root node or the selected nodes. The box at the right lets you pick the case you want.

Get – fills the "Format name" field.

Set – reads the "Format name" field and sets the node's format.

Clear – removes the node's format name. This is the same as specifying "default" format.

Styles

A style is a collection of interpretations for looks and formats. The Style commands let you read and modify the name of the style for the root node or the selected nodes. The new style applies to the specified node and all the nodes within its sub-branches that do not themselves have explicit styles.

Get – fills the "Style name" field.

Set – reads the "Style name" field and sets the node's style.

Clear – removes any style specification from the node.

LoadStyleDefinition – reads the "Style name" field and reloads the style definition from that file with extension "Style". (Don't put the extension in the field – just put the style name and let the system add the extension.) Do this operation after you have edited (and saved) the style definition and want to load the new version.

LoadAbbreviations – reads the "Style name" field and reloads the abbreviation definitions from that file with extension "Abbreviations". Do this after you have edited the abbreviations and want to load the new version.

Properties

A node can have an arbitrary set of "properties". Each property consists of a name and a value. Certain properties are used by the system, but you (and your programs) are free to add others.

The "Property name" field specifies the name of a property. (Incidentally, case distinctions do matter in property names – "foo" is a different property than "Foo".) The "Property value" field specifies a value. The box lets you pick whether you are talking about properties of the root node or the selected nodes.

Get – fills the "Property value" field with the current value of the named property.

Set – reads the "Property value" field and sets the property value.

Remove – removes the named property from the node.

List – fills the "Property name" field with the names of the node's properties.

In addition to setting and reading properties, there is a mechanism that lets you find nodes with a certain property value. For example, if you have annotated a document with comments under the property name "MyOpinion", and you want to find nodes in which your comment included the word "good", enter the name in the "Property name" field and the text in the "Value pattern" field. Then use the "Find" command to search forward or backwards from the caret node for a node with a value of the property that matches the pattern. (Left-click to search forward, right-click to search backwards.) Value patterns can use the standard set of special characters for searches. If a match is found, the node is selected and the property value is displayed.

Comment Property

All nodes have a "Comment" property which is either "TRUE" or "FALSE". If its value is TRUE, the text of the node is not seen by programs such as the compiler that are only interested in the basic text contents. This makes it possible to intermix documentation and program text without requiring special escape characters to mark the start and end of comments. You can set the Comment property by using the Edit Tool or with the following commands.

CTRL-\ – set comment property of all selected nodes to TRUE

CTRL-SHIFT-\ – set comment property of all selected nodes to FALSE

If you are worried about not knowing at a glance what is a comment node and what is not, find a StyleRule in the style that applies to all the formats you are interested in (e.g., "standard" in Cedar.style), and insert the line

```
isComment {visible underlining} {none underlining} .ifelse cvx .exec
```

This will cause the contents of all comment nodes to be underlined. (You might want to say "strikeout" instead of "underlining" if the style already uses underlining to mean something else, and legibility is not your main concern.)

Miscellaneous

Sort and Reverse

These commands let you sort or reverse lists of things. The "Sort" command sorts things in alphabetical order, ignoring case. "Sort-and-remove-duplicates" is useful when you are merging sets of things. The box below the Sort button lets you pick whether you want increasing or decreasing order in the result. The right box lets you pick what will be sorted: text delimited by blanks, lines delimited by carriage returns, or branches of the document tree. Leading blanks are ignored when sorting lines or branches.

Operations

These commands let you construct simple edit macros. The "Operations" field holds a text description of a command sequence. The "GetLast" command fills in the Operations with the description of the most recent sequence, i.e., the one a Cancel would cancel or a Repeat would repeat. "Do" executes the description in the operations field. "Begin" marks the start of a command sequence. "End" fills the Operations with the description of the commands since the most recent "Begin". "SetCom" stores the operations under the CTRL-number key for the number in the "Command [0..9]" field. Conversely, "GetCom" fills the "Operations" field with the current CTRL-number definition.

To see how this works, select the following word: "hello". Hit DEL and type in "howdy". Now hit the "GetLast" button. The Operations field should now contain: Delete "howdy". Edit the Operations field contents to replace "howdy" by "goodbye", then hit the "SetCom" button (first put 1 in the Command field if it's not already there). Now select the "howdy" you typed earlier and hit CTRL-1. If all went well, the "howdy" will have been deleted and "goodbye" inserted in its place. More examples of edit macros will be given later.

Searches and Operations on Files

You can use the EditTool to look through a list of files for one in which the current search specifications are satisfied. This can be accomplished by clicking the "SearchEachFile" button after filling in the "Files" field with the file names (or "@" followed by the name of a file that holds the list of file names). When you left-click SearchEachFile, a new viewer is created, and one-by-one the files will be loaded and searched until a match is found. The list of files will be updated when a match is found, so that when you are finished with one file you can left-click SearchEachFile again to look for the next one. You can hit the "Stop" button at the top of the EditTool to interrupt the search, but you should not try doing other operations while the search is in progress since it resets the selection each time it loads a file.

Occasionally you will want to apply certain operations to an entire set of files. You can do this by filling in the "Operations" field and the "Files" field, and then clicking "DoForEachFile". A new viewer will be created, and one-by-one the files will be loaded, selected, edited, and saved. No confirmation is required for the saves, so the entire process can go on without you. In fact, you should not try to do anything else while this is going on since the operations use the primary selection. The one exeception is the "Stop" button which you can hit to terminate the process.

Operations via the User Exec

You can invoke a set of operations from the User Exec as well as directly from the Edit Tool. When you run TiogaExecCommands, one of the commands it registers is DoTiogaOps which expects a command line containing operations in the same format as in the operations field. One possible application of this is to create a command file that initializes various Edit Tool switches to your favorite settings. For example, you could use the following to set up some of the search parameters: DoTiogaOps IgnoreCase MatchWords MatchPattern.

Edit Commands

This section of the Edit Tool contains buttons for all the basic edit commands. These are useful if you can't remember what keys to hit for an infrequent command or if you've set your user category to beginner or intermediate to filter out certain commands. Many of the buttons correspond to commands that have been documented above. However, a few of them are primarily of use in constructing edit macros and have not been mentioned before. For completeness, all the buttons will be given a brief description.

Modifying selections

CaretBefore – move caret to front of selection
CaretAfter – move caret to end of selection
CaretOnly – reduce selection to a caret
Grow – selection grows in hierarchy of point, character, word, node, branch
Document – select entire document
PendingDelete – make primary selection pending delete
NotPendingDelete – make primary selection not pending delete
MakeSecondary – make the primary selection become the secondary selection
MakePrimary – make the secondary selection become the primary selection
CancelSecondary – remove the secondary selection
CancelPrimary – remove the primary selection

Copy, Move, and Translate

ToPrimary – copy/move secondary to primary
ToSecondary – copy/move primary to secondary
Transpose – transpose the primary and the secondary

Moving the caret

GoToNextChar – make primary caret only and move one character toward end of document or one toward start if you right-click instead of left-clicking
GoToNextWord – like GoToNextChar, but move toward end by "words" or one toward start if you right-click instead of left-clicking
GoToNextNode – move caret one node toward end or one toward start if you right-click instead of left-clicking

Saving and Restoring the selection

SaveSel-A – save primary selection to restore later
RestoreSel-A – restore the selection previously saved by SaveSel-A
SaveSel-B – save primary selection to restore later

RestoreSel-B – restore the selection previously saved by SaveSel-B

Extend the selection to matching brackets
(...) – extend primary selection to matching parens
<...> – extend to matching angle brackets
{...} – extend to matching curly brackets
[...] – extend to matching square brackets
"..." – extend to matching double quotes
'...' – extend to matching single quotes
-...- – extend to matching dashes
▶...◀ – extend to matching placeholder brackets

Find placeholders
Next▶...◀ – move primary selection to next placeholder
Prev▶...◀ – move to previous placeholder

Delete and Paste
Delete – delete the primary selection
Paste – insert saved text at caret
SaveForPaste – save text for later pasting; overwrites text saved by Delete

Repeat and Undo
Repeat – repeat the last command sequence
Undo – undo the last command sequence

Deleting character/word next to caret
BackSpace – delete the character to the left of the caret
BackWord – delete the word to the left of the caret
DeleteNextChar – delete the character to the right of the caret
DeleteNextWord – delete the word to the right of the caret

Inserting matching brackets around the selection
Add( ) – insert parens around the selection
Add< > – insert angle brackets
Add{ } – insert curly brackets
Add[ ] – insert square brackets
Add" " – insert double quotes
Add' ' – insert single quotes
Add- - – insert dashes
Add▶ ◀ – insert placeholder brackets

Capitalization
AllCaps – make the selection upper case
AllLower – make the selection lower case
InitialCaps – make the words in the selection start with caps
FirstCap – make the selection start with a cap

Special characters

MakeOctalCharacter – convert the 3 digits before the caret to the corresponding character

MakeControlCharacter – convert the character before the caret to a control character

UnMakeOctalCharacter – convert the character before the caret to 3 digit representing its character code

UnMakeControlCharacter – convert the control character before the caret to a normal character

Tree structure commands

Break – break node at caret

Join – join caret node to one before it

Nest – move selection deeper in tree

UnNest – move selection higher in tree

Miscellaneous commands

CommentNode – set Comment property of all selected nodes to TRUE

NotCommentNode – set Comment property of all selected nodes to FALSE

ExpandAbbreviation – expand the abbreviation to the left of the caret

MesaFormatting – add Mesa looks, formats, and style to selection

Command 0 1 2 3 4 5 6 7 8 9 – do saved command macro

Writing Edit Macros

A few examples should get you started writing your own edit macros. First, assume you find yourself doing a lot of edits of the form <stuff> becomes <pred>[<stuff>] for various values of <stuff> and <pred>. You might like a single command to insert the brackets and move the caret to the place where you will insert the <pred>. To construct such a command, first select a particular <stuff>, hit the Add[ ] button in the Edit Tool, the CaretBefore button, and the GoToNextChar button using the right mouse button. Then hit GetLast to fill the Operations field, enter "1" in the "Command [0..9]" field, and hit SetCom to save the macro definition. Now you can select <stuff>, hit CTRL-1, and be ready to insert before the left bracket.

A macro to delete matching parentheses will show the use of the SaveSel and RestoreSel commands. Make a selection anywhere inside a pair of matching parens. Give the following sequence of commands: (...) to extend the selection, CaretAfter to position the caret at the right, SaveSel-A so we can restore the selection later, BackSpace to delete the right paren, RestoreSel-A to get the selection back, CaretBefore to move the caret to the front, and finally DeleteNextChar to delete the left paren. Now you can hit GetLast and SetCom to save this macro.

Other operations on the Edit Tool can also be programmed using edit macros. For example, assume you want a macro to substitute "which" for "that". The following set of commands will load the target and replacement fields and do the substitute: right-click to select and clear the target field, type "that" as the target text, right-click to select and clear the replacement field, type "which" as the replacement text, and finally hit the Substitute button. This macro will only change the target and replacement fields. The other parameters of the substitute are not changed and thus behave like "free variables" of the macro. If you want to bind more of the choices, such as Match Case or Ignore Case, you simply include those commands as part of the macro. For example, you could select Ignore Case just before restoring the selection, and the macro would always set the ignore case flag when it was executed. Note that you must hit the "Target" and "Replacement" buttons to enter the target and replacement text. Directly selecting in those fields would not produce a correct macro because it would terminate the edit sequence and would also fail to identify which field was being filled in.

When you save an edit macro under a CTRL-number key, it only stays around for the rest of the session. If you want to save one for tomorrow, you can of course copy the operations to a file and redefine it another time. But if you really want to make a macro a permanent part of your user interface, you can add it to your "TIP table" which describes the translation from keyboard and mouse actions into executable tokens such as those in edit macros. The details of how to do this are given in a later section.

The Edit History Tool

The Edit History tool lets you undo edits in the same way that Undo does, but it lets you go back farther in history than just the most recent sequence. To create an Edit History tool, type "EditHistory" to the Exec. At the top of the tool are two fields and four commands. The bottom part of the tool is a text field to hold descriptions of previous edit events.

The system keeps a history of a certain number of the most recent edit events. You can find out how large this history is by the "Get" command which will enter a number in the "history size" field. The "Set" command will change the history size to whatever value you've entered in the size field. The default size is 20; you can change this by making an entry in your User.Profile of the form "EditHistory: <history-size>".

The "Show" command will display the events starting with the number in the "since event number" field. If that field is empty it will show as many as are still remembered. The format of the entries is <event-number>, TAB, and then a list of operations. The "Undo" command undoes the edits since the specified event number.

Printing and the TypeSetter Tool

In the glorious future, Tioga will have an interactive typesetter that will let you make incremental revisions to a typeset document to adjust pagination, page layout, and other details before actually printing it. Michael Plass and I have started work on this, but until it's available, the current typesetter will do a more than adequate job of letting you print your files. Documentation for the TSetter tool may be found in [Indigo]<Cedar>Documentation>TSetterDoc.Tioga.

TIP Tables for Tioga

The acronym "TIP" stands for "terminal interface package". TIP tables describe the translation from keyboard and mouse actions into executable tokens. The standard TIP table for Tioga is found in the file "Tioga.TIP" and contains all the gory details about things such as multi-clicks of mouse buttons with various combinations of shift keys. It's unlikely that you want to try changing anything in Tioga.TIP, but you can consider adding commands to your own TIP table for things such as your favorite set of edit macros. Your TIP table can be "layered" on top of the Tioga table so that the system will try to interpret actions according to your table before looking at the standard one. The User.Profile contains an entry named "TiogaTIP" in which you can enter the file name of your TIP table to be layered over the standard Tioga table. The entry should look like something like this

```
TiogaTIP: MyOwn.tip Default
```

where "MyOwn.tip" is the file name for your table. Notice that your table goes first in the list; definitions in tables occurring early in the list take precedence over those that appear later. The special name "Default" refers to Tioga's default TIP table and will typically be the last entry in the list.

The file "MyOwn.tip" contains a sample table that you can edit to produce your own. It also contains documentation about the syntax of TIP tables and the macro package which is used with them.

Use the Exec command ReadTiogaTipTables to reload a TIP table after you've changed it, or, if your user category is advanced, hit CTRL-! to reload the Tioga profile information.

Styles

A style is a collection of interpretations for looks and formats. The interpretations are represented as procedures written in a simple language that uses the JaM interpreter. The procedures set various formatting parameters such as font size and line spacing.

Note: If a document doesn't specify a style, the system will use a default. You can say what the default will be by adding user profile entries for DefaultStyle or ExtensionStyles (described below).

To determine the formatting parameters for a particular node, the system first gets the parameters for its parent and then executes the appropriate formatting procedure from the style. If the node doesn't have an explicit format, or if the format it has is not defined in the style, the system will execute the default formatting procedure instead. For root nodes, the default is the rule named "root"; for other nodes, it is the rule named "default".

Formatting parameters for the text within a node are determined in a similar manner. The system first gets the parameters for the node and then executes the formatting procedures for the looks of the text. For look "a", it executes the rule named "look.a", for look "b", the rule named "look.b", etc.

Properties related to styles

Prefix and Postfix properties

The values of these properties are command sequences that might be part of a formatting procedure. The Prefix commands are executed just before the standard formatting procedure for the node, and the Postfix commands are executed just after it. This makes it possible to modify the values of the formatting parameters that are the input and output of the format. You may want to use this to make local formatting changes, such as modifying tab stops for a particular table. However, don't abuse this facility. If you find you are making many local changes, you should probably modify the style instead.

StyleDef property

In typical use, a style lives in its own file and is referred to by the various documents that use it. However, in some cases you may have a style that is only used in one document, and you'd like to include it as part of the document to avoid the trouble of maintaining the style as a separate file. You can do this by using the StyleDef property. The value of this property is a style definition. The style is automatically saved and loaded as part of the file and applies to the node and its children just as if you had set the node style in the usual manner with the EditTool.

[Note: If you use a StyleDef and it doesn't seem to do anything, do a Clear of the style name using the EditTool.]

Style definitions

A style definition begins with the command "BeginStyle" and ends with "EndStyle". Between these come any number of commands and definitions. The style is parsed and interpreted using JaM, so you will want to read the JaM documentation if you're going to spend much time writing styles.

Attached styles

You may want to define a new style that is only slightly different than an existing one.

One approach would be to copy the existing style and edit it to make the changes. However, it may be preferable to track whatever modifications to the other style that may happen in the future. This can be done by "attaching" the old style to the new one. For example, if your new style includes a command of the form

```
(Cedar) AttachStyle
```

then the new style will automatically include everything from the current version of Cedar style. Thus, if a format is not defined in the new style, it will be taken from Cedar style instead.

ScreenRules, PrintRules, and StyleRules

The basic form for a rule definition in a style is

```
(name) "comment string" { commands } StyleRule
```

In many cases it will be desirable for a rule definition to be different depending on whether the output is for printing or for display. To simplify this, you may define a format both as a ScreenRule and as a PrintRule. The ScreenRule definition will be used for display, while the PrintRule will be used for printing. If there is only a StyleRule definition, it will be used for both printing and display.

Formatting Parameters

This section lists some of the formatting parameters currently supported. Typically, the name of the parameter is a command defined in JaM that pops an item from the stack and makes it the new parameter value. However if the item on the top of the stack is the word "the", the commands push the current parameter value so procedures can read parameters as well as write them. For numeric parameters, the following mechanisms are provided to simplify making incremental changes to the current parameter value:

<amount> bigger	adds the amount to the current parameter value
<amount> smaller	substracts the amount from the parameter value
<amount> percent	multiplies the parameter value by the specified percentage i.e., value ← (amount/100)*value
<amount> percent bigger	increases value by specified percentage
<amount> percent smaller	decreases value by specified percentage

Font Parameters

family	the name of current font family, such as "TimesRoman"
size	value is the font size in points
face	one of regular, bold, italic, or bold+italic

You can also add or remove italic or bold by means of the following commands:

+bold face, -bold face, +italic face, or -italic face

underlining	one of all, visible, letters+digits, or none
strikeout	one of all, visible, letters+digits, or none

Indent Parameters

leftIndent	left indent for start of lines
rightIndent	right indent for end of lines
firstIndent	added to leftIndent for first line of paragraph
restIndent	added to leftIndent for remaining lines of paragraph
topIndent	distance from top of viewer/column to first baseline

Leading Parameters

Leading parameters are stored as triples of <size, stretch, and shrink> which, following Knuth, we refer to as "glue". You can set the separate components individually, or you can push three values on the stack and use one of the leading glue commands to set them all at once.

There are three kinds of leading corresponding to the spaces between lines in a node, the space above a node, and the space below it. The actual space between a node is the maximum of the "below" leading of the first node and the "above" leading of the second.

leading	distance between baselines within a node
leadingStretch	how much leading can increase
leadingShrink	how much leading can decrease
leadingGlue	size, stretch, and shrink for leading
topLeading	distance between baselines above a node
topLeadingStretch	how much top leading can increase
topLeadingShrink	how much top leading can decrease
topLeadingGlue	size, stretch, and shrink for top leading
bottomLeading	distance between baselines below a node
bottomLeadingStretch	how much bottom leading can increase
bottomLeadingShrink	how much bottom leading can decrease
bottomLeadingGlue	size, stretch, and shrink for bottom leading

Layout Parameters

lineFormatting	FlushLeft, FlushRight, Justified, Centered
minLineGap	min distance between line top and previous bottom (can be negative)
leftIndent	all lines indent this much on left
rightIndent	all lines indent this much on right
firstIndent	first line indent this much more on left
restIndent	other lines indent this much more on left
topIndent	top line at least this much down from top of viewer/page

bottomIndent	bottom baseline at least this up from bottom of page
Page Layout Parameters	
pageWidth	width of the paper
pageLength	height of the paper
leftMargin	whitespace at left of the page
rightMargin	whitespace at right of the page
topMargin	whitespace at top of the page
bottomMargin	whitespace at bottom of the page
headerMargin	height of area below topMargin for headers
footerMargin	height of area above bottomMargin for footers
bindingMargin	not used at present
lineLength	width of lines of text
Dimensions	
pt	point
pc	pica
in	inches
cm	centimeters
mm	millimeters
fil	10↑4 points
fill	10↑8 points
Miscellaneous	
style	the name of the current style
isComment	pushes .true or .false according to comment property of node
isPrint	pushes .true if executing print rules, else .false
nestingLevel	pushes integer; 0 for root, 1 for top level, etc.

Tioga User Exec Commands

Note: If the following commands are not known to the UserExecutive, type "Run TiogaExecCommands".

ReadTiogaTipTables

Causes Tioga to read its TIP tables again. If your user category is advanced, you can also invoke this operation by selecting in any Tioga document and hitting CTRL-!.

WritePlain

Make Tioga files unformatted by eliminating everything except the plain text. Inserts leading tabs before each node according to its nesting in the tree and terminates each node with a carriage return.

WriteMesaPlain

Same as WritePlain, except inserts double dashes at start of comment nodes.

ReadIndent

Build Tioga files with one node per line of source with indenting based on white space at the start of lines.

TiogaMesa

Convert Mesa files to Tioga format by combining a ReadIndent with a CTRL-M over the entire file.

DoTiogaOps

Expects a command line containing operations in the same format as in the EditTool operations field. Among other things, you can use this to initialize various EditTool choices such as IgnoreCase or MatchWords.

Tioga User Profile Entries

Default file extensions

This determines what extensions Tioga should look for in opening files. The entry is of the form

```
SourceFileExtensions: mesa tioga df cm config style
```

You may also have an entry for implementation extensions to be used with the "load impl" commands.

```
ImplFileExtensions: cedar mesa
```

Open First Level Only

If set to true, documents will be opened with only their first level showing. Default is false.

```
OpenFirstLevelOnly: TRUE
```

Default Styles Determined by File Extensions

This entry specifies the default style to be used with documents that do not explicitly name a style. The style is determined by the extension in the file name. The entry is of the form

```
ExtensionStyles: <extension1> <stylename1> <extension2> <stylename2> ...
```

To specify a default style for files with no extension in their name, use the fake extension name "null" in this list.

Default Style

This entry specifies the default style to be used with documents that do not explicitly name a style and do not have an extension given in the ExtensionStyles list. The entry is of the form

```
DefaultStyle: Cedar
```

Default submenus

This entry specifies which menus, if any, should automatically be displayed when you create a new Tioga viewer by clicking one of the buttons in the upper right corner or by giving an Exec command. The entry is of the form

```
DefaultTiogaMenus: places levels
```

or

```
DefaultTiogaMenus: none
```

You may delete or reorder the menu names to suit your tastes.

Scroll bottom offset

When you are typing and the caret goes to a new line just off the bottom of the viewer, Tioga will automatically scroll the viewer up a little to make the caret visible again. This parameter controls how far up to scroll; a big number causes larger but less frequent glitches.

```
ScrollBottomOffset: 3
```

Scroll top offset

When you do a Find command you may want to see a few lines in front of the match to give you more context. This parameter tells Tioga how many extra lines to want in such situations. The entry is of the form

ScrollTopOffset: 1

Selection Caret

The default behaviour for Tioga is to place the caret at the end nearer the cursor when the selection is made. Some people have requested to have the caret always placed at one end or the other, hence this profile entry. The choices are before, after, and balance. The entry is of the form

SelectionCaret: before

Selection Displacement

This lets you specify a vertical displacement for making selections. Tioga behaves as if you had pointed this number of points higher up the screen so that you can point at things from slightly below them. The entry is of the form

YSelectFudge: 2

TIP Table

This entry specifies the TIP tables to use with Tioga documents. The entry is of the form

TiogaTIP: MyOwnTip.tip Default

See the section on TIP tables for more information.

Unsaved Documents Cache Size

This controls the number of unsaved documents the system will remember. The entry is of the form

UnsavedDocumentsCacheSize: 4

Show Unsaved Documents List

If this is true, a viewer will be created holding an up-to-date list of the unsaved documents that can still be reloaded. The entry is of the form

ShowUnsavedDocumentsList: TRUE

User category

As described above, this entry lets you control your user category. The alternatives are Beginner, Intermediate, and Advanced. The entry is of the form

UserCategory: Intermediate

Command Summary

This is a short summary of the Tioga commands available using the keyboard and mouse. Other commands are available through the Edit Tool, the Edit History Tool, and various User Exec operations.

The extra keys on the keyboard are used for common editing actions. Recall that the bottom right blank key can be used as another CTRL key, and you can use either CTRL key and either SHIFT key interchangeably.

ESC	Repeat last action
SHIFT-ESC	Undo last action
ESC-select	Automatic repeat of last action when finish selection
DEL	Delete
LF	Load file in "No Name" viewer
CTRL-LF	Load Impl file in "No Name" viewer
BS	Backspace character
SHIFT-BS	Delete next character
CTRL-BS	Backspace word
CTRL-SHIFT-BS	Delete next word
NEXT	Find next placeholder (middle blank key)
SHIFT-NEXT	Find previous placeholder
RETURN	Insert carriage return with leading spaces copied from previous line
SHIFT-RETURN	Insert carriage return
CTRL-RETURN	Break node

Mouse button clicks are used for making selections.

CLICKS	LEFT	MIDDLE	RIGHT
SINGLE	Select letter	Select word	Extend selection at current level
DOUBLE	Select node	Select branch	Reduce selection level and extend
TRIPLE			Increase selection level and
extend			

Selections are used for delete, copy, and move.

CTRL-select	Delete when finish selection
SHIFT-select	Copy to primary
CTRL-SHIFT-select	Move to primary

LOOK commands are used for editing looks. (The LOOK shift is the top blank key.)

LOOK-char	Add look to selection
LOOK-SHIFT-char	Remove from selection
LOOK-space	Remove all from selection

Except for CTRL-A, CTRL-H, and CTRL-W, the following commands are not enabled for beginning users. We provide both CTRL-A and CTRL-H as a convenience to users with strong habits from previous systems. A "*" indicates a command enabled for advanced users only.

CTRL-A	Backspace character
CTRL-SHIFT-A	Delete next character
CTRL-B	Insert matching placeholder brackets
CTRL-C	Lower case
CTRL-SHIFT-C	Upper case
CTRL-CLICK-C	Initial caps
CTRL-SHIFT-CLICK-C	First cap

CTRL-D	Select document
CTRL-E	Expand abbreviation
CTRL-F-select	Copy format to primary *
CTRL-H	Backspace character
CTRL-SHIFT-H	Delete next character
CTRL-I	Indent (does Break and Nest) *
CTRL-SHIFT-I	Unindent (does Break and UnNest) *
CTRL-J	Join nodes *
CTRL-K	Make control character
CTRL-SHIFT-K	Unmake control character
CTRL-M	Automatic MESA formatting
CTRL-N	Nest *
CTRL-SHIFT-N	UnNest *
CTRL-O	Make octal character
CTRL-SHIFT-O	Unmake octal character
CTRL-P	Paste
CTRL-Q-select	Copy looks to primary
CTRL-S-select	Copy primary to selected destination
CTRL-T	Time
CTRL-V	Select visible (expand to selection to blanks)
CTRL-W	Backspace word
CTRL-SHIFT-W	Delete next word
CTRL-X-select	Select for transpose with primary
CTRL-Z-select	Move primary to selected destination
CTRL-]	Select matching [..]'s (same for },), and >)
CTRL-[	Add matching [..]'s (same for ", `, -, {, (, and <)
CTRL-!	Have Tioga read its TIP tables again *
CTRL-←	Set format to word before caret *
CTRL-SHIFT-←	Insert format name *
CTRL-\	Set comment property TRUE *
CTRL-SHIFT-\	Set comment property FALSE *

How To Use Walnut

Version 4.2

Release as [Indigo]<Cedar>Documentation>WalnutDoc.tioga, .press
Came from [Indigo]<CedarDocs>Manual>WalnutDoc.tioga, .press
By Willie-Sue Haugeland
Last edited By Willie-Sue Haugeland on June 8, 1983 4:03 pm
Jim Donahue on June 2, 1983 1:17 pm

Abstract Walnut is a computer mail system interface that runs in Cedar. It provides facilities to send and retrieve mail (using the Grapevine mail transport system), and to display and classify previously retrieved messages.

Walnut is under active development. This document describes how to obtain and use Walnut 4.2, the latest version of Walnut released with Cedar 4.2.

How to Use Walnut: Contents

[If you are reading this document on-line, try using the Tioga Levels and Lines menus (if you can) to initially browse the top few levels of its structure before reading it straight through.]

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

0. Introduction

Walnut is a mail system that runs in Cedar. Walnut provides facilities to send and retrieve mail (using the Grapevine mail transport system), and to display and classify stored (i.e. previously retrieved) messages. Walnut uses the Cypress database system to maintain information about stored messages; we hope that this will allow Walnut to integrate smoothly with new applications, such as a calendar system or an online "white pages".

Walnut is under active development. It is not unusual for new versions of Walnut to be distributed to willing users between formal Cedar releases. Though the underlying database structures used by Walnut change from time to time, these changes never require any hand-editing on the part of users; Walnut performs the conversions automatically.

1. Database Structure

We shall describe a user's model of Walnut's database. This model suppresses many details whose understanding would be required in writing a new application on Walnut's database, but are irrelevant for accessing the database through Walnut's user interface.

Walnut's database contains two entity types: *message* and *message set.*

A message entity corresponds to a message retrieved from the Grapevine mail transport system. Like all database entities it has a *name*, consisting of the sender's RName concatenated with a unique message ID provided by Grapevine. A message also has several immutable properties: its *sender*, its *subject*, and so on. Its *unread* property is a BOOL whose value is TRUE when the message is first stored in the database, and is set to FALSE when the message is first displayed.

A message entity is also a member of one or more message sets. A message set entity is named by a text string containing no embedded blanks. There are two distinguished message sets: *Active* and *Deleted.* A newly-retrieved message is made a member of Active. A message that is removed from all other message sets is added to Deleted. Using the Active and Deleted message sets in this way ensures that each message belongs to at least one message set.

2. User Interface

Walnut implements four viewer types: the Walnut control viewer, the message set display viewer, the message display viewer, and the message composition viewer. When Walnut is running there is one Walnut control viewer, and any number of instances of the other viewer types. The iconic form of the Walnut control viewer is a mailbox. Anything that can be done with Walnut can be done by starting from the control viewer (sometimes by creating other viewers).

Walnut's user interface attempts to be consistent with conventions used elsewhere in Cedar. Clicking LEFT on a button representing a Walnut entity (a message or a message set) "selects" the entity (makes it an implied parameter to other operations); clicking MIDDLE on such a button opens the entity (displays more information somehow). This is analogous to the behavior of icons in Cedar.

Unless otherwise specified, hereafter "click" means "click with the LEFT mouse button".

2.1 Message sets

The message sets in a Walnut database are represented by buttons in the Walnut control viewer. The **Active** and **Deleted** message set buttons always appear first; other message set buttons appear in alphabetical order. There may be several rows of these buttons.

To create a new message set (containing no messages) type its name into the **MsgSet**: field of the control viewer, then click **Create**. A message set button with this name will appear. Similarly, to delete an existing message set, type its name and click **Delete**. Walnut requests confirmation if the message set contains any messages. If you delete a message set containing messages, the messages are deleted from the set (as if clicking each message with CTRL-LEFT as described below) before the set is destroyed. To find out how many messages are currently in a message set, type the name of the set and click **SizeOf**.

To create a message set display viewer, click MIDDLE on the corresponding message set button of the control viewer. The iconic form of a message set displayer is a stack of envelopes.

The message set displayer contains a one-line button for each message in the set. Each message button looks much like a line in the top window of Laurel: it shows the date of the message, the name of the sender (or To: field if the sender is the current user), and the message subject. At most one of the message buttons in a set is selected (shown with a grey background). To select a message, click LEFT on it; to select and display a message, click MIDDLE on it. To delete a message from the message set, click CTRL-LEFT on it; its button will disappear. (Note that if this message belonged to no other message set, it is added to the Deleted set, and hence is still accessible.)

A message set displayer also contains several command buttons that operate on the selected message.

Categories lists (in the Walnut control window) the message sets in which the selected message appears (a message can simultaneously be in several message sets).

MoveTo adds the message from the message set to all of the message sets selected in the control window, after first deleting it from the message set.

Display displays the selected message.

Delete deletes the message from the message set; if it thus becomes a member of no other message sets, then it is moved to the **Deleted** message set.

AddTo adds the messages to the selected sets without first doing the deletion.

Finally, the **Active** message set includes a **NewMail** button that reads new mail and adds the messages read to this message set.

For the **MoveTo**, **Delete**, and **AddTo** buttons, LEFT-clicking simply performs the operation described above, while RIGHT- or MIDDLE- clicking performs the operation and displays the next message in the set.

2.2 Messages

As described above, a message can be displayed by clicking MIDDLE on a message button of a message set displayer. This creates a message display viewer, whose iconic form is an envelope (clicking SHIFT-MIDDLE on a message button causes the created viewer to fill the entire column). The message within such a viewer is not editable. This viewer is associated with the message set that created it, so clicking MIDDLE on another message of the same message set shows this new message in the same message displayer. This is designed to avoid a proliferation of message displayers. Of course, there are times when you really want to create viewers on several different messages in one set. Clicking a message displayer's **Freeze** button (which then disappears) permanently binds the message to the message displayer. If all message displayers for a given message set are frozen, then MIDDLE clicking in the message set creates a new displayer. A frozen message displayer cannot be unfrozen, but can be destroyed.

A message displayer has several other buttons. The **Categories** button is the same as that on message set displayers. **Answer** and **Forward** create a Walnut Send viewer initialized either with a proper heading for an answer to the message or a copy of the message for forwarding. The **Print** button uses the TSetter to print the message -- it will complain (in the Walnut control window) if the TSetter is not loaded. And the remaining **Split**, **Places** and **Levels** buttons are from Tioga.

2.3 Sending mail

To create a message composition viewer click the **NewForm** menu item in the Walnut control viewer, or use the **Answer** or **Forward** buttons on a message display viewer. In iconic form this viewer is the back of an envelope, with the title "WalnutSend ...6/02/83". The message composition viewer is a Tioga viewer for typing in the message header fields and message body. **NewForm**, **PrevMsg**, and **GetForm** menu items are guarded while the Sender is dirty. When a command requires confirmation, appropriate buttons appear in the viewer menu area; the control window is used for messages.

Clicking the **Clear** button of a message composition viewer causes its message text to be replaced by a standard message form. **PrevMsg** restores the contents of the last successfully sent message. **GetForm** loads into the viewer the file whose name is currently selected (if no file extension is given and no file is found with the selected name, the extension ".form" is used); it also sets the input focus at the first placeholder. **Confirm** and **Deny** buttons appear for confirmation if the existing message text has been edited. **StoreMsg** stores the contents of the viewer in the selected file: this gives a way to save partially composed messages.

The Split, Places and Levels menu items are the usual Tioga operations; if a message composition viewer is split, then **NewForm**, **PrevMsg** and **GetForm** operations on a split viewer will cause all but the one bugged to be destroyed (the message composition viewer becomes "unsplit"). This will also happen if **Answer** or **Forward** reuse a split message composition viewer.

Clicking the **Send** button initiates the sending process. If the message is addressed to a public distribution list or to more than twenty individuals, it probably should contain a Reply-To: field. If it does not, a message is printed in the control window and buttons labeled **Self**, **All**, and **Cancel** appear. These buttons mean "Reply-To: sender", "no Reply-To: field", and "Reply-To: sender but don't send", respectively. If errors occur in sending, a message is printed in the control window. If the transmission is successful, the message is saved and a new form appears; RIGHT-clicking **Send** causes the message composition viewer to become iconic after syntax checking has been done. While a **Send** is in progress, the viewer is not editable. An **AbortSend** button is visible during part of the sending process, should you decide not to send the message. The last successfully sent message can to restored by clicking **PrevMsg**.

Clicking the **Destroy** button asks for confirmation if there is an unsent message in the viewer. It is a bad idea to destroy a message composition viewer while message transmission is in progress.

2.4 Retrieving mail

Walnut polls the mail servers at regular intervals. If there is new mail for the logged-in user, the Walnut control viewer displays a message like

You have new mail at 27-Sep-82 14:28:45 PDT

Clicking the NewMail button in either the control viewer or the Active message set viewer retrieves all the new mail. You can "button ahead" during message retrieval. The new messages appear as buttons at the bottom of the Active message set viewer, with a "?" in the leftmost column of each new message button to show that the message is unread. If the control window is iconic and there is new mail, the flag on the icon is raised; for convenience, there is a NewMail button in the Active MsgSet displayer, which will retrieve the waiting mail.

2.5 Global operations

The Walnut control viewer has a few more buttons that will now be described. Clicking Commit commits all changes that you have made to Walnut's database. The only other operations that automatically commit database changes are Destroy and CloseAll. If you perform a Walnut operation that updates the database and then boot or rollback without doing a Commit, the effects of that operation are in the log and will be performed on the database and then committed when you next run Walnut.

The CloseAll button furnishes a quick way of ending a session with Walnut. CloseAll destroys all message displayers, and closes all message set displayers. Then it executes a Commit and closes the Walnut control viewer.

The Archive operation provides a way to copy a set of messages into a file that can later be read by either Walnut or Laurel. Type a filename into the OnFile: field and click Archive. All messages in the currently selected message sets are copied to the named file (if the file name has no extension, ".ArchiveLog" is assumed).

Clicking the (guarded) Expunge button expunges the Walnut database: all messages in the Deleted set disappear without a trace. (Laurel purges deleted messages every time you Quit or change mail files.) Without the Expunge operation, Walnut's database would grow without bound. The Expunge operation may fail if your disk is nearly full; this is a motivation for regular Expunges. Section 4 contains more information on this topic.

Finally, clicking the Destroy button causes Walnut to commit and then close the Walnut database. This is a good thing to do before booting or rolling back; it reduces the possibility of mangling the database.

Walnut registers several infrequently-used commands with the Cedar Executive. All of the command names contain the prefix "Walnut", so typing Walnut*? to the Executive enumerates them. The Walnut command creates a Walnut control viewer if you should happen to Destroy yours. The WalnutExpunge command has the same effect as clicking Expunge, but can be used when the control viewer does not exist. The WalnutOldMailReader command is described in Section 3, and the WalnutScavenge command is described in Section 4. WalnutNewMail simulates clicking the NewMail button.

3. The log

Walnut keeps a record of all retrieved messages and all database updates for a single user in a *Walnut log file.* This is a text file with a very simple format (an extension of the **.mail** format used by Laurel and Hardy); load your Walnut log into a Tioga viewer to see this. (Since your Walnut log be a large file, you may wish to use the **OpenHuge** command to load it into a Tioga viewer.)

The important point is that the truth about your Walnut mail resides in a Walnut log file; the Cypress database that Walnut uses for query processing can always be reconstructed by replaying a Walnut log file. The Walnut log mechanism is very robust, which makes Walnut's mail storage quite reliable even if Walnut (or some other part of Cedar) crashes.

Walnut locates the log file for a particular user by consulting the Walnut.WalnutLogFile entry of the user profile. If there is no such entry, the name **Walnut.DBLog** is assumed. At present, the log file must reside on the Cedar workstation's local disk. This means that to use Walnut on a public Dorado, you must copy the log to the Dorado from a file server, then start Walnut (**Walnut** command to the Executive). At the end of a public Dorado session you must stop Walnut (**Destroy** button), then copy the log to a file server. Soon, Walnut will be capable of accessing a Walnut log stored on an Alpine file server, just as it can access Cypress databases today.

A Walnut log grows with each retrieved message and database update until an **Expunge** command is given to the Executive. This command writes a new Walnut log that only contains information about the messages that have not been deleted, and updates the database to be consistent with this. (Note that this requires disk space for two copies of the log.) Because the cost of a full Expunge is proportional to the size of your Walnut log (which can grow to be quite large, if you are a "pack rat"), there is a "short cut" Expunge that only deletes messages from the point in the log of the previous Expunge; it is enabled by setting the Walnut.EnableTailRewrite user profile option. When enabled, LEFT-clicking **Expunge** causes the short-cut expunge to be performed, while RIGHT-clicking performs the full expunge. Note that with the tail-rewrite expunge any quite old messages that you delete (which appear before the expunge cut-off point) will not be removed from the log or database; thus, once in a while you will still need to do a full expunge.

Jim Morris's advice concerning files on the local disk is to "keep your bags packed". A prudent individual will apply this philosophy to his Walnut log. Include this file in a personal .df file (such as the one that contains your user profile), and make it a habit to SModel it every few days.

4. Becoming a User

4.1 Standard usage

First, bring over the latest Walnut by typing

```
Bringover /a /p <Cedar>Top>Walnut
```

to the Executive; this will also retrieve WalnutSend, Cypress, and AlpineUserImpls. These latter files will be loaded by Walnut (AlpineUserImpls only if needed).

Edit your personal profile to contain all of the entries specified in WalnutDefault.profile (public in Walnut.df). The only profile entry that most users will want to experiment with is "InitialActiveRight: TRUE"; making it FALSE causes the Active message set displayer to create itself in the left viewer column, like all other message set displayers.

To start Walnut, type

```
Walnut
```

to the Executive. This will spend a long time loading, but finally a Walnut control viewer will appear.

You can include Walnut in a checkpoint. Be sure not to click checkpoint until the message "Walnut 4.2" appears in the Walnut control viewer typescript. Message and Message set displayers get updated after each rollback; if a displayed message or message set has since been deleted, the viewer will be destroyed.

To read a Laurel or Hardy mail file, or a file created by Walnut's Archive operation, first run Walnut as just described. Then type

```
WalnutOldMailReader <complete mail file name> {optional message set name}
```

to the Executive. If you fail to specify a message set name, the messages will be placed in Active. If the specified message set does not exist, it will be created.

4.2 Using Alpine for Walnut database storage

The Alpine server "Luther.alpine" can be used to store Walnut databases. Using Alpine improves Walnut performance somewhat, especially for operations that write to the Walnut database. It frees up space on the local disk that is otherwise occupied by the Walnut database. It also reduces the cost of using Walnut on a public machine (you move only the log to the new machine, not the log and database.) The drawback of using Alpine is that on occasion it may abort Walnut's transaction; Walnut will recover from this gracefully, but it may take some time to replay your uncommitted actions stored in the log. Transaction aborts are infrequent, so on balance the Alpine server is an improvement over the local disk. Contact Karen Kolling to obtain an Alpine account.

Using Alpine changes the procedure for Walnut installation only slightly.

Edit your personal profile to contain the entries:

```
Walnut.WalnutSegmentFile: "[Luther.alpine]<YourName.pa>Walnut.segment"
Walnut.WalnutLogFile: "YourName.WalnutDBLog"
```

With your profile in this state, you must run AlpineUserImpls before invoking the Walnut command. One way of ensuring this is to include the items

```
Run AlpineUserImpls; RunAndCall Walnut
```

in your CommandsFrom: profile entry. It is ok to take a checkpoint after running AlpineUserImpls and Walnut.

There are two ways to create a Walnut database on Alpine. The first is to follow the procedure above, in which case Walnut will notice the absence of a database and create one by scavenging from the log. The second is to copy the database from the local disk to the Alpine server, using the procedure call

```
← AlpineCmds.Copy[to: "[Luther.alpine]<YourName.pa>Walnut.segment", from: "Walnut.segment"]
```

You must run AlpineUserImpls before attempting to call this procedure. For more information on Alpine operations consult [Indigo]<Cedar>Documentation>AlpineDoc.*.

5. Coping with Releases and Crashes

Walnut sometimes crashes because its database has gotten into a bad state. Also, a new release of Walnut or the Cedar database system will occasionally change the database format that Walnut understands. From Walnut's point of view these circumstances are very similar.

A Walnut database can always be reconstructed by replaying a Walnut log file. If Walnut is not loaded, you can reconstruct the Walnut database by executing the command file WScav.cm, which is included in Walnut.df; when the command file completes, you will be running Walnut just as if you had used Walnut.cm. If Walnut is already loaded, you can scavenge by typing **WalnutScavenge** to the Executive. Walnut will also scavenge automatically if the database cannot be found.

6. User profile options

Below is a complete list of all of the current Walnut user profile options (copied from UserProfile.doc):

Walnut.ReplyToSelf: BOOL ← FALSE;
if TRUE, *causes walnut to automatically supply a Reply-To: field, if appropriate.*

Walnut.DestroyAfterSend: BOOL ← FALSE;
if TRUE, *causes sender to be destroyed after a successful delivery, if Send was clicked with RIGHT*

Walnut.InitialActiveRight: BOOL ← TRUE;
true says to bring up the active message set on the right column, false on left.

Walnut.InitialActiveOpen: BOOL ← FALSE;
true says open a message set viewer on Active.

Walnut.InitialActiveIconic: BOOL ← FALSE;
if true and InitialActiveOpen = TRUE, then the Active message set viewer is opened as an icon.

Walnut.MsgSetButtonBorders: BOOL ← FALSE;
if TRUE, puts borders around the MsgSet buttons in the control window.

Walnut.EnableTailRewrite: BOOL ← FALSE;
if TRUE, performs "tail rewrite" on Expunge.

Walnut.WalnutSegmentFile: TOKEN ← "Walnut.Segment";
value is the name of the file to be used for the walnut data base.

Walnut.WalnutLogFile: TOKEN ← "Walnut.DBLog";
Name of log file.

7. Shortfalls and Wishes

What follows is a listing of known deficiencies and contemplated extensions to Walnut. Nobody guarantees that everything listed below will be implemented. But the list does indicate some directions for future work, and may provide context for your own Walnut wishes. Send both bug reports and wishes to WalnutSupport↑.

7.1 Message sets

It would be nice to allow selection of more than one message in a message set (perhaps even spanning message sets).

When a message set displayer is created, it should display the newest messages first (perhaps by painting from the bottom of the viewer towards the top), since these are most likely to be accessed.

7.2 Retrieving mail

It seems desirable to make mail retrieval a continuous background activity. This would tend to insulate users from the response time of Grapevine.

Once mail retrieval is implemented in the background, a natural next step is to provide some means for a user procedure to classify incoming mail according to its significance, file it in sets other than Active, let the user know the status of his new mail ("You have *important* new mail").

The procedure that stores new mail in the database should understand the In-Reply-To relationship. Eventually, users should be able to write queries or other commands that exploit this relationship.

7.3 Sending mail

It should be possible to forward or answer multiple messages. This seems to require the ability to select multiple messages.

Feedback from mail parse errors can be improved. When displaying a message containing a parse error or bad a recipient, the point of error in the message header should be highlighted.

When sending a sequence of messages with the message composition viewer, you tend to click Send, wait for the feedback "sending ...", then make the viewer iconic (to reclaim the screen area) and finally click NewForm to create a new viewer. It would be smoother to reuse the same message composition viewer, but without waiting for the message to be sent (since this can take quite awhile). Since it is quite unusual to have two messages in transit (as contrasted with two messages being composed) at the same time, this can be achieved by passing responsibility for the message from the message composition viewer to the Walnut control viewer when message parsing is complete, and clearing the composition viewer for reuse. Any errors in transmission would be reported in the control viewer rather than the composition viewer.

7.4 Global operations

Walnut needs a way to make queries. (This is what databases are all about!) For starters, we'd like to have something analogous to Laurel's SearchMail program for performing text pattern matching in the messages of a message set.

The Walnut Expunge operation currently requires more resources (such as disk space) than Walnut requires to perform other operations on the same database. This is unfortunate, since it means that a user can get stuck in a situation where he must hand-edit his Walnut.DBLog in order to recover.

Cedar Language Overview

Version 4.2

Release as [Indigo]<Cedar>Documentation>Overview.tioga, .press
Came from [Indigo]<CedarDocs>Manual>Overview.tioga, .press
Last edited by Horning on June 1, 1983 6:48 pm

Abstract: This Overview is intended to introduce you to the basic vocabulary and concepts that you need before plunging into sources of more detailed information about the Cedar Language. **It assumes that you have already read the Briefing Blurb and the Introduction to Cedar. If you haven't, read them first and return.** It starts with a brief review of the common concepts that Cedar shares with other members of the Pascal family, then gives a somewhat less hasty tour of the more novel features of Mesa, followed by a discussion of the additional changes that produced Cedar. Finally, there is a guide to sources of further information.

Version 4.2 of the Cedar language documentation corresponds to Release 4.2 of the Cedar system. It is intended to supersede all descriptions prior to June 1983. Previous documents may be read for historical interest, but are believed only at the reader's peril.

[If you are reading this document on-line, I suggest that you use the Tioga Levels and Lines menus to initially browse the top few levels of its structure before reading it straight through.]

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Cedar Language Overview: Contents

Introduction

The programming language of the Cedar Programming Environment (hereafter, Cedar Language, or just Cedar) has resulted from an evolutionary process in PARC and SDD that spanned more than a decade. Understanding what the language is, and why it is that way, may be somewhat easier with a little historical background:

> Mesa is a system implementation language in the "Pascal family," with extensive facilities for modularization and separate compilation, processes and monitors, exceptional-condition handling, and control of low-level hardware functions. It was initially designed and implemented in the PARC Computer Science Laboratory, primarily by Butler Lampson, Chuck Geschke, Jim Mitchell, Ed Satterthwaite, and Dick Sweet. Subsequently, the OSD System Development Department assumed responsibility for development and maintenance. It has gone through a series of releases.
>
> When CSL launched the Cedar Project in 1979, it chose to use the Mesa language and system as a starting point. (Mesa 6, 7, and 8 are its closest relatives.) However, Mesa did not have a few of the features that seemed to be important for an experimental programming environment, so some extensions and changes were designed. The major changes resulted from adding automatic storage deallocation (garbage collection) and facilities for delaying the binding of type information, without sacrificing complete type-checking in either case.

This Overview is intended to introduce a competent programmer to the basic vocabulary and concepts that are needed before plunging into sources of more detailed information about the Cedar Language. It assumes that you know some other language in the Pascal family. **It also assumes that you have already read the Briefing Blurb and the Introduction to Cedar. If you haven't, read them first and return.**

This Overview starts with a brief review of the common concepts that Cedar shares with other members of the Pascal family, then gives a somewhat less hasty tour of the more novel features of Mesa, followed by a discussion of the additional changes that produced Cedar. It ends with a survey of sources for further information.

This Overview does not provide the detail you need to actually write Cedar programs. (In particular, the reference grammar is included but not discussed.) But when you finish reading it, you should have a fair acquaintance with Cedar terminology and concepts, and you should have a good idea of what you need to learn. Different things are discussed in varying depth; generally the long discussions cover things that you should plan to study carefully.

Cedar documentation is still evolving. Comments and suggestions on how it can be made more useful are welcome at any time. Although we plan a systematic attempt to assess the effectiveness of the various kinds and pieces of documentation, you should not wait until asked to let us know what you think about it.

Various proposals and descriptions of interim implementations from September 1979 onward have been given labels such as 5C1, 5C2, 6C2, 6C5, 7T11, and Version 3. **Version 4.2 of the Cedar language documentation corresponds to Release 4.2 of the Cedar system. It is intended to supersede all descriptions prior to June 1983. Previous documents may be read for historical interest, but are believed only at the reader's peril.** This Overview has been compiled by Jim Horning; errors and sources of confusion should be reported to him. Most of the contents have been abstracted from previous documents, with a small amount of editing and validity checking.

Review of the Pascal-like features

The following summarizes aspects of Cedar (and Mesa) that are basically similar to those of other members of the "Pascal family" of languages (e.g., Euclid, Modula, Ada). If there are any concepts in this section that are not already familiar to you, you should probably find a Pascal textbook and study it before proceeding to further material on Cedar. (You will find that the *names* for these concepts vary somewhat from language to language.)

An algorithm or computer program consists of two essential parts, a description of *actions* that are to be performed, and a description of the *data* that are manipulated by these actions. Actions are described by *statements*, and data are described by *type definitions*.

Data and types

Data are represented by *values*. Values are *immutable;* they are not changed by computation. A *constant* always denotes the same value within a scope. A *variable* is a value that may *contain* another value; assignment changes the value contained by a variable, but not the value that *is* the variable.

A value used in a program may be represented by a *literal constant*, the *name* of a constant or variable, or by an *expression*, which will itself contain other values. Every name occurring in the program must be introduced by a *declaration*. A declaration associates with a name both a data type and a constant value (which may itself be a variable, and *contain* different values at different times).

A *data type* defines both a set of values and the actions that may be performed on elements of that set. It may either be directly described in a declaration that uses it, or it may be referenced by a type name, introduced in a *type declaration*. The type of every constant, variable, and expression can be deduced from static analysis. This analysis is performed by the compiler to ensure that all programs are type-correct; thus the language is said to be *strongly typed*.

An *enumerated* type definition indicates an ordered set of values, i.e., introduces names standing for each value in the set. The *simple* types are the enumerated types, the subrange types, and the *built-in types*, including BOOL, INT, REAL, and CHAR. There are standard denotations for literal constants of the built-in types: TRUE and FALSE for BOOL, numbers for INT and its subranges and for REAL, quotations for CHAR. Numbers and quotations are syntactically distinct from names—as are the "reserved words" of the language. The set of values of type CHAR is an 8-bit variant of the ASCII character codes.

A type may be defined as a *subrange* of a simple type by indicating the smallest and largest value of the subrange.

Structured types are defined by describing the types of their components, and indicating a *structuring method:* ARRAY or RECORD. These differ in the mechanism for selecting a component of a value.

In an *array structure*, all components are of the same type. A component is selected by a computable selector, or *index*. The index type, which must be simple, is indicated in the array type definition. It is usually a programmer-defined enumerated type, or a subrange of INT. Given a value of the index type, an *array selector* yields a value of the component type. Every array structure value can therefore be regarded as a mapping of the index type into the component type.

In a *record structure*, the components (called *fields*) are not necessarily of the same type. In order that the type of a selected component be evident from the program text (without executing the program), a record selector is not a computable value, but must instead be a name uniquely denoting the component to be selected.

A record type may be specified as consisting of several *variants*. This allows different record values of the same type to have structures that differ in the number of components, their types, or their names. The variant describing a particular value is indicated by a special field, called its *tag*. Variants of a type may also share fields in addition to the tag.

An explicit variable declaration associates a name and a *static* variable; the name is used to denote the variable in expressions. *Dynamic* variables are generated by a special procedure (NEW) that yields a *pointer* or *reference* value that subsequently serves in place of a name to refer to the variable. Finite graphs in their full generality may be represented using pointers or references.

Statements

The simplest statement is the *assignment statement.* It specifies that a newly computed value be assigned to a variable (or a component of a variable). The value is obtained by evaluating an *expression.* Expressions consist of variables, constants, operators, and procedure values operating on arguments to produce new values. Constants are literal or declared: variables and procedures are built-in or declared; the set of *operators* is defined within the language, and includes operators for arithmetic, comparison, and logical operations.

The *procedure statement* causes the *application* (invocation, call) of a designated procedure value to the values of its *arguments* (actual parameters).

Basic statements are the components of *structured statements,* which specify sequential, selective, or repeated execution of their components. Sequential execution of a sequence of statements is specified by separating them by semicolons; conditional or selective execution by the *if statement* and the *select statement;* and repeated execution by *loop statements.*

A *block* can be used to associate declarations with statements. The names so declared have significance only within the block. Hence, the block is the *scope* of these names, and they are said to be *local* to the block. Since a block may appear as a statement, scopes may be nested.

A block can be the *body* of a *procedure value.* A procedure has a fixed number of *parameters,* each of which is denoted within the procedure by a name called the *formal parameter.* Actual argument values are supplied for parameters at each application.

Procedures may also have *results;* applications of such procedures may appear within expressions.

From Pascal to Mesa

Mesa extended Pascal in a number of directions intended to make it more effective for the development of large systems. Students of programming languages will discern influences from Algol 68, BCPL, and several other system implementation languages. It is a larger language, and is rather more difficult to master in its entirety, than Pascal. It is intended for professional programmers, not for beginning students.

Mesa *modules* are separately compiled program units, with type-checking preserved across module boundaries. Mesa provides mechanisms for systematic handling of *exceptions*, *processes* and *monitors*, procedures as first-class values that can be assigned to variables, and a fair number of syntactic and semantic amenities intended to make programming more convenient.

The following sections introduce each of the major conceptual extensions, but do not explain them in great depth. See [Geschke, *et al.*] for a more extensive rationale, and CSL-79-3 for full details.

Modules

Mesa modules are a "programming in the large" mechanism for partitioning a system into manageable units. They can be used to encapsulate abstractions, to provide a degree of protection, and to enforce "information hiding." They are also the units of separate compilation.

There are two kinds of modules: *DEFINITIONS* modules, which define *interfaces*, and *PROGRAM* modules, which contain the executable code to *implement* these interfaces.

Definitions (or defs) modules define interfaces to abstractions. They typically declare some shared types, useful constants, and the domains and ranges of a set of procedure names. They compile into *symbol tables*, which are shared by both *clients* and *implementations*. Checks are performed when modules are *bound* into a *configuration* to ensure that separately compiled pieces have used consistent versions of the shared definitions. *Interfaces* produce no executable code; they manifest themselves at runtime primarily as symbol tables that are accessible for debugging and similar purposes.

Program modules provide implementations of abstractions. They typically declare collections of variables that define their state and provide bodies for the procedures of their interfaces. Viewed as source text, they are similar to Pascal procedures and Simula class definitions. They can be loaded and interconnected to form complete systems.

At runtime, one or more *instances* of an *implementation* may be created. A separate *global frame* (activation record) is allocated for each, containing storage for its *global variables* (those which are declared outside its procedures), which *persist* between applications of its procedures. The lifetimes of implementation instances (unlike those of procedure applications) are not restricted to follow any particular discipline. Communication paths among implementations are established dynamically and are not constrained by any (static or dynamic) nesting relationships; lifetimes and access paths are completely decoupled. The module body itself generally contains the code to initialize the global variables and establish any necessary invariants. It will be executed when the module is *started*, or upon application of one of the module's procuedures, whichever comes first.

A module that *accesses* (relies on declarations from) other modules must include DIRECTORY statements, so the necessary symbol tables can be acquired. If it uses only a subset of the declarations, it is good practice to indicate which ones with a USING list. Declarations in an interface are *public* unless declared to be PRIVATE. Normally the importing module accesses only the public names; private declarations may be accessed by implementing modules that indicate they SHARE the interface. A directory statement may list the name of a file containing the symbol table to be used, but if the file name is the same as the module name (except for the extension .bcd) it is omitted.

A module that uses non-constant declarations (e.g., exported types and procedures) from another module must explicitly *import* it. If a module implements any part of an interface (e.g., by supplying the value of a procedure or type that it declares), it must explicitly *export* it. The compiler will check that its PUBLIC declarations are type-consistent with the corresponding declarations in the exported interface(s).

Each module is effectively parameterized by a set of *interface records*, one for each interface it imports, and supplies a set of *export records*, one for each interface it exports. Note that interfaces and implementations need not be in one-to-one correspondence. Binding a group of modules together into a *configuration* involves assigning values from the export records to the corresponding fields in the interface records. There is a special sublanguage, C/Mesa, to control this process.

Accessing other modules introduces compilation order dependencies. Each module must be compiled *after* the modules it accesses (and recompiled if they change), since the compiler needs their symbol tables. But information does not flow in the other direction. Modules that are not accessed by others (virtually all implementations) may be freely recompiled without invalidating previous compilation and checking of any other modules.

Types, as well as procedures, can be declared *opaquely* in interfaces and subsequently bound to concrete values supplied by implementations. This makes the internal structure of the type invisible to clients of the interface, and ensures that there can be no compilation dependencies between the definition of the concrete type and the interface module. The definition of the type can be changed at any time without requiring recompilation of the interface or any clients of the interface.

Effective use of Mesa requires a thorough understanding of modules and their use. They have significantly influenced our program design and construction techniques.

Programs are almost never self-contained modules; the importation and re-use of existing code has all the advantages of theft over honest toil—without the moral stigma. Considerable emphasis is laid on the careful design of interfaces, and on their documentation. Since it is only interface changes that force recompilation (or perhaps even rewriting) of client programs, it is important that interfaces remain stable for substantial periods, even while their implementations are undergoing change.

A recommended approach is to define, comment, and circulate for review, all of the interfaces in a (sub)system before writing any of the implementations. Interfaces play much the same role as "program design languages" in other environments, with the additional advantages of being precisely defined and mechanically enforced.

The Mesa language definition omits many of the features commonly expected in programming languages, such as input/output and string-manipulation operations. Of course, these facilities are available to Mesa programmers, but they are provided by packages written in the language itself. The descriptions of standard packages in the Mesa Programmer's Manual, Version 8.0, run to more than 300 pages.

When managing large collections of modules (and in systems like the Mesa Development Environment and Cedar they run into the thousands), module names become very important. The use of cryptic or acronymic names is discouraged. By convention, source file names have the extension .mesa, and object file names have the extension .bcd (for Binary Configuration Description). The definitions module for an interface X is customarily named X; if it is implemented by a single program module, that is customarily named XImpl.

Exceptions

Mesa provides a way to indicate when exceptional conditions arise in the course of execution and an orderly means for dealing with them that is inexpensive if they do not arise. *Exceptions* cause a transfer of control from the statement that *raises* them to a dynamically-selected part of the program intended to *handle* the situation. They may be raised in response to the detection of "impossible" situations, invalid inputs, the inability of an abstraction to supply its specified service, or simply unusual events.

Mesa exceptions are conceptually similar to procedures, except that the binding to the handler is determined by searching the *catch phrases* in the call stack of the process in which the exception is raised; the dynamically innermost handler that *accepts* the condition is applied. Like normal procedures, handlers can take parameters and return values. They are written in a distinctive syntax that clearly identifies them as code for the exceptional case.

Catch phrases are syntactically and semantically similar to SELECT statements, with test items indicating the exceptions for which the associated handler should be applied. There are special test items to catch arbitrary exceptions and to catch an attempt to *unwind* the application stack in response to an exception. A series of catch phrases may be associated with a procedure application, or *enabled* throughout a block.

A handler is like a procedure body, but when it completes, there are a number of additional control options: GOTO, EXIT, LOOP, RETRY, CONTINUE, REJECT, and RESUME. Resumption is analogous to returning from a procedure, possibly with a result. Exceptions are divided into SIGNALs, which may be resumed, and ERRORs, which may not; in common parlance they are generally all called signals.

Since handlers may take parameters and return results, each exception name must be declared in a scope that includes all the points where it is raised as well as all the catch phrases that accept it.

The cost of raising an exception is significantly higher than the cost of procedure application, but it shouldn't happen very often. The system guarantees that all exceptions are handled at some level; those that the program fails to catch are accepted by the debugger, keeping intact the state of the program that raised it.

Exceptions can be used in very intricate ways to achieve subtle effects (e.g., by raising another exception within a handler). Experience has shown that this is almost always a mistake. Some call it elegance, others call it incomprehensible:

> "For the programmer, the main import of nested signals is that one needs to consider, when writing a routine, not only what signals can be generated, directly or indirectly, by the called procedures, but also those which can be generated by catch phrases in that procedure or even the catch phrases of any calling procedures, also both directly and indirectly." [*Mesa Language Manual*]

Although his language proposals have not been implemented, Roy Levin's discussion in the working paper [Indigo]<CedarDocs>Style>SignallingGuidelines.press is the best source of guidance on tasteful and appropriate uses of exceptions. The most important point is that the exceptions a procedure may raise must be considered part of its interface, and documented as such. Unfortunately, the compiler currently doesn't enforce this, and many otherwise excellent interfaces do not comply.

Processes, monitors, and condition variables

Mesa provides efficient mechanisms for concurrent execution of multiple *processes* within a single system. This makes it natural to structure programs to reflect their inherent concurrency. Mesa also provides facilities for mutually exclusive access to resources and process synchronization by means of entry to *monitors* and waiting on *condition variables*.

FORK makes it possible to start the execution of another procedure concurrently with the program that applies it. It returns a process, which may either be *detached* to proceed independently, or saved for a future JOIN. There is no rule against multiple coexisting instances of a procedure, either forked or applied, although care must be taken to ensure mutual exclusion on accesses to shared global data.

JOIN takes a single process argument. When the forked procedure has executed a RETURN *and* the JOIN has been executed (in either order), the returning process is deleted, and the joining process receives its results and continues execution. A process type is declared similarly to a procedure type, except that only the type of the result is specified.

All processes execute in the same address space. This means that they are not protected from each other, which is presumably acceptable in a single-user system. It also means that process creation and switching between processes is cheap (not much more time-consuming than a procedure call).

Generally, two or more cooperating processes need to interact in more complicated ways than simply forking and joining. The interprocess synchronization mechanism provided in Mesa is a variant of "monitors" adapted from the work of Hoare, Brinch Hansen, and Dijkstra. The underlying view is that interaction among processes is always based on access to shared resources (e.g., data) and that a proper vehicle for this interaction must unify the synchronization, the shared data, and the procedures that perform the accesses.

A monitor is typically a module instance, with shared data in its global frame, and its own procedures for accessing them. Some of the procedures are public, allowing applications of monitor procedures from outside. Obviously, conflicts could arise if two processes were executing in the same monitor at the same time. To prevent this, a *monitor lock* is used for mutual exclusion. Application of one of a monitor's ENTRY procedures automatically acquires its lock (waiting if necessary), and a return releases it. An integrity constraint that the programmer imposes on the monitor's data is called a *monitor invariant*. The lock makes it possible for the programmer to ensure that this invariant will be true whenever an entry procedure begins execution—regardless of what is happening in various processes—simply by making sure that it is true initially and that *every* entry procedure restores it before returning.

Of course, a process may enter the monitor and find that the monitor data is in a good state but indicates that the process may not proceed until some other process enters the monitor and changes the situation. The WAIT operation allows a process to release the monitor lock temporarily (and suspend execution) without returning. The *wait* is performed on a *condition variable*, which is associated by agreement with the actual condition needed. After making a change that may have changed the condition, some other process must perform a BROADCAST or NOTIFY on the condition variable; this allows a waiting process to reacquire the lock, retest the condition, and resume execution if it is true. Note that since a wait releases the lock, the monitor invariants must be restored before waiting.

The procedures of a monitor are classified as *entry*, *internal*, and *external*. Internal procedures may only be applied by entry or internal procedures of the same monitor, since they are intended to be executed within the monitor's mutual exclusion, but do not acquire the monitor lock. External procedures are logically outside the monitor, but are declared within the same module for reasons of logical packaging. Being outside, they must *not* reference any monitor data nor apply any internal procedures; they are often used to provide a convenient interface that "hides" one or more applications of entry procedures.

The attributes ENTRY and INTERNAL are associated with a procedure's body, not with its type; thus they do not appear in interfaces. From the client side of an interface, a monitor appears like any other module.

In simple cases, a monitor's data comprises its global variables, protected by an implicit lock that is automatically allocated in its global frame. However, many applications deal with multiple *objects*, represented, say, as records accessed through pointers. It may be necessary to ensure that operations on these objects are *atomic*, i.e., once the operation has begun, the object will not be otherwise referenced until the operation is finished. It is possible to associate a lock with the object, rather than with the module's global frame, by declaring the data as a MONITORED RECORD. A single module instance can then implement each operation as an entry procedure, taking the object as a parameter. Locking is specified in the module heading by a LOCKS clause.

A somewhat subtle source of deadlocks occurs if control leaves an entry procedure by means of an uncaught exception. Unless it is certain that all exceptions (including those raised by invoked procedures) are handled, each entry procedure should include an UNWIND catch phrase, which will implicitly release the monitor lock.

Control constructs

Mesa's facilities for ordinary sequential "programming in the small" are extensive, but fairly conventional. The syntax is not exactly like that of any other language, but for the most part it can be picked up easily with a few minutes study of the grammar. (In fact, since most program text is produced either by editing existing programs or by the use of the Tioga editor to expand syntactic templates, you may be able to just "fake it.") This section mentions a number of areas where Mesa provides "convenience" extensions or conceptually small changes.

SELECT statements generalize Pascal's "case" construct by allowing several ways to specify how one statement is to be chosen for execution from an ordered list. The most common form is based on the relation between the value of a given expression and those of expressions associated with each selectable statement. The relation may be equality (the default), any relational operator appropriate to the types of the values involved, or containment in a subrange. A single selection may be prefixed by several selectors, and an optional ENDCASE statement is selected only if none of the others are. *Discriminating* selection is used to branch on the type of a variant record value (and in Cedar, on the current type referred to by a REF ANY). SELECT expressions are analogous, but choose from an ordered list of expressions.

Iteration is provided by loop statements in which several different kinds of control can be freely intermixed. A loop has a *control clause* and a *body*. The control clause may specify a logical condition for normal termination, possibly combined with a range or a sequence of assignments for a *controlled variable*. In addition to ordinary statements, the body may contain EXIT or GOTO statements to explicitly terminate its execution, and may be followed by a REPEAT clause that acts like a selection on the GOTO used to terminate the loop. (GOTO *cannot* be used to synthesize arbitrary control structures. It is much more like a "local" exception.)

In Pascal, procedure execution must proceed somehow to the end of the body before terminating; in Mesa, it can be terminated anywhere by executing a RETURN statement. If the procedure's type includes results, the return statement may supply the values to be returned—otherwise they are taken from the result variables named in the type. Each procedure body is followed by an implicit return.

Pascal procedures are not values that may be assigned to variables; Mesa procedures are. In most cases, the programmer still thinks of a constant association between a procedure name and its body, but to truly understand what is going on when interface records are bound, it helps to realize that procedure values from the export records are being assigned to appropriate fields of the interface records. This same power is available to the Mesa programmer; one popular form of "object-oriented programming" is based on the creation of an explicit record of procedures for each kind of object, and passing around together a pair of pointers, one to the procedure record, and another to the object instance data.

INLINE procedure constants may be declared in interfaces or locally. This is an instruction to the compiler to expand the procedure body inline for each application, rather than compiling a call to out-of-line code. It is intended to improve the speed *without* changing the semantics of the procedure—inlines are not macros. INLINE should be considered a form of tight binding best reserved for late stages of system tuning; among other things, it can cause the compiler to run out of resources, even when compiling what appear to be small modules.

In addition to procedures and exceptions, Mesa has a third mechanism for transfer of control, called a PORT. When used in pairs, ports can provide a very general form of *coroutine* implementation. In some circumstances, coroutines have advantages similar to processes, at slightly lower cost, but they are not used much in Mesa or Cedar.

Miscellaneous

Every expression in a Mesa program has a *syntactic type* that can be deduced from its structure by static analysis of the program text, a process called *type determination*. The language imposes constraints on the type of each expression according to the context in which it is used, even in separately compiled modules.

The syntactic type of a name is established by declaration.

The form of a literal implies its type.

Each operator produces a result with a type that is a function of the types of the operands.

The type rules in Mesa take two general forms:

The type required by the context is known exactly, and a given expression must have it. The required type is called the *target type*. Examples occur in assignment, initialization, record construction, array construction, argument list construction, and array subscripting. Several *coercions* (e.g., pointer dereferencing, base/subrange conversion, single-component record to field) will be applied if needed to convert a value whose syntactic type is not its target type to one that is.

The exact type is not implied by context, but a relation that must be satisfied by a set of types is known. The process of finding types to satisfy that relation is called *balancing*. Examples include generic operators (such as relationals) that require two operands of the same type, conditional expressions, and select expressions. The common type selected will be the one requiring the fewest coercions.

A *sequence* in Mesa is an indexable collection of items, all of which have the same type. In this respect, a sequence resembles an array; however, the length of the sequence is not part of its type. The (maximum) length of a sequence is specified when the object containing that sequence is created, and it cannot subsequently be changed. It is the responsibility of the programmer to keep track of the number of items in the sequence at any time.

Mesa allows a default initial value to be associated with a type. If a type is constructed from other types using one of Mesa's structures, such as RECORD, an implicit default value for the constructed type is derived from the default values of the component types, but it can be overridden with an explicit default value. Default values for arguments can simplify procedure applications; default fields of records make the corresponding constructors more concise and more convenient; initial values are useful to ensure that the corresponding storage is always well-formed, even before the variable has been used by the program.

Dynamic variables in Mesa are allocated in *zones*. These are not necessarily associated with fixed areas of storage; rather, they are objects characterized by procedures for allocation and deallocation. There is a standard system zone, but programs that allocate substantial numbers of similar dynamic variables can often improve performance by segregating each kind into its own zone. The operator NEW is used to create a dynamic variable in a zone, and FREE to release it.

The MACHINE DEPENDENT attribute allows precise control of the representation of values at the bit level.

From Mesa to Cedar

The Cedar Language is very closely related to Mesa. The most radical change is the provision of automatic deallocation of dynamic storage, or *garbage collection*. Several other changes extend the range of *binding times* available for such important attributes as the types of variables.

It is intended that most Cedar programs will be written in the *safe subset*, which imposes a number of restrictions not present in Mesa to ensure the safe operation of the garbage collector, and introduces some new (safe) features to make these restrictions tolerable. The full (unsafe) language is generally "upward compatible" with Mesa.

Garbage collection, collectible storage, and REFs

Although Mesa pointers are typed, they provide a rich source of opportunities for creation of safety problems, including the classical *dangling pointer* problem, where a pointer is used after the storage it refers to has been deallocated, and the opposite *storage leak* problem, where storage becomes inaccessible without being deallocated for reuse. Freeing the programmer from responsibility for deallocating storage at just the right time was a major goal of Cedar. It adds a new class of REF types that are just like the corresponding pointer types except that the system is responsible for freeing the dynamic variables they refer to *after* they have become inaccessible.

Cedar provides three types of storage:

Frame: This is storage that is implicitly allocated by a procedure application or an implementation instantiation to hold variables declared in the corresponding scope. It is also implicitly deallocated, upon exit from the scope (e.g., return from the procedure).

Collectible: This is storage that is explicitly allocated by NEW, and implicitly deallocated after there are no more accessible REFs to it. FREE applied to a REF variable will cause it (and REF fields in the dynamic variable it refers to) to be "NILed out," but the dynamic variable will only be freed when no other REFs to it remain.

Heap: This is storage that is explicitly allocated by NEW, and deallocated by (unsafe) FREE statements, as in Mesa. Heap storage is referenced by pointers, which may not be dereferenced in checked regions, and should not refer to dynamic variables containing REFs.

The introduction of collectible storage has substantially revised programming style and interface design in Cedar. When the project was being contemplated, some Mesa programmers indicated that as much as 40% of their time went into designing and checking the code to avoid dangling pointers and storage leaks, to tracking errors in this code, and to wasting time in tracking other errors by suspecting storage deallocation problems. With REFs and a reliable garbage collector that all goes away.

Frame (static) variables are still less expensive than dynamic variables, since entire frames are allocated and freed on procedure entry and exit (and the mechanism for doing it has been rather carefully tuned). However, it is entirely reasonable to use dynamic variables for data whose lifetime is not closely connected to a particular procedure application or module instance. Objects of large or varying size are almost always passed across interfaces *by reference*. Definitive measurements on the cost of garbage collection have not yet been made, but preliminary data indicates that it is generally less than 20%. Only in very special circumstances is heap storage worth the added program complexity and potential for errors.

Safety

A desirable property of a high-level language system is *implementation independence*. This means that the effects of (even erroneous) programs can be understood in terms of the *language*—rather than requiring an understanding of the particular implementation. Mesa comes rather close to meeting this goal (as evidenced by the fact that most Mesa debugging can be done "at the Mesa level," without ever worrying about the format of frames or the details of storage management), but it does contain some *unsafe* features whose use can lead to messy implementation dependencies.

It was desirable on general grounds to reduce implementation-dependence in Cedar. However, the decision to include facilities for garbage collection made it imperative. A collector can cause storage to be deallocated (permitting its subsequent reallocation and re-use) at times that are completely unpredictable from examination of the source program. A single programming error that smashes a REF used by the collector can destroy data structures in ways that make it difficult to reconstruct any evidence of the original cause of the crash.

A major goal for the Cedar Language was that it contain a useful subset for which garbage collection would be safe. The *safe subset* of Cedar is basically that part of the language where even incorrect programs cannot interfere with the reliable operation of the collector. The vast majority of Cedar programs should be written primarily (or entirely) in the safe subset. Safe Cedar does not provide acceptably efficient substitutes for every use of Mesa's unsafe features, so Cedar provides a means for indicating that some regions of a program are *trusted.* This inhibits compiler enforcement of the safety restrictions and indicates that the programmer has assumed the additional responsibility of ensuring that these regions of the program do not violate the integrity of the system.

Invulnerability, safety, and checking

It is an obviously desirable property of a programming system that no user programming error can "break" its *abstract machine* and reduce its world to a rubble of bits. We call this property *invulnerability.* In general, it can be ensured only by maintaining the integrity of certain data structures known to the runtime system. Collectively, the properties that must be maintained to ensure invulnerability are called the *safety invariants;* each part of the system is responsible for ensuring that they are not destroyed, and must assume that the rest of the system does likewise.

Unfortunately, invulnerability is not a local property. If any part of the system fails to maintain the invariants, the entire system (including programs that are themselves correct) is potentially vulnerable. We use the term *safety* for the property that the invariants cannot be invalidated locally, even by incorrect programs. Cedar operations, both built-in and programmer-defined, are classified as safe or unsafe. Most of the Cedar Language is safe.

Unsafe constructs include LOOPHOLE, dereferencing POINTERS (but REFS are safe), JOIN, @ (address of), computed variant records, and non-copying variant discrimination.

A region of program text, bracketted to form a block, may be prefixed with CHECKED, TRUSTED, or UNCHECKED.

In checked program regions, language-enforced restrictions guarantee safety. If a block is checked, then within that block only safe operations may be used, the block itself implements a safe operation, and procedures declared in the block are treated as safe.

Even unchecked regions are supposed to maintain the safety invariants, but the guarantee must be by the programmer, rather than the system. If a block is unchecked, unsafe operations may be used internally, the block itself is considered to implement an unsafe operation, and procedures declared in the block are treated as unsafe. Generally even unchecked regions can be composed primarily of safe operations; unsafe operations should be used only for good reasons and with due caution.

A trusted block may also invoke unsafe operations, but it is assumed to implement an operation that is safe by programmer guarantee. TRUSTED is a programmer assertion that cannot be checked by the compiler, and therefore represents a special kind of loophole.

For easy upward compatibility from Mesa, the following defaults have been adopted: If a module is prefixed with CEDAR, then the outermost block is CHECKED and all interfaces are assumed to be safe;

otherwise,. the outermost block is UNCHECKED and all interfaces are assumed to be unsafe. The checking attribute is inherited: unless a nested block is explicitly prefixed, it is checked or unchecked like the textually enclosing block.

If a system consists entirely of safe regions (and the invariants hold initially), then by induction the system is invulnerable. However, an error in an unchecked region can make even the checked regions vulnerable. Thus the CHECKED/UNCHECKED boundary limits responsibility, but not vulnerability. Confidence that errors in checked regions will not cause system crashes is based on the the automatic enforcement of safety restrictions. Confidence that unchecked regions will not cause system crashes is based on trust that they are free from errors that violate the safety invariants.

Caveat: The conversion of the Cedar system to safe interfaces is presently underway. The unsafe interfaces are beginning to disappear. You should program as safely as you can, but do not be surprised by the initial density of safety complaints from the compiler. A good rule is to prefix each module with CEDAR, and then to put TRUSTED on each block about which the compiler complains, *after convincing yourself that the complaint is not your fault*, because it results from a necessary use of an unsafe system interface. The reason for each TRUSTED should be documented in an accompanying comment.

Type confusion

Mesa is a strongly typed language, which means that the types of names are declared, and that the language imposes restrictions to keep values of one type from being accidentally interpreted as values of another. Because knowledge of the type structure of values in memory is so essential to the garbage collector (it must locate and follow REFs in order to determine current storage usage), it is particularly vulnerable to any operations that cause data in memory to be interpreted as having other than their true types. Thus, much of the effort in designing the safe subset went into identifying all the features in Mesa that allow type-checking to be circumvented (accidentally or deliberately) and designing safe replacements for the important uses of those features.

LOOPHOLE is a "type converter" in Mesa that allows any value to be treated as having any specified type; it is the most obvious breach of type security. It causes a safety problem only if it allows mistyped data to be stored into memory (i.e., if the target type contains an address, such as a pointer or procedure value); other uses will introduce implementation dependencies, but not threaten safety. Within checked regions, LOOPHOLE is not allowed to produce a value of a reference-containing (RC) type.

Narrowing and type discrimination

Cedar introduces a number of new type distinctions, frequently leading to a number of separate, but closely related types. It is often desirable to coerce a value of one of these types into a value of a related type. Where the types are such that it can be statically guaranteed that no information will ever be lost by the coercion, it is called a *widening*, and is performed automatically whenever demanded by context (e.g., assigning a bound variant value to a variant record variable). In general, conversion in the other direction requires a runtime check to ensure that information is not being lost. To make the possibility of such failure explicit in the program text, the NARROW type converter may be applied (and may include a catch phrase to handle the NarrowFault exception).

The built-in test ISTYPE can be applied to a value to determine whether it can be narrowed to a specified type without error. If so, it is said to satisfy the type's *predicate.*

If the target type of a narrowing is uniquely determined by context, it need not be an explicit argument to NARROW.

Delayed binding

A desirable property of a high-level programming language is that is allow a wide range of *binding times:* that is, it should allow the programmer maximal control over when the attributes of a particular variable are determined, with different choices not requiring changes in all expressions containing the variable. Examples of such attributes are its type, storage allocation method, implementation (for abstract objects), and actual value; examples of binding times include program-writing time, compilation, configuration binding, program initialization, block entry, and statement execution. Generally speaking, deferring the binding of an attribute leads to greater generality in the program at the cost of decreased static checkability and (often) lower runtime efficiency.

Experience with languages like LISP and Smalltalk, in which most binding is done dynamically, shows that it is much easier to write certain kinds of programs, if type and/or implementation binding can be deferred. Programming tools (debuggers, performance monitors) and knowledge representation systems are typical examples. But few programs take full advantage of this flexibility very often. Cedar was designed to take advantage of early binding, as Mesa does, but to allow certain bindings to be explicitly deferred.

Dynamic typing, REF ANY, and dynamically typed procedure variables

Mesa provides very limited variability in the binding time of an object's type. Variant records allow a deferred choice between specific enumerated alternatives, and sequences allow deferring the specification of an object's length until it is allocated. Otherwise, all types must be static. This makes it virtually impossible to avoid LOOPHOLEs and *ad hoc* type tagging schemes when writing schedulers, sorters, output formatters, etc. that must operate on objects of unpredictable type.

Cedar's solution to this problem requires two new mechanisms: a runtime representation for types, and a way to associate a type with an object at runtime that is guaranteed consistent with the type system and static checking. (Note that Cedar adopts the view that an object's type is inherent in the object itself, rather than in the way the object is referred to.)

TYPE is a type in the Cedar Language. The "structuring methods" (e.g., ARRAY, RECORD, and REF) are viewed as operators that take type arguments and return type values as results. In the current language, the arguments to such operators must be static (compile-time) constants.

ANY is not a type in Cedar, but can stand in place of a type in the arguments to two operators: REF and PROC.

A REF ANY value may refer to a dynamic variable of any type whatsoever. Thus a REF *T* value, for any *T*, can be widened to a REF ANY value. But a REF ANY value cannot be directly dereferenced, because the type of the result is not static. The discriminating selection statement has been generalized to allow discrimination on the referent type of a REF ANY; within each selectable statement, the type is (statically) known to be the type specified in its test item. NARROW can also be used to safely convert a REF ANY value back to a REF *T* value; ISTYPE can be used to check whether NARROW will succeed.

A PROC type may also have ANY in place of the type of its formal parameter record type and/or result record type. PROC values with specific domains and ranges may be widened to these dynamic types, and later tested and narrowed analogously to REF ANYs. They must be narrowed before being applied.

In principle, each value in Cedar carries its syntactic type with it at all times. In practice, almost all analysis and checking of types is done by the compiler, and both space and time efficiency are gained by not storing constant types with values. However, the symbol tables produced by the compiler contain enough information to recover any type on demand, made available through a standard package. RuntimeTypes provides type-conversion routines in both directions between *typed values* (with type RTTypesBasic.TV) and ordinary Cedar values, and numerous operations on typed values to examine the type and structure of a typed value, to change its attributes, etc. Thus it is possible to write a program that deals with any given Cedar value or type without anticipating the specific type when the program is written. Programs such as BugBane (the Cedar debugger) absolutely require such flexibility.

The current implementation is too slow to be used effectively by client programs as a substitute for true polymorphism in the language, but is suitable for examining and changing variables interactively with the Cedar debugger.

Miscellaneous

Although Cedar was not intended as a research project in programming languages, its developers were not immune to the temptation to make Mesa better in ways that were not strictly required to enable the new programming environment. This section discusses a few of these new features.

Types as clusters of operations

Each type has an associated *cluster* of operations. The main purpose of this association is to support a style of "object oriented" notation. Using a record-like notation, a procedure "field" will be looked up in the cluster of the object's type, and then applied to the object and the other arguments.

It is preferred style in Cedar to use this object notation in invoking operations of interfaces designed to support it. Consult the relevant package documentation if in doubt.

Each built-in type and type constructor in Cedar implicitly supplies a standard cluster. The cluster extension mechanism is that each opaque or record type defined in a interface acquires all procedures declared in the same module as parts of its cluster.

ROPEs and IO

Mesa STRINGs are rather awkward objects, having been tuned for efficiency in a small-machine (Alto) world, rather than for flexibility and convenience. They are POINTERs to fixed-length sequences of characters. Considerable care is required to avoid surprising results, even for rather straightforward string-processing applications. Cedar ROPEs, on the other hand, are somewhat heavier-weight, more convenient to use, and less prone to surprises. Several different implementations of ropes, efficient for different purposes, provide the same interface.

Rope is a Cedar package that supports the creation and manipulation of immutable reference-counted sequences of characters. Procedures are provided for concatenation, taking substrings, scanning, and other operations. A client can provide specialized implementations for rope objects. The standard implementation attempts to avoid copying when performing Substr, Concat and Replace operations. The Rope package is the standard support for sequences of characters in Cedar,.

Most of the common operations on input/output streams, plus string conversions that are commonly used in dealing with input or formatting output, have been collected in the IO interface. Implementations are available for stream interfaces to all common devices, and to allow ropes and streams to be readily interconverted.

LISTs and ATOMs

Cedar includes LIST OF as a new type constructor for singly-linked (by REFs) lists, and a constructor for list values that mimics that of LISP, avoiding the need for a lot of NEWs or CONSs. The analog of LISP's CAR and CDR are provided by the standard fields *first* and *rest*. Unlike LISP, Cedar lists are statically typed (although the element type may be REF ANY).

Cedar also has a built-in type ATOM, which can be used for values that are uniquely determined by their print names. Any rope can be converted to an atom and conversely; the advantage of atoms is that, unlike ropes, it is very cheap to compare them for equality; atoms may also have *property lists.* Atom literals are just names prefixed by $.

Converting Mesa Programs to Cedar — Jim Morris

This section assumes you already know how to program in Mesa (or that you have a Mesa program to be converted), and is intended to explain the differences for programming in Cedar.

Simple programs

Let's suppose you want to run a simple program in Cedar. If an existing Mesa 5 or 6 program uses fairly vanilla stuff, it's easy to convert:

- The names of most interfaces and some procedures have changed, but the functionality is basically the same.
- The most obvious differences will be with strings and I/O. You should only need to know about two interfaces for these: Rope and IO, respectively.
- In general, the Cedar/Pilot community has dropped the use of "Defs" as a suffix for definition file names, and introduced the suffix "Impl" for implementation files; e.g. "InlineDefs" became "Inline".

Here's what you need to do to your Mesa 5 or 6 program:

- Change all STRINGs to ROPEs (actually Rope.ROPE). Remove all allocations and deallocations of strings. Change all references to StringDefs routines to use Rope or IO routines. Rope provides procedures to parse and manipulate ropes. IO provides procedures to convert ROPEs to numbers and back as noted below. One can now put special characters in rope literals by using the escape character "\". ". . . \n . . ." inserts a carriage return (newline), ". . . \t . . ." a tab, ". . . \\ . . ." a backslash, and ". . . \123 . . ." the character whose octal code is 123. Note a ROPE is immutable, unlike a string. Appending a character creates a new ROPE.
- You should use specific subranges for numeric variables whenever possible. If you don't know the range, use INT (32-bit integer), unless you know you don't need that big a number and know you need efficiency. In those cases use INTEGER or NAT = [0..77777B]. Avoid using CARDINALS or LONG CARDINALS; their main use is in dealing with STRINGs. The compiler recognizes the abbreviation INT for LONG INTEGER, BOOL for BOOLEAN, CHAR for CHARACTER, and PROC for PROCEDURE.
- Change all references to I/O packages of all kinds (streams, files, TTY) to use equivalent IO routines. IO is the only interface you should need to know about for I/O of almost any type of variable or constant (ROPE, INT, etc.) to almost any type of device (keyboard, display, files, temporary buffer etc.). IO contains:
  - A set of Create*X* routines for each kind of stream *X*—file, display, etc.
  - A set of Get*X* routines for each type *X* (integers, ropes, etc.)
  - A PutF routine that can be used with any type (integers, ropes, etc.) via a set of inline procedures (int, rope, etc.) which are used to tag the type of the arguments. It also provides a format argument which may be used to get FORTRAN-style formatting of output. For example, the format "%g" prints almost anything in default free-format:

    ```
    stream.PutF["The sum of %g and %g is %g.\n", int[x], int[y], int[x+y] ]
    ```
  - A PutFR routine that is identical to PutF except it produces a rope as output instead of putting its result on a stream, and a RS routine that makes a rope look like a stream so that the Get*X* procedures can be used. Thus one can convert various types to and from ropes, e.g. the following code which converts an integer to a rope and back:

    ```
    r: ROPE← PutFR[, int[i]];
     j: INT← GetInt[RS[r]];
    ```
- Make use of LISTs and SEQUENCEs instead of ARRAYs and DESCRIPTORs for ARRAYs. The interface List contains some useful routines.

New language features

The changes in the Cedar *language* from Mesa 6 are fairly easy to understand for simple programs:

(a) REFs provide automatic deallocation and easier allocation:

```
Node: TYPE = REF Rec;
    Rec: TYPE = RECORD[first: INTEGER, rest: Node];
    . . .
    x: Node ← NEW[Rec ← [5, NIL]];
```

(b) Runtime types via REF ANY give looser binding:

```
TNode: REF BlRec;
    Node: REF B2Rec;
    x: Node ← . . . ;
    t: TNode ← . . .;
    q: REF ANY;
    . . .
    q ← t; q ← x;  -- both of these are legal
    t ← NARROW[q];  -- raises NarrowRefFault if q is not a TNode
    -- q↑ ← E  is always illegal. You cannot update through a REF ANY.
    . . .
-- type can also be checked explicitly:
        WITH q SELECT FROM
            m: TNode => {t ← m; q ← m.lson};
            n: Node => {x ← n; q ← n.rest};
            ELSE ERROR;
-- or
    IF ISTYPE[q, TNode] THEN {t ← NARROW[q]; q ← t.lson}
        ELSE IF ISTYPE[q, Node] THEN {x ← NARROW[q]; q ← x.rest}
        ELSE ERROR.
REF ANY is preferred to the use of variant records.
```

(c) Lists are built into the language:

```
Node: TYPE = LIST OF INT;
    x: Node ← CONS[5, NIL];
    y: Node ← LIST[5, 6]; -- same as CONS[5, CONS[6, NIL]]
    i: INT← y.first; -- i is 5
    z:Node← y.rest; -- z is CONS[6, NIL]
    FOR l: Node ← y, l.rest UNTIL l=NIL DO . .
```

(d) ROPEs, ATOMs, SEQUENCEs, and INTs are also built-in.

(e) To protect yourself and the garbage collector from obscure errors you should program in the *safe subset* of the language. To get a program into the safe subset prefix each module (PROGRAM, MONITOR, or DEFINITIONS) with the word CEDAR. The compiler will then tell you when you are straying outside the safe subset. You can wave the compiler off any block by placing the word TRUSTED before it. If you call a procedure declared in an unsafe interface (i.e., one that doesn't start with CEDAR DEFINITIONS), the compiler will complain unless the call is in a TRUSTED block. Most of the high-level interfaces in the Cedar system are now safe.

Restrictions of the safe language

The @ operator is not permitted. There are three general ways to cope with this restriction: specializing, copying, and indirecting. For example, suppose you have a program that says

```
W: ARRAY [0..100) OF Z;
```

```
        P[@W];
        FOR i IN [0..100) DO . . . Q[@W[i]] . . . ENDLOOP;
```

To eliminate the first @ by specializing we would make a copy of the procedure P that dealt with the W directly—not very satisfactory. To eliminate the first @ by copying we would pass the array W in by value and back by result—also not very satisfactory. It is best to deal with the first @ by indirecting; just allocate W from collectable storage, writing

```
    W: REF ARRAY [0..100) OF Z = NEW[ARRAY [0..100) OF Z];
        P[W];
```

Eliminating the second @ by specialization is plausible if Q knows it is always dealing with array elements: pass a reference to W along with an index. Otherwise, deciding between copying and indirecting depends upon the size of a Z. If it is small copy it, writing "W[i] ← Q[W[i]]". If it is big create references to it and pass those, writing

```
    W: REF ARRAY [0..100) OF REF Z;
        P[W];
        FOR i IN [0..100) DO . . . Q[W[i]] . . . ENDLOOP;
```

The form of variant record discrimination that does not copy the value to a new location cannot be used. Suppose you have a variant-record data structure like

```
    T: TYPE = REF TR;
        TR: TYPE = RECORD[SELECT t:* FROM
            name, string => [x: ROPE];
            link => [i: INT, r: T];
            ENDCASE];
```

and are accustomed to performing discriminations like

```
    e: T;
        WITH x: e↑ SELECT FROM
            name, string => "Statements using x";
            link => {S1[x.i]; S2[@x]};
            ENDCASE;
```

You should declare a set of REFs to bound variant types like

```
    Name: TYPE = REF name TR;
        String: TYPE = REF string TR;
        Link: TYPE = REF link TR;
```

and rewrite the discrimination to be

```
    WITH e SELECT FROM
        x: Name => "Statements using x";
        x: String => "Statements using x";
        x: Link => {S1[x.i]; S2[x]};
        ENDCASE;
```

The type of x is now a REF type, not a TR, so various other types need to be adjusted and the @ in S2 is no longer needed. If "Statements using x" is a large block, you will probably want to introduce a procedure to avoid copying it.

Variant records cannot be overwritten. Similiar techniques can be used for sanitizing a program that overwrites variant records. Assuming the declarations of T and TR from above, suppose you wanted to write

```
    x: T ← NEW[TR ← [name["END"]];
        x↑ ← [link[5, x]];
```

The specialization/copying technique is to simply update the thing that points at the record, writing "x ← NEW[TR ← [link[5, x]]]". However, if you don't know all the places that point at the record, you must introduce another level of indirection, writing

```
    T: TYPE = REF REF TR;
        x: T ← NEW[REF TR ← NEW[TR ← [rope["END"]]];
```

```
x↑ ← NEW[TR ← [link[5, x]]];
```

Unsafe procedures cannot be passed as arguments to safe ones. The symptom of a violation of this rule is generally a message complaining about an incorrect type when there is no obvious type mismatch. *All* procedure types in an interface prefixed by CEDAR are implicitly prefixed with SAFE. The simplest thing to do is to put SAFE in front of PROC in the argument procedure declaration, and put TRUSTED in front of its body. As with all uses of TRUSTED, you should verify that the safety invariants are actually maintained, and document the reason for the TRUSTED in a comment.

For More Information . . .

Cedar Language Syntax

This is a one-page reference grammar describing the complete syntax of the Cedar Language, in a compact variation on BNF developed by Butler Lampson. Keep it handy as you write programs. It provides a relatively compact source of information on the exact form of constructs accepted by the compiler. It will also alert you to much of the available variety in the language—but of course, not every syntactically valid program makes semantic sense.

The parsing grammar used by the compiler is somewhat larger and more complex than the Reference Grammar. Some of this is for technical reasons associated with LALR(1) parsing, and some of it to enable the compiler to make certain semantic distinctions while parsing. The differences should be invisible when dealing with correct programs, but may affect the error messages given for incorrect ones.

Annotated Cedar Examples

This document contains four complete, runnable Cedar programs chosen to illustrate the use of most of the major features of the language, and to provide an introduction to the style of programming that is preferred in Cedar. You should certainly invest time in studying them before attempting to write Cedar programs. If you are one of those who learns best from examples, you may find them virtually the only tutorial information you need to learn the language.

These examples have been chosen so that they are also useful prototypes of kinds of programs you may want to write in Cedar. If you are like most Cedar programmers, you will probably find it easier to start from such a prototype, and change it to do what you want, than to enter a whole program "from scratch."

Stylizing Cedar Programs

Because Cedar programmers so frequently read each other's code, it is considered good citizenship to adhere to certain stylistic conventions. Stylizing Cedar Programs discusses the generally agreed conventions.

You can save yourself a lot of typing, and produce nicely formatted code at the same time, by using Tioga's abbreviation expansion mechanism to generate all the high-level structure of your program (at least, all the bits that aren't simply copied). The file Cedar.abbreviations lists the available macros and their expansions; you can add your own favorites.

Cedar Program Style Sheet

This is an annotated prototype that you will probably want to keep close to hand, because it compactly illustrates the most important principles from the previous document.

Cedar Language Reference Manual

Eventually, this is intended to be a precise definition of the complete syntax and semantics of the Cedar Language. It is still incomplete.

The formal definition of the language is given in terms of a *kernel language*, into which all Cedar constructs can be *desugared* to determine their precise semantics. The Reference Manual contains both the definition of the kernel, and an explanation of the desugarings. It also contains several tables that collect important information about the primitive types and type constructors of Cedar.

Cedar Language Reference Summary Sheets

This is intended to be the essence of the entire Cedar Language carefully condensed into two pages for ready reference. It covers both syntax and semantics, with examples and notes. It is definitely not for those with weak eyes, and should probably not even be read until you have studied the Reference Manual proper. But it should be very helpful in checking details that you may have forgotten. Keep it handy.

Cedar Catalog

Since so much Cedar programming is done "at the component level," you need to know what packages and tools are available and what they do. In general, full documentation (or at least the best available approximation thereto) for each component is stored on [Indigo]<Cedar>Documentation>, or is referenced in the component's DF file, stored on [Indigo]<Cedar>Top>.

The problem is finding out which components you should be interested in. That's where the Cedar Catalog comes in handy. It contains a somewhat structured list of all the components in Cedar considered "interesting" by their maintainers. A component may be interesting

because of what it provides (your program may become a *client*),

because of what it does (you personally may become a *user*), or

because of how it does it (you may study it or copy some part of it in your program).

For each entry, the Catalog indicates why it is considered interesting, and how to acquire documentation and the component itself. It also identifies the maintainer, who is the ultimate source of advice and help.

Mesa 5.0 Manual

The *Mesa Language Manual, Version 5.0*, PARC Technical Report CSL-79-3, is the most recent self-contained manual on the Mesa Language. It falls somewhere between a tutorial and a reference manual, and many users have complained that it isn't entirely satisfactory for either purpose. But if you need more information about the Mesa-like parts of Cedar, it may be your best source.

Chapter 4 gives the details of Mesa's basic control constructs.

Chapter 5 tells all about procedures.

Chapter 7 goes into more detail than you probably want about the fine points of modules, programs, and configurations. You may be better off extrapolating from the Annotated Cedar Examples.

Chapter 8 gives some of the gory details of exceptions and exception handling. It is easy to get in trouble unless you use them in straightforward ways.

Chapter 10 provides a pretty reasonable discussion of how to make effective use of processes, monitors, condition variables, etc.

Who to see

If you haven't managed to find information that you want after you have looked in what you consider to be the obvious places (or if you don't understand what you have found), don't hesitate to ask. Almost anyone in CSL is a fount of wisdom, willing to be asked almost any question on almost any subject. (Of course, the answers aren't equally reliable, but you can't have everything.) If the first person you ask doesn't know the answer, chances are good that you'll get a pointer to either a person or document that will have the answer. More specifically here are some good people to ask:

John Maxwell	assistant of first resort for general problems with Cedar
Russ Atkinson	BugBane

Roy Levin	runtime system
Scott McGregor	Viewers and Tioga
Bill Paxton	Tioga
Paul Rovner	runtime-type system
Ed Satterthwaite	compiler
Warren Teitelman	user interaction, UserExecutive, IO

Cedar Safe Language Syntax

3.3 1 **module** ::= DIRECTORY (n_d ?(: TYPE n_t) ?(USING [n_u, ...])), ... ;
(interface | implementation)

2 **interface** ::= n_m !.. : CEDAR DEFINITIONS ?locks
?(IMPORTS ((n_{iv} : |) n_{it}), ...) ~ { ?open^7 (d | b); !.. } .

3 **implementation** ::= n_m : CEDAR
(PROGRAM ?drType^{42} | MONITOR ?drType^{42} ?locks)
?(IMPORTS ((n_{iv} : |) n_{it}), ...) ?(EXPORTS n_e, ...) ~ block .

5 **locks** ::= LOCKS e ?(USING n_u: t)

3.4 6 **block** ::= ?(CHECKED | UNCHECKED | TRUSTED)
{ ?open ?enable ?body ?(EXITS (n, !.. => s); ...) } *--In 3, 13, 14.*

7 **open** ::= OPEN (n ~~ e | e), !.. ; *--In 2, 5.*

8 **enable** ::= ENABLE { enChoice; ... };

9 **enChoice** ::= (e, !.. | ANY) => s *--In 7, 27.1.*

10 **body** ::= (d | b); !.. ; s; ... | s; !.. *--In 5, 17.*

3.5 11 **d***eclaration* ::= n, !.. : ?(PUBLIC | PRIVATE) varTC^{40} *--In 2, 10, 43.*

13 **b***inding* ::= n, !.. : ?(PUBLIC | PRIVATE) t ~ (*--In 2, 10.*
e | t_2 -- *t=*TYPE-- | CODE | ?INLINE ?(ENTRY|INTERNAL) block^6)

3.6 14 **s***tatement* ::= e_1←e_2 | e | block^6 | escape | loop | NULL

16 **escape** ::= GOTO n | EXIT | CONTINUE | (RETURN|RESUME) ?e

17 **loop** ::= ?iterator ?(WHILE e | UNTIL e)
DO ?body^{10} ?(REPEAT FINISHED => s) ENDLOOP

18 **iterator** ::= THROUGH e |
FOR n : t (?DECREASING IN e | ← e_1 , e_2)
e is a subrange. n is readonly.

3.7 19 **e***xpression* ::= n | literal^{57} | (e) | application^{26} |
(e | typeName^{37}) . **(9)** n |
prefixOp e | e_1 infixOp e_2 | e_1 AND **(2)** e_2 | e_1 OR **(1)** e_2 |
e ↑ **(9)** | ERROR | [argBinding^{27}] |
builtIn [e, ... ?applEn$^{27.1}$] |
funnyAppl e ?([?argBinding^{27} ?applEn$^{27.1}$]) |
s | subrange^{25} | if^{28} | select^{29} | safeSelect^{32}
Precedence is in **bold** *in rules 19-21. All operators associate to the left except ←, which associates to the right. Application has highest precedence. Subrange only after* IN *or* THROUGH. *s only in if^{28} and select choices$^{30\ 33}$.*

20 **prefixOp** ::= @ **(8)** | **(7)** | (~ | NOT) **(3)**

21 **infixOp** ::= * | / | MOD **(6)** | + | **(5)** | relOp **(4)** | ← **(0)**

22 **relOp** ::= ?NOT (?~ (= | < | >) | <= | >= | # | IN) *--In 21, 30.*

23 **builtIn** ::= -- These are enumerated in Table 45.

24 **funnyAppl** ::= FORK | JOIN | WAIT | NOTIFY | BROADCAST |
SIGNAL | ERROR | RETURN WITH ERROR

25 **subrange** ::= ?typeName^{37} ([| ( ) e_1 .. e_2 ( ] |)) *--In 19, 39.*

26 **application** ::= e [?argBinding ?applEn]

27 **argBinding** ::= (n ~ ?e), !.. | (?e), !.. *--In 19, 26.*

27.1 **applEn** ::= ! enChoice^9; ... *-- In 19, 26.*

3.8 28 **if** ::= IF e_1 THEN e_2 ?(ELSE e_3)

29 **select** ::= SELECT e FROM choice; ... endChoice
The ";" is "," in an expression, here and in 32.

30 **choice** ::= (?relOp^{22} e_1), !.. => e_2

31 **endChoice** ::= ENDCASE ?(=> e_3) *--In 29, 32, 34.*

32 **safeSelect** ::= WITH e SELECT FROM safeChoice; ... endChoice^{31}

33 **safeChoice** ::= n : t => e_2

3.2 56 **n***ame* ::= letter (letter | digit)...

57 **literal** ::= num ?(D | B) | digit (digit |A|B|C|D|E|F) ... H |
?num . num ?exponent | num exponent |
' extendedChar | " extendedChar ... " | $ n

58 **exponent** ::= E ?(+ |) num

59 **num** ::= digit !..

60 **extendedChar** ::= space | \ extension | anyCharNot'"Or\

61 **extension** ::= digit_1 digit_2 digit_3 | N | R | T | B | F | L | ' | " | \

36 **t***ype* ::= typeName | builtInType | typeCons

4 37 **typeName** ::= n | typeName . n | typeName[e] *--In 19,25,36,40.1*

4.2 38 **builtInType** ::= INT | REAL | TYPE | ATOM |
CONDITION | MONITORLOCK
See Table 42. TYPE *only in a b or an interface's d.*

39 **typeCons** ::= subrange^{25} | paintedTC$^{40.1}$ | transferTC^{41} |
arrayTC^{44} | seqTC^{45} | refTC^{46} | listTC^{47} |
recordTC^{50} | unionTC^{52} | enumTC^{54} | defaultTC^{55}

4.3 40 **varTC** ::= (| READONLY | VAR) t | ANY
In 11, 46, 47. ANY *only in refTC.* VAR *only in interface decl.*

40.1 **paintedTC** ::= typeName^{37} PAINTED t

4.4 41 **transferTC** ::= ?(SAFE | UNSAFE) xfer ?drType

41.1 **xfer** ::= PROC | PORT | PROCESS | SIGNAL | ERROR | PROGRAM

42 **drType** ::= ?fields_1 RETURNS fields_2 | fields_1 *--In 3, 41.*

43 **fields** ::= [d^{11}, ...] | [t, ...] | ANY *--In 42, 50, 52.* ANY *only in drType.*

44 **arrayTC** ::= ARRAY t_1 OF t_2

45 **seqTC** ::= SEQUENCE n : t_1 OF t_2 *-- Only as last type in 50 or 52.*

4.5 46 **refTC** ::= REF ?varTC^{40}

47 **listTC** ::= LIST OF varTC^{40}

4.6 50 **recordTC** ::= ?MONITORED RECORD fields^{43}

52 **unionTC** ::= SELECT n : (t | *) FROM (n => fields^{43}), ... ?,
ENDCASE *-- Only as last type in fields of 50 or 52.*

4.7 54 **enumTC** ::= {n, ...} | MACHINE DEPENDENT {(?n (e) | n), ...}

4.11 55 **defaultTC** ::= t ← | t ← e
Only as t in a decl in body^9 or field^{43} (n: t ← e), in a TYPE *binding^{13} or in* NEW.

tation: item | item = choose one; ?item = zero or one item; terminals: SMALL CAPS, underlined or punctuation other than bold ()?| (parens are terminals only in rules 19, 25, 54).
item s ... = zero or more items, separated by s; item s !.. = one or more items, separated by s; with s = ";", a trailing "," is optional; s is one of: empty "," ";" .

breviated non-terminals: b = binding^{13}; d = declaration^{11}; e = expression^{19}; n = name^{56}; s = statement^{14}; t = type^{36}.

16 Feb 8

ments: n^{56} means n is defined in rule 56; n_i = n (the subscript is only for the desugaring).

CLRMSafeGram.pres

Cedar Full Language Syntax

3.3 1 **module** ::= DIRECTORY (n_d ?(: TYPE ?n_t)
?(USING [n_u, ...])), ... ;
(interface | implementation)

2 **interface** ::= n_m !.. : ?CEDAR DEFINITIONS ?locks
?(IMPORTS ((n_{iv} : |) n_{it}), ...)
?(SHARES n_s, ...) ~ ?access^{12} { ?open^{7} (d | b); !.. } .

3 **implementation** ::= n_m : ?CEDAR ?safety
(PROGRAM ?drType^{42} | MONITOR ?drType^{42} ?locks)
?(IMPORTS ((n_{iv} : |) n_{it}), ...) ?(EXPORTS n_e, ...)
?(SHARES n_s, ...) ~ ?access^{12} block .

4 **safety** ::= SAFE | UNSAFE *--In 3, 41.*

5 **locks** ::= LOCKS e ?(USING n_u: t)

3.4 6 **block** ::= ?(CHECKED | UNCHECKED | TRUSTED)
{ ?open ?enable ?body ?(EXITS (n, !.. =>s); ...) } *--In 3,13,14.*

7 **open** ::= OPEN (n ~~ e | e), !.. ;
In 2, 5, 17. The ~~ may be written as :.

8 **enable** ::= ENABLE (enChoice | {enChoice; ...}); *--In 5, 17.*

9 **enChoice** ::=(e, !.. | ANY) => s *--In 7, 27.1.*

10 **body** ::= (d | b); !.. ; s; ... | s; !.. *--In 5, 17.*

3.5 11 **d***eclaration* ::= n, !.. : ?access varTC^{40}
In 2, 10, 43. VAR, READONLY *only for interface var.*

12 **access** ::= PUBLIC | PRIVATE *--In 2, 3, 11, 13, 50, 51, 53.*

13 **b***inding* ::= n, !.. : ?access t ~ (
e | t_2 -- *if t=*TYPE-- | CODE |
?TRUSTED MACHINE CODE { (e, ...): ... } |
?INLINE ?(ENTRY | INTERNAL) block^{6})
In 2, 10. The ~ may be written as =. ENTRY *or* INTERNAL *may be written before t. Block or* MACHINE CODE *only for proc types.*

3.6 14 **s***tatement* ::= e_1←e_2 | e | block^{6} | escape | loop | NULL
--In 6, 10, 17, 19.

16 **escape** ::= GOTO n | GO TO n | EXIT | CONTINUE | LOOP |
RETRY | REJECT | (RETURN | RESUME) ?e | e ← STATE

17 **loop** ::= ?iterator ?(WHILE e | UNTIL e)
DO ?open^{7} ?enable^{8} ?body^{10}
?(REPEAT (n, !.. => s); ...) ENDLOOP

18 **iterator** ::= THROUGH e |
FOR (n : t | n) (?DECREASING IN e | ← e_1 , e_2)
e is a subrange. In FOR *n: t ...., n is readonly.*

3.7 19 **e***xpression* ::= n | literal^{57} | (e) | application^{26} |
(e | typeName^{37}) . (9) n |
prefixOp e | e_1 infixOp e_2 | e_1 AND (2) e_2 | e_1 OR (1) e_2 |
e ↑ (9) | STOP | ERROR | [argBinding^{27}] |
builtIn [e, ... ?applEn$^{27.1}$] |
funnyAppl e ?([?argBinding^{27} ?applEn$^{27.1}$]) |
s | subrange^{25} | if^{28} | select^{29} | safeSelect^{32} | withSelect^{34}
Precedence is in **bold** *in rules 19-21. All operators associate to the left except ←, which associates to the right. Application has highest precedence. Subrange only after* IN *or* THROUGH. *s only in if^{28} and select choices$^{30\ 33\ 35}$.*

20 **prefixOp** ::= @ (8) | (7) | (~ | NOT) (3)

21 **infixOp** ::= * | / | MOD (6) | + | (5) | relOp (4) | ← (0)

22 **relOp** ::= ?NOT (?~ (= | < | >) | <= | >= | # | IN) *--In 21, 30.*

23 **builtIn** ::= -- These are enumerated in Table 45.

24 **funnyAppl** ::= FORK | JOIN | WAIT | NOTIFY | BROADCAST |
SIGNAL | ERROR | RETURN WITH ERROR | NEW | START |
RESTART | TRANSFER WITH | RETURN WITH

25 **subrange** ::= ?typeName^{37}
([e_1 .. e_2] | [e_1 .. e_2) | (e_1 .. e_2] | (e_1 .. e_2)) *--In 19, 39, 48.*

26 **application** ::= e [?argBinding ?applEn]

27 **argBinding** ::= (n ~ (e | | TRASH)), !.. | (e | | TRASH), !..
In 19, 26. The ~ may be written as :. NULL *may be written for* TRASH.

27.1 **applEn** ::= ! enChoice^{9}; ... *-- In 19, 26.*

3.8 28 **if** ::= IF e_1 THEN e_2 ?(ELSE e_3)

29 **select** ::= SELECT e FROM choice; ... endChoice
The ":" is "." in an expression, here and in 32 and 34.

30 **choice** ::= (?relOp^{22} e_1), !.. => e_2

31 **endChoice** ::= ENDCASE ?(=> e_3) *--In 29, 32, 34.*

32 **safeSelect** ::= WITH e SELECT FROM safeChoice; ... endChoice^{31}

33 **safeChoice** ::= n : t => e_2

34 **withSelect** ::= WITH (n_1 ~~ e_1 | e_1) SELECT ?e_{11} FROM
withChoice; ... endChoice^{31} *--The ~~ may be written as :.*

35 **withChoice** ::= n_2 => e_2 | n_2, n_2, !.. => e_2

---

4 36 **t***ype* ::= typeName | builtInType | typeCons

37 **typeName** ::= n_1 | typeName . n_2 | typeName[e] |
n_2 typeName *--In 19, 25, 36, 40.1, 49.*

4.2 38 **builtInType** ::= INT | REAL | TYPE | ATOM | CONDITION |
MONITORLOCK | LONG CARDINAL | ?LONG UNSPECIFIED |
MDSZone | ?UNCOUNTED ZONE
See Table 42. TYPE *only in a b or an interface's d.*

39 **typeCons** ::= subrange^{25} | paintedTC$^{40.1}$ | transferTC^{41} |
arrayTC^{44} | seqTC^{45} | descriptorTC$^{45.1}$ | refTC^{46} | listTC^{47} |
pointerTC^{48} | relativeTC^{49} | recordTC^{50} | unionTC^{52} |
enumTC^{54} | defaultTC^{55}

4.3 40 **varTC** ::= (| READONLY | VAR) t | ANY
In 11, 4548. ANY *only in refTC.* VAR *only in interface decl.*

40.1 **paintedTC** ::= typeName^{37} PAINTED t

4.4 41 **transferTC** ::= ?safety^{4} xfer ?drType

41.1 **xfer** ::= PROCEDURE | PROC | PORT | PROGRAM |
PROCESS | SIGNAL | ERROR

42 **drType** ::= ?fields_1 RETURNS fields_2 | fields_1 *--In 3, 41.*

43 **fields** ::= [d^{11}, ...] | [t, ...] | ANY *--In 42, 50, 52.* ANY *only in 42.*

44 **arrayTC** ::= ?PACKED ARRAY ?t_1 OF t_2

45 **seqTC** ::= ?PACKED SEQUENCE tag^{53} OF t
Legal only as last type in the fields of a recordTC or unionTC.

45.1 **descriptorTC** ::= ?LONG DESCRIPTOR FOR varTC^{40}
varTC must be an array type.

4.5 46 **refTC** ::= REF ?varTC^{40}

47 **listTC** ::= LIST OF varTC^{40}

48 **pointerTC** ::= ?LONG ?ORDERED ?BASE POINTER
?subrange^{25} ?(TO varTC^{40}) | POINTER TO FRAME [n]
Subrange only in a relativeTC; no typeName^{37} on it.

49 **relativeTC** ::= typeName^{37} RELATIVE t

4.6 50 **recordTC** ::= ?access^{12} (?MONITORED RECORD fields^{43} |
MACHINE DEPENDENT RECORD (mdFields | fields^{43}))

51 **mdFields** ::= [((n pos), ... : ?access^{12} t), ...] *-- In 50, 52.*

51.1 **pos** ::= (_ e_1 ?(: e_2 .. e_3)) *-- In 51, 53.*

52 **unionTC** ::= SELECT tag FROM
(n, ... => (fields^{43} | mdFields^{51} | NULL)), ... ?, ENDCASE
Legal only as last type in the fields of a recordTC or unionTC.

53 **tag** ::= (n ?pos$^{51.1}$: ?access^{12} | (COMPUTED|OVERLAID))
(t | *) *--In 44, 52. * only in unionTC.*

4.7 54 **enumTC** ::= {n, ...} | MACHINE DEPENDENT {(?n (e) | n), ...}

4.11 55 **defaultTC** ::= t ← | t ← e | t ← e | _ TRASH | t ← TRASH
defaultTC legal only as the type in a decl in a body^{9} or field^{43} (n: t ← e), in a TYPE *binding^{13}, or in* NEW. *Note the terminal* |. TRASH *may be written* NULL.

3.2 56 **n***ame* ::= letter (letter | digit)...

57 **literal** ::= num ?((<u>D</u>|<u>d</u> | <u>B</u>|<u>b</u>) ?num) *--* INT *literal.* |
digit (digit |<u>A</u>|<u>B</u>|<u>C</u>|<u>D</u>|<u>E</u>|<u>F</u>) ... (<u>H</u>|<u>h</u>) ?num *-- Hex* INT *literal.* |
?num . num ?exponent | num exponent *--* REAL *literal.* |
' extendedChar | digit !.. <u>C</u> | " extendedChar ... " ?<u>L</u> | $ n

58 **exponent** ::= (<u>E</u>|<u>e</u>) ?(+ |) num

59 **num** ::= digit !..

60 **extendedChar** ::= space | \ extension | anyCharNot'"Or\

61 **extension** ::= digit_1 digit_2 digit_3 |
<u>n</u>|<u>N</u> | <u>r</u>|<u>R</u> | <u>t</u>|<u>T</u> | <u>b</u>|<u>B</u> | <u>f</u>|<u>F</u> | <u>l</u>|<u>L</u> | ' | " | \

ation: item | item = choose one; ?item = zero or one item; terminals: SMALL CAPS, <u>underlined</u> or punctuation other than bold ()?| (parens are terminals only in rules 19, 25, 51.1, 54, | only in 55).
item s ... = zero or more items, separated by s; item s !.. = one or more items, separated by s; with s = ";", a trailing ";" is optional; s is one of: empty "," ";" .
breviated non-terminals: **b** = binding^{13}; **d** = declaration^{11}; **e** = expression^{19}; **n** = name^{56}; **s** = statement^{14}; **t** = type^{36}.
nments: = obsolete; = efficiency hack; = unsafe; = machine dependent; n ° means n is defined in rule 56. n_j = n (the subscript is only for the desugaring).

Syntax	Meaning	Examples	Notes
odule ::= DIRECTORY (n_d(: TYPE (n_t |) |)	λ [(n_d : ((TYPE n_t | TYPE n_d) | TYPE n_d), ...] IN	DIRECTORY	
?(USING [n_u, ...])), ... ;	LET (n_d~RESTRICT[n_d, [$\$n_u$, ...]]), ...	Rope: TYPE USING [ROPE, Compare],	-- There should always be a USING c
(interface | implementation)	IN (**interface | implementation**)	CIFS: TYPE USING [OpenFile,Error,Open,read],	-- unless most of the interface is use
.terface ::= n_m !.. : ?CEDAR DEFINITIONS	λ [((n_{iv} | n_{it}):MKINSTANCETYPE[n_{it}]), ...] =>	IO: TYPE IOStream,	
?locks ?(IMPORTS ((n_{iv} : |) n_{it}), ...)	[n_m: TYPE n_m] IN	Buffer: TYPE;	-- or it is exported.
?(SHARES n_s, ...)	-- Access to PRIVATE names in n_s is allowed.		
~ ?access^{12} { ?open^{7} (d | b); !.. } .	LET REC n_m~**open** [?(lock′~**locks**,) **(d | b)**, ...] IN n_m		
nplementation ::= n_m . ?CEDAR	λ [((n_{iv} | n_{it}):MKINSTANCETYPE[n_{it}]), ...] =>	BufferImpl: MONITOR [f: CIFS.OpenFile]	-- Implementations can have argume
?safety (PROGRAM ?drType^{42} |	[(n_e: n_e) , ... , n_m: TYPE n_m , CONTROL: PROGRAM]] IN	LOCKS Buffer.GetLock[h]↑	-- LOCKS only in MONITOR, to speci
MONITOR ?drType^{42} (| locks))	(| (LET LOCK:MONITORLOCK IN LET l′~(λ IN LOCK) IN |))	USING h: Buffer.Handle	-- a non-standard lock.
?(IMPORTS ((n_{iv} : |) n_{it}), ...)	LET b′~NEWPROGINSTANCE[block].UNCONS IN	IMPORTS Files: CIFS, IO, Rope	-- Note the absence of semicolons.
?(EXPORTS n_e, ...)	[(n_e~BINDDFROM[n_e, b′]), ... , n_m~b′, CONTROL~b′.n_m]	EXPORTS Buffer	-- EXPORTS in PROGRAM or MONIT
?(SHARES n_s, ...)	where the body of the block is desugared thus:	~ { -- module body -- } .	-- Note the final dot.
~ ?access^{12} block .	**[(| (| l′~locks.)) (d | b)**, ... , n_m: PROGRAM **drType~{s;**		
...}]			
fety ::= SAFE | UNSAFE --*In 3, 41.*			
cks ::= LOCKS e ?(USING n_u: t)	λ ?([n_u : t]) IN e		
ock ::=		CHECKED {	-- Unnamed OPEN OK for exported
?(CHECKED | UNCHECKED | TRUSTED)		OPEN Buffer, Rope;	-- interface or one with a USING cla
{ ?open ?enable ?body	open LET n″, ... : EXCEPTION~NEWLABEL[] , ...	ENABLE Buffer.Overflow =>GOTO HandleOvfl;	-- A single choice needn't be in {}.
?(EXITS (n, !.. =>s); ...) }	IN (body enable) BUT { (n″, ... => s); ... }	stream: IO.Stream~IO.CreateFileStream["X"];	-- Use a binding if a name's value is f
--*In 3, 13, 15.*	-- *But n″ is not visible in s.*	x: INT←7;	-- Better to initialize declared names.
pen ::= OPEN (n ~~ e | e), !.. ;	(LET n~λ$_{open}$ IN e.DEREF | --*The* IN *before !.. is a separator.*	{OPEN b~~GetBuffer[stream];	-- A statement may be a nested block
In 2, 5, 17. The ~~ may be written as :.	LET BINDP[∇(e.DEREF).P,	ENABLE {	-- Multiple enable choices must be in
	OPENPROCS[∇(e.DEREF).P, λ IN e.DEREF]]) IN !.. IN	Files.Error--[error, file]-- =>{	-- ERRORs can have parameters.
nable::=ENABLE(enChoice|	BUT ({ enChoice } |	stream.Put[IO.rope[error]];	
{enChoice; ...});	{ enChoice; ... })	ERROR Buffer.Error["Help"] };	-- Choices are separated by semicolon
In 5, 17.		ANY =>{ x←12; GOTO AfterQuit } };	-- ANY must be last. ENABLE ends w
nChoice ::=(e, !.. | ANY) => s	(e | ANY), ... => { s; REJECT; EXITS	y: INT←9; ... };	-- Other bindings, decls and statemen
In 7, 27.1.	Retry′ =>GOTO Retry″^{14}; Cont′ =>GOTO Cont″^{14} }	x←stream.GetInt; ...	-- Other statements in the outer block
ody ::= (d | b); !.. ; s; ... |s; !..	LET NEWFRAME[REC [(d | b), ...]].UNCONS IN { s; ...}	EXITS	-- Multiple EXIT choices are *not* in {}
In 5, 17.		AfterQuit =>{...};	-- *AfterQuit, HandleOvfl* declared her
		HandleOvfl =>{...} };	-- legal only in a GOTO in the block.
eclaration ::= n, !.. : ?access^{12} varTC^{40}	**(n: varTC)**, ...	HistValue: TYPE[ANY];	-- Interface: An exported type.
In 2, 10, 43. VAR, READONLY *only for interface var.*		Histogram: TYPE~REF HistValue;	-- A type binding.
ccess ::= PUBLIC | PRIVATE		baseHist: READONLY Histogram;	-- An exported variable .
In 2, 3, 11, 13, 50, 51, 53.		AddHists: PROC[x, y: Histogram]	-- An exported proc.
inding ::= n, !.. : ?access^{12} t ~ (	LET x′ : t ~ (-- *The desugaring for n is at the end.*	RETURNS [Histogram];	
e |	e |	LabelValue: PRIVATE TYPE~RECORD[	-- PRIVATE only for sec
t_2 -- *if t=*TYPE |	t_2 -- *Same as e except for conflicting syntax.* |	first,last:INT,s:ROPE,x:REAL,f,g:INT,r:REF ANY];	-- stuff in an interface.
CODE |	NEWEXCEPTIONCODE[] --*t*⇒SIGNAL *or* ERROR |	Label: TYPE~REF LabelValue;	
?INLINE ?(ENTRY | INTERNAL) block^{6} |	λ [d′: t.DOMAIN] IN LET r′~NEWFRAME[t.RANGE].UNCONS	Next: PROC[l: Label] RETURNS[Label]~	-- An inline proc binding
	IN (LET r′ IN {t.DOMAIN~d′; **block**; RETURN}	INLINE { RETURN [NARROW[l.r]] };	
	BUT {Return″′ =>r }) |		
?TRUSTED MACHINE CODE {(e, ...); ...}	MACHINECODE[(BYTESTOINSTRUCTION[e, ...]), ...]	H: TYPE~Histogram^{11}; Size: INT~10;	-- Implementation: Binds a TYPE and
)	) IN n, ... ~ x -- *e is evaluated only once.*	HistValue: PUBLIC TYPE~HV$^{40.1}$;	-- PUBLIC for exports.
In 2, 10. The ~ may be written as =		baseHist: PUBLIC H←NEW[HistValue←ALL[17]];	-- An exported variable
Block or MACHINE CODE *only for proc types.*		x, y: HistValue←[20, 18, 16, 14, 12, 10, 8, 6, 4, 2, 0];	-- with initialization.
ENTRY *and* INTERNAL *can also be before t.*		FatalError: ERROR[reason: ROPE]~CODE;	-- Binds an error.
		Setup: PROC [bh: Handle^{3}, a: INT]~ENTRY {...};	-- Binds an entry proc.
		i,j,k: INT←0; p,q: BOOL; lb: Label; main: Handle;	
atement ::= sS	{ SIMPLELOOP {sS; GOTO Cont″; EXITS Retry″ =>NULL};	x←AddHists[baseHist, baseHist]↑;	-- A statement can be an assignment,
In 6, 10, 17, 19.	EXITS Cont″ =>NULL }	Setup[bh~main, a~3];	-- or an application without results,
S ::= e_1←e_2 | e | block^{6} | escape | loop | NULL	[e_1←e_2].TOVOID | e --*must yield* VOID-- | --*all yield* VOID--	{ENABLE FatalError =>RETURN[0]; []←f[3]; ...};	-- or a block,
scape ::= GOTO n | GO TO n |	EXORVAL[exception[code~ n″, args~NIL]] |	IF i>3 THEN RETURN[25] ELSE GOTO NotPresent;	-- or an IF or an escape statement.
EXIT | CONTINUE | LOOP | RETRY |	GOTO (Exit′^{17} | Cont′^{9} | Loop′^{17}| Retry′^{9}) |		
(RETURN | RESUME) ?e |	{ ?(r′^{13}←e;) GOTO (Return′^{13} | Resume′^{13}) } |		
REJECT | e ← STATE	THISEXCEPTION[] | DUMPSTATE[e]		
op ::= (iterator |)	{ (iterator ; | done′~FALSE; Next′: PROC~{};)	FOR t:INT DECREASING IN [0..5) UNTIL f[t]>3 DO	-- or a loop. Try to declare *t* in the FO
(WHILE e | UNTIL e |)	{ Test′~λ IN (NOT e | e | FALSE);	u: INT←0; ... ; u←t+4; ...	-- as shown. Avoid OPEN or ENABLE
DO ?open^{7} ?enable^{4} ?body^{10}	{ open SIMPLELOOP {	REPEAT Out =>{...}; FINISHED =>{...} ENDLOOP;	-- after DO (use a block). FINISHED
	IF Test′[] OR done′ THEN GOTO FINISHED;		-- must be last.
	{ **enable body** EXITS Loop′ =>NULL }; Next′[] }		
?(REPEAT (n, !.. =>s); ...) ENDLOOP	EXITS Exit′⇒NULL; **(n, !..⇒s)**; ...; FINISHED⇒NULL}}}		
erator ::= THROUGH e |	FOR x′: e IN e |	THROUGH [1..5) DO i←i*i ENDLOOP;	-- Raises *i* to the 6th power.
FOR (n : t | n)	(n: t: |)	FOR i: INT←1, i+2 WHILE i<8 DO j←j+i ...;	-- Accumulates odd numbers in [1..8).
((| DECREASING) IN e |	(Range′: TYPE~e; done′: BOOL←Range′.ISEMPTY;	FOR l: Label←lb, l.Next WHILE l # NIL DO ...;	-- Sequences through a list of *Labels*.
	Next′: PROC~{ IF n (≥Range′.LAST | ≤Range′.FIRST)		
	THEN done′←TRUE ELSE n←n.(SUCC | PRED) };		
	n←Range′.(FIRST | LAST); |		
← e_1 , e_2)	done′: BOOL~FALSE; Next′: PROC~{n←e_2}; n←e_1) ;		
e is a subrange. In FOR *n: t ..., n is readonly except for the assignment in the iterator's desugaring.*			
xpression ::= n | literal^{57} | (e) | application^{26} |		lv: LabelValue^{13}←[i, 3, "Hello", 31.4E1, (i+1),	-- A constructor with some sample
(e | typeName^{37}) . (9) n |		g[x]+lb.f+j.PRED];	-- expressions.
prefixOp e | e_1 infixOp e_2 |	e . prefixOp | e_1 . infixOp[e_2] |	p1: PROCESS RETURNS [INT]←FORK f[i, j];	-- FunnyAppls take one unbracketted
e_1 relOp (4) e_2 |	(λ [x′: ∇e_1, y′ :∇e_2] IN relop) [e_1, e_2] |	ERROR NoSpace; WAIT bufferFilled;	-- arg; many return no result, so
e_1 AND (2) e_2 | e_1 OR (1) e_2 |	IF e_1 THEN e_2 ELSE FALSE | IF e_1 THEN TRUE ELSE e_2 |	RT: RTBasic.Type←CODE[LabelValue^{13}];	-- must be statements.
e ↑ (9) | STOP | ERROR |	e . DEREFERENCE | STOP[] | ERROR NAMELESSERROR |	h[3, NOT(i>j), i*j, i←3, i NOT >j, p OR q, lb.r↑];	-- An application with sample expressi
builtIn [e_1 ?(, e_2, !..) ?applEn^{27}] |	e_1 . **builtIn** ?([e_2, ...]) |	lv^{19}←[first~0, last~10, x~3.14, g~2, f~5];	-- Short for *lv←LabelValue*^{13}[...].
funnyAppl e ?([?argBinding^{27} ?applEn^{27}]) |	e . **funnyAppl** ?([**argBinding** ?applEn]) |	[first~i, last~j]←lv^{19};	-- Assignment to VAR binding
[argBinding^{27}] |	--Binding must coerce to a record, array, or local string-- |		-- (extractor).
s | subrange^{25} | if^{28} | select^{29} | safeSelect^{32} | withSelect^{34}			
Precedence is in bold in rules 19-21. All operators associate to the left except ←, which associates to the right. Application has highest precedence. Subrange only after IN *or* THROUGH. *s only in if*^{28} *and select choices*$^{30\ 33\ 35}$.			
efixOp ::= @ (8) | (7) | (~ | NOT) (3)	VARTOPOINTER | UMINUS | NOT		
fixOp ::= * | / | MOD (6) | + | (5) | ← (0)	TIMES | DIVIDE | REM | PLUS | MINUS | ASSIGN		
lOp ::= ?NOT (?~ (= | < | >) | # |	?NOT (?NOT x′.(EQUAL | LESS | GREATER)[y′] | x′~ = y′ |		
(<= | >=) | IN)	x = y′ OR x′ (< | >) y′ | x′>= y .FIRST AND (x′<= y′.LAST		
-*In 19, 30.*	BUT {BoundsFault =>FALSE}))		
iltIn ::= -- These are enumerated in Table 45.			
nnyAppl ::= FORK | JOIN | WAIT | NOTIFY |	BROADCAST | SIGNAL | ERROR | RETURN WITH ERROR |		
NEW | START | RESTART |TRANSFER WITH |	RETURN WITH		
brange ::= (typeName^{37} |)	LET t′~(**typeName** | INT) . first′~(e_1 | e_1.SUCC) IN	b: BOOL←i IN [1..10]; FOR x: INT IN (0..11) DO ...;	-- Subrange only in types or with IN.
([| ( ) e_1 .. e_2 ( ] |))	t′.MKSUBRANGE[first , (e_2 | e_2.PRED)] BUT	b←(c IN Color^{54}(red..green] OR x IN INT[0..10));	-- The INT is redundant.
-*In 19, 39, 48.*	{BoundsFault =>t′.MKEMPTYSUBRANGE[e_1]}		

otation: item | item = choose one; ?item = zero or one item; terminals: SMALL CAPS, underlined or punctuation other than bold ()?| (parens are terminals only in rules 19, 25, 51.1, 54, | only in 55).
item s ... = zero or more items, separated by s; item s !.. = one or more items, separated by s; with s = ";", a trailing ";" is optional; s is one of: empty "," ";" IN ELSE OR.

breviated non-terminals: b = binding^{13}; d = declaration^{11}; e = expression^{19}; n = name^{56}; s = statement^{14}; t = type^{36}.

mments: = obsolete; = efficiency hack; = unsafe. = machine dependent; n^{56} means n is defined in rule 56. n_2 = n (the subscript is only for the desugaring).

<table>
<tr><th>Syntax</th><th>Meaning</th><th>Examples</th><th>Notes</th></tr>
<tr><td><b>pplication</b> :: = e [?argBinding ?applEn]</td><td>LET m′~e, a ~[argBinding] IN ((m . APPLY ▸a′) ?applEn)</td><td>fh←Files.Open[name~lb.s, mode~Files.read</td><td>-- Keywords are best for multiple a</td></tr>
<tr><td><b>rgBinding</b> :: = (n ~ (e | | TRASH)), !.. |</td><td>(n ~ (e | OMITTED | TRASH)), !.. |</td><td>! AccessDenied=>{...}; FatalError=>{...}];</td><td>-- Semicolons separate choices.</td></tr>
<tr><td>(e | | TRASH), ...</td><td>(e | OMITTED | TRASH), ...</td><td>(GetProcs[j].ReadProc)[k];</td><td>-- The proc can be computed.</td></tr>
<tr><td><i>In 19, 26.</i> TRASH <i>may be written as</i> NULL, ~ <i>as</i> :.</td><td></td><td>file.Read[buffer~b, count~k];</td><td>-- ≡<i>File.Read</i>[<i>file, b, k</i>] (object not</td></tr>
<tr><td><b>applEn</b> :: = ! enChoice^{9}; ...-- <i>In 19, 26.</i></td><td>BUT { enChoice; ... }</td><td>f[i~3, j~ , k~TRASH]; f[i~3, k~TRASH];</td><td>-- <i>j</i> and <i>k</i> may be trash (see default</td></tr>
<tr><td></td><td></td><td>f[3, , TRASH];</td><td>-- Likewise, if <i>i, j,</i> and <i>k</i> are in that</td></tr>
<tr><td>:: = IF e_1 THEN e_2 (ELSE e_3 |)</td><td>IF e_1 THEN e_2 ELSE (e_3 | NULL)</td><td>i←(IF j<3 THEN 6 ELSE 8);</td><td>-- An IF with results must have an</td></tr>
<tr><td><b>elect</b> :: = SELECT e FROM</td><td>LET selector′~e IN</td><td>IF k NOT IN Range THEN RETURN[7];</td><td></td></tr>
<tr><td>choice; ... endChoice</td><td>choice ELSE ... endChoice</td><td>SELECT f[j] FROM</td><td>-- SELECT expressions are also poss</td></tr>
<tr><td><i>The "," is ";" in an expression; also in 32 and 34.</i></td><td>-- <i>ELSE is a separator for repetitions of the choice.</i></td><td><7=>{...};</td><td>-- ≡<i>t</i>:INT~<i>f</i>[<i>j</i>]; IF <i>t</i><7 THEN {...} El</td></tr>
<tr><td><b>hoice</b> :: = ((| relOp^{22}) e_1), !.. =>e_2</td><td>IF ((selector′ (= | <b>relOp</b>) e_1) OR ...) THEN e_2</td><td>IN [7..8]=>{...};</td><td>-- 7, 8=> or =7, =8=>{...} is the</td></tr>
<tr><td><b>ndChoice</b> :: = ENDCASE (| => e_3)</td><td>ELSE (NULL | e_3)</td><td>NOT <=8=>{...};</td><td>-- ENDCASE=>{...} is the same he</td></tr>
<tr><td><i>In 29, 32, 34.</i></td><td></td><td>ENDCASE=>ERROR;</td><td>-- Redundant: choices are exhausti</td></tr>
<tr><td><b>afeSelect</b> :: = WITH e SELECT FROM</td><td>LET v′~e IN</td><td>WITH r SELECT FROM</td><td>-- Assume <i>r</i>: REF ANY in this exam</td></tr>
<tr><td>safeChoice; ... endChoice^{31}</td><td>safeChoice ELSE ... endChoice</td><td>rInt: REF INT=>RETURN[Gcd[rInt↑, 17]];</td><td>-- <i>rInt</i> is declared in this choice onl</td></tr>
<tr><td><b>afeChoice</b> :: = n : t => e_2</td><td>IF ISTYPENOTNIL[v′, t] THEN LET <b>n</b> : t←NARROW[v′, t] IN e_2</td><td>rReal: REF REAL=>RETURN[Floor[Sin[rReal↑]]];</td><td></td></tr>
<tr><td><b>ithSelect</b> :: = WITH (n_1 ~~ e_1 | e_1)</td><td>OPEN v ~~e_1 IN LET n′~(n_1 | NIL), type ~v′.</td><td>ENDCASE=>RETURN[IF r=NIL THEN 0 ELSE 1]</td><td>-- Only the REF ANY <i>r</i> is known he</td></tr>
<tr><td>SELECT (| e_{11}) FROM</td><td>selector′~(e_1.TAG | e_{11}) IN withChoice ELSE ... endChoice</td><td>nr: REF Node^{52}~...; WITH dn~~nr SELECT FROM</td><td>-- See rule 52 for the variant record</td></tr>
<tr><td>withChoice; ... endChoice^{31}</td><td>-- <i>e_{11} must be defaulted except for a</i> COMPUTED <i>variant.</i></td><td>binary=>{nr←dn.b};</td><td>-- <i>dn</i> is a <i>Node.binary</i> in this choice</td></tr>
<tr><td><i>The ~~ may be written as</i> :.</td><td></td><td>unary=>{nr←dn.a};</td><td>-- <i>dn</i> is a <i>Node.unary</i> in this choice</td></tr>
<tr><td><b>ithChoice</b> :: = n_2 => e_2 |</td><td>IF selector′=n_2 THEN OPEN</td><td>ENDCASE=>{nr←NIL};</td><td>-- <i>dn</i> is just a <i>Node</i> here.</td></tr>
<tr><td>n_2, n_2, !.. => e_2</td><td>(BINDP[n′, LOOPHOLE[v ,type [n_2]]] | BINDP[n′, v′]) IN e_2</td><td></td><td></td></tr>
<tr><td><i>ype</i> :: = typeName | builtInType | typeCons</td><td></td><td>P: PROC[b: Buffer^{1}.Handle,</td><td>-- A type from an interface.</td></tr>
<tr><td><b>ypeName</b> :: = n_1 | typeName . n_2 |</td><td></td><td>i: INT←TEXT[20].SIZE];</td><td>-- A bound sequence; only in SIZE.</td></tr>
<tr><td>typeName [e] | n_2 typeName</td><td><b>typeName</b>.SPECIALIZE[e] | <b>typeName</b> . n_2</td><td></td><td></td></tr>
<tr><td><i>In 19, 25, 36, 40.1, 49.</i></td><td>--<i>n_2 names a variant.</i></td><td></td><td></td></tr>
<tr><td><b>uiltInType</b> :: = INT | REAL | TYPE | ATOM | MONITORLOCK | CONDITION |
?UNCOUNTED ZONE | MDSZone | LONG CARDINAL | ?LONG UNSPECIFIED -- <i>See Table 42.</i>
TYPE <i>only as t in a b or an interface's d.</i> INTEGER, CARDINAL, NAT, TEXT, STRING, BOOL, CHAR <i>are predefined.</i></td><td></td><td></td><td></td></tr>
<tr><td><b>ypeCons</b> :: = subrange^{25} | paintedTC$^{40.1}$ | transferTC^{41} | arrayTC^{44} | seqTC^{45} | descriptorTC$^{45.1}$ |</td><td></td><td>TypeIndex: TYPE~[0..256);</td><td>-- A subrange type.</td></tr>
<tr><td>refTC^{46} | listTC^{47} | pointerTC^{48} | relativeTC^{49} | recordTC^{50} | unionTC^{52} | enumTC^{54} | defaultTC^{55}</td><td></td><td>BinaryNode: TYPE~Node^{52}.binary;</td><td>-- A bound variant type.</td></tr>
<tr><td><b>arTC</b> :: = (| READONLY | VAR) t | ANY</td><td>(VAR | READONLY | VAR) t | ANY</td><td></td><td></td></tr>
<tr><td><i>In 11, 4548.</i> ANY <i>only in refTC.</i> VAR <i>only in interface decl.</i></td><td></td><td></td><td></td></tr>
<tr><td><b>aintedTC</b> :: = typeName PAINTED t</td><td>REPLACEPAINT[in: t, from: <b>typeName</b>]</td><td>HV: TYPE~Interface.HistValue PAINTED</td><td>-- See 13 for use.</td></tr>
<tr><td><i>typeName must be an opaque type, t recordTC or enumTC.</i></td><td></td><td>RECORD[...]</td><td></td></tr>
<tr><td><b>ansferTC</b> :: =?safety^{4} xfer ?drType</td><td>MKXFERTYPE[drType, flavor~<b>xfer</b>]</td><td>Enumerate: PROC[</td><td></td></tr>
<tr><td><b>fer</b> :: = PROCEDURE | PROC | PORT |</td><td></td><td>l: RL,</td><td></td></tr>
<tr><td>PROCESS | SIGNAL | ERROR | PROGRAM</td><td></td><td>p: PROC[x: REF ANY] RETURNS [stop: BOOL]]</td><td></td></tr>
<tr><td><b>rType</b> :: = ?fields_1 RETURNS fields_2 | fields_1</td><td>domain~$\textbf{fields}_1$, range~$\textbf{fields}_2$</td><td>RETURNS [stopped: BOOL];</td><td></td></tr>
<tr><td><i>No domain for</i> PROCESS. <i>In 3, 41.</i></td><td></td><td>p2:PROCESS RETURNS[i:INT]←FORK stream.Get;</td><td></td></tr>
<tr><td><b>elds</b> :: = [d^{11}, ...] | [t, ...] | ANY</td><td></td><td>failed: ERROR [reason: ROPE]~CODE;</td><td></td></tr>
<tr><td>ANY <i>only in drType. In 42, 50, 52.</i></td><td></td><td></td><td></td></tr>
<tr><td><b>rrayTC</b> :: = ?PACKED ARRAY ?t_1 OF t_2</td><td>MKARRAY[domain~t_1, range~t_2]</td><td>Vec: TYPE=ARRAY [0..maxVecLen) OF REF TEXT;</td><td></td></tr>
<tr><td><b>eqTC</b> :: =?PACKED SEQUENCE tag^{53} OF t</td><td>MKSEQUENCE[domain~<b>tag</b>, range~t]</td><td>Chars: TYPE~RECORD [text: PACKED SEQUENCE</td><td>-- A record with just a sequence in i</td></tr>
<tr><td><i>Legal only as last type in a recordTC or unionTC.</i></td><td></td><td>len: [0..LAST[INTEGER]] OF CHAR]; ch: Chars;</td><td>-- <i>ch.text</i>[<i>i</i>] or <i>ch</i>[<i>i</i>] refers to an elem</td></tr>
<tr><td><b>escriptorTC</b> :: =</td><td></td><td>v: Vec~ALL[NIL];</td><td></td></tr>
<tr><td>?LONG DESCRIPTOR FOR varTC^{40}</td><td>MKARRAYDESCR[arrayType~<b>varTC</b>]</td><td>dV: DESCRIPTOR FOR ARRAY OF REF TEXT~</td><td></td></tr>
<tr><td><i>varTC must be an array type.</i></td><td></td><td>DESCRIPTOR[v];</td><td></td></tr>
<tr><td><b>efTC</b> :: = REF (varTC^{40} |)</td><td>MKREF[target~(<b>varTC</b> | ANY)]</td><td>ROText: TYPE~REF READONLY TEXT;</td><td>-- NARROW[<i>rl.first, ROText</i>]↑ is a</td></tr>
<tr><td><b>stTC</b> :: = LIST OF varTC^{40}</td><td>MKLIST[range~<b>varTC</b>]</td><td>RL: TYPE~LIST OF REF READONLY ANY; rl:RL;</td><td>-- READONLY TEXT (or error).</td></tr>
<tr><td><b>ointerTC</b>:: =?LONG ?ORDERED ?BASE</td><td>MKPOINTER[target~<b>varTC</b>] |</td><td>UnsafeHandle: TYPE~LONG POINTER TO Vec^{44};</td><td></td></tr>
<tr><td>POINTER ?subrange^{25} ?(TO varTC^{40}) |</td><td></td><td></td><td></td></tr>
<tr><td>POINTER TO FRAME [n]</td><td>MKINSTANCETYPE[<b>n</b>]</td><td></td><td></td></tr>
<tr><td><i>Subrange only in a relativeTC; no typeName^{37} on it.</i></td><td></td><td></td><td></td></tr>
<tr><td><b>elativeTC</b> :: = typeName^{37} RELATIVE t</td><td>MKRELATIVE[range~t, baseType~<b>typeName</b>]</td><td></td><td></td></tr>
<tr><td><i>t must be a pointer or descriptor type, typeName a base pointer type.</i></td><td></td><td></td><td></td></tr>
<tr><td><b>ecordTC</b> :: =?access^{12} (</td><td></td><td>Cell: TYPE~RECORD[next: REF Cell, val: ATOM];</td><td></td></tr>
<tr><td>?MONITORED RECORD fields^{43} |</td><td>MKRECORD[<b>fields</b>] |</td><td>Status: TYPE~MACHINE DEPENDENT RECORD [</td><td>-- Don't omit the field positions.</td></tr>
<tr><td>MACHINE DEPENDENT RECORD</td><td></td><td>channel (0: 8..10): [0..nChannels),</td><td>-- <i>nChannels</i> ≤ 8.</td></tr>
<tr><td>(mdFields | fields^{43}))</td><td>MKMDRECORD[<b>mdFields</b> | <b>fields</b>]</td><td>device (0: 0..3): DeviceNumber,</td><td>-- <i>DeviceNumber</i> held in ≤ 4 bits.</td></tr>
<tr><td><b>dFields</b> :: = [((n pos), ... : --<i>In 50, 52.</i></td><td>MKMDFIELDS[LIST[(LIST[([<b>$n, pos</b>]), ...] , <b>t</b>), ...]]</td><td>stopCode (0: 11..15): Color, fill (0: 4..7): BOOL,</td><td>-- No gaps allowed, but any orderin</td></tr>
<tr><td>?access^{12} t), ...]</td><td></td><td>command (1: 0..31): ChannelCommand];</td><td>-- Bit numbers ≥16 OK; fields can c</td></tr>
<tr><td><b>os</b> :: = (_ e_1 ?(: e_2 .. e_3)) -- <i>In 51, 53.</i></td><td>MKPOSITION[firstWord~e_1, firstBit~e_2, lastBit~e_3]</td><td></td><td>-- word boundaries only if word-al</td></tr>
<tr><td><b>nionTC</b> :: = SELECT tag FROM</td><td>MKUNION[selector~<b>tag</b>, variants~LIST[</td><td>Node: TYPE~MACHINE DEPENDENT RECORD [</td><td>-- <i>rands</i> is a union or variant part.</td></tr>
<tr><td>(n, ... =>(fields^{43} | mdFields^{51} | NULL)), ...</td><td>([labels~LIST[<b>$n</b>, ...], value~<b>fields</b>]), ...]]</td><td>type (0: 0..15): TypeIndex, rator (1: 0..13): Op^{54},</td><td>-- This is the common part.</td></tr>
<tr><td>?, ENDCASE</td><td></td><td>rands (1: 14..79): SELECT n (1: 14..15): * FROM</td><td>-- Both union and tag have pos.</td></tr>
<tr><td><i>Legal only as last type in a recordTC or unionTC.</i></td><td></td><td>nonary=>[],</td><td>-- Type of <i>n</i> is {<i>nonary, unary, binar</i></td></tr>
<tr><td><b>g</b> :: = (n (pos$^{51.1}$ |) : ?access^{12} |</td><td>[([<b>$n</b>, (<b>pos</b> | NIL)] | $COMPUTED′ | $OVERLAID) ,</td><td>unary=>[a (1: 16..47): REF Node],</td><td>-- Can use same name in several var</td></tr>
<tr><td>COMPUTED | OVERLAID) (t | *)</td><td>(<b>t</b> | TYPEFROMLABELS)]</td><td>binary=>[a (1:16..47), b(1:48..79): REF Node]</td><td>-- At least one variant must fill 1: 14</td></tr>
<tr><td><i>In 44, 52. * only in unionTC^{52}.</i></td><td></td><td>ENDCASE];</td><td></td></tr>
<tr><td><b>numTC</b> :: = { n, ... } |</td><td>MKENUMERATION[LIST[<b>$n</b>, ...]] |</td><td>Op: TYPE~{plus, minus, times, divide };</td><td></td></tr>
<tr><td>MACHINE DEPENDENT {((n |) (e) | n), !.. }</td><td>MKMDENUMERATION[LIST[([(<b>$n</b> | NIL), <b>e</b>] | [<b>$n</b>, 1]), ...]]</td><td>Color: TYPE~MACHINE DEPENDENT {</td><td>-- A <i>Color</i> value takes 4 bits; <i>green</i>≡</td></tr>
<tr><td></td><td></td><td>red(0), green, blue(4), (15)}; c: Color;</td><td></td></tr>
<tr><td><b>faultTC</b> :: =</td><td>CHANGEDEFAULT[type~<b>t</b>, (</td><td>-- Except as noted, a constructor or application must mention each name and give it a va</td><td></td></tr>
<tr><td>t ← |</td><td>proc~NIL, trashOK~FALSE] |</td><td>Q: TYPE~RECORD[</td><td>-- Otherwise there's a compile-time</td></tr>
<tr><td>t ← e |</td><td>proc~INLINE λ IN e, trashOK~FALSE] |</td><td>i: INT,</td><td>-- <i>Q</i>[], <i>Q</i>[<i>i</i>~] trash <i>i</i> (not in argBindi</td></tr>
<tr><td>t ← e | _ TRASH |</td><td>proc~INLINE λ IN e, trashOK~TRUE] |</td><td>j: INT←,</td><td>-- No defaulting or trash for <i>j</i>.</td></tr>
<tr><td>t ← TRASH</td><td>proc~t.Trash, trashOK~TRUE])</td><td>k: INT←3,</td><td>-- <i>Q</i>[], <i>Q</i>[<i>k</i>~] leave <i>k</i>=3.</td></tr>
<tr><td><i>defaultTC legal only as the type in a decl in a body^{9} or field^{43} (n: t ← e), in a</i> TYPE <i>binding^{13}, or in</i> NEW. <i>Note the terminal</i> |.</td><td></td><td>l: INT←3 | TRASH,</td><td>-- As <i>k</i>, but <i>Q</i>[<i>l</i>~TRASH] trashes <i>l</i>.</td></tr>
<tr><td>TRASH <i>may be written as</i> NULL.</td><td></td><td>m: INT←TRASH];</td><td>-- <i>Q</i>[], <i>Q</i>[<i>m</i>~] trash <i>m</i>.</td></tr>
<tr><td><i>me</i> :: = letter (letter | digit)...</td><td>-- But not one of the reserved words in Table 31.</td><td>m, x1, x59y, longNameWithSeveralWords: INT;</td><td></td></tr>
<tr><td><b>eral</b> :: = num ?((D|d | B|b) ?num) |</td><td>-- INT literal, decimal if radix omitted or D, octal if B. |</td><td>n: INT~1+12D+2B3+2000B</td><td>-- = 1+12+1024+1024</td></tr>
<tr><td>digit (digit |A|B|C|D|E|F) ... (H|h) ?num |</td><td>-- INT literal in hex; must start with digit. |</td><td>+1H+0FFH;</td><td>-- +1+255</td></tr>
<tr><td>?num . num ?exponent |</td><td>-- REAL as a scaled decimal fraction; note no trailing dot. |</td><td>r1: REAL~0.1+.1+1.0E1</td><td>-- = 0.1+0.1+0.1</td></tr>
<tr><td>num exponent |</td><td>-- With an exponent, the decimal point may be omitted. |</td><td>+1E1;</td><td>-- +0.1</td></tr>
<tr><td>'extendedChar | digit !.. C_ |</td><td>-- CHAR literal; the C form specifies the code in octal. |</td><td>a1: ARRAY [0..3] OF CHAR~['x, '\N, '\', '\141];</td><td></td></tr>
<tr><td>" extendedChar ... " ?L_ |</td><td>[(<b>'extendedChar</b>), ...] -- <i>Rope</i>.ROPE, TEXT, or STRING. |</td><td>r2: ROPE~"Hello.\N...\NGoodbye\F";</td><td></td></tr>
<tr><td>$ n</td><td>-- ATOM literal.</td><td>a2: ATOM~$NameInAnAtomLiteral;</td><td></td></tr>
<tr><td><b>ponent</b> :: = (E|e) ?(+ |) num</td><td>-- Optionally signed decimal exponent.</td><td></td><td></td></tr>
<tr><td><b>um</b> :: = digit !..</td><td></td><td></td><td></td></tr>
<tr><td><b>tendedChar</b> :: = space | \ extension | anyCharNot'"Or\</td><td></td><td></td><td></td></tr>
<tr><td><b>tension</b> :: = digit_1 digit_2 digit_3 |</td><td>-- The character with code $\textbf{digit}_1$ $\textbf{digit}_2$ $\textbf{digit}_3$ B. |</td><td></td><td></td></tr>
<tr><td>(n|N | r|R) | (t|T) | (b|B) |</td><td>-- CR, '\015 | TAB, '\011 | BACKSPACE, '\010 |</td><td></td><td></td></tr>
<tr><td>(f|F) | (l|L) | ' | " | \</td><td>-- FORMFEED, '\014 | LINEFEED, '\012 | ' | " | \</td><td></td><td></td></tr>
</table>

tation: item | item=choose one; ?item=zero or one item; terminals: SMALL CAPS, underlined or punctuation other than bold ()?| (parens are terminals only in rules 19, 25, 51.1, 54, | only in 55).
item s ... =zero or more items, separated by s; item s !.. =one or more items, separated by s; with s=";", a trailing ";" is optional; s is one of: empty "," ";" IN ELSE OR.

breviated non-terminals: **b**=binding^{13}; **d**=declaration^{14}; **e**=expression^{19}; **n**=name^{56}; **s**=statement^{14}; **t**=type^{36}.

mments: =obsolete; =efficiency hack, =unsafe; =machine dependent; n^{56} means n is defined in rule 56, n_1=n (the subscript is only for the desugaring).

FileIO.mesa

This interface contains procs to create file streams. It also specifies a model for the behavior of file streams.

```
Last edited by:
   MBrown on January 6, 1983 8:21 pm
DIRECTORY
   CIFS USING [OpenFile],
   Environment USING [bytesPerPage],
   File USING [Capability],
   IO USING [STREAM],
   Juniper USING [LFH, Transaction],
   Rope USING [ROPE],
   Transaction USING [Handle, nullHandle];
FileIO: CEDAR DEFINITIONS
   IMPORTS
      Transaction =
   BEGIN
   ROPE: TYPE = Rope.ROPE;
   STREAM: TYPE = IO.STREAM;
   bytesPerPage: CARDINAL = Environment.bytesPerPage;
```

File streams

Basic usage

Most programs that create file streams can do so with a very simple call to FileIO.Open. Suppose that r is a Rope.ROPE containing a file name, and your program needs to read characters from the file named by r (local file name or a full path name) using an IO.STREAM s. The call

```
s ← FileIO.Open[r];
```

in your program will accomplish this. If your program will completely rewrite the file named by r (ignoring the old contents and creating a file of that name if none exists), the call is

```
s ← FileIO.Open[r, overwrite];
```

If your program is simply logging output to the file named by r (it does not read the file, but simply adds new characters to the end, and creates a new file of that name if none exists), it calls

```
s ← FileIO.Open[r, append];
```

Finally, if your program will both read and write the file named by r (treats the file as an extendible, random-access sequence of bytes that it updates "in place"), it uses

```
s ← FileIO.Open[r, write];
```

(The last three forms only work for local files unless you have the Pine package loaded and started, in which case they also work for files on the Juniper server. Note that it is an unusual program that requires write mode, while overwrite mode is used frequently.) If you do not wish to land in the debugger if the file name r is misspelled or otherwise garbled, your program should catch FileIO.OpenFailed with why = fileNotFound or why = illegalFileName.

The moral of this tale: doing simple things is simple. You don't need to understand the multitude of parameters to the stream creation procs, since they default correctly for most purposes. You don't need to use any proc from this interface except Open unless you are doing something special. There is only one signal to catch.

At the same time; it is possible to do more ambitious things. If you want to know more, read on.

File stream model

A file is a sequence [0..fileLen) of mutable bytes; the length (fileLen) of a file can be changed.

The state of a file stream is a file file (perhaps opened under a transaction) with length fileLen, an index *streamIndex* IN [0..fileLen], plus a number of readonly flags (*accessOptions, closeOptions*) and a boolean variable *closed* which jointly determine the effect of stream procs. In a section below, "What STREAM operations do when applied to a file STREAM", we describe each stream proc defined in the IO interface (GetChar, PutChar, etc.) in terms of these quantities.

Concurrency

File streams provide no interlocks to control concurrent access to files. Instead, they rely on the underlying file system for concurrency control. Juniper uses locking to provide concurrency control. The Pilot file system used on the local disk provides no concurrency control, though if all of its clients used CIFS and played strictly by the rules there would be no problems. Until that happy day, clients of file streams for local files should take care that if a file stream is open for writing a file, no other readers or writers of that file exist. It is particularly dangerous to have two file streams open for writing the same file.

Individual file streams are independent objects, so there is never any need to synchronize calls to their procedures. If two processes are to share a single file stream, they must synchronize their accesses to that stream at a level above the file stream calls; individual file stream calls (PutChar, PutBlock, etc.) are not guaranteed to be atomic.

Pragmatics

A Pilot file is represented as a sequence of disk pages containing bytesPerPage bytes. There is a single leader page (overhead) that is used to hold file properties such as the length and create date. There is no functional relationship between the length of a file (in bytes) and the size of that file (in pages).

Allocating disk pages to a file is an expensive operation, especially when done incrementally (one page at a time). If a client is creating a file whose eventual length it can estimate, it should use the createLength parameter to Open to allocate enough pages to hold the entire length all at once; if the file already exists the client should create a stream and then extend the file using SetLength. If the file stream implementation must extend a file it will always extend it by several pages; if asked to shorten a file it will simply adjust the length field in the leader page without freeing any pages. To free extra pages when a stream is closed, use the truncatePagesOnClose CloseOption in the call that creates the stream.

Creating a file STREAM

FileSystem: TYPE = { pilot, juniper };

```
Trans: TYPE = RECORD [
    body: SELECT type: FileSystem FROM
        pilot => [trans: Transaction.Handle],
        juniper => [trans: Juniper.Transaction ← NIL],
        ENDCASE];
```

Used to unify the two types of transaction, for Open only.

OpenFailed: SIGNAL [why: OpenFailure, fileName: ROPE] RETURNS [retryFileName: ROPE];

OpenFailure: TYPE = { fileNotFound, illegalFileName, fileAlreadyExists, cantUpdateTiogaFile, wrongTransactionType, unknownFileCapability, notImplementedYet };

This signal is raised by the various file stream creation procedures.

CreateOptions: TYPE = {none, newOnly, oldOnly};
See description of Open below.

AccessOptions: TYPE = {read, append, write, overwrite};
Used at the time a stream is created to specify the set of operations allowed on a stream. Disallowed operations raise IO.Error[NotImplementedForThisStream] when called. The characteristics of each option:

- read: PutChar, PutBlock, SetLength are disallowed, and the initial streamIndex is 0.
- overwrite: all operations are allowed, the file is truncated to zero length at stream creation time, and the initial streamIndex is 0.
- append: GetChar, GetBlock, SetLength, SetIndex are disallowed, and the initial streamIndex is fileLen.
- write: all operations are allowed, and the initial streamIndex is 0.

CloseOptions: TYPE = CARDINAL;
CloseOptions determine optional processing during Flush and Close calls. If commitAndReopenTransOnFlush, then Flush "checkpoints" the transaction being used by the stream. If truncatePagesOnClose, then Close causes extra pages of the file to be freed. If finishTransOnClose, then Close causes the transaction to be committed or aborted, according to the abort flag to Close.

```
   noCloseOptions: CloseOptions = 0;
   commitAndReopenTransOnFlush: CloseOptions = 1;
   truncatePagesOnClose: CloseOptions = 2;
   finishTransOnClose: CloseOptions = 4;
   defaultCloseOptions: CloseOptions = truncatePagesOnClose + finishTransOnClose;
```

RawOption: TYPE = BOOL;
This parameter determines the mode of access to Tioga format files. If raw = FALSE and file is in Tioga format, then if accessOptions = read, read only the plain text portion of the file (ignore "looks" and nodes with the comment property); if accessOptions # read or overwrite, raise OpenFailure [cantUpdateTiogaFile]. If raw = TRUE or file is not in Tioga format, then operate on the entire file.

```
StreamBufferParms: TYPE = RECORD [
   bufferSize: INT [2 .. 127],
      Specifies the number of pages of VM used by the stream for buffering. Juniper streams
        always use 1 page of buffering.
   bufferSwapUnitSize: INT [1 .. 32]];
      Specifies the number of pages in each uniform swap unit of the stream buffer.
   defaultStreamBufferParms: StreamBufferParms = [bufferSize: 25, bufferSwapUnitSize: 5];
      Good for most purposes.
   minimumStreamBufferParms: StreamBufferParms = [bufferSize: 2, bufferSwapUnitSize: 1];
      Good when opening a stream just to create a file or set its length.
```

```
Open: PROC [
   fileName: ROPE,
   accessOptions: AccessOptions ← read,
   createOptions: CreateOptions ← none,
   closeOptions: CloseOptions ← defaultCloseOptions,
   transaction: Trans ← [juniper[]],
   raw: RawOption ← FALSE,
   createLength: INT ← 5 * bytesPerPage,
   streamBufferParms: StreamBufferParms ← defaultStreamBufferParms]
   RETURNS [STREAM];
   ! OpenFailed with why =
      fileNotFound: createOptions = oldOnly and file does not exist (including server not found
    when fileName is a full path name).
   illegalFileName: syntax or other error caused directory lookup to fail.
   fileAlreadyExists: createOptions = newOnly and file already exists.
   cantUpdateTiogaFile: raw = FALSE, accessOptions # read, and file is in Tioga format.
   wrongTransactionType: transaction is a non-null Pilot transaction but fileName is a Juniper
    file, or vice-versa.
   notImplementedYet: CIFS access is implied but accessOptions # read.
! Juniper.Error with why = transactionReset, notDone: don't know how to handle these.
! CIFS.Error with code # illegalFileName, noSuchFile, noSuchHost: don't know how to handle
 these.
! Volume.InsufficientSpace: couldn't create file on local volume as requested.
```

Create a new stream on the file specified by fileName (a full path name).

If accessOptions = read then createOptions = oldOnly are assumed.

If createOptions = none or newOnly and the file specified by fileName does not exist, then create it, with initial size (in pages excluding leader page) of createLength/bytesPerPage (rounded up), and initial length (in bytes) zero.

If fileName specifies the server "Juniper" and transaction either has pilot variant and contains Transaction.nullHandle or has juniper variant but contains NIL ([juniper[]] produces this), then call Juniper.UserInit[], and create a new transaction by calling Juniper.BeginTransaction.

```
StreamFromOpenFile: PROC [
   openFile: CIFS.OpenFile,
   accessOptions: AccessOptions ← read,
   closeOptions: CloseOptions ← defaultCloseOptions,
   transaction: Transaction.Handle ← Transaction.nullHandle,
   raw: RawOption ← FALSE,
   streamBufferParms: StreamBufferParms ← defaultStreamBufferParms]
   RETURNS [STREAM];
   ! OpenFailed with why =
      notImplementedYet: accessOptions # read.
```

Create a new file stream on the open file.

```
StreamFromCapability: PROC [
   capability: File.Capability,
   accessOptions: AccessOptions ← read,
   closeOptions: CloseOptions ← defaultCloseOptions,
   fileName: ROPE ← NIL,
   transaction: Transaction.Handle ← Transaction.nullHandle,
   raw: RawOption ← FALSE,
   streamBufferParms: StreamBufferParms ← defaultStreamBufferParms]
   RETURNS [STREAM];
   ! OpenFailed with why =
      cantUpdateTiogaFile: raw = FALSE, accessOptions # read, and file is in Tioga format.
      unknownFileCapability: no file identified by capability is present on the local disk volume.
```

Create a new stream on the file identified by capability (note that StreamFromCapability adds permissions to capability as required to perform the accesses as specified in accessOptions). fileName is stored in the stream, for diagnostic purposes when a stream error occurs.

```
CapabilityFromStream: PROC [
   self: STREAM]
   RETURNS [File.Capability];
   ! IO.Error[NotImplementedForThisStream]: self is not a stream on a Pilot file
```

Return the file capability for the file underlying self. This capability has the permissions that are necessary to perform stream operations.

```
StreamFromLFH: PROC [
   lfh: Juniper.LFH,
   accessOptions: AccessOptions ← read,
   closeOptions: CloseOptions ← defaultCloseOptions,
   fileName: ROPE ← NIL,
   transaction: Juniper.Transaction,
   raw: RawOption ← FALSE]
   RETURNS [STREAM];
   ! OpenFailed with why =
      cantUpdateTiogaFile: raw = FALSE, accessOptions # read, and file is in Tioga format.
      unknownFileCapability: no file with the given LFH is present on Juniper.
```

Create a new stream on the Juniper file identified by lfh. fileName is a debugging aid, for diagnostic purposes when a stream error occurs.

END.

What STREAM operations do when applied to a file STREAM

Basic STREAM procs

IO.GetChar: PROC [self: STREAM] RETURNS [CHAR]

If streamIndex = fileLen then ERROR IO.EndOfStream. Else return file[streamIndex], and set streamIndex ← streamIndex + 1.

IO.PutChar: PROC [self: STREAM, char: CHAR]

If streamIndex = fileLen then fileLen ← fileLen + 1. Then set file[streamIndex] ← char, streamIndex ← streamIndex + 1.

IO.GetBlock: PROC [self: STREAM, block: REF TEXT, startIndex: NAT ← 0, stopIndexPlusOne: NAT ← LAST[NAT]] RETURNS [nBytesRead: NAT]

Equivalent to (but faster than)

```
stopIndexPlusOne ← MIN [block.maxLength, stopIndexPlusOne];
nBytesRead: NAT ← MIN[fileLen-streamIndex, stopIndexPlusOne-startIndex];
FOR i: NAT IN [0..nBytesRead) DO block[startIndex+i] ← GetChar[self] ENDLOOP;
IF nBytesRead # 0 THEN block.length ← startIndex + nBytesRead;
RETURN[nBytesRead]
```

IO.PutBlock: PROC [self: STREAM, block: REF READONLY TEXT, startIndex: NAT ← 0, stopIndexPlusOne: NAT ← LAST[NAT]]

Equivalent to (but faster than)

```
IF stopIndexPlusOne > block.maxLength THEN stopIndexPlusOne ← block.length;
FOR i: NAT IN [startIndex..stopIndexPlusOne) DO PutChar[self, block[i]] ENDLOOP;
```

IO.UnsafeGetBlock: UNSAFE PROC [self: STREAM, block: IO.UnsafeBlock] RETURNS [nBytesRead: INT]

Equivalent to (but faster than)

```
IF block.startIndex < 0 OR block.stopIndexPlusOne < 0 THEN
  ERROR IO.Error[BadIndex];
nBytesRead: INT ← MIN[fileLen-streamIndex, block.stopIndexPlusOne-block.startIndex];
FOR i: INT IN [0..nBytesRead) DO
  block.base↑[block.startIndex+i] ← GetChar[self] ENDLOOP;
RETURN[nBytesRead]
```

IO.UnsafePutBlock: PROC [self: STREAM, block: IO.UnsafeBlock]

Equivalent to (but faster than)

```
IF block.startIndex < 0 OR block.stopIndexPlusOne < 0 THEN
  ERROR IO.Error[BadIndex];
FOR i: INT IN [block.startIndex..block.stopIndexPlusOne) DO
  PutChar[self, block.base↑[i]] ENDLOOP;
```

IO.CharsAvail: PROC [self: STREAM] RETURNS [BOOL]
Return TRUE.

IO.EndOf: PROC [self: STREAM] RETURNS [BOOL]
Return streamIndex = fileLen.

IO.Flush: PROC [self: STREAM]
Force all stream writes since the time of stream creation or the preceding Flush to be written to disk. If commitAndReopenTransOnFlush then first commit trans, then begin a new transaction as trans.

IO.Reset: PROC [self: STREAM]
If accessOptions # append then set streamIndex ← 0.

IO.Close: PROC [self: STREAM, abort: BOOL ← FALSE]
If NOT abort, then force all stream actions since stream creation or the preceding Flush to be written to disk; otherwise discard them. If truncatePagesOnClose, then discard unused pages from end of file. If finishTransOnClose, then commit or abort trans depending on the state of abort. Invalidate self (all operations on self other than Flush, Reset, and Close will raise ERROR IO.Error[StreamClosed]; these three do nothing).

Less basic STREAM procs

IO.PutBack: PROC [self: STREAM, char: CHAR]
If streamIndex = 0 then ERROR IO.Error[IllegalPutBack]. Otherwise, set streamIndex ← streamIndex - 1, and if file[streamIndex] # char then ERROR IO.Error[IllegalPutBack].

IO.PeekChar: PROC [self: STREAM] RETURNS [char: CHAR];
Equivalent to:

```
c: CHAR ← self.GetChar[]; self.PutBack[c]; RETURN [c];
```

File-specific STREAM procs

IO.GetIndex: PROC [self: STREAM] RETURNS [INT]
Return streamIndex.

IO.SetIndex: PROC [self: STREAM, index: INT]
ERROR IO.EndOfStream if index > fileLen. Set streamIndex ← index.

IO.GetLength: PROC [self: STREAM] RETURNS [INT]
Return fileLen.

IO.SetLength: PROC [self: STREAM, length: INT]
Set fileLen ← l, then set streamIndex ← MIN[streamIndex, fileLen]. The contents of file[oldFileLen .. fileLen) are undefined.

Change Log

Created by MBrown on 7-Dec-81 10:33:20
By editing FileByteStream.

Changed by MBrown on March 26, 1982 4:41 pm
Added "raw" parm to stream create operations.

Changed by MBrown on August 23, 1982 10:25 pm

Handle -> STREAM (-> Stream in proc names), introduce FileIO.OpenFailed, CIFS access in readonly mode, initial file size in Open, buffer pages and swap units. Format this file using Tioga nodes.

Changed by MBrown on October 23, 1982 9:59 pm

Default transaction to Open is now [juniper[]], since this does not require the average user to import Transaction (to use nullHandle). Format to use current Cedar style.

Changed by MBrown on January 6, 1983 8:15 pm

Attempted to reduce the confusion over what is comment and what is not comment in the interface, and to improve the explanation of AccessOptions.

IO.mesa

Top-level stream and I/O interface for Cedar

Last edited by: Teitelman on January 12, 1983 3:01 pm

Note: this interface is heavily structured using Tioga Nodes. Thus, you can use the level clipping to obtain a table of contents, and to successively refined levels of detail in the areas you are especially interested.

```
DIRECTORY
    Ascii USING [CR, SP, TAB, LF, BS, ControlA, ControlX, FF, NUL, ESC, DEL, BEL],
    Atom USING [PropList],
    Rope USING [ROPE],
    RTBasic USING [TV, Type],
    System USING [GreenwichMeanTime],
    Time USING [Current]
    ;

IO: CEDAR DEFINITIONS

IMPORTS Time

= BEGIN
```

Types

```
    STREAM: TYPE = REF STREAMRecord;
    STREAMRecord: TYPE = RECORD[
        streamProcs: REF StreamProcs, -- see below
        streamData: REF ANY, -- data private to streamProcs
        propList: Atom.PropList ← NIL, -- permits associating arbitrary information with stream. see StoreData, Lookup
        backingStream: STREAM ← NIL -- for use with layered streams.
        ];
    Handle: TYPE = STREAM; -- for backward compatibility
    ROPE: TYPE = Rope.ROPE;
    TV: TYPE = RTBasic.TV;
    Type: TYPE = RTBasic.Type;
    Base: TYPE = [2..36];
    GreenwichMeanTime: TYPE = System.GreenwichMeanTime;
```

Creating Streams

The following procedures enable the user to create input and/or output streams to various entities such as viewers, files, ropes, as well as streams that are layered on top of another stream, the backing stream, so as to modify or augment the behaviour of the backing stream, such as edited streams, dribble streams, filtered comment streams, etc. All of these streams are defined in terms of a general mechanism for implementing streams, CreateProcsStream, which is described below under the section Defining New Streams.

```
    CreateViewerStreams: PROC [name: ROPE] RETURNS [in: STREAM, out: STREAM];
        creates a new viewer with indicated name, and returns two streams "connected" to that viewer, one
         for input and one for output. The input stream is an edited stream. Note that
         ViewerIO.CreateViewerStreams...
```

enables creating streams that are connected to an existing viewer, or specifying that the input stream is not an edited stream, e.g. so that the client can then specify his own deliverWhen procedure when creating an edited stream on top of the input stream.

CreateFileStream *To create a stream for reading/writing on a file use FileIO.Open.*

CreateMessageWindowStream *To create a stream for writing to the MessageWindow, use ViewerIO.CreateMessageWindowStream.*

CreateInputStreamFromRope, RIS: PROC [rope: ROPE, oldStream: STREAM ← NIL] RETURNS [stream: STREAM];

stream gets chars from user's rope. If oldStream is non-NIL, it is reused rather than allocating space for a new stream.

CreateOutputStreamToRope, ROS: PROC RETURNS [stream: STREAM];

stream puts chars into a rope which can be retrieved via GetOutputStreamRope, described below.

Fine point...

in order to perform this operation efficiently, a REF TEXT is obtained from a pool of scratch REF TEXTs. As a result, it is a good idea to close the stream when it is no longer going to be used in order to return the scratch structure to a common pool. However, if you do not, e.g. if an error occurs, all that will happen is that the structure will be garbage collected when it is no longer referenced, and new scratch structure will be allocated to take its place in the pool.

CreateInputStreamFromText: PROC [text: REF READONLY TEXT, oldStream: STREAM ← NIL] RETURNS [stream: STREAM];

stream gets chars from user's REF TEXT. If oldStream is non-NIL, it is reused rather than allocating space for a new stream.

CreateOutputStreamToText: PROC [text: REF TEXT, oldStream: STREAM ← NIL] RETURNS [stream: STREAM];

stream puts characters into text. If oldH is non-NIL, it is reused rather than allocating space for a new stream.

CreateEditedStream: PROCEDURE [in: STREAM, echoTo: STREAM, deliverWhen: DeliverWhenProc ← IsACR] RETURNS [stream: STREAM];

creates a layered inputstream on top of in which buffers input characters and allows the user to perform character editing with ↑A, ↑W, ↑Q. Typing a DEL will cause CreateEditedStream to raise IO.Signal[Rubout]. Default is to buffer to carriage return. Input characters are echoed to and erased from echoTo. ChangeDeliverWhen can be used to change buffering condition.

DeliverWhenProc: TYPE = PROC[char: CHAR, stream: STREAM] RETURNS[BOOL];

IsACR: DeliverWhenProc; -- *returns TRUE for CR.*

ChangeDeliverWhen: PROC [self: STREAM, proc: DeliverWhenProc] RETURNS [oldProc: DeliverWhenProc];

currently implemented by Edited Stream and Viewer Edited Stream (See ViewerIO).

GetBufferContents: PROC [self: STREAM] RETURNS[buffer: ROPE];

returns contents of the buffer without disturbing. For use in conjunction with DeliverWhenProcs which might need to see what has been typed in order to decide whether to terminate, e.g. parenthesis balancing, etc. Currently implemented by Edited Stream and Viewer Edited Stream.

CreateDribbleStream: PROCEDURE [stream: STREAM, dribbleTo: STREAM, flushEveryNChars: INT ← -1] RETURNS [s: STREAM];

creates a layered stream, s, on top of stream, such that characters read from/printed to s will be *read from/printed to stream, and also output to dribbleTo. If flushEveryNChars is > 0, will automatically call Flush on dribbleTo every nChars.*

CreateBufferedOutputStream: PROC [stream: STREAM, deliverWhen: DeliverWhenProc ← IsACR, bufferSize: NAT ← 256] RETURNS [s: STREAM];
creates an output stream on top of stream such that characters are buffered until (a) the bufferSize is exceeded, or (b) deliverWhen[char] is TRUE, or (c) an explicit Flush is performed, at which point the characters are output to stream using PutBlock.

CreateFilterCommentsStream: PROCEDURE [stream: IO.STREAM] RETURNS [s: STREAM];
creates a input stream on top of stream such that any comments are filtered out, i.e. the characters comprising the comments are not returned by GetChar[s]. A comment consists of that sequence of characters beginning with -- and ending with -- or a CR. In the latter case, the CR is not considered to be part of the comment.

AppendStreams: PROC [in: STREAM, from: STREAM, appendToSource: BOOL ← TRUE];
modifies 'in' so that an attempt to read characters from 'in' will cause the characters to be obtained from 'from' until 'from' runs dry, at which point 'in' is returned to its previous state. Note that it is permissible to nest, i.e. concatenate, more than one stream in this fashion.
If appendToSource = TRUE then modification is done to the root stream, i.e. descends through non-NIL backingstreams. For example, if in is an edited stream, then this will cause characters to be inserted into buffer, from which they can be backspaced over, exactly as though they had been typed, rather than delivered to the client immediately. If in is a dribbleStream, then the characters will be dribbled etc.

noWhereStream: STREAM;
output stream that simply discards its characters.

Unformatted stream procedures: GetChar, PutChar, Close, etc.

Note: Not all of the following are implemented for all streams, e.g. GetLength is not implemented (does not make sense) for a keyboard stream. If an operation is invoked that is not implemented, the Error[NotImplementedForThisStream] is raised. However, many operations are defaulted even when they are not implemented. See discussion below of exactly which operations are implemented for which streams.)

Input Procedures: GetChar, EndOf, CharsAvail, GetBlock, Backup, PeekChar, SetEcho, ...

GetChar: PROC[self: STREAM] RETURNS[CHAR]
= INLINE {RETURN[self.streamProcs.getChar[self]]};
EndOf: PROC[self: STREAM] RETURNS [BOOL]
= INLINE {RETURN[self.streamProcs.endOf[self]]};
true if reading from this stream would raise an endofstream error.
CharsAvail: PROC[self: STREAM] RETURNS [BOOL]
= INLINE {RETURN[self.streamProcs.charsAvail[self]]};
true if self.GetChar[] would return a character without waiting.
Note: ...

> *For most streams, EndOf is equivalent to ~CharsAvail, and in fact, if CharsAvail is not specified, a default procedure is supplied which will returns as its value NOT EndOf. However, note that EndOf is always FALSE for a keyboard stream, i.e. it is always ok to try to read from a keyboard stream, even when CharsAvail is FALSE, i.e. there are no more characters available now.*

GetBlock: PROC[self: STREAM, block: REF TEXT, startIndex: NAT ← 0, stopIndexPlusOne: NAT ←

LAST[NAT]] RETURNS[nBytesRead: NAT]
= INLINE {RETURN[self.streamProcs.getBlock[self, block, startIndex, stopIndexPlusOne]]};
value is number of characters actually returned. Equivalent to, but more efficient than, n calls to getChar...
for those streams that provide an implementation for getBlock, e.g. file streams. Otherwise, the getBlock is exactly the same as, i.e. is accomplished by, performing n calls to GetChar. See discussion of default procedures below.

UnsafeGetBlock: UNSAFE PROC[self: STREAM, block: UnsafeBlock] RETURNS[nBytesRead: INT]
= UNCHECKED INLINE {RETURN[self.streamProcs.unsafeGetBlock[self, block]]};
value is number of characters actually returned. The type UnsafeBlock...
is defined below in the section "Private and Semi-private types."

Backup: PROC [self: STREAM, char: CHAR];
Used when input went too far...
Note that Backup is an operation on the input stream. It modifies self *so that the next* getChar[self] *will return the same character and leave* self *in the state it was in before the Backup. The implementor is only required to implement Backup for those situations where char is in fact the character that was last read by getChar. (The character argument is required in order to provide a generic implementation for the operation, i.e. when the particular stream does not implement Backup itself).*

PeekChar: PROC[self: STREAM] RETURNS [char: CHAR];
does a GetChar followed by a Backup, i.e. effectively looks at the next character without actually reading it.

SetEcho: PROC[self: STREAM, echoTo: STREAM] RETURNS [oldEcho: STREAM];
causes every character read from stream to be echoed to echoTo if non-NIL.

Output procedures: PutChar, PutBlock, Flush, EraseChar, GetOutputStreamRope, ...

PutChar: PROC[self: STREAM, char: CHAR]
= INLINE {self.streamProcs.putChar[self, char]};

PutBlock: PROC[self: STREAM, block: REF READONLY TEXT, startIndex: NAT ← 0, stopIndexPlusOne: NAT ← LAST[NAT]]
= INLINE {self.streamProcs.putBlock[self, block, startIndex, stopIndexPlusOne]};

UnsafePutBlock: PROC[self: STREAM, block: UnsafeBlock]
= INLINE {self.streamProcs.unsafePutBlock[self, block]};

Flush: PROC [self: STREAM]
= INLINE {self.streamProcs.flush[self]};
causes characters that have been output to stream, but not yet sent, e.g. because of buffering, to be sent

EraseChar: PROC[self: STREAM, char: CHAR];
erases char from output stream, e.g. erases the bits from the display...
Note that EraseChar, is an operation on the output *stream. Whatever the corresponding input stream might have done about the character that is being erased is an entirely separate operation EraseChar just specifies what is to be done to the output stream.*

CurrentPosition: PROC[self: STREAM] RETURNS[position: INT];
number of characters output since last CR.

NewLine: PROC [self: STREAM]
= INLINE {IF self.CurrentPosition[] # 0 THEN self.PutChar['\n]};
outputs a CR unless already at beginning of the line.

SpaceTo: PROC [self: STREAM, n: INT, nextLine: BOOLEAN ← TRUE];
spaces to position n. If current position already beyond n and nextLine is TRUE, then new line and n spaces.

GetOutputStreamRope: PROC [self: STREAM] RETURNS [ROPE];
used in conjuction with the CreateOutputStreamToRope to get the resulting rope

Miscellaneous Procedures: Reset, Close, UserAbort, GetIndex, SetIndex, GetLength, SetLength

Reset: PROC[self: STREAM]
= INLINE {self.streamProcs.reset[self]};
restores the stream to a clean state, e.g. may flush or restore internal buffers. typically invoked following an error,

Close: PROC[self: STREAM, abort: BOOL ← FALSE]
= INLINE {self.streamProcs.close[self, abort]};

UserAbort: PROC [stream: STREAM] RETURNS [abort: BOOLEAN];
If unimplemented for corresponding stream or backing stream, returns FALSE. TRUE means user indicated desire to abort operation. It is up to program to take the appropriate steps, which should culminate in raising IO.UserAborted. Currently implemented for ViewerStreams, TRUE if user types control-Del to the viewer, or if the viewer is connected a userexec, can also be set by clicking STOP.

SetUserAbort: PROC [stream: STREAM];
sets the user abort, i.e. provides a programmable way to abort (the operation currently running in) an exec. Generates an error if not implemented for the corresponding stream. Currently only implemented for ViewerStreams.

ResetUserAbort: PROC [stream: STREAM] ;
turn the abort bit off, if implemented. Otherwise, nop.

GetIndex: PROC[self: STREAM] RETURNS [index: INT]
= INLINE {RETURN[self.streamProcs.getIndex[self]]};

SetIndex: PROC[self: STREAM, index: INT]
= INLINE {self.streamProcs.setIndex[self, index]};

GetLength: PROC[self: STREAM] RETURNS [length: INT];

SetLength: PROC[self: STREAM, length: INT];

Formatted stream procedures: Put, PutF, PutList, PutRope, PutTV, PutType, ...

Put: PROC [stream: STREAM, v1, v2, v3: Value ← [null[]]];
The basic output routine, handles any type via Values, defined later. Provides convenient default behaviour for simple output, i.e. outputs integers and cardinals in decimal, time using both date and time, etc. Example: s.Put[int[x+y], rope[" is greater than "], int[x]] would produce '8 is greater than 5', for x = 5 and y = 3. For more control over the output...
use PutF, described below, which allows specification of field Width, base, etc.

Put only takes three value arguments...
so that the procedure call can be "fast" (i.e. take 11 or fewer words on the stack). PutList can be used for calls specifying more arguments at the expense of the allocation of space for the LIST, or the user can write several calls to Put.

PutF: PROC [stream: STREAM, format: ROPE ← NIL, v1, v2, v3, v4, v5: Value ← [null[]]];
produces output according to a collection of values and a format control string which can include embedded format codes. For example: s.PutF["%d is greater than %d", int[x+y], int[x]] is equivalent to the above call on Put. For more discussion, see section "PF Package".

format = NIL is equivalent to "%g%g%g..." which means to print each value the same as would be printed by Put.

Note that PutF is a slow procedure call, i.e. its arguments require more than 11 words on the stack.

PutF may raise signals. RESUME will always work and will either do nothing or print "#####".

PutList, PutL: PROC [stream: STREAM, list: LIST OF Value];
for outputting more than three quantities via a single call to Put.

PutFList, PutFL: PROC [stream: STREAM, format: ROPE ← NIL, list: LIST OF Value];

PutFToRope, PutFR: PROC [format: ROPE ← NIL, v1, v2, v3, v4, v5: Value ← [null[]]] RETURNS [r: ROPE]

= INLINE
{stream: STREAM ← CreateOutputStreamToRope[]; stream.PutF[format, v1, v2, v3, v4, v5]; r ← stream.GetOutputStreamRope[]; stream.Close[];
};

PutText: PROC [self: STREAM, t: REF READONLY TEXT]
= INLINE {self.PutBlock[t]};
much more efficient than self.Put[text[t]];

PutRope: PROC [self: STREAM, r: ROPE];
much more efficient than self.Put[rope[r]];

PutTV: PROCEDURE[stream: STREAM, tv: TV, depth: INT ← 4, width: INT ← 32, verbose: BOOL ← FALSE];
depth and width control the cutoff for printing of the tv, verbose specifies whether to print additional information, e.g. the octal value for global frames, etc. stream.Put[tv[x]] is equivalent to stream.PutTV[x], and is the proper way to print tvs for most applications..

PutType: PROC[stream: STREAM, type: Type, depth: INT ← 4, width: INT ← 32, verbose: BOOLEAN ← FALSE];
If verbose is FALSE, just the name of the type is printed. If verbose is TRUE, the undertype is also printed for named types. For example, if x is of type BreakProc, then stream.PutType[x, FALSE] prints: IO.BreakProc, whereas stream.PutType[x, TRUE] prints as: IO.BreakProc = PROC [c: CHAR] RETURNS [IO.BreakAction].
In addition, the range type is printed for undertypes that are refs or pointers. For example, if x is of type STREAM, stream.PutType[x, TRUE] prints as: IO.STREAM = REF IO.STREAMRecord; (and on the next line) IO.STREAMRecord: TYPE = RECORD [...];
stream.Put[type[x]] is equivalent to stream.PutType[type]

PutSignal: PROC [stream: IO.STREAM, signalTV, argsTV: TV ← NIL];
Prints the indicated signal on stream. If signalTV = NIL, prints the "current" signal, i.e. for use inside of a catch phrase.

Procedures for constructing Values: atom, bool, card, char, int, real, refAny, rope, tv, ...

saves writing two extra brackets. the type Value is defined in section 4.

atom: PROC [v: ATOM] RETURNS [Value]
= INLINE {RETURN[[atom[v]]]};
bool: PROC [v: BOOL] RETURNS [Value]
= INLINE {RETURN[[boolean[v]]]};
card: PROC [v: LONG CARDINAL] RETURNS [Value]
= INLINE {RETURN[[cardinal[v]]]};
char: PROC [v: CHAR] RETURNS [Value]
= INLINE {RETURN[[character[v]]]};
int: PROC [v: INT] RETURNS [Value]
= INLINE {RETURN[[integer[v]]]};
real: PROC [v: REAL] RETURNS [Value]
= INLINE {RETURN[[real[v]]]};
refAny: PROC [v: REF READONLY ANY] RETURNS [Value]
= INLINE {RETURN[[refAny[v]]]};
rope: PROC [v: ROPE] RETURNS [Value]
= INLINE {RETURN[[rope[v]]]};
string: PROC [v: LONG STRING] RETURNS [Value]
= INLINE {RETURN[[string[v]]]};
text: PROC [v: REF READONLY TEXT] RETURNS [Value]
= INLINE {RETURN[[text[v]]]};
time: PROC [v: GreenwichMeanTime ← Time.Current[]] RETURNS [Value]

= INLINE {RETURN[[time[v]]]}; -- *i.e. time[] is the current time*
tv: PROC [v: RTBasic.TV] RETURNS [Value]
= INLINE {RETURN[[tv[v]]]};
type: PROC [t: RTBasic.Type] RETURNS [Value]
= INLINE {RETURN[[type[t]]]};

Parsing the input stream: GetCedarToken, GetToken, GetInt, GetReal, GetRefAny, ...

Parsing the input stream as a sequence of characters: GetSequence

CharProc: TYPE = PROC [char: CHAR] RETURNS [quit: BOOL ← FALSE, include: BOOL ← TRUE];

GetSequence: PROC [stream: STREAM, charProc: CharProc ← LineAtATime] RETURNS[ROPE];
reads characters from stream until EOF or quit is TRUE, including those characters for which include is TRUE. For example, the definition of LineAtATime is RETURN[char = CR, FALSE]. Thus s.GetSequence[] will read to the next CR and return the characters read as a rope.

LineAtATime: CharProc;
EveryThing: CharProc:
definition is {RETURN[FALSE, TRUE]} *i.e. in.GetSequence[EveryThing] returns the contents of in.*

Parsing the input stream as a sequence of Cedar Tokens: GetCedarToken

GetCedarToken: PROC [stream: STREAM] RETURNS[ROPE];
uses CedarScanner to scan the input stream for the next mesa token, which is returned as a rope, e.g. for the stream IO.RIS["[$foo, a.b, 3.2]", successive calls on GetCedarToken would return the nine tokens "[" "$foo" "," "a" "." "b" "," "3.2" and "]". GetCedarToken automatically filters out all comments. For convenience in use with the interpreter, & is treated as a regular character, rather than causing a syntax error, i.e. "&23" will parse as a single token.

GetAtom, GetBool, GetCard, GetInt, GetReal, and GetRope described below provide ways of parsing the input stream into objects of the corresponding type. If the client knows what type of object is next expected, he can use the input routine for that type, e.g. i: INT ← stream.GetInt[], rather than calling GetCedarToken and then converting the resulting rope to the corresponding type. GetRefAny provides a compromise between these two positions by returning an object that is a REF to the corresponding type, e.g. REF INT, REF BOOL, etc. The client can then discriminate on the type of the REF.

Basic input routines for each value type: GetCard, GetInt, GetReal, ...

Example: i: INT ← stream.GetInt[];

Note: all of the following procedures use the CedarScanner to obtain the next token. If the token is of the appropriate 'kind', the corresponding value is returned, otherwise, raise SyntaxError with an appropriate message.

SyntaxError: ERROR[stream: IO.STREAM, msg: ROPE ← NIL];

GetAtom: PROC [stream: STREAM] RETURNS [ATOM];
raises SyntaxError if next token is not an atom.
GetBool: PROC[stream: STREAM] RETURNS[BOOLEAN];
returns TRUE if next token is TRUE, FALSE if FALSE, otherwise SyntaxError.
Note: if you are planning on using GetBool to ask for confirmation:
UserExec.ExecConfirm and UserExec.ViewerConfirm provide convenient and standard ways of asking for confirmation. These procedures also post menu buttons and handle the case where

the user has typed ahead before the need for confirmation arose.

GetCard: PROC [stream: STREAM] RETURNS [LONG CARDINAL];
GetInt: PROC [stream: STREAM] RETURNS [INT];
GetReal: PROC [stream: STREAM] RETURNS [REAL];
GetRope: PROC [stream: STREAM] RETURNS [ROPE];
raises SyntaxError if next token is not a rope.

GetId: PROC [stream: STREAM] RETURNS [ROPE];
raises SyntaxError if next token is not an identifier.

Reading Ref Anys: GetRefAny, GetRefAnyLine

GetRefAny: PROC [stream: STREAM] RETURNS [REF ANY];
Use cedar scanner to parse the input stream, then return the resulting token as a REF to the appropriate type, e.g. REF INT, REF REAL, REF BOOL, ATOM, ROPE, or LIST OF REF ANY, etc. ↑ in front of an object is ignored in order that read and print be inverses, e.g. ↑3 will produce NEW[INT ← 3]. Identifiers are read in as atoms, e.g. foo will produce $foo. The input syntax for lists is (followed by a sequence of tokens terminated by), e.g. (a b c) will read in as LIST[$A, $B, $C]. Commas are permitted (and ignored) between elements of a list.

GetRefAnyLine: PROC [stream: STREAM] RETURNS[LIST OF REF ANY];
calls GetRefAny, constructing a list of the values returned, until an object is immediately followed by a CR...
e.g. if user types FOO FIE FUM{cr} returns ($FOO, $FIE, $FUM). If user types FOO FIE FUM{sp}{cr} continues reading on next line. Useful for line oriented command interpreters. (Note that a CR typed inside of a list simply has the same effect as a space, i.e. terminates the previous identifier, but not the read operation).

Parsing the input stream as a sequence of arbitrary tokens: GetToken

GetSequence parses the input stream into a sequence of characters, e.g. the next line. The application then typically has to parse this sequence into smaller units, called tokens. GetCedarToken is one way to do this. However, occasionally the user may not need the full power of the cedar scanner, or may wish to employ some other algorithm. The charProc argument of GetToken described below provides for a limited form of parsing by dividing all characters into three classes: {**sepr**, **break**, **other**}.

A **sepr** character is a character that separates tokens. Sepr characters never appear in tokens. For example, for most applications, SP would be a sepr.

A **break** character is a character that is a token all by itself. For example, for some applications,] might be a break character.

other characters are everything else.

BreakProc: TYPE = PROC [char: CHAR] RETURNS [CharClass];
CharClass: TYPE = {break, sepr, other};

GetToken: PROC [stream: STREAM, breakProc: BreakProc ← TokenProc] RETURNS[ROPE];
skips over characters which breakProc says are seprs, *and then reads until it encounters either a* break *or* sepr. *If break is the first thing encountered, return a token consisting of the single* break *character, otherwise return the sequence of* other *characters read before the first* break *or* sepr.

Thus each time you call **GetToken**, you get back a rope consisting of either one or more **other** characters, or else a rope consisting of just a single character that is a **break** character.

For example, suppose your application involved parsing an input stream into a sequence of identifiers separated by commas, where you want to make sure that there is a comma between each identifier, but don't care whether there are any spaces before or after the comma. Here is what you might write:

YourProc: BreakProc = {RETURN[IF char = '. THEN break ELSE IF char = SP THEN sepr ELSE other]};

Successive calls on **GetToken** using this BreakProc would alternately return identifiers and ",".

Since it is a good example of the use of GetSequence, here is the definition of GetToken:

```
GetToken: PROC[stream: STREAM, breakProc: BreakProc] = {
  anySeen: BOOL ← FALSE;
  proc: CharProc = {
    SELECT breakProc[char] FROM
      break => {include ← NOT anySeen; quit ← TRUE};
      sepr => {include ← FALSE; quit ← anySeen};
      other => {include ← TRUE; quit ← FALSE; anySeen ← TRUE};
      ENDCASE => ERROR;
      };
  RETURN[GetSequence[stream, charProc]];
  };
```

The following procedures may be useful in conjunction with GetToken.

WhiteSpace: BreakProc;
Returns sepr *on SP, CR, LF, TAB,* other *for all others*

IDProc: BreakProc;
Returns sepr *for SP, CR, ESC, LF, TAB, comma, colon, and semicolon, and* other *for all others, i.e. s.GetToken[IdProc] is approximately the same as the old s.GetRope[].*

TokenProc: BreakProc;
Returns sepr *for SP, CR, ESC, LF, TAB, comma, colon, and semicolon,*
break *for [,], (,), {, }, '", '+, '-, '*, /, '@, '←,*
and other *for all others, i.e. s.GetToken[TokenProc] is approximately the same as s.GetCedarToken[] (but real numbers, quoted literals, quoted text, etc. are not handled the same).*

SkipOver: PROC [stream: STREAM, skipWhile: BreakProc ← WhiteSpace];
skips over all seprs, or until it reaches the end of the stream. If skipWhile = NIL, no characters are skipped.

Stream Property Lists

The stream's property list can be accessed and used by the client for storing and retrieving data associated with a particular key, the same as with an atom's property list. The procedures StoreData, LookupData, and RemoveData correspond to PutProp, GetProp, and RemProp.

StoreData: PROC [self: STREAM, key: ATOM, data: REF ANY];
adds an entry to property list, or replaces existing one.

AddData: PROC [self: STREAM, key: ATOM, data: REF ANY] ;
like StoreData, except adds it to front of property list, even if something already there, i.e. so can use property list as a push down list.

LookupData: PROC [self: STREAM, key: ATOM] RETURNS [REF ANY];
returns first occurrence of key

RemoveData: PROC [self: STREAM, key: ATOM];
removes first occurrence of key from property list

Defining new kinds of streams

A stream is defined by specifying a collection of procedures which implement the various operations that a stream must support, such as reading or printing a character, erasing a character, closing the stream, changing its echoing, etc. Not all of the procedures need to be specified. For example, all procedures relating to output need not be defined for an input stream. Similarly, some operations that are not defined can be carried out in terms of others that are, e.g. if the **GetBlock** procedure is not specified, a call to **GetBlock** will be automatically transformed into n calls to **GetChar**, etc. Finally, if the stream is layered on top of another stream, it will automatically inherit the latter streams definitions for those procedures that are not explicitly specified. For a more complete discussion of what it means to default a stream procdure, see the discussion below following the definition of **CreateProcsStream**.

The following is an example of defining a new kind of stream.

An example: defining a stream to the Message Window

The application is to define a stream which can be used for output to the Viewer Message Window. Since the MessageWindow interface allows one to display ropes, the idea is to dump the characters into a rope, and then display the rope in the message window when a Flush is executed. We can write characters into a rope using a Rope Output Stream, described above. Furthermore, if we make the Rope Output Stream be the backing stream for our Message Window Stream, we do not even have to define PutChar, but can simply allow for that operation to be executed by a call through to the Rope Output Stream's PutChar. All we have to do is to define what Flush and Reset mean:

```
MessageWindowStreamProcs: REF StreamProcs ← NIL; -- the block of procedures for the stream.
          Initialized the first time one of these streams is created.

CreateMessageWindowStream: PROCEDURE RETURNS [STREAM] = {
   IF MessageWindowStreamProcs = NIL THEN MessageWindowStreamProcs ←
            CreateRefStreamProcs[
      flush: MessageWindowStreamFlush,
      reset: MessageWindowStreamReset,
      name: "Message Window"
      ]; -- initialize the block of procedures
   RETURN[IO.CreateProcsStream[streamProcs: MessageWindowStreamProcs, backingStream:
            IO.ROS[], streamData: NIL]];
   }; -- of CreateMessageWindowStream

MessageWindowStreamFlush: PROC[self: STREAM] = {
   r: ROPE ← self.backingStream.GetOutputStreamRope[]; -- get the characters that have been
            printed.
   i: INT ← 0;
   self.backingStream.Reset[]; -- reset the rope output stream.
   WHILE (i ← Rope.Find[s1: r, s2: "\n", pos1: i]) # -1 DO
      r ← Rope.Replace[base: r, start: i, len: 1, with: " "]; -- change crs to spaces (since message
               window is only one line high!)
   ENDLOOP;
```

```
    MessageWindow.Append[message: r, clearFirst: TRUE]; -- display the rope
    };

MessageWindowStreamReset: PROC[self: STREAM] = {
    self.backingStream.Reset[]; -- reset the rope output stream.
    MessageWindow.Clear[]; -- clear the message window.
    };
```

CreateProcsStream: PROC[streamProcs: REF StreamProcs, streamData: REF ANY, backingStream: STREAM ← NIL] RETURNS[stream: STREAM];

the general mechanism for implementing streams.

streamProcs *is intended to be the result of a call to* CreateRefStreamProcs *(described below).* CreateRefStreamProcs *is responsible for inserting the appropriate default procedures for those operations not explicitly implemented by the stream. See discussion of default operations below.*

backingStream *is used when building a layered stream: those operations not defined in* **streamProcs** *will be automatically inherited from (passed down to) the backingStream.*

Note: if no backingStream is specified, the minimum required to implement a stream to be used for input is getChar and endOf. The minimum required to implement a stream to be used for output is putChar. Everything else can be defaulted...

In other words, if these operations are implemented, the only way of encountering the signal NotImplementedForThisStream is if the client explicitly calls GetIndex, SetIndex, GetLength, SetLength, GetunsafeBlock, or PutUnsafeblock when they are not implemented, or if the client attempts to obtain input from a stream intended to be used only for for output or vice versa. All other procedures will be handled via the defaults described below.

```
CreateRefStreamProcs: PROC[
    getChar: PROC[self: STREAM] RETURNS[CHAR] ← NIL,
    endOf: PROC[self: STREAM] RETURNS[BOOL] ← NIL,
    charsAvail: PROC[self: STREAM] RETURNS[BOOL] ← NIL,
    getBlock: PROC[self: STREAM, block: REF TEXT, startIndex: NAT, stopIndexPlusOne: NAT]
      RETURNS[nBytesRead: NAT] ← NIL,
    unsafeGetBlock: UNSAFE PROC[self: STREAM, block: UnsafeBlock] RETURNS[nBytesRead: INT] ←
      NIL,

    putChar: PROC[self: STREAM, char: CHAR] ← NIL,
    putBlock: PROC[self: STREAM, block: REF READONLY TEXT, startIndex: NAT, stopIndexPlusOne:
      NAT] ← NIL,
    unsafePutBlock: PROC[self: STREAM, block: UnsafeBlock] ← NIL,
    flush: PROC[self: STREAM] ← NIL,

    reset: PROC[self: STREAM] ← NIL,
    close: PROC[self: STREAM, abort: BOOL ← FALSE] ← NIL,
    getIndex: PROC[self: STREAM] RETURNS [INT] ← NIL,
    setIndex: PROC[self: STREAM, index: INT] ← NIL,

    getLength: PROC[self: STREAM] RETURNS [length: INT] ← NIL,
    setLength: PROC[self: STREAM, length: INT] ← NIL,
    backup: PROC[self: STREAM, char: CHAR] ← NIL,
    userAbort: PROC[self: STREAM] RETURNS[abort: BOOL] ← NIL,
    setUserAbort: PROC[self: STREAM]  ← NIL,
    resetUserAbort: PROC[self: STREAM]  ← NIL,
    setEcho: PROC[self: STREAM, echoTo: STREAM] RETURNS [oldEcho: STREAM] ← NIL,
    eraseChar: PROC[self: STREAM, char: CHAR] ← NIL,
currentPosition: PROC[self: STREAM] RETURNS[position: INT] ← NIL,
name: Rope.ROPE ← NIL -- used for printing the stream itself as a ref any.
]
RETURNS [REF StreamProcs];
```

creates a REF streamprocs, supplying appropriate defaults for the NIL procedures. Intended to be performed only once for each different stream class and the result saved, i.e. all instances of this stream can share the same streamProcs.

What it means to default a stream operation

The default procedure is to first check if there is a backingStream, and if so, call the corresponding operation in the backingStream. If no backingStream is specified, the default procedure is as follows:

CharsAvail: check whether endOf was specified, and if so return ~endOf[handle], otherwise raise signal NotImplementedForThisStream as described below

EndOf: check whether charsAvail was specified, and if so return ~charsAvail[handle], otherwise raise signal NotImplementedForThisStream as described below

GetBlock: implement via n Calls on GetChar

PutBlock: implement via n calls to PutChar

Flush, Reset: Nops, do not signal

Close: if abort = NIL, call Flush, otherwise, call Reset. do not signal

Backup: replace, i.e. ambush, the stream's original StreamProcs by a different set of procedures in which getChar, getBlock, endOf, charsAvail, and reset have different definitions when the stream is in the "backed up" state. The remaining procedures call through to the original procedures. When the character has been consumed, the streams original procedures are restored. Note that this causes the stream to run more slowly only when in the backed up state.

SetEcho: similar to Backup, i.e. ambushes the getChar and getBlock procedures, causing them to also output to the indicated handle. This causes all procedure calls in the stream to run more slowly due to the extra level of procedure call and lookup;

CurrentPosition: ambushes the streams putChar and putBlock procedures, and counts the characters as they go by. Since this does not occur until the first call to CurrentPosition, characters that were output before this call will not be counted, i.e. the first call to CurrentPosition will always return 0, and subsequent calls until the first CR is seen will return the number of characters output since the first call;

EraseChar: output \ followed by the character. Defaulting EraseChar does not affect the performance of the stream on other operations.

For all other cases, the default procedure is to raise SIGNAL Signal[NotImplementedForThisStream] (i.e. client can resume) , if the corresponding operation does NOT return a value (i.e. is executed for effect only, such as putChar), or raise ERROR Error[NotImplementedForThisStream] if the corresponding operation requires a return value, such as getChar.

In general, the decision about whether or not to implement a particular operation for a stream, must be made on the basis of performance and frequency of use versus convenience For example, if the stream implements backup itself, it can include the necessary code in its own getChar, getBlock, reset, etc., so that these procedures will continue to run fast, even when the streeam is backed up. However, it is more work for the implementor. For example, viewer streams implement SetEcho, but InputRopeStream does not. Similarly, an edited stream implements backup but viewer stream does not, on the grounds that most calls to a viewer stream that involve parsing and backing up will instead be going to an an edited stream.

AmbushProcsStream: PRIVATE PROCEDURE[self: STREAM, streamProcs: REF StreamProcs, streamData: REF ANY, reusing: STREAM ← NIL] ;

> *used to build a layered stream on top of another stream, for those cases where the desired effect is to have any program that has hold of* self *to see the new behaviour. Implemented by smashing* self *to point to the new procs and data, with a backing stream that contains the same data, procs, etc. as* self *originally did. if reusing is non-NIL, it is reused and no allocations are performed. Typically, reusing is saved on the property list of a stream which is going to be ambushed and then unambushed repeatedly.*

UnAmbushProcsStream: PRIVATE PROCEDURE[self: STREAM];

> *inverse of AmbushProcsStream, i.e. elevates the backing stream to self. Nop if no backing stream.*

Defining new stream operations

CreateRefStreamProcs takes as arguments 22 predefined operations on a stream. Suppose we want to define new operations which might not appear in this interface which are to be (optionally) implemented by a various kinds of streams. The procedure **Implements** allows us to do this. It is used to provide a generic implementation for AppendStreams for all streams that do not implement this operation (currently no stream does), to implement ChangeDeliverWhen for both edited streams and viewer edited streams (see ViewerIO), and also the mechanism used to supply the default behaviour for Backup and CurrentPosition described above. For examples of its use, see IOImpl.

Implements: PRIVATE PROC [self: STREAM, operation, via: PROC ANY RETURNS ANY];

> *used to supply the implementation of an operation other than those specified as arguments to CreateRefStreamProcs.* 'operation' *is the generic procedure, and Via is the implementing procedure for this stream class. Via is checked (using the RT interfaces) to make sure it is of the correct type. (The names of the arguments as well as the return values must agree.) If it is not, the signal* IO.Error[ProcedureHasWrongType] *is raised. If the type is correct, Via is saved on the property list of the streamProcs, so that the type checking need be done only once for each stream class. Subsequent calls to Implements for the same stream class are essentially nops.*
>
> *When the* 'operation' *is invoked,* 'via' *will be called with the same arguments as the generic procedure was called with.*
>
> > *For example, if* CreateRefStreamProcs *did not take* setLength *as an argument, this operation could be implemented for a particular stream class by performing:* stream.Implements[SetLength, MySetLength], *where* MySetLength *is of type* PROC[self: STREAM, length: INT].

UncheckedImplements: -- Even more -- PRIVATE PROC [self: IO.STREAM, operation, via: PROC ANY RETURNS ANY, data: REF ANY ← NIL, procRef: REF ANY, key: ATOM];

> *used by implements. Calling directly and specifying procRef and key enables avoiding use of RT interfaces*

LookupProc: PRIVATE PROC [self: STREAM, operation: PROC ANY RETURNS ANY] RETURNS[proc: REF ANY];

PF (PrintFormatted) Package

Concept

PutF (and PutFR and PutFList) produce output according to a collection of Values and a format control string which can including embedded format codes. The Values that are the arguments to PutF are associated with the format codes in the order that they appear in the format string. For example:

h.PutF["This is %g in a 5 position field: |%5d|", rope["an integer"], int[17]];

would produce the result:

"This is an integer in a 5 position field: | 17|".

If a format code is encountered which is not recognized, the signal IO.Signal[UnknownFormat] is raised. If there are not enough format codes to match the values supplied as arguments, or not enough values then ???

PreDefined Format Codes

The Value refAny is accepted by all formats; if dereferencing yields a value of an acceptable type, it is printed directly. Otherwise, if %g or %h are used, IO.PrintRefAny is called. If some other code was used, the Signal IO.Signal[UnprintableValue] or IO.Signal[TypeMismatch] is raised. If this signal is RESUMEd, some '# characters are printed.

In the following descriptions, "number" means a Value that is either card, char, int, real, or time. CHARACTERS passed to a number routine print as their Ascii code.

"string" means a Value that is either a string, rope, text, bool, or a number. BOOLEANs passed to a "string" routine print as TRUE or FALSE and CHARACTERs print as characters.

%**e**: accepts numbers. Like Fortran E format, prints in "scientific notation" (mantissa and exponent).

%**f**: accepts numbers. Like Fortran F format, prints in "fixed point" format.

%**g**: accepts all values. Prints whatever you give it sensibly. When in doubt, use %g.

%**h**: like %g, except that control characters (code < 40B) print as "↑<char + '@>.

%**l**: accepts ROPEs. Changes looks for ViewerStreams, is a NOP for all other streams so that the client does not have to know whether he is printing to a viewer or a file or whatever. The value is interpreted as a sequence of looks, e.g. h.PutF["Make this %lbold%l for emphasis", rope["b"], rope["B]] will print "Make this **bold** for emphasis". (Note that B means remove the bold look because it is shift-b.) The character space has the same interpretation as for tioga, i.e. removes all looks.

%**r**: accepts numbers, interprets as time interval in seconds, and prints as HH:MM:SS

%**t**: accepts numbers and Time[], prints in usual date format with time zone (e.g. " PDT")

The following formats apply to all numbers (and to BOOLEANs with the conversion TRUE = 1 and FALSE = 0). (To print a POINTER, LONG POINTER, REF, etc. value as a number, PUN it into a number of the correct length before converting to a Value.)

%**b**: print in octal.

%**d**: print in decimal.

%**x**: print in hex.

Field Width Specifications

Information about printing a particular value can be inserted between the "%" and the code letter. **Is this true for all control codes???** Field width, left or right justification, and zero filling can be controlled. The default is to print the value in its entirety, with no spacing on either side. If a number is present between the "%" and code (e.g. "%6d"), it is interpreted as a minimum field width, useful for printing columns, etc.

Extra spaces are inserted before the number (right adjusted) to fill out to the indicated number of spaces.

unless a "-" precedes the number (e.g. "%-6d") to indicated filling with spaces on the right (left-adjusted). If "-" is not present, a "0" may be inserted (e.g. "%06d") to indicate right-adjustment and filling with zeroes rather than spaces. In any case the field width is a minimum, not a maximum. Should the size of the output exceed the field width, it is printed in its entirety.

A field width may have the form xx.yy, viz: "%12.4g". These specifications are interpreted by the floating point number printing routines. Other types of conversions ignore the "fraction".

Defining new format codes

Users may specify their own procedures to interpret codes following the "%" via **SetPFCodeProc**. For example, h.SetPFCodeProc['z, ZProc] will cause subsequent references to "%z" to call ZProc. Predefined codes (e.g. d, e, f) may be redefined. Only alphabetic (['a..'z]) code characters are legal, case does no matter.

```
SetPFCodeProc: PROC [stream: STREAM, char: CHAR, codeProc: PFCodeProc];
    when %char is encountered, codeProc is called with format = ???, and val = the Value that
     corresponded to this control code.
PFCodeProc: TYPE = PROC [stream: STREAM, val: Value, format: Format, char: CHAR];
Format: TYPE = RECORD [form: ROPE, first: INT];
```

Another way of defining new commands is via SetPFStarCodeProc. PFStarCodeProcs differ from PFCodeProcs in that they do not take arguments, i.e. none of the Values that comprise the arguments to the output procedure are associated with this PFStarCode. PFStarCodes are indicated in the format string by preceding the code with a * instead of a %, and then following it immediately with the code letter (non-alphabetic characters following * print as themselves, e.g. "**" prints as *),

Undefined StarCodes also raise IO.Signal[NotImplementedForThisStream].

```
PFStarCodeProc: TYPE = PROC [stream: STREAM, char: CHAR];
SetPFStarCodeProc: PROC [stream: STREAM, char: CHAR, codeProc: PFStarCodeProc];
```

PrintProcs

PrintProcs allow the client to specify an alternative way of printing an object of a specified type. Usually this is employed to summarize the object in some useful way. For example, a viewer is an object of type REF ViewerClasses.ViewerRec where ViewerRec is a RECORD consisting of nineteen fields! Printing this object without a PrintProc would thus produce copious amounts of information, most of which is probably not of interest to the user who simply wants to see which viewer it is. Instead, we can define a printproc which prints a viewer showing just its class and name as follows:

```
ViewerPrintProc: IO.RefPrintProc = {
    h: REF READONLY ViewerClasses.ViewerRec ← NARROW[ref];
    stream.PutF["{Viewer - class: %g, name: %g}", atom[h.class.flavor], rope[IF
     Rope.Length[h.name] # 0 THEN h.name ELSE "(no name)"]];
    };
```

We register this printProc via:

```
AttachRefPrintProc[refType: CODE[ViewerClasses.Viewer], refPrintProc: ViewerPrintProc];
```

Following this, the viewer for the initial UserExec would print as:

```
{Viewer - class: Typescript, name: Work Area A: Executive}
```

AttachRefPrintProc: PROC [refType: RTBasic.Type, refPrintProc: RefPrintProc];

Used to register a procedure to be called for printing an object of the specified ref type. The procedure will be called with the corresponding ref and a stream. If the class of refType is not ref or list, raises IO.Signal[NotARefType].

RefPrintProc: TYPE = PROC [ref: REF READONLY ANY, stream: STREAM, depth: INT ← 4, width: INT ← 32, verbose: BOOL ← FALSE];

Occasionally it is useful to define a printproc for a non-ref TYPE. In this case, the client can define a PrintProc which is a given a tv describing the corresponding object. Here is an example of a PrintProc which will print an object of type System.GreenwichMeanTime as a time, rather than a LONG CARDINAL (which is the type of System.GreenwichMeanTime).

```
TimePrintProc: TVPrintProc = {
   t: System.GreenwichMeanTime;
   t ← LOOPHOLE[AMBridge.TVToLC[tv]];  -- get the "time" out of the tv.
   stream.Put[time[t]];
   };
AttachTVPrintProc[type: CODE[System.GreenwichMeanTime], tvPrintProc: TimePrintProc,
 canHandleRemote: FALSE];
```

AttachTVPrintProc: PROC [type: RTBasic.Type, tvPrintProc: TVPrintProc, canHandleRemote: BOOL ← FALSE];

More general print proc mechanism for use with any type. The procedure will be called with a tv for the corresponding object and a stream. IF canHandleRemote is TRUE, the printproc will also be invoked on remote tvs, otherwise not.

TVPrintProc: TYPE = PROC [tv: RTBasic.TV, stream: STREAM, depth: INT ← 4, width: INT ← 32, verbose: BOOL ← FALSE];

Signals and Errors

UserAborted: ERROR [abortee: REF ANY ← NIL, msg: ROPE ← NIL];

the standard error to be raised by an application to indicate that the user has aborted. Replaces UserExec.UserAborted.

EndOfStream: ERROR [stream: STREAM];

attempt to do a getChar at end of stream, or attempt to do a setindex beyond end of stream. Separate error for convenience.

Error: ERROR [ec: ErrorCode, stream: STREAM];
ErrorCode: TYPE = {

NotImplementedForThisStream,

a signal, if the corresponding operation does not return a value, e.g. SetEcho, SetLength, etc., and not resumable, i.e. an ERROR, if the operation does return a value, e.g. GetLength,

IllegalBackup,

attempt to do a second backup without having done an intermediate getChar, or attempt to backup a character other than the one just read.

SyntaxError,

raised by GetBool, GetInt, GetCard, GetReal, GetRefany.

```
    StreamClosed,
        raised by file stream operations after Close (except Flush, Reset, Close.)

    FileTooLong,
        may be raised by SetLength

    BadIndex
        negative length or index to SetLength/SetIndex, or
        negative or too-large index to UnsafeGetBlock/UnsafePutBlock
    };

Signal: SIGNAL [ec: SignalCode, stream: STREAM];
SignalCode: TYPE = {
    NotImplementedForThisStream,
        a signal, if the corresponding operation does not return a value, e.g. SetEcho, SetLength, etc.,
          and not resumable, i.e. an ERROR, if the operation does return a value, e.g.  GetLength,

    ProcedureHasWrongType,
        Raised by Implements when via is not of the type of operation. If resumed, effect is simply not
          to implement the operation,

    NotARefType,
        Raised by AttachRefPrintProc when refType is not of class ref or list. If resumed, effect is
          simply not to implement the print proc.

    EmptyBuffer.
        raised by EditedStream when attempt to do an eraseChar and nothing is there.

    UnmatchedLeftParen, UnmatchedStringDelim,
        raised by GetRefAny if endofstream reached while reading a list or string. If resumed, the effect
          is to supply the missing character, e.g. doing GetRefAny on the rope "(A B C" will return the
          list (A B C) if the client resumes the signal UnmatchedLeftParen.
    Rubout,
        raised by EditedStream when a DEL is seen.

    UnprintableValue, TypeMismatch, UnknownFormat
        raised by PF.
    };

BufferOverFlow: SIGNAL[text: REF TEXT] RETURNS[REF TEXT];
    used by text streams, should return a bigger one, with characters copied over, or else can return
      same text with length reset after having done something with chartacters.
```

Private and semi-private Types

```
ValueType: TYPE = {
    null, atom, boolean, character, cardinal, integer, real, refAny, rope,
    string, text, time, tv, type};

Value: TYPE = RECORD [SELECT type: ValueType FROM
    null => NULL,
    atom => [value: ATOM],
    boolean => [value: BOOL],
    character => [value: CHAR],
    cardinal => [value: LONG CARDINAL],
    integer => [value: INT],
    real => [value: REAL],
    refAny => [value: REF READONLY ANY],
    rope => [value: ROPE],
    string => [value: LONG STRING],
    text => [value: REF READONLY TEXT],
    time => [value: GreenwichMeanTime],
    tv => [value: RTBasic.TV],
    type => [value: RTBasic.Type],
    ENDCASE];

UnsafeBlock: TYPE = RECORD [
    base: LONG POINTER, startIndex, stopIndexPlusOne: INT --bytes--];
       used in conjunction with UnsafeGetBlock, UnsafePutBlock described above.
```

Note: slots in the StreamProcs record are allocated to those procedures that are either so basic to the operation of a stream that nearly every stream will implement them, or else for which performance is an issue, such as getBlock, getIndex, etc. Other generic operations such as Backup, PeekChar, CheckForAbort, SetEcho, EraseChar, Position, are implemented by storing the corresponding procedure on the streams property list, and looking it up (each time) the generic operation is called. If the stream (or its backing stream) does not supply an implementation, a default procedure is supplied which will accomplish the operation. {16}

```
StreamProcs: TYPE = PRIVATE RECORD[
    getChar: PROC[self: STREAM] RETURNS[CHAR] ← NIL,
    endOf: PROC[self: STREAM] RETURNS[BOOL] ← NIL,
    charsAvail: PROC[self: STREAM] RETURNS[BOOL] ← NIL,
    getBlock: PROC[self: STREAM, block: REF TEXT, startIndex: NAT, stopIndexPlusOne: NAT]
      RETURNS[nBytesRead: NAT] ← NIL,
    unsafeGetBlock: UNSAFE PROC[self: STREAM, block: UnsafeBlock] RETURNS[nBytesRead: INT] ←
      NIL,

    putChar: PROC[self: STREAM, char: CHAR] ← NIL,
    putBlock: PROC[self: STREAM, block: REF READONLY TEXT, startIndex: NAT, stopIndexPlusOne:
      NAT] ← NIL,
    unsafePutBlock: PROC[self: STREAM, block: UnsafeBlock] ← NIL,
    flush: PROC[self: STREAM] ← NIL,

    reset: PROC[self: STREAM] ← NIL,
    close: PROC[self: STREAM, abort: BOOL ← FALSE] ← NIL,
    getIndex; PROC[self: STREAM] RETURNS [INT] ← NIL,
    setIndex: PROC[self: STREAM, index: INT] ← NIL,
    otherStreamProcs: LIST OF StreamProperty ← NIL, -- used for specifying the implementation of
      optional, generic procedures. See Implements below.
    name: Rope.ROPE
    ];

StreamProperty: TYPE = REF StreamPropertyRecord;

StreamPropertyRecord: TYPE = PRIVATE RECORD[operation, via: PROC ANY RETURNS ANY, proc:
  REF ANY, key: ATOM];
```

Character constants. Included for convenience.

```
CR: CHAR = Ascii.CR;
SP: CHAR = Ascii.SP;
TAB: CHAR = Ascii.TAB;
LF: CHAR = Ascii.LF;
BS: CHAR = Ascii.BS;
ControlA: CHAR = Ascii.ControlA;
ControlX: CHAR = Ascii.ControlX;
FF: CHAR = Ascii.FF;
NUL: CHAR = Ascii.NUL;
ESC: CHAR = Ascii.ESC;
DEL: CHAR = Ascii.DEL;
BEL: CHAR = Ascii.BEL;
```

Miscellaneous

BackSlashChar: PROC [char: CHAR, stream: STREAM ← NIL] RETURNS [CHAR];

interpreters \ conventions, e.g. maps \n to CR, \t to TAB, etc. Raises SyntaxError if \ not followed by acceptable character. IF char is in ['0..9], then stream must be supplied, or else syntaxerror. If stream is supplied, two more characters are read. If these are digits, returns corresponding character, otherwise, raises SyntaxError.

Zone: PRIVATE ZONE; -- *prefixed zone used for storing stream data, stream procs, etc.*

END.

Converting from 3.5

The following are some suggestions about how to convert a program using IO.GetRope or IO.GetToken to the new scheme of things. These suggestions should cover about 95% of the cases.

Let us consider GetToken first. A old-style call of the form s.GetToken[], which defaults the second argument, can stay the same. But you might want to consider using GetCedarToken instead, especially if what you are doing is reading mesa: GetCedarToken will do more Cedar-like things for real numbers, quoted literals, quoted text, and the like.

GetRope is only a little more complicated. An old-style call of the form s.GetRope[], which defaults the second and third arguments, can be replaced by the new call s.GetToken[IDProc]. Note that the new call s.GetToken[] would also probably do what you want, the only difference being in the treatment of break characters: for example, the old s.GetRope[] would read @foo as a single token, while the new s.GetToken[] would read it as two separate tokens. Similarly, /indigo/cedar/top/io.df would be read as 8 tokens by the new s.GetToken[], but as only one token by the old s.GetRope[] and the new s.GetToken[IDProc]. All three of these have the same effect if your desired tokens are always separated by white space.

Beware! If you use GetRope, you cannot depend on compile error messages alone to tell you where changes have to be made. The new GetRope means something quite different from the old GetRope, but any call to the old GetRope that defaults the second and third arguments will still look OK to the compiler as a call on the new GetRope. Clients of the old GetRope were just parsing some random input into things separated by white space. A client of the new GetRope should be in the process of parsing the input stream as a sequence of mesa tokens. Such a client calls the new GetRope when she expects a rope literal to be the next thing in the input stream, and the call to GetRope either returns the corresponding rope or raises an error. Thus, all calls to the old GetRope must be changed to GetToken as described in the previous paragraph, and you must search your program to find them all.

If you have defined your own IO.BreakProcs for use with GetToken, the following recipe should work: change all of the StopAndPutBackChar and StopAndTossChar to sepr, StopAndIncludeChar to break, and KeepGoing to other. If you have defined your own IO.BreakProcs for use with GetRope, you should change KeepGoing to other and everything else to sepr, and then use GetToken in place of GetRope. However, you might want to rewrite in terms of GetSequence instead.

```
-- Rope.mesa, "Thick" string definitions
--   Russ Atkinson, August 27, 1982 11:51 am

-- Table of Contents:
-- *** Part 1: Basic operations and definitions
-- *** Part 2: Extended operations and definitions
-- *** Part 3: Miscellaneous definitions

-- A Rope is (nominally) an immutable object containg a sequence of characters indexed starting at 0 for
Size characters. The representation allows certain operations to be performed without copying all of the
characters at the expense of adding additional nodes to the object.

-- NOTE: the bit pattern of the text variant is GUARANTEED to be consistent with the built-in Cedar
type TEXT, although the types will not agree either at compile-time or runtime. This means that one can
use objects of type Text or TEXT interchangeably provided that the runtime type is not examined, and
provided that the compiler can be fooled (via LOOPHOLE). Since REF TEXT is roughly compatible with
Mesa LONG STRING, this is a means for passing Rope refs to Pilot routines that expect LONG
STRING.  For short STRING you are on your own (see ConvertUnsafe).

DIRECTORY
  Environment USING [Comparison];

Rope: CEDAR DEFINITIONS

  -- *** Part 1:  Basic operations and definitions *** --

  = BEGIN

  ROPE: TYPE = REF RopeRep;

  NoRope: ERROR;
  -- signalled if rope is invalid variant
  -- usually indicates severe illness in the world

  -- Note: BoundsFault = Runtime.BoundsFault
  -- It is raised by many of the Rope operations

  Cat: PROC [r1, r2, r3, r4, r5, r6: ROPE ← NIL] RETURNS [ROPE];
    -- returns the concatenation of up to six ropes (limit based on eval stack depth)
    -- BoundsFault occurs if the result gets too large

  Concat: PROC [base,rest: ROPE ← NIL] RETURNS [ROPE];
    -- the two-rope (faster) version of Cat
    -- BoundsFault occurs if the result gets too large

  Compare: PROC
      [s1, s2: ROPE, case: BOOL ← TRUE] RETURNS [Environment.Comparison];
    -- based on CHAR collating sequence
    -- case => case of characters is significant

  Equal: PROC [s1, s2: ROPE, case: BOOL ← TRUE] RETURNS [BOOL];
    -- contents equality of s1 and s2
    -- case => case of characters is significant
```

```
Fetch: PROC [base: ROPE, index: INT ← 0] RETURNS [c: CHAR];
  -- fetches indexed character from given ropes
  -- BoundsFault occurs if index is >= the rope size

Find: PROC [s1, s2: ROPE, pos1: INT ← 0, case: BOOL ← TRUE] RETURNS [INT];
  -- like Index, returns position in s1 where s2 occurs (starts looking at pos1)
  -- returns -1 if not found
  -- case => case of characters is significant

Index: PROC
    [s1: ROPE, pos1: INT ← 0, s2: ROPE, case: BOOL ← TRUE] RETURNS [INT];
  -- Returns the smallest character position N such that
  -- s2 occurs in s1 at N and N >= pos1.  If s2 does not
  -- occur in s1 at or after pos1, s1.length is returned.
  -- case => case of characters is significant

IsEmpty: PROC [r: ROPE] RETURNS [BOOL];
  -- returns Length[r] = 0

Length: PROC [base: ROPE] RETURNS [INT];
  -- returns the length of the rope (Length[NIL] = 0)

Replace: PROC
    [base: ROPE, start: INT ← 0, len: INT ← MaxLen, with: ROPE ← NIL]
    RETURNS [ROPE];
  -- returns rope with given range replaced by new
  -- BoundsFault occurs if range invalid or result too long

Size: PROC [base: ROPE] RETURNS [INT];
  --  Size[base] = Length[base]

Substr: PROC [base: ROPE, start: INT ← 0, len: INT ← MaxLen] RETURNS [ROPE];
  -- returns a subrope of the base
  -- BoundsFault occurs if the range given is not valid

-- character conversions (RRA sez: why are they here?)

Control: PROC [ch: CHAR] RETURNS [CHAR] = INLINE {
  RETURN [IF ch IN ['A..'Z] THEN ch - controlOffset ELSE ch]
  };

Upper: PROC [ch: CHAR] RETURNS [CHAR] = INLINE {
  RETURN [IF ch IN ['a..'z] THEN ch - caseOffset ELSE ch]
  };

Lower: PROC [ch: CHAR] RETURNS [CHAR] = INLINE {
  RETURN [IF ch IN ['A..'Z] THEN ch + caseOffset ELSE ch]
  };

Letter: PROC [ch: CHAR] RETURNS [BOOL] = INLINE {
  RETURN [ch IN ['A..'Z] OR ch IN ['a..'z]]
  };

Digit: PROC [ch: CHAR] RETURNS [BOOL] = INLINE {
  RETURN [ch IN ['0..'9]]
  };

  -- *** Part 2: Extended operations and definitions *** --

Run: PROC
    [s1: ROPE, pos1: INT ← 0, s2: ROPE, pos2: INT ← 0, case: BOOL ← TRUE]
    RETURNS [INT];
  -- Returns largest number of chars N such that s1 starting at pos1
  -- is equal to s2 starting at pos2 for N chars. More formally:
  --   FOR i IN [0..N): s1[pos1 + i] = s2[pos2 + i]
  -- If case is true, then case matters.

Match: PROC [pattern, object: ROPE, case: BOOL ← TRUE] RETURNS [BOOL];
  -- Returns TRUE iff object matches the pattern, where the pattern may contain
  -- * to indicate that 0 or more characters will match.
  -- If case is true, then case matters.

SkipTo: PROC [s: ROPE, pos: INT ← 0, skip: ROPE] RETURNS [INT];
  -- Return the lowest position N in s such that s[N] is in the skip rope
  -- and N >= pos. If no such character occurs in s, then return s.length.

SkipOver: PROC [s: ROPE, pos: INT ← 0, skip: ROPE] RETURNS [INT];
  -- Return the lowest position N in s such that s[N] is NOT in the skip rope
  -- and N >= pos. If no such character occurs in s, then return s.length.

Map: PROC
    [base: ROPE, start: INT ← 0, len: INT ← MaxLen, action: ActionType]
    RETURNS [BOOL];
  -- Applies the action to the given range of characters in the rope
  -- Returns TRUE when some action returns TRUE
  -- BoundsFault occurs on attempting to fetch a character not in the rope

Translate: PROC
    [base: ROPE, start: INT ← 0, len: INT ← MaxLen, translator: TranslatorType ← NIL]
    RETURNS [new: ROPE];
  -- applies the translation to get a new rope
  -- if the resulting size > 0, then new does not share with the original rope!
  -- if translator = NIL, the identity translation is performed

Flatten: PROC [base: ROPE, start: INT ← 0, len: INT ← MaxLen] RETURNS [Text];
  -- Returns a flat rope from the given range of characters
  -- BoundsFault occurs if the resulting length would be > LAST[NAT]

FromProc: PROC
   [len: INT, p: PROC RETURNS [CHAR], maxPiece: INT ← MaxLen] RETURNS [ROPE];
  -- returns a new rope given a proc to apply for each CHAR

FromRefText: PROC [s: REF READONLY TEXT] RETURNS [Text];
  -- makes a rope from a REF READONLY TEXT
  -- causes copying

ToRefText: PROC [base: ROPE] RETURNS [REF TEXT];
  -- makes a REF TEXT from a rope
  -- causes copying

FromChar: PROC [c: CHAR] RETURNS [Text];
  -- makes a rope from a character

MakeRope: PROC
    [base: REF, size: INT, fetch: FetchType, map: MapType ← NIL,
     pieceMap: PieceMapType ← NIL] RETURNS [ROPE];
  -- Returns a rope using user-supplied procedures and data
  -- the user procedures MUST survive as long as the rope does!

PieceMap: PROC
    [base: ROPE, start: INT ← 0, len: INT ← MaxLen, action: PieceActionType,
     mapUser: BOOL ← TRUE] RETURNS [BOOL];
  -- Applies the action to the given subrope in pieces (max of 1 piece/Substr,
  -- 2 pieces/Concat, 3 pieces/Replace, either 1 piece/MakeRope or use user's
  -- routine). Returns TRUE when some action returns TRUE.
  -- BoundsFault occurs on attempting to fetch a character not in the rope

ContainingPiece: PROC
    [ref: ROPE, index: INT ← 0] RETURNS [base: ROPE, start: INT, len: INT];
  -- Find the largest piece containg the given index such that the result is
  -- either a text or an object variant.
  -- (NIL, 0, 0) is returned if the index is NOT in the given rope

Balance: PROC
    [base: ROPE, start: INT ← 0, len: INT ← MaxLen, flat: INT ← FlatMax]
    RETURNS [ROPE];
  -- Returns a balanced rope, possibly with much copying of components
  -- flat ← MIN[MAX[flat,FlatMax], LAST[NAT]]
  -- len ← MIN[MAX[len,0], Size[base]-start]
  -- start < 0 OR start > Size[base] => bounds fault
  -- the resulting maxDepth will be limited by 2+log2[len/flat]

VerifyStructure: PROC [s: ROPE] RETURNS [leaves, nodes, maxDepth: INT];
  -- traverse the structure of the given rope; return the number of leaves,
  -- nodes and the max depth of the rope extra checking is performed to verify
  -- invariants a leaf is a text or object variant a node is a non-NIL,
  -- non-leaf variant shared leaves and nodes are multiply counted

VerifyFailed: ERROR;
   -- occurs when VerifyStructure finds a bad egg
   -- should not happen, of course

  -- *** Part 3: Miscellaneous definitions *** --

controlOffset: NAT = 100B;
caseOffset: NAT = 'a - 'A;

FetchType: TYPE = PROC [data: REF, index: INT] RETURNS [CHAR];
  -- type of fetch routine used to make a user rope

MapType: TYPE =
   PROC [base: REF, start, len: INT, action: ActionType] RETURNS [quit: BOOL ← FALSE];
  -- type of user routine used to map over a subrope
  -- returns TRUE if some action returns TRUE

ActionType: TYPE = PROC [c: CHAR] RETURNS [quit: BOOL ← FALSE];
  -- type of routine applied when mapping
  -- returns TRUE to quit from Map

TranslatorType: TYPE = PROC [old: CHAR] RETURNS [new: CHAR];
  -- type of routine supplied to Translate

PieceMapType: TYPE =
   PROC [base: REF, start, len: INT, action: PieceActionType]
     RETURNS [quit: BOOL ← FALSE];
  -- type of routine used to piecewise map over a subrope
  -- returns TRUE if some action returns TRUE

PieceActionType: TYPE =
   PROC [base: ROPE, start: INT, len: INT] RETURNS [quit: BOOL ← FALSE];
  -- type of routine applied when mapping pieces
  -- returns TRUE to quit from PieceMap

RopeRep: PRIVATE TYPE =
  RECORD
   [SELECT tag: * FROM
     text => [length: NAT, text: PACKED SEQUENCE max: CARDINAL OF CHAR],
     node => [SELECT case: * FROM
              substr => [size: INT, base: ROPE, start: INT, depth: INTEGER],
              concat => [size: INT, base, rest: ROPE, pos: INT,
                         depth: INTEGER],
              replace => [size: INT, base, replace: ROPE,
                          start, oldPos, newPos: INT, depth: INTEGER],
              object => [size: INT, base: REF, fetch: FetchType,
                         map: MapType, pieceMap: PieceMapType]
              ENDCASE]
     ENDCASE];

MaxLen: INT = LAST[INT];
FlatMax: CARDINAL = 24;
Text: TYPE = REF TextRep; -- the small, flat variant handle
TextRep: TYPE = RopeRep[text]; -- useful for creating new text variants

END.
```

For those who care, this is the official explanation of the RopeRep variants:

Note: NIL is allowed as a valid ROPE.

Note: ALL integer components of the representation must be non-negative.

```
SELECT x: x FROM
  text => {-- [0..x.length) is the range of char indexes
          -- [0..x.max) is the number of chars of storage reserved
     -- all Rope operations creating new text objects init x.length = x.max
```

```
    -- x.length <= x.max is required
    -- the bit pattern of the text IS IDENTICAL to TEXT and StringBody!!!!
    };
node =>
 {SELECT x:x FROM
   substr =>
 {-- [0..x.size) is the range of char indexes
  -- x.base contains chars indexed by [0..x.size)
  --   [0..x.size) in x ==> [x.start..x.start+x.size) in x.base
  -- Size[x.base] >= x.start + x.base
  };
concat =>
 {-- [0..x.size) is the range of char indexes
  -- x.base contains chars indexed by [0..x.pos)
  --   [0..x.pos) in x ==> [0..x.pos) in x.base
  -- x.rest contains the chars indexed by [x.pos..x.size)
  --   [x.pos..x.size) in x ==> [0..x.size-x.pos) in x.base
  -- x.pos = Size[x.base] AND x.size = x.pos + Size[x.rest]
  };
replace =>
 {-- [0..x.size) is the range of char indexes
  -- x.base contains chars indexed by [0..x.start), [x.newPos..x.size)
  --   [0..x.start) in x ==> [0..x.start) in x.base
  --   [x.newPos..x.size) in x ==> [x.oldPos..Size[x.base]) in x.base
  -- x.rest contains the chars indexed by [x.start..x.newPos)
  --   [x.start..x.newPos) in x => [0..x.newPos-x.start) in x.base
  -- x.size >= x.newPos >= x.start AND x.oldPos >= x.start
  -- x.size - x.newPos = Size[x.base] - x.oldPos
  };
object =>
 {-- [0..x.size) is the range of char indexes
  -- x.base if the data needed by the user-supplied operations
  -- x.fetch[x.base, i] should fetch the ith char AND x.fetch # NIL
  -- x.map[x.base, st, len, action]
  --   implements Map[x, st, len, action]
  -- x.pieceMap[x.base, st, len, action] should behave
  --   implements PieceMap[x, st, len, action, TRUE]
  -- it is OK to have x.map = NIL OR x.pieceMap = NIL
  };
ENDCASE => ERROR NoRope}
ENDCASE => ERROR NoRope};
```

Spelling Corrector/Pattern Completer Edited by Teitelman on November 3, 1982 11:23 am

```
DIRECTORY
   Generator USING [Handle],
   Rope USING [ROPE]
   ;

Spell: CEDAR DEFINITIONS =

BEGIN
```

General comments

This interface supports two distinct operations, spelling correction and pattern completion. Spelling correction involves attempting to transform an unrecognized rope, called the unknown, into a known rope, which is a member of some computable set of ropes, according to various heuristics, e.g. removal of doubled characters, transpositions, case errors, etc. For example, correct Compille to Compile (doubled character), UsrExecImpl to UserExecImpl (missing character), ViewerOsp to ViewerOps (transposition). Pattern completion involves transforming the unknown rope into a known candidate or candidates using specified transformations. (Currently the only pattern supported is *, which matches any sequence of characters. In the future, all patterns recognized by the Tioga Edit Tool will be supported.)

There are two ways to call this package. In the first case, the client has in hand an unknown rope and wants to find the single correct rope that it specifies. The unknown rope may be a misspelling of some candidate in the source of candidates (see discussion of sources below), or it may be a pattern which uniquely specifies the desired candidate, e.g. the user might type R*.profile in a context where a file name was required, knowing that this is sufficient to be unambiguous. In order to permit the client not to have to distinguish these two cases, a single procedure, GetTheOne, is provided. In the case that the unknown is a pattern, GetTheOne calls the pattern matcher, and if the resulting list consists of a single candidate, returns that candidate, otherwise GetTheOne returns NIL (i.e. if there are zero or more than one successful matches). If the unknown is not a pattern, then actual spelling correction will be performed. In either case, if the user's profile has so indicated as described below, the user will be informed of the action, or confirmation will be requested before the action is taken.

The second use of this package occurs when the client has in hand an unknown and wants all candidates that match it, i.e., a list of ropes. Again, the unknown may be a pattern, or simply a misspelling. For example, the user might type to the userexec Delete *.mesa$, or Delete Fooo.mesa. In order to permit the client not to have to distinguish these two cases, there is a single procedure, GetMatchingList, which handles both cases. If the unknown is a pattern, GetMatchingList returns a list of all candidates that match the pattern, if a misspelling, GetMatchingList returns a list consisting of the single correct spelling (if one is found, otherwise return NIL).

sources of candidates for correction/completion

For both GetTheOne and GetMatchingList, the source of candidates can either be a LIST OF ROPE, or a generator which produces ROPEs. Generators can be easily created using CreateGenerator as described below. For both cases, an optional filter predicate can be specified which can be used to select out a subset of the candidates for consideration.

Confirmation and messages for spelling correction

Spelling correction is divided into three classes in order of decreasing certainty: (1) corrections involving case errors only, (2) corrections in which all mistakes are accounted for, i.e., the only mistakes are doubled characters or transpositions, e.g., typing compille or compiel for compile (but not compil), (3) corrections in which there are some (albeit very few) actual errors, i.e., extraneous characters or omitted characters.

(The case of pattern completion is considered to fall between case (1) and (2).) The user can independently control which classes of corrections confirmation is required for, for which classes he is to be informed, or even for which classes of corrections the spelling corrector is enabled at all, by appropriate settings for the parameters confirm, inform, and disabled (which are packaged into a single record of type Modes, described below). The value of disabled is an enumerated type {never, someMistakes, allAccountedFor, patternMatch, caseError, always}, which specifies that corrections are disabled for that class and all classes that precede it in the enumerated type. For example, if disabled is someMistakes, then compille will be corrected to compile, but compil won't, because it has a missing character and hence falls into the class someMistakes. If disabled is always, then the corrector is turned off completely. Similarly, the value of inform and confirm specify that informing/confirming is requested for that class and all that precede it in the enumerated type. For example, if inform is allAccountedFor, then the user will be informed for corrections of class allAccountedFor as well as those with someMistakes. If informing is requested, but no inform procedure was supplied in the corresponding call, then simply do not inform the user.

These parameters can be specified by entries in the user profile for Spell.Confirm, Spell.Inform, and Spell.Disabled, or explicitly passed in as arguments to GetTheOne and GetMatchingList. The default settings are disabled = never, confirm = allAccountedFor, inform = patternMatch.

For those cases for which confirmation is requested, a timeout and a default value for confirmation can be specified via user profile entries Spell.Timeout and Spell.DefaultConfirm respectively. The default setting for timeout = -1, meaning never timeout. If confirmation is requested, but no confirming procedure has been supplied in the corresponding call, then the behaviour is the same as though the user rejected the correction, i.e. did not confirm.

Types, global parameters

```
ROPE: TYPE = Rope.ROPE;

SpellingList: TYPE = LIST OF ROPE;

SpellingGenerator: TYPE = REF SpellingGeneratorRecord;
SpellingGeneratorRecord: TYPE = RECORD[clientData: REF ANY, private: REF
  SpellingGeneratorPrivateRecord];
SpellingGeneratorPrivateRecord: TYPE;

Filter: TYPE = PROCEDURE[candidateRope: ROPE, unknown: ROPE] RETURNS [accept: BOOLEAN];
  -- true means consider the candidate

AbortProc: TYPE = PROC;
    responsible for checking for an abort condition, and also doing the abort. For example, when
      called from UserExecUtilities.GetTheOne, this procedure would be IF UserExec.UserAbort[exec]
      THEN ERROR UserAborted;

ConfirmProc: TYPE = PROC [msg: ROPE, timeout: INT, defaultConfirm: BOOL] RETURNS [yes:
  BOOL];
    output rope to your favorite stream, display in viewer, whatever, then request confirmation. timeout
      is how long to wait, in milliseconds, before returning default. If timeout = -1, then wait forever,
      i.e. user must confirm.

InformProc: TYPE = PROC [msg: ROPE];
    output to your favorite stream, display in viewer, whatever

CorrectionClass: TYPE = {never, someMistakes, allAccountedFor, patternMatch, caseError, always};
```

```
Modes: TYPE = REF READONLY ModesRecord;
ModesRecord: TYPE = RECORD[
    inform, confirm, disabled: CorrectionClass,
    timeout: INT, -- In msecs. -1 means never time out
    defaultConfirm: BOOL -- the default value to be used in case a confirmation times out, i.e.
      TRUE means the correction is confirmed, FALSE means it is rejected.
    ];

defaultModes: Modes;
    the modes obtained from the userprofile
```

GetTheOne, GetMatchingList

```
GetTheOne: PROCEDURE[
    unknown: ROPE,
    spellingList: SpellingList ← NIL,
    generator: SpellingGenerator ← NIL, -- if both spellingList and generator are non-NIL, candidates
      will be taken from generator until it runs out, and then from spellingList. (If both spellingList
      and generator are NIL, then the correction will fail immediately.)
    abort: AbortProc ← NIL,-- NIL => never abort.
    confirm: ConfirmProc ← NIL, -- NIL => If confirmation is required, act as though user said No.
        inform: InformProc ← NIL,-- NIL => no output.
        filter: Filter ← NIL,
    modes: Modes ← NIL -- NIL means use values in user profile
    ]
    RETURNS [ROPE];

GetMatchingList: PROCEDURE[
    pattern: ROPE,
    spellingList: SpellingList ← NIL, -- arguments same interpretation as for GetTheOne
    generator: SpellingGenerator ← NIL,
    abort: AbortProc ← NIL,
    confirm: ConfirmProc ← NIL,
    inform: InformProc ← NIL,
    filter: Filter ← NIL,
    modes: Modes ← NIL
    ]
    RETURNS [LIST OF ROPE];
```

Note...

the interface UserExecUtilities provides a convenient way of calling GetTheOne and GetMatchingList by specifying either an ExecHandle or a Viewer. These procedures will specify appropriate values for abort, confirm, and inform when calling the corresponding procedures in Spell, e.g. if given an ExecHandle, a Yes No menu will be posted and the user can confirm either via menu or by typing Y or N.

Spelling Generators

```
GeneratorFromProcs: PROC [
    initialize: PROCEDURE[self: SpellingGenerator] ← NIL,
    generate: PROCEDURE [self: SpellingGenerator] RETURNS [candidate: REF ANY], -- must narrow
      to ROPE or REF TEXT.
    terminate: PROCEDURE[self: SpellingGenerator] ← NIL, -- to be called when finished.
    clientData: REF ANY ← NIL
    ] RETURNS [SpellingGenerator];
```

Note...

the value of generate must narrow to ROPE or REF TEXT. If the value narrows to a REF TEXT, a ROPE will be created only for those candidates that the spelling corrector is going to retain across calls to generate. This enables generate to return a scratch ref text for inspection, and not have to allocate a new rope for each candidate it generates.

```
GeneratorFromEnumerator: PROC [
    enumerator: *ρOn [self: Generator.Handle],
    clientData: ρ є_ αT↑
    ] ρ є~+ρTΓ  [SpellingGenerator];
```

This procedure is useful when you have a procedure that enumerates, e.g. EnumerateGlobalFrames, and wish to use it to provide candidates for spelling corrector. Simply write an enumerator procedure which uses Generator.Produce (see Generator interface for an example) to produce candidates from inside of the enumerator, and pass this procedure in as the enumerator argument.

File correction/completion.

```
GetTheFile: PROC [
    unknown: Rope.ρO*є ,
    defaultExt: Rope.ρO*є  ← T ι□
    abort: AbortProc ← T ι□
    confirm: ConfirmProc ← T ι□
    inform: InformProc ← T ι□
    modes: Modes ← T ι□
    ] ρ є~+ρTΓ  [correct: Rope.ρO*є ];
```

If unknown does not have an extension, defaultExt will be used. If the unknown does not have an extension and defaultExt is NIL, then unknown will only be compared against files with no extension.

If unknown has an extension, and the extension (minus trailing $ if any) is not one of those on extensionList (defined below), first attempt spelling correction on extensionList. If this succeeds, see if the file name is now correct, and if so, return without doing any directory enumeration. Otherwise, attempt correction considering only files with the indicated extension.

If the unknown extension does not correspond to one of those on extensionList, attempt correction considering files with any extension, i.e. both the root and the extension may or may not be misspelled. (Note that this differs from the procedure followed if unknown does not contain an extension, and defaultExt is specified, in that in this case it is assumed that defaultExt is correct, and only files with the indicated extension are examined.)

In the event of a successful correction, the informing message will contain an extension only if the unknown contained an extension. However, the value returned, if non-NIL, will always be the full name of the file.

Here are some examples:

GetTheFile["Mumble.mesa"] => look at files with extension mesa for a misspelling of mumble (becauuse "mesa" is on extensionList)

GetTheFile["Mumble", "mesa"] => same as above, except correction message will say Mumble ->

rather than Mumble.mesa ->

GetTheFile["Mumble.msa"] => correct msa to mesa, see if mumble.mesa exists, and if not, proceed as in case 1

GetTheFile["Mumble", "msa"] => only look at files with extension msa

GetTheFile["Mumble.frob"] => examine all files with non-Null extensions, because "frob" is not on extensionlist.

GetTheFile["Mumblefrob"] => only look at files with no extensions

```
GetMatchingFileList: PROC [
    unknown: Rope.ρο*ϵ,
    defaultExt: Rope.ρο*ϵ ← τι□
    abort: AbortProc ← τι□.
    confirm: ConfirmProc ← τι□
    inform: InformProc ← τι□
    modes: Modes ← τι□
    ] ρϵ~+ρτΓ  [files: □ιΓ~ο_ ρο*ϵ ];
```

file extensions

extensionList: SpellingList;

used in spelling correction of file names as described below. initialized to (mesa, bcd, cm, config, commands, profile, df, doc, press, style, abbreviations). The user is free to add entries to this list.

AddExtension: PROC [ext: ROPE]; -- *adds ext to extensionList.*

Miscellaneous

IsAPattern: PROC [unknown: ROPE] RETURNS[BOOLEAN];

*currently true if unknown contains * not preceded by a '*

END.

Edited by Teitelman on November 3, 1982 10:45 am
this interface is intended to be fairly stable, and contains various routines used in and implemented by the userexecpackage which may also be of general use.

```
DIRECTORY
   Rope USING [ROPE],
   Spell USING [SpellingList, SpellingGenerator, Modes, Filter],
   UserExec USING [ExecHandle],
   ViewerClasses USING [Viewer]
   ;
```

UserExecUtilities: CEDAR DEFINITIONS =

BEGIN

Types

```
   ROPE: TYPE = Rope.ROPE;
   ExecHandle: TYPE = UserExec.ExecHandle;
   Viewer: TYPE = ViewerClasses.Viewer;
```

Invoking spelling corrector

procedures that invoke the spelling corrector using execHandle to supply convenient defaults for informing, confirming, and aborting. If client has an execHandle in hand, this is the right way to call the spelling corrector.

```
   GetTheOne: PROCEDURE [
      unknown: ROPE,
      spellingList: Spell.SpellingList ← NIL,
      generator: Spell.SpellingGenerator ← NIL,
      exec: ExecHandle,
      viewer: Viewer ← NIL, -- either exec or viewer should be supplied. If exec # NIL then uses
        ExecConfirm, otherwise if viewer # NIL, uses ViewerConfirm.
      filter: Spell.Filter ← NIL,
      modes: Spell.Modes ← NIL
      ]
      RETURNS [correct: ROPE];
      calls Spell.GetTheOne, providing arguments for abortProc, confirmProc, and informProc that use
        the corresponding ExecHandle. Aborting can be performed by either typing control-DEL when
        the exec viewer has the input focus, or using the STOP button. Confirmation is performed via
        ExecConfirm, described below. .
   GetMatchingList: PROCEDURE [
      unknown: ROPE,
      spellingList: Spell.SpellingList ← NIL,
      generator: Spell.SpellingGenerator ← NIL,
      exec: ExecHandle,
      viewer: Viewer ← NIL, -- either exec or viewer should be supplied. If exec # NIL then uses
        ExecConfirm, otherwise if viewer # NIL, uses ViewerConfirm.
      filter: Spell.Filter ← NIL,
      modes: Spell.Modes ← NIL
      ]
```

RETURNS [matches: LIST OF ROPE];

files

CheckForFile: PROC [file: ROPE] RETURNS [found: BOOLEAN];
returns TRUE if file is on the local disk

GetTheFile: PROC [file: ROPE, defaultExt: ROPE ← NIL, exec: ExecHandle, modes: Spell.Modes ← NIL] RETURNS [correct: ROPE];
First checks to see if file (or file.defaultExt) exists, and if not, calls Spell.GetTheFile providing suitable arguments for abortProc, confirmProc and informProc that use the corresponding ExecHandle.

GetMatchingFileList: PROC [file: ROPE, defaultExt: ROPE ← NIL, exec: ExecHandle, modes: Spell.Modes ← NIL] RETURNS[matches: LIST OF ROPE];
like GetFileName except always returns a list of files. If file does not contain a pattern, then GetMatchingFileList calls GetFileName and returns LIST of that file if found, otherwise NIL. If file does contain a pattern, calls Spell.GetFileNames, with appropriate arguments for inform, confirm, and abort.

confirmation

ViewerConfirm: PROC [msg: ROPE, viewer: Viewer, timeout: INT ← -1, defaultConfirm: BOOL ← FALSE] RETURNS[BOOLEAN];
provides a standard way of asking user to confirm an operation in a viewer. The confirmation message, msg, is printed in the messagewindow. Then, the yes/no button is posted, which allows the user to confirm or reject the correction. The user can confirm ahead, or reject ahead, as soon as SetupViewerConfirm has been called. (If the implementor wishes to abort the operation sooner than the call to ViewerConfirm, he can determine if indeed the user has confirmed/denied ahead using GetConfirmation defined below. In this case, the implementor must call FinishedViewerConfirm himself).
if timeout # -1, and the user has not confirmed before timeout milliseconds have elapsed, then defaultConfirm is returned as the value of the confirmation.
SetupConfirm: PROC [viewer: Viewer];
Call to set up menu, etc., to enable confirmation ahead.
FinishedConfirm: PROC [viewer: Viewer];
Call to take down menu, etc. Not a part of ViewerConfirm because ViewerConfirm may not be called in case that confirmation is not required.
ExecConfirm: PROC [msg: ROPE, timeout: INT ← -1, defaultConfirm: BOOL ← FALSE, exec: ExecHandle] RETURNS[BOOLEAN];
like ViewerConfirm, except (a) the confirmation message is printed in the executive, rather than the message window. The user can confirm using the keyboard, i.e. type y, n, or del, provided he has not typed ahead. Typing anything else, e.g. ?, will cause the user to be prompted with his options and allow him to confirm/reject once again. If the user has typed ahead, he can only confirm using the menu button, in order that the type ahead not be interfered with.
note that SetupConfirm and FinishedConfirm must be used the same as for ViewerConfirm.

lower level facilities.

Provided mainly in case implementor wants to be able to check whether user has confirmed or denied ahead, e.g. to abort the operation.
GetConfirmation: PROC [viewer: Viewer] RETURNS [hasConfirmed, confirmed: BOOL];

SetConfirmation: PROC [viewer: Viewer, confirmed: BOOL];
ResetConfirmation: PROC [viewer: Viewer];

sharing global resources

AcquireResource: PROC [resource: ATOM, owner: ROPE ← NIL, exec: ExecHandle] ;

a package used for sharing a global resource between several processes. If one process has already acquired the resource, an attempt to acquire it by another process will print a suitable message, and the process to wait for the resource to be freed, i.e. AcquireResource will not return until the resource is available. However, while waiting for the resource to be available, the user can abort via the usual mechanisms, and AcquireResource will raise UserExec.UserAborted.

For example, this package is used to make sure that the compiler is not run simultaneously by more than one exec, by having the compiler call AcquireResource[$Compiler] before attempting to run.

Owner is used in the message that is printed, when waiting. If owner = NIL, it defaults to (the pname of) resource. owner is typically specified where the same resource is being shared by several different clients. For example, it is not possible for two commands registered via the Exec interface (as opposed to UserExec) to run simultaneously. In this case, AcquireResource is called with resource = $ExecDotW, and owner = name of the command, so that AcquireResource can print out who it is that owns the resource.

ReleaseResource: PROC [resource: ATOM];

releases the named resource. Nop if not currently acquired.

miscellaneous

RopeSubst: PROC [old, new, base: ROPE, case: BOOLEAN ← FALSE, allOccurrences: BOOLEAN ← TRUE] RETURNS[value: ROPE];

if old is not found in base, then value = base.

if alloccurrences THEN substitute for each occurrence of old, otherwise only for first.

RopeFromFile: PROC [file: ROPE] RETURNS [value: ROPE] ;

Returns contents of file as rope, or raises IO.Error FileNotFound.

END. -- *UserExecUtilities*

UserProfile.mesa; Edited by Teitelman on December 16, 1982 11:59 am

```
DIRECTORY
   Rope USING [ROPE]
   ;
```

UserProfile: CEDAR DEFINITIONS = BEGIN

Overview

UserProfile is a package for reading information from the disk file <YourName>.profile, or if none exists, User.profile. The purpose of the profile is to allow personalized tailoring of the system. The current catalogue of ways the Cedar can be parameterized may be found in the file **UserProfile.doc**.

Entries in the user's profile are of the form: <key>: <value>RETURN, where value is either (1) a BOOLEAN, (2) an INT, (3) a TOKEN, or (4) a ListOfTokens (for more details, see UserProfile.doc). When Cedar is booted, rolledBack, or any file whose extension is "profile" is edited, the profile is parsed and a data structure which describes the information that was found in the profile is constructed. Comments (indicated using the standard "--" convention) can appear anywhere in the profile and will be ignored. The parsing will also ignore extra spaces or blank lines. Any errors or anomalies discovered when parsing or accessing the profile will cause diagnostics to be printed to the file UserProfile.log. The user will be informed that problems have been encountered via the Message Window. In no case however will the system break.

Access to individual profile entries via the procedures below is relatively efficient, since this access does not require any file reads or parsing. Therefore, many simple applications can simply read information from the profile as needed. However, some applications derive complex data structures from the profile (consider the effect of the "UserCategory" profile entry on the state of Tioga), and hence cannot afford to interrogate the profile whenever some information derived from it is used. Such an application must register a **ProfileChangedProc** and perform all of its profile interrogations from within this procedure. This guarantees that the corresponding values will be updated whenever the profile changes. The **ProfileChangedProc** will be called when it is first registered, whenever the system is booted or rolled-back, or whenever a file with extension ".profile" is saved. Fine point:

Since the ProfileChangedProc for an application is not called from that application's process(es), the application must explicitly synchronize its reads to variables derived from the profile with the ProfileChangedProc's writes to these variables. This may involve adding a monitor to an application that previously did not require one.

Accessing the Profile

The procedures described below all read simple results from the value field given a key. The default value is passed in as an argument so that the client can specify what the procedure returns should the value be absent or malformed, or the profile be missing.

Boolean: PROC [key: Rope.ROPE, default: BOOLEAN ← FALSE] RETURNS [value: BOOLEAN];

Number: PROC [key: Rope.ROPE, default: INT ← 0] RETURNS [value: INT];

Token: PROC [key: Rope.ROPE, default: Rope.ROPE ← NIL] RETURNS [value: Rope.ROPE];
returns next token following <key>:, i.e. effectively does an IO.GetRope[IDBreak]. If the first character encountered is ", then reads everything to the next matching ", and returns this as a single rope.

ListOfTokens: PROC [key: Rope.ROPE, default: LIST OF Rope.ROPE ← NIL] RETURNS [value: LIST OF Rope.ROPE];

For example, if your profile contains:

PreRun: Clock.bcd, VersionMapOpsImpl.bcd, HideousKludge.bcd, -- just until 3.4 -- WalnutSend.bcd

then ListOfTOkens["PreRun"] *will return* ("Clock.bcd", "VersionMapOpsImpl.bcd", "HideousKludge.bcd", "WalnutSend.bcd).

Note that if the user profile contains Key:{cr}, then ListOfTokens["key", default] will return NIL, not default, i.e. the entry for "key" is neither malformed or absent.

Line: PROC [key: Rope.ROPE, default: Rope.ROPE ← NIL] RETURNS [value: Rope.ROPE];

Like ListOfTokens except returns the result as a single rope, e.g. for above example, the value of Line["PreRun"] *would be* "Clock.bcd VersionMapOpsImpl.bcd HideousKludge.bcd WalnutSend.bcd".

GetProfileName: PROC RETURNS [Rope.ROPE];

returns name of file used to build the profile, i.e. either <YourName>.profile, or UserName.Profile. Returns NIL if no profile.

Registering procedures for accessing the profile

As mentioned above, if the result of accessing the user profile is going to be cached by the client, then the profile should not be accessed directly, but via a **ProfileChangedProc** which has been registered with **CallWhenProfileChanges**. This guarantees that the corresponding values will be updated whenever the profile changes.

In other words, rather than writing:

ClientCheckpointFlag: BOOL ← UserProfile.Boolean["ClientCheckpoint", TRUE];

the implementor should write:

ClientCheckpointFlag: BOOL;

Init: ProfileChangedProc = {ClientCheckpointFlag ← UserProfile.Boolean["ClientCheckpoint", TRUE];

UserProfile.CallWhenProfileChanges[Init];

CallWhenProfileChanges: PROC [proc: ProfileChangedProc];

proc is called when it is first registered with reason = firstTime.

ProfileChangedProc: TYPE = PROC [reason: ProfileChangeReason];

ProfileChangeReason: TYPE = {firstTime, rollBack, edit};

ProfileChanged: PROC [reason: ProfileChangeReason];

Calls all of the ProfileChangedProcs. *This procedure is automatically called after a rollback, whenever the* UserExec.Login *is called, or whenever the a file with extension 'profile' is saved by tioga.*

GetErrorLog: PRIVATE PROC RETURNS [fileName: Rope.ROPE];

if there were any errors reported, this closes the stream and returns the name of the file.

END.

UserProfile.Doc, documentation for UserProfile options
last edited by Teitelman, April 19, 1983 11:45 am

A number of components of Cedar permit the user to parameterize their behaviour along certain predefined dimensions via a mechanism called the User Profile. Whenever the user boots or rollsback, his user profile is consulted to obtain the value for these parameters. This operation is performed by consulting a file whose name is <YourName>.profile, e.g. McGregor.Profile, or if no such file exists, a default profile User.Profile. The entries in this profile are of the form <key>: <value>RETURN, where key is a sequence of alphanumerics or .'s (case does not matter) and value is either (1) a BOOLEAN, (2) an INT, (3) a TOKEN, which is a rope consisting of either a sequence of characters delimited by SP, CR, TAB, Comma, Colon, or SemiColon, or, in the case that the first character is '", the sequence of characters up to the next matching '", or (4) a ListOfTokens. Comments can appear at any point in the profile, and are ignored.

This file defines and documents the entries in the user profile. Each entry is presented as though it were a mesa declaration, although of course it is not. Comments following the entry explain the effect of that entry.

In all cases, if the corresponding entry does not appear in your personalized user profile (or the default user profile), the value of the corresponding parameter is the indicated default. If you want to specify some value other than this default, or want to include the corresponding entry along with its default in your profile just so that you can see at a glance what settings are in effect, simply include it in your own profile. Unless otherwise specified, changing or adding an entry should be "noticed" as soon as you click the Save menu button.

Viewers/Tioga related parameters

UserCategory: TOKEN ← Intermediate;
affects which Tioga commands are turned on. Categories are {Beginner, Intermediate, Advanced}. *You need Advanced to enable the various commands for manipulating nodes. Will not be noticed until you boot.*

DefaultFontFamily: TOKEN ← Tioga;
the name of the default font to use for all viewers. Will not be noticed until you boot. Any changes to this entry will not be noticed until the next time you boot. (Consult a wizard before changing this to anything other than Tioga.)

DefaultStyle: TOKEN ← Cedar;
the name of the default style.

ExtensionStyles: ListOfTokens ← NIL;
specifies the default style to be used with documents that do not explicitly name a style. The style is determined by the extension in the file name. The entry is of the form <extension1> <stylename1> <extension2> <stylename2>.

DefaultTiogaMenus: ListOfTokens ← places;
specifies which menus, if any, should automatically be displayed when you create a new Tioga viewer, e.g. places levels. none *means no menus will be added, i.e. only the "*Clear Reset...*" menu will be displayed.*

OpenFirstLevelOnly: BOOL ← FALSE;
If set to true, documents will be opened with only their first level showing.

DefaultFontSize: INT ← 10;
Note: strike fonts only come in a few sizes. Any changes to this entry will not be noticed until the

next time you boot. (Consult a wizard before changing this to anything other than Tioga.)

PreLoad: ListOfTokens ← NIL;
a sequence of files for which viewers are automatically opened when you boot. Many profiles have "TiogaDoc.tioga" here, which will open a viewer on the documentation for Tioga. Any changes to this entry will not be noticed until the next time you boot.

TiogaTIP: TOKEN ← Default;
defines your user interface to Tioga. Value is a sequence of TIP files or the special token "Default", which will be layered so that operations defined in earlier tables take precedence over those defined later. Any changes to this entry will not be noticed until the next time you boot, or you invoke the userexec command ReadTiogaTipTables.

SourceFileExtensions: ListOfTokens ← mesa tioga df cm config style;
This determines what extensions Tioga should look for in opening files.

ImplFileExtensions: ListOfTokens ← mesa cedar;
This determines what extensions Tioga should look for in opening files requested via GetImpl.

ScrollTopOffset: INT ← 3;
When you do a Find command you may want to see a few lines in front of the match to give you more context. This parameter tells Tioga how many extra lines to want in such situations.

ScrollBottomOffset: INT ← 5;
When you are typing and the caret goes to a new line just off the bottom of the viewer, Tioga will automatically scroll the viewer up a little to make the caret visible again. This parameter controls how far up to scroll; a big number causes larger but less frequent glitches.

SelectionCaret: TOKEN ← balance;
"balance" *means place the caret at the end nearer the cursor when the selection is made. Some people have requested to have the caret always placed at one end or the other, hence this profile entry. The choices are {*before, after, balance*}.*

YSelectFudge: INT ← 2;
This lets you specify a vertical displacement for making selections. Tioga behaves as if you had pointed this number of points higher up the screen so that you can point at things from slightly below them.

UnsavedDocumentsCacheSize: INT ← 4;
controls the number of unsaved documents the system will remember.

ShowUnsavedDocumentsList: BOOL ← TRUE;
If true, a viewer will be created holding an up-to-date list of the unsaved documents that can still be reloaded.

EditHistory: INT ← 20;
Tioga keeps a history of the specified number of edit events. The EditHistory tool will let you undo these events.

EditTypeScripts: BOOL ← TRUE;
If true, typescripts behave the same as Tioga documents when the selection point is not at the end of the document, i.e. DEL means delete, ↑X means exchange, and typein is simply inserted at the selection point. Thus, you can edit material that has been typed but not yet read, i.e. anything up to the last carriage return, and the edited characters will be what the client program sees.

However, this also means that you cannot simply point anywhere *in the typescript and start typing and have the material automatically be inserted at the end of the document the way it used to (it will be inserted where at the selection point, the same as when editing a Tioga document). Note that a convenient, single keystroke way of moving the selection to the end of the document is to use the NEXT key (Spare2).*

Printer related options.

Note: most of the following can be overriden on the command line invoking the print command using appropriate switches.

Hardcopy.PressPrinter: TOKEN ← NIL;
the printer where your output gets sent, e.g. Clover. *Used by both print and tsetter.*

Hardcopy.PrintedBy: TOKEN ← NIL;
If omitted, defaults to name of currently logged-in user

Hardcopy.Landscape: BOOL ← TRUE;
TRUE *means print in landscape mode,* FALSE *in portrait mode*

Hardcopy.Font: TOKEN ← IF Hardcopy.Landscape THEN "Gacha6" ELSE "Gacha8";

Hardcopy.Columns: INT ← IF Hardcopy.Landscape THEN 2 ELSE 1;

Hardcopy.Tab: INT ← 8;
the number of spaces between tab stops

Hardcopy.TemporaryPressFiles: BOOL ← FALSE;
Temporary Press Files are automatically deleted as soon as they are successfully sent to the printer. This profile parameter controls the initial *setting when a new tsetter tool is created. It can be changed on a tool by tool basis using a button in the tsetter tool.*

UserExecutive options

RegisteredCommands: TOKEN ← NIL;
the token is interpreted as either (a) a sequence of commands that you want the UserExecutive to recognize in addition to the ones that it already knows about, or (b) the the name of a file containing those commands. Each command must be on a separate line, and the line must be of the form: CommandName Explanation name-of-bcd, where Explanation is a rope delimited by "'s, and name-of-bcd is optional and defaults to the same as CommandName. (Note that any " appearing inside of the rope must be preceded by a \ or it would terminate the rope. Similarly, any \ to be included in the rope must itself be preceded by a \.)

Example:

RegisteredCommands: "
Bringover \"Retrieves files from a remote server to your local disk, using a specified df file.\"
Chat \"Creates a Chat viewer, i.e. a \\\"terminal\\\" for communicating with Maxc, Ivy, Ernestine, etc.\"
JaM \"Creates a JaM typescript.\"

```
Print \"Sends either a plain-text file or a press file to a printer.\"
SModel \"Stores files on remote servers using a specified df file.\"
VerifyDF \"Verifies that the contents of specified df file are consistent.\"
WalnutSend \"Creates a viewer for sending mail.\"
"
```

Note: all of the above are already *registered by virtue of their being in the file RegisteredCommands.catalogue, which is on your disk. Use this profile entry to register new, or esoteric commands.*

CommandsFrom: TOKEN ← NIL;
either (a) a sequence of command lines to be executed by the UserExecutive, exactly as though you had typed them in, or (b) the name of a file which contains the commands. The commands can appear directly in the user profile, e.g.

```
CommandsFrom: "
WalnutSend -- loads starts, and calls WalnutSend, i.e. brings up the little envelope
Alias DoBoth (file) compile file '; bind file
"
```

or this entry can be the name of a file which contains the commands, e.g.

```
CommandsFrom: Teitelman.commands
```

Note: any changes to this entry will not be noticed until the next time you boot.

ShowStatistics: BOOL ← FALSE;
if true, causes userexecutive to print out computation time, number of words allocated, and page faults for each command that it executes.

CreateSessionLog: BOOL ← TRUE;
if true, causes userexecutive to create a log file, called Session.log, in which is written all of the material that appears in each of the workAreas, suitably bracketed so you can tell which is which, as well as a record of all files that were saved.

CreateWorkAreaLogs: BOOL ← FALSE;
if true, causes userexecutive to create log files for each work area, of the form WorkAreax.log.

CreateChangesLog: BOOL ← TRUE;
if true, causes userexecutive to create log file containing record of all changes made to files. Each time a file is saved, its name, date, and changelog entry, if any, is written to this log (requires NewStuffImpl to be loaded).

WhenLogFileExists: TOKEN ← "Rename";
Determines what happens when a log file already exists. Value is either {Rename, Append, OverWrite}. Rename means rename the existing log file by appending a $ to its name. Append means append the new log file to the existing one.

Spelling corrector options

Spell.inform: TOKEN ← allAccountedFor;
what classes of corrections to inform the user about

Spell.confirm: TOKEN ← allAccountedFor;
what classes of corrections to request confirmation before performing

Spell.disabled: TOKEN ← never;
what classes of corrections not to do at all

the values of the previous three entries are spelling correction classes, which are drawn from the following set: {never, someMistakes, allAccountedFor, patternMatch, caseError, always}. *The value of the parameter specifies that the corresponding operation should be performed for the indicated class, and all of those that precede it in the enumerated type. For example, if disabled is* someMistakes, *then* compille *will be corrected to* compile, *but* compil *won't, because it has a missing character and hence falls into the class* someMistakes. *If disabled is* always, *then the corrector is turned off completely. Similarly, the value of inform and confirm specify that informing/confirming is requested for that class and all that precede it in the enumerated type. For example, if inform is* allAccountedFor, *then the user will be informed for corrections of class* allAccountedFor *as well as those with* someMistakes. *With the above defaults, the system will correct* compille *to* compile *without confirmation (since it is of class* allAccountedFor*), but tell you what it has done, will correct* compil *to* compile *but ask you to confirm first (since it is of class* someMistakes*), and correct* rttypes *to* RTTypes *without asking you or telling you (since it is of class* caseError*). For more details, see the spelling interface, spell.mesa*

Spell.timeout: INT ← -1;
number of milliseconds to wait before timeout on a proposed correction. -1 means never timeout.

Spell.defaultConfirm: BOOL ← FALSE;
what to return if a proposed correction involves confirmation and you allow the correction to timeout. TRUE *means ok to do the correction,* FALSE *means not.*

Spell.giveUpAfter: INT ← -1;
number of milliseconds to attempt correction before giving up.

Spell.assumeFirstCharCorrect: BOOL ← FALSE;
For use in context of correcting a where there is a large potential set of candidates, e.g. misspelled frame name, file name, but this set of candidates can be structured so that only those candidates with the same first character as the one that was misspelled will be considered. Note that it is always the case that these candidates are considered first. This parameter simply says if you don't find the correct spelling there, whether or not to search the other candidates.

Spell.viewerSpellDisabled: BOOL ← FALSE;
If TRUE, *then no correction for mistyped filenames in context of loading or opening a viewer.*

Compiler/Binder

Compiler.IconicLogs: BOOL ← FALSE;
If TRUE, tells userexecutive to always create compiler/binder logs iconic, regardless of whether or not the input focus is in the executive in which the compilation has taken place.

Compiler.BlinkLogs: BOOL ← TRUE;
If FALSE, tells userexecutive never to blink compiler/binder logs.

Compiler.SeparateLogs: BOOL ← FALSE;
If TRUE, tells userexec to create separate logs and viewers on these logs for each file that contains

errors or warnings. Moreover, these viewers are created as soon as the errors are encountered. Thus, if you compile a sequence of files, and the first file contains errors, you can begin working on those errors before the compilation finishes.

Compiler.AutoSave: BOOL ← FALSE;
If TRUE, when the user attempts to compile a file which has been edited but not saved, causes the system to automatically save the file without asking the user.

Compiler.Switches: TOKEN ← NIL;
default switches for compiler. Note that /-g can be used to cause separate compiler error logs for each file compiled.

Binder.Switches: TOKEN ← NIL;
default switches for binder

Walnut and Squirrel

Walnut.ReplyToSelf: BOOL ← FALSE;
if TRUE, *causes walnut to always supply a Reply-To: field.*

Walnut.InitialActiveRight: BOOL ← TRUE;
true says to bring up the active message set on the right column, false on left.

Walnut.InitialActiveOpen: BOOL ← FALSE;
true says open a message set viewer on Active.

Walnut.InitialActiveIconic: BOOL ← FALSE;
if true and InitialActiveOpen = TRUE, then the Active message set viewer is opened as an icon.

Walnut.MsgSetButtonBorders: BOOL ← FALSE;
if TRUE, puts borders around the MsgSet buttons in the control window.

Walnut.WalnutSegmentFile: TOKEN ← "Walnut.Segment";
value is the name of the file to be used for the walnut data base. Primarily to allow users to experiment with alpine.

Walnut.WalnutLogFile: TOKEN ← "";
Name of log file. If not specified, then derived from Walnut.WalnutSegmentFile by changing extension to .DBLog.

Squirrel.SquirrelWindow: BOOL ← FALSE;
TRUE *if you want the squirrel window on the screen.*

miscellaneous

AutoCheckpoint: BOOL ← FALSE;
TRUE *says to automatically make a checkpoint when you boot.*

ClientCheckpoint: BOOL ← TRUE;
If TRUE, *puts a CheckPoint button at the top of your screen.*

WorldSwapDebug: BOOL ← FALSE;

TRUE *means go to worldswap debugger on an uncaught signal.*

AutoIdleTimeout: INT ← 20;
if you leave your terminal unattended for more than the corresponding number of minutes, is equivalent to pressing the "Idle" button. 0 means never time out.

FileSwitches: TOKEN ← NIL;
argument is interpreted as the switches (sequence of characters) to be supplied when booting using the "File" button.

Edited on December 2, 1982 1:10 pm, by Teitelman
changes to: OpenFirstLevelOnly, UnsavedDocumentsCacheSize, ShowUnsavedDocumentsList, SourceFileExtensions
Edited on December 9, 1982 12:59 pm, by Teitelman
changes to: Walnut.InitialActiveOpen, Walnut.InitialActiveIconic, Walnut.WalnutOnlyUser
Edited on February 9, 1983 3:10 pm, by Teitelman
changes to: CreateLogFiles
Edited on March 4, 1983 11:59 am, by Teitelman
changes to: Walnut, Walnut, Walnut, Walnut, Squirrel
Edited on March 10, 1983 5:22 pm, by Teitelman
changes to: Hardcopy
Edited on March 13, 1983 2:01 pm, by Teitelman
changes to: CreateChangesLog, Spell, WorldSwapDebug
Edited on March 25, 1983 4:51 pm, by Teitelman
changes to: EditTypeScripts, CreateSessionLog
Edited on April 19, 1983 11:45 am, by Teitelman
changes to: CreateSessionLog, CreateWorkAreaLogs, CreateChangesLog, WhenLogFileExists

Last edited by Mitchell on December 20, 1982 4:33 pm
by vanLeunen on June 6, 1983 10:49 pm

By Jim Mitchell

Annotated Cedar Examples

Version 3.5.2

XEROX — FOR INTERNAL USE ONLY

Abstract This section contains a set of examples of Cedar programs for your reading pleasure. These are actual programs that can be run, used as parts of other programs, or treated as templates to be edited into new programs with similar structures.

This memo is probably out of date if it is in hardcopy form. It documents Release 3.5.2 of Cedar, December 1982.

[If you are reading this document on-line in Cedar, try using the Tioga Levels and Lines menus tobrowse through the top few levels of its structure before reading it straight through.]

Annotated Cedar Examples: Contents

0. Introduction

This section contains a set of four examples of Cedar programs for your reading pleasure (it assumes you are already familiar with the Cedar Language, either by previous osmosis or by having read the Cedar Language Overview). These are actual programs that can be run, used as parts of other programs, or treated as templates to be edited into new programs with similar structures.

For each example there is a short discussion of its purpose followed by the program.

After each program there is a set of notes to help in reading it. The notes are keyed to the programs by the numbers in parentheses to the *right* of some lines, e.g., as "--**(note 5.1)**".

You will also find references to lines in the program from the notes, as, e.g., "**[line 1.1]**". The corresponding lines in the programs are marked as, e.g., **[1.1]**.

When Mesa identifiers appear in the notes, they are displayed in italics to make reading easier; e.g., *ReverseName*, *CommandProc*.

You should read the examples for understanding and use the notes as references, rather than the other way round.

1. A simple, but complete program

This first example shows how to write a simple, but complete Cedar program. It illustrates a number of features of the Cedar language and system as well as some of the stylistic conventions used by Cedar programmers (see the Style section):

It uses Cedar ROPEs for string manipulation;

It uses the *IO* interface to create and use terminal input and output streams;

It registers procedures with the Cedar User Executive (*UserExec*) so that you can invoke them like builtin commands of the UserExec;

One of these registered commands creates a process by means of a FORKed procedure call each time you invoke it from the UserExec;

The process then creates a simple *Viewer* with which you can interact to calculate the values of simple expressions.

The program can be run by typing "Run SimpleExample" to the Cedar UserExec. All it does at that point is register two commands with the UserExec.

The first command, *ReverseName*, simply types your logged-in user name backwards in the UserExec typescript (unless you are Bob Taylor).

The second command, *Calculate*, creates a separate viewer into which you can type simple expressions of the form "constant (("+" | "−") constant)*", terminated by a carriage return. It will display the value of the expression. If you wish, you can create multiple calculator viewers by giving the *Calculate* command to the UserExec more than once.

1.1 SimpleExample.mesa — Rick Cattell

```
-- File: SimpleExample.mesa                                              (note 1.1)
-- Last edited by: Mitchell on December 16, 1982 2:32 pm

DIRECTORY
   IO USING [char, Close, CreateViewerStreams, Error, GetChar, GetInt, Handle, IDBreak, int,
            PeekChar, PutF, PutFR, Reset, rope, SkipOver],
   Process USING [Detach],
   Rope USING [Cat, Equal, Fetch, Find, FromChar, Length, ROPE, Substr, Upper],
   UserExec USING [CommandProc, GetNameAndPassword, RegisterCommand];

SimpleExample: CEDAR MONITOR                                          --(note 1.2)
   IMPORTS IO, Process, Rope, UserExec =                               --(note 1.3)

BEGIN
ROPE: TYPE = Rope.ROPE;                                                         --(note 1.4)
windowCount: INT ← 0; -- a count of the number of calculators on the screen

ReverseName: UserExec.CommandProc -- [exec: UserExec.ExecHandle, clientData: REF ANY ← NIL] RETURNS
         [ok: BOOLEAN ← TRUE, msg: ROPE ← NIL] -- =
-- Reverses the user's login name and prints it out in the exec window.    (note 1.5)
   BEGIN
   userName: ROPE ← UserExec.GetNameAndPassword[ ].name;                      --(note 1.6)
   execStream: IO.Handle ← exec.out;          -- exec is an arg to ReverseName     (note 1.7)
   backwordsName: ROPE ← NIL;
   -- Remove ".PA" if it is on the end of the user name, and check for user name Taylor.
   dotPos: INT = userName.Find["."];                                          --(note 1.8)
   IF dotPos#-1 THEN userName ← userName.Substr[0, dotPos];
   IF userName.Equal[s2: "Taylor", case: FALSE] THEN execStream.PutF["Hi, Bob\n"];
   -- Now reverse the name: convert chars to upper cases and concatenate them in reverse order
   FOR i: INT DECREASING IN [0..userName.Length[ ]) DO
      backwordsName ← backwordsName.Cat[
        Rope.FromChar[Rope.Upper[userName.Fetch[i]]]]              --[1.1] (note 1.9)
      ENDLOOP;
   execStream.PutF["Your user name backwards is: %g\n", IO.rope[backwordsName]];
                                                                    --(note 1.10)
   END;

MakeCalculator: ENTRY UserExec.CommandProc = BEGIN                   --(note 1.11)
-- Puts up a calculator window.
   title: ROPE;
   windowCount ← windowCount+1;
   title ← IO.PutFR["Adding machine number %g", IO.int[windowCount]];   --(note 1.12)
   TRUSTED {Process.Detach[FORK Calculate[title]]}                 --(note 1.13)
   END;

Calculate: PROC[title: ROPE] = BEGIN
-- Creates typescript window separate from UserExec with given title.
-- Uses IO procedures to read an integer expression from the typescript and print its value.
   opChar: CHAR;
   number: INT;
   answer: INT;
   in, out: IO.Handle;
   [in: in, out: out] ← IO.CreateViewerStreams[title];                 --(note 1.14)
   DO     -- Read an expression, terminated by a CR:
      ENABLE IO.Error =>                                                  --(note 1.15)
         IF ec=SyntaxError THEN {
            in.Reset[ ];
            out.PutF["\nIncorrect input.  Please retype the expression.\n\n"];
            LOOP}
         ELSE EXIT;
      opChar ← '+;          -- first integer is positive
      answer ← 0;       -- initialize sum to zero
      out.PutF["Type one or more integers separated by + and - and terminate with CR to
       compute value:\n"];
      DO -- Read an integer and an operator, skipping leading blanks in both cases
         number ← in.GetInt[ ];
         SELECT opChar FROM
            '+ => answer ← answer + number;
            '- =>  answer ← answer - number;
            ENDCASE => {
               out.PutF["Illegal operator (%g).  Please retype the expression.\n\n",
                IO.char[opChar]];
               EXIT};
         IF in.PeekChar[ ]='\n THEN GO TO NextLine;    -- the normal way out
         in.SkipOver[IO.IDBreak];
         opChar ← in.GetChar[ ];
         REPEAT
            NextLine => {
               [ ] ← in.GetChar[ ]; -- toss CR                            (note 1.16)
               out.PutF["\nThe answer is: %g.\n\n", IO.int[answer]] };
         ENDLOOP;
      ENDLOOP;
   END;

-- Start code registers a Calculate and ReverseName command, which must be invoked for this
      program to do anything:
UserExec.RegisterCommand[                                               --(note 1.17)
   name: "Calculate", proc: MakeCalculator, briefDoc: "A simple adding machine"];
UserExec.RegisterCommand[
   name: "ReverseName", proc: ReverseName, briefDoc: "Reverses your user name"];
END.
```

CHANGE LOG --(note 1.18)

Created by Cattell on 21-Apr-82 13:55:09

Changed by Cattell on May 25, 1982 10:58 am
-- added use of RegisterCommand to fire up a Calculator viewer instead of creating it when program first run.

Changed by Mitchell on August 11, 1982 5:00 pm
-- changed main loop of Calculate to correctly catch IO.Error when Destroy menu button invoked.
Edited on December 16, 1982 2:22 pm, by Mitchell
changes to: DIRECTORY *IO, --CreateTTYStreams --> CreateViewerStreams, --,* UserExec.RegisterCommand[*moved out.PutF to NextLine exit of inner loop so it doesn't print out an answer when an incorrect operator is typed*

1.2. Notes for SimpleExample

(1.1) These two stylized comments give the name of the module, who last edited it, and when (and Tioga contains features that will automatically update the last-edited time when a save is done - see the *Tioga Editor* chapter of this documentation). The section on Style contains a list of the stylistic conventions recommended for Cedar programmers.

(1.2) A MONITOR module is like a PROGRAM module except that it can be used to control concurrent access to shared data (note 1.11).

(1.3) *SimpleExample* imports four interfaces, *IO, Process, Rope,* and *UserExec* because it needs to call procedures defined in those interfaces.

(1.4) Since ROPEs are so heavily used in the program, an unqualified version of the type name is generated simply by equating it with the type in the *Rope* interface. This is a commonly used means for making one or a small set of names from interfaces available as simple identifiers in a module.

(1.5) The argument and returns lists given here as comments show the type of *Userexec.CommandProc*; they were inserted semi-automatically using Tioga's "Expand Abbreviation" command to assist in reading the program. *ReverseName* has this type so that it can be registered with the Cedar *UserExec* as a command that a user can invoke by typing a simple identifier **(note 1.18).**

The *ReverseName* procedure prints out the user's login name in reverse order. It strips off the registry extension (e.g. ".PA") and does something different for user name "Taylor".

(1.6) *GetNameAndPassword* has the type

PROC RETURNS [name, password: ROPE]

userName is initialized to the value of the "*name*" return value from *UserExec.GetNameAndPassword* by qualifying the call with "*.name*".

(1.7) In the *object-oriented style* adopted by many Cedar interfaces, the object type provided by an interface, *Foo*, is conventionally named *Foo.Handle.*

(1.8) The notation, *userName.Find*[".'], is interpreted by the compiler as follows: Look in the interface where the type of *userName* is defined (*Rope* in this case) and look for the procedure *Find*, whose first parameter is a ROPE. Generate code to call that procedure, inserting *userName* at the head of its argument list. Thus *userName.Find*["."] is an alternate way of saying *Rope.Find*[*userName*, "."]. The *userName.Find* form is called *object-style notation*, and the *Rope.Find* form is called *procedure-oriented notation.*

Note also that *dotpos* is constant over the scope of its declaration, so it is initialized with "=" to prevent any subsequent assignment to it.

(1.9) *Find, Fetch, Equal, Substr, Cat,* and *FromChar* are all procedures from the *Rope* interface. The program uses object-style notation for those calls whose first argument is a nameable object, e.g., *userName*, and procedure-oriented notation otherwise. [line 1.1] is a good example of both. Here it is, spread out to exhibit the two forms:

```
backwordsName ←
    backwordsName.Cat[             -- object-oriented notation
        Rope.FromChar[              -- procedure-oriented notation
            Rope.Upper[                        -- procedure-oriented notation
                userName.Fetch[i]      -- object-oriented notation
            ]
        ]
    ];
```

(1.10) *IO.PutF* is the standard way of providing formatted output to a stream (much like FORTRAN output with format). Its first argument is an *IO.Handle.* The second is a ROPE with embedded Fortran-like formatting commands where variables are to be output. The "%g" format is the most useful one; it will handle any sort of variable: INTEGER, CHARACTER, ROPE, etc., in a general default format. The third through last arguments are values to be output, surrounded by calls to inline *IO* procedures that tell *PutF* the type of the argument: *int*[*answer*], for example. For details, see the description of *IO* in the Catalog chapter.

(1.11) *MakeCalculator* is an ENTRY procedure to this monitor because it updates the variable *windowCount*, which is global in *SimpleExample* and therefore could be incorrectly updated if multiple processes were to invoke *MakeCalculator* concurrently. When invoked, *MakeCalculator* creates a new calculator viewer on the screen by invoking *Calculate* and forking it as a new process **(note 1.13)**.

(1.12) *IO.PutFR* (also known by the name *PutFToRope* has the same arguments as *IO.PutF*, but returns a ROPE containing the resultant output rather than sending it to a stream

(1.13) The FORK operation creates a new process to execute a regular procedure call. It returns a *ProcessHandle* which can either be used to synchronize with the process later (when its root procedure executes a RETURN) and acquire its return values. Alternatively, it can be passed to the procedure *Process.Detach*, in which case the process can no longer be JOINed to and will simply disappear when its root procedure RETURNs.

Note that the combined FORKing and detaching is enveloped in a TRUSTED block. This is because the *Detach* operation is not intrinsically SAFE, although the stylized combination of FORK and *Detach* used here actually is. You should use this paradigm when detaching a FORKed process: don't assign the process handle returned by FORK to anything, just use it as the single argument to *Process.Detach* and surround the whole by a TRUSTED block.

(1.14) *CreateViewerStreams* makes a pair of *IO.Handle* objects, *in* for terminal input, and *out* for (teletype-style) output. These two returned values are assigned to the local variables *in* and *out* using a keyword extractor to ensure that we assign them correctly.

(1.15) *Calculate* parses a simple expression from the *in* stream using a number of useful *IO* functions for input, such as *GetInt* to provide an INT (skipping over leading blanks), *PeekChar* to look at the next character while leaving it in the stream for the next input call, and *SkipOver* to scan to the first occurence of an interesting character.

These various *IO* procedures can raise the ERROR *IO.Error*, so there is a catch phrase enabled over the outer (endless) loop to deal with *IO.Error*. *IO.Error* also carries an argument, *ec*, which is an enumerated type defined in *IO*.

If *ec=SyntaxError*, the user has typed something that is not lexically correct, and the program will regain control and go around the outer loop again.

The only other possibility is that the *in* and *out* streams were closed (*ec=StreamClosed*) as a side effect of the user having destroyed the *Calculator* viewer by bugging the *Destroy* menu item. This error and code can emanate from any of the *IO* operations in the loop. In this case the program exits the outer loop, returns from *Calculate*, and the process that was forked then terminates and disappears.

(1.16) If a procedure returns values, the Cedar language requires you to assign them to something, even if you have no use for them in a specific case. The standard mechanism for ignoring a procedure's returned value(s) is an empty *extractor*, as here.

(1.17) Here, at the end of the module, is the code that is executed when the module is started. You can create an instance of a module and start it using the *UserExec* "Run" command. The loader creates instances of programs when loading configurations. If a component of a configuration is not STARTed explicitly, then it will be started automatically (as the result of a trap) the first time one of its procedures is called. See the Language Reference Manual for more information.

In this case, the start code for the module consists of two calls on the procedure *UserExec.RegisterCommand*, which register the commands "Calculate" and "ReverseName" with the *UserExec* so that you can invoke the procedures *MakeCalculator* and *ReverseName*, respectively, by typing their command names.

These procedure calls also illustrate the ability to specify the association between a procedure's formal parameters and the arguments in a specific call using keyword instead of positional notation. Generally, keyword notation is preferred over positional for all but simple one- or two-argument calls, and it is definitely better for two-argument calls if the types of the arguments are the same, e.g., either

Copy[*to*: *arg1*, *from*: *arg2*] or

Copy[*from*: *arg2*, *to*: *arg1*]

is preferable to

Copy[*arg1*, *arg2*]

(1.18) By convention each module has a CHANGE LOG following the Cedar text. It is used to record an abbreviated history of changes to the module, saying who made each change, when they made it, and a brief description of what was done. You can insert a new entry in the log by left-clicking Tioga's ChangeLog viewer button; see the Tioga chapter for details.

2. A general sort package for lists

This is an example of a Cedar *package* and consists of two modules: *ListSortRef* specifies the interface that client programs must import to make use of the package; *ListSortRefImpl* implements that interface.

The package sorts lists of items according to a compare function passed to it along with the list to be sorted. The algorithm is presented as exercise 5.2.4-17 (p. 169) in Knuth Volume III, and is attributed to John McCarthy. The comments in the code give some of the important invariants.

2.2. ListSortRef.mesa — Mark Brown

```
-- File: ListSortRef.mesa
-- Last edited By Mitchell on December 20, 1982 3:09 pm

DIRECTORY
   Environment USING [Comparison];

ListSortRef: CEDAR DEFINITIONS =                                    --(note 2.1)

BEGIN
Comparison: TYPE = Environment.Comparison;
CompareProc: TYPE = PROC[r1, r2: REF ANY] RETURNS [Comparison];
     -- Returns less if r1↑ < r2↑, equal if r1↑ = r2↑, greater if r1↑ > r2↑.
CompareError: ERROR[reason: Reason];
Reason: TYPE = {typeMismatch, invalidType}; -- errors from a CompareProc

Sort: PROC[list: LIST OF REF ANY, compareProc: CompareProc]         --(note 2.2)
   RETURNS[LIST OF REF ANY];
-- Destructive sort; copy list first if nondestructive sort is needed.
-- Returns a list containing the same REFs as list, arranged in ascending order
--according to compareProc. Order of equal items is not preserved.

Compare: CompareProc;
   -- ! SafeStorage.NarrowRefFault
   -- (if both parameters do not NARROW to one of ROPE or REF INT.)
   -- Compares ROPE : ROPE (case significant) and REF INT : REF INT
   END.--ListSortRef
```

CHANGE LOG

Created by MBrown on 21-Apr-81 13:55:09

Changed by MBrown on 10-Dec-81 9:50:15
-- *Change comment in Compare (now compares ROPE instead of REF TEXT.)*

Changed by MBrown on July 5, 1982 4:59 pm
-- CEDAR definitions, eliminate type Item.

Edited on December 20, 1982 3:24 pm, by Mitchell
added CompareError and Reason to make errors generated by a CompareProc more germane to it instead of the facilities it calls.

2.3. Notes for ListSortRef interface

(2.1) Interfaces are defined in DEFINITIONS modules. Basically, a DEFINITIONS module contains the constants, types, and procedure headings (actually procedure types) needed to use a package.

Frequently, an interface defines some kind of object with operations (procedures) for that class of object (the *Rope* interface is a good example of such an object-oriented interface).

(2.2) This definition for *Sort* is used for type-checking in any module that contains a call on *Sort* or provides an actual implementation of it (note 2.3). Thus, client and implementing programs can count on the fact that they agree on the type of *Sort*.

This separation of definition and implementation enables client and implementing modules to be developed independently of each other, which provides some parallelism in the program development process. Since they are more abstract, interfaces are typically more stable than either clients or implementations.

The first argument to *Sort* has type LIST OF REF ANY. Lists in Cedar are similar to Lisp lists. In this case the type "LIST OF REF ANY" is equivalent to the following pair of Cedar type definitions:

```
ListOfRefAny: TYPE = REF ListCell;
        ListCell: TYPE = RECORD[first: REF ANY, rest: ListOfRefAny];
```

The only things that are special about lists are that the compiler automatically generates the equivalent of the above two type definitions for each list type you define, the language provides a CONS operator for allocating and initializing list cells [line 2.1], and list types with the same elements are equivalent, whereas record types are not.

2.4. OnlineMergeSortRefImpl.mesa — Mark Brown

```
-- File OnlineMergeSortRefImpl.mesa
-- Last edited by MBrown on July 5, 1982 3:57 pm,
-- Mitchell on December 20, 1982 3:21 pm

DIRECTORY
   Rope USING [ROPE, Compare],
   ListSortRef,
   SafeStorage USING [NarrowRefFault];

OnlineMergeSortRefImpl: CEDAR PROGRAM                          --(note 2.3)
   IMPORTS Rope, SafeStorage
   EXPORTS ListSortRef =

BEGIN
ROPE: TYPE = Rope.ROPE;
Comparison: TYPE = ListSortRef.Comparison;
CompareProc: TYPE = ListSortRef.CompareProc;

Sort: PUBLIC PROC [list: LIST OF REF ANY, compareProc: CompareProc]     --(note 2.4)
   RETURNS [LIST OF REF ANY] = {
   -- The sort is destructive and NOT stable, that is, the relative positions in the result of nodes with
            equal keys is unpredictible. For a nondestructive sort, copy list first.
      a, b, mergeTo: LIST OF REF ANY;
      mergeToCons: LIST OF REF ANY = CONS[NIL, NIL];                 -- [2.1] (note 2.5)
      index: TYPE = INT [0..22]; -- 22 is the number of bits in word-address space minus 2 (cons
            cell takes 2**2 words).
      sorted: ARRAY index OF LIST OF REF ANY ← ALL [NIL];        -- sorted[i] is a sorted list of length $2^i$ or NIL
            (note 2.6)
      -- make each pair of consecutive elements of list into a sorted list of length 2, then merge it
            into sorted.
      UNTIL (a ← list) = NIL OR (b ← a.rest) = NIL DO
         list ← b.rest;
```

```
            IF compareProc[a.first, b.first] = less THEN                    --(note 2.7)
                { a.rest ← b; b.rest ← NIL }
            ELSE { b.rest ← a; a.rest ← NIL; a ← b };
            FOR j: index ← 0, j+1 DO
                IF (b ← sorted[j]) = NIL THEN { sorted[j] ← a; EXIT }
                ELSE {       --merge (equal length) lists a and b
                    sorted[j] ← NIL;
                    mergeTo ← mergeToCons;
                    DO --assert a#NIL, b#NIL
                        IF compareProc[a.first, b.first] = less THEN {                  --[2.2]
                            mergeTo.rest ← a;  mergeTo ← a;
                            IF (a ← a.rest) = NIL THEN { mergeTo.rest ← b; EXIT }}
                        ELSE {
                            mergeTo.rest ← b;  mergeTo ← b;
                            IF (b ← b.rest) = NIL THEN { mergeTo.rest ← a; EXIT }}
                        ENDLOOP;
                    a ← mergeToCons.rest }
                ENDLOOP;
            ENDLOOP;
        -- if list's length was even, a = NIL; if list's length was odd, a = single element list.  Merge a
          and elements of sorted into result (held in a).
        { j: index ← 0;
        UNTIL a # NIL DO
            a ← sorted[j];
            IF j < LAST[index] THEN j ← j+1 ELSE RETURN[a];
            ENDLOOP;
        DO --assert a#NIL
            IF (b ← sorted[j]) # NIL THEN {
                mergeTo ← mergeToCons;
                DO -- assert a#NIL AND b#NIL
                    IF compareProc[a.first, b.first] = less THEN {
                        mergeTo.rest ← a;  mergeTo ← a;
                        IF (a ← a.rest) = NIL THEN { mergeTo.rest ← b; EXIT } }
                    ELSE {
                        mergeTo.rest ← b;  mergeTo ← b;
                        IF (b ← b.rest) = NIL THEN { mergeTo.rest ← a; EXIT } }
                    ENDLOOP;
                a ← mergeToCons.rest };
            IF j < LAST[index] THEN j ← j+1 ELSE RETURN[a];
            ENDLOOP;
        }};--Sort

CompareError: PUBLIC ERROR[reason: ListSortRef.Reason] = CODE;

Compare: PUBLIC CompareProc --[r1, r2: REF ANY] RETURNS [Comparison] --  = TRUSTED {
    ENABLE SafeStorage.NarrowRefFault => GOTO BadType;
    IF r1 = NIL THEN {
        IF r2 = NIL THEN RETURN [equal]       -- define NIL=NIL regardless of type
        ELSE { temp: REF ANY ← r1;  r1 ← r2;  r2 ← temp }};
    WITH r1 SELECT FROM     -- assert r1#NIL                                   (note 2.8)
        rope: ROPE => RETURN[rope.Compare[s2: NARROW[r2, ROPE], case: TRUE]];
        ri: REF INT => RETURN[CompareINT[ri↑, NARROW[r2, REF INT]↑]];             --[2.4]
        ENDCASE => ERROR CompareError[invalidType]
    EXITS
        BadType => ERROR CompareError[typeMismatch]
    };

CompareINT: PROC [int1, int2: INT] RETURNS [Comparison] = INLINE {
    RETURN [
        SELECT TRUE FROM
            int1 < int2 => less,
            int1 = int2 => equal,
            ENDCASE => greater ] };

END.--OnlineMergeSortRefImpl
```

CHANGE LOG

Created by MBrown on 21-Apr-81 13:26:10

Changed by MBrown on 19-Aug-81 15:14:34
-- *CedarString -> Rope (used only in Compare.)*

Changed by MBrown on 23-Aug-81 17:45:12
-- *Fix bug in sorting NIL.*

Changed by MBrown on 10-Dec-81 9:57:58
-- *Cosmetic changes: ROPE, INT.*

Changed by MBrown on March 9, 1982 5:03 pm
-- *Use a bounded array in local frame instead of consing up a list of lists on each call. Even for a 32 bit address space, this is only 60 words of array in the frame.*

Changed by MBrown on June 28, 1982 11:43 am
-- *CEDAR implementation. CompareINT no longer uses MACHINE CODE.*

Changed by MBrown on July 5, 1982 3:58 pm
-- *Eliminate CARDINALs, redundant range check on j, type Item.*
Edited on December 20, 1982 3:20 pm, by Mitchell
changes to: CompareError *added generation of CompareError to Compare to map SafeStorage.NarrowRefFault and to replace unnamed error when an invalid type is presented.*

2.5. Notes for OnlineMergeSortRefImpl

(2.3) To be a general-purpose package, this progam must work properly in the presence of multiple processes. This means that any global state must be properly protected by monitors. The writer of this package chose to eliminate all global state from the program, so that no monitor is required. The cost is that one CONS is done on each call (for the "merge-to" list head) [line 2.1].

Clearly the package will get fouled up if asked to sort the same list by two concurrent processes.

(2.4) Since *Sort* is a PUBLIC procedure and has the same name as one of the procedures of the EXPORTed *ListSortRef* interface, the compiler will check that its type matches the one in *ListSortRef.* Similarly when a client program imports *ListSortRef*, the compiler will check that all its uses of *ListSortRef.Sort* are type-correct by comparing them against the interface. In this way, the client and the implementing modules are known to be in agreement, *assuming that they were both compiled using the* same *ListSortRef.*

To ensure that they are in agreement; the compiler places a unique identifier (UID) on each module that it compiles. When the client is bound to the implemented procedure (e.g., when they are loaded), the UID that *ListSortRef* had when the client was compiled is compared against the UID it had when the implementing module was compiled. They must agree for the binding to be allowed, otherwise there is a *version mismatch.*

(2.5) The list head, *mergeToCons*, simplifies the code a bit. Without it, the first comparison in a merge [line 2.2] would have to be treated specially. Note also that *mergeToCons* is initialized to refer to a new empty list cell every time *Sort* is called.

(2.6) The array *sorted* in the local frame is long enough to hold any list of REFs that will fit in the Cedar virtual memory. By keeping this array in the local frame we save both the cost of another allocation and the cost of ref-counting the assignments to array elements.

(2.7) The client's *compareProc* is responsible for narrowing its two REF ANY arguments to specific types. Using a client-supplied procedure and one or more REF ANY parameters is a standard way of introducing some polymorphism into Cedar programs. The costs associated with doing business this way are extra procedure calls and run-time discrimination of REF ANY arguments in client-supplied procedures (see, for example, (note 2.8)).

When these costs are deemed too high, one can get some amount of compile-time polymorphism to avoid them. Nothing is free, however, and achieving this run-time performance requires recompiling both the interface and the implementation modules for each application. There is a version of this sorting package that does just that: see the description of *OnlineMergeSort* in the Cedar Catalog.

(2.8) *Compare* provides clients with a procedure to pass to *Sort* for comparing ROPEs or INTs. It is a good model for writing your own comparison routine.

Since *Compare* is passed two REF ANY arguments, it must perform run-time type discrimination to determine that it can cope with them. The standard way of doing this is with a *discriminating select* statement and the NARROW operation.

One of the three arms will be executed, depending on the type of value that *r1* refers to (note that *r1* cannot be NIL at this point because of the preceding code).

If *r1* is a ROPE (which is actually a REF to a data structure describing the ROPE), then the first arm will be selected. In that arm, *rope* will be an alias for *r1*. If *r1* is a REF INT, the seond arm will be selected, and *ri* will be an alias for *r1*. Otherwise, the ENDCASE arm will be selected and the ERROR *CompareError*[*invalidType*] will be raised.

Inside the *rope*: ROPE arm, *Compare* simply returns the value returned by *Rope.Compare.* Since *Rope.Compare* expects parameters *s1* and *s2* to be ROPEs, however, NARROW is used to cast *r2* as a ROPE. If *r2* were not a ROPE, NARROW would generate the error *SafeStorage.NarrowRefFault*, which is caught by the encompassing ENABLE clause and then mapped into *CompareError*[*typeMismatch*] by the procedure's EXITS clause.

3. A sample tool using Viewers

This program illustrates how to build tools in the CedarViewers' world. The user interface it creates uses both existing classes of viewers as well as a new class of viewer for special effects. It contains a number of sections, each of which demonstrates a particular set of techniques. The best use of this code would be to copy interesting sections as models for creating your own tool.

It would be a good idea to try the SampleTool before looking at this example. When you type "Run SampleTool" to the UserExec, the SampleTool will appear in iconic form on the lower left side of the Cedar display. Open it by selecting it with the YELLOW (middle) mouse button and then try it out to see how it behaves.

The first part of the SampleTool viewer consists of the standard line of menu buttons at the top of any viewer with a few of the operations deleted and a new one, "MyMenuEntry", added.

The second part holds a "factorial computer", which allows you to compute N! by changing the value of N in the input part of the viewer and then selecting the button labelled "Compute Factorial" with the RED (left) mouse button.

The third part of the viewer presents a horizontal, logarithmic bar graph that dynamically updates itself and shows "Words Allocated Per Second" in the Cedar system.

3.1. SampleTool.mesa — Scott McGregor

```
-- SampleTool.mesa; Written by Scott McGregor on June 9, 1982 9:51 am
-- Last edited by Mitchell on August 16, 1982 4:16 pm
-- Last edited by McGregor on October 4, 1982 11:01 am

DIRECTORY
   Buttons USING [Button, ButtonProc, Create],
   Containers USING [ChildXBound, Container, Create],
   Convert USING [IntFromRope, ValueToRope],
   Graphics USING [Context, DrawBox, SetStipple],
   Labels USING [Create, Label, Set],
   Menus USING [AppendMenuEntry, CreateEntry, CreateMenu, Menu, MenuProc],
   MessageWindow USING [Append, Blink],
   Process USING [Detach, Pause, SecondsToTicks, Ticks],
   Rope USING [Cat, ROPE, Length],
   Rules USING [Create, Rule],
   SafeStorage USING [NarrowFault, NWordsAllocated],
   ShowTime USING [GetMark, Microseconds],
   UserExec USING [CommandProc, GetExecHandle, RegisterCommand],
   VFonts USING [CharWidth, StringWidth],
   ViewerClasses USING [PaintProc, Viewer, ViewerClass, ViewerClassRec],
   ViewerOps USING [CreateViewer, PaintViewer, RegisterViewerClass, SetOpenHeight],
   ViewerTools USING [MakeNewTextViewer, GetContents, SetSelection];

SampleTool: CEDAR PROGRAM                                              --(note 3.0)
   IMPORTS Buttons, Containers, Convert, Graphics, Labels, Menus, MessageWindow, Process,
     Rope, Rules, SafeStorage, ShowTime, UserExec, VFonts, ViewerOps, ViewerTools =

BEGIN
-- The Containers interface is used to create an outer envelope or "container" for the different sections
  below. For uniformity, we define some standard distances between entries in the tool.
```

```
entryHeight: CARDINAL = 15;  -- how tall to make each line of items
entryVSpace: CARDINAL = 8;  -- vertical leading space between lines
entryHSpace: CARDINAL = 10;     -- horizontal space between items in a line

Handle: TYPE = REF SampleToolRec;             -- a REF to the data for a particular instance of the
          sample tool; multiple instances can be created.
SampleToolRec: TYPE = RECORD [   -- the data for a particular tool instance
   outer: Containers.Container ← NIL,    -- handle for the enclosing container
   height: CARDINAL ← 0,        -- height measured from the top of the container
   fact: FactorialViewer,        -- the factorial viewer's state
   graph: GraphViewer ];         -- the bar graph viewer's state

MakeSampleTool: UserExec.CommandProc = TRUSTED BEGIN
   my: Handle ← NEW[SampleToolRec];
   myMenu: Menus.Menu ← Menus.CreateMenu[];
   Menus.AppendMenuEntry[  -- add our command to the menu
      menu: myMenu,
      entry: Menus.CreateEntry[
         name: "MyMenuEntry",        -- name of the command
         proc: MyMenuProc            -- proc associated with command
         ]
      ];
   my.outer ← Containers.Create[[-- construct the outer container              (note 3.1)
      name: "Sample Tool",   -- name displayed in the caption
      iconic: TRUE,          -- so tool will be iconic (small) when first created
      column: left,            -- initially in the left column
      menu: myMenu,        -- displaying our menu command
      scrollable: FALSE ]];     -- inhibit user from scrolling contents
   MakeFactorial[my];          -- build each (sub)viewer in turn
   MakeGraph[my];
   ViewerOps.SetOpenHeight[my.outer, my.height];    -- hint our desired height
   ViewerOps.PaintViewer[my.outer, all];              -- reflect above change
   END;

MyMenuProc: Menus.MenuProc = TRUSTED BEGIN
this procedure is called whenever the user left-clicks the entry labelled "MyMenuEntry" in the tool
 menu.
   MessageWindow.Append[
      message: "You just invoked the sample menu item with the ",
      clearFirst: TRUE];
   IF control THEN MessageWindow.Append[
      message: "Control-",
      clearFirst: FALSE];
   IF shift THEN MessageWindow.Append[
      message: "Shift-",
      clearFirst: FALSE];
   MessageWindow.Append[
      message: SELECT mouseButton FROM
         red      => "Red",
         yellow   => "Yellow",
         ENDCASE   => "Blue",
      clearFirst: FALSE];
   MessageWindow.Append[message: " mouse button.", clearFirst: FALSE];
```

```
    MessageWindow.Blink[ ];
    END;

FactorialViewer: TYPE = RECORD [
    input: ViewerClasses.Viewer ← NIL,    -- the Text Box for user input
    result: Labels.Label ← NIL ];    -- result of the computation

MakeFactorial: PROC [handle: Handle] = TRUSTED BEGIN                --(note 3.2)
    promptButton, computeButton: Buttons.Button;
    initialData: Rope.ROPE = "5";
    initialResult: Rope.ROPE = "120";
    handle.height ← handle.height + entryVSpace;  -- space down from the top of the viewer
    promptButton ← Buttons.Create[
        info: [
            name: "Type a number:",
            wy: handle.height,
            -- default the width so that it will be computed for us --
            wh: entryHeight, -- specify rather than defaulting so line is uniform
            parent: handle.outer,
            border: FALSE ],
        proc: Prompt,
        clientData: handle];  -- this will be passed to our button proc
    handle.fact.input ← ViewerTools.MakeNewTextViewer[ [
        parent: handle.outer,
        wx: promptButton.wx + promptButton.ww + entryHSpace,
        wy: handle.height+2,
        ww: 5*VFonts.CharWidth['0],  -- five digits worth of width
        wh: entryHeight,
        data: initialData,  -- initial contents
        scrollable: FALSE,
        border: FALSE]];
    computeButton ← Buttons.Create[
        info: [
            name: "Compute Factorial",
            wx: handle.fact.input.wx + handle.fact.input.ww + entryHSpace,
            wy: handle.height,
            ww:,    -- default the width so that it will be computed for us        (note 3.3)
            wh: entryHeight,  -- specify rather than defaulting so line is uniform
            parent: handle.outer,
            border: TRUE],
        clientData: handle,   -- this will be passed to our button proc
        proc: ComputeFactorial];
    handle.fact.result ← Labels.Create[ [
        name: initialResult,  -- initial contents
        wx: computeButton.wx + computeButton.ww + entryHSpace,
        wy: handle.height,
        ww: 20*VFonts.CharWidth['0],  -- 20 digits worth of width
        wh: entryHeight,
        parent: handle.outer,
        border: FALSE]];
    handle.height ← handle.height + entryHeight + entryVSpace;    -- interline spacing
    END;
```

```
Prompt: Buttons.ButtonProc = TRUSTED BEGIN
-- force the selection into the user input field
   handle: Handle ← NARROW[clientData];  -- get our data
   ViewerTools.SetSelection[handle.fact.input];      -- force the selection
   END;

ComputeFactorial: Buttons.ButtonProc = TRUSTED BEGIN
   handle: Handle ← NARROW[clientData];  -- get our data
   contents: Rope.ROPE ← ViewerTools.GetContents[handle.fact.input];
   inputNumber: INT;
   resultNumber: REAL ← 1.0;
   IF Rope.Length[contents]=0 THEN inputNumber ← 0
   ELSE inputNumber ← Convert.IntFromRope[contents
    ! SafeStorage.NarrowFault => {inputNumber←-1; CONTINUE}];                --(note 3.4)
   IF inputNumber NOT IN [0..34] THEN {
      MessageWindow.Append[
         message: "I can't compute factorial for that input",
         clearFirst: TRUE ];
      MessageWindow.Blink[ ] }
   ELSE FOR n: INT IN [2..inputNumber] DO
      resultNumber ← resultNumber*n;
      ENDLOOP;
   Labels.Set[handle.fact.result, Convert.ValueToRope[[real[resultNumber]]]];
   END;

-- the bar graph reflecting collector activity
GraphViewer: TYPE = RECORD [viewer: ViewerClasses.Viewer ← NIL];

MakeGraph: PROC [handle: Handle] = TRUSTED BEGIN                 --(note 3.6)
   xIncr: INTEGER;       -- temporarily used for labelling graph below
   xTab: INTEGER = 10;
   label: Rope.ROPE ← "1";          -- used to place labels on the graph
   rule: Rules.Rule ← Rules.Create[ [ -- create a bar to separate sections 1 and 2
      parent: handle.outer,
      wy: handle.height,
      ww: handle.outer.cw,
      wh: 2]];

   Containers.ChildXBound[handle.outer, rule];    -- constrain rule to be width of parent
   handle.height ← handle.height + entryVSpace;  -- spacing after rule
   [ ] ← Labels.Create[[
      name: "Words Allocated Per Second", parent: handle.outer,
      wx: xTab, wy: handle.height, border: FALSE ]];
   handle.height ← handle.height + entryHeight + 2; -- interline spacing
   handle.graph.viewer ← CreateBarGraph[
      parent: handle.outer,
      x: xTab, y: handle.height, w: 550, h: entryHeight,
      fullScale: 5.0 ];  -- orders of magnitude
   handle.height ← handle.height + entryHeight + 2; -- interline spacing
   xIncr ← handle.graph.viewer.ww/5;    -- so we can space labels at equal fifths
   FOR i: INTEGER IN [0..5) DO    -- place the labels, 1, 10, 100, 1000, 10000 along the graph
      [ ] ← Labels.Create[[name: label, parent: handle.outer,
         wx: xTab+i*xIncr - VFonts.StringWidth[label]/2,
         wy: handle.height, border: FALSE ]];
```

```
        label ← label.Cat["0"];              -- concatenate another zero each time
        ENDLOOP;
    handle.height ← handle.height + entryHeight + entryVSpace;   -- extra space at end
    TRUSTED {Process.Detach[FORK MeasureProcess[handle]]}; -- start the update process
    END;

MeasureProcess: PROC [handle: Handle] = TRUSTED BEGIN                    --(note 3.7)
-- Forked as a separate process. Updates the bar graph at periodic intervals.
    updateInterval: Process.Ticks = Process.SecondsToTicks[1];
    mark, nextMark: ShowTime.Microseconds;
    words, nextWords, deltaWords, deltaTime: REAL;
    mark ← ShowTime.GetMark[ ];
    words ← SafeStorage.NWordsAllocated[ ];
    UNTIL handle.graph.viewer.destroyed DO
        nextMark ← ShowTime.GetMark[ ];
        deltaTime ← (nextMark - mark) * 1.0E-6;
        nextWords ← SafeStorage.NWordsAllocated[ ];
        deltaWords ← nextWords - words;
        SetBarGraphValue[handle.graph.viewer, deltaWords/deltaTime];
        words ← nextWords;
        mark ← nextMark;
        Process.Pause[updateInterval];
        ENDLOOP;
    END;

-- this section creates the viewer class for a logarithmic bar graph.
-- private data structure for instances of BarChart viewers
GraphData: TYPE = REF GraphDataRec;
GraphDataRec: TYPE = RECORD [
    value: REAL ← 0, -- current value being displayed (normalized)
    scale: REAL ];      -- "full scale"

PaintGraph: ViewerClasses.PaintProc = TRUSTED BEGIN                      --(note 3.8)
    myGray: CARDINAL = 122645B;   -- every other bit
    data: GraphData ← NARROW[self.data];
    Graphics.SetStipple[context, myGray];
    Graphics.DrawBox[context, [0, 0, data.value, self.ch]];
    END;

CreateBarGraph: PROC [x, y, w, h: INTEGER, parent: ViewerClasses.Viewer,
                     fullScale: REAL]
RETURNS [barGraph: ViewerClasses.Viewer] = TRUSTED BEGIN                  --(note 3.9)
    instanceData: GraphData ← NEW[GraphDataRec];
    instanceData.scale ← fullScale;
    barGraph ← ViewerOps.CreateViewer[
        flavor: $BarGraph,  -- the class of viewer registered in the start code below
        info: [
            parent: parent,
            wx: x, wy: y, ww: w, wh:h,
            data: instanceData,
            scrollable: FALSE]
        ];
    END;
```

```
-- use this routine to set the bar graph to new values
SetBarGraphValue: PROC [barGraph: ViewerClasses.Viewer, newValue: REAL] = BEGIN
   my: GraphData ← NARROW[barGraph.data];

   Log10: --Fast-- PROC [x: REAL] RETURNS [lx: REAL] = BEGIN
   -- truncated for values of [1..inf), 3-4 good digits
   -- algorithm from Abramowitz: Handbook of Math Functions, p. 68
      sqrt10: REAL = 3.162278;
      t: REAL;
      lx ← 0;
      WHILE x > 10 DO x ← x/10; lx ← lx+1 ENDLOOP;   -- scale to [1..10]
      IF x > sqrt10 THEN {x ← x/sqrt10; lx ← lx+0.5};   -- scale to [1..1/sqrt10]
      t ← (x-1)/(x+1);
      lx ← lx + 0.86304*t + 0.36415*(t*t*t)   -- magic cubic approximation
      END;

   my.value ← Log10[1+newValue] * barGraph.cw / my.scale;
   TRUSTED {ViewerOps.PaintViewer[viewer: barGraph, hint: client, clearClient: TRUE]};
   END;

-- graphClass is a record containing the procedures and data common to all BarGraph viewer
         instances (class record).
graphClass: ViewerClasses.ViewerClass ←                                          --[3.1]
   NEW[ViewerClasses.ViewerClassRec ← [paint: PaintGraph]];
-- Register a command with the UserExec that will create an instance of this tool.
UserExec.RegisterCommand[name: "SampleTool", proc: MakeSampleTool,
   briefDoc: "Create a sample viewers tool" ];
-- Register the BarGraph class of viewer with the Window Manager
TRUSTED {ViewerOps.RegisterViewerClass[$BarGraph, graphClass]};
[ ] ← MakeSampleTool[UserExec.GetExecHandle[ ]];   -- and create an instance
END.
```

3.2 Notes for SampleTool

(3.0) [hopefully temporary note] To make this module a valid CEDAR PROGRAM, many sections had to be declared TRUSTED regions because they use other, not yet SAFE, interfaces. Most of these regions are entire procedure bodies, but in some cases it has been possible to narrow the TRUSTED region to a smaller scope.

(3.1) Many of the Viewer procedures take a fair number of arguments in order to provide a number of options. Many of these can be defaulted, as in this section of the program. Using keyword notation makes these calls significantly more clear about which parameters are being specified as well as making them independent of the order specified for the parameters.

(3.2) The *factorial* sub-Viewer contains a "Text Box" for the user to enter a number, and a button that computes the factorial of the number in the text box when pushed and writes it in a result field. The fields for the user's input and the result are stored in *handle.fact.*

Note that the name of *MakeFactorial*'s parameter, *handle*, and its type, *Handle*, differ only in the case of the "h". In general, having names that differ only in case shifts of letters is a *very bad idea.* However, this one exception is so compelling and so common that it has become part of the Cedar stylistic conventions: If there is only one variable of a type in a scope, its name can be the same as that of its type except that it should begin with a lowercase letter. Think of it as equivalent to the definite article in English ("the Handle").

(3.3) *Buttons.Create* will compute the width from the button's name, so we can default this argument to the procedure. Specifying the keyword parameter name *w* without a value indicates that it should get whatever default value is specified in the definition of *Buttons.Create.*

(3.4) *Convert.IntFromRope* lets the error *SafeStorage.NarrowFault* escape from it (stylistically, it really ought to map that error into one defined in its interface). *ComputeFactorial* catches this error and assigns -1 to *inputNumber* to trigger the subsequent test for its being in a suitable range and so give the error message "I can't compute factorial for that input" in the message window.

(3.5) Atoms provide virtual-memory-wide unique identifiers like $BlackOnGrey (all atom constants are denoted by a leading "$" followed by an identifier).

(3.6) The "bar graph" sub-Viewer contains a logarithmic bar graph depicting the number of words per second currently being allocated via the Cedar counted storage allocator. Unlike the factorial sub-Viewer, which only makes use of pre-defined viewer classes, this section defines a new viewer class, $BarGraph [line 3.1], to display information and makes a viewer which is an instance of the class.

(3.7) *MeasureProcess* runs as an independent process, looping as long as the viewer denoted by *handle* continues to exist. It computes the average number of words allocated per second in Cedar safe storage and then sleeps for a short time before repeating the cycle.

(3.8) *PaintGraph* is called whenever the bar graph contents are to be redisplayed on the screen. It will always be called by the window manager, which will pass it a *Graphics.Context*, correctly scaled, rotated and clipped for the viewer on the screen. It is passed to the Viewers machinery when the $BarGraph class is created [line 3.1].

(3.9) *CreateBarGraph* is called to create a new BarGraph viewer. It creates the private data for the new instance and then calls a Viewers' system routine to create the actual viewer.

4. An evaluator for FUN, a functional programming language

This standalone program provides interactive entering and evaluation of expressions in a small, functional programming language, FUN, based on Landin's ISWIM language. The language is defined by the following BNF rules:

```
    Prog ::= Exp0 "#"
    Exp0 ::= "let" Id "=" Exp1 "in" Exp0 | Exp1
    Exp1 ::= "lambda" Id "." Exp1 | Exp2
    Exp2 ::= Exp2 Exp3 | Exp3
    Exp3 ::= Id | "(" Exp0 ")"
```

An example FUN program is

```
    let Twice = lambda f. lambda y. f(f y) in
            let ApplyThyself = lambda z . z z in
                    (ApplyThyself Twice Twice) (PLUS 1) 2
  #↓
```

Twice is a function that produces a function which applies Twice's parameter *f* twice to whatever comes next (*y*). ApplyThyself applies its parameter *z* to itself. The result of this program is 18 because (ApplyThyself Twice) = (Twice Twice) = FourTimes, (FourTimes Twice) = SixteenTimes, and SixteenTimes plus 2 is 18.

4.1 FUN.mesa — Jim Morris

```
-- FUN.mesa
-- Last edited by Mitchell on December 16, 1982 2:50 pm
-- Last edited by Jim Morris, May 12, 1982 10:28 am
DIRECTORY
   IO USING [BreakAction, BreakProc, CR, CreateViewerStreams, EndOf, Error, GetInt, GetToken,
     Handle, int, PutF, PutFR, refAny, rope, RIS, SP, TAB],
   Process USING [Detach],
   Rope USING [Digit, Equal, Fetch, Letter, ROPE],
   SafeStorage USING [NarrowRefFault, NewZone];
FUN: CEDAR PROGRAM
   IMPORTS IO, Process, Rope, SafeStorage =
BEGIN
ROPE: TYPE = Rope.ROPE;
z: ZONE = SafeStorage.NewZone[ ];                                  --(note 4.1)

Exp: TYPE = REF ANY;-- always a REF to one of the following               (note 4.2)
   Variable: TYPE = ROPE;
   Application: TYPE = REF ApplicationR;
      ApplicationR: TYPE = RECORD[rator, rand: Exp];
   Lambda: TYPE = REF LambdaR;
      LambdaR: TYPE = RECORD[bv: Variable, body: Exp];
   Closure: TYPE = REF ClosureR;
      ClosureR: TYPE = RECORD[exp: Exp, env: Environment];
   Primitive: TYPE = REF PrimitiveR;
      PrimitiveR: TYPE = RECORD[p: PROC[Exp, Exp] RETURNS [Exp] , state: Exp];
   Environment: TYPE = LIST OF RECORD[var: Variable, val: Exp];

din, dout: IO.Handle;

FUNError: ERROR = CODE;                                      --(note 4.3)
```

```
NoteError: PROC [msg: ROPE] = BEGIN
-- this procedure always generates the error FUNError.
   dout.PutF["\n %g \n\n", IO.rope[msg]];
   ERROR FUNError;
   END;

-- The FUN lexical analysis and parsing machinery
cToken: ROPE;                                                       --(note 4.4)

Next: PROC = {
   cToken ← IF din.EndOf[ ] THEN "#" ELSE din.GetToken[StTokenProc]};

StTokenProc:  IO.BreakProc = BEGIN                                          --(note 4.5)
   OPEN IO;                                                             --(note 4.6)
   RETURN[
      IF Rope.Letter[c] OR Rope.Digit[c] THEN KeepGoing
      ELSE IF c=SP OR c=CR OR c=TAB
         THEN StopAndTossChar ELSE StopAndIncludeChar];
   END;

Id: PROC RETURNS [i: Variable] = BEGIN
   IF IsId[cToken] THEN {i ← cToken; Next[ ]}
   ELSE NoteError["Input Error: No Id"]
   END;

IsId: PROC[x: ROPE] RETURNS [BOOLEAN] = BEGIN
   RETURN[Rope.Digit[x.Fetch[0]] OR Rope.Letter[x.Fetch[0]]
    AND NOT Rope.Equal[x, "let"]
    AND NOT Rope.Equal[x, "in"]
    AND NOT Rope.Equal[x, "lambda"]];
   END;

Prog: PROC RETURNS [e: Exp] = BEGIN                                 --(note 4.7)
   e ← Exp0[ ];
   IF Rope.Equal[cToken, "#"] THEN RETURN;
   UNTIL Rope.Equal[cToken, "#"] DO Next[ ] ENDLOOP;
   NoteError["Input Error: Whole Expression not consumed"];
   END;

Exp0: PROC RETURNS [Exp] = BEGIN
   IF Rope.Equal[cToken, "let"] THEN {
      Next[ ];
         {v: Variable = Id[ ];
         IF NOT Rope.Equal[cToken, "="] THEN NoteError["Input Error: No ="];
         Next[ ];
            {val: Exp = Exp1[ ];
            IF NOT Rope.Equal[cToken, "in"] THEN NoteError["Input Error: No in"];
            Next[ ];
            RETURN[
               z.NEW[ApplicationR ← [z.NEW[LambdaR ← [v,.Exp0[ ]]], val]]] } } };
   RETURN[Exp1[ ]];
   END;

Exp1: PROC RETURNS [Exp] = BEGIN
   IF Rope.Equal[cToken, "lambda"] THEN
      {Next[ ];
```

```
            {i: Variable = Id[ ];
            IF NOT Rope.Equal[cToken, "."] THEN NoteError["Input Error: No ."];
            Next[ ];
            RETURN[z.NEW[LambdaR ← [i, Exp1[ ]]]] } };
    RETURN[Exp2[ ]];
    END;

Exp2: PROC RETURNS [e: Exp] = BEGIN
    e ← Exp3[ ];
    WHILE Rope.Equal[cToken, "("] OR IsId[cToken] DO
        e ← z.NEW[ApplicationR ← [e, Exp3[ ]]];                         --[4.0]
        ENDLOOP;
    END;

Exp3: PROC RETURNS [e: Exp] = BEGIN
    IF Rope.Equal[cToken, "("] THEN {
        Next[ ]; e ← Exp0[ ];
        IF NOT Rope.Equal[cToken, ")"] THEN NoteError["Input Error: No )"];
        Next[ ]}
    ELSE e ← Id[ ];
    END;

-- The FUN interpreter

Eval: PROC[x: Exp, env: Environment] RETURNS [Exp]= BEGIN               --(note 4.8)
    DO
        WITH x SELECT FROM
            v: Variable => {t: Environment ← env;
                UNTIL t=NIL OR Rope.Equal[v, t.first.var] DO t ← t.rest ENDLOOP;
                RETURN[IF t=NIL THEN x ELSE t.first.val]};
            p: Primitive => RETURN[x];
            l: Lambda => RETURN[z.NEW[ClosureR ← [l, env]]];
            a: Application =>
                {rator: Exp = Eval[a.rator, env];
                rand: Exp = Eval[a.rand, env];
                WITH rator SELECT FROM
                    f: Closure => {
                        l: Lambda = NARROW[f.exp
                            ! SafeStorage.NarrowRefFault =>
                              NoteError["Evaluation Error: Illegal application"] ];     --[4.1]
                        x ← l.body; env ← CONS[[l.bv, rand], f.env] };
                    prim: Primitive => RETURN[prim.p[prim.state, rand]];
                    ENDCASE => NoteError["Evaluation Error: Illegal application"]};      --[4.2]
            f: Closure => RETURN[x];
            ENDCASE
        ENDLOOP;
    END;

Plus: PROC[d, first: Exp] RETURNS [Exp] =                               --(note 4.9)
    {RETURN[z.NEW[PrimitiveR ← [Plus1, first]]]};

Plus1: PROC[first, second: Exp] RETURNS[v: Variable] = BEGIN
    ENABLE IO.Error => NoteError["Evaluation Error: Not a number"];          --[4.3]
    a: INT = IO.GetInt[IO.RIS[NARROW[first, ROPE]]];                        --(note 4.10)
    b: INT = IO.GetInt[IO RIS[NARROW[second, ROPE]]];
```

```
   RETURN[IO.PutFR[, IO.int[a+b]]];
   END;

EvalLoop: PROC = BEGIN
   DO
      ENABLE {
         FUNError => LOOP;                                              --(note 4.11)
         IO.Error => EXIT};                                            --(note 4.12)
      result: Exp ← NIL;
      Next[ ];
      result ← Eval[Prog[ ], LIST[["PLUS", z.NEW[PrimitiveR ← [Plus, NIL]]]]]; --(note 4.13)
      dout.PutF["\nResult is %g\n\n",IO.refAny[result]];            --(note 4.14)
      ENDLOOP
   END;

[din, dout] ← IO.CreateViewerStreams["Fun"];
TRUSTED {Process.Detach[FORK EvalLoop[ ]]};
END.
```

4.2 Notes for FUN

(4.1) This declares *z* to be a storage zone, and initializes it by creating a new zone.

Throughout the program you will see expressions of the form "z.NEW[...]", which allocate data structures using storage taken from this zone. It is not required that we create a zone to use for allocation, but experience has shown that performance is better if each application takes its storage from its own zone rather than the public, default zone one gets by saying simply "NEW[...]".

Some commonly used packages provide zones in which to allocate the objects they manage. For example, the zone *List.ListZone* can be used to make a list element, e.g., *List.ListZone.*CONS[NIL, NIL]. Alternatively, the procedure *List.Cons* can be used with the same effect. Also, the *RefText* interface contains the procedure

```
RefText.New[nChars: NAT] RETURNS [REF TEXT],
```

which uses a zone provided by the package for allocating the REF TEXT rather than the default system zone.

(4.2) This sequence of declarations defines the representation for FUN expressions. An expression is parsed into an abstract tree form which can then be interpreted by the Eval procedure.

An expression (Exp) can refer to one of five different node types:

A Variable, which is represented as a ROPE containing its name;

An Application, which consists of an operator (rator) and an operand (rand);

A Lambda expression, which consists of a bound variable (bv) and a body;

A Closure, which is an expression (exp) together with an Environment, which is a LIST defining a value for each its free variables;

A Primitive, which is a Cedar procedure together with some state.

Notice that it is perfectly legal to store procedures in data structures. What is actually stored, however, is a pointer to the code. Since Cedar does not support procedures as values in full generality, such procedures cannot be local to other procedures; nor can they refer to their parents' local variables (because, for efficiency of procedure invocation, the Cedar garbage collector does not treat procedures' activation records as collectible storage).

We had to define a name for both the node record types and for references to them, e.g., ApplicationR and Application. The record type name is needed when calling the NEW procedure as in

```
z.NEW[ApplicationR ← [e, Exp3[ ]]]             --[4.0]
```

The reference types are used in nodes to provide the tree structure for a FUN expression

In declaring an Exp to be a REF ANY, we have told the compiler that it may be a reference to anything at all. This is the recommended way of achieving type variation in Cedar. (A version of Pascal-like variant records is also available, but should only be used when efficiency is crucial.)

(4.3) The procedure *NoteError* types an error message to the user and then generates the ERROR *FUNError* to clean up the call stack back to *EvalLoop*, where the error is caught and dealt with (note 4.11).

(4.4) The tokenizer and parser for FUN use IO for reading tokens from the input stream *din* using the StTokenProc to direct the tokenizing facilities provided by IO.

(4.5) The value returned by *StTokenProc* is an enumerated type from the IO interface.

(4.6) OPENing *IO* allows us to omit the *IO.* prefix from SP, CR, TAB, *KeepGoing*, *StopAndTossChar*, and *StopAndIncludeChar*. This is one of the few cases in which an unqualified OPEN should be used. Generally, programs are more easily read when it is obvious which identifiers are imported and which interfaces they come from. If you use OPEN over any but the smallest scope, it is recommended that you use the qualified form, e.g., "OPEN *Safe*: *SafeStorage*" over a limited scope, or just "*Safe*: *SafeStorage*" in the imports list if the scope is the entire module.

(4.7) *Prog*, *Exp0*, *Exp1*, *Exp2*, and *Exp3* constitute a simple recursive descent parser for FUN.

(4.8) *Eval*, *Plus*, *Plus1*, and *EvalLoop* constitute the FUN interpreter. *Eval* uses WITH-SELECT to discriminate among the various *Exp* node types and NARROW when there is only one possible type that an *Exp* could be [line 4.1]. If the NARROW fails, it will raise the signal *SafeStorage.NarrowRefFault*. If the user were to see .that error, it would be somewhat mystifying, so the program catches it and generates

NoteError["Evaluation Error: Not a lambda expression"] [line 4.2],

which at least says what is actually happening in terms of the FUN language. The same effect is achieved in *Plus1* [line 4.3] using an ENABLE clause in the body of the procedure.

(4.9) *Plus* is a "primitive-returning-primitive" that returns a function prepared to add *first* to whatever comes second.

(4.10) This is a standard way to convert a ROPE to an integer; create a stream from the rope using *RS* and pull (scan) an integer from it. A NARROW could fail if the user typed something like "(PLUS x 1) #↓". Should that happen, the signal *SafeStorage.NarrowRefFault* would be caught, an error message given to the user, and then the *FUNError* signal would be generated to unwind control cleanly out to *EvalLoop*.

(4.11) If a problem is encountered while parsing or evaluating a FUN expression, the error *FUNError* will be generated. *EvalLoop* catches it and simply continues around the main read-evaluate-print loop.

(4.12) When the user destroys the FUN viewer, its *din* and *dout* streams will be closed, and any pending call on *IO* routines will generate the error *IO.Error*[*StreamClosed*]. This error is caught so that *EvalLoop* can then gracefully exit and it process can then disappear.

(4.13) The second parameter of *Eval* is an environment mapping "PLUS" into a primitive.

(4.14) This *PutF* will print result, even if it is a paritally evaluated function or an environment. Try typing "(PLUS 1) #↓" to see this.

Release as [Indigo]<Cedar>Documentation>CedarStyle.tioga, press
Draft [Indigo]<Cedardocs>Manual>CedarStyle.tioga, press
By Jim Mitchell
Last edited by Mitchell on December 20, 1982 5:28 pm

Stylizing Cedar Programs

Version 3.5.2

XEROX — FOR INTERNAL USE ONLY

Abstract **styl·ize** *tr. v.* **-ized, -izing, -izes.**
1. To subordinate verisimilitude to principles of design in the representation of.
2. To represent conventionally; conventionalize.

This memo is probably out of date if it is in hardcopy form. It documents Release 3.5.2 of Cedar, December 1982.

[If you are reading this document on-line, try using the Tioga Levels and Lines menus (if you can) to initially browse the top few levels of its structure before reading it straight through.]

Introduction to Cedar: Contents

0. Introduction

styl•ize *tr. v.* **-ized, -izing, -izes. 1.** To subordinate verisimilitude to principles of design in the representation of. **2.** To represent conventionally; conventionalize.

Well, maybe we don't want to "subordinate verisimilitude to principles of design," but using some set of reasonable conventions in Cedar programs certainly does have value.

The conventions presented here have been generally agreed upon by the Cedar programming community and are approximations to current practice. Consequently, they cannot be hard and fast rules, and there can be good and compelling reasons for not following some convention in a particular instance. Nevertheless, to the extent that we all *usually* follow them, it will help us in reading (and therefore in modifying) one anothers' programs.

This section has two major parts: The first consists of a set of conventions for constructing Cedar identifiers according to their intended uses, for constructing types, for declaring and using SIGNALs and ERRORs, for documentation embedded in modules, for program layout (indentation, etc.), and for object-oriented programming. The second part consists of a set of skeletal programs that employ and exemplify these conventions.

In the first part, each convention is given as a short rule followed by some examples of its application. In the examples, the parts that illustrate the rule are **highlighted like this.**

1. Names

1.1. Capitalization

Capitalize the first letter of a name
if it identifies a module (interface or implementation), procedure, signal, or type, otherwise the first letter is lower-case:

```
Fugleman: DEFINITIONS = ...
Factorial: PROC[i: INT] RETURNS [INT];
Complex: TYPE = RECORD[real, imag: REAL];
NarrowRefFault: ERROR;
```

Capitalize the first letter of each embedded word
of a multi-word name:

```
NarrowRefFault: ERROR;
ComplexSet: TYPE = LIST OF Complex;
aSet: ComplexSet ← CONS[Complex[real: 1.0, imag: 0.0], NIL];
```

Case shift alone should not be used to distinguish identifiers.
There are some exceptions: if one has a type, *Foo*, it is okay to declare a variable of that type as *foo*; it is acceptable to name a condition variable and a BOOLEAN describing the condition similarly:

```
badID, badId, bADid, BADid: INT                      -- BAD!
complex: Complex;                           -- acceptable
c: Complex;                                          -- acceptable
```

1.2. Qualification

Identifiers from interfaces should be qualified
by their interface names or abbreviations for them:

```
DIRECTORY
    IO, Process, Rope, UserExec;
SimpleExample: CEDAR MONITOR
    IMPORTS IO, Process, Rope, UserExec =
    . . .
    ReverseName: UserExec.CommandProc = BEGIN . . . END;
    execStream: IO.Handle;
```

Renaming an interface in the IMPORTS clause
is okay, provided that you don't use both names in the program.

```
SimpleExample: CEDAR MONITOR
        IMPORTS Process, Rope, X: UserExec =
        . . .
        ReverseName: X.CommandProc = ..
```

The qualified/renaming form of OPEN should be used
instead of the unqualified form.

There are a few valid reasons for using an unqualified OPEN:
It is acceptable over the scope of an entire PROGRAM module only for the DEFINITIONS module that the module exports. It is also okay to OPEN any interface over a limited scope (e.g., a procedure or a block) within which identifiers from that interface are used heavily. It is never okay to unqualifiedly open anything except an interface.

1.3. Module Naming

All module source file names have the extension ".mesa".

```
ObjectSupport.mesa
ObjectSupportImpl.mesa
ListSortRef.mesa
SimpleExample.mesa
```

A DEFINITIONS module does not need any standard suffix on its name.

```
ObjectSupport: DEFINITIONS = . . .
ListSortRef: DEFINITIONS = . . .
Rope: DEFINITIONS = . . .
```

A PROGRAM/MONITOR module name should have the suffix "Impl"
if its name without the suffix would conflict with the name of a DEFINITIONS module.

```
ObjectSupportImpl: PROGRAM       -- name would conflict with ObjectSupport without Impl
    EXPORTS ObjectSupport = . . .
ObjectMachinery: PROGRAM -- name doesn't conflict with ObjectSupport
    EXPORTS ObjectSupport = . . .
Fun: PROGRAM =  -- name conflicts with nothing
```

A CONFIGURATION module does not need any standard suffix on its name.

```
Rigging: CONFIGURATION
   EXPORTS Rope = . . .
Spy: CONFIGURATION
   EXPORTS SpyOps = . . .
Compiler: CONFIGURATION
   EXPORTS ExecOps, CompilerOps = . . .
```

A CONFIGURATION file name should have the extension ".config".

```
Rigging.config
Spy.config
Compiler.config
```

2. Types

Name types

rather than defining anonymous types, especially in definitions modules:

```
DeckIndex: TYPE = [0..52);                             -- YES
CardDeck: TYPE = ARRAY DeckIndex OF Card;
CardDeck: TYPE = ARRAY [0..52) OF Card;                -- NO
```

Use closed lower bounds and open upper bounds

for interval types:

```
[0..upperLimit)                                        -- preferred form
FOR i IN [0..n) DO IF a[i]=a[n] THEN . . .  ENDLOOP;
```

Use positional notation for single-component argument lists,

record constructors, and extractors. Positional notation is also acceptable if there are multiple components, all of different types:

```
ViewerTools.SetSelection[handle.fact.input];           -- force the selection
execStream.PutF["Your user name backwards is: %g\n", IO.rope[backwordsName]];
```

Use the keyword form

for argument lists, record constructors, and extractors when there are two or more constituents, especially if any have equivalent types. It is also preferred when most components are being defaulted and only a few given values:

```
context.DrawBox[
   Box[xmin: 0, ymin: 0, xmax: data.value, ymax: self.ch]];
Menus.AppendMenuEntry[
   menu: my.outer.menu, -- the outer container's menu (already defaulted)
   name: "MyMenuEntry",          -- name of the command
   proc: MyMenuProc,                          -- proc associated with command
   fork: TRUE,                   -- causes a new process to be forked & detached on
            invocation
   copy: TRUE ];          -- make this a new menu (i.e. don't modify the default menu)
[in: in, out: out] ← IO.CreateTTYStreams[title];
R.Compare[s1: rope, s2: NARROW[r2, ROPE], case: TRUE]
NEW[ViewerClasses.ViewerClassRec ← [paint: PaintGraph]];
```

Name procedure parameters
in the definitions of procedure types in DEFINITIONS modules:

```
        CommandProc: TYPE = PROC [exec: ExecHandle, clientData: REF ANY ← NIL] RETURNS[ok:
                BOOLEAN ← TRUE];
```

Name procedure result fields,
especially if there is more than one:

```
        CreateBarGraph:
            PROC [x, y, w, h: INT, parent: ViewerClasses.Viewer, fullScale: REAL]
                RETURNS [barGraph: ViewerClasses.Viewer]
        FindItem: PROC[k: Key] RETURNS [v: Value, found: BOOLEAN];
```

3. Exceptions: SIGNALs and ERRORs

3.1. General

Use ENDCASE only to generate an ERROR or to treat "none of the above",
but not to handle specific cases.

Use SIGNAL or ERROR when generating a SIGNAL or ERROR;
e.g., write "SIGNAL foo" or "ERROR foo", not just "foo".

Avoid using the anonymous ERROR.
except for "impossible" conditions.

Locally defined signals shouldn't escape from an abstraction
and need not conform to the conventions below.

3.2. In DEFINITIONS modules

Declare ERRORs as ERRORs and SIGNALs as SIGNALs.
Don't declare a SIGNAL and then generate an ERROR with it.

Try to define a small number of ERRORs and SIGNALs in an interface.
Normally define a single ERROR/SIGNAL with a parameter to distinguish the exact reason for it. Only use separate names for separate signals when they must have different types for reasons of resumption:

```
        IOError: ERROR [ec: ErrorCode];
        ErrorCode: TYPE = {NotImplementedForThisStream, IllegalPutBack, SyntaxError,
                StreamClosed, FileNotFound, FileAlreadyExists, IllegalFileName,
                WrongTransactionType, FileTooLong, BadIndex};
        IOSignal: SIGNAL [ec: SignalCode];
        SignalCode: TYPE = {EmptyBuffer, UnmatchedLeftParen, UnmatchedStringDelim, Rubout,
                UnprintableValue, TypeMismatch, UnknownFormat};
        BufferOverFlow: SIGNAL[text: REF TEXT] RETURNS[REF TEXT];
                -- used by text streams, should return a bigger one, with characters copied over, or else
                can return same text with length reset after having done something with chartacters.
```

Indicate which SIGNALS/ERRORS are generated by a procedure
in an interface and what happens if a given SIGNAL is RESUMEd:

```
FindItem: PROC[k: Key] RETURNS [v: Value, found: BOOLEAN];
        -- ERRORS: HashTableDamaged
        -- SIGNALS: HashTableProblem[tableFull]: tries again if resumed
GetRefAny: PROC [stream: STREAM] RETURNS [REF ANY];    -- from IO
        -- SIGNALS: IOSignal[UnmatchedLeftParen, UnmatchedStringDelim]; resumption
        causes GetRefAny to supply the missing right parenthesis or quote.
```

3.3. In PROGRAM modules

Only exceptions that are part of the abstraction should emanate from a module.
Don't let exceptions that are "part of" the interfaces of invoked routines escape: handle them completely within your abstraction, or transform them into exceptions that are part of your interface (i.e., insulate your clients from details of your implementation).

```
... inputNumber ← Convert.IntFromRope[contents            -- from SampleTool
        ! SafeStorage.NarrowFault => {inputNumber←-1; CONTINUE}];
```

4. Programming constructs:

Ensure that the target type for a NARROW is obvious
either by naming it explicitly or using it in a simple context.

```
handle: Handle ← NARROW[clientData];            -- target type = Handle
a: INT = IO.GetInt[IO.RS[NARROW[first, ROPE]]];            -- target type = ROPE
```

Use TRUSTED over the most confining region in which it is needed.

```
TRUSTED {Process.Detach[FORK EvalLoop[ ]]};
```

5. Module histories:

The first two lines of a source module
should be comments, one giving the file name for the module, and the other giving the name of the person who last edited it and when she/he did so. If you use the conventions described in the Tioga chapter for these lines, Tioga can be made to update the last-edited date automatically when you save the file.

```
-- FILE: OnlineMergeSortRefImpl.mesa
-- Last Edited by MBrown on March 9, 1982 5:02 pm
```

Immediately following each procedure declaration in a definitions module
should be a brief comment, providing the information most useful to a potential client, who may be reading the listing, or querying via the User Exec.

```
Sort: PROC
      [itemList: LIST OF Item,
      compareProc: PROC[Item, Item] RETURNS [Comparison]]
   RETURNS[LIST OF Item];
   -- Destructive sort of itemList; returns sorted list containing same items.
   -- Order of equal items is not preserved.
```

Keep a CHANGE LOG
at the end of each source module for keeping track of who first created the module, who has changed it, why it was changed, and when. (see the description of the Tioga ChangeLog button in the Tioga chapter for a way of preparing change log entries semi-automatically).

```
CHANGE LOG
Created by MBrown on 21-Apr-81 13:26:10
Changed by MBrown on 19-Aug-81 15:14:34
    -- CedarString -> Rope (used only in Compare.)
Changed by MBrown on 23-Aug-81 17:45:12
    -- Fix bug in sorting NIL.
Changed by MBrown on 10-Dec-81  9:57:58
    -- Cosmetic changes: ROPE, INT.
Changed by MBrown on March 9, 1982 5:03 pm
    -- Use a bounded array in local frame instead of consing up a list of lists on each call.  Even
     for a 32 bit address space, this is only 60 words of array in the frame.
```

6. Program layout

Use Cedar.abbreviations
and see the description of the "Expand Abbreviation" command in the Tioga chapter.

7. The Fuglemen

This section consists of a set of skeletal programs, Fugleman, FuglemanImpl, and FuglemanClient, that use the conventions expounded in the above style discussion. The name Fugleman comes from the following definition:

fu·gle·man *n., pl.* **-men.**
1. *Archaic.* A soldier who serves as a guide and model for his company.
2. A leader; especially a religious leader.

And, Lord knows, there is no more religious issue than one's favorite programming style.

7.1. Fugleman.mesa

This is a template for defining the interface for a class, Fugleman, of objects. It provides for multiple, coexisting implementations of the class. The file FuglemanClient.mesa shows how a client program can create instances of the objects of this class, and the file FuglemanImpl.mesa is a template for an implementation of the class.

```
-- FILE:  Fugleman.mesa
-- Last Edited by Horning, June 7, 1982 2:55 pm

Fugleman: CEDAR DEFINITIONS =

BEGIN
Handle:  TYPE = REF Rep;
Rep: TYPE;                          -- opaque type for the representation of the object
FugleFailure: ERROR;

NewFugleman: PROC RETURNS [Handle];
            -- Makes a new Fugleman object

Print: PROC [h: Handle];

Check: PROC [h: Handle];
-- ERRORs: FugleFailure
-- Returns normally if Fugleman object OK

SomeProc: PROC [h: Handle, otherArgument: INT] RETURNS [INT];
-- Comment goes here saying what it does

END.

CHANGE LOG
   Created by Mitchell,  March 17, 1980  11:00 AM
      -- This is a template for interim Cedar object interfaces.  It shows what one would write to
        define the interface for a class FugleMan of objects with operations Release, Print, Check,
        and SomeProc
   Changed by Mitchell, June 4, 1982 3:34 pm
      -- Convert to present Cedar conventions
   Changed by Horning, June 7, 1982 2:55 pm
      -- Use templates from Mesa.abbreviations
```

7.2. FuglemanClient.mesa

This is an example intended only to show syntactically how client programs create object instances, invoke operations on them and catch signals that those operations might generate. The files Fugleman.mesa and FuglemanImpl.mesa contain the definitions of the object interface used here and a sample implementation of it, respectively.

```
FILE:  FuglemanClient.mesa
Last Edited by Mitchell, August 16, 1982 2:04 pm

DIRECTORY
            Fugleman USING [Handle, NewFugleman, Check, SomeProc, FugleFailure];

FuglemanClient: CEDAR PROGRAM
   IMPORTS Fugles: Fugleman =

BEGIN
TestError: ERROR;

SimpleTest: PROC = BEGIN
   a nonsense program to show how clients invoke operations on objects implemented this way
   fm: Fugles.Handle = Fugles.NewFugleman[];
   anotherFm: Fugles.Handle = Fugles.NewFugleman[];
   x: INT ← 1;
   fm.Check[ ! Fugles.FugleFailure => ERROR TestError];      -- check the object, convert error
   IF fm=anotherFm THEN ERROR TestError;  -- should get unique objects
   IF fm.SomeProc[1] = anotherFm.SomeProc[2] THEN x←2;
   END;

SimpleTest[ ];                          -- call the test procedure

END. -- FuglemanClient

CHANGE LOG
   Created by Mitchell, March 27, 1980 9:23 AM
      Produce exemplary template
      Changed by Horning: June 7, 1982 2:58 pm
         Cosmetic changes for conformance with current recommendations
   Changed by Horning: June 8, 1982 2:10 pm
      Insert CEDAR prefix
```

7.3. FuglemanImpl.mesa

This is a template for an implementation of the class of objects defined by Fugleman.mesa. There may be more than one implementation of this same class. An example of a client program can be found in FuglemanClient.mesa.

```
FILE: FuglemanImpl.mesa
Last Edited by Mitchell, August 16, 1982 2:01 pm

DIRECTORY
   Fugleman;

FuglemanStdImpl: CEDAR PROGRAM
   EXPORTS Fugleman =

BEGIN OPEN Fugleman;

Implementation types
Handle: TYPE = REF Rep; -- declare the concrete type locally
Rep: PUBLIC TYPE = RECORD[
   count: INT,
   flag: BOOLEAN ← TRUE,
   mumble: INT];

FugleFailure: PUBLIC ERROR = CODE;

NewFugleman: PUBLIC PROC RETURNS [Handle] = BEGIN
   RETURN [NEW[Rep ← [count: 5, mumble: -10]]];
   END;

Operations on a Fugleman

Print: PUBLIC PROC [h: Handle] = BEGIN
   -- some body would go here
   END;

Check: PUBLIC PROC [h: Handle] = BEGIN
   IF h.count>100 OR h.mumble<0 AND NOT h.flag THEN ERROR FugleFailure; -- example only
   END;

SomeProc: PUBLIC PROC [h: Handle, otherArgument: INT] RETURNS [r: INT] = BEGIN
   IF otherArgument>100 THEN otherArgument ← 100;
   h.count ← otherArgument;
   r ← 17;
   END;

END. -- FuglemanStdImpl

CHANGE LOG
   Created by Mitchell: March 17, 1980 11:33 AM
      Exemplary template
   Changed by Horning: June 7, 1982 6:17 pm
      Use Mesa.abbreviations
   Changed by Horning: June 8, 1982 2:15 pm
      Fix the use of the exported opaque type, add CEDAR prefix
   Changed by Mitchell: August 15, 1982 8:55 pm
      added needed declaration of FugleFailure
```

Cedar Style Sheet

```
-- FILE: CedarSample.mesa
-- Last edited by Mitchell, April 23, 1980 12:03 PM

CedarSample: DEFINITIONS =
BEGIN
upperLimit: INTEGER = 32;  -- name the upper limit of an interval type
IntervalType: TYPE = [0 .. upperLimit);  -- preferred form for intervals
Sample: TYPE = REF SampleRec;
SampleRec: TYPE = RECORD[val: Value, next: Sample];
SampleSet: TYPE = REF SampleSetRec;
SampleSetRec: TYPE = RECORD[head: Sample, count: Value];
Problem: ERROR [reason: ErrorCode];
ErrorCode: TYPE = {damagedSample, callingError, programError};
NewSample: PROCEDURE [val: Value] RETURNS [Sample];
NewSampleSet: PROCEDURE RETURNS[nilSet: SampleSet];
RemoveSample: PROCEDURE[toBeRemoved: Sample, from: SampleSet]
                                   RETURNS [inSet: BOOLEAN, setMinus: SampleSet];
  -- ERRORs: Problem[damagedSample]
  -- SIGNALs: ResumableCondition, SomeSignal
emptySet: SampleSet = NIL;
END.
This is where a global description of the module goes
CHANGE LOG
Changed by: YourName: DateTime
  DescriptionOfChange
```

No common suffix on the names of DEFINITIONS modules

Capitalize the first letter of module, procedure, signal, or type names

Capitalize the first letter of each imbedded word of a multi-word name

Standard prelude for a module

Naming a type is better than using an anonymous type constructor.

Use a small set of ERRORs with an error code parameter.

attach stylized comments to procedures that generate ERRORs or SIGNALs.

Standard postlude for a module includes a history log of (non-trivial) changes to the module

```
-- FILE: CedarSampleImpl.mesa
-- Last edited by Mitchell, April 23, 1980 3:19 PM
DIRECTORY
  SomeInterface USING [SomeProc, SomeType],
  CedarSample;
CedarSampleImpl: PROGRAM IMPORTS SI: SomeInterface
                         EXPORTS CedarSample =
BEGIN OPEN CedarSample;
Problem: ERROR[reason: ErrorCode] = CODE;
ResumableCondition: PUBLIC SIGNAL = CODE;
SomeSignal: PUBLIC SIGNAL = CODE;
NewSample: PUBLIC PROCEDURE [val: Value] RETURNS [Sample] =
  BEGIN
  sample: Sample;
  x: SI.SomeType = 0;
  IF val = 0 THEN val ← SI.SomeProc[x];
  . . .
  RETURN [sample];
  END;
NewSampleSet: PUBLIC PROCEDURE RETURNS [nilSet: SampleSet] =
  BEGIN
  nilSet ← AllocateSampleSet[ ! AllocFault => ERROR Problem[programError] ];
  nilSet ← [head: NIL, count: 0];
  END;
Bug: ERROR = CODE;
RemoveSample: PUBLIC PROCEDURE [toBeRemoved: Sample, from: SampleSet]
                                   RETURNS [inSet: BOOLEAN, setMinus: SampleSet] =
  -- ERRORs: Problem[damagedSample]
   BEGIN OPEN SI;
   IF . . . THEN ERROR Problem[damagedSample];
   WITH refAnyVar SELECT FROM
     r: REF REAL => rt ← rt + 1.0;
     i: REF INT => { . . .   SomeProc[y]; . . . };
     b: REF BOOLEAN => SIGNAL ResumableCondition;
     ENDCASE => ERROR Bug;
  . . .
  END;
AllocFault: ERROR = CODE;    -- error for local use in this module
AllocateSampleSet : PROCEDURE RETURNS [uninitedSet: SampleSet] =
  BEGIN
  . . .
  END;
END.
CHANGE LOG
Changed by: YourName: DateTime
  DescriptionOfChange
```

PROGRAM module name is normaly formed by suffixing interface name with "Impl". Alternatively, name may be totally different than interface's

NO names in same scope differing only by letter case distinctions except a value with same name as its type but with lowercase first letter

Qualify identifiers from interfaces

Keyword constructors, argument lists, and extractors preferred for multiple-component constructors

Use REF ANY instead of variant records and discriminate

Calls on single-argument procedures don't have to use keyword notation

OK to use unnamed OPEN of interfac if DIRECTORY entry has a USING list

OK to OPEN interface that module implements

Only let signals that are part of the abstraction escape out of it.

OK to OPEN an interface in a local scope where it is heavily used

Only raise SIGNALs using SIGNAL, and ERROR using ERROR

NO using ENDCASE to handle a singl remaining case: use as "OTHERWISE or to generate an ERROR

OK to have locally defined ERRORs and SIGNALs

JGM 16 AUG 82 Print on color printer if you can.

Filed on [Indigo]<CedarDocs>Language>StyleSheet.sil, .pre

BugBane - the Cedar Debugger

Russ Atkinson

May 27, 1983 3:44 pm

BugBane is the standard debugging support in the Cedar environment. It is organized as a collection of facilities, which include a simple Mesa expression interpreter, with support for breakpoints and uncaught signals.

BugBane is usually called from a Work Area. When the input focus is in a Work Area viewer you can type Mesa expressions (preceeded by "←"). Each expression is then evaluated and the results printed to the typescript. Each value is assigned to an interpreter variable for later use.

I. The Mesa subset

The subset of Mesa expressions interpreted by BugBane is roughly described by the following list of constructs. BugBane attempts to perform a reasonable amount of coercion to get values to agree with their targets. This coercion is usually more than the compiled langauge provides.

1, 1.2, "abc", 'A, TRUE, $atom-- *fixed, float, rope, char, bool, atom constants*

&x, foo, FooImpl -- *simple variables, according to search rules below*

X.Y -- *X is a record, ref to a record, pointer to a record, a global frame*

X[Y] -- *X is a sequence or array, ref or pointer to a sequence or array*

Proc[X, ...] -- *Proc is a procedure taking given arguments*

Type[X, ...] -- *a record or array constructor (Type may be implicit)*

NEW [T ← expr] -- *only in the local world*

X *op* Y -- *for op in {+, -, *, /,* MOD*}*

op[X, ...] -- *for op in {*MIN, MAX, ALL, LIST, CONS*}*

X ← Y -- *X and Y are expressions, [] ← Y is permitted*

II. Actions and multiple processes

Actions

In BugBane, an action represents the occurence of an uncaught signal (or error) or a breakpoint. An address fault is treated as an uncaught signal, and is therefore also an action. When an action occurs, the associated process is stopped. Other processes can continue, subject to the usual constraints (e.g. monitor locks).

The occurence of a new action usually spawns (creates) a new work area or reuses an old work area. When an action is associated with a work area the expression executed in that work area are interpreted with respect to the call stack for that action. Setting of breakpoints, stack display, proceeding and aborting are all performed with respect to that action.

Multiple processes

A process stack may only be examined by BugBane when the associated process is stopped. Any other approach is highly unsafe. The DebugTool can be used to stop processes. Another way to stop a process is to set a breakpoint in code that it executes. One difficulty with stopping processes by either method is that the process may be holding a resource necessary for BugBane. Since no automatic method for determing these resources is available, the decision as to when and where to plant the breakpoint is left to the user (sigh).

Proceeding a breakpoint will continue with program execution. Proceeding an uncaught signal will attempt to resume the signal (although there is no provision for resuming with arguments).

III. Name lookup

Name lookup in BugBane differs from name lookup in Mesa, primarily because the facilities available to the compiler are not available to the runtime, but also because there are more places to look at runtime. Names are found only if they are the right case and in the current "context." A context is like a search path, and is searched in the following order:

Debugger table lookup

The debugger name table contains a name to value association for identifiers. Aside from a few special cases (like TRUE, FALSE, and some built-in types), all names in this table start with "&". There is a separate debugger table, and hence a separate name space, for each work area.

Stack lookup (including global frames)

A stack exists for every action. When a name is looked up from a work area associated with an action the local frames (and associated global frames) are searched (last-created-first-found) for names up to some reasonable depth (currently 8).

Interface names

For lookup of X in expressions of the form X.Y, BugBane will attempt to treat X as an interface record and Y as a selector. This facility is quite new, quite slow, and somewhat frail. The user should try to use implementation module names when this facility does not work.

Global frame table

The global frame table is searched for global frame names only in last-loaded-first-found order. Variables in those global frames will not be found without qualification. The user should note that multiple instances of global frames are not supported.

Default global frame

The default global frame is set whenever one evaluates the expression that is the simple global frame name. Variables in that global frame will be found in this search. There is a separate default global frame for each work area.

IV. Octal commands

The following commands are intended for use by experienced users. They are made available as expressions, since they are applicable to contexts inherited from the interpreter (to allow for remote debugging).

&ar[ptr,len]

Ascii Read - Displays the Ascii characters starting at address *ptr* for *len* characters. Values of *ptr* that are smaller than 64K are interpreted as being MDS pointers.

&ars[ptr,len]

Ascii Read Short - Like &ar, but does not interpret values of *ptr* that are smaller than 64K as MDS pointers.

&clob[gf,pc]

CLear Octal Break - Clears the breakpoint set at the numerically specified global frame and program counter.

&fm[pattern]

Find Matching - If the pattern (as a ROPE literal) does not contain a period, &fm finds and prints the names of the global frames whose names match the pattern (case does not matter). If the pattern does contain a period, &fm finds the matching components of matching global frames.

&lob[]

List Octal Breaks - List all of the current breakpoints that were set by &sob.

&ofc[gf,pc,b0,b1,b2,b3,...,b9]

Octal Find Code - Starting at the specified global frame and program counter, find the program counter for the given code bytes.

&or[ptr, len ← 4, width ← 16, base ← 8]

Octal Read - Displays the contents of memory in the given *base* starting at *ptr* for the number of units given by *len*. The value of *width* must be in {1,2,4,8,16,32}. The value of *base* must be in [2..36]. Values of *ptr* that are smaller than 64K are interpreted as being MDS pointers.

&orc[gf,pc,len]

Octal Read Code - Displays the code bytes (in octal, unfortunately), starting at the given global frame and program counter for *len* bytes.

&ors[ptr,len]

Octal Read Short - Like &or, but does not interpret values of *ptr* that are smaller than 64K as MDS pointers.

&ow[ptr,word,len]

Octal Write - Writes the given 16-bit word into the given address for the number of words given by *len*. Values of *ptr* that are smaller than 64K are interpreted as being MDS pointers.

&ows[ptr,word,len]

Octal Write Short - Like &ow, but does not interpret values of *ptr* that are smaller than 64K as MDS pointers.

&pgf[gf]

Print Global Frame - Prints the components of the specified global frame.

&plf[lf]

Print Local Frame - Prints the components of the specified local frame.

&sob[gf,pc]

Set Octal Break - Sets a breakpoint at the speicifed global frame and program counter.

&tgf[gf]

Trust Global Frame - "Trusts" the given number as a global frame. This allows examining or setting named components, as in &tgf[*gf*].name.

&tlf[lf]

Trust Local Frame - "Trusts" the given number as a local frame. This allows examining or setting named components, as in &tgf[*lf*].name.

V. Shortfalls

Performance is only moderately acceptable.

Although there is substantial caching of results in AMTypes and in BugBane, the first time through for many expressions may take a long time, particularly if the symbols are not available on your local disk. Patience is the only current recommendation, although we are certainly concerned about performance.

One name - one global frame.

Recompiling and reloading a module will work only with very simple systems. BugBane only provides a linear search space for global frame names (last loaded, first found). Therefore you can only find one instance of a global frame with a given name.

Peculiar name interpretation.

BugBane has both better and worse interpretation of names than the compiler or CoPilot. Unlike the compiler, BugBane knows nothing about OPEN, so many names that can be left unqualified in compiled programs must be fully qualified for interpreted expressions. BugBane will search for names both in the current call stack and in the global frame table.

Opaque types are not fully supported.

The mapping between opaque and concrete types is not supplied by the runtime system. This leads to type clashes when you try to hand a value of opaque type to a procedure that expects a concrete type, and vice versa. There is only one automatic coercion practiced, which is when a REF opaque is passed to a routine that expects a REF concrete. In this one case, the object carries the concrete type, and the coercion is made automatically. In other cases, LOOPHOLE should be used as a work-around.

Some safe expressions are not yet supported.

SELECT expressions are not supported. Very few expressions involving types are supported, and there may always be a few shortfalls here. Statements are not supported, so catch phrases in expressions will not work.

Variant record constructors are limited to positional notation (keyword notation is not supported).

LIST constructors are only partially implemented, since they ignore the zone option, and only work for LIST OF REF ANY.

Object notation is supported for procedure invocation, but may not do all that the compiler does. In particular, NIL as the first argument in object notation will not work.

Selection from interfaces is only partially implemented. The most likely result of not having the appropriate module loaded or not having the interface exported to the top level is that NIL will be returned. It is also the result for INLINE procedures.

There are no plans for BugBane to interpret more than expressions.

Some type constructors are not supported.

AMTypes does not allow the construction of types on the fly, so BugBane does not support type constructors. This means that type constructors such as POINTER TO Foo *cannot* be interpreted. BugBane does treat types as expressions where possible, so type constructors such as FooDefs.FooPointer are legal (assuming that the symbols are available to BugBane). The difference is that with FooDefs.FooPointer sufficient information is passed from the compiler to the runtime to allow interpretation of the type.

Some unsafe constructs are not supported.

In particular: DESCRIPTOR constructors and overlaid record constructors. UNSPECIFIED and LONG UNSPECIFIED are discouraged, since they are both obsolete and buggy (CODE[LONG UNSPECIFIED] will crash the compiler, for example).

Although LOOPHOLE is supported, no checking for lengths is provided, so you can really do terrible things by typing the wrong thing. It is, of course, your machine.

Procedure type equivalence is only partially implemented.

While this is essentially an AMTypes problem, it does impact BugBane users. Procedure types must match *exactly* in order for a procedure to be an argument. If you can't get the interpreter to understand, use LOOPHOLE (sigh).

Remote debugging is a bit buggy.

We're working on it. It is not possible to pass ROPE or ATOM constants to remote procedures, for example. LIST, CONS, and NEW do not work for the remote world.

Context reporting is unreliable.

There are several parts to this problem.

1: The compiler performs certain optimizations, such as cross-jumping and constant folding, that keep the object code from being linearly related to the source code. This drawback will not be overcome for some time.

2: There is a long-standing bug in reporting locations inside of catch phrases.

3: Context reporting involving INLINE procedure and cross-jumped code is unreliable.

4: Due to a long-standing bug in the format of fine-grain tables, the source position of the last statement in a procedure is not known. Therefore, when setting breakpoints on the last statement, make sure that the selection points to the first character of the last statement.

The abstract machine has some rough spots.

This is a subset of the know problems.

1: The eval stack is not always empty between statements. If the value of a variable has been placed on the stack in one statement, it may be left on the stack for the next statement. Attempts to set that variable may not work for the latter statement.

2: It may not be possible to form the address of a variable that does not occupy a whole word.

3: The program counter used to report breakpoints may be wrong by 1, which may cause the previous statement to be reported as the location of the breakpoint.

VI. Recent changes

for Cedar 4.2

Remote debugging

Various bug fixes should improve remote debugging.

Bug fixes

Minor bug fixes only (particularly to octal commands).

Structural merge

The octal commands mentioned above are now a standard part of InterpreterPackage, which is a part of the boot file.

Additional features

Expressions of the form **v[tag]**, where v is an overlaid or computed variant record and tag is the name of one of the variants, now coerce the record to be of the specified variant type.

for Cedar 4.1

Remote debugging

Remote debugging is supported through the use of DebugTool, a program by Andrew Birrell. Since this tool is in the boot file, previous methods of remote debugging are now inoperative. There have been significant bug fixes in this area, especially to remove problems when the remote world has been booted, and the use of interfaces.

Structural split

BugBane.bcd and BBV.bcd have been rearranged into BugBaneBelow (lower-level modules) and BugBaneAbove (higher-level modules). BugBaneBelow exports everything that BugBane used to export *except* BBObjectLocation, BBDisableBreaks, BBAction, BBBreak, BBFocus, and BBNub. BugBaneAbove exports everything that BBV used to export, plus the former BugBane interfaces not exported by BugBaneBelow. Most clients should not notice the difference.

Bug fixes

Performance has improved somewhat. A few minor bugs have been repaired. Error messages have been slightly improved.

Additional features

Array constructors, including ALL[*expr*], are supported. NEW[*type* ← *expr*] is supported. MAX[*expr*, ...] and MIN[*expr*, ...] are supported.

for Cedar 4.0

Remote debugging

Remote debugging is supported through the use of DebugTest, a program by Andrew Birrell. The Cedar debugger will be automatically set up if Cedar is installed on the CoPilot volume. To run the remote debugger in the Client volume, perform:

run AMProcessBasicImpl; run AMProcessImpl; run DebugTest

DebugTest *RemoteMachineName*

Bug fixes

Minor bug fixes only.

Additional features

Variant record constructors now work (positional notation only).

for Cedar 3.5

Remote debugging

Remote debugging is now supported, although the user interface is quite incomplete. Actions may now come from any enabled world. To start listening to a remote machine, evaluate BBNubImpl.FindWorld[name: *name*, new: TRUE], which will return a world (WorldVM.World) for the named machine. A new action should appear from that machine (if it is set up for remote debugging). Please note that invocation of remote procedures and some remote assignments (especially those involving allocation and reference-counting) are not yet allowed.

Bug fixes

Minor bug fixes only.

Restructuring

BBObjectLocation is now implemented through the AMModel. This interface will probably disappear completely in Cedar 4.0.

for Cedar 3.4

New approach to read-eval-print

Removing the default read-eval-print loop and the default error reporting provides greater separation between BugBane facilities and BugBane clients (the UserExec & BBV). The choice of whether to make a UserExec "stall" if it gets an error, to make it look at the new error, or to provide some other service is no longer dictated by the code of BugBane.

BBV provides default reporting of new actions (breakpoints & uncaught signals). The user is able to select the "current" action through the current menu interface. Facilities for displaying the current stack and actions are also provided.

The UserExec provides an work area (evaluator) for each action. A work area is a read-eval-print loop in the classic style. However, when a breakpoint or uncaught signal occurs for that evaluator other work areas may be spawned in response.

All of these user interfaces should be regarded as experimental.

Separation of printing

Printing of TVs and types has been unified under the control of the IO package maintainer (Warren Teitelman). The implementation no longer appears in BugBane. Russ Atkinson will assist in maintaining this package during the life of Cedar 3.4. However, Warren should be consulted about formatting.

New features

LOOPHOLE is now better supported. An implicit LOOPHOLE with a blank target now defaults to LONG CARDINAL as the target.

BBV provides better error messages. It now reports the source version its wanted if it can't get the right one.

Parsing has become a little faster. This is due to more cases handled by the quick-kill parser, and caching of intermediate storage by the long-winded parser.

The load state is used instead of the GFT to establish global frame search order. This provides true last-loaded-first-found search order. Caching of global frame searching is now substantially improved as well.

Non-started global frames can be found and manipulated much like started global frames.

It should now be possible to set breakpoints in viewers that view remote sources.

Coercion of REF opaque to REF concrete at procedure boundaries has been improved. Coercion of REF ANY variables to REF something values has also been improved. Coercion between opaque and concrete types is not otherwise supported.

Remote debugging

Most users should not see any changes from the BugBane changes in support of remote debugging. Remote debugging is still not supported in Cedar 3.4.

for Cedar 3.2

New features

LOOPHOLE is somewhat supported. Non-RC (reference containing) values of less than 3 words in length can be freely LOOPHOLEd based on the target type. Therefore, LOOPHOLE is only useful when constructing a procedure argument or assigning to a variable in either a global or local frame (not a debugger variable). Explicit types for LOOPHOLE are not yet supported..

Some octal debugging features have been added. In particular, Octal Read and Ascii Read are useful in attempting to examine variables..

BugBane now can access (through RTTypes) symbols files in the Cedar release that are not on the local disk. This access should only be noticable if the server is down or especially slow.

Bug fixes

Improvements have been made (and bugs fixed) in signal handling and reporting. In particular, features regarding catch frames should now work.

The infamous doubled breakpoint problem (where a breakpoint would immediately recur upon proceeding it) will not entirely disappear until the Trinity release of Pilot. The problem lies in the interaction between breakpoints and page faults. In the meantime, the code is touched immediately before proceeding the breakpoint to force it in.

The Cedar DebugTool

Version 4.2

Release as [Indigo]<Cedar>Documentation>DebugTool.tioga, .press
Came from [Indigo]<CedarDocs>Manual>DebugTool.tioga, .press
Last edited by Birrell on June 14, 1983 2:24 pm
by Horning on June 7, 1983 4:23 pm

Abstract: This tool appears automatically in CoCedar, and may be created in the normal Cedar environment with the "Debug" command. The tool can be set up to inspect the environment in which it is running, or to teledebug an instance of Cedar running on another machine. In CoCedar, the tool can be used to look at the normal Cedar environment which has been stopped by outloading it to disk. The tool allows you to inspect and modify the state of parallel processes, and to enumerate the set of loaded packages and modules.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

DRAFT - For Internal Xerox Use Only - DRAFT

The Cedar DebugTool

Introduction

This tool appears automatically in CoCedar, where it is initially set up to look at the normal Cedar environment, which was outloaded to disk when you swapped to CoCedar. The tool can be created with the "Debug" command. If you give the command with no arguments, it creates a tool for inspecting the environment in which the tool is executing. If you give the command with the name of another machine as argument, it creates a tool for teledebugging that machine. The environment which the tool is looking at is called the tool's "client". Most of the tool is concerned with looking at the parallel processes of the client; there are also a few facilities not related to processes.

Non-process-related facilities

At the bottom right corner of the menu are the "Screen" and "Kill" commands. These have no effect if the client is the currently executing environment. In CoCedar, the "Screen" command swaps back to the client to allow you to look at the client display (for about 20 seconds, or until you press the "swat" key); when teledebugging, thgis causes the machine which is being debugged to show the client's screen. In CoCedar, the "Kill" command is similar to pressing the boot button; when teledebugging, it is similar to pressing the boot button on the machine which is being debugged. At the bottom left of the menu are "List LoadState" and "List Context". "List LoadState" enumerates the currently loaded configurations in the client. If you type the name of a configuration as the text beside the "Context" button, then click "List Context", it will enumerate the modules in that configuration.

Processes

The process facilities are based on the concept of "freezing" a process at some point on its call stack. This means that the process continues execution normally, but when it returns to that point on its call stack, the process will stop executing until it is thawed. Of course, there is the special case where you freeze the process at its current frame, so it stops executing immediately. The DebugTool allows you to freeze and thaw processes, and to look at the frozen frames of processes (even while the rest of the process is still executing). There is no way to look at a process (or part of a process) that isn't frozen. There are four "Freeze" buttons: "All", which freezes all processes at their current frame; "Ready", which freezes all ready processes at their current frame (i.e. processes that aren't suspended for a monitor lock, condition variable wait or fault); "Process" which freezes the process whose number (PSB index) is given in *octal* as the text beside the "PsbIndex" button on the right of the viewer. Finally, the "Context" button will freeze all processes associated with a perticular configuration or module, freezing them at the boundary of the context. More precisely, "Context" enumerates all processes, and determines every process that has somewhere on its call stack a frame within the specified module or configuration; each such process is frozen at the point where control would return to within the module or configuration; any process currently executing (or waiting or faulted) within the module or configuration stops executing immediately. If you click the RIGHT mouse button on the "Freeze Context" button, then the entire call stack of each such process is frozen.

Examining frozen processes

For each process that is frozen, the DebugTool displays three buttons, labelled with "Adjust", "Thaw", and a description of the process's current state. The state says things like "ready" or "waitingCV" or "pageFault", followed by the name of the frame at which the process is frozen and the priority of the process. There is no way to look at non-frozen frames of the process. The "ready", "waitingCV", etc. are wrapped in parentheses if the process's current frame is not frozen (i.e. if the process is still executing). If you click the LEFT mouse button on the process-state button, DebugTool gives a brief summary of the process's call stack (from the point at which it is frozen). If you click the RIGHT mouse button there, DebugTool will create an action area for that process, which allows you to look at the frozen parts of the call-stack in more detail. The "Thaw" button associated with a frozen process will un-freeze the entire process. If you click the LEFT mouse button on the "Adjust" button, it will change the point at which the process's call-stack is frozen to be the boundary of the module or configuration specified in the text for "Context" at the right of the viewer. CLicking the RIGHT mouse button there will freeze the entire call stack of the process. You can't thaw or adjust a process while it has an action area—proceed the action area first.

They're really frozen!

If you resume an outloaded client (for example, by proceeding the action that invoked the world-swap debugger) while some processes are frozen, they really are frozen when the client resumes. This can be important when debugging parallel computations, but it can be dangerous. In particular, beware of resuming the client while critical system processes are frozen!

Style

You will find the DebugTool more pleasant if you use the full power of its process enumeration facilities. You should very rarely need to use "Freeze All". Almost always, you know what set of modules or configurations you're debugging; if you freeze just the processes associated with them, DebugTool will do the filtering for you and you won't have so much trouble deciding what process to look at.

Bugs

There are still several bugs. **Please** let an implementor know if something doesn't work. If CoCedar breaks, however, you still have a chance to look at your outloaded client. If CoCedar has been invoked by a client and you haven't resumed the client at all (e.g. UserScreen), you can restart the debugger without losing the client outload by booting your Debugger volume with the "W" switch. (CoCedar does this automatically in the situation where CoPilot says "inloaded twice, click to boot"). Alternatively, you can reinstall CoCedar by booting your debugger volume with no switches, in which case after outloading itself the debugger will boot your physical volume.

Symbol tables

Just like Cedar, CoCedar often manages to find the correct symbol table automatically using the version maps. However, this is unlikely to be true for the modules you're debugging. CoCedar will not automatically look at files on the client volume—you'll need to fetch them explicitly onto the debugger volume using BringOver, or the ClientFileTool.

DFFiles documentation

Release as [Indigo]<Cedar>Documentation>DFFiles.tioga

Last edited Russ Atkinson on June 8, 1983

Abstract This documentation describes the software that handles DF files. Of particular interest are the programs BringOver, SModel and VerifyDF.

Table of Contents

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Introduction

The programs described below comprise a general package for file management with explicit version control. Each program takes some number of DF files, which are lists of filenames, one per line, specifying a remote host and directory where the file is normally stored, followed by the file's creation date. (In certain situations some of this information can be omitted.)

The DF files system was initially used by people running Mesa on public Dorados who wanted to guarantee they had the correct version of files they needed and as an easy way to save changed versions of file without unnecessary copying. It is now being used to partially automate the Cedar and Mesa group release process as well. Eric Schmidt was primarily responsible for the programs and the initial versions of this documentation.

Of the commands described in this document, these three are most heavily used:

> *BringOver* - will insure that the files listed in a DF file are retrieved from their remote file servers, possibly overwriting different versions already on the local disk.
>
> *SModel* - once BringOver has been run and a few files have been changed (possibly by editing them), SModel will store the newest versions on the remote file servers and produce a new DF file containing references to the newest versions.
>
> *VerifyDf* is applied to a Cedar .Bcd file and a DF file and verifies that all files needed to re-build that .Bcd are listed in the DF.

This document is intended as the definitive documentation for the commands and procedures described here. Please send messages with comments, questions, and discrepancies to CedarSupport↑.PA. Messages about changes to the DF files system are sent to CedarImplementors↑.PA and/or CedarUser↑.PA.

You might also read "Cedar Releases: Policies and Procedures" by Roy Levin, stored on [Indigo]<Cedar>Documentation>ReleaseProcedures.Press and .Bravo, for information on use of DF files in the Cedar release process.

Simple DF File Format

A simple DF file is composed of some control lines and lines of the form:

Directory [host]<directory-path> <file name and version> <create-date>

For example,

Directory [Indigo]<Poplar>Cedar>pl.mesa!13 2-Oct-80 15:43:09 PDT

specifies a specific version number for the file server, and also lists the file's create-date. Note that a file cannot have the name **Directory**, and upper- and lower-case differences are ignored.

Blank lines are OK and lines are treated as comments if they begin with "//" or "--".

The format of DF files is similar to that produced by the Local-List or Remote-List sub-commands in the FileTool. You may make it almost identical by setting the options in the FileTool so that only the file name and file create dates are listed.

Example DF File:

An example file might be named "Poplar.DF" and contain:

-- *File Poplar.DF, created 5-Oct-80 19:48:32*

Directory [Indigo]<Poplar>Cedar>

```
    poplar.bcd!16                         3-Oct-80 13:26:28
```

```
checkpoint.bcd!3              24-Sep-80 11:14:26
AFiles.bcd!5                   2-Oct-80 17:17:32
overviewdefs.bcd!6             2-Oct-80 17:16:16
coredefs.bcd!7                 2-Oct-80 17:16:20
filepageusedefs.bcd!5          2-Oct-80 17:17:28
filepageusedefs.mesa!1        26-Oct-79 12:45:39
filesystemdefs.bcd!5           2-Oct-80 17:17:31
filesystemdefs.mesa!1         22-Oct-79 13:21:59
```

BringOver

If the name of your file is "Poplar.DF", you type

```
    BringOver Poplar.DF
```

to the Executive.

BringOver works as follows: It reads the DF file one line at a time. It takes the remote file name, strips off the directory information and looks to see if it is on the local disk. One of three things can happen:

(1) If the file is not on the local disk, BringOver will try to retrieve it.

(2) If the file is on the local disk, BringOver looks at the version on the local disk. If the create-date on the local file differs from the the create date listed in the DF file, BringOver will try to retrieve the remote version.

(3) If the create-date is omitted from the DF file, BringOver will ALWAYS try to retrieve the file.

If you omit the IFS version number (e.g. "!3"), BringOver will enumerate all the versions of that particular file looking for one with the correct create time. If there are no versions of the file you list in the DF file on the remote host in the directory you specify, BringOver will give you a warning message. If there are files with the same name and none of the create-dates avaliable match that listed in the DF file, BringOver will give you a warning and offer to retrieve the latest version.

After running BringOver you can be sure the files listed in the DF file are on your local disk, and that their create-dates agree with the create dates listed in the DF file, or BringOver will have printed out error messages.

Normally BringOver will list each file to be retrieved and will ask for confirmation. (You may reply "y" or CR to confirm, "n" to skip retrieval of this file, "q" to quit the BringOver program altogether, and "a" to retrieve this file and transfer subsequent files without further confirmation.) BringOver will transfer all the files without confirmation if you provide the /a switch:

```
    BringOver.~ /a Poplar.DF
```

BringOver can read a remote DF file as easily as a local one:

```
    BringOver.~ [Indigo]<Poplar>Poplar.DF
```

reads Poplar.DF from the remote file server as specified. (If the host file system is "Indigo", the "[Indigo]" can be omitted.)

As files are brought over, a property (called the *RemoteFileName* property) is added to their leader page recording where the remote copy can be found. These properties can be printed out by the DFDisk program (see below.)

Comments may be indicated by lines beginning with // or --, as in the Poplar.DF example above. In addition, BringOver will print comments at the end of the file after the last line listing files when it is run. This way distributors of software can print out useful reminders to their users.

Instead of giving create dates

If the create date entry is a "~=" rather than a normal date, BringOver will retrieve the file only if the version on the remote server is different than the version on the local disk. If the file is not on the local disk, it will be retrieved. For example,

```
Foo.Mesa                    ~=
```

If the create date entry is a ">" rather than a normal date, BringOver will retrieve the file only if the version on the remote server is newer (or more recent) than the version on the local disk. If the file is not on the local disk, it will be retrieved. For example,

```
Foo.Mesa                    >
```

will retrieve from the remote server only if there is a newer version on the remote server.

Indirect Use of Other DF Files

There are two common, distinct uses of one or more DF files by another. The first is syntactic inclusion, similar to macro-expansion, where a DF file references another and all programs treat these DF files as if they were a single DF file with the text of the inner DF file substituted in the outer one at the point of reference. The second use is as an "Importer" of another system, package, or set of files controlled by someone else. By analogy with these concepts in Mesa, there are **Exports** as well.

The DF file syntax allows both forms of use, which are indicated by different keywords before the name and date of the nested DF file.

Syntactic Inclusion

A line of the form

Include [host]<path>file.DF **Of** <date>

will cause BringOver to invoke itself on file.DF at the point it encounters the **Include** statement. If the included file itself has an **Include** statement, then BringOver will again invoke itself on the inner DF file, and so on, in a recursive fashion. Furthermore, the DF file is retrieved using the usual BringOver rules before the recursive call.

Examples of **Include:**

Include [Indigo]<Cedar>CompilerSources.DF **Of** 2-Oct-81 15:43:09 PDT

Include [Indigo]<Cedar>CompilerObjects.DF **Of** >

(The **Of** keyword is optional, and the **Include** statement may be split on multiple lines.)

Exports and Imports

DF files users are encouraged to use a single DF file for packages they maintain. Some, but usually not all, of the files an implementor needs are also needed by users, or clients, of the package. Files that are useful to both users and implementors are said to be *exported.* These files are indicated by being listed below a statement similar to the **Directory** statement:

Exports [host]<directory>

<list of files of use to users and the implementor>

Directory [host]<directory>

<list of files of use only to the implementor>

BringOver normally examines all the files in a DF file (i.e. it assumes you are an implementor.) If you are a user, you can get only those marked **Exports** by giving BringOver the /p switch:

```
BringOver /p Poplar.DF
```

Users often need to specify the /p and /a switches. The correct way to invoke BringOver is

```
BringOver.~ /a /p Poplar.DF
```

(Note the space before the /a and between /a and /p.)

When a DF file needs to refer to other DF files as a user, it may have an **Imports** statement

Imports [host]<path>file.df **Of** <date>

which causes BringOver to (1) retrieve file.df to the local disk if necessary and (2) examine all files marked **Exports** in file.df and retrieve them if necessary. Of course file.df may have **Include** or **Imports** statements, so this is a recursive algorithm.

Sometimes the **Exports** files are too few or too many files and the user of a package would like to list the files he/she needs from the imported DF file. Appending a **Using** clause to the **Imports** statement, analogous to the one in the Mesa language, gives the user explicit control over which files he/she needs:

Imports [host]<path>Package.DF **Of** <date>

Using [list of files, separated by commas]

(The **Of** *<date>* clause can be omitted.)

Use of the **Using** clause is strongly recommended, but is not required.

Examples of **Imports**:

Imports [Indigo]<Cedar>Pilot>PilotMesaFriends.DF

Imports [Indigo]<Cedar>CoPilot>CoPilot.DF **Of** 24-Sep-81 11:14:26 PDT

Using [CPSwapDefs.Bcd, CPSwap2.Bcd]

As with the **Include** clause, the **Imports** statement (and **Using** clause) may span multiple lines.

In the CoPilot.DF example above, CPSwapDefs.Bcd and CPSwap2.Bcd can appear anywhere in CoPilot.DF. (It does not matter whether they are **Exports** or not when a **Using** clause is used to access them.)

The files referred to by an **Imports** statement may themselves be exported to the next outer DF file by preceding the word **Imports** by the word **Exports**. This is useful when users of a package you maintain need to have files from some other package in order to (e.g.) compile their system. You can import and export these files to assure your users will have them.

Using Clause Semantics

Since, in general, CoPilot.DF (in the example above) may have internal structure (i.e. it references other DF files with **Include** and **Imports** lists), CPSwapDefs.Bcd and CPSwap2.Bcd may be defined in some of the DF files referenced by CoPilot.DF. To avoid an explosion in search time, these restrictions apply when using **Using** lists on a DF file with internal structure:

(1) A using list file (e.g. CPSwapDefs.Bcd) may be retrieved from a DF file included by an imported DF file (e.g. included by CoPilot.DF). (In other words, included DF files are treated as if they were macro-substituted inline in the top DF file.)

(2) A using list file may be retrieved from a DF file imported (without a **Using** list) by an imported DF file (e.g., imported without a **Using** list by CoPilot.DF) as long as the file is exported in the inner DF file. (I.e., imports are significant one level below the top DF file, CoPilot.DF.)

(3) A using list file may be retrieved from a DF file imported with a **Using** list by an imported DF file (e.g. imported with a **Using** list by CoPilot.DF) as long as the using list file being retrieved also appears on the inner using list. (I.e., multiple, nested **Using** lists must intersect.)

Useful Modes

A option (/v) will run BringOver in *verify* mode, where it will verify that the files listed in the DF file are present on the remote servers. (No files are retrieved in this mode.) If the file is listed correctly and a local copy exists, BringOver will add the RemoteFileName property to the leader page. Note for large DF files the verify option takes a few minutes. BringOver will inform the user if there are newer versions of files on the remote file servers, and will make a new DF file (on a file called "NewDFFile.DF") listing the newer versions of files listed in the original DF file. If any files were listed in the DF file without their IFS version numbers (e.g. !5), BringOver will make a new DF file with those version numbers filled in (also on "NewDFFile.DF").

An option (/o) will instruct BringOver to only BringOver the file(s) listed after the /o. For example,

```
BringOver /o x.mesa files.df
```

will examine only the file "x.mesa" in files.df, even if other files should be brought over. If no "." is given, BringOver will append ".Mesa", so this is equivalent to the preceding example:

```
BringOver /o x files.df
```

More than one file can be listed after /o, in which case the last DF file on the command line is searched.

Five options are available to reduce the number of files that are retrieved when BringOver is run. /b causes BringOver to retrieve only those files that are "derived" from source files, i.e. files that end in ".Bcd", ".Signals", ".Boot", and ".Press". /s causes BringOver to retrieve only non-derived files, i.e. all files that do not end in ".Bcd", ".Signals", ".Boot", and ".Press". /r causes BringOver to retrieve files that are in directories marked **ReadOnly** in the DF file and to retrieve files that are imported from other DF files. /w is the inverse of /r: only files that are not marked **ReadOnly** or imported are retrieved. Finally, /u causes BringOver to retrieve a file if a version is already on the local disk and the version listed in the DF file differs from the version on the local disk.

The /f option causes BringOver to ignore local files when it decides to retrieve a version of a file, thus /f "forces retrieval" of files on the local disk.

Other information about BringOver

When the filename on a line in a DF file is preceded by a tilde ("~"), BringOver will retrieve the indicated version if it is not present on the disk. If there is already a version on the disk, then it will not be overwritten, regardless of its create date.

BringOver listens for Control-DEL. When typed, BringOver will abort itself.

BringOver can handle DF files with fewer than 500 files.

Should you need to supply a connect name and password, BringOver will prompt you for them. When a CR for the connect name and a CR for the connect password are typed, BringOver will ask if you want to use a different login name or abort the program.

BringOver can be given a list of DF files on the command line. It will behave as if each file had been given separately.

BringOver can be used to retrieve files that are not listed in DF files by putting them on the command line

```
    BringOver [Ivy]<Schmidt>User.CM!3 [Indigo]<Cedar>Lister>Lister.Bcd
```

will retrieve those two files. If a file ends in ".DF", BringOver will retrieve the contents as well.

If the version of a file on the local disk is newer than the one BringOver is retrieving, BringOver will first rename the local version by appending "$$" to the filename and then retrieve the local file. If the file ends in ".Bcd", BringOver does not bother to rename the file.

Summary of options to BringOver:

/a Transfer files without requesting confirmation.

/b Retrieve only "derived" files.

/f Retrieve files to the local disk even if create time of local copy agrees with the create time of the remote copy.

/o Transfer only the following files (extension defaults to ".Mesa"). The last file in the list is the DF file.

/p Transfer only those files marked **Exports** or **Public.**

/r Transfer only those files marked **ReadOnly** or imported from other DF files.

/s Transfer only those files that are not "derived".

/u Transfer only those files that are already on the local disk, but the local version differs from the remote version (update mode).

/v Verify mode; No files are transferred but checks the DF file for accuracy.

/w Transfer only those files not marked **ReadOnly** and not imported from other DF files.

SModel

The SModel command (SModel stands for "Simple Modeller") can be used to store back new versions of files you've changed since you last ran the BringOver command to fetch files under control of a DF file. For example, if your DF file is called "Poplar.DF" and you've already run the "BringOver" command, the command

```
    SModel Poplar.DF
```

will execute as follows: The files listed in Poplar.DF are checked on the local disk. If any have different create dates SModel will store them on the remote servers specified in Poplar.DF. Then SModel copies the DF file to one ending in "$" (e.g. Poplar.DF$), and produces a new Poplar.DF file with the new create dates and remote file system version numbers (e.g. !4).

If a file on the local disk is listed without any create date in the DF file, SModel will fill in the create date from the version on the local disk and then store the file out on the remote directory. If the file is listed in the DF file followed by a ">" or "~=", this file is ignored and will not be transferred.

If the DF file contains a reference to itself (e.g. System.DF lists "System.DF" as one of the files you want), then SModel will also store a new version of the DF file on the remote server. Since SModel has to write out a new DF file before it can store the DF file, SModel cannot put the IFS version number (e.g. !5) on the DF file name in the case of self-referencing DF files. However, BringOver will always get the correct version of the DF file since it will use the create-date of the DF file instead.

SModel will pause before storing each file and ask if it should go ahead. You may respond with "y", "n", "q", or "a". "y" means go ahead and store it, "n" means do not store this file and keep running, "q" means do not store this file and stop running immediately, and "a" is like "y" except subsequent questions will not be asked, and the "y" answer will be assumed. You can give the "/a" option on the command line to SModel, e.g.

```
    SModel /a Poplar.DF
```

and all questions will automatically be answered "y".

ReadOnly files

When you don't want any of the files in a directory to be changed by SModel, regardless of any changes you may make, you can indicate the files in the directory are read-only by preceding the **Directory** keyword with **ReadOnly**. For example:

ReadOnly Directory [Indigo]<Mesa>System>

indicates the files listed below should NEVER be stored back on <Mesa>System>.

Maintaining Consistency

It is important to remember that SModel NEVER enumerates the remote file system, so SModel may not detect that certain files listed in a DF file are not present on remote servers.

A common mistake is to assume that if you run SModel on a DF file successfully, and then simply change a **Directory** in the DF file, then all the files will be transferred (again) to the new directory. This is WRONG! After SModel has been run the first time the create dates listed in the DF file will agree with the dates of the files on the local disk. Since SModel checks the create dates listed in the DF file, on the second invocation SModel will not detect that any files need be transferred EVEN though the **Directory** has been changed.

To resolve these problems, SModel will take a /v option (/v stands for verify.) If given this /v option, SModel will not only apply the algorithm described above to store files, but if it decides a file does not need to be stored, it will look on the remote file server and check that in fact the file does not need to be stored. If the file is not on the remote directory, or the version listed in the DF file is not on the remote directory, then SModel will store the file. In this way SModel /v will try to force the remote directory to agree with the DF file.

Include and Imports

SModel will invoke itself recursively on DF files that are specified in **Include** statements (and non-**ReadOnly** DF files preceded by @.) SModel will NOT invoke itself on DF files listed in **Imports** statements (or on DF files listed in **ReadOnly** directories preceded by @.)

If the **Include** or **Imports** statement is not followed by an **Of** *<date>* clause, SModel will insert such a clause in its output DF file with <date> replaced by the create date of the file on the local disk.

Release Option

If the /p option is given, SModel will store all files listed in the DF file on subdirectories of [Indigo]<PreCedar>. The subdirectory will be the same as the subdirectory given by the **ReleaseAs** statement. If the files listed in the DF file are normally stored on a working directory that is not <PreCedar>, the copy of the DF file on the local disk is not rewritted by SModel /p. If the normal working directory of the files in the DF file is <PreCedar>, then /n is equivalent to /v.

Other Options

1. The "/n" option instructs SModel to do everything it would normally do BUT it never transfers any files. For example,

```
SModel /n Poplar.DF
```

2. If you give the "/s" option to SModel, it will store onto the remote server(s) all the files you've listed in the DF file, except those marked **ReadOnly**, whether or not local create dates match the ones in the DF file. For example,

```
SModel /s Poplar.DF
```

3. If the /r option is given, then SModel will store files it decides should be stored by the above algorithm *even* if the directory they are stored onto in marked **ReadOnly**.

Getting Started with SModel

New users are often confused about the relationship of the entries in a DF file to the local and remote directories, and what SModel will do in certain cases. The easiest way to understand it is that SModel assumes 1) that the DF file was an accurate description of the remote directory at some point in the past and 2) files with differing create dates that it finds on the local disk are the "Truth" and that it should transfer any files that are different. However, assumption #1 allows SModel to assume that files with the same create date in the DF file and on the local disk ALSO exist on the remote server. This assumption is often not true when a new DF file is being created. To get started, there are at least these things you can do:

1. Make a DF file of just file names (no create dates) and run SModel on it. The files will be transferred and the create dates will be filled in.

2. Use VerifyDF (see below) to automatically generate a DF file, edit the DF file as you need to, and use the /s option to store all the files out onto remote servers.

3. If you have a complete consistent world out on remote servers, simply retrieve all those files onto the local disk and make a DF file with all their filenames but no create dates. Run SModel with the /n option to get the DF file filled in with all the create dates. (/n will not needlessly make extra copies of the files on your remote directories).

If you are maintaining a system for others, you may need to have entries in the DF file that are **ReadOnly** (such as for <Pilot>Defs>, etc.) Filling in all the create dates by hand is painful, so you may combine the /r and /n options to fill in all the create dates in the DF file but not transfer any of the files! For example,

```
SModel /r /n Foo.DF
```

will fill in all the create dates for files in Foo.DF but none will be stored.

CameFrom handling

SModel normally changes entries like

Directory [host2]<directory2> **CameFrom** [host1]<directory1>

into lines like

Directory [host1]<directory1> **ReleaseAs** [host2]<directory2>

and changes **Imports** like

Imports [host1]<directory>file.DF **Of** <date>

CameFrom [host2]<directory2>

into

Imports [host1]<directory>file.DF **Of** <date>

since the **CameFrom** is inserted by the Release Tool. If you want to prevent SModel from making these changes, give SModel the /f option.

Other Information about SModel

SModel listens for Control-DEL. When typed, SModel will abort itself.

SModel handles Connect user names and passwords exactly as BringOver does (see above).

If a file is transferred, the *RemoteFileName* property (see above) is added to the leader page of the file on the local disk to record where a remote copy can be found.

Summary of options to SModel:

/a Store files without asking the user for confirmation.

/c Try to write DF files that are compatible with Alto DF software. (No warnings are given if this is impossible.)

/f Do not switch **CameFrom** clauses to **ReleaseAs** clauses and do not delete **CameFrom** clauses from **Imports.**

/n Do everything it would normally do BUT never transfer any files.

/p Store files on the <PreCedar> directory for the next Cedar release.

/r Ignore the **ReadOnly** attribute when deciding to store files.

/s Store all the files in the DF file, even if the create dates are correct.

/t Do not invoke SModel recursively on inner, included DF files. Only examine the top DF file given.

/v Verify mode: Check every entry to see if it is correct. Files will be stored onto remote directories if necessary.

VerifyDF

VerifyDF attempts to answer the question: Does this DF file have entries for all the files I need to rebuild this program? VerifyDF will analyze a .Bcd file and compare the files recorded in the .Bcd with entries in a DF file. If a file entered in the .Bcd is not in the DF file, VerifyDF will give a warning message about the omitted file(s).

To the Simple Executive, type

```
> VerifyDF DFFile.DF Package.Bcd
```

where DFFile.DF is supposed to be a complete description of all the source files that were compiled and bound to produce "Package.Bcd", and all the object files produced while building "Package.Bcd". VerifyDF will analyze "Package.Bcd" and give warning messages about those files needed to build Package.Bcd not listed in DFFile.DF. If Package.Bcd was produced by the Binder, VerifyDF will complain if the .Config file or if any of the implementation modules listed in the .Config are not also listed in "DFFile.DF". If Package.Bcd was produced by the Compiler, VerifyDF will complain if the .Mesa source file or any of the Definitions Files (.Bcd's) it depends on are not listed. (A list of files omitted is written to the screen and to a file "MissingEntries.DFFile.DF$". Similarly, .Mesa, .Bcd, and .Config files that are not actually required to build "Package.Bcd" are listed.)

Once it has analyzed Package.Bcd, VerifyDF will then analyze any .Bcd's needed by Package.Bcd (described above). This analysis will continue until all .Bcd's that are referenced in the closure of the dependencies among .Bcd's are analyzed.

If the DF file is omitted, VerifyDF will put a list of the top-level dependencies onto "MissingEntries.DF$", which can then be given a more suitable name and VerifyDF can be invoked again. This is a simple way to bootstrap a complete DF file for a package.

VerifyDF will look on remote file servers for the correct version of a Bcd file and will use the Leaf protocol to read it page by page, so the Bcds do not necessarily have to be on the local disk for VerifyDF to do its job.

Other Information about VerifyDF:

The DF file that describes a package may optionally have entries preceded by a "+". If VerifyDF is invoked without a .Bcd file it will look inside the DF file and assume that any file name preceded by a "+" is a top-level Bcd and proceed to analyze it and all its dependencies, as described above. More than one .Bcd may be preceded by a "+" sign. For example, if DFFile.DF had an entry of the form

    +Package.Bcd

then

```
VerifyDF DFFile.DF
```

will analyze "Package.Bcd" and all its dependencies.

The DF file(s) involved do not need to be on the local disk. For example, if DFFile.DF were stored on [Indigo]<Cedar>Top>,

```
VerifyDF [Indigo]<Cedar>Top>DFFile.DF
```

is equivalent to retrieving DFFile.DF and invoking VerifyDF on the local copy.

VerifyDF has two options:

> When given /f, VerifyDF will write out a single DF file containing (1) the contents of the top-level DF file, and (2) the entries this top-level DF file refers to using **Imports** and **Include** statements. Thus /f "flattens" the top-level DF file. (This is useful for generating DF files to input to *ad hoc* DF parsers, such as the Cedar cross-reference program, etc.)
>
> When given /a, VerifyDF will retrieve any DF files referenced by the DF file being verified without asking for confirmation. If /a is not given, then the user will be asked to respond with either "y", "n", "a", "l", or "q". "y" means retrieve the DF file, "n" means skip the Df file, "a" is like "y" but subsequent DF files are retrieved as if "y" had been typed, "l" means use the copy of the Df file on the local disk in place of the version listed in the DF file, and "q" means stop running immediately.

VerifyDF listens for Control-DEL. When typed, VerifyDF will abort itself.

Summary of options to VerifyDF:

/a	Retrieve DF files to the local disk without confirmation..
/f	Write out a single DF file containing all the parts included and imported by the top-level DF file.

Summary of command lines for Pilot VerifyDF:

```
VerifyDF DFFile.DF Package.Bcd        -- analyze Package.Bcd
VerifyDF DFFile.DF                    -- analyze files preceded by "+"
VerifyDF Package.Bcd                  -- analyze Package.Bcd and construct a DF file
```

DFDisk

DFDisk will produce a file "Disk.DF" that describes an entire Pilot volume. It lists most of the files on the Pilot volume (except files ending in $ and a few kinds of log files). If present the RemoteFileName property is used to give a remote host and directory where the file may be found. Note reading leader pages is slow, even on a Dorado, so expect DFDisk to take about two minutes.

DFDisk is most useful when used to print out the RemoteFileName property and when trying to save all the files on your Client volume before you re-initialize the volume.

DFDisk listens for Control-DEL. When typed, DFDisk will abort itself.

DFDiff

DFDiff will take two files specified on the command line and compare them. If a file is present in the first but not the second, it says the file is deleted. If a file is present on the second but not the first, it says the file has been added. If the create dates differ for a file listed in both DF files DFDiff will print a message to that effect. For example, if OldPoplar.DF and NewPoplar.DF are respectively old and new versions of the Poplar system,

```
DFDiff OldPoplar.DF NewPoplar.DF
```

will compare the two.

This is most useful when used with DFDisk, before and after major changes, and if the file output by DFDisk (the first time it is run) is copied to some other file name like "SaveDisk.DF". Then

```
DFDisk
DFDiff SaveDisk.DF Disk.DF
```

will compare the two DF files and tell you what you've added or deleted.

DFDiff listens for Control-DEL. When typed, DFDisk will abort itself.

DeleteAll

The DeleteAll command will delete any file on the local disk that does not match one specified in an exception list. If invoked with no arguments, DeleteAll uses an exception list on BasicClientVolumeFiles.DF or BasicCoPilotVolumeFiles.DF (stored on [Indigo]<CedarLib>DFFiles), depending on which volume DeleteAll is run from. If DeleteAll is given the /n switch, it will skip the Basic...VolumeFiles.DF exception list. Users may themselves specify exception lists. The simplest way is to list the files you do NOT want deleted on the command line. If you have a file of filenames (one local filename per line) or a DF file you can use it as an exception list by typing its filename preceded by a '@ (i.e. quote, at-sign). For example,

```
DeleteAll /n Foo.Mesa '@Schmidt.DF '@OtherFiles.List
```

will delete all files on the volume except those specified in either the DF file or the other file list. In the case of DF files, DeleteAll will include the contents of nested DF files in the exception list.

DeleteAll will list each file not eliminated by an entry in the exception list and ask the user to type "y" or CR for yes, delete it, "n" for no, skip it (don't delete it), and "q" to quit. If DeleteAll is given the /a option it will delete all non-eliminated files without further confirmation. For example,

```
    DeleteAll /a '@Schmidt.DF
```

will delete all files not in the exception list on Basic...VolumeFiles.DF and not in the DF file Schmidt.DF. Use /a very CAREFULLY! Within CSL and ISL it is common practice to

```
    DeleteAll /a
```

on public Dorados to delete previous user's files or to cleanup after you are finished.

DFDelete

DFDelete is the reverse of DeleteAll. Given a list of DF files, DFDelete will read in all DF files (and nested ones too) and produce a list of all the filenames in the DF file that appear on the local disk (subject to two restrictions.) You may then execute the Delete.~ command to delete these files and free up the space they occupy on the local disk.

DFDelete will not add to the list of files you should delete

(1) any file that is marked **ReadOnly**, or any file appearing in a DF file nested in another where the outer DF file marked the inner DF file **ReadOnly** (as in VerifyDF)

(2) any file listed in the DF file with an explicit create-date whose copy on the local disk does NOT agree in create-date.

DFDelete takes one option (/r):

```
    DFDelete /r Package.DF
```

will list all the files in Package.DF including those marked **ReadOnly**.

RemoteDeleteAll

RemoteDeleteAll should be used to clean up old files from release directories. It chooses as a candidate for deletion any file that is on the remote directory but is not listed (with the same create-date) in a set of DF files. To do this, it takes a directory name and a list of DF files and then enumerates the remote directory. Two lists are written in files on the local disk: a list of files that will not be deleted and a list of candidates for deletion. It then lists the candidate files on the terminal and the user is given a chance to type "y" to delete it, "n" or "CR" to skip it (not delete it), and "q" to quit. For example,

```
    RemoteDeleteAll  [Ivy]<A>B>  [Ivy]<A>B>B.DF!5  [Ivy]<A>B>C.DF!6
```

would offer to delete all files on [Ivy]<A>B> that are not entered in the same name and create date as those listed in [Ivy]<A>B>B.DF!5 and [Ivy]<A>B>C.DF!6. Should a file listed in a DF file be listed without a create date (such as with ~= or >), then *no* versions of that file will be deleted.

The user may also enter "a" when asked whether to delete this file. When given "a," RemoteDeleteAll will ask if you are sure you want to go ahead and then delete the remaining candidate files without pausing for confirmation for each file.

ReleaseTool

The ReleaseTool should be run by the Cedar Release Master. Contact me for more information.

Other Information

You can find all the commands in Cedar .Bcd form on [Indigo]<Cedar>DFFiles>. All programs, except the RemoteGetTool, will run in the Cedar Viewers and Cedar UserExec world.

You may retrieve the newest released versions of these Ced^r DF programs by running

```
    BringOver /p [Indigo]<Cedar>Top>DFFiles.DF
```

followed by

```
    Run BringOver.Bcd
```

to load in the new version of BringOver.

Also, you may find newer, experimental versions on [Indigo]<PreCedar>Top>*.DF. Use these at your own risk!

All of these programs listen for control-DEL. When typed, they will abort themselves.

Should you need to supply a connect name and password, any of these programs will prompt you for them. When a CR for the connect name and a CR for the connect password are typed, these programs will ask whether you would like to login using different login credentials or abort the program.

If you get a message that begins with "Schmidt's Debugging:" you may safely ignore it-- these messages are usually due to certain objects in memory not being freed properly.

Cedar Release Procedures

[The following section is an edited version of a set of messages written by Roy Levin and sent to Cedar implementors in September 1981. Although the procedures described below are required of Cedar implementors, many of the extensions added to DF files will be helpful for non-Cedar users as well. Although it represents a restriction on DF file usage, I recommend non-Cedar users follow this style as much as possible.]

Background

Releasing large systems is a tricky business. SDD has known this for a long time and has developed elaborate release procedures to cope with the matter. We're only just discovering how messed up things can get. However, since we have many fewer human cycles to expend on this problem than SDD does, it is imperative that we devise natural, easy-to-use, and efficient (of people time) procedures for our releases of Cedar.

Release Procedures

The [Indigo]<Cedar> directory will be used exclusively for Cedar releases: that is, a file will appear on this directory if and only if it is a part of a (reasonably current) Cedar release. There will be a "release master" (initially Levin) who is responsible for insuring this invariant. Only the release master will be permitted to create or delete files on the release directory.

In the future, we expect to use system modelling facilities to describe and issue releases. Until the necessary software is in service, we will use DF files and a number of manually-executed conventions to achieve a similar effect. The procedures I will describe have the virtue that: (1) they place only a slight additional hurden on the individual implementor, (2) they enable the release master to insure the release invariant without building the release files from the original sources, and (3) they are amenable to substantial automation. The software needed to perform this automation is nearly complete.

Since the DF file is the sole descriptive mechanism presently at our disposal, we will use it in a stylized way as a "poor man's system model". Cedar implementors will observe the following conventions in producing components for release as part of the Cedar system:

(1) Each component will be described by a DF file, which may, in turn refer to other DF files. (Indeed, cross-reference is encouraged, as will be obvious shortly.)

(2) A DF file should mention (directly or indirectly) every file needed to build and use the component it describes. This requirement does not apply to the standard construction programs (PGS, Compiler, Binder, TableCompiler, Packager, MakeBoot) except in the DF files that define them, but such programs may be mentioned if circumstances seem to warrant. Thus, it should be possible to build the component by starting with a "bare disk" (i.e., one with nothing but the standard construction programs and BringOver), and, by applying BringOver to the DF file, acquire all the pieces necessary to build the component.

(3) All files that form an intrinsic part of this component (e.g., the .mesa and .config files and their derived .bcds) should appear in the DF file under **Directory** headings that are NOT **ReadOnly**. This applies also to miscellaneous command files useful in building the component as well as the DF file for the component itself. All other files mentioned in the DF file should appear under **ReadOnly Directory** headings. Thus, a **ReadOnly Directory** heading says, in effect, "the following files are 'imported' from other components", and a non-**ReadOnly Directory** heading says, "the following files comprise a part of this component".

(4) It is perfectly permissable to mention a superset of the files actually needed to build the component. For example, instead of painstakingly discovering all of the Pilot public definitions files needed by your program, you could simply include

Imports [Indigo]<Cedar>Pilot>PilotMesaPublic.df **Of** ~=

in the DF file. There are several convenient DF files of this sort available, e.g.,

[Indigo]<Cedar>Pilot>

PilotMesaPublic.df
PilotMesaFriends.df
ComSoftPublic.df

[Indigo]<Cedar>Tajo>

TajoPublic.df
TajoFriends.df

and others may appear soon.

(5) The "end result" of building a component is usually a single BCD that implements the component. You should use a program, named VerifyDF, which takes a BCD and a DF file and determines whether or not the latter mentions all the files necessary to build the former. For the purposes of checking the completeness of a release component, it is useful to have a DF file indicate the BCD(s) named within it that represent(s) the "end result". The DF file syntax will allow the character "+" to appear immediately before a file name. BringOver, SModel, etc. all ignore this character, but the release-checking software uses it to prepare the input for VerifyDF. All "top-level" BCDs in a DF file defining a release component should be marked with a "+" (more than one per DF file is permitted). The author/maintainer of a component should apply VerifyDF to it before submitting it to the release master.

(6) When the component has been built, the (non-**ReadOnly**) **Directory** headings in the DF file identify the location of the files intrinsic to the component. However, when the component is released, this location is changed to (an appropriate subdirectory of) the release directory. The DF

file syntax has been extended to enable **Directory** headings to identify both locations, e.g.,

```
Directory [Indigo]Levin>SpyBuild ReleaseAs [Indigo]<Cedar>Spy>
```

The release automation software uses this information to move the files under this heading to the indicated place on the release directory and to update DF files (including others that reference the original directory with **ReadOnly**) accordingly. The **ReleaseAs** clause is only appropriate for non-**ReadOnly Directory** headings, and all DF files involved in a release must have **ReleaseAs** clauses in all of their non-**ReadOnly Directory** headings.

(7) It is highly desirable to have a single DF file identify which of the files that comprise a release component are of interest to the client of the component and which files are of interest to the implementor. Several syntactic changes to DF files and the BringOver command line together provide the facilities to do this. They are best explained by an example. Consider the following fragment of a DF file for the Spy:

```
-- Spy.df
Exports [Ivy]<Levin>SpyBuild> ReleaseAs [Indigo]<Cedar>Top>
    Spy.bcd                             <date>
    SpyNub.bcd                          <date>
    SpyClient.bcd                       <date>
Directory [Ivy]<Levin>SpyBuild> ReleaseAs [Indigo]<Cedar>Spy>
    Spy.df                              <date>
    SpyImplA.bcd                        <date>
    SpyImplA.mesa                       <date>
    . . .
Imports [Indigo]<Cedar>CoPilot>CoPilot.df Of <date>
```

Now suppose that a user of the Spy wants to retrieve the public files of the Spy.
The user types:

```
BringOver /p [Ivy]<Levin>SpyBuild>Spy.df
```

The /p means "exports only" and will cause Spy.bcd, SpyNub.bcd, and SpyClient.bcd to be retrieved (assuming they aren't already on the local disk). These are the only files whose **Directory** headings include **Exports**. Since CoPilot.df is imported but not exported, neither it nor any of its contents are retrieved.

Suppose instead that the implementor wants to make a change to the Spy. He types:

```
BringOver [Ivy]<Levin>SpyBuild>Spy.df
```

Since no /p appears, all files in Spy.df will be retrieved (unless, of course, they are already present in correct versions on the local disk). This includes CoPilot.df and, since its contents are imported, BringOver will recur on the contents of this DF file. However (the nested instance of) BringOver will retrieve only the files in CoPilot.df that appear under **Exports** headings. Thus, the implementor of the Spy will get all of the Spy's files, but only the exported portions of CoPilot (which is appropriate since the Spy is a client of CoPilot). To summarize: **Exports** in place of a **Directory** heading identifies files that are of interest to the client, i.e., those files that will be retrieved by

BringOver /p. **Imports** causes BringOver to modify its behavior when recurring to retrieve only files identified as **Exports.** If /p is omitted, all files are retrieved, except that, once BringOver enters "exports only" mode, all nested retrievals of DF files will be exports only, regardless of the subsequent appearance of **Include** statements in nested DF files. This has the natural, intuitive behavior.

(8) As each implementor completes a component needed for an upcoming release, he/she sends a message to the release master containing the name of the DF file that defines the component. The implementor then must leave the component's constituent files undisturbed until the release master notifies him/her that the files have been safely moved to the release directory.

The following summary of qualifiers that appear in DF files may be helpful:

Qualifier	Interpreted by	Ignored by
ReadOnly	SModel, VerifyDF, ReleaseTool	BringOver
Exports	BringOver	SModel, VerifyDF, ReleaseTool
Imports	BringOver	SModel, VerifyDF, ReleaseTool
Include	BringOver, VerifyDF, ReleaseTool, SModel	
ReleaseAs	ReleaseTool	BringOver, VerifyDF, SModel

Guidelines for Date Specifications

Within a DF file that is intended to conform to the guidelines for release:

(a) files that are an intrinsic part of the component defined by the DF file (i.e., those appearing under non-**ReadOnly Directory** headings) should specify explicit dates,

(b) files which the component imports (i.e., those appearing under **ReadOnly Directory** headings or those imported DF files) should specify "#" (or equivalently "~=").

However, there is an important exception to (b). It occasionally happens that a single person is concurrently developing both the implementation of an interface and a client of the same interface. (Example: a collection of BCD manipulation facilities is imported by both the compiler and the binder. Ed Satterthwaite is the implementor of the BCD facilities as well as these two illustrious clients. When developing a new version of the BCD stuff, Ed frequently and rapidly switches between his roles as implementor-of-BCD-package and implementor-of-client-of-BCD-package.) In such situations, specifying "#" in the client's DF file will cause the wrong thing to happen; it was to accommodate this situation that the ">" form of date specification was originally added to DF files. Implementors who find themselves in this predicament should use ">" instead of "#" where necessary.

It should also be understood that when the ReleaseTool is constructing the DF files to be stored on the release directory, it will uniformly replace "#" and ">" specifications by explicit dates. This is important for a release, since it is intended to be a snapshot of the state of the files at some instant of time, and the loose binding implied by "#" and ">" is inappropriate. During the development process, however, these "dynamically bound" date specifications are very useful, and it is unreasonable and unnecessary to expect people to perform the manual translation to specific dates when release time arrives.

How, then, does this transformation of dates fit in with the ongoing development process? When an implementor finds it necessary to change a component following a release, he/she has the choice of starting from the DF file on the release directory or from the one used to build the component in preparation for the release. (Of course, the latter may validly disappear after the release; only the DF file on the release directory is guaranteed to be retained.) In the former case, the maintainer will have to edit the released DF file to make it convenient for subsequent development, e.g., by adjusting **Directory** headings to reference working directories and by replacing specific dates with "#" or ">". In the latter case, the DF file is already (still) suitable for development work. If a particular component is frequently changed and usually by the same person, the latter approach will likely be followed. If the component is infrequently modified or if many different people have occasion to change it, the former approach will be less error-prone. The choice should be made on a package-by-package basis.

Complete Syntax Description of DF Files

Directory *[host]<path>*
Exports *[host]<path>*
ReadOnly *[host]<path>*

Any of the above can be followed by **ReleaseAs** *[host]<path>* or **CameFrom** *[host]<path>*. **Exports**, **ReadOnly**, and **Directory** can be combined on one line. (**Public** is a synonym for **Exports**.)

filename ***<date>***

Multiple lines consisting of filenames and (optionally) create dates may follow **Exports**, **ReadOnly**, and **Directory** headings. *filename* may be preceded by "+", "@", and/ or {**PublicOnly**}. The *filename* may have a directory path (but no host), a short filename, possibly followed by a version number. The *<date>* can have the usual form, "~=", "#", ">", or be omitted.

Include *[host]<path>file.df!version* **Of** *<date>*

The **Include** statement may be followed by a **ReleaseAs** *[host]<path>* or **CameFrom** *[host]<path>* statement. The **Of** *<date>* is optional.

Imports *[host]<path>file.df!version* **Of** *<date>*
Using *[list of filenames, separated by commas]*

The **Imports** can be preceded by **Exports**, the **Of** *<date>* and **Using** [ ] can be omitted, and the **Using** clause may be preceded by a **CameFrom** statement.

Semantics of DF Files

All DF programs in this set use a consistent view of what a line in a DF file means. In summary, here is a list of ways to specify files in a DF file:

filename!vers ***<date>***

Any file on the remote directory with create time *<date>*. If necessary, Df programs like BringOver will enumerate the remote server looking for such a file. The version number *vers* (if present) is used as a hint.

filename!vers >

filename!vers #

filename!vers ~=

The highest version (!H). If ~= or # is given, a program like BringOver will retrieve *filename* from the remote directory if its create time differs from that of the file on the local disk. The version number *vers* (if present) is used as a hint. > is like ~= except that the file will be retrieved only if its create time is newer.

filename!vers

When no date is present, and a version # is given, then this refers to the file *filename* on the remote server with version number !*vers*.

filename

When no date and no version number is present, then it is treated like ~=. (Except by SModel, which fills in the local create time.)

The cases for DF files that are included or imported are exactly analogous.

For example,

Directory [Indigo]<Cedar>Top>

Spy.Bcd!2 1-Jan-82 11:11:11

Refers to a file created on 1 Jan 82. Version # is a hint.

Spy.Bcd!3 ~=

Refers to the highest version on the directory. The version # is ignored.

Spy.Bcd!4

Refers to version 4.

Spy.Bcd

Refers to the highest version on the directory.

Inter-Office Memorandum

To	Dorado users	Date	February 19, 1983
From	Ed Taft	Location	PARC/CSL
Subject	Dorado booting—operation (edition 5)	File	[Indigo]<DoradoDocs>DoradoBooting.press

XEROX

Dorado bootstrapping is a complex process, involving a number of hardware, microcode, and software components. This memo describes how to boot a Dorado and how to configure it to boot in certain ways. A companion memo, "Dorado booting—implementation", file [Indigo]<DoradoDocs>DoradoBootingImpl.press, is a comprehensive description of how it works and how the components are constructed and maintained.

1. Operation

Booting a Dorado from scratch involves two main activities: loading its control store with the correct *microcode*, and then forcing it to load and run the correct *software*. The descriptions here concentrate on the first step.

1.1 Emulators

The Dorado is able to execute programs written in a particular language (Mesa, Lisp, Smalltalk, or Alto BCPL) by virtue of running *emulator microcode*—that is, microcode for emulating that language. In addition, there is some common *I/O microcode* for controlling the I/O devices (display, disk, Ethernet, etc.) This microcode is not built into the Dorado but rather must be loaded into its control store from some external source.

The Dorado's control store is not large enough to hold the emulator microcode for all languages simultaneously. Therefore, the microcode is divided up into several independent microprograms, only one of which is used at a time.

Many microprograms include "Alto emulation" capability: each contains the Alto emulator and I/O microcode, and additionally contains the emulator for one other language. That is, the "Alto/Mesa microcode" consists of the Alto emulator, the I/O microcode, and the microcode for emulating the Mesa language; similarly for the "Lisp microcode" and the "Smalltalk microcode".

The "Cedar/Mesa" microprogram includes only the Cedar/Mesa emulator and Pilot I/O facilities, and does not include an Alto emulator.

1.2 Microcode booting

When the Dorado is first turned on, it is in a "shut down" state that minimizes power consumption. Pushing the boot button on the back of the keyboard initiates a bootstrap sequence that takes about one minute. (Most of this time is spent waiting for the disk to become ready.) If all goes well, at the end of this time, microcode is boot-loaded from some source (the disk or an Ethernet boot server), and further actions depend on what that microcode is.

If the Dorado is *already* on, this full bootstrap sequence can also be initiated by pressing the reset button on the front panel of the Dorado chassis (inside the cabinet, if the Dorado is cabinet-mounted). However, if the Dorado is *not* already on, i.e., is in the shut down state, pushing the reset button has no effect.

Assuming that the Dorado is already on, other forms of booting are initiated by pressing the boot button on the back of the keyboard, as detailed in the following paragraphs. There are single- and multiple-push boots. You must push the boot button for no more than 2.5 seconds; multiple pushes must be spaced no more than 1.5 seconds apart; and the bootstrap operation does not commence until 1.5 seconds after you release the button for the last time.

1-push boot. This causes the currently-running microcode to go through the software bootstrap sequence, i.e., loading the Alto OS, or the NetExec if you have the BS and Quote keys pressed down, or the physical volume boot file (Pilot or Othello) if you were running Pilot, or whatever. A 1-push boot is the nearest equivalent to pressing the boot button on an Alto. It works only if the correct emulator microcode is already loaded and the Dorado is still running normally.

2-push boot. This causes the currently-running microcode to be forcibly restarted. It is similar to a 1-push boot, but it will work even if the Dorado has halted, and it does more extensive microcode initialization (in particular, it zeroes main memory, and it reverts to the default disk partition for the current emulator.) This does require, however, that there be intact emulator microcode in the Dorado's control store.

3-push boot. This initiates a complete microcode bootstrap sequence, similar to the automatic power-on boot. This is the normal way to switch from one emulator to another (see below). It assumes nothing about the current state of the machine.

4, 5-, or 6-push boot. Same as 3-push boot.

7-push boot. This isn't really a boot: it causes the Dorado to shut down, a process that takes about 30 seconds. When the machine is in the shut down state, pushing the boot button (one or more times) will initiate a complete microcode bootstrap, as described at the beginning of this section.

If you have no keys pressed down during a power-on boot or a 3-push boot, and your Dorado's disk has Initial microcode installed on it, and the disk is on-line, then you get the installed Initial microcode; subsequent actions depend on what that microcode is (see the next section). If no Initial microcode is present or the disk is not on-line, you get the Alto/Mesa emulator.

Regardless of what is installed on the disk, you may bootstrap a specific emulator from an Ethernet boot server by holding a key down during the boot:

C	for DoradoCedar.eb (Pilot as modified for Cedar)
L	for DoradoLisp.eb
M	for DoradoMesa.eb (Alto/Mesa)
S	for DoradoSmalltalk.eb (Smalltalk-76)
T	for DoradoTest.eb (usually Cedar microcode under development)

For correct operation, you must hold the key down for at least 5 seconds after you initiate a 3-push boot.

These keys are independent of the ones used to select the desired *software* to run (with Alto-based emulators; see below). The latter keys are the same as on the Alto; that is, to run the NetExec, you hold down the BS and Quote keys while booting. Thus, to bootstrap the Smalltalk microcode and then run the NetExec, you might have to hold down S, BS, and Quote and initiate a 3-push boot. Since selecting this key combination would require you to have a pet octopus, some more convenient (though less mnemonic) alternate keys are defined for selecting the microcode to bootstrap. (In fact, these are the *only* microcode keys that work while BS is pressed down, because the normal keys form part of the Alto boot file number in this case.)

[	for Lisp
RETURN	for Alto/Mesa (same as all-keys-up boot)
SHIFT	for Smalltalk

1.3 Software booting

When Alto Emulator based microcode is loaded and started, it will then boot-load software from disk or Ethernet according to the standard Alto conventions. That is, if you are not holding down the BS key, the Alto Operating System will be booted from the disk; if you are holding down BS, a program will be booted from the Ethernet according to what other keys you are holding down simultaneously (e.g., BS and Quote boot the NetExec).

A Dorado's disk can have up to 5 *partitions* (complete self-contained Alto file systems), numbered 1 through 5. Ordinarily, when you boot Alto Emulator based microcode, it selects partition 5 by default. However, if you hold down a digit key in the range 1 through 5 while booting, the corresponding partition will be selected instead. Also, a 1-push boot of already-running Alto Emulator based microcode will not change partitions unless you are holding down a digit key.

When Pilot or Cedar microcode is loaded and started, it will boot-load software from the disk according to standard Pilot conventions. Briefly, the installed Pilot "germ" file is boot-loaded, which in turn loads the installed "physical volume boot file".

Note that Ethernet booting of *software* is possible only with Alto emulator-based microprograms, not with Pilot or Cedar. To boot Pilot software (e.g., Othello) from the Ethernet, you must first boot the NetExec using Alto/Mesa, then use the NetExec to call the MesaNetExec, then use the MesaNetExec to call the desired program.

1.4 Initial and Pilot microcode

There are two places on a Dorado's disk where you may install microcode. One is called the *Initial* region, which is a special region of the disk reserved for microcode. The other is called the *Pilot microcode* or *soft microcode* file, which is a more-or-less ordinary Pilot file which may reside in any Pilot volume. (If you don't have a Pilot file system on your disk then you cannot install Pilot microcode. There can actually be a Pilot microcode file in each volume; but the booting logic pays attention only to the one that has been declared to be the "physical volume Pilot microcode".)

The Initial microcode is what is initially loaded and started when you boot microcode from the disk, as described in section 1.2. You can choose from among many different pieces of microcode to install on the Initial region of your Dorado's disk. The most useful ones are the following.

DoradoCedar.eb, DoradoLisp.eb, DoradoMesa.eb, DoradoSmalltalk.eb. These are identical to the microcode that is booted from the Ethernet if you hold down C, L, M, or T while booting.

DoradoInitialDisk.eb. This microcode simply invokes the installed Pilot microcode file.

DoradoInitialCedar.eb, DoradoInitialLisp.eb, DoradoInitialMesa.eb, DoradoInitialSmalltalk.eb. Each of these is essentially a combination of DoradoInitialDisk.eb and a copy of a particular emulator. If booted with the P key depressed, it invokes the installed Pilot microcode file; otherwise it starts the emulator microcode that was part of the Initial file itself.

DoradoInitialEtherCedar.eb, DoradoInitialEtherLisp.eb, DoradoInitialEtherMesa.eb, DoradoInitialEtherSmalltalk.eb. Each of these is essentially a combination of DoradoInitialDisk.eb and an Ethernet microcode boot loader configured to boot a particular emulator. If booted with the P key depressed, it invokes the installed Pilot microcode file; otherwise it boots the appropriate microcode from an Ethernet boot server.

If you normally use only standard, released versions of emulators, you are advised to install only one of the last group of microcode files on the Initial region of your disk. That way, you are guaranteed always to get the most up-to-date version of microcode from a boot server, and you need not install new versions on your disk every time the microcode's maintainer fixes a bug or makes an upward-compatible improvement.

On the other hand, if you are running non-standard or pre-release versions of software that require incompatible microcode, you will find it convenient to install that microcode on your disk. Disk booting of emulator microcode should also prove useful during transitions between incompatible versions of microcode.

All microcode is kept on [Indigo]<Dorado>, and is copied to [Indigo]<Cedar>Top> during each Cedar release. Both Initial and Pilot microcode are installed using the Othello utility program. Note that even though Othello is a Pilot program, it can be used to install *any* Initial microcode, even if you don't have a Pilot file system on your disk; of course, you do have to have a Pilot file system in order to install Pilot microcode.

The following is a sample dialogue for installing Initial and Pilot microcode:

Get into the Alto NetExec somehow.

```
> MesaNetExecCR
```

Wait a few seconds for Ether booting.

```
> Switches: nCR              do this only if you don't have a Pilot file system
> OthelloDoradoCR
```

Wait a few seconds for Ether booting.

```
Othello 7.0 of 17-Jun-82 17:34:02
Please login...
User: xxxxxx  Password: xxxxxxCR...GV...OK
> Open IndigoCR
Indigo Parc IFS 1.36L, File Server of May 17, 1982
> Directory Cedar>TopCR  or Directory Dorado
> Initial Microcode Fetch
Drive name: RD0CR
File name: DoradoInitialMesa.ebCR
Are you sure? Yes
Fetching...done
> Pilot Microcode Fetch
Logical volume name: OthelloCR
Pilot microcode file name: DoradoCedar.ebCR
Fetching...installing...done
Shall I use this for the physical volume? Yes
>
```

1.5 Recommended microcode configurations

There are many possible ways of configuring the microcode on a Dorado's disk. The following configurations are suggested, though others are reasonable also.

For a Dorado running standard versions of Alto/Mesa, Lisp, or Smalltalk-76, and never running Pilot/Cedar:

Initial microcode: DoradoInitialEtherMesa.eb, DoradoInitialEtherLisp.eb, or DoradoInitialEtherSmalltalk.eb (Smalltalk-80 users should install DoradoInitialEtherMesa.eb, as the Smalltalk-80 system loads its own microcode)

Pilot microcode: none

For a Dorado running non-standard versions of Alto/Mesa, Lisp, or Smalltalk-76, and never running Pilot/Cedar:

Initial microcode: DoradoMesa.eb, DoradoLisp.eb, or DoradoSmalltalk.eb

Pilot microcode: none

For a Dorado running both an Alto-based emulator and Pilot/Cedar, and which is to boot the Alto-based emulator by default:

Initial microcode: DoradoInitialEtherMesa.eb, DoradoInitialEtherLisp.eb, or DoradoInitialEtherSmalltalk.eb

Pilot microcode: DoradoCedar.eb (invoked by booting with "P")

For a Dorado running Pilot/Cedar, and which is to boot Pilot/Cedar by default:

Initial microcode: DoradoInitialDisk.eb

Pilot microcode: DoradoCedar.eb

Regardless of what microcode is installed, booting with one of "C", "L", "M", "S", or "T" (or the alternate RETURN, SHIFT, or "]") invokes the microcode designated by the depressed key; and this microcode always comes from the Ethernet.

1.6 Microcode self-loading

The Dorado can also load its own microcode under program control, using a special instruction (LoadRam) described in section 2. There exists a program called LoadMB that runs on the Dorado (in Alto emulation mode) and loads microcode from a file on the Dorado's disk. The file must be in MB (micro binary) format, as produced by MicroD, and the microcode contained within that file must conform to certain conventions in order to be soft-bootable in this manner.

For example, the Mesa microcode is contained in a file called Mesa.MB. To load and start that microcode, type:

```
>LoadMB Mesa.mb
```

This loads the Dorado's control store with the contents of Mesa.mb and then starts it at a special entry point that causes it to reset I/O devices but does not disturb emulator execution or the contents of main memory.

This means of loading microcode isn't very interesting for bootstrapping the standard emulators, since the 3-push boot is easier and doesn't require the new microcode to be on your Dorado's disk. But it is useful for loading non-standard or experimental emulators. (The LoadMB program has some other features, described in section 3. The program is stored as [Indigo]<Dorado>LoadMB.run.)

1.7 Loading microcode using Midas

Each cluster of Dorados has an umbilical Alto that is able to control a connected Dorado for purposes such as hardware checkout and microcode debugging. The Alto program that performs these functions is called Midas.

Most Dorado users will have no occasion to use Midas. But if the normal microcode bootstrap mechanisms don't seem to be working, you can use Midas to load your desired emulator microcode.

If Midas is not already running, boot the Alto and type:

>Midas

Midas first displays a menu of the serial numbers of all the Dorados that are connected to this Alto. Select your Dorado by positioning the cursor over your Dorado's serial number and clicking RED (left or top mouse button). This step is skipped automatically if there is only one Dorado connected to the Alto.

If the selected Dorado is running, Midas displays a status of "Running" and puts up a menu (near the bottom of the screen) consisting of ABORT, DTACH, and EXIT items only. To halt the Dorado, select ABORT.

Now Midas displays a large menu near the bottom of the screen. Select RUN-PROGRAM, and the menu changes to a list of microprograms that you can run; these include CEDAR, LISP, MESA, and SMALLTALK. Select one of those. The screen then goes blank while Midas loads the Dorado's control store, a process that takes about a minute. After that, Midas starts the Dorado as if you had initiated a 2-push boot.

Important: *Do not leave Midas running on the Alto after you have loaded your microcode*, because multiple-push boots from the Dorado's keyboard don't work if Midas is controlling the Dorado. Therefore, exit Midas by selecting the EXIT menu item.

The versions of emulator microcode stored on the Midas Alto's disk might not always be up-to-date. If you are in doubt, update the microcode files by issuing the Executive command:

>@UpdateEmulators

which runs FTP to retrieve the current versions of all emulator microcode.

1.8 Machine status

There is a green light on the front panel of the Dorado that indicates the machine's present state, as follows:

1 flash. Bootstrap sequence in progress (power-on or 3-push boot).

2 flashes. Bootstrap sequence failed.

3 flashes. Transient power-supply problem—voltage or current was out-of-spec at least once since the last boot.

4 flashes. Present power-supply problem.

5 flashes. Dorado is in shut-down state (never booted, or shut down by 7-push boot).

6 flashes. Dorado has overheated and shut itself down automatically.

7 flashes. Midas has taken over control of the Dorado.

Continuously on. Bootstrap sequence is believed to have completed successfully, and there have been no occurrences of the conditions indicated by 3 through 7 flashes.

Continuously off. Unknown but probably serious malfunction, or AC power turned off.

Lupine User's Guide

An Introduction to Remote Procedure Calls in Cedar

Version 1.0

by

Bruce Nelson

and

Andrew Birrell

XEROX

Computer Science Laboratory
Palo Alto Research Center

July 8, 1982

[Indigo]<Cedar>Documentation>LupineUsersGuide.press

This document describes research software for Xerox internal use only.

Contents

1 Introduction

Lupine is the translator for Cedar's [Cedar] remote procedure call mechanism. A thorough understanding of remote procedure call (RPC) principles *is assumed* throughout this guide—indeed, once-skeptical readers have assured us it is foolish to continue without adequate background. Nelson's dissertation [RPC] provides more than the necessary foundation, as Lupine, not surprisingly, resembles that thesis's Emissary mechanism closely (Lupine, however, includes no orphan algorithms). Be particularly conversant with chapters 1 and 2, and section 5.4.

Relationship to Courier

Xerox's Office Products Division has its own RPC mechanism, Courier [CourierFS, CourierPS]. Courier is for use in standard Pilot [Pilot] environments and has its own Lupine-like translator, Diplomat [Diplomat]. The Courier system is supported by OPD and is released with Pilot; it is the product software RPC mechanism. Lupine, in contrast, is supported by CSL for use primarily within CSL and PARC. Lupine's remote calls execute in both the Cedar and Pilot environments, are about ten times faster than Courier's, and are significantly easier to program.

Operational Overview

Lupine reads a compiled Mesa interface and mechanically writes four Mesa source modules that implement the same interface remotely. For a given DEFINITIONS interface named *Target*, Lupine's stub control interface and three stub modules are:

> *TargetRpcControl.mesa.* This public definitions module contains declarations for the remote binding of *Target.*
>
> *TargetRpcClientImpl.mesa.* This program module executes in hosts that want to import *Target.* *TargetRpcClientImpl* is the client stub module. It exports *Target* and *TargetRpcControl.*
>
> *TargetRpcBinderImpl.mesa.* This optional program module is used to instantiate and import a dynamically varying number of remote *Target* interfaces. It exports *TargetRpcControl* but not *Target.* It depends on module *TargetRpcClientImpl.*

TargetRpcServerImpl.mesa. This program module executes in hosts that want to export *Target*. *TargetRpcServerImpl* is the server stub module. It imports *Target* and exports *TargetRpcControl*.

Appendix B contains an example of each of these generated modules. As you use this guide, refer to appendix B as necessary for concrete examples.

To use the stub modules compiled from the generated programs, clients include them in configurations on their client and server hosts. Aside from these configuration modifications, the only required program changes are two procedure calls to routines in *TargetRpcControl.* These binding calls, discussed in detail later, cause the serving machine to export *Target* and the client machine to import it. This scenario is slightly more complicated with secure calls or multiple instantiation, and much more complicated with dynamic crash recovery.

To establish bindings, make secure calls, and handle exceptions, Cedar RPC programmers use an *RPCRuntime* package interface called *RPC.* There is a parallel interface, *MesaRPC*, for standard Pilot programmers. Appendix A reproduces both of these interfaces.

Acknowledgments

Cedar's RPC mechanism is the work of two people: Andrew Birrell designed and wrote the RPCRuntime package with its transport, binding, and secure communication subsystems; Bruce Nelson designed the stub format and wrote the Lupine translator and diagnostic tools. In addition, members of the Alpine and Voice projects—Mark Brown, Dan Swinehart, and Ed Taft, in particular—made numerous good suggestions about parameter marshaling and multiple interface instances. Dan was also a revealing alpha tester.

This guide is primarily Nelson's venture, with Birrell contributing material on secure communication and remote binding.

2 Translator Operation

Installation

The Lupine translator runs in the Cedar environment (but a Tajo version would be trivial if needed). To install Lupine and all the necessary RPCRuntime underpinnings, use Bringover to retrieve the public components of [Indigo]<Cedar>Top>Lupine.df.

Translation

Lupine reads the BCD of a compiled interface and writes four output files. At present, Lupine is used from the Executive, although eventually it will be under the control of the Cedar system modeller. Establishing a command file to translate each remote interface, or package of interfaces, is recommended for now.

The basic procedure is this:

```
When your target interface is ready for remote use, compile it:
>Compile Target
Feed the resulting interface to Lupine:
>Lupine TranslateInterface["Target"]
Compile the four stub modules, in this order:
>Compile TargetRpcControl TargetRpcClientImpl,TargetRpcBinderImpl
  TargetRpcServerImpl
```

Lupine records its translation progress in the Executive's window and log. The source and object files for the stubs will be many times larger than the original interface, so be sure you have sufficient space for them.

To abort a translation just type control-DEL.

Commands and Syntax

Lupine can be invoked from the command line or used interactively from the keyboard. There is one real command, **TranslateInterface**, and a number of option-setting commands. They are presented here in procedural keyboard style. Consult a Lupine wizard before using a nondefault value for any option except **TargetLanguage**, **InterMdsCalls**, **InlineDispatcherStubs**, and **LineLength**.

<>**TranslateInterface** [fileName: STRING]

> Lupine processes **fileName**.bcd, e.g., *Target*.bcd.

<>**TargetLanguage** [language: STRING ← "Cedar"]

> **TargetLanguage** sets the programming language used to write the output files (do not confuse the **TargetLanguage** command with the use of *Target* as an example interface). The acceptable languages are Cedar and Mesa; Cedar is the default.

<>**DefaultParameterPassing** [method: STRING ← "VALUE"]

> This command sets the default parameter passing method to be VAR, VALUE, RESULT, or HANDLE. (These are explained in a later section on Parameter Passing Methods.) VALUE is the default, and take heed that this is different from Mesa's default semantics for address-containing values.

<>**InterMdsCalls** [yes: BOOLEAN ← FALSE]

> When **InterMdsCalls** is true, the importers and exporters of *Target* must operate within the same host (the remote binder insures this), but can execute in different MDSs. This tighter binding yields much faster parameter marshaling for address-containing parameters. Note, however, that intraMDS addresses such as POINTERs and STRINGs are necessarily illegal in interMDS calls, and Lupine issues warnings if they are used. The default is false.

<>**InlineDispatcherStubs** [yes: BOOLEAN ← TRUE]

> Lupine usually writes a single large procedure in the server stub program. For large or complicated remote interfaces, however, the compiler can overflow when it compiles this huge dispatcher procedure. If this happens, setting

InlineDispatcherStubs to false may cure the overflow problem by writing many small procedures instead. If not, there two distasteful alternatives: either simplify the troublesome interface, or split it into two smaller interfaces. The default is true.

<>FreeServerArguments [yes: BOOLEAN ← TRUE]

When FreeServerArguments is true, Lupine automatically deallocates any noncollected storage that it allocates to unmarshal arguments in the server stubs. When false, it does not deallocate this storage, presumably because the underlying server implementation is going to assume responsibility for deallocation itself. This unSAFE feature is only for Mesa—not Cedar—RPC interfaces that use this contorted allocation strategy. It applies only to arguments: storage for results is never deallocated in either the client or the server. The default is true.

<>DynamicHeapNews [heap, mds: INTEGER ← 50, 50]

A Mesa remote interface can have parameters with a dynamic number of noncollected objects (i.e., objects NEWed from the heap, such as a descriptor of strings or a sequence of pointers). Lupine imposes a translation-time upper bound on the total number of these dynamic objects allowed in the argument list—not result list—of a single call. The default is 50 heap (long pointer) objects and 50 mds (short pointer) objects.

<>LineLength [length: INTEGER ← 82]

LineLength specifies the approximate maximum length of source lines in the generated output files. The default, 82, is a good choice for Viewers and Tajo.

There are additional debugging commands for wizards only. Type ? at any point for a list of possible commands or responses.

Command Files

The following example illustrates option-setting and translation in command files:

```
>Lupine DefaultParameterPassing["VAR"] InlineDispatcherStubs[FALSE]
  LineLength[60] TranslateInterface["Target"]
```

Lupine Errors

Fortunately, because Lupine's input is compiler output, few translation errors are possible. Most errors are caused by parameter passing restrictions. Lupine reports errors both in the typescript and in the output file itself. Error and warning messages in the typescript include the output file name and character position of the error, and messages in the output file are preceded with "#####" to make them easy to spot. Many errors cause an ERROR statement to be inserted in the output file. This prevents a bogus stub from executing when the result would be chaos. If you can't understand a particular error message, see a Lupine wizard.

As mentioned above, if a set of stubs is too big to compile, the InlineDispatcherStubs option may remedy the problem.

Compiler Warnings

The compiler will occasionally issue warning messages about referenced but unused DIRECTORY items as it compiles stub programs. Lupine is a mechanical translator, and eliminating the spurious warning., would be difficult. Live with them.

3 Conversations, Encryption, and Secure Communication

Lupine and the RPC runtime provide facilities for making secure remote calls. Such calls are encrypted using the DES encryption algorithm [DES]. The protocols used provide secure two-way authentication without any passwords being passed in the clear [Needham-Schroeder]. Secure RPC calls prevent an intruder from seeing the procedure names, parameters, or results of the calls; from modifying calls in transit; from replaying previously valid calls; from observing passwords (encryption keys) in transit; from observing the cipher text of known (or chosen) plain text.

Secure RPC communication is based on the concept of a *conversation.* A conversation takes place between two authenticated Grapevine individuals, known as the *originator* and the *responder* of the conversation.

```
RPC.Principal: TYPE = Rope.ROPE;

RPC.EncryptionKey: TYPE = BodyDefs.Password;

RPC.Conversation:TYPE = . . . ;

RPC.SecurityLevel = {
    none,
    authOnly,
    ECB,
    CBC,
    CBCCheck};

RPC.GenerateConversation: PROC RETURNS[Conversation];

RPC.StartConversation: PROC [
    caller: Principal,
    key: EncryptionKey,
    callee: Principal,
    level: SecurityLevel[authOnly..CBCCheck] ]
    RETURNS[Conversation];

RPC.EndConversation: PROC [conversation: Conversation];

RPC.GetCaller: PROC [conversation: Conversation] RETURNS [Principal];

RPC.GetLevel: PROC [conversation: Conversation] RETURNS [SecurityLevel];

RPC.ConversationID: TYPE [3];

RPC.GetConversationID: PROC[conversation: Conversation]
    RETURNS [id: ConversationID];

RPC.AuthenticateFailed: ERROR [why: RPC.AuthenticateFailure];
```

A conversation is created by the originator calling *RPC.StartConversation* and is destroyed by the originator calling *RPC.EndConversation.* The originator provides his own name and password and the name of the responder. *StartConversation* communicates with Grapevine to verify the validity of these arguments; at this time there is no communication with the responder. See below for a description of the *SecurityLevel* of a conversation. **At present, the communication with Grapevine does not use secure protocols.**

Having created the conversation, the originator may now make secure calls to the responder by passing the *Conversation* as the first argument of any remote call to a procedure in an interface that is exported by the responder. (The same conversation may be used to make calls in multiple interfaces.) The RPC runtime guarantees that the call can be understood only inside a host containing the authenticated responder and that the call will be received by the responder at most once. The responder can determine the authenticated identity of the originator by calling *RPC.GetCaller* when he receives a call having the *Conversation* as its first argument. Every conversation is uniquely identified for all time by an identifier that either party may obtain by calling *RPC.GetConversationID.* Calling *RPC.StartConversation* may raise the exception *AuthenticateFailed*, with the following values for its argument:

communications Unable to contact Grapevine.

badCaller The originator's name is not a Grapevine individual.

badKey The originator's password was incorrect.

badCallee The responder's name is not a Grapevine individual.

The procedure *GenerateConversation* may be called to obtain a *Conversation* to be used solely for calls within a single machine; such a *Conversation* must not be used for remote calls.

The *SecurityLevel* passed to *StartConversation* indicates how much security the originator wants for this conversation. The *SecurityLevel* of any conversation may be obtained by calling *GetLevel.* The meanings of the conversation levels are as follows. *none* indicates that the conversation was created by calling *GenerateConversation*, and that no authentication or encryption has been employed. *authOnly* will perform the authentication, but calls are not encrypted, and modification, manufacture or replay of calls or results is not detected. *ECB* uses the ECB mode of DES [DES], which protects calls and results from eavesdropping (and makes modifying, manufacturing or replaying calls or results somewhat more difficult). ECB mode does not hide patterns of repeated data within or between calls. *CBC* uses the CBC mode of DES [DES],, which behaves like ECB except that it hides repeated data patterns. *CBCCheck* is like *CBC*, with the addition of a checksum on each call and result; this prevents any eavesdropping, modification, manufacture or replay of a call or result. *CBCCheck* should normally be used for secure communication, and *authOnly* should normally be used for applications where only authantication is desired; the other values of *SecurityLevel* are advisable only where hardware or performance constraints dictate. **At present the DES algorithm is replaced by a trivial exclusive-or algorithm, because of the absence of encryption hardware.**

4 Binding and Configuration

Lupine does not perform remote binding by using remote versions of the modeller or C/Mesa binder [Mesa]. Instead, the modeller or binder are used to bind—locally—to an instance of *TargetRpcControl.* A remote interface, say *Target,* is then bound remotely by calling binding procedures in the *TargetRpcControl* interface. These procedures use the RPCRuntime, in cooperation with Grapevine [Grapevine], to perform remote binding. This is elaborated below, after a discussion of remote interface naming.

Interface Names, Instances, and Versions

Naming remote interfaces is much more complicated than naming local-machine interfaces because remote interfaces must be uniquely identifiable in a distributed environment. Furthermore, ability to control the degree of typechecking is required for widespread systems such as Laurel (see sections 4.1.3.1 and 4.1.3.4 in Nelson's thesis [RPC]). Lupine addresses this problem with *InterfaceNames,* defined below. (*ShortROPEs* are discussed in the Parameter Passing section.)

```
RPC.ShortROPE: TYPE = Rope.ROPE;
RPC.VersionRange: TYPE = RECORD [first, last: CARDINAL];
RPC.matchAllVersions: VersionRange = . . .;
RPC.InterfaceName: RECORD [
    type: ShortROPE ← NIL,
    instance: ShortROPE ← NIL,
    version: VersionRange ← matchAllVersions ];
RPC.defaultInterfaceName: InterfaceName = [];
```

InterfaceName.type

The *type* in an interface name specifies which interface is being exported or imported. This is not a precisely defined concept, since it is up to the implementors of interfaces to separate their interfaces into types, but it is intended to correspond to a single Mesa definitions module. Only the implementor can decide at what stage in the incremental development of an interface it should be considered as a new *type.* In the distributed environment, there may be multiple exporters of a given *type* of interface. For example, if the interface is for access to a printer, each printer would export the same *type* of interface. These multiple interfaces are distinguished by their *InterfaceName.instance,* described below.

The *type* of an interface may be a Grapevine *R-Name,* or an arbitrary string chosen by the implementor. One approach is to use the corresponding definitions module name as the *type.* The effects of using these forms of *type* are described below.

InterfaceName.instance

The *instance* in an interface name specifies which particu'ar one of the (potentially) many instances of this interface is being named. For example, if the interface is for access to a printer and is exported by each printer, each printer would export the same interface *type* but a different interface *instance.*

The *instance* of an interface may be a Grapevine *R-Name,* or a host name (for example, "Ivy"), or a Pup network address of the exporting host (for example, "3#17#"). The effects of using these forms of *instance* are described below.

Multiple remotely exported interfaces may have the same *instance*, provided that they each have a different *type* and they are exported from the same host.

InterfaceName.version

The *version* in an interface name may be used to control the extent to which the remote binder will check compatibility of versions between the exporter and importer. If the exporter of a remote interface specifies [*a,b*] as the *InterfaceName.version* and the importer specifies [*c,d*], then the remote binding will be allowed to succeed only if the ranges [*a..b*] and [*c..d*] have a non-empty intersection. However, *RPC.matchAllVersions* is assumed to intersect with every version range. That is, if *RPC.matchAllVersions* is specified by either the importer or the exporter, then the remote binding will not fail because of version mismatch.

By using this facility, the implementors are asserting that they support, at a bit-wise compatible level, multiple versions of an interface. This freedom is in marked contrast to the Mesa binder (and loader), which insist on identity of versions for binding to succeed. This extra freedom seems necessary in the distributed environment of RPC, but may well be a fertile source of bugs.

To remain upward compatible with previous versions of an interface, you may add procedures (or signals or errors) only at the end of the interface. You must not make any significant changes to any types (for example, changing the order of fields in a record or of parameters to a procedure is significant, as is adding fields or parameters). You may change the names of things (like procedures, fields, parameters). You may also use this *version* field to indicate semantic changes that do not involve changes to the definitions file.

Exporting

For interface *Target*, calling *TargetRpcControl.ExportInterface* causes *Target* (as specified by *interfaceName*) to be exported for immediate remote use. Any attempts to import *Target* before *ExportInterface* completes, or after *UnexportInterface* completes, will raise an exception (see the description of importing, below). These two routines must be called in the strict sequence *ExportInterface*, *UnexportInterface*, *ExportInterface*, ... Calling *UnexportInterface* has no effect on calls in progress.

```
TargetRpcControl.ExportInterface: PROCEDURE [
   interfaceName: InterfaceName ← defaultInterfaceName,
   user: RPC.Principal,
   password: RPC.EncryptionKey,
   parameterStorage: Zones ← standardZones ];

TargetRpcControl.UnexportInterface: PROCEDURE;

RPC.ExportFailed: ERROR [why: RPC.ExportFailure];
```

The precise behavior of *ExportInterface* depends on the value of *interfaceName*. If the *interfaceName.instance* is a Grapevine *R-Name*, the Grapevine registration service is consulted to ensure that the *instance* is an individual whose *connect-site* is a Pup net address for the exporting machine. If necessary, the database is updated, using the given *user* and *password* as credentials. If the *instance* is not an *R-Name* (that is, it does not contain a dot), it should be either a Pup Name Lookup Server name that maps to the exporting machine, or a Pup net address constant for the exporting machine. If the *interfaceName.type* is an *R-Name*, the Grapevine database is consulted (and updated if necessary) to ensure that the *type* is a group, one of whose members is the *interfaceName.instance*. **At present, these actions on an *interfaceName.type* which is an *R-Name* are not implemented; every *type* is treated as just a string.**

If you call *ExportInterface* and default the *interfaceName.type*, Lupine will use for the *type* a combination of the interface name (for example, "Target") and the compiler's version stamp for the interface. The default *type* is thus a unique name for the remote interface, and this uniqueness permits the default *version* to be *matchAllVersions* — of which there can be just one.

Calling *ExportInterface* with a defaulted *interfaceName.instance* is not yet supported; exporters and importers must give an explicit *instance* each time. For simple experiments, it is convenient to use your own Grapevine *R-Name* as the *instance*, and your own credentials as parameters to *ExportInterface*. For example:

```
myInterface: InterfaceName = [instance: "Nelson.pa"]; -- Use default type & version.

TargetRpcControl.ExportInterface [
   interfaceName: myInterface,
   user: "Nelson.pa",
   password: Rpc.MakeKey[ "Nelson's password"] ];
```

If an attempt to export a remote interface fails, the exception *ExportFailed* is raised. No monitors are locked when this signal is raised, so further calls into the RPC runtime may be made from catch-phrases. The parameter of the signal is one of the following:

communications Exporting failed because of inability to contact Grapevine or a Pup Name Lookup server.

badType The *interfaceName.type* was illegal (a Grapevine *R-Name*, but not a group).

badInstance The *interfaceName.instance* was illegal (NIL, not a Grapevine *R-Name*, not an individual, not a Name Lookup server name for this host, not a net address constant for this host).

badVersion The *interfaceName.version* was illegal (upper bound less than lower bound).

tooMany Too many current exports for local tables — consult a Lupine wizard.

badCredentials The credentials were inadequate for updating the Grapevine database.

Importing

Remote importing of an interface *Target* may be caused by calling *TargetRPCControl.ImportInterface* or *TargetRPCControl.ImportNewInterface*, as described below. In either case, an *interfaceName* is provided, which is interpreted as follows.

If *interfaceName.instance* is not NIL, then it must be: a Grapevine *R-Name* that is an individual whose *connect-site* is a Pup net address, a Pup Name Lookup server host name, or a Pup net address constant. Each of these maps to a net address of a machine, which is assumed to be the machine that has previously exported the desired interface. If that machine is running the Cedar RPC package and has exported an interface whose *InterfaceName* has the same *type* and *instance*, and whose *version* is compatible with the importer's *interfaceName.version*, then the import succeeds and the importer's stub is now bound to the exporter's stub.

If *interfaceName.instance* is NIL, then the remote binder will attempt to find any exported interface whose *type* and *version* match those given in *interfaceName*. If the *type* is a Grapevine *R-Name*, then it should specify a Grapevine group. The binder enumerates the members of that group, obtains the network addresses that are the connect-sites of the members, and tries to bind to them in turn until one succeeds. The order of these binding attempts is undefined, but is closely related to hop-count and responsiveness. The algorithm usually chooses the nearest instance of the type that is operational. If the *type* is a Pup Name Lookup server name string, then this algorithm is performed with the network addresses that are the value of that name. **Importing with *interfaceName.instance* = NIL is not yet implemented.**

If an attempt to import a remote interface fails, the ERROR *ImportFailed* is raised. No monitors are locked when this signal is raised, so further calls into the RPC runtime may be made from catch-phrases. The parameter of the signal is one of the following:

communications Importing failed because of inability to contact one of: Grapevine, a Pup Name Lookup server, the exporting host.

badType The *interfaceName.type* was illegal (*interfaceName.instance* is NIL and *type* is not a Grapevine *R-Name* of a group, and not a Name Lookup server name).

badInstance The *interfaceName.instance* was illegal (not a Grapevine *R-Name*, not an individual, not a Name Lookup server name, not a net address constant).

badVersion The *interfaceName.version* was illegal (upper bound less than lower bound).

wrongVersion The exporter and importer version ranges don't overlap.

unbound No currently running host has exported the appropriate *type* and *instance*.

stubProtocol The exporter and importer are using stub modules produced by incompatible verisons of Lupine.

Importing One Remote Instance

For interface *Target*, calling *TargetRpcControl.ImportInterface* causes *Target* (as specified by *interfaceName*) to be remotely imported for immediate use. In the C/Mesa sense, *Target* was bound when *TargetRpcClientImpl*, which exports *Target* locally, was bound. But *remote* binding is not complete until *ImportInterface* is called. As a consequence, any attempts to use operations in

Target before *ImportInterface* completes, or after *UnimportInterface* completes, will fail unrecoverably. Calling *UnimportInterface* has no effect on calls in progress. These two routines must be called in the strict sequence *ImportInterface, UnimportInterface, ImportInterface,* .... Note that *ImportInterface*[*someInterface*] will raise the exception *RPC.ImportFailed*[*unbound*] unless some host has previously called *ExportInterface*[*someInterface*].

```
TargetRpcControl.ImportInterface: PROCEDURE [
   interfaceName: InterfaceName ← defaultInterfaceName,
   parameterStorage: Zones ← standardZones ];

TargetRpcControl.UnimportInterface: PROCEDURE;

RPC.ImportFailed: ERROR [why: {. . .}];
```

Importing Multiple Remote Instances

For interface *Target*, calling *TargetRpcControl.ImportNewInterface* causes a new instance of *Target* (as specified by *interfaceName*) to be created and remotely imported for immediate use. Typically, each call of *ImportNewInterface* will specify a different value for *interfaceName.instance*, thus binding to a different, distributed instance of the *Target* interface. The operations in this new instance must be accessed through the *interfaceRecord* that *ImportNewInterface* returns, *not* through *Target* itself, as in the static importing case discussed above. (The *Target* interface is useless to RPC programmers using multiple instances and should not even be referenced.) For example, if *Target* contains procedure *WritePages*, a call to *WritePages* must be written *interfaceRecord.WritePages*[. . .], not *Target.WritePages*[. . .]. Any attempts to use operations in *interfaceRecord* before *ImportNewInterface* completes, or after *UnimportNewInterface* completes, will fail unrecoverably. Calling *UnimportNewInterface* has no effect on calls in progress. *UnimportNewInterface* is available only for the Mesa target environment. In Cedar, discarded remote interfaces are garbage collected. Note that *ImportNewInterface*[*someInterface*] will fail unless some host has previously called *ExportInterface*[*someInterface*].

```
TargetRpcControl.InterfaceRecord: TYPE = REF RECORD [. . .];

TargetRpcControl.ImportNewInterface: PROCEDURE [
   interfaceName: InterfaceName ← defaultInterfaceName,
   parameterStorage: Zones ← standardZones ]
   RETURNS [interfaceRecord: InterfaceRecord];

TargetRpcControl.UnimportNewInterface: PROCEDURE [
   interfaceRecord: InterfaceRecord ]
   RETURNS [nilInterface: InterfaceRecord ← NIL]; -- Only for Mesa, not Cedar.

RPC.ImportFailed: ERROR [why: {. . .}];
```

Importing Multiple Local Instances

When a distributed program deals with multiple remote instances, it is often true that one (or more) of the instances will actually reside on the local host. In this case, performing remote calls to the local instance can have an undesirable performance impact. RPC programmers who want to optimize "remote" calls to local dynamic instances can do so in a simple and transparent fashion:

Copy and rename *TargetRpcBinderImpl* to be *TargetLocalBinderImpl.*

Edit *TargetLocalBinderImpl*, removing all the RPCLupine calls and changing *ImportNewInterface* to NEW *TargetImpl* and not *TargetRpcClientImpl.*

TargetRpcControl.ImportNewInterface—as implemented by *TargetLocalBinderImpl*—now returns an *interfaceRecord* for a local instance, not a remote one. The binding and communication optimizations are transparent to users of *TargetRpcControl* and *interfaceRecord.* See a Lupine wizard for more information.

Parameter Storage Zones

To unmarshal certain parameters, Lupine's stubs must allocate private storage. This storage is taken from zones that RPC programmers can specify at binding time. For simple interfaces a set of standard zones is adequate, and thus the binding operations have default *standardZones* (below). For production interfaces that use address-containing (AC) parameters frequently, however, establishing special-purpose zones can increase performance significantly. If a remote interface uses a large number of small AC objects (except for ropes, which get special treatment from the Rope package), or it uses any large AC objects, consult a Lupine wizard.

```
RPC.Zones: RECORD [
   gc: ZONE ← NIL,
   heap: UNCOUNTED ZONE ← NIL,
   mds: MDSZone ← NIL ];
RPC.standardZones: Zones = [];
```

Module Dependencies

The compile-time relationships of *Target* modules, Lupine's stub modules, and the RPCRuntime packages are shown below. If module *B* is to the right of module *A*, then *B* depends on *A*.

```
RPC or MesaRPC
    RPCLupine
        LupineRuntime
            RPCRuntime package
    Target
        Target's clients
        TargetImpl
        TargetRpcControl
            Target's remote binding modules
            LupineRuntime (repeated from above)
                TargetRpcClientImpl
                    TargetRpcBinderImpl
                TargetRpcServerImpl
```

DF Files for Remote Interfaces

As a general rule, Lupine.df exports all the Lupine, LupineRuntime, RPCRuntime, and Cedar runtime components needed by remote programs. Thus DF files describing remote applications should say [Exports] Imports Lupine.df to get all required components, although tailoring with a

USING list may be worthwhile.

Configurations Using Remote Interfaces

Remote calls typically take place between independent hosts, and thus, for a given *Target* interface, *Target's* client and implementation modules, along with the necessary stubs, must be split between the cooperating hosts. A simple division of the *Target* service into remote *TargetClient* and *TargetServer* C/Mesa configurations is illustrated below. Constructing corresponding Cedar models would be similar.

```
TargetClient: CONFIGURATION
    IMPORTS
      -- For Lupine-generated modules:
         Atom, ConvertUnsafe, Heap, Inline, LupineRuntime, Rope, RopeInline,
         RPCLupine, SafeStorage, UnsafeStorage,
      -- For ClientRpcImportImpl and TargetClientImpl:
         . . .
    CONTROL ClientRpcImportImpl, TargetClientImpl
    = BEGIN
    TargetRpcClientImpl;  -- EXPORTS Target, TargetRpcControl.
    TargetRpcBinderImpl;  -- Optional, for multiple instances via ImportNewInterface.
    ClientRpcImportImpl;  -- IMPORTS TargetRpcControl, calls Import(New)Interface.
    TargetClientImpl;  -- IMPORTS Target (one instance) or uses interfaceRecord (for multiple).
    END;

TargetServer: CONFIGURATION
    IMPORTS
      -- For Lupine-generated modules:
         Atom, ConvertUnsafe, Heap, Inline, LupineRuntime, Rope, RopeInline,
         RPCLupine, SafeStorage, UnsafeStorage,
      -- For ServerRpcExportImpl and TargetImpl:
         . . .
    CONTROL ServerRpcExportImpl
    = BEGIN
    TargetRpcServerImpl;  -- IMPORTS Target, EXPORTS TargetRpcControl.
    ServerRpcExportImpl;  -- IMPORTS TargetRpcControl, calls ExportInterface.
    TargetImpl;  -- EXPORTS Target.
    END;
```

The purpose of the user-written *ClientRpcImportImpl* and *ServerRpcExportImpl* modules is to import and export the remote *Target* interface by making calls on *TargetRpcControl's Import(New)Interface* and *ExportInterface* routines. These trivial binding operations could just as well be placed in *TargetClientImpl* and *TargetImpl*, or combined with the binding operations of other remote interfaces into one master binding module. Note that these configurations do not include the RPC runtime package, *RPCRuntime.bcd*. This package should normally be loaded separately, because it is essential that only one copy of this package is loaded into any machine. Check with current Cedar release documentation to see whether *RPCRuntime.bcd* is included in the Cedar boot file.

It is sometimes advantageous to have both client and server configurations loaded into one host, as for single-machine debugging. Such configurations must be specified cautiously, however, so that remote imports and exports are not misconnected locally, short circuiting all remote calls. The easiest way to do this is to use nested configurations, as in the following example.

```
TargetTestCombined: CONFIGURATION
  IMPORTS
   -- For Lupine:
      Atom, ConvertUnsafe, Heap, Inline, LupineRuntime, Rope, RopeInline,
      RPCLupine, SafeStorage, UnsafeStorage,
   -- For TargetImpl, ClientImpl:
      . . .
  CONTROL TargetServer, TargetClient
  = BEGIN
  TargetClient;
  TargetServer;
  END;
```

These *TargetClient* and *TargetServer* examples are simple; more elaborate configurations will be commonly required. See appendix A of the Diplomat document [Diplomat] for further, more complicated, examples, or consult a Lupine wizard.

5 Crash Detection and Recovery

A general discussion of crashes and crash recovery in the context of RPC is outside the scope of this guide. The main reason is insufficient experience with distributed systems: there is no widespread agreement on an RPC-level crash recovery methodology, or, indeed, if such a methodology is even needed or possible at this level. In currently operational distributed systems, those with the best crash behavior appear to use highly application-specific techniques. See Nelson's thesis [RPC] for more discussion (sections 4.1.1, 4.2.3, 5.1, 7.2).

RPC.CallFailed: SIGNAL [*why: RPC.CallFailure*];

Lupine's RPCRuntime uses the *RPC.CallFailed* exception to report crashes and other remote call execution problems. This exception can be reported by *any* invocation of a remote procedure, or on any signal of a remote signal or error. RPC programmers interested in continuous service must therefore be prepared to catch *CallFailed* on each remote operation. Recovery actions usually consist of restoring some invariants before trying to rebind to another instance of the remote interface. No monitors are locked when *RPC.CallFailed* is raised, so further calls into the RPC runtime may be made from catch-phrases. The parameter of the signal is one of the following:

timeout The other participant in the call is not responding to attempts to communicate. The remote call may or may not have taken place. Note that this is not raised merely because the remote call takes a long time to execute. The RPC runtime places no limit on the execution time of a remote call, provided the server still responds to probes. A client can terminate a call that is taking too long by using the *Abort* mechanism of the Mesa process machinery. *RPC.CallFailed* may be resumed iff its parameter is *timeout*.

unbound The server has un3exported this interface, or has been restarted, since the interface was imported. The call did not take place.

busy After repeated attempts, the server is too busy to accept the call. The call did not take place.

protocol The caller and server runtimes disagree about the communication protocol; consult a Lupine wizard.

stubProtocol The caller and server stubs disagree about the communication protocol; consult a Lupine wizard.

Note for experts: In servers, the RPCRuntime is prepared to catch *CallFailed* itself and abort the incoming call that caused it. Thus if *CallFailed* is signalled out of a remote exception or a callback (procedure parameter), and the exception or procedure was invoked by a remote call executing in a server, the RPC programmer's server code does not *have* to catch *CallFailure*. The RPCRuntime will automatically clean up and ignore the original call, if desired.

6 Parameter Passing

Semantics

For parameters that do not contain addresses, Lupine's parameter passing semantics are exactly those of Cedar and Mesa: call-by-value. For address-containing (AC) parameters, however, Lupine has four passing semantics: for arguments, VAR, VALUE, RESULT, and HANDLE; for results, just VALUE. Each of these is a copy semantics; Lupine does not support reference semantics (call-by-reference) between disjoint address spaces. Here is a brief description of each; the next secion has examples:

VALUE. Value semantics cause the actual argument to be evaluated and assigned (copied) to the formal argument at the time of invocation. For AC types used as RPC parameters, the referent and not the address is used. For example, if *p* is a REF INT, then passing *p* by VALUE uses the integer *p*↑, not the pointer *p* (HANDLE passing would use just *p*). NIL semantics are preserved—e.g., if *p* is NIL, then NIL is passed. This dereferencing behavior also applies to POINTERs (same as REFs), DESCRIPTORs (the underlying array is copied), LISTs (the individual elements are copied to a new list), and strings (ropes, atoms, and strings).

RESULT. Result semantics cause the value of the formal AC argument to be assigned to the actual argument at the completion of the call. Arguments are dereferenced as described above. The initial value of the actual argument is unspecified.

VAR. VAR means VALUE-and-RESULT: the actual argument is evaluated and assigned to the formal argument at the start of the call, and the final value of the formal argument is assigned back to the actual argument when the call finishes. In the absence of aliasing, VAR semantics are identical to reference semantics.

HANDLE. An AC argument passed as a handle is treated as a numeric quantity, not an address. There is no interpretation or dereferencing of the address itself. Lupine does not consider a handle type to be an AC type. The compiler, however, does, and this can cause serious safety breeches for REF-containing types used as RPC handles. Basically, Cedar RPC programmers cannot use REF-containing handles; see a Lupine wizard.

Syntax

The specification of parameter passing semantics is somewhat awkward because Mesa and Cedar support only call-by-value semantics. There are two methods:

> *Explicit.* To explicitly declare a passing method, RPC programmers include VAR, VALUE, RESULT, or HANDLE as a prefix to a parameter's type name. For example,
>
> ```
> Buffer: TYPE = DESCRIPTOR FOR ARRAY OF DiskPage;
> VALUEBuffer: TYPE = Buffer;
> RESULTBuffer: TYPE = Buffer;
> VARBuffer: TYPE = Buffer;
> ReadPages: PROCEDURE [..., inputPages: RESULTBuffer];
> WritePages: PROCEDURE [..., outputPages: VALUEBuffer];
> XORPages: PROCEDURE [previousPages: VARBuffer, nextPage: VALUEBuffer];
> ```
>
> *Implicit.* To implicitly establish a passing method RPC programmers use the **DefaultPassingMethod** option at translation time. Such blanket declarations are not recommended, and you should see a Lupine wizard before using **DefaultPassingMethod.**

Applicability

The following table indicates which passing methods are applicable to each type (a blank entry means No). Value semantics is the default in all cases. Further restrictions are listed below.

The types below can be passed by	VAR	VALUE	RESULT	HANDLE
INT, BOOL, CHAR, REAL, ...		Yes		
Subrange, Enumeration		Yes		
RECORD, variant RECORD		Yes		
ARRAY, SEQUENCE		Yes		
OPAQUE (with assignment)		Yes		
ANY, OPAQUE (without assignment)		No		
CONDITION, MONITORLOCK		No		
ZONE		No		Yes
PROCEDURE, SIGNAL, ERROR		Not yet		Yes
REF, POINTER, DESCRIPTOR	Yes[1]	Yes	Yes[1]	Yes
RELATIVE POINTER[2]		Yes		
LIST, ROPE, ATOM, *ShortROPE*, *ShortATOM*[3]		Yes		Yes
STRING, *ShortSTRING*[3]	Yes	Yes	Yes	Yes
TEXT, STRINGBODY		Yes		

[1] These AC types can be called by VAR and RESULT, but their referents may not in turn contain any AC types. Restated, VAR and RESULT are only one level deep.

[2] Lupine does not consider relative pointers to be AC, and always treats them as numeric values.

[3] *ShortROPE*, *ShortSTRING*, and *ShortATOM* are string types that should be used by RPC programmers only when a string's length has a guaranteed limit. Grapevine's *RNames* are a primary example because they will frequently be used in remote calls to specify users, principals, servers, and so forth. The use of short strings is always optional: when used, Lupine can usually optimize stubs by guaranteeing that a call always fits in one packet. The short-string length limit, defined by *RPC.maxShortStringLength*, is currently 64. The other properties of these types are identical to those of their longer brothers.

Guarantees

Lupine's typechecking is no stronger than the compiler's. In generating code, Lupine assumes that the value of a parameter is always a legal value for a parameter of that type—there is no special checking. If a client programmer passes a REF which has been loopholed to an invalid address, or passes an out-of-bounds enumerated or subrange value, the results are unspecified. For example, if Lupine must dereference the bogus REF during marshaling, either an address fault will occur or an equally bogus referent will be handed to an unsuspecting server. For an invalid enumerated or subrange value, a fault will occur only if the stub modules were compiled with bounds checking enabled.

The upshot is that RPC programmers must be just as suspicious of invalid parameters as they would be in the absence of RPC. Furthermore, programmers of servers must be especially careful because RPC client code need not be generated by Lupine, or even written in Mesa.

7 Interface and Parameter Restrictions

Lupine tries hard to make local and remote procedure semantics identical, and as a result there are very few restrictions. Here are the details.

Interface Remarks and Restrictions

PROGRAMs, PROCESSes, and PORTs. Lupine supports only remote procedures, signals, and errors. Remote programs, processes, and ports are not permitted.

Interface variables. Lupine does not support global variables in interfaces, including procedures and other items accessed through POINTER TO FRAME.

INLINE procedures. Lupine ignores inline procedures in remote interfaces, since inline procedure bodies appear in the interface itself and are compiled directly into clients.

Monitors. A procedure's ENTRY attribute is invisible in interfaces, and thus stub procedures never have the ENTRY attribute. This is exactly the desired behavior, for the actual implementation is the only module that may need to be a monitor.

Multiple module names. Lupine does not support multiple module names for remote interfaces (e.g., *A, B, C*: DEFINITIONS).

Default parameters and dynamic interface instances. Dynamic interface instantiation via *ImportNewInterface* uses an explicit interface record declared in *TargetRpcControl.* Because the parameter declarations in this record do not duplicate any explicit defaults declared with the formal parameters in *Target* (a symbol table shortcoming), explicit parameter defaulting cannot occur. (Implicit default values from the parameter's *type* declaration, however, continue to work properly.) This is in contrast to remote calls through a single *Target* instance, where the defaults, if any, are available.

Identifier conflicts. While unlikely, it's possible that an internal stub-module identifier used by Lupine (e.g., temporary variable) will conflict with an external identifier used by a remote interface (e.g., procedure or parameter name). The compiler will discover and complain about such conflicts, which should be reported to Lupine's implementors straightaway. The short-term repair, however, is renaming the conflicting identifiers in the remote interface.

Parameter Marshaling Restrictions

Lists, trees, and graphs. Lupine does not automatically marshal these recursive data structures when they are constructed from explicit REFs or POINTERs. Only noncyclic LIST structures are supported.

Callback parameters. Procedure, signal, and error parameters are on the verge of working. If you need them, tell a Lupine wizard, who might finish the job just for you.

Computed sequences and variant records. Lupine cannot marshal any computed sequences, nor can Lupine marshal computed or overlaid variant records with AC components.

Sequences inside variant records. Lupine cannot marshal a sequence contained directly within a variant record. However, Lupine happily accepts references to sequences Thus the first line below is illegal, but the second is acceptable:

RECORD [SELECT *color*: * FROM *red* => [SEQUENCE . . . OF *Intensity*]];

RECORD [SELECT *color*: * FROM *red* => [REF RECORD[SEQUENCE . . . OF *Intensity*]]];

Anonymous record components and single-component records. Lupine cannot cope with anonymous record components. (Anonymous parameters, however, are fine, except for signal and error arguments.) Therefore unnamed, single- or multi-component, address-containing (AC) records will cause a translation error. To recover, just give the components names.

Default values for variant records. Sometimes, to marshal a variant record, Lupine must NEW storage for it. Safety requires that the new record have an initial tag value, but Lupine is not in a position to choose the correct one. An RPC programmer must therefore specify a default value for the record in the record's declaration, even if Lupine's stubs are the only place it is used. You needn't worry about this, however, until the compiler first complains about it.

Record type declarations. To marshal certain records, Lupine must NEW each of the embedded components. If one of these components is an explicit record specification (e.g., RECORD [*red, green, ...*]), and not a record type identifier (e.g., *ColorRecord*), the compiler will report a type clash. If this happens, RPC programmers must redeclare all such records to be record types.

Packed arrays of AC types. Lupine will properly translate these ill-used types (e.g., PACKED ARRAY OF STRING), but the compiler will not accept the result because of an @ applied to the packed elements.

8 References

[Cedar] Cedar Language Committee.
Cedar Mesa—Version 6T5.
File [Indigo]<CedarLang>Doc>Cedar6T5.press, 16 January 1981.
Cedar documentation is evolving, so make current inquiries: this is a founding document.

[CourierFS] Remote procedure calls.
Pilot Programmer's Manual [Pilot], chapter 6.3.
The *Courier Functional Specification* is now an integral part of the PPM.

[CourierPS] *Courier: The Remote Procedure Call Protocol.*
Xerox System Integration Standard XSIS 038112, December 1981.

[DES] National Bureau of Standards.
Data Encryption Standard.
FIPS Publication 46, January, 1977.

[Diplomat] Bruce Nelson.
Diplomat, Attache to Envoy.
File [Idun]<Nelson>Diplomat>Diplomat.press, version 5.0, 8 August 1980.
Envoy is an ancestor of Courier, so this document is somewhat dated.

[Grapevine] Andrew Birrell, Roy Levin, Roger Needham, and Michael Schroeder.
Grapevine, an Exercise in Distributed Computing.
Communications of the ACM 25(4): 260–74, April 1982.

[Mesa] James G. Mitchell, William Maybury, Richard Sweet.
Mesa Language Manual.
Xerox PARC technical report CSL-79-3, version 5.0, April 1979.

[Needham-Schroeder]
Roger Needham and Michael Schroeder.
Using encryption for authentication in large networks of computers.
Communications of the ACM 21(12):993–99, December, 1978.

[Pilot] *Pilot Programmer's Manual.*
Xerox Office Products Division, version 8.0, March 1982.

[RPC] Bruce Nelson.
Remote Procedure Call.
Xerox PARC technical report CSL-81-9, May 1981.

Appendix A: Public RPC Interfaces

The following are the public interfaces to Lupine's RPC runtime, RPC.mesa and MesaRPC.mesa. Small details are likely to have changed since this document was last edited, so remember that Lupine.df always refers to the latest version.

```
-- RPC.mesa
-- Andrew Birrell        3-Dec-81 10:14:28
-- BZM                  29-Oct-81 11:46:31

DIRECTORY
Rope            USING [ROPE],
MesaRPC         USING [AuthenticateFailure, CallFailure, Conversation,
                  ConversationID, ConversationLevel, EncryptionKey,
                  EndConversation, ExportFailure, GetLevel, ImportFailure,
                  GetConversationID, GenerateConversation, matchAllVersions,
                  maxPrincipalLength, maxShortStringLength, SecurityLevel,
                  ShortSTRING, unencrypted, VersionRange];

RPC: DEFINITIONS
  IMPORTS MesaRPC =
  BEGIN


-- Short string/rope/atom types. Used only by Lupine clients.

  maxShortStringLength: CARDINAL = MesaRPC.maxShortStringLength;
  -- maximum length of ShortSTRING/ShortROPE/ShortATOM values --

  ShortSTRING: TYPE = MesaRPC.ShortSTRING;
  ShortROPE: TYPE = Rope.ROPE;
  ShortATOM: TYPE = ATOM;


-- Types for Import and Export calls.

  InterfaceName: TYPE = RECORD [
        type: ShortROPE ← NIL,       -- e.g., "AlpineAccess.Alpine"
        instance: ShortROPE ← NIL,   -- e.g., "MontBlanc.Alpine"
        version: VersionRange ← matchAllVersions ];

  defaultInterfaceName: InterfaceName = [];

  VersionRange: TYPE = MesaRPC.VersionRange;

  matchAllVersions: VersionRange = MesaRPC.matchAllVersions;


-- Parameter storage zones.  Used only by Lupine clients.

  Zones: TYPE = RECORD [
        gc:   ZONE ← NIL,
        heap: UNCOUNTED ZONE ← NIL,
        mds:  MDSZone ← NIL ];

  standardZones: Zones = [];

-- Encryption and Authentication facilities.

  maxPrincipalLength: CARDINAL = MesaRPC.maxPrincipalLength;
    -- Limit on length of ropes used for Principal names --

  Principal: TYPE = ShortROPE;

  EncryptionKey: TYPE = MesaRPC.EncryptionKey;

  MakeKey: PROCEDURE [text: Rope.ROPE] RETURNS[EncryptionKey];

  Conversation: TYPE = MesaRPC.Conversation;
  SecurityLevel: TYPE = MesaRPC.SecurityLevel;

  ConversationLevel: TYPE = MesaRPC.ConversationLevel;

  unencrypted: Conversation = MesaRPC.unencrypted;

  GenerateConversation: PROC RETURNS[Conversation] = INLINE
    { RETURN[ MesaRPC.GenerateConversation[] ] };

  StartConversation: PROCEDURE[caller: Principal, key: EncryptionKey,
                               callee: Principal,
                               level: ConversationLevel ]
                       RETURNS[conversation: Conversation];

  EndConversation: PROCEDURE [conversation: Conversation] = INLINE
    { MesaRPC.EndConversation[conversation] };

  GetCaller: PROCEDURE [conversation: Conversation]
               RETURNS [caller: Principal];

  GetLevel: PROCEDURE [conversation: Conversation]
               RETURNS [level: SecurityLevel] = INLINE
    { RETURN[ MesaRPC.GetLevel[conversation] ] };

  ConversationID: TYPE = MesaRPC.ConversationID;

  GetConversationID: PROC[conversation: Conversation]
             RETURNS[id: ConversationID] = INLINE
    { RETURN[ MesaRPC.GetConversationID[conversation] ] };


-- Public signals:

  AuthenticateFailure: TYPE = MesaRPC.AuthenticateFailure;

  ExportFailure:       TYPE = MesaRPC.ExportFailure;

  ImportFailure:       TYPE = MesaRPC.ImportFailure;

  CallFailure:         TYPE = MesaRPC.CallFailure;

  AuthenticateFailed: ERROR[why: AuthenticateFailure];
    -- Raised by StartConversation --

  ExportFailed: ERROR[why: ExportFailure];
    -- Raised by ExportInterface --

  ImportFailed: ERROR[why: ImportFailure];
    -- Raised by ImportInterface --

  CallFailed: SIGNAL[why: CallFailure];
    -- Raised by any remote call; only why=timeout is resumable --

END.
```

```
-- MesaRPC.mesa
-- Andrew Birrell         3-Dec-81 10:12:00
-- BZM                   29-Oct-81 11:45:47

DIRECTORY
BodyDefs        USING[ maxRNameLength, Password ];

MesaRPC: DEFINITIONS =

  BEGIN


-- Short string types. Used only by Lupine clients.

  maxShortStringLength: CARDINAL = 64;
  -- Maximum length of ShortSTRING values. --

  ShortSTRING: TYPE = STRING;


-- Types for Import/Export calls --

  InterfaceName: TYPE = RECORD [
        type:     LONG ShortSTRING ← NIL, -- e.g. "AlpineAccess.Alpine" --
        instance: LONG ShortSTRING ← NIL, -- e.g. "MontBlanc.Alpine" --
        version:  VersionRange ← matchAllVersions];

  defaultInterfaceName: InterfaceName = [];

  VersionRange: TYPE = MACHINE DEPENDENT RECORD[first, last: CARDINAL];
    -- client-defined, closed interval --

  matchAllVersions: VersionRange = [1,0];
    -- importer: use any version;  exporter: no versioning implied --



-- Parameter storage zones.  Used only by Lupine clients, not the runtime.

  Zones: TYPE = RECORD [
        heap: UNCOUNTED ZONE ← NIL,
        mds:  MDSZone ← NIL ];

  standardZones: Zones = [];

-- Encryption and Authentication facilities --

  maxPrincipalLength: CARDINAL = MIN[maxShortStringLength,
                                       BodyDefs.maxRNameLength];
    -- Limit on length of strings used for Principal --

  Principal:    TYPE = LONG ShortSTRING;
  -- Name of authentication principal --

  EncryptionKey: TYPE = BodyDefs.Password;
    -- DES key --

  MakeKey: PROC[text: LONG STRING] RETURNS[EncryptionKey];

  Conversation: TYPE = LONG POINTER TO ConversationObject;

  ConversationObject: PRIVATE TYPE;

  SecurityLevel: TYPE = MACHINE DEPENDENT {
        none(0),        -- unauthenticated, insecure; used for "unencrypted"
        authOnly(1),    -- authenticated, but unencrypted calls
        ECB(2),         -- authenticated, encrypt with ECB mode of DES
        CBC(3),         -- authenticated, encrypt with CBC mode of DES
        CBCCheck(4)     -- authenticated, encrypt with CBC mode of DES + checksum
        };

  ConversationLevel: TYPE = SecurityLevel[authOnly..CBCCheck];
  unencrypted:  Conversation = NIL;
    -- Dummy conversation; may be passed to RPC runtime.
    --    GetConversationID[unencrypted] = ERROR;
    --    GetCaller[unencrypted] = NIL;
    --    GetLevel[unencrypted] = none; --

  GenerateConversation: PROC RETURNS[Conversation];
    -- Returns a handle for a previously unused Conversation.  This
    -- conversation is only for local use, it must not be passed to
    -- the RPC runtime.
    --    GetConversationID[GenerateConversation[]] = unique ID;
    --    GetCaller[GenerateConversation[]] = NIL;
    --    GetLevel[GenerateConversation[]] = "none"; --

  StartConversation: PROC[caller: Principal, key: EncryptionKey,
                          callee: Principal,
                          level: ConversationLevel]
                   RETURNS[conversation: Conversation];
    -- Obtains authenticator for conversation, registers it with runtime,
    -- and allocates ConversationID --

  EndConversation: PROC[conversation: Conversation];
    -- Terminates use of this conversation --

  GetCaller:     PROC[conversation: Conversation]
             RETURNS[caller: Principal];
    -- Returns the caller name for a current call.  The result
    -- string has lifetime at least equal to the duration of the
    -- call.  Result is NIL if conversation's security level is "none" (including
    -- conversation = "unencrypted"). --

  GetLevel:      PROC[conversation: Conversation]
             RETURNS[level: SecurityLevel];

  ConversationID: TYPE[3];
    -- UID allocated by initiator host --

  GetConversationID: PROC[conversation: Conversation]
             RETURNS[id: ConversationID];
    -- Returns permanently unique ID of this conversation --



-- Public signals --

  AuthenticateFailure: TYPE = {
        communications, -- couldn't contact authentication server(s) --
        badCaller,      -- invalid caller name --
        badKey,         -- incorrect caller password --
        badCallee       -- invalid callee name --
        };

  ExportFailure: TYPE = {
        communications, -- couldn't access binding database --
        badType,        -- unacceptable interface type name --
        badInstance,    -- unacceptable interface instance name --
        badVersion,     -- statically silly version range --
        tooMany,        -- too many exports for local tables --
        badCredentials  -- not allowed to change the database --
        };

  ImportFailure: TYPE = {
        communications, -- couldn't access binding database --
        badType,        -- unacceptable interface type name --
        badInstance,    -- unacceptable interface instance name --
        badVersion,     -- statically silly version range --
        wrongVersion,   -- exported version not in req'd range --
        unbound,        -- this instance not exported --
        stubProtocol    -- exporter protocol incompatible with importer --
        };

  CallFailure:  TYPE = {
        timeout,        -- no acknowledgement within reasonable time --
        unbound,        -- server no longer exports the interface --
        busy,           -- server says it's too busy --
        runtimeProtocol,-- user/server runtimes don't understand each other --
        stubProtocol    -- user/server stubs don't understand each other --
        };

  AuthenticateFailed: ERROR[why: AuthenticateFailure];
    -- Raised by StartConversation --

  ExportFailed: ERROR[why: ExportFailure];
    -- Raised by ExportInterface --

  ImportFailed: ERROR[why: ImportFailure];
    -- Raised by ImportInterface --

  CallFailed:   SIGNAL[why: CallFailure];
    -- Raised by any remote call; only why=timeout is resumable --

END.
```

Appendix B: Example Remote Interface and Stubs

There follows a very simple interface, *Target.mesa*, and its stub modules as an example. Once again, small details may have changed, but this is unimportant since RPC programmers need not be concerned with stub implementations. These modules are un-retouched output from the Lupine translator.

```
-- Lupine: example interface
-- Target.mesa
-- Andrew Birrell  July 8, 1982 9:22 am

DIRECTORY
Rope    USING[ ROPE ];

Target: DEFINITIONS =

BEGIN

Basic: PROC;

Simple: PROC[first: INT, second: REF INT] RETURNS[a: Rope.ROPE, b: ATOM];

Reason: TYPE = { x, y, z };

Exception: ERROR[why: Reason];

Consultation: SIGNAL;

END.
```

```
-- Stub file TargetRpcControl.mesa was translated on  8-Jul-82  9:59:49
    -- PDT by Lupine of  7-Jul-82 17:14:26 PDT.

-- Source interface Target came from file Target.bcd, which was created
    -- on  8-Jul-82  9:59:46 PDT with version stamp 52#64#37310473537 from
    -- source of  8-Jul-82  9:33:34 PDT.

-- The RPC stub modules for Target are:
    -- TargetRpcControl.mesa;
    -- TargetRpcClientImpl.mesa;
    -- TargetRpcBinderImpl.mesa;
    -- TargetRpcServerImpl.mesa.

-- The parameters for this translation are:
    -- Target language = Cedar;
    -- Default parameter passing = VALUE;
    -- Deallocate server heap arguments = TRUE;
    -- Inline RpcServerImpl dispatcher stubs = TRUE;
    -- Maximum number of dynamic heap NEWs = 50, MDS NEWs = 50;
    -- Acceptable parameter protocols = VersionRange[1,1].


DIRECTORY
  Rope,
  Target,
  RPC USING [defaultInterfaceName, EncryptionKey, InterfaceName, Principal,
      standardZones, VersionRange, Zones];


TargetRpcControl: DEFINITIONS
  SHARES  Target
  = BEGIN OPEN Target, RpcPublic: RPC;


-- Public RPC types and constants.

  InterfaceName: TYPE = RpcPublic.InterfaceName;
  VersionRange: TYPE = RpcPublic.VersionRange;
  Principal: TYPE = RpcPublic.Principal;
  EncryptionKey: TYPE = RpcPublic.EncryptionKey;
  Zones: TYPE = RpcPublic.Zones;

  defaultInterfaceName: InterfaceName = RpcPublic.defaultInterfaceName;
  standardZones: Zones = RpcPublic.standardZones;


-- Standard remote binding routines.

  ImportInterface: SAFE PROCEDURE [
        interfaceName: InterfaceName ← defaultInterfaceName,
        parameterStorage: Zones ← standardZones ];

  UnimportInterface: SAFE PROCEDURE;

  ExportInterface: SAFE PROCEDURE [
        interfaceName: InterfaceName ← defaultInterfaceName,
        user: Principal,
        password: EncryptionKey,
        parameterStorage: Zones ← standardZones ];

  UnexportInterface: SAFE PROCEDURE;
```

```
-- Dynamic instantiation and binding.

  ImportNewInterface: SAFE PROCEDURE [
        interfaceName: InterfaceName ← defaultInterfaceName,
        parameterStorage: Zones ← standardZones ]
    RETURNS [interfaceRecord: InterfaceRecord];

  InterfaceRecord: TYPE = REF InterfaceRecordObject;

  InterfaceRecordObject: TYPE = RECORD [
      Basic: PROCEDURE,
      Simple: PROCEDURE [first: INT, second: REF INT] RETURNS [a: Rope.ROPE,
          b: ATOM],
      Exception: ERROR [why: Reason],
      Consultation: SIGNAL,
      lupineDetails: PRIVATE REF LupineDetailsObject←NIL];

  LupineDetailsObject: PRIVATE TYPE;


-- Definitions for the stubs.

  LupineProtocolVersion: PUBLIC VersionRange = [first: 1, last: 1];

  InterMdsCallsOnly: PUBLIC BOOLEAN = FALSE;

  ProcedureIndex: PRIVATE TYPE = MACHINE DEPENDENT {
      LupineUnusedIndex (0), LupineLastIndex (3),
      Basic (4), Simple (5)};

  SignalIndex: PRIVATE TYPE = MACHINE DEPENDENT {
      LupineUnusedIndex (0), LupineLastIndex (3),
      Exception (4), Consultation (5)};


    END.  -- TargetRpcControl.
```

```
-- Stub file TargetRpcClientImpl.mesa was translated on  8-Jul-82
    -- 9:59:50 PDT by Lupine of  7-Jul-82 17:14:26 PDT.

-- Source interface Target came from file Target.bcd, which was created
    -- on  8-Jul-82  9:59:46 PDT with version stamp 52#64#37310473537 from
    -- source of  8-Jul-82  9:33:34 PDT.

-- The RPC stub modules for Target are:
    -- TargetRpcControl.mesa;
    -- TargetRpcClientImpl.mesa;
    -- TargetRpcBinderImpl.mesa;
    -- TargetRpcServerImpl.mesa.

-- The parameters for this translation are:
    -- Target language = Cedar;
    -- Default parameter passing = VALUE;
    -- Deallocate server heap arguments = TRUE;
    -- Inline RpcServerImpl dispatcher stubs = TRUE;
    -- Maximum number of dynamic heap NEWs = 50, MDS NEWs = 50;
    -- Acceptable parameter protocols = VersionRange[1,1].


DIRECTORY
  Rope,
  Target,
  TargetRpcControl USING [InterMdsCallsOnly, LupineProtocolVersion,
      ProcedureIndex, SignalIndex],
  RPC USING [InterfaceName, standardZones, Zones],
  RPCLupine --USING SOME OF [Call, DataLength, Dispatcher, GetStubPkt,
      -- ImportHandle, ImportInterface, maxDataLength, maxPrincipalLength,
      -- maxShortStringLength, pktOverhead, ReceiveExtraPkt, SendPrelimPkt,
      -- StartCall, StartSignal, StubPkt, UnimportInterface]--,
  LupineRuntime --USING SOME OF [BindingError, CheckPktLength, CopyFromPkt,
      -- CopyFromMultiplePkts, CopyToPkt, CopyToMultiplePkts, DispatchingError,
      -- FinishThisPkt, ListHeader, MarshalingError, MarshalingExprError,
      -- NilHeader, ProtocolError, RopeHeader, RpcPktDoubleWord, RuntimeError,
      -- SequenceHeader, SHORT, StartNextPkt, StringHeader, StubPktDoubleWord,
      -- TranslationError, UnmarshalingError, UnmarshalingExprError, WordsForChars]--,
  Atom --USING SOME OF [GetPName, MakeAtom]--,
  ConvertUnsafe USING [AppendRope],
  Heap USING [systemMDSZone],
  RopeInline --USING SOME OF [InlineFlatten, NewText]--,
  SafeStorage USING [GetSystemZone],
  UnsafeStorage USING [GetSystemUZone];


TargetRpcClientImpl: MONITOR
  IMPORTS RpcPrivate: RPCLupine, Lupine: LupineRuntime, Atom, ConvertUnsafe,
      Heap, RopeInline, SafeStorage, UnsafeStorage
  EXPORTS Target, TargetRpcControl
  SHARES  Target, TargetRpcControl, Rope
  = BEGIN OPEN Target, RpcControl: TargetRpcControl, RpcPublic: RPC;
```

```
-- Standard remote binding routines.

  bound: BOOLEAN ← FALSE;
  myInterface: RpcPrivate.ImportHandle ← NULL;
  paramZones: RpcPublic.Zones ← RpcPublic.standardZones;

  ImportInterface: PUBLIC ENTRY SAFE PROCFƆURE [
        interfaceName: RpcPublic.InterfaceName,
        parameterStorage: RpcPublic.Zones ] =
    TRUSTED BEGIN ENABLE UNWIND => NULL;
    IsNull: PROCEDURE [string: LONG STRING] RETURNS [BOOLEAN] =
      INLINE {RETURN[ string=NIL OR string.length=0 ]};
    IF bound THEN Lupine.BindingError;
    BEGIN
    type: STRING = [RpcPrivate.maxShortStringLength];
    instance: STRING = [RpcPrivate.maxShortStringLength];
    ConvertUnsafe.AppendRope[to: type, from: interfaceName.type];
    ConvertUnsafe.AppendRope[to: instance, from: interfaceName.instance];
    myInterface ← RpcPrivate.ImportInterface [
      interface: [
        type: IF ~IsNull[type]
          THEN type ELSE "Target~52#64#37310473537"L,
        instance: instance,
        version: interfaceName.version ],
      localOnly: RpcControl.InterMdsCallsOnly,
      stubProtocol: RpcControl.LupineProtocolVersion ];
    END;
    paramZones ← [
      gc: IF parameterStorage.gc # NIL
        THEN parameterStorage.gc ELSE SafeStorage.GetSystemZone[],
      heap: IF parameterStorage.heap # NIL
        THEN parameterStorage.heap ELSE UnsafeStorage.GetSystemUZone[],
      mds: IF parameterStorage.mds # NIL
        THEN parameterStorage.mds ELSE Heap.systemMDSZone ];
    bound ← TRUE;
    END;

  UnimportInterface: PUBLIC ENTRY SAFE PROCEDURE =
    TRUSTED BEGIN ENABLE UNWIND => NULL;
    IF ~bound THEN Lupine.BindingError;
    myInterface ← RpcPrivate.UnimportInterface[myInterface];
    paramZones ← RpcPublic.standardZones;
    bound ← FALSE;
    END;
```

```
-- Remote public procedure stubs.

  Basic: PUBLIC PROCEDURE =
    BEGIN
    ArgumentOverlay: TYPE = MACHINE DEPENDENT RECORD [
        transferIndex (0): RpcControl.ProcedureIndex ← Basic];
    pktBuffer: ARRAY [1..RpcPrivate.pktOverhead+1] OF WORD;
    pkt: RpcPrivate.StubPkt = RpcPrivate.GetStubPkt[space: @pktBuffer];
    argPkt: POINTER TO ArgumentOverlay = @pkt.data[0];
    pktLength: RpcPrivate.DataLength ← 1;
    lastPkt: BOOLEAN;
    RpcPrivate.StartCall[callPkt: pkt, interface: myInterface];
    argPkt.transferIndex ← Basic;
    [returnLength: , lastPkt: lastPkt] ←
      RpcPrivate.Call[ pkt: pkt, callLength: pktLength,
          maxReturnLength: 0, signalHandler: ClientDispatcher];
    Lupine.CheckPktLength[pkt: pkt, pktLength: 0];
    RETURN[];
    END;  -- Basic.

  Simple: PUBLIC PROCEDURE [first: INT, second: REF INT] RETURNS [a:
      Rope.ROPE, b: ATOM] =
    BEGIN
    ArgumentOverlay: TYPE = MACHINE DEPENDENT RECORD [
        transferIndex (0): RpcControl.ProcedureIndex ← Simple, first (1):
        INT];
    pktBuffer: ARRAY [1..RpcPrivate.pktOverhead+254] OF WORD;
    pkt: RpcPrivate.StubPkt = RpcPrivate.GetStubPkt[space: @pktBuffer];
    argPkt: POINTER TO ArgumentOverlay = @pkt.data[0];
    pktLength: RpcPrivate.DataLength ← 3;
    lastPkt: BOOLEAN;
    RpcPrivate.StartCall[callPkt: pkt, interface: myInterface];
    argPkt↑ ← [first: first];
    BEGIN  -- Marshal second: REF INT to pkt.data[pktLength].
      pkt.data[pktLength] ← second=NIL;  pktLength ← pktLength+1;
      IF second # NIL THEN
        BEGIN
        Lupine.StubPktDoubleWord[pkt, pktLength]↑ ← second↑;
        pktLength ← pktLength + 2;
        END;
      END;  -- Marshal second.
    [returnLength: , lastPkt: lastPkt] ←
      RpcPrivate.Call[ pkt: pkt, callLength: pktLength,
          maxReturnLength: 254, signalHandler: ClientDispatcher];
    pktLength ← 0;
    BEGIN  -- Unmarshal a: Rope.ROPE from pkt.data[pktLength].
      ropeIsNIL: Lupine.NilHeader;
      IF pktLength+2 > RpcPrivate.maxDataLength
        THEN pktLength ← Lupine.FinishThisPkt[pkt: pkt, pktLength:
          pktLength];
      ropeIsNIL ← pkt.data[pktLength];  pktLength ← pktLength+1;
      IF ropeIsNIL
        THEN a ← NIL
        ELSE BEGIN
          ropeLength: Lupine.RopeHeader;
          textRope: Rope.Text;
          ropeLength ← pkt.data[pktLength];  pktLength ← pktLength+1;
          IF ropeLength > LAST[NAT]
            THEN Lupine.UnmarshalingError;
          a ← textRope ← RopeInline.NewText[size: ropeLength];
          pktLength ← Lupine.CopyFromPkt[pkt: pkt, pktLength: pktLength,
              dataAdr: BASE[DESCRIPTOR[textRope.text]], dataLength:
Lupine.WordsForChars[ropeLength],
              alwaysOnePkt: FALSE];
          END;  -- IF ropeIsNIL.
      END;  -- Unmarshal a.
    BEGIN  -- Unmarshal b: ATOM from pkt.data[pktLength].
      pNameOfAtom: Rope.ROPE;
      ropeIsNIL: Lupine.NilHeader;
      IF pktLength+2 > RpcPrivate.maxDataLength
        THEN pktLength ← Lupine.FinishThisPkt[pkt: pkt, pktLength:
            pktLength];
      ropeIsNIL ← pkt.data[pktLength];  pktLength ← pktLength+1;
      IF ropeIsNIL
        THEN pNameOfAtom ← NIL
        ELSE BEGIN
          ropeLength: Lupine.RopeHeader;
          textRope: Rope.Text;
          ropeLength ← pkt.data[pktLength];  pktLength ← pktLength+1;
          IF ropeLength > LAST[NAT]
            THEN Lupine.UnmarshalingError;
          pNameOfAtom ← textRope ← RopeInline.NewText[size: ropeLength];
          pktLength ← Lupine.CopyFromPkt[pkt: pkt, pktLength: pktLength,
              dataAdr: BASE[DESCRIPTOR[textRope.text]], dataLength:
Lupine.WordsForChars[ropeLength],
              alwaysOnePkt: FALSE];
          END;  -- IF ropeIsNIL.
      b ← Atom.MakeAtom[--pName:-- pNameOfAtom];
      END;  -- Unmarshal b.
    Lupine.CheckPktLength[pkt: pkt, pktLength: pktLength];
    RETURN[a, b];
    END;  -- Simple.
```

```
-- Remote public signals and errors.

 Exception: PUBLIC ERROR [why: Reason] = CODE;

 Consultation: PUBLIC SIGNAL = CODE;


-- Public signal and error dispatcher.

 ClientDispatcher: --PROCEDURE [pkt: RPCPkt, callLength: DataLength,
   -- lastPkt: BOOLEAN, localConversation: Conversation] RETURNS [returnLength:
   -- DataLength]-- RpcPrivate.Dispatcher =
  BEGIN

  SELECT LOOPHOLE[pkt.data[0], RpcControl.SignalIndex] FROM
   Exception => RETURN[
    ExceptionStub[pkt: pkt, callLength: callLength, lastPkt: lastPkt,
      localConversation: localConversation]];
   Consultation => RETURN[
    ConsultationStub[pkt: pkt, callLength: callLength, lastPkt:
      lastPkt, localConversation: localConversation]];
   ENDCASE => RETURN[Lupine.DispatchingError[]];

  END;  -- ClientDispatcher


-- Public signal and error dispatcher stubs.

 ExceptionStub: --ERROR [why: Reason]-- RpcPrivate.Dispatcher =
  INLINE BEGIN
  ArgumentOverlay: TYPE = MACHINE DEPENDENT RECORD [
    transferIndex (0): RpcControl.SignalIndex, why (1): Reason];
  argPkt: LONG POINTER TO ArgumentOverlay = @pkt.data[0];
  Lupine.CheckPktLength[pkt: pkt, pktLength: 2];
  ERROR Exception[argPkt.why];
  END;  -- ExceptionStub.

 ConsultationStub: --SIGNAL-- RpcPrivate.Dispatcher =
  INLINE BEGIN
  pktLength: RpcPrivate.DataLength;
  Lupine.CheckPktLength[pkt: pkt, pktLength: 1];
  SIGNAL Consultation[];
  pktLength ← 0;
  RETURN[returnLength: pktLength];
  END;  -- ConsultationStub.


-- No module initialization.

 END.  -- TargetRpcClientImpl.
```

```
-- Stub file TargetRpcServerImpl.mesa was translated on  8-Jul-82
    -- 9:59:52 PDT by Lupine of  7-Jul-82 17:14:26 PDT.

-- Source interface Target came from file Target.bcd, which was created
    -- on  8-Jul-82  9:59:46 PDT with version stamp 52#64#37310473537 from
    -- source of  8-Jul-82  9:33:34 PDT.

-- The RPC stub modules for Target are:
    -- TargetRpcControl.mesa;
    -- TargetRpcClientImpl.mesa;
    -- TargetRpcBinderImpl.mesa;
    -- TargetRpcServerImpl.mesa.

-- The parameters for this translation are:
    -- Target language = Cedar;
    -- Default parameter passing = VALUE;
    -- Deallocate server heap arguments = TRUE;
    -- Inline RpcServerImpl dispatcher stubs = TRUE;
    -- Maximum number of dynamic heap NEWs = 50, MDS NEWs = 50;
    -- Acceptable parameter protocols = VersionRange[1,1].


DIRECTORY
  Rope,
  Target,
  TargetRpcControl USING [InterMdsCallsOnly, LupineProtocolVersion,
      ProcedureIndex, SignalIndex],
  RPC USING [EncryptionKey, InterfaceName, Principal, standardZones,
      Zones],
  RPCLupine --USING SOME OF [Call, DataLength, Dispatcher, ExportHandle,
      -- ExportInterface, GetStubPkt, maxDataLength, maxPrincipalLength,
      -- maxShortStringLength, pktOverhead, ReceiveExtraPkt, SendPrelimPkt,
      -- StartCall, StartSignal, StubPkt, UnexportInterface]--,
  LupineRuntime --USING SOME OF [BindingError, CheckPktLength, CopyFromPkt,
      -- CopyFromMultiplePkts, CopyToPkt, CopyToMultiplePkts, DispatchingError,
      -- FinishThisPkt, ListHeader, MarshalingError, MarshalingExprError,
      -- NilHeader, ProtocolError, RopeHeader, RpcPktDoubleWord, RuntimeError,
      -- SequenceHeader, SHORT, StartNextPkt, StringHeader, StubPktDoubleWord,
      -- TranslationError, UnmarshalingError, UnmarshalingExprError, WordsForChars]--,
  Atom --USING SOME OF [GetPName, MakeAtom]--,
  ConvertUnsafe USING [AppendRope],
  Heap USING [systemMDSZone],
  RopeInline --USING SOME OF [InlineFlatten, NewText]--,
  SafeStorage USING [GetSystemZone],
  UnsafeStorage USING [GetSystemUZone];


TargetRpcServerImpl: MONITOR
  IMPORTS Target, RpcPrivate: RPCLupine, Lupine: LupineRuntime, Atom,
      ConvertUnsafe, Heap, RopeInline, SafeStorage, UnsafeStorage
  EXPORTS TargetRpcControl
  SHARES  Target, TargetRpcControl, Rope
  = BEGIN OPEN Target, RpcControl: TargetRpcControl, RpcPublic: RPC;
```

```
-- Standard remote binding routines.

 bound: BOOLEAN ← FALSE;
 myInterface: RpcPrivate.ExportHandle ← NULL;
 paramZones: RpcPublic.Zones ← RpcPublic.standardZones;

 ExportInterface: PUBLIC ENTRY SAFE PROCEDURE [
    interfaceName: RpcPublic.InterfaceName,
    user: RpcPublic.Principal,
    password: RpcPublic.EncryptionKey,
    parameterStorage: RpcPublic.Zones ] =
  TRUSTED BEGIN ENABLE UNWIND => NULL;
  IsNull: PROCEDURE [string: LONG STRING] RETURNS [BOOLEAN] =
   INLINE {RETURN[ string=NIL OR string.length=0 ]};
  IF ~bound THEN Lupine.BindingError;
  BEGIN
  type: STRING = [RpcPrivate.maxShortStringLength];
  instance: STRING = [RpcPrivate.maxShortStringLength];
  userString: STRING = [RpcPrivate.maxPrincipalLength];
  ConvertUnsafe.AppendRope[to: type, from: interfaceName.type];
  ConvertUnsafe.AppendRope[to: instance, from: interfaceName.instance];
  ConvertUnsafe.AppendRope[to: userString, from: user];
  myInterface ← RpcPrivate.ExportInterface [
   interface: [
    type: IF ~IsNull[type]
     THEN type ELSE "Target~52#64#37310473537"L,
    instance: instance,
    version: interfaceName.version ],
   user: userString, password: password,
   dispatcher: ServerDispatcher,
   localOnly: RpcControl.InterMdsCallsOnly,
   stubProtocol: RpcControl.LupineProtocolVersion ];
  END;
  paramZones ← [
   gc: IF parameterStorage.gc # NIL
    THEN parameterStorage.gc ELSE SafeStorage.GetSystemZone[],
   heap: IF parameterStorage.heap # NIL
    THEN parameterStorage.heap ELSE UnsafeStorage.GetSystemUZone[],
   mds: IF parameterStorage.mds # NIL
    THEN parameterStorage.mds ELSE Heap.systemMDSZone ];
  bound ← TRUE;
  END;

 UnexportInterface: PUBLIC ENTRY SAFE PROCEDURE =
  TRUSTED BEGIN ENABLE UNWIND => NULL;
  IF ~bound THEN Lupine.BindingError;
  myInterface ← RpcPrivate.UnexportInterface[myInterface];
  paramZones ← RpcPublic.standardZones;
  bound ← FALSE;
  END;
```

```
-- Public procedure dispatcher and public signal and error catcher.

  ServerDispatcher: --PROCEDURE [pkt: RPCPkt, callLength: DataLength,
      -- lastPkt: BOOLEAN, localConversation: Conversation] RETURNS [returnLength:
      -- DataLength]-- RpcPrivate.Dispatcher =
    BEGIN

    -- Catch public signals.

      ENABLE BEGIN

      Exception --ERROR [why: Reason]-- =>
        BEGIN
        ArgumentOverlay: TYPE = MACHINE DEPENDENT RECORD [
            transferIndex (0): RpcControl.SignalIndex ← Exception, why
            (1): Reason];
        argPkt: LONG POINTER TO ArgumentOverlay = @pkt.data[0];
        pktLength: RpcPrivate.DataLength ← 2;
        lastPkt: BOOLEAN;
        RpcPrivate.StartSignal[signalPkt: pkt];
        argPkt↑ ← [why: why];
        [returnLength: , lastPkt: lastPkt] ←
          RpcPrivate.Call[ pkt: pkt, callLength: pktLength,
              maxReturnLength: 0];
        Lupine.RuntimeError;  -- Impossible to RESUME an ERROR.
        END;  -- Exception.

      Consultation --SIGNAL-- =>
        BEGIN
        ArgumentOverlay: TYPE = MACHINE DEPENDENT RECORD [
            transferIndex (0): RpcControl.SignalIndex ← Consultation];
        argPkt: LONG POINTER TO ArgumentOverlay = @pkt.data[0];
        pktLength: RpcPrivate.DataLength ← 1;
        lastPkt: BOOLEAN;
        RpcPrivate.StartSignal[signalPkt: pkt];
        argPkt.transferIndex ← Consultation;
        [returnLength: , lastPkt: lastPkt] ←
          RpcPrivate.Call[ pkt: pkt, callLength: pktLength,
              maxReturnLength: 0];
        Lupine.CheckPktLength[pkt: pkt, pktLength: 0];
        RESUME[];
        END;  -- Consultation.

      END;  -- Catch public signals.


    -- Call public procedures (still in dispatcher).

    SELECT LOOPHOLE[pkt.data[0], RpcControl.ProcedureIndex] FROM
      Basic => RETURN[
        BasicStub[pkt: pkt, callLength: callLength, lastPkt: lastPkt,
            localConversation: localConversation]];
      Simple => RETURN[
        SimpleStub[pkt: pkt, callLength: callLength, lastPkt: lastPkt,
          localConversation: localConversation]];
      ENDCASE => RETURN[Lupine.DispatchingError[]];

    END;  -- ServerDispatcher


-- Public procedure dispatcher stubs.

  BasicStub: --PROCEDURE-- RpcPrivate.Dispatcher =
    INLINE BEGIN
    pktLength: RpcPrivate.DataLength;
    Lupine.CheckPktLength[pkt: pkt, pktLength: 1];
    Basic[];
    pktLength ← 0;
    RETURN[returnLength: pktLength];
    END;  -- BasicStub.

  SimpleStub: --PROCEDURE [first: INT, second: REF INT] RETURNS [a:
      -- Rope.ROPE, b: ATOM]-- RpcPrivate.Dispatcher =
    INLINE BEGIN
    second: REF INT;
    a: Rope.ROPE;
    b: ATOM;
    ArgumentOverlay: TYPE = MACHINE DEPENDENT RECORD [
        transferIndex (0): RpcControl.ProcedureIndex, first (1): INT];
    argPkt: LONG POINTER TO ArgumentOverlay = @pkt.data[0];
    pktLength: RpcPrivate.DataLength ← 3;
    BEGIN  -- Unmarshal second: REF INT from pkt.data[pktLength].
      isNIL: Lupine.NilHeader;
      isNIL ← pkt.data[pktLength];  pktLength ← pktLength+1;
      IF isNIL
        THEN second ← NIL
        ELSE BEGIN
          second ← (paramZones.gc.NEW[INT]);
          BEGIN
          second↑ ← Lupine.RpcPktDoubleWord[pkt, pktLength]↑;
          pktLength ← pktLength + 2;
          END;
          END;  -- IF isNIL.
      END;  -- Unmarshal second.
    Lupine.CheckPktLength[pkt: pkt, pktLength: pktLength];
    [a, b] ←
      Simple[argPkt.first, second];
    pktLength ← 0;
    BEGIN  -- Marshal a: Rope.ROPE to pkt.data[pktLength].
      IF pktLength+2 > RpcPrivate.maxDataLength
        THEN pktLength ← Lupine.StartNextPkt[pkt: pkt, pktLength: pktLength];
      pkt.data[pktLength] ← a=NIL;  pktLength ← pktLength+1;
      IF a # NIL
        THEN BEGIN
          textRope: Rope.Text = RopeInline.InlineFlatten[r: a];
          pkt.data[pktLength] ← textRope.length;  pktLength ← pktLength+1;
          pktLength ← Lupine.CopyToPkt[pkt: pkt, pktLength: pktLength,
              dataAdr: BASE[DESCRIPTOR[textRope.text]], dataLength:
Lupine.WordsForChars[textRope.length],
              alwaysOnePkt: FALSE];
          END;  -- IF a # NIL.
      END;  -- Marshal a.
    BEGIN  -- Marshal b: ATOM to pkt.data[pktLength].
      pNameOfAtom: Rope.Text = Atom.GetPName[atom: b];
      IF pktLength+2 > RpcPrivate.maxDataLength
        THEN pktLength ← Lupine.StartNextPkt[pkt: pkt, pktLength: pktLength];
      pkt.data[pktLength] ← pNameOfAtom=NIL;  pktLength ← pktLength+1;
      IF pNameOfAtom # NIL
        THEN BEGIN
          textRope: Rope.Text = RopeInline.InlineFlatten[r: pNameOfAtom];
          pkt.data[pktLength] ← textRope.length;  pktLength ← pktLength+1;
          pktLength ← Lupine.CopyToPkt[pkt: pkt, pktLength: pktLength,
              dataAdr: BASE[DESCRIPTOR[textRope.text]], dataLength:
Lupine.WordsForChars[textRope.length],
              alwaysOnePkt: FALSE];
          END;  -- IF pNameOfAtom # NIL.
      END;  -- Marshal b.
    RETURN[returnLength: pktLength];
    END;  -- SimpleStub.


-- No module initialization.

  END.  -- TargetRpcServerImpl.
```

```
-- Stub file TargetRpcBinderImpl.mesa was translated on  8-Jul-82
    -- 9:59:51 PDT by Lupine of  7-Jul-82 17:14:26 PDT.

-- Source interface Target came from file Target.bcd, which was created
    -- on  8-Jul-82  9:59:46 PDT with version stamp 52#64#37310473537 from
    -- source of  8-Jul-82  9:33:34 PDT.

-- The RPC stub modules for Target are:
    -- TargetRpcControl.mesa;
    -- TargetRpcClientImpl.mesa;
    -- TargetRpcBinderImpl.mesa;
    -- TargetRpcServerImpl.mesa.

-- The parameters for this translation are:
    -- Target language = Cedar;
    -- Default parameter passing = VALUE;
    -- Deallocate server heap arguments = TRUE;
    -- Inline RpcServerImpl dispatcher stubs = TRUE;
    -- Maximum number of dynamic heap NEWs = 50, MDS NEWs = 50;
    -- Acceptable parameter protocols = VersionRange[1,1].

-- NOTE: Discard this module unless you use dynamic client binding.


DIRECTORY
  TargetRpcControl USING [InterfaceRecord, InterfaceRecordObject],
  TargetRpcClientImpl,
  RPC USING [InterfaceName, Zones],
  RTTypesBasic USING [EstablishFinalization, FinalizationQueue, FQEmpty,
      FQNext, NewFQ];


TargetRpcBinderImpl: MONITOR
  IMPORTS ClientPrototype: TargetRpcClientImpl, RTT: RTTypesBasic
  EXPORTS TargetRpcControl
  SHARES  TargetRpcControl
  = BEGIN OPEN RpcControl: TargetRpcControl, RpcPublic: RPC;


-- Dynamic instantiation and binding routines.

  ImportNewInterface: PUBLIC SAFE PROCEDURE [
        interfaceName: RpcPublic.InterfaceName,
        parameterStorage: RpcPublic.Zones ]
      RETURNS [interfaceRecord: RpcControl.InterfaceRecord] =
    TRUSTED BEGIN
    interfaceRecord ← NewInterface[];
    LupineDetails[interfaceRecord].module.ImportInterface [
        interfaceName: interfaceName,
        parameterStorage: parameterStorage
      ! UNWIND => FreeInterface[interfaceRecord] ];
    END;

  UnimportNewInterface: SAFE PROCEDURE [
      interfaceRecord: RpcControl.InterfaceRecord ] =
    TRUSTED BEGIN
    LupineDetails[interfaceRecord].module.UnimportInterface[];
    FreeInterface[interfaceRecord];
    END;


-- Utility routines for interface instantiation and caching.

  ConcreteLupineDetails: TYPE = REF LupineDetailsObject;

  LupineDetailsObject:  PUBLIC TYPE = RECORD [
        module: ClientModule←NIL,
        next: RpcControl.InterfaceRecord←NIL,
        self: RpcControl.InterfaceRecord←NIL ];

  LupineDetails: PROCEDURE [abstractInterface: RpcControl.InterfaceRecord]
      RETURNS [ConcreteLupineDetails] =
    INLINE {RETURN[abstractInterface.lupineDetails]};
  ClientModule: TYPE = POINTER TO FRAME[TargetRpcClientImpl];


  clientInterfaceCache: RpcControl.InterfaceRecord ← NIL;

  NewInterface: PROCEDURE RETURNS [interface: RpcControl.InterfaceRecord]=
    BEGIN
    GetCachedInterface: ENTRY PROCEDURE
        RETURNS [cachedIR: RpcControl.InterfaceRecord] =
      INLINE BEGIN ENABLE UNWIND => NULL;
      IF (cachedIR←clientInterfaceCache) # NIL
        THEN clientInterfaceCache ← LupineDetails[clientInterfaceCache].next;
      END;
    ReclaimInterfaces;
    IF (interface ← GetCachedInterface[]) = NIL
      THEN BEGIN
        module: ClientModule = NEW ClientPrototype;
        interface ← NEW[
          RpcControl.InterfaceRecordObject ← [
              Basic: module.Basic, Simple: module.Simple, Exception: module.Exception,
              Consultation: module.Consultation]];
        interface.lupineDetails ← NEW[
          LupineDetailsObject ← [module: module, self: interface]];
        END;
    END;

  FreeInterface: ENTRY PROCEDURE [interface: RpcControl.InterfaceRecord]=
    INLINE BEGIN ENABLE UNWIND => NULL;
    LupineDetails[interface].next ← clientInterfaceCache;
    clientInterfaceCache ← interface;
    END;


-- Finalization for dynamic interfaces.  Just cache and reuse for now.

  freedInterfaces: RTT.FinalizationQueue = RTT.NewFQ[20];

  ReclaimInterfaces: PROCEDURE =
    INLINE BEGIN
    WHILE ~RTT.FQEmpty[freedInterfaces] DO
      UnimportNewInterface[
        interfaceRecord: NARROW[RTT.FQNext[freedInterfaces]] ];
      ENDLOOP;
    END;


-- Module initialization.

  RTT.EstablishFinalization[
    type: CODE[RpcControl.InterfaceRecordObject],
    npr: 1,  fq: freedInterfaces ];


  END.  -- TargetRpcBinderImpl.
```

Cedar Creature Comforts

Release as [Indigo]<Cedar>Documentation>NewStuff.tioga
By Warren Teitelman
Last edited By Warren Teitelman on March 12, 1983 1:41 pm

Abstract This document describes a collection of facilities that generally fall under the category of creature comforts: they are not essential, but they make life more pleasant. These facilities are enabled by running NewStuffImpl.bcd, e.g. by including "run newstuffimpl" in the CommandsFrom entry in your user profile. (NewStuffImpl.bcd automatically comes over as part of the release; you may obtain the latest version via [Indigo]<PreCedar>Top>NewStuff.df.) In addition, some of the facilities can be parameterized, or disabled, via various user profile entries described below.

Extensions to Abbreviation Expansion

CTRL-E can now be used following any expression whose value is of type procedure, record, error, signal, etc., i.e. any type whose syntax involves square brackets and named fields, to generate an appropriate "form" for that expression. For example:

User types: Rope.Find{CTRL-E}

produces: Rope.Find[s1: ▶ROPE◀, s2: ▶ROPE◀, pos1: ▶INT ← 0◀, case: ▶BOOL ← TRUE◀]

User types: IO.Error{CTRL-E}

produces: IO.Error[ec: ▶IO.ErrorCode◀, stream: ▶STREAM◀].

The user can move back and forth among the fields using {NEXT} (Spare2) and {SHIFT-NEXT} in the standard fashion. When finished with the form, he can type CTRL-NEXT which will have the effect of deleting any fields not filled in, plus advancing the caret to just beyond the terminating ']. For example:

User types: Rope.Find{CTRL-E}r{NEXT}key{CTRL-NEXT}

produces: Rope.Find[s1: r, s2: key]

i.e. typing "r" filled in the first ▶ROPE◀ field, and typing "key" filled in the second ▶ROPE◀ field, and {CTRL-NEXT} deleted ", pos1: ▶INT ← 0◀, case: ▶BOOL ← TRUE◀" and moved the caret to after '].

Note that CTRL-E can also be used to construct forms when typing to the interpreter in a work-area (since typescripts are now editable).

Another useful application is typing CTRL-E following an expression whose value is a procedure *type*. The will cause the insertion of a comment describing the arguments and return values for the procedure type, and then an appropriate form for the procedure body. For example:

User types: RedSave: Menus.ClickProc{CTRL-E}

produces: RedSave: Menus.ClickProc -- *[parent: REF ANY, clientData: REF ANY ← NIL, mouseButton: Menus.MouseButton ← red, shift: BOOL ← FALSE, control: BOOL ← FALSE]* -- = {

▶Body◀

};

Since this operation involves the interpreter, it can be slow (but so is selecting Rope.Find and clicking Open in order to find out the arguments), especially the first time for a particular interface.

In the case of misspellings, spelling correction will take place according to your user profile. If confirmation is required, Yes No menu buttons will be posted in the corresponding viewer.

CTRL-NEXT

As described above, CTRL-NEXT is used as a "I'm finished with this, go on" in conjunction with CTRL-E abbreviation expansion. It deletes all of the fields that have default values. If there are any fields that have not been filled out that do *not* have default values, the first of these fields is selected and a message printed in the message window. Thus the user can use CTRL-NEXT to eliminate all fields with default values, and then proceed to fill in the remaining fields.

For convenience, CTRL-NEXT can also be used in other contexts involving forms. If the current selection is a placeHolder, CTRL-NEXT deletes the placeHolder and then does moves to the next placeholder.

Updating Last Edited Entry

When you (red) click save on a Tioga document, the system will search for a comment containing a date and your name, and if found, will automatically update the date. The algorithm used is as follows: start searching at the beginning of the document looking at all comments, where a comment is either a tioga node with the comment property, or any line beginning with -- or //, (i.e. this feature also works equally well for documents that have not been converted to Tioga node structure). In each comment, look and see if your name (i.e. the name of the user that is logged in) appears, and if there is also a date. Stop searching at the first non-empty non-comment node. If a comment of the appropriate form was found, update the date of the last such comment (so that user can have separate CreatedOn, LastEditedOn comments). Otherwise, provided that at least one comment was seen, insert a new comment of the form "Last edited by: your name, date", following the last comment.

For example, the first few lines of the file CompatibilityPackage.df are:

```
-- CompatibilityPackage.df
-- last edited by Levin on November 16, 1982 11:33 am
-- last edited by Andrew Birrell on August 5, 1982 11:46 am
```

```
Directory [Indigo]<Cedar>Top>      CameFrom [Indigo]<PreCedar>Top>

  CompatibilityPackage.df                  1-Dec-82 13:19:57 PST
```

If Andrew Birrell were to edit this file and click save, the file would look like:

```
-- CompatibilityPackage.df
-- last edited by Levin on November 16, 1982 11:33 am
-- last edited by Andrew Birrell on December 1, 1982 9:57 pm

Directory [Indigo]<Cedar>Top>      CameFrom [Indigo]<PreCedar>Top>

  CompatibilityPackage.df                  1-Dec-82 13:19:57 PST
```

But if Warren Teitelman were to edit the file and click save, the file would look like:

```
-- CompatibilityPackage.df
-- last edited by Levin on November 16, 1982 11:33 am
-- last edited by Andrew Birrell on August 5, 1982 11:46 am
-- Last Edited by: Teitelman, December 1, 1982 9:57 pm

Directory [Indigo]<Cedar>Top>      CameFrom [Indigo]<PreCedar>Top>

  CompatibilityPackage.df                  1-Dec-82 13:19:57 PST
```

Note that this feature will *never* insert a comment unless one is already there, and will also never insert a comment at the top of the file. It can be disabled by including the entry UpdateLastEdited: FALSE in your profile (the default is TRUE), or can be bypassed for a particular file by clicking the save button with the blue mouse button, rather than the red.

Updating Change Logs

A facility for automatically generating change log entries for documents that employ Tioga node structure is available through the ChangeLog menu button, which is added to the first menu line of all Tioga documents as a result of running NewStuffImpl.bcd. Red-clicking ChangeLog will cause the system to create a change log entry at the end of the document being edited. The entry consists of a Tioga node of the form: "Edited on", the date, "by ", your name, followed by a nested node beginning with "changes to:", followed by a list of the items changed. PlaceHolders for comments are inserted following each item. (Filling in these comments is optional: any placeHolder in the change log entry that is not filled in will be automatically deleted when the file is saved.)

The changed list is generated as follows: for each Tioga node that has been edited, add an entry to the changed list, if one does not already exist, corresponding to the first (most tightly bound) type declaration that this node is under. Only consider type declarations which contain a block as part of its definition (i.e. local variable declarations of a procedure that are edited do not rate separate entries, local procedures do). If there is no such declaration, e.g. the edit is to the DIRECTORY list, or to some expression in the start code, etc., use the highest node that is not a comment. If there is such a declaration, and it is local to some higher declaration, indicate that declaration also when printing the change entry.

For example, if I edit the file PrintTVImpl.mesa by changing: (a) several of the DIRECTORY clauses, (b) the IMPORTS list, (c) the top level declaration for the type UnderLongString, and (d) the procedures EnsureInit, PutCardRope, and innerPut, the latter two of which are local procedures, the change log entry that would be produced would look like:

Edited on December 1, 1982 10:36 pm, by Teitelman

ğeneral commentsi
changes to: DIRECTORY▸◂, IMPORTS▸◂, UnderLongString▸◂, EnsureInit▸◂, PutCardRope *(local of Print)*▸◂, innerPut *(local of PutRecord, local of Print)*▸◂

Continuing an edit "session"

A new change log entry is created only once for a particular edit session. This change log entry is said to be *active* as long as the corresponding document exists, and the corresponding node is part of that document, i.e. has not been deleted. In other words, the system will continue to treat edits as part of the same session as long as the user edits in the same viewer (or a viewer created as a result of splitting this viewer), and as long as the corresponding change log entry still exists. In this case, when the user clicks ChangeLog or Save (see below), a new change log entry is *not* created, but instead the active change log entry is updated by appending those entries that were not previously written to it at the end of the entry. (Thus the user can edit comments in the change log entry and when that entry is subsequently updated, the comments will not be touched.)

Often a single edit session will logically span several boot or rollback operations, e.g. the user makes some edits, recompiles, rollsback, reloads the file, discovers some additional problems which require more editing. In this case, it is desirable to have the latter set of edits look as though they had been performed in the same session as those performed a short while previously. In order to accomodate this behaviour, if a previous change log entry is found, and this entry is dated within 24 hours of the current time, the current set of changes will be merged with those in the previous entry, i.e. the effect is the same as though they occurred in the same edit session. (The test for whether or not such an entry exists is to search backwards from the end of the file looking at nodes with comment properties. If a node is found which contains a date, and one of the node's children begins with the characters "changes to: ", then the test succeeds. The test fails at the first non-empty, non-comment node.)

The user can override the merging of change logs and force a new change log entry by yellow-clicking ChangeLog. Similarly, the user can explicitly request merging with the previous change log, even if more than 24 hours have elapsed (e.g. to continue Friday afternoon edits on Monday morning) by blue-clicking ChangeLog.

Note that if the user clears or destroys the viewer, and then subsequently opens another viewer on the same document, or if the user deletes the active change log entry from the document, the effect is the same as though the user had booted or rolledback, i.e. a new session is started unless the previous change log entry occurred within 24 hours, etc.

Saving the file

In many situations, the user does not want to insert any comments in the change log, but simply wants to save the file with the list of changes. Rather than forcing the user to first click ChangeLog and then click Save, in those cases where the user has already indicated that he wants a change log maintained for this file, either by virtue of having previously created one by clicking ChangeLog, or by an appropriate declaration in his user profile, the user can simply click Save. Thus

(a) If there is an active change log entry, Red-clicking Save will update it, i.e. is the same as red-clicking ChangeLog and then doing the save.

(b) If the user has an entry in his profile of the form UpdateChangeLog: **addIfPrevious** (the default), and a change log entry was inserted by this facility in some previous edit session, then merge the two entries if their times are within 24 hours, otherwise create a new entry.

(c) If the user has an entry in his profile of the form UpdateChangeLog: **createNew**, and the file looks like a mesa file, then proceed as in (b). (The test for this is to search backwards from the end of the file for the first non-empty, non-comment node. This node must begin with the characters "END.". Note that this test will always fail for files that do not use Tioga node structure.)

(d) If you are feeling paranoid, Blue-clicking Save will simply cause the file to be saved without doing anything to the change log (or the Last-Edited date), regardless of your user profile, and regardless of whether a change log entry is currently active.

Summary

RedClick ChangeLog

If there is an active change log entry, update it, otherwise if there is a change log entry in the file created within the past 24 hours, merge with this entry, otherwise create a new change log entry.

YellowClick ChangeLog

Create a new change log entry.

BlueClick ChangeLog

Merge with previous change log entry if one exists,, regardless of when it was written, otherwise Create a new change log entry.

RedClick Save

If there is an active change log entry, update it, otherwise, if UpdateChangeLog = addIfPrevious, and there is a previous change log entry, create a new one (or merge with previous one if within 24 hours, otherwise, if UpdateChangeLog = createNew and file looks like a mesa file, create a new one or merge with previous one if within 24 hours.

BlueClick Save

Just write out the file.

Note that the system will never mess with your change log unless:

(1) You explicitly click ChangeLog.

(2) You explicitly clicked ChangeLog once before in this edit session.

(3) Your user profile contains UpdateChangeLog: addIfPrevious (the default), and there is already at least one change log entry there of the recognized form. In other words, in this mode, you have to start each file off with a change log the first time by explicitly yellow-clicking ChangeLog.

(4) Your user profile contains UpdateChangeLog: createNew, and the file looks like a mesa file. In other words, in this mode, change logs will automatically be started for you, but only if the file looks like a mesa file with node structure.

The system will never add duplicate entries to the change log.

The system will never disturb any changes that you make by hand to your change log, e.g. editing comments.

Remember: this facility is designed to be used ONLY in conjunction with Tioga node structures. The change log will be singularly uninteresting for a flat structured document, since all of the changes will appear to be at the same place.

Master Change Log

A facility for generating a log of all of the files that you save during a session is automatically enabled when NewStuffImpl is loaded, unless you have an entry in your user profile of the form **CreateChangesLog: FALSE**. This log is kept on the file Changes.Log which is created when it is first needed. Each time you save a file, the date, file name, and changes log entry that was inserted in your file via the facility Updating Changes Log discussed above, if there is such an entry, is added to this log. Here is an example of such a log:

```
Changes log created at 10-Mar-83 15:20:27 PST

 Changed File: my.dfs
    Edited on 10-Mar-83 15:20:27 PST (no change log entry)

 Changed File: io.DF
    Edited on 10-Mar-83 15:52:10 PST (no change log entry)

 Changed File: teitelman.profile
    Edited on 10-Mar-83 17:16:44 PST (no change log entry)

 Changed File: UserProfile.doc
    Edited on March 10, 1983 5:22 pm, by Teitelman
       changes to: Hardcopy

 Changed File: UserExecImpl.df
    Edited on 10-Mar-83 17:23:44 PST (no change log entry)
```

The first time in a particular session that you save a file, a new Changes.Log file is created, and the existing one renamed to be Changes.Log$. For the purposes of this facility, a new session starts when you boot: rolling back reopens the existing change log, if any, and continues the previous session.

Remembering Previous Search

Whenever a Tioga document is searched via the Find, Word, or Def menu buttons, the target of the search is saved. If the target for this particular search, i.e. the current selection, is either empty (point selection), or consists of a single character, then the previous target is used instead. This enables searching for Foo, deleting it, and then searching for the next Foo by simply clicking the menu button again. It also enables searching for a particular target, changing the input focus in order to type something somewhere else, for example, to obtain a new viewer and load it with a particular file, and then searching again without having to reselect the target.

Swapping icons with opened viewers

The viewers package allows you to load an existing viewer with a file while requesting that the current contents of the viewer be made iconic. This is accomplished by selecting the name of the file, and then right clicking get. NewStuff extends this feature to allow you to select an icon, and then right-click get in a viewer. This causes the icon to be opened in place of the viewer, and the viewer to be made iconic, i.e. the two are swapped. Similarly, selecting an icon and then middle clicking get will open the icon just below the viewer, and left clicking get will cause the icon to replace the viewer.

Inter-Office Memorandum

To	Cedar Implementors	Date	May 26, 1982
From	Roy Levin	Location	Palo Alto
Subject	Cedar Releases: Policies and Procedures	Organization	PARC/CSL

XEROX

Filed on: [Indigo]<Cedar>Documentation>ReleaseProcedures.press

This memo describes the policies and procedures that individuals contributing to Cedar releases need to understand and observe. It assumes general familiarity with Cedar and DF files.

The Release Process

Components and DF Files

Throughout this memo, we call the unit of software submitted to the Cedar release process a *component.* Each component is described by a *DF file*, a simple description mechanism that is chiefly a list of file names (complete IFS paths) and creation dates (either explicit or implicit). This memo assumes familiarity with DF files; complete information may be found in the reference manual stored on [Indigo]<Cedar>Documentation>DFFilesRefMan.press. On occasion, a particularly large or complicated component may be more easily described by a collection of DF files. This is acceptable, but there must be a single "root" of the collection which includes (in the technical sense) the rest. A DF file contains information that describes the component for three distinct but related purposes. First, it identifies everything needed by the implementor to build the component from scratch. Second, it identifies the subset of files that are necessary for a client of the component. Third, it describes the way in which the files that make up the component are to be distributed at release time. It is the responsibility of the implementor to ensure that a DF file intended for release as part of Cedar fulfills all three of these descriptive functions; this memo contains rules that help him in doing so.

Release Phases

A release cycle consists of a *development phase* following by an *integration phase* culminated by a *release.* There is an individual, the Release Master, who monitors the release cycle and is responsible for the orderly progression from one phase to the next. The development phase lasts for days or weeks and is a period of largely unconstrained program modification and enhancement. The scope and duration of the development is release-dependent and is generally not completely specified when the development phase begins, although each implementor generally knows approximately what he intends to contribute to the next release. At some point, the Release Master initiates the integration phase and the following sequence of events then occurs:

1) The Release Master sends a message to CedarImplementors↑.pa announcing the beginning of the integration phase. Obviously, you must be a member of CedarImplementors↑ pa to receive this message: if you are a new implementor, you should verify that you are on this list. The message requests implementors to complete the development of their release components and submit them by responding to the Release Master with a specific

message form. Both the contents of the form and the submission itself must satisfy a number of rules before the Release Master will accept the submission for inclusion in the upcoming release. These rules are presented in detail later in this memo. The message announcing a submission is also sent to CedarImplementors↑.pa so that other implementors can complete the preparation of their components.

A component cannot be submitted to the Release Master until all of the components it depends upon have been announced. Experience shows that, if the release is expected to contain changes to fundamental interfaces (e.g., Pilot, Rope, IOStream), the Release Master should probably send the "call for integration" message only to the implementors of those central components. Once these submissions have been accepted, the Release Master can then send the "call for integration" to all implementors.

2) The Release Master assembles a top-level description of the contents of the release and runs the Release Tool. This program has three phases, the first two of which check the release submissions for consistency and completeness. The third phase moves the files to the release directory and produces descriptions (DF files) of the release contents. Generally, however, the Release Tool will detect some problems during the execution of its first two phases.

3) The Release Master analyzes the error messages produced by the initial phases of the Release Tool and contacts the implementors of the components that are in error. *The integration phase of the release cycle is stalled until the implementors correct their erros;* consequently, it is the responsibility of the implementors to do so as soon as possible. They notify the Release Master when they completed the necessary repairs.

4) Steps 2 and 3 are repeated until the Release Master is satisfied of the consistency and completeness of the submissions. The Release Master then sends a message to CedarUsers↑.pa announcing that a release is imminent and that the release directory will be in an inconsistent state while the release is installed.

5) The Release Master runs the third phase of the Release Tool, which copies the release files to the release directory. The Release Master tries to make this phase as brief as possible, anticipating potentially disabling problems (e.g., insufficient disk space on the release directory) and taking the necessary precautions.

6) When the release has been completely installed, the Release Master sends a message to CedarUsers↑.pa announcing the release. The Release Master should compose this message during step 5 in order to minimize the time during which the release directory is inconsistent. The Release Master stores a copy of the release message on the documentation subdirectory of the release directory, then prints and saves the logs produced by the Release Tool. The release is then complete.

The Need for Rules

A Cedar release typically includes thousands of files and hundreds of DF files, the latter with a complex interconnection structure. It is not surprising that errors are committed during the integration phase that force steps 3 and 4 to be repeated several times. Attention to the details of preparing a release submission can significantly reduce the duration of the integration phase. Experience shows that the three phases of the Release Tool can be completed in a few hours for a major release, but, nevertheless, such releases generally take two days. The rest of the time is chiefly spent waiting for implementors to correct errors in their submissions. It is important to understand that normally these are not programming bugs in the usual sense; rather, they are errors in the DF files that describe the components. Certain kinds of errors occur again and again, release after release, and the rules below are intended to eliminate these recurring problems.

These rules should be considered stronger than guidelines or suggestions but somewhat weaker than commandments. Uniformity promotes smooth and speedy releases, but exceptional cases invariably arise that do not fit the general model. Occasional reasonable deviations will be tolerated, but the Release Master establishes the definition of "reasonable". Unreasonable deviations jeopardize the release process and cannot be permitted. In all cases the implementor assumes complete responsibility for any deviation from these rules. The Release Master will not correct errors in submissions and will delay a release as necessary until an offending submission conforms to the requirements below.

Rules for Constructing a Cedar Release DF File

The following is a complete list of requirements that a DF file must satisfy in order to be acceptable as a component of a Cedar release. Unless explicitly indicated otherwise, the terms "import" and "export" refer to the Imports and Exports clauses in a DF file and not the similarly-named Cedar language notions. Also, the phrase "released to [X]<Y>Z>" is an abbreviation for "has a ReleaseAs clause with path [X]<Y>Z>".

1) *The DF file name is considered to be the "name" of the component and should generally correspond to the name of the major exported interface (in the Mesa sense) supplied by the component.* This rule is most easily applied to straightforward packages with a single public interface.

2) *The DF file must contain a self--reference that is released to [Indigo]<Cedar>Top>. The self-reference must be exported.*

3) *If separate documentation files accompany the component, they should be included in the DF file and released to [Indigo]<Cedar>Documentation>. Documentation files in suitable form for online use (not press files) should also be exported.* It is not mandatory that documentation files accompany a component; comments in the public interfaces may be sufficient for use. However, most substantial components will have separate documentation files, and these should be included in the DF file as indicated. If a component has recently undergone a substantial revision that is visible to its users, the implementor should probably release separate files containing the "change summary" and the "complete truth".

4) *All component-specific files other than the DF file and documentation should be released to a subdirectory with the same name as the component (e.g., the interfaces and implementation of SomePackage.df should be released to [Indigo]<Cedar>SomePackage>.* It is permissible to have additional subdirectory structure within the component-specific subdirectory. Subdirectories of either [Indigo]<Cedar>Top> or [Indigo]<Cedar>Documentation> are not permitted.

5) *If the component supplies a "public interface", the source and object files must be exported from the DF file.* ·The debugging and help facilities tend to work better if they can access the sources of interfaces as well as their compiled representations. Disk space concerns are irrelevant here, since the files involved are small and improvements in the file system will soon eliminate the size limitations of the local disk.

6) *The BCD or BCDs that contain the bound-up implementation of the component must be exported from the DF file. Exception: if the implementation is intended for inclusion in the Cedar boot file, it should not be exported.* The motivation here is obvious; the component can't be used unless its implementation is available.

7) *The entry in the DF file for the bound-up implementation of the component must be preceded by a "+" to indicate that it is a root BCD for processing by VerifyDF.* If the component exports multiple BCDs, each of them must obey this rule. It is permissible to have other BCDs marked with a "+", e.g., test programs.

8) *A file that is not a part of the component should be referenced using an "Imports" entry with a USING list.* The entry forms introduced by "ReadOnly" and "@", while documented in the DF reference manual, are obsolete and should not be used. The USING list documents explicitly dependencies on other components, protects against certain mistakes in imported DF files, and speeds up BringOver.

9) *An import from the release directory may include either an explicit or an implicit (i.e., ">" or "~=") date. An import from a working directory must specify an implicit date.* While these rules may seem a bit unintuitive at first, experience shows that they greatly streamline the release process. Last-minute, minor changes in a DF file are common during a release integration, and a failure to observe the second rule in a DF file depending on the one that changed forces an otherwise unnecessary reconstruction of the dependent DF file. The last-minute changes are rarely significant to the importer(s); when they are, inconsistencies will detected by the release machinery.

10) *The DF file for a component should usually include a text file containing command lines for compiling and binding the package.* This is not mandatory, but strongly recommended. It is not necessary that this file be a complete command file; rather, it is intended more as an documentary aid to someone unfamiliar with the structure of the component who finds it necessary to make a small change and rebuild it. There is no standardization on the name of such text files; most have the extension ".cm" (whether or not they are true command files) and many include the word "Make" and the component name in their names.

11) *If programs other than the Compiler and Binder are needed to build the component, entries importing them should be included in the DF file.* Examples of such programs include the Packager, MakeBoot, and the TableCompiler. If specific versions of the Compiler and/or Binder are required (a rare situation), entries for them should be included as well; however, it is not necessary to reference the standard ones.

12) *The working directory used for all non-imported files must grant read and create access to CedarAdministrators↑.PA.* The need for read access is obvious; the component can't be released unless it can be read. The need for create access is rarely exercised but is occasionally needed when a small change to a component is required in order to complete a release integration and the implementor is not available. Note that the heavily-used working directories [Indigo]<CedarLang> and [Indigo]<CedarLib> satisfy these requirements, while personal directories typically do not. Implementors who use their personal directories for release submissions must explicitly alter the IFS access controls to their directories to conform to this rule.

A Sample DF File

The DF file below satisfies the requirements for submission to a Cedar release. It describes a component named `Sample`, which consists of a single configuration named `SampleImpl`. `Sample` has two public interfaces, named `Sample` and `SampleExtras`. The implementation consists of two modules, `SampleImplA` and `SampleImplB`, which share a private definitions module named `SamplePrivate`. It requires three other components, defined by DF files named `PilotInterfaces.df`, `Rigging.df`, and `Runtime.df`. Finally, the component is documented by a press file whose source is a Tioga document.

Notations of the form {n} are not actually part of the DF file. They indicate that the entry so marked conforms to rule n, above.

```
// Sample.df
// last edited by Harry Bovik: May 10, 1982  5:26 PM

Exports [Ivy]<Bovik>Sample>     ReleaseAs [Indigo]<Cedar>Top>    {2}

  Sample.df!3                                10-May-82 11:47:21 PDT  {1}

Exports [Ivy]<Bovik>Sample>     ReleaseAs [Indigo]<Cedar>Sample> {2}

  Sample.bcd!2                                9-May-82 17:45:27 PDT {3}
  Sample.mesa!2                               9-May-82 17:44:55 PDT {3}
  SampleExtras.bcd!1                         10-May-82 11:45:31 PDT {3}
  SampleExtras.mesa!1                        10-May-82 11:33:19 PDT {3}

  +SampleImpl.bcd!7                          10-May-82 14:53:09 PDT {6,7}

Directory [Ivy]<Bovik>Sample>     ReleaseAs [Indigo]<Cedar>Sample> {4}

  SampleImpl.config!2                        10-May-82 13:24:52 PDT
  MakeSample.cm!3                            10-May-82 13:25:19 PDT {11}

  SamplePrivate.bcd!2                        10-May-82 12:36:12 PDT
  SamplePrivate.mesa!1                       10-May-82 12:34:51 PDT

  SampleImplA.bcd!7                          10-May-82 14:47:39 PDT
  SampleImplA.mesa!6                         10-May-82 14:47:21 PDT
  SampleImplB.bcd!4                          10-May-82 14:47:43 PDT
  SampleImplB.mesa!2                         10-May-82 14:32:06 PDT

Exports [Ivy]<Bovik>Sample>  ReleaseAs [Indigo]<Cedar>Documentation> {4}

  Sample.press!2                              6-May-82  9:40:18 PDT
  Sample.tioga!2                              6-May-82  9:37:23 PDT

Imports [Indigo]<Cedar>Top>PilotInterfaces.df!1 Of 1-Apr-82 15:43:56 PST
    Using [Environment.bcd, Process.bcd] {8,9}

Imports [Indigo]<Cedar>Top>Rigging.df!9 Of 7-May-82 20:18:46 PDT
    Using [Rope.bcd] {8,9}

Imports [Indigo]<CedarLang>Runtime>Runtime.df Of >
    Using [SafeStorage.bcd] {8,9}
```

Notice that the files that make up the component are all stored on a working directory chosen by the implementor, in this case his personal directory on Ivy. (Note that, in accordance with rule 12, Bovik must grant create access on [Ivy]<Bovik> to CedarAdministrators↑.pa.) The DF file specifies the locations on a release directory, [Indigo]<Cedar>, where the files are to be stored at release time. The component depends on three other components, two of which have been previously released (since they are on the release directory) and one of which (Runtime.df) is not (and presumably will be released when Sample is).

Rules for Submitting a Component for Release

The message from the Release Master that announces the integration phase of a release contains a message form. This form, properly completed and returned, serves three important functions:

1) It enables the Release Master to include the DF file describing the component in the top-level description of the release.

2) It notifies other implementors that the component is now available for reference in other DF files.

3) It provides the information to be included in the release message that will announce the release to all Cedar users.

The message form has been constructed to facilitate all three of these purposes. Since deviations from the standard form complicate and delay the integration phase, implementors must observe the following rules for submission of components:

13) *The component must satisfy the construction rules before the submission form is sent.* Sending the form before the DF files are ready and stored confuses everyone.

14) *The implementor must have run VerifyDF on the component's DF file before submitting it for release.* VerifyDF performs most of the completeness checks that the Release Tool's second phase does; consequently, most errors of this form can be detected early in the integration phase and without delaying other implementors.

15) *The submission form is the only acceptable way of announcing a release submission.* Free-form messages are not acceptable and will be rejected by the Release Master. If an implementor expects to be unavailable during the integration phase and wishes to prepare his submission "in advance", it is his responsibility to obtain from the Release Master a copy of the form and use it.

16) *All fields of the form must be completed appropriately.* In particular, the working path and release path must be filled in correctly and expressed in conventional syntax (server name in square brackets, directory and subdirectories in angle brackets); "/" syntax is not acceptable. The documentation pointer must conform to the requirements stated on the form. The portion of the form that contains information destined for the release message should be filled in judiciously; it is generally not necessary or desirable to describe every bug fix and minor enhancement. The reference documentation should include this kind of information, while the release message should contain a summary of the significant changes in the component since the last release. A chronology of changes is *not* appropriate. The release message portion of the form must also be coherent English prose that observes accepted standards of capitalization, spelling, punctuation, and grammar.

Avoiding Common Mistakes

When the integration phase of a release cycle is underway, it is easy to forget one or more of the above rules. Experience has shown that some rules are more likely to be broken than others and that certain mistakes occur repeatedly. Careful observation of the following rules will help implementors avoid the most common errors that delay release integrations.

17) *When beginning development of a component following a release, use the released DF file as the base, not the working DF file supplied to the previous release.* Obsolete working DF files are perhaps the single greatest source of trouble in the integration phase. When an implementor uses a working DF file from a previous release as the base for new development, he invites inconsistencies that cannot be easily detected until the Release Tool is run. The typical problem is a reference to a working DF file that is either obsolete or non-existant. SModel can only detect the latter of these problems, and then typically only if /v is specified.

18) *Be certain that all imports come from the correct location (working directory or release directory).* In a sense, this rule includes the preceding one, since incorrect imports frequently arise when an obsolete DF file has been used as the starting point for development of the component.

19) *When correcting a problem uncovered by the Release Tool during the integration phase, always run VerifyDF before notifying the Release Master that the component is again ready for inclusion.* It is quite common for phase one of the Release Tool to uncover a problem in a DF file which, when "fixed" quickly by the implementor, leads to a problem in phase two. This causes unnecessary delay while the implementor once again fixes the problem; running VerifyDF before the first resubmission will nearly always avoid such delays.

20) *Before submitting a component for release, double-check the subdirectories to which files are being released.* The Release Tool cannot, in general, check that rules 2-4 are obeyed. Uniformity here greatly simplifies browsing.

21) *After successfully running VerifyDF (rule 19), be sure to run SModel /v,.* In the course of making a DF file acceptable to VerifyDF, it is useful to run SModel /n, which, although it improves the state of the files on the local disk, performs no transfers to file servers. SModel /v ensures that the cumulative effect of repeated runs of SModel /n is moved to the file servers.

Cedar Document Style — title node

A Sample Sheet — subtitle node

Release as	[Indigo]<Cedar>Documentation>SampleSheet.tioga, .press
Came from	[Indigo]<CedarDocs>Manual>SampleSheet.tioga, .press
Last edited	By Horning on December 17, 1982 12:11 pm by Beach, June 6, 1983 11:37 am

Abstract: This sampler is intended to show "some of everything" from the standard Cedar.style for use with Tioga. It should serve as a reminder of the formats and looks available for preparing uniformly formatted Cedar programs, documentation, and memoranda. It also contains itemized lists of the formats and looks. Mostly, though, the contents are a jumble, words collected to add versimilitude to an otherwise bald and unconvincing assemblage

XEROX

Xerox Corporation — boilerplate node
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

DRAFT - For Internal Xerox Use Only - DRAFT

Contents — contents nodes, begins new page

Node Format Sampler — This is a head Node, begins a new page

This is a **body** node, nested under the head node (actually, in the *branch* that it heads). Note that nesting is displayed on the screen by a small indent which is *not* reflected in the hardcopy form.

Some people find that this indentation helps them to keep track of the nesting structure of their documents. If, on the other hand, you find it more distracting than helpful, you should change the ScreenRule for Nest in Cedar.style to be the same as the PrintRule. Some node formats cause indentation that is independent of nesting. This indentation is larger than the indentation that indicates nesting. This **note** node illustrates the difference. It is *not* nested in the body node above.

However, this note node is nested in the note above it, and hence gets the sum of the two relative indentations.

This is a nested head node

This is a **body** node, nested in the head node. This is a **body** node, nested in the head node. This is a **body** node, nested in the head node. This is a **body** node, nested in the head node.

This is a **block** node, also nested in the head node.

This is an **indent** node, also nested in the head node.

This indent node is nested in the previous one (hence indented relative to it).

This is a **quote** node. It is nested in the head node, not the indent node.

And this is a **center** node, nested in the head node.

*This is a **head** node, also nested in the head node*

This is a body node, nested in the head, with an embedded program fragment.

```
Sample: CEDAR PROGRAM            -- This is an example node
    IMPORTS Something               -- Online comment in a code node
    EXPORTS Another
    = { IF a < b THEN a ← b }.
```

This is a **continuation** node, nested under the body node with the embedded program fragment.

This is an **item** node. Item nodes are typically nested in other nodes, and contain the elements of displayed lists (note the hanging indent).

This is another item node.

This **lead2** node is set with more lèading, which leaves more space between lines, e.g., to accomodate LARGE TYPE. The choices are **lead1**, **lead2**, and **lead3**.

This	is	a
table	node.	Only
simple	tabs	are available.

A	**table1**	node	has
smaller	tabs	for	more columns.

A	**table2**	node	has	even
smaller	tabs	for	still	more.

While	a	**table3**	node	has	yet
smaller	tabs	for	lots	of	columns.

This is a **pageBreak** node. (Contents are appropriate only if they are to appear at the very bottom of the page. Not needed before first level **head** nodes.)

Node Formats

Much of the control of the format of a document can be handled consistently by associating "formats" with nodes of the document. The set of formats selected for Cedar is intended to cover the common kinds of textual units in documentation, memos, and programs. It is best to pick a small subset of them for any particular document.

Formats for use in programs

Format **code** is intended for program statements that do not require special spacing. Nesting is reflected by indentation.

Format **unit** is intended for program statements that start a new logical unit, and hence should have some extra space in front.

Formats for use in prose

Format **block** is intended for block-style paragraphs.

Format **body** is intended for ordinary paragraphs with the first line indented.

Format **center** is intended for centered paragraphs.

Format **contents** is intended for headings in tables of contents. The same type fonts are used as for **head**, but with less leading, and with indentation corresponding to nesting structure.

Format **continuation** is intended for nodes that logically "continue" previous nodes (e.g., following an example or an itemized list), and hence should not have extra leading at the beginning.

Format **display** is intended for displayed equations, etc. embedded in text paragraphs.

Format **example** is intended for things like program statements included within explanatory text, and gets extra indentation to set it off.

Format **head** is a "generic" format for headings that will expand to one of **head1**, . . ., **head5**, as a function of the level of nesting. If it is used in place of the more specific head formats, heading nodes can be copied, levels can be added or removed, etc., with minimum effort.

Format **indent** is intended for paragraphs that are to be indented (relative to the containing node). In the usual case, this is a head, rather than the preceding body node, although the difference is not visible on the hardcopy.

Format **item** is intended for elements of an itemized list (like this one). Items are indented relative to their containing node, and continuation lines are further indented.

Formats **lead1**, **lead2**, and **lead3** are like **body**, except that they have progressively more inter-line leading, which can be useful for paragraphs with larger format, superscripts, etc.

Format **logo** is intended for the Xerox logo line at the start of a document. It does more than just set the font to Logo24, it provides extra space before and after, extends into left margin, etc.

Format **memoHead** is intended for the heading lines at the start of memos.

Format **note** is intended for fine points. Someday we will have real footnotes.

Format **pageBreak** is intended to force a new page. It shouldn't have content—unless you want it at the very bottom of the page.

Format **quote** is intended for displayed quotations.

Format **reference** is intended for items in reference lists [van Leunen 1978].

Format **table** is intended for tables with widely-spaced columns.

Formats **table1**, **table2**, and **table3** provide successively smaller tab stops to accomodate more columns.

Format **title** is intended for titles, typically at the start of a document.

Format **subtitle** uses a slightly smaller font, and can be used under a title.

Looks

It is good practice to use the "abstract" (or "intentional") looks wherever possible (e.g., **Annotation**, rather than **Bold**, *Emphasis*, rather than *Italic*, Keyword, rather than Smaller). This will make it possible to produce other versions of the document where different decisions have been made about the appropriate way to display some abstract property. Also, it is better to avoid looks entirely, if the information can be captured in the format of the containing node (none of the headings in this document have **Bold** looks).

Looks **a** **b** *c* $_{d}$ *e* f γ h *i* j k l ll **n** o p r s t ^{u} v w X ~~y~~ z.

look.a	annotation font
look.b	bold font
look.c	Cedar comments
look.d	down for subscript—lower (not yet visible on screen) and a smaller font
look.e	emphasis
look.f	fixed-pitch font—Gacha
λοοκ.g	Greek—Hippo
look.h	hidden by asterisks (unfortunately prints normally)
look.i	italic
look.k	Cedar KEYWORDS
look.l	larger font
ꓶooꓶ.m	Math font
look.n	Cedar procedure names
look.o	other font—Helvetica
look.p	plain font and regular size—overrides node format
look.q	quaint XEROX logo – for boilerplate format
look.r	regular face—overrides node format
look.s	smaller font
look.t	Tioga font
look.u	up, for superscript—higher (not yet visible on screen) and a smaller font
look.x	extra large
~~look~~.y	y—arbitrary for **strikeout**
look.z	z—arbitrary for **underlined**

Look.s Format **block.** Mohammed (also Mahomet, Muhammad; 570?-632) asserted a doctnne of unqualified monotheism (suras 8, 22, 33-37, 89, 91, Koran).

Look.l Format **lead1.** Scholarly writing is formal, accurate, and allusive. It has to be. It does not have to be wooden, finicking, and cabalistic. The idea of this book is to help you achieve the first set of characteristics without sinking into the second.

Look.x Format **lead2.** "There is no God but Allah."

Of course, *α variety of* **looks** ~~can be mixed~~ in a single paragraph, although the appe^{a}rance of the result is *often* *somewhat less* than pleasing. $A_i \times B^j$

Programs

```
Cedar Comment Standing on its own as a unit node
File: SimpleExample.mesa                                                  (note 1.1)
Last edited by: Cattell on May 12, 1982 3:05 pm
DIRECTORY
  IOStream, Process, Rope, UserExec;
SimpleExample: MONITOR                                                   --(note 1.2)
IMPORTS IO: IOStream, Process, R: Rope, UserExec =                       --(note 1.3)
BEGIN
  ROPE: TYPE = R.ROPE                                          --(note 1.8)
  windowCount: INT ← 0; -- a count of the number of calculators on the screen
  ReverseName: UserExec.CommandProc = BEGIN                    --(note 1.4)
    Reverses the user's login name and prints it out in the exec window.
    userName: ROPE ← UserExec.GetNameAndPassword[ ].name;          --(note 1.5)
    execStream: IOStream.Handle ← exec.out; -- exec was passed as arg to this proc (note 1.6)
    backwordsName: ROPE ← NIL;
    Remove ".PA" if it is on the end of the user name, and check for user name Taylor.
    dotPos: INT ← userName.Find["."];                              --(note 1.7)
    IF dotPos#-1 THEN
      userName ← userName.Substr[0, dotPos];
    IF userName.Equal[s2: "Taylor", case: FALSE] THEN
      execStream.PutF["Hi, Bob\n"];
    Now reverse the name: convert chars to upper cases and concatenate them in reverse order
    FOR i: INT DECREASING IN [0..userName.Length[ ]) DO
      backwordsName ← backwordsName.Cat[R.FromChar[R.Upper[userName.Fetch[i]]]]
                                                                   --[1.1] (note 1.9)
      ENDLOOP;
    execStream.PutF["Your user name backwards is: %g\n", IO.rope[backwordsName]];
                                                                   --(note 1.13)
    END;
  Start code registers a Calculate and ReverseName command, which must be invoked for this
        program to do anything:
  UserExec.RegisterCommand[                                    --(note 1.17)
    name: "Calculate", proc: MakeCalculator, briefDoc: "A simple adding machine"];
  UserExec.RegisterCommand[
    name: "ReverseName", proc: ReverseName, briefDoc: "Reverses your user name"];
  END.
CHANGE LOG                                                               --(note 1.18)
  Created by Cattell on 21-Apr-82 13:55:09
  Changed by Cattell on May 25, 1982 10:58 am
    added use of RegisterCommand to fire up a Calculator viewer instead of creating it when
          program first run.
```

Text

Prose

Scholarly writing is formal, accurate, and allusive. It has to be. It does not have to be wooden, finicking, and cabalistic. The idea of this book is to help you achieve the first set of characteristics without sinking into the second.

Let us not deceive ourselves. "There is no God but Allah" is a more gripping sentence than

Mohammed (also Mahomet, Muhammad; 570?-632) asserted a doctrine of unqualified monotheism (suras 8, 22, 33-37, 89, 91, Koran).

By its very nature, scholarly prose lacks the rhetorical virtues of reckless passion. Custom and propriety hem the scholar in on every side. His obligation to his material and his obligation to his sources restrain him. He may not gloss over imperfections, smooth out irregularities, turn a blind eye to objections, no matter what good effects these temptations might offer in the way of clarity and simplicity.

On the other hand, the built-in limitations of scholarly prose are no excuse for bad writing. Bad scholarly prose results, as all bad prose does, from laziness and hurry and muddle. Good scholarly prose is probably even harder to produce than other kinds of good prose. All that means is that the scholar must work even harder at it. [van Leunen 1978]

References

[Carroll 1865]

Lewis Carroll
Alice's Adventures in Wonderland
Washington Square Press edition, 1960

[Knuth 1968]

Donald E. Knuth
The Art of Computer Programming, vol. 1
Addison-Wesley

[van Leunen 1978]

Mary-Claire van Leunen
A Handbook for Scholars
Knopf

Poetry — The Mouse's Tale

—"Fury said to
a mouse, That
he met in the
house, 'Let
us both go
to law: *I*
will prose-
cute *you.*—
come, I'll
take no de-
nial: We
must have
the trial;
For really
this morn-
ing I've
nothing
to do.'
Said the
mouse to
the cur,
'Such a
trial, dear
sir, With
no jury
or judge
would
be wast-
ing our
breath.'
'I'll be
judge,
I'll be
jury,
said
cun-
ning
old
Fury:
'I' ll
try the
whole
cause
and
con-
demn
you to
death.'
[Carroll 1865]

Inter-Office Memorandum

To	▶Recipients◀	Date	▶CTRL-T◀
From	▶Your name◀	Location	PARC/CSL
Subject	▶Topic◀	File	▶File Name◀

XEROX

head Node, repeat as needed, nest for subheads if appropriate

body node, repeat as needed

Computing Sciences Laboratory
Xerox Corporation
Palo Alto Research Center
Palo Alto, California 94304
415 494-4000

XEROX

▶Date◀

▶Destination Address Field◀

Dear ▶Name Field◀.

▶File: BusinessLetter.form This form is designed to match the current PARC letterhead. It is organized as a set of Tioga fields to be filled in. To use it, select the XEROX logo node above and use the NEXT key to select the first field to be replaced, the Destination Address Field (the NEXT key is the blank key second from the bottom on the right side of the keyboard). This is a body paragraph node and may be replaced by selecting it as a field and then simply typing◀

Sincerely,

▶Your Name Field◀

CSL Notebook Entry

To	▶Recipients◀	Date	▶CTRL-T◀
From	▶Your name◀	Location	PARC/CSL
Subject	▶Topic◀	File	▶File Name◀

XEROX

Release as	▶TargetFileName◀
Draft	▶WorkingFileName (this form is on [Indigo]<CedarDocs>Style>Memo.form)◀
Last edited	by ▶Person (this form was last edited by Rick Beach, June 6, 1983 11:31 am)◀

Abstract ▶A pithy descriptive paragraph, if appropriate◀

head Node, repeat as needed, nest for subheads if appropriate

body node, repeat as needed

Tioga TSetter

Typesetting Tioga Documents

Release as [Indigo]<Cedar>Documentation>TSetterDoc.Tioga
Last edited by Michael Plass on May 2, 1983 9:57 am.
Last edited by Rick Beach on May 31, 1983 2:23 pm

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

The TypeSetter Tool

In the glorious future, Tioga will have an interactive typesetter that will let you make incremental revisions to a typeset document to adjust pagination, page layout, and other details before actually printing it. Work has started on this, but until it's available, the current typesetter will do a more than adequate job of letting you print your files.

The TSetter tool provides a convenient way of driving the Tioga typesetter. Start it up by typing "TSetter" to the executive. The typesetter icon will come up showing the name of the print server it is "connected" to; this name can be specified in your profile under the heading "Hardcopy.PressPrinter", or as the first argument on the command line.

The remainder of the command line is taken as a list of file names to be printed. If this list is non-empty and the files are succesfully printed before you touch any of the tool's buttons, the tool will quietly go away.

When you open the typesetter icon, you will see the menu

Pause Stop StopSending Get Print All < Screen > New

plus a fill-in field labeled "Documents:". The normal way to use the typesetter is to select the viewer or file name, and click "Get" to add the document name to the queue. Then click "Print" to start the typesetter up; if all goes well, the only other thing you have to do next is to walk over to the printer and pick up your output.

You can put several documents in the queue before starting up the typesetter, and go do something else while it is working away. The queue is shown after the "Documents:" button and is in fact editable; it is a good idea to click "Pause" before editing, though, to keep the typesetter from taking the queue out from under you. Clicking "All" instead of "Print" funnels the whole queue into one big press file, which avoids a lot of header pages if you want to print many small files.

To print the bitmap on the screen, use the "Screen" button. The "<" button will print just the left half of the screen, the ">" the right. The half-screen output may be printed on a Spruce printer (e.g., Clover or Menlo) if it is not too complicated; otherwise, use a full Press printer (such as Stinger, Quoth, or Lilac).

Hitting "Stop" will bring the typesetter to a halt.

Hitting "StopSending" will abort any transmission to a print server.

Destroying a tool will not stop it; it will just finish running and then go away.

If you try to typeset a press file, it will just get sent to the server without modification. After it is successfully transmitted, it will be deleted if its name ends in a "$".

To make a TSetter tool that sends its output to a different place, type the server name somewhere, select it, and click "New". If the selection is less than two characters long, the resulting tool will be called "TSetter", and will just create a press file without trying to send it anywhere. The name of the press file is generated by appending ".press" to the input file name, unless the input file had a name of the form "xxx.tioga"; in the latter case the press file will be named just "xxx.press". It is OK to run multiple typesetter tools at the same time.

If you want to print multiple copies, change the "Copies" field near the bottom of the TSetter viewer. This number is reset to 1 after each file is transmitted to a server, to keep you from wasting a lot of paper by forgetting to reset it.

The "Temporary Press Files" button is followed by a Boolean value. If this is TRUE, the typesetter will create press files with names that end in "$", so that they will be deleted after they are successfully transmitted. Clicking the button toggles the Boolean. The initial setting of this flag is controlled by the user profile entry "Hardcopy.TemporaryPressFiles".

The feedback from the typesetter tool is logged. You can see the log by making the tool bigger and scrolling the appropriate subwindows. The top subwindow is for the formatting process and the bottom subwindow is for the spooling process.

Multiple column and landscape output is now possible. Look at [Indigo]<Cedar>Styles>*Print.style to see if there is a style there that will let you do what you want. If not, let TiogaImplementors↑.PA know. (You are welcome to try writing your own print styles, but be warned that they may not work in the future, as incompatible changes in this mechanism are planned.) As an example, you can get two column landscape output by making a style that has in it just the following:

```
BeginStyle
(Cedar) AttachStyle
(TwoColumnLandscapePrint) AttachStyle
EndStyle
```

and setting this little style on your document. (For now, you must make sure all the styles you need are on your disk.)

TSViewer Interface

To drive the typesetter from a program (or the executive), there is an interface called "TSViewer" that lets you create a tool and simulate clicks on its buttons.

TSExtras Interface

For more exotic formatting requirements of clients of Tioga, try the TSExtras interface. Interfaces for formatting a Tioga viewer or individual nodes of a Tioga document are provided.

Mark Properties

To implement running headers and footers, the TSetter "marks" special Tioga nodes. These marked nodes are later used for page layout purposes. In fact, the entire branch descendant from a marked node is retained. Caution is advised that text nodes are not inadvertenly nested under a marked node.

Marked nodes have a **Mark** property value which is a rope name. The present set of known mark property values is established by the page layout styles, e.g. BasicPrint.Style. No checking is done on mark property values, so they may seem to disappear when the document is printed if the property value was incorrectly named.

Page layout routines access the marks for the current page. Two operations are provided for extracting the first and last marks of a given kind defined on a page. At the end of each page the list of marks is pruned of duplicates. This provides for normal running headers and footers as implemented by BasicPrint.Style, as well as for more exotic dictionary slug footers.

Inter-Office Memorandum

To	Cedar Interest	Date	December 21, 1982
From	Scott McGregor	Location	Palo Alto
Subject	The Viewers Window Package	Organization	PARC/ISL

XEROX

Filed on: [Indigo]<Cedar>Documentation>ViewerDoc.Tioga and ViewerDoc.Press.

Documents: [Indigo]<Cedar>Viewers>Viewers.df, as exported by the Cedar boot file.

The Viewers Window Package

Disclaimer

This document is currently in progress and hence is incomplete (as witnessed by a number of sections not yet written). It reflects the state of the Viewers package for Cedar version 4.0. You may be able to find a more recent version on [Indigo]<CedarViewers>Viewers>ViewerDoc.tioga.

Introduction

The Viewers Window Package is the arbiter of the user input and display hardware in the Cedar programming environment. It provides the illusion to the programmer that there is a private display, mouse and keyboard associated with each application, while allowing the user to simultaneously interact with many such applications.

The basic object manipulated by client programs and visible to the user is the *viewer*, a rectangular area with arbitrary contents which may be made visible on the user display. A viewer takes its name in that it allows the human user to *view* and interact with the data associated with a Cedar application. The underlying applications software has complete control over the displayed contents of a viewer and has available a rich user interface for user input. The screen position and size of a viewer may be modified by the user as well as under program control.

This documentation is written for the programmer intending to use the Viewers Window Package to build a new application. It is organised along the broad areas of functionality that the Viewers system provides and attempts to explain design theory and some pragmatics. For examples of usage, see the references within each section, and for exact details consult the interfaces directly. One point of notation: used throughout this document, *client* refers to a program calling the Viewers interfaces, whereas *user* refers to the human invoking Viewers operations with the mouse and keyboard.

Screen Layout

The User Desktop

From the user point of view, the black and white screen is divided into four areas. The top quarter inch of the display is used for the messages and a small set of system command buttons. The bottom of the display is a variable height area reserved for displaying icons, which are small pictorial representations of normal viewers made iconic to conserve screen space. The remainder of the display is divided into left and right columns with a moveable partition between them. The user may optionally attach a color display, which is equivalent to an additional color column.

Unlike many window packages, viewers do not normally overlap, but instead spread out to cover as much as possible of the available space on the user display. Within a column, the height of a particular viewer is determined by satisfying a number of constraints set by the Viewers system, the implementing program and the user. Viewers constraints include balancing of the columns; i.e. viewers will be expanded so that the entire column area is used, and that no viewer is smaller than the height required for the caption and command menu at the top of each viewer. A client program may specify a height hint in the viewer instance's *openHeight* field, which will be honored if it doesn't conflict with other constraints. In addition, the user may set a height hint that overrides the *openHeight* value by invoking the *Adjust* menu command. User-set hints are discarded when a viewer is made iconic.

> A fine point: The current implementation does not deal particularly well with overcontrained columns; i.e. when the sum of the Viewers enforced heights plus the client openHeight hints exceeds the height of the column. In order to solve the conflicting constraints, each viewer is given one nth of available column height. Ideally, Viewers should either pro-rate the existing space according to the client requests or force some of the viewers in the overconstrained column to become iconic.

Internally, the tree of viewers is partitioned into four subtrees, corresponding to *static* (typically those viewers with a permanent, fixed position on the screen), the *left* column, the *right* column, and the *color* display. Viewers in the static subtree include all icons, the message window at the top of the display, as well as the system buttons located in the upper right hand corner.

The boundary between the left and right columns as well as the height of the columns may be set by invoking the *Adjust* menu command in any viewer and then sliding the mouse across the column boundary. The column height is quantized to the height of each row of icons, permitting up to two rows of icons or as few as none. Icons are displayed behind the columns; in other words, viewers in a column overlapping the display of an icon will hide the icon until the user shortens the column, or removes all the viewers from that column.

Multiple Desktops

Users can save configurations of viewers in a special viewer called a desktop. The user can then move back and forth between different configurations of viewers with a simple command. There is always one desktop which represents the configuration of viewers currently on the screen. This desktop (called the current desktop) is indicated on its icon by having its name inverted. The user can "fly" to another desktop by middle clicking the desktop icon while holding down the control key. The viewers and icons on the screen will be stuffed into the current desktop, to be replaced by the viewers and icons in the new desktop. The new desktop then becomes current.

To get a new desktop, type "Desktop 'name'" to the UserExec. This will create an icon that looks like a minature screen. The first desktop created becomes the current one. A second desktop must be created before you can move to a new desktop. When you fly to the new desktop, only desktops and the UserExec will go with you, all other viewers will be stored in hyperspace. Viewers can be brought over one by one as described below.

To move a viewer from another desktop to the screen, open the desktop icon and left-click the button representing the viewer. This will move the viewer from that desktop onto the screen. If the viewer is already on the screen but iconic, clicking the button will open the viewer. Clicking the button with the shift key down causes the viewer to grow to full column. Viewers can be added to a desktop by selecting the viewer and clicking "AddSelected" in the desktop's menu. This will cause the viewer to be removed from the screen.

To move an individual viewer from the screen onto a particular desktop, open the desktop icon, select the viewer and left-click "AddSelected" in the menu of the desktop. The viewer will then disappear from the screen. You can remove a viewer from a desktop by clicking its button with control held down. If this was the last reference to the viewer, it will automatically be moved back onto the screen. Destroying a desktop viewer may also cause hidden viewers to appear on the screen.

Although some viewers may be inaccessible to the user because they are on different desktops, all viewers are accessible to client programs via ViewerOps.EnumerateViewers. This means that programs don't have to enumerate desktops to see all of the viewers. The bit "offDesktop" will tell the client program whether the viewer is currently accessible to the user. Opening, closing, repositioning, or blinking a viewer will automatically move it onto the screen, whether or not it was on the screen before. Painting a viewer will not. If the client program wants to manually move a viewer onto the screen, it can do so with ViewerOps.ChangeColumn[viewer, {left, right, color}]. All viewers are stored off the screen as icons, therefore a program should only have to worry about manually moving viewers onto the screen if it wants the user to see it repaint an icon.

Viewers

The display of a viewer consists of two nested rectangles. The outer rectangle bounds the viewer window *overhead*, consisting of space for an optional black border around the viewer, an optional scrollbar space on the left, a black caption bar at the top, and an optional menu below the caption containing an arbitrary number of menu lines. The inner rectangle defines the viewer *client* area, which is available to an application for display of its data.

Each viewer has a parent, a sibling and a child viewer, any of which may be NIL. Viewers may be recursively nested within (i.e. children of) other viewers, in which case display of the caption and menu will be supressed and no space reserved in the outer window rectangle.

The coordinates for the window and client rectangles are computed with respect to the enclosing parent's client rectangle origin. If the viewer has no parent, it is known as a *top level viewer* and its rectangle coordinates are with respect to the lower right hand corner of the display. A viewer may extend outside of its parent's client rectangle, in which case it will be clipped during display. Sibling viewers may not overlap.

Viewer Classes

The Viewers package allows the programmer to create a *class* of viewer, all instances of which will share code for common operations. A viewer class is a record that defines a set of operations and parameters common to all viewers of a particular user defined type. For example, the Tioga document preparation system creates a viewer class for all of the editable text viewers, so that each instance of a Tioga viewer uses the same code to display, scroll, accept input from the user and so forth.

A viewer class record (as described in ViewerClasses.mesa) contains the following information:

flavor: ViewerClasses.ViewerFlavor ← NIL;

Each class has a unique atom which associates a name with the class record.

init: ViewerClasses.InitProc ← NIL;

During the creation of a new viewer, this procedure is called in order to permit the client to initialise private data structures. The viewer will not appear on the screen until the procedure returns. Class implementors should use this procedure to insert menu entries and add sub-viewers.

paint: ViewerClasses.PaintProc ← NIL;

The Viewers package assumes that the bitmap is write-only and that the client can reconstruct its screen image on demand. Whenever the information on the screen for a particular viewer becomes invalid, the PaintProc will be called by the system to restore the correct display. The PaintProc is passed a graphics context clipped to the bounds of the viewer, a whatChanged parameter which the client may use to encode private information about what to update, and a clear boolean that indicates whether the viewer has been whitened before the paint call. There is a complete section on painting elsewhere in this document.

destroy: ViewerClasses.DestroyProc ← NIL;

The Cedar garbage collector normally deallocates storage associated with a viewer when it is destroyed, but sometimes a client finds it convenient or necessary to free resources or note the destruction of a viewer instance explicitly. This procedure is called just before the viewer is removed from the viewer tree.

notify: ViewerClasses.NotifyProc ← NIL;

Mouse and keyboard input from the user is communicated to a viewer as a list of results described in a tip table (see the tipTable field below). This procedure will be called whenever an input event for a particular viewer is received.

tipTable: TIPUser.TIPTable ← NIL;

A TIP Table is a list of productions, mapping mouse and keyboard operations into a list of results meaningful to the client. Please see the section on Mouse and Keyboard input for more information.

modify: ViewerClasses.ModifyProc ← NIL;

Keyboard input depends on a viewer owning the *input focus* described later. This procedure is called whenever a viewer loses or acquires the input focus in order to provide feedback, such as a blinking caret.

set: ViewerClasses.SetProc ← NIL;

Some viewer classes choose to implement this interface as a way of setting the contents or state of a viewer. The *data* and *op* arguments may be interpreted any way the client chooses and the *finalise* parameter determines whether the new information should be reflected on the display. Tioga, Buttons, and Labels are three viewer classes which implement this interface to set the text contents when passed a ROPE or REF TEXT.

get: ViewerClasses.GetProc ← NIL;

A viewer class may implement this interface as a means of allowing a client to query the contents or state of a viewer. Like the SetProc, the *op* parameter may be interpreted by the class implementor as they see fit.

scroll: ViewerClasses.ScrollProc ← NIL;

The ScrollProc is called with an operation parameter, corresponding to scrolling up, scrolling down, thumbing to a particular place in the data structure, or querying the client as to its current scroll position. For the scroll up and scroll down operations, the amount passed is the user's request, measured in points. It is up to the client to interpret the amount with respect to their internal data structures, adjust their display data structures and request repaint. For the thumb operation, the amount passed is the percentage to scroll into the viewer; i.e. 47 would mean to start the top of the display 47% into the data displayed, as interpreted by the client. The query operation requests that the client return two numbers for feedback to the user, corresponding to the percentage of the data displayed at the top and bottom of the viewer; i.e. if the entire data structure were visible, the client would return 0 and 100. If the client cannot reasonably compute the query percentages or chooses not to, then it may simply return from the ScrollProc when this operation is encountered.

menu: Menus.Menu ← NIL;

A viewer may optionally display a menu of commands underneath the caption. See the section on Menus for more information.

icon: Icons.IconFlavor ← document;

When a viewer is iconic, the client PaintProc is intercepted and is instead displayed as a small picture, determined by the IconFlavor set here. The *Icons* interface has an enumerated set available, as well as an interface to create new icons. The client may also paint their own icon using the CedarGraphics facilities by setting the iconFlavor to *private*, which will cause the painting of iconic viewers to no longer be intercepted. If the iconFlavor is private, then it is up to the client to notice that they are iconic and paint appropriately.

cursor: Cursors.CursorType ← textPointer;

Viewers will display a cursor when moved over a viewer, determined by the value here. The Cursors interface has an enumerated set available, as well as an interface to create new cursors.

clipChildren: BOOL ← FALSE;

The normal paint order for viewers with embedded children is the parent first and then the children. There is normally no protection from the parent painting over a child unless this bit is set to true, in which case all child viewers will be clipped (excluded) from the context supplied to the parent. This field is not defaulted to true, since it makes normal CedarGraphics operations computationally expensive for the parent.

The following viewer class fields are either rarely used or not currently implemented. They are documented here for completeness:

save: ViewerClasses.SaveProc ← NIL;

The SaveProc is called when the client is expected to write the private data structure to a disk file. This is currently only used by Tioga.

copy: ViewerClasses.CopyProc ← NIL;

The data associated with a particular viewer is private to the class implementation and hence the Viewers package doesn't know how to replicate the structure. This procedure passes an old viewer and a new viewer and requests that the data for the new viewer be made the same as the old.

caption: ViewerClasses.CaptionProc ← NIL;

The caption normally displays the name and status of the viewer. If this is not acceptable to a particular client, then they may implement this procedure and paint the caption themselves. The graphics context passed is clipped to the caption area.

coordSys: ViewerClasses.CoordSys ← bottom;

The coordinate system of the graphics context passed to the viewer PaintProc will normally have its origin at the lower left corner of the viewer, with x and y increasing to the right and upward. If the coordSys is set to *top*, then the origin will be set to the top left corner with y increasing downward.

bltContents: BOOLEAN ← FALSE;

This is currently unused.

paintRectangles: BOOLEAN ← FALSE;

This is currently unused.

It is important to note that the implementor of a viewer class can default any of the fields if they choose not to provide that particular functionality; the system will *do the right thing* with requests on viewers that do not implement a particular operation.

The programmer creates a new viewer class by allocating and initialising a ViewerClasses.ViewerClassRec and then calling ViewerOps.RegisterViewerClass, passing the class record and a unique atom which names the new class. This operation creates a binding between the class operations and any newly created viewer instances (see the Viewer Instances section below). Subclassing (as in SmallTalk) is not directly supported by the system; the programmer must explicitly copy and modify an existing viewer class.

A simple example of creating and registering a viewer class can be found on [Indigo]<Cedar>Viewers>LabelsImpl.mesa.

Viewer Instances

Once a Viewer Class is defined, instances may be created which will inherit information from the class. Many of the predefined viewer classes (listed in the next section) provide instance creation operations, otherwise a client may call ViewerOps.CreateViewer, passing the class and initial data for the instance.

A ViewerClasses.ViewerRec defines the data associated with each viewer instance. During viewer creation, the client may initialise some of the fields, the Viewers package will initialise others, with the rest defaulting. The remainder of this section describes the data structure in some detail, and is of particular interest to a client implementing a viewer class or creating a tool comprised of many embedded viewers.

Each viewer instance maintains a pointer back to its class data record:

class: ViewerClass ← NIL

The class field is initialised by Viewers during instance creation and remains constant for the lifetime of the viewer.

Each viewer contains two rectangles, describing the outer boundary of the viewer and a properly contained inner area available for displaying client information:

wx, wy, ww, wh: INTEGER ← 0,

cx, cy, cw, ch: INTEGER ← 0

The wx and wy fields define the corner of the viewer with respect to its parent's *client* origin. In the case of a top level viewer, wx and wy will be with respect to the bottom left corner of the display screen. The ww and wh fields describe the width and height of the viewer, measured in screen pixels. The cx, cy, cw, and ch fields define an inner rectangle and represent the clipping boundary for display of the client information. cx and cy are measured from the parent's client origin, just as the wx and wy fields. The client should initialise the wx, wy, ww, and wh fields during viewer creation if and only if the viewer will be embedded in a parent viewer, since the column constraint algorithms will recompute these values. Since viewer position and other information is cached by Viewers, clients should never directly set any of the viewer rectangle values, but instead should use routines in the ViewerOps interface to change the size or position of a viewer.

Other viewer fields of interest to clients include:

data: REF ANY ← NIL

> The implementor of a viewer class may associate instance data with a viewer in this field. Note that only the class implementor may access the data field; other clients should use the property list associated with each viewer.

name: Rope.ROPE ← NIL

> Each viewer has a text name associate with it, which in the case of top level viewers, is displayed in the caption. The client may modify the viewer name, but must be sure to repaint the caption if applicable (via ViewerOps.PaintViewer[viewer: viewer, hint=caption]).

border: BOOL ← TRUE

> An embedded viewer is displayed with a one-bit wide black border if the border field is set. Top level viewers are always displayed with a border.

tipTable: TIPUser.TIPTable ← NIL

> The TIPTable defines how each viewer interprets mouse and keyboard input and is copied from the class record when a viewer is created (there is a later section describing the mouse and keyboard input mechanism in more detail). In some cases, the client may wish to change the behaviour of a particular viewer and may do so by modifying this field. A fine point: if a viewer owns the input focus, then its TIPTable is cached by the notifier, which presently must be reset by releasing and then recapturing the input focus.

file: Rope.ROPE ← NIL

> Some viewers chose to associate a backing file with their data structures. While this logically belongs with the implementors data, it appears in the viewer instance data as a convenience to clients. It may be removed in the future.

menu: Menus.Menu ← NIL

> A top level viewer may optionally have a menu of commands permanently displayed below the caption. Ideally, the client should pass a menu when a viewer is created. To add a menu to an existing viewer, the client must use ViewerOps.SetMenu to reset internal rectangle caches and to repaint the new menu on the display.

icon: Icons.IconFlavor ← tool

> When a top level viewer is made iconic, it no longer displays the client data, but instead appears as a small symbol, easily recognisable by the user. The instance's icon field is normally initialised from the viewer's class, but may be overridden by the client during viewer creation. The *Icons* interface provides routines to access existing Viewers icons (Viewers includes symbolic pictures for a document, tool, typescript, and a file drawer) or to create new icons from either a procedure or a set of bitmaps stored in a file. The client may specify whether an icon is to be labelled by Viewers with the viewer name. If the instance's icon field is set to *private*, then the client PaintProc will be called even though the viewer is iconic, which is how the clock icon displays the current time.

column: ViewerClasses.Column ← left

> When a viewer is created, the client may specify in which column a viewer will be displayed. Clients should only change this field in an existing top level viewer by calling ViewerOps.ChangeColumn.

scrollable: BOOL ← TRUE

Viewers will allocate a scrollbar area on the left hand edge of a viewer as well as provide appropriate user feedback if this bit is specified during viewer creation. The actual scrolling operation are performed by calling the viewer's class ScrollProc.

iconic: BOOL ← TRUE

Clients may set the *iconic* bit during the creation of a top level viewer in order to initially display the viewer as iconic at the bottom of the display or open in its column. A client may not change the iconic bit for an existing viewer, except by calling ViewerOps.OpenIcon or ViewerOps.CloseViewer.

newVersion: BOOL ← FALSE

A client may test to see if edits have been made to a viewer by testing the *newVersion* bit. Viewer implementors should use ViewerOps.SetNewVersion to set the *newVersion* bit and display the "[New Version]" message in the viewer caption.

newFile: BOOL ← FALSE

A client may test to see if there is currently no existing backing file for a viewer by testing the *newFile* bit. Viewer implementors should use ViewerOps.SetNewFile to set the *newFile* bit and display the "[New File]" message in the viewer caption.

offDeskTop: BOOLEAN ← FALSE

A client may test to see if the viewer is currently accessible to the user. ViewerOps.ChangeColumn[viewer, {left, right, color}] will move it back onto the screen.

openHeight: INTEGER ← 0

A client may set the value of openHeight to be the desired vertical space taken up by a viewer when open in a column. The window manager will attempt to honor the request, but may give more or less space than requested.

parent: ViewerClasses.Viewer ← NIL

Each viewer that is nested within another viewer keeps track of its parent through this field. This field will be nil for top-level viewers.

sibling: ViewerClasses.Viewer ← NIL

Viewers of equal depth within a tree are linked via the sibling field.

child: ViewerClasses.Viewer ← NIL

A viewer may have children nested within it.

props: Atom.PropList ← NIL

Clients may associate arbitrary data and properties with a viewer by adding them to the property list. The AddProp and FetchProp operations in ViewerOps are the recommended way of accessing the viewer property list.

A number of fields in the viewer record are private to Viewers or are not particularly of interest to most clients. They are included here for completeness:

lock: ViewerClasses.Lock ← [NIL, 0]

The viewer data structure may be locked with non-exclusive *read* or exclusive *write* semantics, making use of the lock data structure within each viewer instance. The client should never access this field directly, but should use routines provided in the ViewerLocks interface. A more detailed treatment of locking may be found elsewhere in this document.

visible: BOOL ← TRUE

The Viewers package maintains a private visibility hint to speed some repainting operations. Clients should not rely on the value of this field.

offDeskTop: BOOL ← FALSE

Some viewers may be on other than the current desktop, in which case the offDeskTop bit will be set.

destroyed: BOOL ← FALSE

A viewer is destroyed by calling ViewerOps.DestroyViewer, at which time it is removed from the viewer tree and this field is set to false.

saveInProgress: BOOL ← FALSE

Some clients need to know when a *Save* operation is currently pending for a viewer, and may examine the state of the saveInProgress field. Note that the newVersion bit remains set until completion of a *Save*.

init: BOOL ← FALSE

Viewers maintains a private hint indicating that a new viewer has been properly initialised. It has no other uses.

inhibitDestroy: BOOL ← FALSE

The inhibitDestroy bit prevents the user from invoking the *Destroy* operation in a top-level viewer menu. This field will be removed in future releases, so clients should not depend on it.

guardDestroy: BOOL ← FALSE

The *Destroy* menu command for top-level viewers is normally guarded iff there are new edits. If the client wishes the *Destroy* menu command to always be guarded, then they may set this bit.

link: ViewerClasses.Viewer ← NIL

Linked viewers are connected via a ring data structure. Currently, linked viewers are not completely supported by the system, but this field is included for clients to implement linking semantics, as in Tioga.

position: INTEGER ← 0

The position field is a private value used by Viewers for icon positioning and adjusting of viewer size when open in a column.

Predefined Viewer Classes

Listed below is a set of viewer classes for client use with implementations provided in the Cedar boot file.

Buttons

Buttons are a class of viewers that display a text label and call a procedure when the user clicks the mouse over them. Buttons are usually created within a container (see below) as part of a tool, either directly throught the Buttons interface or as part of a higher level object created with the ChoiceButtons interface. Parameters to the client procedure include the button viewer, the mouse button that invoked the button, and the state of the shift and control keys at the time of invocation.

Containers

A container is a viewer that permits other viewers to be recursively embedded and scrolled, and is created using the Containers interface. Routines are also provided to constrain embedded viewers to be clipped to the edges of the parent container. Unlike most viewer classes, the coordinate system inside a container has its origin at the upper left corner, with the y coordinate increasing in the downward direction. A container initially has no menu, but many tool implementors find it convenient to include a menu of tool-related commands when cr ating a container.

Labels

A label is a simple kind of viewer that displays a text message in a single font. The label may not be selected or edited, but has the advantage over Tioga text viewers (described below) of having very low data structure overhead.

Rules

A rule viewer displays as a rectangle of a single color on the display, and is created using the Rules interface. They are useful for creating line graphics when designing the layout of tools.

Text

Text viewers are implemented by the Tioga Document Preparation system described in detail on [Indigo]<Cedar>Documentation>TiogaDoc.tioga. The ViewerTools interface provides a set of operations that allow a client program to create text viewers as well as to fetch and store their contents. There are a great many additional operations supported by the Tioga editor, exported by the TiogaOps interface, which is described in the Tioga documentation.

Implementation Guidelines

The procedures and variables in a viewer were designed to support a particular style of implementation for new classes. Many of the procedures and variables were added specifically to help solve some general problem of concurrency or user interaction. Implementors are not required to use these procedures, but they should only depart from them when they have good reasons.

The best way to write a user application that uses Viewers is to first write a applications package accessible to client programs and then make a thin veneer over it for users. It is tempting to tailor your implementation to the user interface, but this temptation should be resisted. No matter how user-oriented your program is, some user some day will want to write a program that uses your application directly. It is best that you prepare for that eventuality now.

There are four different ways that a particular function in your program could be invoked: notification of an user event through the NotifyProc, a call on a pre-defined function in the Viewers class (such as set, get, and save), invokation through a button or menu item on the viewer, and a client call on an interface you export. If you use more than one of these paths to invoke a function you should be absolutely certain that they all have exactly the same semantics. The best way to do this is to have them all call the same procedure. Thus the NotifyProc that handles special user actions should do no more than gather parameters and dispatch to procedures that are defined in an interface exported by your program. The same should be true of procedures that are invoked with buttons and menus.

The functions 'create' and 'destroy' are special functions in the Viewers world. All of the creation and destruction that is necessary fc a part'cular Viewers class should happen in the class's InitProc and DestroyProc. In particular it is very important that all of the menu construction, sub-viewer creation and private data initialization happen in the InitProc. This allows the Viewers package to create and destroy instances of a class without having to know about interfaces exported by the class's implementation. This is important since opening a desktop sometimes requires the re-creation of a viewer that the user had destroyed. All that the Viewers package can do is call ViewerOps.CreateViewer[viewerFlavor, info: [name: viewerName]] and hope that this is sufficient to create and initialize the viewer. Application tools should create their own viewer's class just so they can have their own InitProc, even if there is never more than one instance of the tool. Similarly, ViewerOps.DestroyViewer[viewer] should be all the Viewers package needs to make sure that the viewer has cleaned up all of its internal data structures. (Cleaning up internal data structures includes breaking circular links so the garbage collector will reclaim the storage.)

Mouse and Keyboard Input

<not yet written>

Locking

The ViewerLocks interface defines an interlocking mechanism to arbitrate access to the Viewer window tree, viewer columns, as well as individual viewers. A particular viewer may be locked for reading or for writing. A read request implies that the calling code would like to examine values in the ViewerClasses.ViewerRec and insure that these values are invariant for the duration that the read lock is held. Any number of clients can simultaneously hold read locks on a single viewer. A write lock is an exclusive access to a viewer for the implied operation of modifying one or more of the fields in the ViewerClasses.ViewerRec.

Locks are process re-entrant, meaning that a single process may request a lock multiple times without deadlock; i.e. if a process currently has write access to a viewer, it may in addition request read or write access (this obviously doesn't grant any new permissions, but is useful when a client has already locked a viewer and wishes to call code that may attempt a lock). Due to the ordering of the read and write resources, a process holding a read lock may not request a write lock on the same viewer without first releasing the read lock.

The viewer tree as well as the particular columns are another set of resources that may be locked. Locking a column is equivalent to (but not implemented as) locking all of the viewers in that column. Locking the viewer tree is equivalent to locking all of the columns for write in addition to the root pointer for the tree. The viewer tree must be locked by the client any time a top level viewer changes size or position in the tree that crosses column boundaries. Since all painting operations are protected by a read lock on the updated viewer, the viewer tree lock will not be granted until painting activity has subsided.

The hierarchy of locks in the viewers window package, from highest to lowest is: the viewer tree, a write lock on a column, a write lock on a particular viewer, a read lock on a column, and a read lock on a particular viewer. A client may only request a new lock if is at the same or lower logical level. The documentation in the viewer interfaces has been (will be) updated to show which locks the operation requires, so that the client can avoid deadlocks.

A fine point: if a client program needs to lock multiple viewers, it must acquire the lock for each viewer in the proper order. Routines are provided below that correctly sort the parameter list for up to three viewers; otherwise the client must order the lock requests so as to lock the viewers in ref order (treating the viewer ref as a long cardinal and locking the highest first). Columns must be locked in the order {static, left, right, color}.

Users are advised to make use of the call-back operations provided when they need to perform locked operations on viewers, and not try and use the locking primitives directly. The suggested usage is to create a local procedure containing the critical code, which is passed to the routines below. The local procedure will then have full access to all of the variables in the outer procedure.

Painting

<not yet written>

Icons

<not yet written>

Menus

<not yet written>

Pragmatics

<not yet written>

Interface Summary

Buttons

Buttons are a class of viewers that display a text label and call a procedure when the user clicks the mouse over them. This interface provides routines to create and destroy buttons, as well as the ability to change the visual appearance of the text label.

Carets

Implementors of text editors (and maybe some other types of editors) provide feedback to the user where text insertion will go by means of a blinking caret on the screen. The Carets interface allows a viewer class implementor to display blinking carets without having to worry about issues of timing, clipping, etc.

ChoiceButtons

When building tools and other applications using Viewers, clients often require user interface abstractions for enumerated types, string parameters the user can edit, and boolean values. ChoiceButtons is a high level interface based on Buttons and ViewerTools that provides these abstractions as primitives.

Containers

A client often wishes to build a tool that consists of several viewers associated with each other. Containers are a very simple viewer class that permit other viewers to be recursively embedded and scrolled. Routines are provided to constrain embedded viewers to be clipped to the edges of the parent container.

Cursors

The Cursors interface exports a set of bitmaps suitable for display in the hardware cursor as well as primitives enabling a client to create new cursor bitmaps. The cursor bitmaps have a logical *hot spot* associated with them that defines the single point in the cursor where the user is pointing; e.g. for a bullseye cursor the hot spot is in the center whereas the text pointer cursor has its hot spot at the upper left.

DeskTops

A desktop is a configuration of viewers as they are layed out on a screen. DeskTops.Create["name"] will create a new desktop named "name" if none exists. If one of that name exists, it will be moved onto the screen. DeskTops.FlyTo[desktop] will cause the contents of the screen to be saved in the current desktop, to be replaced by the viewers and icons in the new desktop.

EndOps

A boot file client may register a procedure which will be called synchronously after the Cedar boot sequence is completed, but before user input is enabled and any automatic checkpoint made.

HourGlass

This interface allows a client to display an hourglass cursor with flowing sand and is used during Cedar booting.

IconManager

This is a private interface that governs the behaviour of icons in response to selection and user type in.

Icons

The Icon interface exports a set of bitmap pictures used to display iconic viewers. A client may create new icons from either a bitmap or a procedure that makes use of Cedar Graphics to display the image. A client may specify a rectangle within an icon that will be labelled with the name of the particular iconic viewer by Viewers.

ImplErrors

A rudimentary debugger and signal catcher is exported by ImplErrors that gives the user the choice of rejecting, proceeding, or saving edits and rolling back when an error occurs. General use is not recommended.

InputFocus

InputFocus controls the destination of mouse and keyboard actions as well as the interpretation of these actions through TIP tables. A viewer class implementor would use this interface to acquire type-in or to inquire which viewer type-in was currently directed.

Labels

Labels implements a class of viewers that display a simple text message in a single font. The label may not be selected or edited, but has the advantage over Tioga text viewers of very low data structure overhead.

MBQueue

Some applications require a way to serialise user input from buttons, menus, and type-in without tying up the notifier process. MBQueue permits the application to enqueue and dequeue operations.

Menus

Menus are a horizontal sequence of text labels, each with an associated procedure called when the user clicks at the menu item. This interface provides low level routines for creating and modifying menus, but not for associating them with a particular viewer or redisplaying a viewer after its menu has changed.

MenusPrivate

A private interface enabling the window manager to invoke a menu item.

MessageWindow

The message window is a one-line display area at the top of the black and white display. Clients may use this interface to display prompts or other information to the user.

Rules

Rules are a simple viewer class that display as a rectangle of a single color on the display. They are useful for creating line graphics when designing the layout of tools.

TypeScript

This is a low-level interface that captures sequential type-in from the user. Clients are advised to use the IO streams interface instead.

VBootOps

A client may boot a particular volume or file with this interface.

VFonts

VFonts maps a font name into the actual representation, with options to synthesize bold and italic as well as to substitute for fonts that are not available.

ViewerAdjust

A private interface used to control user adjustment of viewer or column size and position.

ViewerBLT

A viewer class implementor may move displayed bits from one part of a viewer to another, with clipping to the viewer boundaries and almost the performance of the hardware BitBlt instruction. [This interface is not currently supported and should not be used].

ViewerEvents

ViewerEvents permits client procedures to be registered that will be called when certain events take place with respect to a viewer, such as save, new edits, open or close, move to a new column, and changes in the input focus.

ViewerMenus

A private interface containing most of the menu items found in the viewer caption menu.

ViewerClasses

ViewerClasses is an important interface that defines the viewer class and instance data structures, as well as many of the basic Viewers primitive types.

ViewerLocks

This interface provides read and write locking of viewers, columns and the entire viewer tree. [This interface is not currently supported and should not be used].

ViewerOps

ViewerOps is an important interface defining standard operations on viewers. Routines that manipulate viewer size and position, register new viewer classes, create new viewers, as well as most paint and input notifications live here.

ViewerSpecs

Screen layout constants are defined in the ViewerSpecs interface.

ViewersStallNotifier

Bugbane uses this private interface to notify Viewers when a process has been suspended, pending handling of a breakpoint of an uncaught signal.

ViewerTools

ViewerTools provides low level access to the contents of Tioga text viewers and permits clients to control selection and editing within them. See the ChoiceButtons interface for creating text prompt fields and other higher level user interface abstractions.

ViewersSnapshot

Client invoked Checkpoint and Rollback routines live here.

VirtualDesktops

This interface is being replaced by "DeskTops".

WindowManager

WindowManager contains a miscellaneous collection of routines, the most important being the ability to enable an additional column of viewers on a color display.

WindowManagerPrivate

This is a private interface containing the current viewer tree root and some input notification routines.

Cedar Catalog

Release 4.2

Release as [Indigo]<Cedar>Documentation>Catalog.tioga

Came from [Indigo]<CedarDocs>Manual>Catalog.tioga

Last edited Russ Atkinson on June 8, 1983

Abstract This catalogue is a list of "interesting" Cedar packages and tools. Each component is described by a *maintainer-supplied* entry of the form

Name P T Maintainer Access Documentation

Comments about why the component is interesting.

In general, all the source, bcds and other files relevant to a package named X can be gotten through the file [Indigo]<Cedar>Top>X.df. Using the "/p" option of Bringover should obtain just what you need to run the program.

The default directory for documentation is [Indigo]<Cedar>Documentation>.

[If you are reading this document on-line, I suggest that you use the Tioga Levels menus to initially browse the top levels.]

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

DRAFT - For Internal Xerox Use Only - DRAFT

Cedar Catalog: Contents

External Structures

CIFS
CIFSCommands
FlushUnneededTempFiles

IO

Streams and Ropes

IO
RopeFile
RopeIO
RopeReader
ShowTime

Scanning and Parsing

CedarScanner
UECP

Graphics

Graphics
PlotPackage
TJaMGraphics

Viewers

Viewers
ViewRec
VTables

Tioga and Press

BravoToTioga
Print
SirPress
Tioga
TiogaMenuOps
TiogaOps
TSetter

Communication

Mail

GrapevineUser
Maintain
Walnut
WalnutSend

Version management

DFFiles
Modeller

Remote procedure call

Lupine

Other communication components

Chat
FileTool
Pup
Pupwatch
STPServer

Command interpretation

Commander
InterpreterTool
UserExec

Miscellaneous

Clock
CoFork
Forms
Init
MType
PerfStats
Poplar
Random
Reminder
SampleTool
Spell
Unique
Waterlily

Wizards' Appendix

AlpineUser
AMEvents
BasicHeads
BCD
BootClientFile
BTree
BTrees
CedarDB
CedarRoot

ClientFileTool
ColorPackage
Communication
CompatibilityPackage
Cypress
D0Microcode
DLionMicrocode
DoradoMicrocode
Germ
IFSFile
Inscript
Lister
Loader
MakeBoot
Othello
Packager
PGS
PilotKernel
Pupwatch
RPCRuntime
Sequin
SpecialTerminal
Squirrel
STP
TerminalMultiplex
TTYIO
VersionMap
VersionMapBuilder
WorldVM

Alphabetical Summary

Introduction

This catalogue is a list of "interesting" Cedar packages and tools. Each component is described by a *maintainer-supplied* entry of the form

Name P T Maintainer Access Documentation

Comments about why the component is interesting.

Entries **with boldface names** are recommended for study and use as prototypes.

Entries marked with *P* are recommended as components for general use.

Entries marked with *T* are recommended as tools; i.e., things you use interactively (via UserExec commands or their own viewers) rather than call from programs.

The **Access** item tells you the normal way of getting the files associated with the component: **Boot**: in the boot file; **Client**: brought over by CedarClient.df; **ClientFat**: brought over by CedarClientFat.df; **DF**: brought over using a DF file—with the same name unless another name is given explicitly.

In general, all the source, bcds and other files relevant to a package named X can be gotten through the file [Indigo]<Cedar>Top>X.df. If that is not the case, the df file is mentioned in the Documentation column. Using the "/p" option of Bringover should obtain just what you need to run the program. A common convention is that the "principal" interface of a component has the same name as the component and its DF file.

Introduction in the documentation column means that information about this component can be found in the *Introduction to Cedar* section of the *Cedar Manual*; **In source**, that the documentation and source code files have been combined; **Internal**, that the comments in the source program(s) may be helpful. In most other cases a file name is given.

For brevity, if a file name is given, the default directory for documentation is [Indigo]<Cedar>Documentation>. Press files are not mentioned if there is a Tioga documentation file. The host is not mentioned unless it is *not* [Indigo].

Complaints are the precursors of improvement. Please send suggestions for improving the form or content of this Catalog to the current Cedar Release Master, or to Jim Horning if you don't know who that is. Send comments about any particular entry to its maintainer, with copies to Jim <Horning> and Mary-Claire <vanLeunen>.

Cedar Abstract Machine

Language

Binder T *Satterthwaite Client* *Mesa 5.0 Manual, Mesa User's Handbook*

The Mesa Binder.

Compiler T *Satterthwaite Client* *Cedar Language Reference Manual, Summary*

The Cedar language compiler.

A few paragraphs about running the compiler, file name and switch conventions, etc., belongs somewhere (Getting Started? A tools document similar to SDD's User's Handbook?). I will try to assemble this from the Mesa User's Handbook and subsequent updates - EHS

GL P *Satterthwaite Lister.df* *Contact maintainer*

The global frame lister GL (in Lister.df) is certainly useful enough to the average person that it should be described as an entity of its own. Maybe the other listers too. - MB

The Lister, as documented in the Mesa User's Handbook, will go away as soon as the conversion effort for a new release is non-trivial. (It uses several packages from SDD no longer supported by them or understood by us.) Mark is probably right that GL is a generally useful tool. Its use was documented in one of the release messages. The other lister derivatives (CL, SL, BL) were intended for wizards only, but a lot of people seem to like reading code listings. - EHS

IncludeChecker T *Rovner* *DF* *Mesa User's Handbook*

A tool for listing compilation dependencies in Mesa programs.

MCross T *Rovner* *DF* *MCross.doc*

Creates a textual cross-reference for the identifiers that appear in a specified collection of Cedar source files. It is happy to accept other source files (like .cm and .config) too.

Debugging

BugBane T *Atkinson* *Boot* *BugBane.tioga*

Provides debugging features (including remote debugging) for Cedar. Supports expression interpretation, breakpoints, uncaught signal handling, and address fault handling. Exports numerous useful facilities.

Runtime

AMModel P *Rovner* *Boot* *In source*

AMModel.df includes the components of the Cedar "abstract machine" that deal with loaded programs, frames, their object modules and source modules, and interdependencies. Its major public interfaces are AMModel.mesa and RTMiniModel.mesa.

AMTypes P *Rovner* *Boot* *RTTypes.tioga (NOT up to date yet)*

AMTypes.df includes the part of the Cedar "abstract machine" that deals with types, values and variables. Its major public interfaces are AMBridge.mesa and AMTypes.mesa. See RTTypes.tioga for more details.

SafeStorage P   *Rovner*   *Boot*   *SafeStoragePrimer.tioga, SafeStorage.tioga*

SafeStorage.df includes the Cedar allocators, collector, and basic runtime type system (i.e. support for Cedar language features related to REF ANY, ATOMS, and ROPE and REF TEXT literals). Its major public interfaces are SafeStorage.mesa and UnsafeStorage.mesa. SafeStoragePrimer.tioga is a short document that every Cedar programmer should understand.

System

The "Mesa machine" section needs an accompanying document from Roy: brief documentation for the portions of Environment, Inline, System, Runtime, and Process that most programmers use (enough so that a new programmer can know when to look at these interfaces). Then a pointer to full Pilot documentation of these interfaces. - MB

CedarSnapshot   *Levin*   *Boot*   *In source*

This package provides the machinery to save and restore the state of virtual memory. Clients that need to revise their state at the time of a Checkpoint or a Rollback can register procedures to be called when these events occur. Details are documented in the interface, CedarSnapshot.mesa.

DateAndTime   *Levin*   *Boot*   *In source*

This package parses dates and times in most intelligible formats. It does not do context-dependent parsing (e.g., "next Tuesday," "the first full moon after my grandmother's birthday," etc.), but it tries to handle all variations that show up in message headers (there are quite a few). There are detailed comments in the interface that describe the formats the package claims to handle. The DateAndTime interface is safe. Users who encounter plausible formats that the package will not handle are encouraged to contact the maintainer for relief.

Those requiring an unsafe version (long strings instead of Ropes), can use DateAndTimeUnsafe.mesa, available through this df.

ScanZones P   *Rovner*   *DF*   *In source*

ScanZones is a utility for gathering and printing (on the file named Storage.log) information about the utilization of collectible storage. Use the UserExec to run it, then use the interpreter to invoke it, as follows:

```
← ScanZones.ShowStorage[wordsCutoff: 100, append: FALSE, specifiedZone: NIL]
```

Spy T   *Maxwell*   *DF*   *Spy.doc*

The Spy is the standard performance tool. It allows the programmer to monitor processor usage, page faults, allocations, or user defined resources. SpaceWatch is now a part of Spy.df and there are now low level facilities for tracing executions (see maintainer).

UserCredentials P   *Levin*   *Boot*   *In source*

This package provides centralized management of the credentials of the user at the terminal. It replaces and extends the old NameAndPasswordOps interface in CompatibilityPackage, which is no longer available. UserCredentials is *not* part of CompatibilityPackage.df. For those requiring an unsafe version, UserCredentialsUnsafe.mesa is also available.

UserProfile P Teitelman Boot UserProfileDoc.tioga

The UserProfile package allows the user to alter the behavior of programs that cater to varying tastes. The programs examine the user's profile, which is a file with a user-dependent name (e.g., Fred's profile is a file with the name *Fred.profile*). Viewers, Tioga, TSetter, and UserExec are examples of programs that examine the profile. UserProfile.bcd is an interface that provides interrogation functions for the profile.

Watch T Atkinson Client Here

This Tool runs in background and cheaply monitors various kinds of resource usage (storage allocation, CPU load, page fault rate, free VM pages, largest run of free VM pages, free MDS pages, etc.) Some data are calculated once per minute, but will be calculated immediately if you left-click the "Sample" button. Others are sampled at a rate given after the "Interval" button. Left-clicking the "Interval" or "GC Interval" button doubles the parameter associated with the button; right-clicking halves it.

Data

Types

Atom P Teitelman Boot In source and here

Atom contains procedures for creating atoms, and for storing and retrieving information on property lists. Property lists are lists of name value pairs that are usually associated with an atom, but can also be manipulated directly using procedures in this interface, e.g., the data field in a Viewer is usually a property list so that various clients can associate data with a viewer without getting in each others way.

CedarReals P Stewart Boot MesaFloat60.press

Floating point routines (IEEE standard 32-bit).

List P Teitelman Boot In source and here

List contains various useful procedures for dealing with LIST OF REF ANY, e.g., Append, Member, NthTail, Union, Reverse, Sort, etc. For applications that traffic in LIST OF particular REF, e.g., LIST OF ATOM, it is necessary (but acceptable) to LOOPHOLE the list into a LIST OF REF ANY and then use a procedure in this interface. (For those procedures that return a LIST OF REF ANY value, it may be necessary to LOOPHOLE this value back into the LIST OF particular REF.)

RefTab P Teitelman Boot In source and here

RefTab provides an alternative to property lists for associating information with unique keys in a global fashion using specific structures (hash tables). RefTab includes procedures for creating a new table, fetching a value for a particular key, storing a new key-value pair, deleting a key-value pair, and enumerating the pairs in a given table.

Rigging P Atkinson Boot Rope.tioga

Rigging exports Rope, RopeInline, RefText, Convert, ConvertUnsafe, ShowTime. These routines allow manipulation of ropes (immutable sequences of characters), and simple conversions between ropes and other values.

The RefText interface includes some simple, generally-useful procedures for REF TEXT (a mutable, garbage-collected sequence of characters). As much as possible, the operations parallel the Rope interface's operations on ROPE. The RefText interface also gives access to a pool of preallocated TEXTs, for use by applications that frequently allocate and discard scratch TEXTs.

Internal Structures

MonitoredQueue P *Paxton* *Boot* *Internal*

Provides a queue of REF ANYs. Remove procedure will wait for next item to be added.

OnlineMergeSort *P* *Brown* *DF* *OnlineMergeSort.press, In source*

A very efficient polymorphic list-sorting package. Sorts LISTs of T, where T (and a procedure for comparing two Ts) is defined in a DEFs module that parameterizes the ListSort interface. The package's implementation uses the "online merge sort" algorithm to sort an n-item list in time O(n log n). A user of the package creates a suitable DEFs module to parameterize ListSort, then compiles ListSort and OnlineMergeSortImpl.

PriorityQueue P *Atkinson* *DF* *In source*

A PriorityQueue object is a collection of items where items can be inserted in any order, and removed in best-first order, where "best" is determined by a user-supplied predicate. This package can be used to sort in O(N log N) time.

RedBlackTree *P* *Brown* *DF* *OrderedSymbolTable.press, In source*

A package for maintaining symbol tables with an ordering among keys. The ordering allows the table to perform searches such as "find the smallest item in the table that is larger than this one," as well as exact-match searches. A table stores items of type I with keys of type K, where I and K are defined in a DEFs module that parameterizes the OrderedSymbolTable interface. The package's implementation uses a binary tree representation of 2-3-4 trees, called "red-black" trees; this means that any search, insertion, or deletion from a table of n items takes O(log n) time. A user of the package creates a suitable DEFs module to parameterize OrderedSymbolTable, then compiles OrderedSymbolTable and RedBlackTreeImpl.

RedBlackTreeRef P *Brown* *DF* *OrderedSymbolTableRef.press*

A variant of the above package that stores items of type REF ANY, and takes a item-comparison procedure as a parameter at the time a table is created.

Set *P* *Rovner* *DF* *Set.doc*

This package is intended for dealing with "sets" of REFs, i.e., variable length collections of elements in which there are no duplicates. Elements are REF ANYS

External Structures

CIFS P *Schroeder* *Boot* *CIFSManual.tioga*

The Cedar Interim File System provides basic file storage and directory services to support Cedar. To salvage, boot Cedar with the "1" switch. The new single packet file server enquiry protocol is made available through the "FileLookup" interface.

CIFSCommands T *Schroeder* *DF* *CIFSManual.tioga*

CIFSCommands provides a way to interact with the Cedar Interim File System from the User Exec. Load it by typing "Run FileSystemCommands" to the UserExec.

FlushUnneededTempFiles P Levin *DF* *Here*

This program combs the disk for orphaned temporary files and reclaims the disk space they occupy. These files result from repeated rollbacks and can eventually consume many hundreds of pages of disk space. Run FlushUnneededTempFiles.bcd from the Executive. Because of various possible races, it is best to do this only when nothing else is going on--don't edit files or load windows when this program is running (it only takes a minute or so).

IO

Streams and Ropes

IO P *Teitelman* *Boot* *In source (IO, FileIO)*

The standard stream package for all File and TTY input output; includes an output format interpreter and a collection of scanners for formatted input. Documentation is found in the various interfaces (IO, FileIO). FileIO.mesa contains file creation procedures plus a pretty complete model for the basic IO stream operations.

RopeFile P *Atkinson* *DF* *In source*

RopeFile is a preliminary facility for creating ropes that logically appear to be backed by files, yet are limited in the amount of VM used. RopeFileImpl registers a command, openHuge, that can be used to open very large files, while using a small amount of VM. It is released for client test purposes.

RopeIO P *Paxton* *Boot* *In source*

Provides efficient read or write filing operations for ropes. Files can be specified by name, capability, or stream.

RopeReader P *Paxton* *Boot* *In source*

Provides fast inline access to characters in ropes designed for speed in reading a sequence of characters, either forwards or backwards from packed arrays (such as created by RopeIO) or from text ropes (the packed sequence of characters variant) defaults to Rope.Fetch for other cases.

ShowTime P *Atkinson* *Boot* *Rope.tioga*

This package is useful to format times for printing.

Scanning and Parsing

CedarScanner P Atkinson Boot In source

CedarScanner supports tokenizing of CedarMesa according to the currently accepted CedarMesa syntax. It does not support actual parsing of CedarMesa syntax.

UECP P Stewart DF In source

UserExec command line parser. UECP.Parse is the only procedure. It parses a UserExec command line rope and returns a SEQUENCE of tokens and switches.

Graphics

Graphics P Wyatt Boot In source

The basic low-level diplay package; provides device independent graphics and text operations.

***PlotPackage** P Rovner DF In source*

A Viewers-based facility for dealing with vectors of REALS, clock-driven interval timers, streams of "events", and simple 2-d graphs of (x,y) coordinate sequences. It features automatic axis labelling and repainting. It provides a good prototype showing how fairly impressive graphic applications can built quickly and easily using existing packages.

TJaMGraphics P Stone DF Internal

This package creates a viewer and registers a collection of JaM commands that call CedarGraphics. This provides an interpreter interface for making pictures on the screen.

Viewers

Viewers P T McGregor Boot Introduction.tioga, Viewers.tioga

The Cedar window package.

ViewRec P T Spreitzer DF ViewRec.Doc

ViewRec is intended for quick and easy user interface construction. Given a record, or the name of a DEFINITIONS module, it will construct a Viewer on the components of that record that are simple enough (numbers, ROPEs, enumerations, subranges of simple enough, records of simple enough stuff, procedures that take simple enough stuff, or explicitly bound by the client). The procedures may be invoked, and the data may be edited.

VTables P Atkinson DF In source

VTables is a package that provides a Viewer class for tabular organization of viewers. It supports rectangular organization with optional borders, automatic sizing, and addition, deletion, and swapping of rows and columns of viewers.

Tioga and Press

BravoToTioga T   Paxton   DF   Here

To convert the Bravo format file X to Y, the Tioga version, RunAndCall BravoToTioga X Y. If X has no extension, ".bravo" is assumed. If Y is omitted and X is U.V then U.doc is used for Y.

This program preserves all the font information and roughly approximates the indenting of your Bravo document. It does not attempt to produce a document that looks identical to the Bravo one. It does not preserve leading, justification, or centering; it's pretty easy for you to do these over again if Tioga's defaults don't suit you. It produces documents in the style Cedar.style.

Tioga does not currently properly support tab stops, overstriking, or underlining. However, BravoToTioga does preserve underlining information by giving characters the "z" looks which Cedar.Style currently makes italic. There is also an interface, BToT, that supplies the procedure BravoToTioga.

Print T   Plass   Client   Print.doc

The standard print command for unformatted text. Print accepts a number of parameters via the user profile; see UserProfile.doc (accessed through UserExec.df).

SirPress P   Plass   Client   In source

SirPress is a module for writing Press files.

Tioga T   Paxton   Boot   TiogaDoc.tioga

The Cedar text editor.

TiogaMenuOps P   Paxton   Boot   Internal

Exports Tioga menu commands for use by client programs.

TiogaOps P   Paxton   Boot   Internal

Exports Tioga editing commands for use by client programs.

TSetter T   Plass   Client   TSetterDoc.tioga

The TSetter tool provides a convenient way of driving the Tioga typesetter or creating press files to containing the current contents of the screen. Start it with the UserExec, and use the middle mouse button on the menu items to find out what they do. A "TSetter" command is registered with the user exec. The first parameter is the server, and the rest of the line is a list of files to typeset.

Communication

Mail

GrapevineUser P   Birrell   Boot   <Grapevine>Docs>Interface.press, Grapevine.press

GrapevineUser is the package for interfacing to the Grapevine mail and registration services.

The Cedar interfaces to Grapevine are GVBasics.mesa, GVNames.mesa, GVSend.mesa and GVRetrieve.mesa. GrapevineUser is included in the Cedar boot file. To use it, read Grapevine.press carefully, read much of Interface.press, then read the interfaces.

Maintain T   *Birrell.*   *DF*   *Introduction, <Grapevine>Docs>Interface.press*

Maintain provides a tool for interrogating and updating the Grapevine registration database.

Walnut T   *Willie-Sue*   *Client*   *WalnutDoc.tioga*

This is the mail system for Cedar. Please read the documentation and send a message to WalnutSupport↑ before trying to use it.

WalnutSend T   *Willie-Sue*   *Client*   *WalnutDoc.tioga*

WalnutSend is a tool for sending messages from Cedar. Typing WalnutSend to the exec creates an iconic viewer. Opening the icon does not capture the input focus; that is done by left-clicking in the message composition area. Tioga Placeholders are provided for filling in header fields. WalnutSend does not handle private distribution lists.

Version management

DFFiles T   *Atkinson*   *Client*   *DFFiles.tioga*

DFFiles is a collection of programs (BringOver, VerifyDF, SModel, etc.) to manipulate DF files.

BringOver will take a list of files and their create dates and retrieve those files not on the local disk. If this list of files is considered a version of a system, then BringOver will guarantee that system is on your disk.

SModel (a Simple Modeller) perform the inverse operation of updating a remote server with changes.

Modeller T   *Satterthwaite*   *DF*   *ModelRefMan.press*

The Modeller is a general configuration management tool. The modeller will track changes to programs and perform automatic recompilation. Faster-than-usual turnaround for small program changes is possible by using the module replacement facilities of the Modeller. See Ed Satterthwaite or Eric Schmidt if you want to try using it.

Remote procedure call

Lupine P   *Birrell*   *DF*   *LupineUsersGuide.press*

Lupine is a program that allows you to make inter-machine "Remote Procedure Calls" ("RPC"). This allows you to write what look like ordinary Mesa procedure calls to invoke operations on another machine on the Pup internet. Lupine and its supporting RPCRuntime are intended to make network communication very easy, very fast, and very cheap. Secure communication facilities are included.

Other communication components

Chat T   *Teitelman*   *DF*   *Introduction, "Chat?" to UserExec*

Tool for talking to file servers. A UserExec command line of simply "Chat" will create an iconic Chat viewer. A command line like "Chat maxc" will create an open chat viewer, connect to Maxc, and log in the current Cedar user.

FileTool T *Stewart* *Boot* *Introduction, "FileTool?" to UserExec*

The FileTool includes the functionality of the old RemoteGetTool (you can do form-fill-out Bringovers). Only the []<>> syntax is accepted. There is a boolean button called Exports only which is essentially the /p switch. The Verify boolean button is used as the /a switch. If the Directory field of the FileTool is not empty, it is prepended to the DFFile field. There is a List-Option called Pages, which works on Local-List only. It prints (in parens) the value returned by File.Size. In Pilot there is no single valued relation between a file's length in bytes and its length in pages. If asked to overwrite a file in use by the runtime system, the FileTool removes its directory entry and makes it temporary.

Local delete of a file may raise "Insufficient permissions." Sorry. Use Control-Del to stop DF operations, "Stop!" to halt others.

Pup P *Birrell* *Boot* [Ivy]<Mesa>*Pup.press

This implements the basic PUP communication protocols - up to the level of byte streams. Most public interfaces are in Communication.df, but EFTP is in Pup.df. Higher level protocols are available through the STP package (for file transfer) and Lupine (for remote procedure calls).

Pupwatch T *Birrell* *DF* *<Grapevine>Pupwatch>Pupwatch.press*

Running Pupwatch.bcd from Pupwatch.df creates a viewer that allows you to watch Pup packets in transit on the local Ethernet. Pupwatch will display packets from or to a particular host, which you specify by NLS name or network address. The pupwatch viewer gives you a brief summary of each packet. The "Write Log" button writes a file giving more details about each packet. Pupwatch understands most common Pup packet types. If you run Pupwatch on a Dolphin, it is liable to miss packets because of lack of processing power; on a Dorado it should miss no packets.

STPServer T *Schmidt* *DF* *Here*

Allows some other machine to retrieve from or store onto your machine when running Cedar. Type "Run STPServer" to the UserExec.

Command interpretation

Commander P *Rovner* *Boot* *Commander.mesa*

Commander is a package for the registration of user commands. It also exports the CommandTool, which is a simple command interpreter.

InterpreterTool P *Rovner* *Boot* *Contact maintainer*

InterpreterTool is a simple expression interpreter.

UserExec P T *Teitelman* *DF* *UserExec.tioga*

The user interacts with Cedar in special viewers called Work Areas. Each work area is either an executive, or an interpreter. You can convert a work area from an interpreter to an executive or vice versa with the ChangeAreaMode command. You can create new work areas via the New Menu button. Left-clicking creates a new work area of the same "flavor"; right-clicking creates one of the opposite flavor. In interpreter areas, you are always prompted with *&n* ←. (You can erase the "←" to execute registered commands by using BS, ↑A, or ↑W.) In executive areas, the prompt is simply *&n* (but you can type ← if you want to interpret and expression in an executive area.)

Miscellaneous

Clock P *Atkinson* *DF* *Internal*

Clock is a graphic display of the current time. It uses Graphics and Viewers to present a circular clock face with hands. It is a good example program as well as an attractive timepiece.

CoFork *Sturgis* *DF* *Cofork.doc, .press*

Provides for the convenient start up and shut down of *coroutines* that use Mesa PORTS.

Forms *Horning* *DF* *Internal*

This is the beginning of a collection of Tioga forms useful for the creation of various kinds of documents within Cedar. SampleSheet.tioga contains examples of all the principal Looks and Formats of Cedar.style. Form.memo approximates the old Bravo memo.form.

Init *Taft* *DF* *see Release message*

Othello command files for formatting the disk, creating logical volumes, and installing new versions of Cedar.

MType T *Schmidt* *Client* *Here, DFFiles.df*

MType (Mesa Type) types a set of files (local or remote), or searches them for a specified pattern and types the matching lines. It is a UserExec registered command. The command line

```
mtype -g "Inline" [ivy]<Garp>Foo.mesa Bar.mesa
```

searches the files [ivy]<Garp>Foo.mesa and Bar.mesa for occurrences of the string "Inline", and prints the lines that contain this pattern. A * in a pattern matches any sequence of characters.

PerfStats P *Brown* *DF* *PerfStats.press, In source*

PerfStats is a simple module used for gathering performance information, often in the context of a test program. It implements counters and stopwatch-like timers, and will print summary statistics to an output stream.

Poplar P *Schmidt* *DF* *Poplar.Press, EASPL.Press*

A string-manipulation language. Note that Poplar is one of the SAFE-est Cedar application programs.

Random P *Brown* *DF* *Random.press, In source*

Random is a module that produces a sequence of random INTs; RandomCard generates random CARDINALs, and RandomInt generates random INTEGERs. These generators all use the same algorithm, which produces very good sequences and is quite fast.

Reminder T *Rovner* *DF* *In source*

This is a simple program for managing a personal reminder calendar, in lieu of the more complete facilities of Hickory, which is not yet released.

***SampleTool** T* *McGregor* *DF* *In source*

The SampleTool is an example program for using Viewers, the Cedar window package.

Spell P *Teitelman* *Boot* *Spell.mesa*

A name completion and spelling correction program.

Unique T *Schmidt* *DF* *Unique.doc*

Unique parses various tokens and sorts them, eliminating duplicates. For example, you can give Unique an input filename and it will parse the contents of the file, eliminating tokens that do not look like filenames (e.g., XX.XX), and then will print out a sorted list of filenames with duplicates eliminated. It can also sort by file name extension (e.g., .mesa, .bcd), by numbers, by words (a sequence of letters separated by white space), and by lines. If an input file is not given, Unique will use the current selection as input. Type "Run Unique" to the UserExec. Select {Files, Extensions, ...} as desired, give a filename or select text to be processed, and click the Go button in the top menu.

Waterlily T *Kolling* *DF* *"Waterlily?" to UserExec*

Waterlily compares two (local or remote) source files. Various switches are available to specify the format of the source files and other options, but in most cases the defaults will be sufficient. The long help message via the UserExec describes the switches. Comment nodes are not seen when input files are specified to be Tioga format (the default).

Wizards' Appendix

AlpineUser P *Kolling* *DF* *Contact maintainer*

Support for access to Alpine, the experimental file server.

AMEvents P *Birrell* *Boot* *Contact maintainer*

AMEvents is the part of the Cedar abstract machine concerned with interaction between executing programs and the debugger: uncaught signals, breakpoints, and interpreted procedure calls. Normal access to these facilities is through BugBane. Consult a wizard before calling AMEvents directly.

BasicHeads *Taft* *Boot* *See interfaces, Contact maintainer*

<Cedar>Heads>BasicHeadsDorado.df
<Cedar>Heads>BasicHeadsD0.df
<Cedar>Heads>BasicHeadsDLion.df
<Cedar>Heads>BasicHeadsCommon.df

BCD *Satterthwaite* *DF* *Contact maintainer*

This is a collection of shared low-level interfaces defining the formats of bcd files, symbol tables and the like. Most Cedar clients should use interfaces from the Runtime and RuntimeTypes packages instead.

This component was formerly available as a sub-component of Compiler.df. It is now separately available.

BootClientFile *Levin* *DF* *Contact maintainer*

This is a tool intended for use by wizards. It permits you to boot from CoPilot a file on the Client volume. To use, simply acquire BootClientFile.bcd and type the following to the CoPilot SimpleExec:

```
BootClientFile <bootfilename>
```

If <bootfilename> has no extension, ".boot" is assumed.

BTree P *Taft* *DF* *BTree.tioga*

This package maintains an ordered collection of objects as a BTree. The objects may be of different sizes, and there may be a large number of them (tens or hundreds of thousands). The amount of virtual memory required does not depend on the size of the BTree, and the cost of finding, inserting, and deleting objects increases only very slowly as the BTree gets larger. The package makes very few assumptions about the representation of the objects being stored or about the properties of the storage itself.

BTrees P *Levin* *DF* *Contact maintainer*

Obsolete. New applications should use BTree.

CedarDB P *Cattell* *DF* *Internal and Contact maintainer*

CedarDB provides database facilities for the Cedar environment, and is currently under experimental use by several applications. Documentation is not yet complete.

CedarRoot P *Levin* *Boot* *Contact maintainer*

This package contains two miscellaneous interfaces related to Cedar initialization, CedarInitOps and CedarVersion.

ClientFileTool T *Birrell* *DF* *Internal*

A trivial translation of the Copilot ClientFileTool into Cedar, intended for 4.0.

ColorPackage P *Stone* *DF* *Internal*

This package is the set of programs outside of CedarGraphics and Viewers required to run color viewers. It exports:

ColorMap.mesa for clients who want to change the color map in the color display.

ColorWorld.mesa for Viewers.

Communication P *Birrell* *Boot* *Pilot users manual*

The Pilot communication package implements the basic communication protocols - up to the level of byte streams. There are two flavors of protocol available: "OISCP," described in the Pilot Programmers' Manual, and "PUP." At present, only the PUP protocols are used within CSL. These are all "unsafe" interfaces. See also the STP package for FTP transfers, and Lupine for remote procedure calls.

CompatibilityPackage *Levin* *DF* *Contact maintainer*

All interfaces remain unsafe.

Cypress P *Cattell* *DF* *<CedarDocs>Database>CypressDoc.press*

Access to the Cedar database facilities.

D0Microcode *Fiala* *Contact maintainer*

This component consists of microcode files that D0 users must install on their disk in order to run Cedar.

DLionMicrocode *Sturgis* *Contact maintainer*

This component consists of two microcode files, SAx000Initial.db and FixedMesa.db, that Dandelion users must install on their disk in order to run Cedar.

DoradoMicrocode *Taft* *<DoradoDocs>DoradoBooting.press, Contact maintainer*

This component consists of microcode files that Dorado users must install on their disk in order to run Cedar.

FTP P *Birrell* *DF* *[Ivy]<Mesa>Doc>FTP.press, Contact maintainer*

Germ *Taft* *Pilot User's Handbook*

<Cedar>Top>GermDorado.df
<Cedar>Top>GermD0.df
<Cedar>Top>GermDLion.df

IFSFile *Levin* *DF* *Contact maintainer*

Leaf-level access to IFS. Wizards only.

Inscript *Stone* *Boot* *InscriptImplementation.memo, .press,*
InscriptImplAppendix.memo

The basic keyboard input subsystem.

Lister *Satterthwaite DF* *Lister.txt*

Lists the mnemonic bytecode representation of compiled programs.

The program CL.bcd (also registered as CodeLister) lists individual procedures from a module. The command

```
   cl output ← input
```

where input is a single identifier, produces a code listing for the module input.bcd on output.cl (input.cl by default). The command

cl output ← input.proc
where input and proc are single identifiers, produces a listing of all procedures with name "proc" in the module input.bcd. Output goes to output.bl (proc.bl by default). Also, CL recognizes the switch /h, which augments the code listing with addresses and instructions listed in hexadecimal.

Loader *Rovner* *Boot* *Internal*

The basic bcd loader. It should be used instead of SubSys.Load, SubSys.Run, Runtime.LoadConfig, Runtime.NewConfig, Runtime.RunConfig, and Runtime.ConfigError.

MakeBoot *Levin* *DF* *MakeBoot.press*

This is a program for wizards that have a need to construct boot files. Documentation is available in MakeBoot.press.

Othello *Taft* *Introduction, Pilot User's Handbook*

This is the program for configuring disks, booting, etc.

Packager *Levin* *DF* *Pilot User's Handbook*

This program enables a user to repackage the procedures within a configuration to improve locality and reduce swapping.

PGS *Satterthwaite DF* *Contact maintainer*

LALR(1) parser generator for Mesa and Cedar. See maintainer if you think you need something like this.

PilotKernel *Levin* *Boot* *Pilot Programmer's Manual*

Pilot is the operating system kernel, which provides basic virtual memory management and local file access. Public interfaces are documented in the Pilot Programmer's Manual. Selected procedures in popular interfaces have been made "safe", in the Cedar sense.

RPCRuntime P *Birrell* *Boot* *LupineUsersGuide.press*

Runtime support for Remote Procedure Call (RPC). See Lupine for more details.

Sequin *Levin* *DF* *Contact maintainer*

SpecialTerminal *Levin* *Boot* *Contact maintainer*

This package permits TTY-style access to the "special" virtual terminal, used by Cedar initialization. The facilities formerly available through CedarControl (CedarInitOps and CedarInitPrivate) for using this virtual terminal are now collected in the interface SpecialTerminal.mesa. However, this is still a facility intended primarily for wizards; consult the implementor before attempting to use this interface.

Squirrel T *Cattell* *DF* *SquirrelDoc.tioga*

This is the alpha release of the Squirrel package of database tools, previously for use only by database wizards. Most of the Squirrel facilities are now available to end users, for creating and examining databases for which no special database application code has been written or is worth writing specially (documentation, bibliographies, service directories). Squirrel is also intended for database application writers, for examining and repairing databases, and can be used to coordinate two or more database applications. The documentation is in two parts, aimed at the end users and database application writers respectively.

STP P *Schmidt* *Boot* *Internal*

Obsolete, use UnsafeSTP for new applications.

TerminalMultiplex *Levin* *Boot* *Internal*

Provides for multiplexing of virtual terminals on the physical one. Although the interface is fairly general, certain of its facilities are effectively preempted by code in the boot file; as a result, potential users of this interface should contact the maintainer before attempting to use it.

TTYIO P *Teitelman* *Boot* *Contact maintainer*

This package includes the implementation of TTY in terms of IO (which will be disappearing one of these days), as well as the implementation of the interface ViewerIO, which contains the procedure CreateViewerStreams.

UnsafeSTP P *Atkinson* *Boot* *In source*

Low-level access to Pilot's Stream Transfer Protocol. Wizards only.

VersionMap P *Atkinson* *Boot* *In source*

Provides a fast mapping from version stamps to long file names.

VersionMapBuilder P *Atkinson* *DF* *In source*

Provides facilities for building version maps from various sources of version, name pairs.

WorldVM *Birrell* *Boot* *Contact maintainer*

Provides low-level access to remote machines. Wizards only.

Alphabetical Summary

AlpineUser P	Kolling	DF	Contact maintainer
AMEvents P	Birrell	Boot	Contact maintainer
AMModel P	Rovner	Boot	In source
AMTypes P	Rovner	Boot	RTTypes.tioga (NOT up to date yet)
Atom P	Teitelman	Boot	In source
BasicHeads	Taft	Boot	See interfaces, Contact maintainer
BCD	Satterthwaite	DF	Contact maintainer
Binder T	Satterthwaite	Client	Mesa 5.0 Manual, Mesa User's Handbook
BootClientFile	Levin	DF	Contact maintainer
BravoToTioga T	Paxton	DF	Here
BTree P	Taft	DF	BTree.tioga
BTrees P	Levin	DF	Contact maintainer
BugBane T	Atkinson	Boot	BugBane.tioga
CedarDB P	Cattell	DF	Internal and Contact maintainer
CedarReals P	Stewart	Boot	MesaFloat60.press
CedarRoot P	Levin	Boot	Contact maintainer
CedarScanner P	Atkinson	Boot	In source
CedarSnapshot	Levin	Boot	In source
Chat T	Teitelman	DF	Introduction, "Chat?" to UserExec
CIFS P	Schroeder	Boot	CIFSManual.tioga
CIFSCommands T	Schroeder	DF	CIFSManual.tioga
ClientFileTool T	Birrell	DF	Internal
Clock P	Atkinson	DF	Internal
CoFork	Sturgis	DF	Cofork.doc, .press
ColorPackage P	Stone	DF	Internal

Commander P	*Rovner*	*Boot*	*Commander.mesa*
Communication P	*Birrell*	*Boot*	*Pilot users manual*
CompatibilityPackage	*Levin*	*DF*	*Contact maintainer*
Compiler T	*Satterthwaite*	*Client*	*Cedar Language Reference Manual, Summary*
Cypress P	*Cattell*	*DF*	*<CedarDocs>Database><CypressDoc.press*
D0Microcode	*Fiala*		*Contact maintainer*
DateAndTime	*Levin*	*Boot*	*In source*
DFFiles T	*Schmidt*	*Client*	*DFFiles.tioga*
DLionMicrocode	*Sturgis*		*Contact maintainer*
DoradoMicrocode	*Taft*		*<DoradoDocs>DoradoBooting.press, Contact maintainer*
FileTool T	*Stewart*	*Boot*	*Introduction, "FileTool?" to UserExec*
FlushUnneededTempFiles P	*Levin*	*DF*	*Here*
Forms	*Horning*	*DF*	*Internal*
FTP P	*Birrell*	*DF*	*[Ivy]<Mesa>Doc>FTP.press, Contact maintainer*
Germ	*Taft*		*Pilot User's Handbook*
GL P	*Satterthwaite*	*Lister.df*	*Contact maintainer*
GrapevineUser P	*Birrell*	*Boot*	*<Grapevine>Docs>Interface.press,* *Grapevine.press*
Graphics P	*Wyatt*	*Boot*	*In source*
IFSFile	*Levin*	*DF*	*Contact maintainer*
IncludeChecker T	*Rovner*	*DF*	*Mesa User's Handbook*
Inscript	*Stone*	*Boot*	*InscriptImplementation.memo, .press,* *InscriptImplAppendix.memo*
Init	*Taft*	*DF*	*see release message*
IO P	*Teitelman*	*Boot*	*In source (IO, FileIO, UserProfile)*
List P	*Teitelman*	*Boot*	*In source and here*
Lister	*Satterthwaite*	*DF*	*Lister.txt*

Loader	*Rovner*	*Boot*	*Internal*
Lupine P	*Birrell*	*DF*	*LupineUsersGuide.press*
Maintain T	*Birrell*	*DF*	*Introduction, <Grapevine>Docs>Interface.press*
MakeBoot	*Levin*	*DF*	*MakeBoot.press*
MCross T	*Rovner*	*DF*	*MCross.doc*
Modeller T	*Satterthwaite*	*DF*	*ModelRefMan.press*
MonitoredQueue P	*Paxton*	*Boot*	*Internal*
MType T	*Schmidt*	*Client*	*Here, DFFiles.df*
OnlineMergeSort P	*Brown*	*DF*	*OnlineMergeSort.press, In source*
Othello	*Taft*		*Introduction, Pilot User's Handbook*
Packager	*Levin*	*DF*	*Pilot User's Handbook*
PerfStats P	*Brown*	*DF*	*PerfStats.press, In source*
PGS	*Satterthwaite*	*DF*	*Contact maintainer*
PilotKernel	*Levin*	*Boot*	*Pilot Programmer's Manual*
PlotPackage P	*Rovner*	*DF*	*In source*
Poplar P	*Schmidt*	*DF*	*Poplar.Press, EASPL.Press*
Print T	*Plass*	*Client*	*Print.doc*
PriorityQueue P	*Atkinson*	*DF*	*In source*
Pup P	*Birrell*	*Boot*	*[Ivy]<Mesa>*Pup.press*
Pupwatch T	*Birrell*	*DF*	*<Grapevine>Pupwatch>Pupwatch.press*
Random P	*Brown*	*DF*	*Random.press, In source*
RedBlackTree P	*Brown*	*DF*	*OrderedSymbolTable.press, In source*
RedBlackTreeRef P	*Brown*	*DF*	*OrderedSymbolTableRef.press, In source*
RefTab P	*Teitelman*	*Boot*	*In source and here*
Reminder T	*Rovner*	*DF*	*In source*
Rigging P	*Atkinson*	*Boot*	*Rope.tioga*

RopeFile P	Atkinson	DF	In source
RopeIO P	Paxton	Boot	In source
RopeReader P	Paxton	Boot	In source
RPCRuntime P	Birrell	Boot	LupineUsersGuide.press
SafeStorage P	Rovner	Boot	SafeStoragePrimer.tioga, SafeStorage.tioga
SampleTool T	McGregor	DF	In source
ScanZones P	Rovner	DF	In source
Sequin	Levin	DF	Contact maintainer
Set P	Rovner	DF	Set.doc
ShowTime P	Atkinson	Boot	Rope.tioga
SirPress P	Plass	Client	In source
SpecialTerminal	Levin	Boot	Contact maintainer
Spell P	Teitelman	Boot	Spell.mesa
Spy T	Maxwell	DF	Spy.doc
Squirrel T	Cattell	DF	SquirrelDoc.tioga
STP P	Birrell	Boot	Internal
STPServer T	Schmidt	DF	Here
TerminalMultiplex	Levin	Boot	Internal
Tioga T	Paxton	Boot	TiogaDoc.tioga
TiogaMenuOps P	Paxton	Boot	Internal
TiogaOps P	Paxton	Boot	Internal
TJaMGraphics P	Stone	DF	Internal
TSetter T	Plass	Client	TSetterDoc.tioga
TTYIO P	Teitelman	Boot	Contact maintainer
UECP P	Stewart	DF	In source
Unique T	Schmidt	DF	Unique.doc

UserCredentials P	*Levin*	*Boot*	*In source*
UserExec P T	*Teitelman*	*Client*	*UserExec.tioga*
UserProfile T	*Teitelman*	*Boot*	*In source*
UnsafeSTP P	*Atkinson*	*Boot*	*In source*
VersionMap P	*Atkinson*	*Boot*	*In source*
VersionMapBuilder P	*Atkinson*	*DF*	*In source*
Viewers P T	*McGregor*	*Boot*	*Introduction.tioga*
ViewRec P T	*Spreitzer*	*DF*	*ViewRec.Doc*
VTables P	*Atkinson*	*DF*	*In source*
Walnut T	*Willie-Sue*	*Client*	*WalnutDoc.tioga*
WalnutSend T	*Willie-Sue*	*Client*	*WalnutDoc.tioga*
Watch T	*Atkinson*	*Client*	*Here*
Waterlily T	*Kolling*	*DF*	*"Waterlily?" to UserExec*
WorldVM	*Birrell*	*Boot*	*Contact maintainer*