STANFORD ARTIFICIAL INTELLIGENCE PROJECT
OPERATING NOTE No. 57

SAIL

by

Dan Swinehart
and
Bob Sproull

STANFORD ARTIFICIAL INTELLIGENCE PROJECT
OPERATING NOTE No. 57

November, 1969

SAIL

by
Dan Swinehart
and
Bob Sproull

ABSTRACT: SAIL is a high-level programming system for the PDP-10 computer. It includes an extended ALGOL compiler and a companion set of execution-time routines. The reasonably standard ALGOL 60 compiler is extended to provide facilities for describing manipulations of an associative data structure. This structure contains information about ITEMS, stored as unordered collections of items (sets) or as ordered triples of items (associations). The algebraic capabilities of the language are linked to the associative capabilities by means of the DATUM operator, which can associate with any ITEM an algebraic datum.

The work reported here was supported in part by the Advanced Research Projects Agency of the Department of Defense under Contract SD-183.

TABLE OF CONTENTS

SECTION 18--BIBLIOGRAPHY

SECTION 1

INTRODUCTION

1-1. SAIL is a high-level programming system for the PDP-10 computer. It includes an extended ALGOL compiler and a companion set of execution-time routines. The reasonably standard ALGOL 60 compiler is extended to provide facilities for describing manipulations of an associative data structure. This structure contains information about ITEMS, stored as unordered collections of items (sets) or as ordered triples of items (associations). The algebraic capabilities of the language are linked to the associative capabilities by means of the DATUM operator, which can associate with any ITEM an algebraic datum.

1-2. Several forerunners (namely the GOGOL compilers developed at the Stanford Artificial Intelligence Project) have contributed to the general appearance of the non-associative portions of the SAIL language. The associative data structure is a slightly reworked version of the LEAP language, which was designed by J. Feldman and P. Rovner, and implemented on Lincoln Laboratory's TX-2. This language is described in some detail in an article entitled "An Algol-Based Associative Language" in the August, 1969 issue of the ACM Communications [Feldman]. The implementation was modified to tolerate the non-paging environment of the PDP-10.

1-3. SAIL in a sense has something for everyone. For those who think in ALGOL, SAIL has ALGOL. For those who want the most from the PDP-10 and the time-sharing system, SAIL allows flexible linking to hand-coded machine language programs. For those who have complex input/output requirements, the language provides complete access to the I/O facilities of the PDP-10 system. For those who aspire to speed, SAIL generates fairly good code. The user should, however, be warned that SAIL falls several man-decades short of the extensive testing and optimization efforts contained in the histories of most commercial compilers.

D. Swinehart
R. Sproull
November, 1969

SECTION 2

PROGRAMS, BLOCKS, STATEMENTS

SYNTAX

2-1.

```
<program>                 ::= <block>
                          ::= <entry_specification> <block>

<block>                   ::= <block_head> ; <compound_tail>

<block_head>              ::= BEGIN <declaration>
                          ::= BEGIN <block_name> <declaration>
                          ::= <block_head> ; <declaration>

<compound_tail>           ::= <statement> END
                          ::= <statement> END <block_name>
                          ::= <statement> ; <compound_tail>


<statement>               ::= <block>
                          ::= <compound_statement>
                          ::= <assignment>
                          ::= <conditional_statement>
                          ::= <if_statement>
                          ::= <go_to_statement>
                          ::= <for_statement>
                          ::= <while_statement>
                          ::= <do_statement>
                          ::= <case_statement>
                          ::= <return_statement>
                          ::= <done_statement>
                          ::= <next_statement>
                          ::= <leap_statement>
                          ::= <procedure_statement>
                          ::= <define_specification>
                          ::= <string_constant> <statement>
                          ::= <label_identifier> : <statement>
                          ::= <empty>

<compound_statement>      ::= BEGIN <compound_tail>
                          ::= BEGIN <block_name> <compound_tail>
```

<block_name> ::= <string_constant>

<entry_specification> ::= ENTRY <id_list>

EXAMPLES

2-2.
Given:
S is a statement,
Sc is a Compound Statement,
D is a Declaration,
B is a Block.

Then:
(Sc) BEGIN S; S; S; ... ; S END
(Sc) BEGIN "SORT" S; S; ... ;S END
(B) BEGIN D; D; D; ... ; S; S; S; ... ; S END
(B) BEGIN "ENTER NEW INFO" D; D; ... ; S; ... ;S END

are syntactically valid SAIL constructs.

SEMANTICS

Declarations

2-3. SAIL programs are organized in the traditional block structure of ALGOL-60.

2-4. Declarations serve to define the data types and dimensions of simple and subscripted (array) variables (arithmetic variables, strings, sets, and items). They are also used to describe procedures (subroutines) and name program labels. The DEFINE construct (see DECLARATIONS, 3-1, USE OF DEFINE, 12-0) may also appear in declarations.

2-5. Any identifier referred to in a program must be described in some declaration. An identifier may only be referenced by statements within the scope (see Scope of declarations, 3-8) of its declaration.

Statements

2-6. As in ALGOL, the statement is the fundamental unit of operation in the SAIL language. Since a statement within a block or compound statement may itself be a block or compound statement, the concept of statement must be understood recursively.

2-7. This definition of a block as a statement has virtues other than its syntactic niceness. In many ways a block behaves as a single complex statement; most importantly, no transfers (Jumps) may be made from outside a block to any statement within it except the first. This assures proper allocation and initialization of the data space for the block.

2-8. The block representing the program is known as the "outer block". All blocks internal to this one will be referred to as "inner blocks".

Block Names

2-9. The block name construct is used to describe the block structure of a SAIL program to a symbolic debugging routine (see DEBUGGING, 14-8). The name of the outer block becomes the title of the binary output file (not necesarily the file name). In addition, if a block name is used following an END, the compiler compares it with the block name which followed the corresponding BEGIN. A mismatch is reported to the user as evidence of a missing (extra) BEGIN or END somewhere.

2-10. The <string_constant> <statement> construct is equivalent in action to the <statement> alone; that is, the string constant serves only as a comment.

Entry Specifications

2-11. See Separately Compiled Procedures, 15-7.

SECTION 3

DECLARATIONS

SYNTAX

3-1.

```
<id_list>               ::= <identifier>
                        ::= <identifier> , <id_list>

<declaration>           ::= <type_declaration>
                        ::= <array_declaration>
                        ::= <preload_specification>
                        ::= <label_declaration>
                        ::= <procedure_declaration>
                        ::= <define_specification>
                        ::= <requirement>

<type>                  ::= <algebraic_type>
                        ::= <leap_type>
                        ::= <algebraic_type> <leap_type>
                        ::= <algebraic_type> ARRAY <leap_type>
                        ::= SET
                        ::= SET <leap_type>
                        ::= SET ARRAY <leap_type>
                        ::= <type_qualifier> <type>

<algebraic_type>        ::= REAL
                        ::= INTEGER
                        ::= BOOLEAN
                        ::= STRING

<leap_type>             ::= ITEM
                        ::= ITEMVAR
```

```
<type_qualifier>          ::= EXTERNAL
                          ::= INTERNAL
                          ::= SAFE
                          ::= FORWARD
                          ::= RECURSIVE
                          ::= FORTRAN
                          ::= GLOBAL


<type_declaration>        ::= <type> <id_list>


<array_declaration>       ::= <type> ARRAY <array_list>

<array_list>              ::= <array_segment>
                          ::= <array_list> , <array_segment>

<array_segment>           ::= <id_list> [ <bound_pair_list> ]

<bound_pair_list>         ::= <bound_pair>
                          ::= <bound_pair_list> <bound_pair>

<bound_pair>              ::= <lower_bound> : <upper_bound>

<lower_bound>             ::= <algebraic_expression>

<upper_bound>             ::= <algebraic_expression>

<preload_specification>   ::= PRELOAD_WITH <preload_list>

<preload_list>            ::= <preload_element>
                          ::= <preload_list> , <preload_element>

<preload_element>         ::= <constant>
                          ::= (constant) <constant>


<procedure_declaration>   ::= PROCEDURE <identifier> <procedure_head>
                                 <procedure_body>
                          ::= <type> PROCEDURE <identifier>
                                 <procedure_head> <procedure_body>

<procedure_head>          ::= <empty>
                          ::= ( <formal_param_decl> )
```

```
<procedure_body>          ::= <empty>
                          ::= ; <statement>

<formal_param_decl>       ::= <formal_parameter_list>
                          ::= <formal_parameter_list> ;
                                <formal_param_decl>

<formal_parameter_list>   ::= <formal_type> <id_list>

<formal_type>             ::= <simpler_formal_type>
                          ::= REFERENCE <simpler_formal_type>
                          ::= VALUE <simpler_formal_type>

<simpler_formal_type>     ::= <type>
                          ::= <type> ARRAY
                          ::= <type> PROCEDURE


<define_specification>    ::= DEFINE <definition_list>

<definition_list>         ::= <definition>
                          ::= <definition> , <definition_list>

<definition>              ::= <define_identifier> = <define_body>

<define_identifier>       ::= <identifier>
                          ::= <identifier> ( <id_list> )

<define_body>             ::= <string_constant>

<requirement>             ::= REQUIRE <require_list>

<require_list>            ::= <require_element>
                          ::= <require_list> , <require_element>

<require_element>         ::= PNAMES
                          ::= <arithmetic_constant> <space_spec>
                          ::= <string_constant> <relfile_spec>

<space_spec>              ::= STRING_SPACE
                          ::= SYSTEM_PDL
                          ::= STRING_PDL
                          ::= ARRAY_PDL
                          ::= NEW_ITEMS
```

<relfile_spec> ::= LOAD_MODULE
::= LIBRARY

RESTRICTIONS

3-2. For simplicity, the type_qualifiers are listed in only one syntactic class. Although their uses are always valid when placed according to the above syntax, most of them only have meaning when applied to particular subsets of these productions:

SAFE is only meaningful in array declarations
INTERNAL/EXTERNAL have no meaning in formal parameter declarations
FORWARD, RECURSIVE, and FORTRAN have meaning only in procedure type specifications.
ITEM ARRAYS do not exist (use ITEMVAR arrays).

3-3. For array declarations in the outer block substitute <constant> for <algebraic_expression> in the productions for <lower_bound> and <upper_bound>.

3-4. A label must be declared in the innermost block in which the statement being labeled appears.

3-5. The syntax for procedure declarations requires semantic embellishment (see Procedure Declarations, 3-37) in order to make total sense. In particular, a procedure body may be empty only in a restricted class of declarations.

EXAMPLES

3-6. Let I,J,K,L,X,Y, and P be identifiers, S a statement:

(<type_declaration>)

```
INTEGER I,J,K
EXTERNAL REAL X,Y
ITEM I
SET P
ITEMVAR X
REAL ITEM Y
INTEGER ARRAY ITEM J
INTERNAL STRING K
```

(<array_declaration>)

```
INTEGER ARRAY X[Ø:1Ø,Ø:1Ø]
REAL ARRAY Y[X:P(L)]; Comment Illegal
    in outer block
STRING ARRAY I[Ø:IF BIG THEN 3Ø ELSE 3]
ITEMVAR ARRAY K[Ø:5,1:L]
REAL ARRAY ITEMVAR ARRAY P[Ø:15]
```

(<label_declaration>)

```
LABEL L,X,Y
```

(<procedure declaration>)

```
PROCEDURE P; S
PROCEDURE P(INTEGER I,J;
    REFERENCE REAL X; REAL Y) ; S
INTEGER PROCEDURE P (REAL PROCEDURE L;
    STRING I,J; INTEGER ARRAY K); S
EXTERNAL PROCEDURE P(REAL X)
FORWARD INTEGER PROCEDURE X(INTEGER I)
FORTRAN REAL PROCEDURE SIN
```

(<define_specification>)

```
DEFINE CRLF = "CR&LF",
    TTY="1",
    TYPE(MSG)="OUT(TTY,MSG&CRLF)"
```

3-7. Note that these sample declarations are all given without the semicolons which would normally separate them from the surrounding declarations and statements. Here is a sample block to bring it all together (again, let S be any statement, D any declaration, and other identifiers as above:

```
BEGIN "SAMPLE BLOCK"
 INTEGER I,J,K;
 REAL X,Y;
 STRING A;
 INTEGER PROCEDURE P(REFERENCE REAL X; REAL Y);
  BEGIN
   D; D; D; ... ;S; ... ; S
  END "P";

 REAL ARRAY DIPHTHONGS[0:10,1:100];

 S;  S;  S;  S
END "SAMPLE BLOCK"
```

SEMANTICS

Scope of declarations

3-8. Every block automatically introduces a new level of nomenclature. Any identifier declared in a block's head is said to be LOCAL to that block. This means that:

a. The entity represented by this identifier inside the block has no existence outside the block.

b. Any entity represented by the same identifier outside the block is completely inaccessible (unless it has been passed as a parameter) inside the block.

3-9. An identifier occurring within an inner block and not declared within that block will be nonlocal (GLOBAL) to it; that is, the identifier will represent the same entity inside the block and in the block or blocks within which it is nested, up to and including the level in which the identifier is declared.

3-10. The Scope of an entity is the set of blocks in which the entity is represented, using the above rules, by its identifier. An entity may not be referenced by any statement outside its scope.

Type Declarations

3-11. SAIL reserves either one or two 36-bit words for each identifier appearing in a type declaration (exception -- no space is reserved for items -- see Item Declarations, 3-18). The use of these cells falls into two classes -- values and descriptors -- depending on the type preceding the identifier. If an identifier represents a REAL or INTEGER (BOOLEAN) variable or an ITEMVAR, its value is stored directly in the reserved cell. For strings (2 words, see String Declarations, 3-15) and sets (1 word, see Set Declarations, 3-24) internal descriptors are placed in the reserved cells which allow the running program to access these entities. These differences are not reflected in the SAIL syntax. The user may treat entities of both kinds as if their values were directly accessible in the reserved locations. For this reason we will henceforth refer synonymously to a simple identifier (one declared in a type declaration) and the simple variable it represents, as a "variable".

3-12. Items do not entirely conform to the structure described above. Please suppress any enpuzzlement concerning the roles of items and itemvars until after you have read the paragraph on Item Declarations, 3-18.

Numeric Declarations

3-13. Identifiers which appear in type declarations with types REAL or INTEGER can subsequently be used to refer to numeric variables. An Integer variable may take on values from -2↑35 to 2↑35-1. A Real variable may take on positive and negative values from about 10↑-38 to 10↑38 with a precision of 27 bits. REAL and INTEGER variables (and constants) may be used in the same arithmetic expressions; type conversions are carried out automatically (see Arithmetic Type Conversions, 8-22 below) when necessary.

3-14. The BOOLEAN type is currently identical to INTEGER. As you will see, BOOLEAN and algebraic expressions are really equivalent syntactically. The syntactic context in which they appear determines their meaning. Algorithms for determining the Boolean and algebraic inteterpretations of these expressions will be given below. The declarator BOOLEAN is included for program clarity.

String Declarations

3-15. A variable defined in a String declaration is a two-word descriptor containing the information necessary to represent a SAIL character string.

3-16. A String may be thought of as a variable-length, one-dimensional array of 7-bit ASCII characters. Its descriptor contains a character count and a byte pointer to the first character (see STRINGS, 16-14). Strings originate as constants at compile time (String Constants, 10-16), as the result of a String INPUT operation from some device (see Input, 11-41), or from the concatenation or decomposition of already existing strings (see Concatenation Operator, 8-37 and Substrings, 8-41).

3-17. When strings appear in arithmetic operations or vice-versa, a somewhat arbitrary conversion is performed to obtain the proper type (by arbitrary we do not mean to imply random -- see String-Arithmetic Conversions, 8-27). For this reason arithmetic and String variables are referred to as "algebraic variables" and their corresponding expressions are called "algebraic expressions". (Suggestions for a better term will be given a high priority). No other direct, or "forced", conversions (except for Integer/Real conversions) are present in the language.

Item Declarations

Prerequisite

3-18. Please make no attempt to understand the sections of this manual describing the associative capabilities of the SAIL language until you have read the article describing its basic flavor in [Feldman]. If you do not have access to a copy of the CACM, reprints are available from the authors. The structure and operations of the associative portions of LEAP and SAIL are so nearly identical that it seemed foolish to repeat them completely here. However, a full description of the syntax and a brief discussion of each construct is given here.

Items

3-19. The "Associative memory" of the SAIL system is constructed from a universe of items and a universe of associations among these items. An Item is an entity which is represented inside the machine by its internal name and is otherwise uninterpreted. Items may be combined to form "associations" which express facts (see Triples, 7-6). They may also be collected into unordered sets (Set Declarations, 3-24).

Item Genesis

3-20. The universe of items is divided into three classes differing in the way an Item enters it:

1) A declared Item results from each declaration of an identifier to be of type ITEM. The declaration causes a single internal name to be created for the item. Declared items do not obey the usual rules in recursive functions. In particular, items behave as if they were declared in the outer block. Although they may referred to by name only within the scope of their declarations (see Scope of declarations, 3-8), they may be accessed from outside the scope if they have been included in (and not removed from) any associations or sets, or assigned to itemvars which are still accessible. They are not deleted at block-exit. It might be helpful to think of declared items as the associative analogue of algebraic constants.

2) A created Item results from the execution of a NEW expresion (see NEW Items, 9-6). Any created Item may be deleted from the universe of Items (see DELETE, 7-10). Again, usual block structure rules do not apply to any Items.

3) An association Item results from the execution of a bracketed construction triple (Construction - Retrieval Distinction, 7-8). These may also be explicitly, but never automatically, deleted.

Datums

3-21. An Item of type 1) or 2) may have an associated value (Datum) of algebraic or SET type which can be used or altered like any other variable. This Datum may represent a simple or array variable of any type except ITEM or ITEMVAR. Datums may be referred to by use of the DATUM operator (Datums, 10-6, Datum Assignments, 4-7).

Itemvar Declarations

3-22. An Itemvar is a variable whose value is an Item (it is a reference to an Item). Just as the statements "X←3; A←X" and "A←3" are equivalent with respect to A, the statements "X←EDGE; A←X" and "A←EDGE" are equivalent with respect to A, if X and A are itemvars, EDGE an Item. The use of an Itemvar is equivalent to the use of the Item to which it refers. The difference is, of course that the itemvars may reference different items at different times.

3-23. Just as algebraic variables may be bound as loop variables in FOR statements, itemvars observe a special binding in the FOREACH statement. This very important construct is described in FOREACH Statement, 7-14 below.

Set Declarations

3-24. Because the answers to many associative questions are many-valued (all the sons of Harry, for example), sets of items are provided. A SAIL Set is an unordered collection of items containing at most one occurrence of any single item. The more common Set operations are available for convenient manipulation of sets.

Array Declarations

3-25. In general, any data type which is applicable to a simple variable may be applied in an array declaration to an array of variables. Note, however, the restriction (see RESTRICTIONS, 3-2) prohibiting ITEM ARRAY X as a legal declaration (ITEMS are "constants"), although ITEMVAR arrays are allowed. The entity represented by the name of an array, qualified with subscript expressions to locate a particular element (e.g. A[I,J]) behaves in every way like a simple variable. Therefore, in the future we shall refer to both simple variables and single elements of arrays (subscripted variables) as "variables". The formal syntax for <variable> can be found in Variables, 10-2.

3-26. Each subscript for an array which is not qualified by the SAFE attribute will be checked to ensure that it falls within the lower and upper bounds given for the dimension it specifies. An overflow triggers an error message and job abortion. The SAFE declaration inhibits this checking, resulting in faster, smaller, and bolder code.

3-27. Arrays are stored by rows. That is, if A[I,J] is stored in location 10000, then A[I,J+1] is stored in location 10001.

3-28. There is no limit to the number of dimensions allowed for an array. However, the efficiency of array references tends to decrease for large dimensions. Avoid large dimensionality if it is not necessary.

3-29. The item instances stored in an Itemvar array may have datums which are themselves algebraic or Set arrays. This provides a good deal of power, since an array of algebraic values can be dynamically associated with any item.

3-30. Arrays declared in the outer block must have constant bounds, since no variable may yet have been assigned a value. A certain degree of extra efficiency is possible in accessing these arrays, since they may be assigned absolute core locations by the compiler, eliminating some of the address arithmetic. Constant bounds always add a little efficiency, even in inner blocks.

3-31. For more details concerning the internal structure of arrays see DEBUGGING, 14-8, Separately Compiled Procedures, 15-7 and ARRAY IMPLEMENTATION, 16-33.

Preload Specifications

3-32. Any arithmetic or String array which is declared in the outer block may be "pre-loaded" with constant information by preceding its declaration with a <preload_specification>. This specification gives the values which are to be placed in consecutive core locations within all arrays declared immediately following the <preload_specification>. "Immediately", in this case, means all identifiers up to and including one which is followed by bound_pair_list brackets (e.g. in REAL ARRAY X,Y,Z[0:10],W[1:5]; -- preloads X,Y, and Z; not W). It is the user's responsibility to guarantee that the proper values will be obtained under the subscript mapping.

3-33. The original values of pre-loaded arrays will not be lost by restarting the program (most arrays are cleared when their declarations are processed), but they will not be re-initialized either. The values can be changed by assignment statements.

3-34. For string arrays, the original pre-loaded values remain if not changed by assignment statements. In general, however, String array elements whose values have been changed during program executions will be set to null strings when the program is restarted.

3-35. Algebraic type conversions will be performed at compile-time to provide values of the proper types to pre-loaded arrays. The compiler will not allow you to fill an array beyond its meager capacity to be filled. You may, however, provide a number of elements less than the total size of the array; remaining elements will be set to zero or the null string.

Example

3-36.

```
PRELOAD_WITH (5) Ø, 3, 4, (4) 6, 2;
INTEGER ARRAY TABL[1:4,1:3];
```

The first five elements of TABL will be initialized to Ø (parenthesized number is used as a repeat argument). The next two elements will be 3 and 4, followed by four 6's and a 2. The array will look like this:

	1	2	3
1	Ø	Ø	Ø
2	Ø	Ø	3
3	4	6	6
4	6	6	2

Procedure Declarations

3-37. If a procedure is typed, it may return a value (see Return Statement, 5-19) of the specified type. If formal parameters are specified, they must be supplied with actual parameters in a one to one correspondence when they are called (see Function Designators, 8-42 and Procedure Statements, 6-2).

Formal Parameters

3-38. Formal parameters, when specified, provide information to the body (executable portion) of the procedure about the kinds of values which will be provided as actual parameters in the call. The type and complexity (simple or array) are specified here. In addition, the formal parameter indicates whether the value (VALUE) or address (REFERENCE) of the actual parameter will be supplied. If the address is supplied, the variable whose identifer is given as an actual parameter may be changed by the procedure. This is not the case if the value is given.

3-39. To pass a PROCEDURE by value or an ITEM by reference has no readily determined meaning. ARRAYs passed by value (requiring a complete copy operation) have not yet been implemented. Therefore these cases are noted as errors by the compiler.

3-40. The proper use of actual parameters is further discussed in the paragraphs on Procedure Statements, 6-2 and Function Designators, 8-42.

Forward Procedure Declarations

3-41. A procedure's type and parameters must be described before the procedure may be called. Normally this is accomplished by entering the procedure declaration in the head of some block containing the call. If, however, it is necessary to have two procedures, declared in some block head, which are both accessible to statements in the compound tail of that block and to each other, the FORWARD construct permits the definition of the parameter information for one of these procedures in advance of its declaration. The procedure body must be empty in a forward procedure declaration. When the body of the procedure described in the forward declaration is actually declared, the types of the procedure and of its parameters must be identical in both declarations. The declarations must appear at the same level (within the same block head).

Example

3-42.

```
BEGIN "NEED FORWARD"
   FORWARD INTEGER PROCEDURE T1(INTEGER I); COMMENT PARAMS DESCRIBED;

   INTEGER PROCEDURE T2(INTEGER J);
       RETURN (T1(J)+3); COMMENT CALL T1 ;

   INTEGER PROCEDURE T1 (INTEGER I); COMMENT ACTUALLY DEFINE T1;
       RETURN (IF I=15 THEN I ELSE T2(I-1)); COMMENT CALLS T2;

...

K←T1(L); ... ; L←T2(K); ...

END "NEED FORWARD";
```

Notice that the forward declaration is required only because BOTH procedures are called in the body of the block. If only T1 were called from statements within the block, this example could be implemented as:

```
BEGIN "NO FORWARD"
   INTEGER PROCEDURE T1(INTEGER I);
   BEGIN
       INTEGER PROCEDURE T2(J);
           RETURN (T1(J)+3);

       RETURN( IF I=15 THEN I ELSE T2(I-1));
   END "T1";

...
   K←T1(L);
...
END "NO FORWARD";
```

Recursive Procedures

3-43. If a procedure is to be entered recursively, the compiler must be instructed to provide code for saving its local variables when the procedure is called and restoring them when it returns. Use the type-qualifier RECURSIVE in the declaration of any recursive procedure.

3-44. The compiler can produce much more efficient code for non-recursive procedures than for recursive ones. We feel that this gain in efficiency merits the necessity for declaring procedures to be recursive.

3-45. If a procedure which has not been declared recursive is called recursively, all its local variables (and temporary storage locations assigned by the compiler) will behave as if they were global to the procedure -- no values will be saved. Otherwise no ill effects should be observed.

External Procedures

3-46. A file compiled by SAIL represents either a "main" program or a collection of independent procedures to be called by the main program. The method for preparing such a collection of procedures is described in Separately Compiled Procedures, 15-7. The EXTERNAL and FORTRAN type-qualifiers allow description of the types of these procedures and their parameters. An EXTERNAL or FORTRAN procedure declaration, like the FORWARD declaration, does not include a procedure body. Both declarations instead result in requests to the loader to provide the addresses of these procedures to all statements which call them. This means that an EXTERNAL Procedure declaration (or the declaration of any External identifier) may be placed within any block head, thereby controlling the scope of this External identifier within this program.

3-47. Any SAIL procedure which is referenced via these external declarations must be an INTERNAL procedure. That is, the type-qualifier INTERNAL must appear in the actual declaration of the procedure. Again, see Separately Compiled Procedures, 15-7.

3-48. The type-qualifier FORTRAN is used to describe the type and name of an external procedure which is to be called using a DEC Fortran calling sequence. All parameters to Fortran procedures are by reference. In fact, the procedure head part of the declaration need not be included unless the types expected by the procedure differ from those provided by the actual parameters--the number of parameters supplied, and their types, are presumed correct. Fortran procedures are automatically External Procedures. See Restrictions on Procedure Declarations, 3-53, Procedure Statements, 6-2, Function Designators, 8-42 for more information about Fortran procedures.

Example:

3-49.
```
FORTRAN PROCEDURE MAX;
Y←MAX(X,Z);
```

Parametric Procedures

3-50. The calling conventions for procedures with procedures as arguments, and for the execution of these parametric procedures, are described in Procedure Statements, 6-2 and Function Designators, 8-42. Any procedure PP which is to be used as a parameter to another procedure CP must not have any procedure or array parameters, or any parameters called by value. In other words, PP may only have simple reference parameters. The number of parameters supplied in a call on PP within CP, and their types, will be presumed correct.

Example

3-51.

```
PROCEDURE CP (INTEGER PROCEDURE FP);
BEGIN INTEGER A,I; REAL X;
  ...
 A←FP(I,X); COMMENT I AND X PASSED BY REFERENCE,
             NO TYPE CONVERSION;
END "CP";

INTEGER PROCEDURE PP (REFERENCE INTEGER J; REFERENCE REAL Y);
BEGIN
  ...
END "PP";

...

CP(PP);
```

Defaults in Procedure Declarations

3-52. If no VALUE or REFERENCE qualification appears in the description, the following qualifications are assumed:

VALUE	Variables -- simple INTEGER, STRING, ITEM, ITEMVAR declarations.
REFERENCE	Arrays and Procedures.

Restrictions on Procedure Declarations

3-53.

1) The scope of a formal parameter for a procedure P does not include statements within any procedure Q declared within P. In other words, Q may refer only to its own formal parameters. It may, however, refer to variables which are local to some global procedure. Here is an example:

```
PROCEDURE P1(INTEGER I);
BEGIN INTEGER J;
 PROCEDURE P2(INTEGER K);
 BEGIN
  INTEGER L;
  L←I; COMMENT THIS IS WRONG -- WON'T WORK;
  L←J; COMMENT THIS IS ALL RIGHT;
  L←K; COMMENT CLEARLY ALL RIGHT;
  ...
```

2) There is no such thing as an ITEM procedure (use ITMVAR).

3) Fortran procedures can not handle String, Set, or Item parameters. Nor can a Fortran procedure return any of these types as a result.

4) Go To Statements appearing in a procedure body may not name statements outside that procedure body as targets.

5) Labels may never be passed as arguments to procedures.

Define Specification

3-54. See the section on USE OF DEFINE, 12-Ø for a complete discussion.

Requirements

3-55. The user may, using the REQUIRE construct, specify to the compiler conditions which are required to be true of the execution-time environment of his programs. The requirements fall into three classifications, described as follows:

Group 1 -- PNAMES

3-56. If the specification "REQUIRE PNAMES" appears in a program, the compiler is instructed to save the external representations (print names) of all declared Item identifiers. The functions CVIS and CVSI may be used to convert from Items to Strings representing the names of these Items (and back). This feature is not available unless "REQUIREd". See Cvis, 11-84 and following for details.

Group 2 -- Space requirements -- STRING_SPACE, SYSTEM_PDL, etc.

3-57. The inclusion of the specification "REQUIRE 1ØØØ STRING_SPACE" will ensure that at least 1ØØØ words of storage will be available for storing Strings when the program is run. Similar provisions are made for various push-down stacks used by the execution-time routines and the compiled code. If a parameter is specified twice, or if separately compiled procedures are loaded (see Separately Compiled Procedures, 15-7), the sum of all such specifications will be used. These parameters could also be typed to the loaded program just before execution (see STORAGE ALLOCATION, 13-22), but it is often more convenient to specify differences from the standard sizes in the source program. Use these specifications only if messages from the running program indicate that the standard allocations are not sufficient. "REQUIRE 3Ø NEW_ITEMS" specifies that 3Ø is a reasonable estimate of the number of items which will be created dynamically using the NEW construct.

Group 3 -- Other files -- LOAD_MODULE, LIBRARY

3-58. The inclusion of the specification REQUIRE "PROCS1" LOAD_MODULE, "HELIB[1,3]" LIBRARY; would inform the Loader that the file PROCS1,REL must be loaded and the library HELIB,REL[1,3] searched whenever the program containing the specification is loaded. The parameter for both features should be a string constant of one of the above forms. The device DSK, and file extension ,REL are the only values permitted for these entries, and are therefore assumed.

3-59. LOAD_MODULES (,REL files to be loaded) may themselves contain requests for other LOAD_MODULES and LIBRARYs. LIBRARYs may only contain requests for other LIBRARYs. Duplicate specifications are in general merged into single requests (if a file is requested twice, it will be loaded only once).

3-60. SAIL automatically places a request for the library "LIBSAI[1,3]" in each main program. This library contains the execution-time routines.

3-61. You have probably noticed that a great deal of prior knowledge is required for proper understanding of this section. For more information about PNAMES see CVIS, 11-84 and following. Storage allocation is discussed in STORAGE ALLOCATION, 13-22 below. The form and use of ,REL files and libraries are described in "The Stanford A-I Project Monitor Manual" [Moorer] and [Weiher].

SECTION 4

ASSIGNMENT STATEMENTS

SYNTAX

4-1.

```
<assignment>              ::= <assignment_statement>
                          ::= <swap_statement>

<item_assignment>         ::= <set_assignment>

<algebraic_assignment>    ::= <algebraic_variable> ←
                                <algebraic_expression>

<item_assignment>         ::= <itemvar_variable> ←
                                <construction_item_expression>

<set_assignment>          ::= <set_variable> ←
                                <construction_set_expression>


<swap_statement>          ::= <variable> ↔ <variable>
```

RESTRICTION

4-2. If the operator is ↔, the expression (of whatever kind) on the right hand side must be a simple or subscripted variable, or DATUM(<item_primary>). The ↔ operator may not be used in an assignment expression (see Assignment Expressions, 8-4). It is valid only at statement level.

SEMANTICS

4-3. The assignment statement causes the value represented by an expression to be assigned to the variable appearing to the left of the assignment symbol. You will see later (see Assignment Expressions, 8-4) that one value may be assigned to two or more variables through the use of two or more assignment symbols. The operation of the assignment statement proceeds in the following order:

a) The subscript expressions of the left part variable (if any) are evaluated from left to right.

b) The expression is evaluated.

c) The value of the expression is assigned to the left part variable, with subscript expressions, if any, having values as determined in step a.

4-4. This ordering of operations may usually be disregarded. However it becomes important when expression assignments (Assignment Expressions, 8-4) or function calls with reference parameters appear anywhere in the statement. For example, in the statements

```
I←3;
A[I]←3+(I←1);
```

A[3] will receive the value 4 using the above algorithm. If no subscript calculations were performed until after the expression evaluation, A[1] would become 4. Be careful.

4-5. As the syntax implies, if the left part variable is of type Itemvar the value to be assigned must be a construction Item expression. Similarly for sets.

4-6. However, any algebraic expression (REAL, INTEGER (BOOLEAN), or STRING) may be assigned to any variable of algebraic type. The resultant type will be that of the left part variable. The conversion rules for assignments involving mixed types are mildly amusing. They are identical to the conversion rules for combining mixed types in algebraic expressions (see Arithmetic Type Conversions, 8-22, String-Arithmetic Conversions, 8-27 below).

Datum Assignments

4-7. The algebraic or Set value associated with an Item is changed using an assignment statement in which the left part is a the word DATUM operating on an Item Primary. This is valid syntactically because the syntax for <variable> (see Variables, 10-2) includes this DATUM construct. The expression is checked for validity and proper type conversions are made before this kind of store occurs. One hazard is that there are times when the compiler cannot verify that an Item assigned to an Itemvar has a datum whose type matches that expected by the Itemvar. Incorrect conversions might well be made in this case.

Swap Assignment

4-8. The ↔ operator causes the value of the variable on the left hand side to be swapped with the value of the variable on the right hand side. Algebraic type conversions are made, if necessary; any other type conversions are, as usual, invalid. Remember, the ↔ operator may not be used in assignment expressions.

Examples

4-9.

```
X←I←A+B; Comment if A, B and X are Real, I Integer,
              the Real value of the sum is truncated,
              converted to an Integer, and stored in I,
              The truncated value is then converted to
              a Real number and stored in X.

BEGIN REAL ITEMVAR X;
  X←LOP(SET3);
  DATUM(X) ← 5; Comment a conversion to 5.0 will be made
      before the store is done, but there is no guarantee
      that the Item obtained by LOP(SET3) was not declared,
      for example, as INTEGER ITEM A;
END;
```

SECTION 5

EXECUTION CONTROL STATEMENTS

SYNTAX

5-1.

```
<conditional_statement>  ::= <if_statement>
                         ::= <if_statement> ELSE <statement>

<if_statement>           ::= IF <boolean_expression> THEN <statement>


<go_to_statement>        ::= GO TO <label_identifier>
                         ::= GOTO <label_identifier>
                         ::= GO <label_identifer>

<label_identifier>       ::= <identifier>


<for_statement>          ::= FOR <algebraic_variable> ← <for_list> DO
                                 <statement>
                         ::= NEEDNEXT <for_statement>

<for_list>               ::= <for_list_element>
                         ::= <for_list> , <for_list_element>

<for_list_element>       ::= <algebraic_expression>
                         ::= <algebraic_expression> STEP
                                 <algebraic_expression> UNTIL
                                 <algebraic_expression>
                         ::= <algebraic_expression> STEP
                                 <algebraic_expression> WHILE
                                 <boolean_expression>


<while_statement>        ::= WHILE <boolean_expression> DO <statement>
                         ::= NEEDNEXT <while_statement>
```

```
<do_statement>          ::= DO <statement> UNTIL <boolean_expression>
                        ::= NEEDNEXT <do_statement>

<case_statement>        ::= <case_statement_head> <compound_tail>

<case_statement_head>   ::= CASE <algebraic_expression> OF BEGIN

<return_statement>      ::= RETURN
                        ::= RETURN ( <expression> )

<done_statement>        ::= DONE

<next_statement>        ::= NEXT
```

SEMANTICS

Conditional Statements

5-2. These statements provide a means whereby the execution of a statement, or a series of statements, is dependent on the logical value produced by a Boolean expression.

5-3. A Boolean expression is an algebraic expression whose use implies that it is to be tested as a logical (truth) value. The rules for determining this value are given in Simple Expressions, 8-9 and following.

If Statement

5-4. The statement following the opeator THEN (the "THEN part") is executed if the logical value of the Boolean expression is TRUE; otherwise, that statement is ignored.

If ... Else Statement

5-5. If the Boolean expresion is true, the "THEN part" is executed and the statement following the operator ELSE (the "ELSE part") is ignored. If the Boolean expresion is FALSE, the "ELSE part" is executed and the "THEN part" is ignored.

Ambiguity in Conditional Statements

5-6. The syntax given here for conditional statements does not fully explain the correspondences between THEN-ELSE pairs when conditional statements are nested. An ELSE will be understood to match the immediately preceding unmatched THEN.

Example

5-7.

```
COMMENT DECIDE WHETHER TO GO TO WORK;

IF ¬WEEKEND THEN
    IF GIANTS_ON_TV THEN BEGIN
        PHONE_EXCUSE("GRANDMOTHER DIED");
        ENJOY(GAME);
        SUFFER(CONSCIENCE_PANGS)
    END
    ELSE IF REALLY_SICK THEN BEGIN
        PHONE_EXCUSE("REALLY SICK");
        ENJOY(Ø);
        SUFFER(AGONY)
    END
    ELSE GO_TO_WORK;
```

Go To Statements

5-8. Each of the three forms of the Go To statement means the same thing -- an unconditional transfer is to be made to the "target" statement labeled by the label identifier. The following rules pertain to labels:

1) All label identifiers used in a program must be declared. The declaration of a label must be local to the block immediately surrounding the statement it identifies. Note that compound statements (BEGIN-END pairs containing no declarations) are not blocks. Therefore the block

```
BEGIN "B1"
 INTEGER I,J; LABEL L1;
 ...
 IF BE3 THEN BEGIN "C1"
  ...
  L1:  ...
  ...
 END "C1";
 ...
 GO TO L1
END "B1"
```

Is legal.

2) No Go To Statement may specify a transfer from a statement S1 outside a given block to a target statement S2 inside that block. This is automatic from rule 1, since the label identifying S2 is not available to S1. Again the rule does not apply to compound statements, as the above example demonstrates.

3) No Go To statement may specify a transfer from a statement within a procedure to a statement outside that procedure (you can't jump out of procedures).

5-9. Labels will seldom be needed for debugging purposes. The block name feature (see DEBUGGING, 14-8) and the listing feature which associates with each source line the octal address of its corresponding object code (see Listing Features, 13-13) should provide enough information to find things easily.

5-10. Many program loops coded with labels can be alternatively expressed as For or While loops. This often results in a source program whose organization is somewhat more transparent, and an object program which is more efficient.

For Statements

5-11. For and While statements (see also FOREACH Statement, 7-14) provide methods for forming loops in a program. They allow the repetitive execution of a statement zero or more times. These statements will be described by means of SAIL programs which are functionally equivalent but which demonstrate better the actual order of processing. Refer to these equations for any questions you might have about what gets evaluated when, and how many times each part is evaluated.

5-12. Let VBL be any algebraic variable, AE1, ... , AE8 any algebraic expressions, BE a Boolean expresion, TEMP a temporary location, S a statement. Then the following SAIL statements are equivalent:

Using For Statements --

```
FOR VBL ← AE1, AE2, AE3 STEP AE4 UNTIL AE5,
    AE6 STEP AE7 WHILE BE, AE8 DO S;
```

Equivalent formulation without For Statements --

```
VBL←AE1;
S;
VBL←AE2;
S;
```

```
  Comment STEP-UNTIL loop;
    VBL←AE3;
LOOP1:
    IF VBL- (SIGN(AE4)*AE5) ≤ Ø THEN
    BEGIN
        S;
        VBL←VBL+AE4;
        GO TO LOOP1
    END;


  Comment STEP-WHILE loop;
    VBL←AE6;
LOOP2:
    IF BE THEN BEGIN
        S;
        VBL←VBL+AE7;
        GO TO LOOP2
    END;

    VBL←AE8;
    S;
```

If AE4 (AE7) is a variable, changing its value within the loop will cause the new value to be used for the next iteration. If AE4 (AE7) is a constant or an expression requiring evaluation of some operator, the value used for the step element will remain constant throughout the execution of the For Statement. If AE5 is an expression, it will be re-evaluated before each iteration.

5-13. Now consider the For Statement:

```
    FOR VBL←AE1 STEP CONST UNTIL AE2 DO S;
```

where const is a positive constant. The compiler will simplify this case to:

```
    VBL←AE1;
LOOP3:
    IF VBL ≤ AE2 THEN BEGIN
        S;
        VBL←VBL+CONST;
        GO TO LOOP3
    END;
```

If CONST is negative, the line at LOOP3 would be:

```
LOOP3:
    IF VBL ≥ AE2 THEN BEGIN
```

5-14. The value of VBL when execution of the loop is terminated, whether it be by exhaustion of the for list or by execution of a DONE or GO TO statement (see Done Statement, 5-23,Go To Statements, 5-8), is the value last assigned to it using the algorithm above. This value is therefore always well-defined.

5-15. The statement S may contain assignment statements or procedure calls which change the value of VBL. Such a statement behaves the same way it would if inserted at the corresponding point in the equivalent loop described above.

While Statement

5-16. The statement

```
    WHILE BE DO S;
```

is equivalent to the statements:

```
LOOP:
    IF BE THEN BEGIN
        S;
        GO TO LOOP
    END;
```

Do Statement

5-17. The statement

```
DO S UNTIL BE;
```

is equivalent to the sequence:

```
LOOP:
    S;
    IF ¬BE THEN GO TO LOOP;
```

Case Statements

5-18. The statement

```
CASE AE OF BEGIN
    SØ; S1; S2;... Sn
END
```

is functionally equivalent to the statements:

```
    TEMP←AE;
    IF          TEMP = Ø THEN SØ
        ELSE IF TEMP = 1 THEN S1
        ELSE IF TEMP = 2 THEN S2
        ...
        ELSE IF TEMP = n THEN Sn
    ELSE ERROR;
```

For applications of this type the CASE statement form will give significantly more efficient code than the equivalent IF statements. Notice that dummy statements may be inserted for those cases which will not occur or for which no entries are necessary. For example,

```
CASE AE OF BEGIN
    SØ; ; ; S3; ; ; S6; END
```

provides for no actions when AE is 1,2,4,5, or 7. When AE is Ø, 3, or 6 the corresponding statement will be executed.

Return Statement

5-19. This statement is invalid if it appears outside a procedure declaration. It provides for an early return from a procedure execution to the statement calling the procedure. If no return statement is executed, the procedure will return after the last statement representing the procedure body is executed (see Procedure Declarations, 3-37).

5-20. An untyped procedure (see Procedure Statements, 6-2) may not return a value. The return statement for this kind of procedure consists merely of the word RETURN. If an argument is given, it will cause the compiler to issue an error message.

5-21. A typed procedure (see Function Designators, 8-42) must return a value as it executes a return statement. If no argument is present an error message will be given. If the procedure has an algebraic type, any algebraic expression may be returned as its value; type conversion will be performed in a manner described by Arithmetic Type Conversions, 8-22 and String-Arithmetic Conversions, 8-27 below. If the procedure is of type SET or ITEM, the argument must be an expression of type SET or ITEM.

5-22. If no RETURN statement is executed in a typed procedure, the value returned is undefined (it could be anything -- try it, it's fun).

Done Statement

5-23. The statement containing only the word DONE may be used to terminate the execution of a FOR, WHILE, or FOREACH loop explicitly. Its operation can most easily be seen by means of an example. The statement

```
    FOR I←1 STEP 1 UNTIL n DO BEGIN
        S;
        ...
        IF BE THEN DONE;
        ...
    END
```

Is equivalent to the statement

```
    FOR I←1 STEP 1 UNTIL n DO BEGIN
        S;
        ...
        IF BE THEN GO TO EXIT;
        ...
    END;
EXIT:
```

In either case the value of I is well-defined after the statement has been executed (see For Statements, 5-14).

5-24. The DONE statement will only cause an escape from the innermost loop in which it appears.

Next Statement

5-25. A Next statement is valid only in a For Statement, While Statement, Do Statement, or Foreach Statement (see For Statements, 5-11, etc., FOREACH Statement, 7-14). processing of the loop statement is temporarily suspended. When the NEXT statement appears in a For or Foreach loop, the next value (set of items) is obtained from the For List (Associative Context) and assigned to the controlled variable (bound variables). The termination test is then made. If the termination condition is satisfied, control is passed to the statement following the For Statement or Foreach statement. If not, control is returned to the inner statement following the NEXT statement. In While and Do loops, the termination condition is tested. If it is satisfied, execution of the loop terminates. Otherwise it resumes at the statement within the loop following the NEXT statement.

5-26. The reserved word NEEDNEXT must precede FOR, WHILE, or DO in any loop using the NEXT statement.

Example

5-27.

```
NEEDNEXT WHILE ¬EOF DO BEGIN
     S←INPUT(1,1);
     NEXT;   Comment check EOF and terminate if TRUE;
     T←INPUT(1,3);
     PROCESS_INPUT(S,T);
END;
```

SECTION 6

PROCEDURE STATEMENTS

SYNTAX

6-1.

```
<procedure_statement>      ::= <procedure_identifier>
                           ::= <procedure_identifier> (
                                  <actual_parameter_list> )

<actual_parameter_list>    ::= <actual_parameter>
                           ::= <actual_parameter_list> ,
                                  <actual_parameter>

<actual_parameter>         ::= <expression>
                           ::= <array_identifier>
                           ::= <procedure_identifier>
```

SEMANTICS

6-2. A procedure statement is used to invoke the execution of an untyped procedure (see Procedure Declarations, 3-37). It may also be used to supply parameters to the procedure.

6-3. No value may be returned from a procedure called by a procedure statement, since there is no specification in the statement telling how to use the value. The compiler determines how a procedure may be used by noticing if a type was specified in the procedure declaration. After execution of the procedure, control returns to the statement immediately following the procedure statement.

Actual Parameters

6-4. The actual parameters supplied to a procedure must in general match the formal parameters described in the procedure declaration. As usual, the exception is algebraic expressions; the transfer functions described in Arithmetic Type Conversions, 8-22 and String-Arithmetic Conversions, 8-27 will be applied to convert the type of any algebraic expression passed by VALUE to the algebraic type required by the procedure.

Call by Value

6-5. If an actual parameter is passed by VALUE, only the value of the expression is given to the procedure. This value may be changed or examined by the procedure, but this will in no way affect any of the variables used to evaluate the actual parameters. Any algebraic expression, any Item or Set expression may be passed by value. Neither arrays nor procedures may be passed by value. See the default declarations for parameters in Procedure Declarations, 3-37.

Call by Reference

6-6. If an actual parameter is passed by REFERENCE, its address is passed to the procedure. All accesses to the value of the parameter made by the procedure are made indirectly through this address. Therefore any change the procedure makes in a reference parameter will change the value of the variable which was used as an actual parameter. This is sometimes useful. However if it is not intended, use of this feature can also be somewhat confusing as well as moderately inefficient. Reference parameters should be used only where needed.

6-7. Variables, constants, procedures, arrays, and most expressions may be passed by reference. Neither Items nor String expressions (or String constants) may be reference parameters.

6-8. If an expression is passed by reference, its value is first placed in a temporary location; a constant passed by reference is stored in a unique location. The address of this location is passed to the procedure. Therefore, any values changed by the procedure via reference parameters of this form will be inaccesible to the user after the procedure call. If the called program is an assembly language routine which saves the parameter address, it is dangerous to pass expressions to it, since this address will be used by the compiler for other temporary purposes. A warning message will be printed when expressions are called by reference.

6-9. The type of each actual parameter passed by reference must be identical to that of its corresponding formal parameter. An exception is made for Fortran calls (see Fortran Procedures, 6-12). If an algebraic type mismatch occurs the compiler will create a temporary variable containing the converted value and pass the address of this temporary as the parameter. A warning message will be printed.

Procedures as Actual Parameters

6-10. If an actual parameter to a procedure PC is the name of a procedure PR with no arguments, one of three things might happen:

1) If the corresponding formal parameter requires a value of a type matching that of PR (in the loose sense given above in Actual Parameters, 6-4), the procedure is evaluated and its value is sent to the procedure PC.

2) If the formal parameter of PC requires a reference procedure of identical type, the address of PR is passed to PC as the actual parameter.

3) If the formal parameter requires a reference variable, the procedure is evaluated, its result stored, and its address passed (as with expressions in the previous paragraph) as the parameter.

6-11. If a procedure name followed by actual parameters appears as an actual parameter it is evaluated (see Function Designators, 8-42). Then if the corresponding formal parameter requires a value, the result of this evaluation is passed as the actual parameter. If the formal parameter requires a reference to a value, it is called as a reference expression.

Fortran Procedures

6-12. If the procedure being called is a Fortran procedure, all actual parameters must be of type INTEGER (BOOLEAN) or REAL. All such parameters are passed by reference, since Fortran will only accept that kind of call. For convenience, any constant or expression used as an actual parameter to a Fortran procedure is stored in a temporary cell whose address is given as the reference actual parameter.

6-13. It was explained in Procedure Declarations, 3-37 that formal parameters need not be described for Fortran procedures. This allows a program to call a Fortran procedure with varying numbers of arguments, a feature which exists in DEC Fortran. No type conversion will be performed for such parameters, of course. If type conversion is desired, the formal parameter declarations should be included in the Fortran procedure declaration; SAIL will use them if they are present.

6-14. To pass an array to Fortran, mention the address of its first element (e.g. A[Ø], or B[1,1]).

Implementation Details

6-15. See the paragraphs concerning procedures in the section on Implementation (PROCEDURE IMPLEMENTATION, 16-46) for descriptions of the calling sequences and basic layout of SAIL procedures. See also Separately Compiled Procedures, 15-7 for more information about these useful constructs.

Examples:

6-16. To call an untyped procedure:

```
BEGIN
 ...;
 PROC(I+J,A[Q],L);
 ...
END;
```

To call a procedure of type Integer with one Integer argument:

```
I←PROC(PROC(I));
```

SECTION 7

LEAP STATEMENTS

SYNTAX

7-1.

```
<leap_statement>          ::= <set_statement>
                          ::= <associative_statement>
                          ::= <loop_statement>

<set_statement>           ::= <set_assignment>
                          ::= PUT <construction_item_expression> IN
                                 <set_variable>
                          ::= REMOVE <retrieval_item_expression> FROM
                                 <set_variable>

<associative_statement>   ::= <item_assignment>
                          ::= DELETE <retrieval_item_expression>
                          ::= MAKE <construction_triple>
                          ::= ERASE <retrieval_triple>


<loop_statement>          ::= FOREACH <binding_list>
                                 <associative_context> DO <statement>
                          ::= NEEDNEXT <loop_statement>

<binding_list>            ::= <id_list> |
                          ::= <id_list> SUCH THAT

<associative_context>     ::= <element>
                          ::= <associative_context> AND <element>
                          ::= <associative_context> ∧ <element>

<element>                 ::= <retrieval_associative_expression> IN
                                 <retrieval_set_expression>
                          ::= <retrieval_triple>
                          ::= ( <boolean_expression> )

<λ_triple>                ::= <λ_derived_set> ≡
                                 <λ_associative_expression>
```

SEMANTICS

LEAP Introduction

7-2. The basic ALGOL facility in SAIL has been extended with syntactic constructs and semantic interpretations to reference an associative data store. This extension was developed by J. Feldman and P. Rovner and is described in [Feldman]. The LEAP facilities in SAIL differ slightly from those published in the CACM article. In the discussion of the use of the associative facilities, reasonably simple examples are given for each construct. These examples and associated discussions should emphasize the differences between the SAIL implementation and the constructs published in the CACM article.

7-3. The LEAP constructs all involve manipulations of one basic entity, the item. An item is a conceptual entity which is represented at execution time by a unique number. Associated with each item in the universe is a DATUM. The DATUM of an item may be an algebraic quantity, an array of such quantities, or a SET. The DATUM assignment statement (see Datum Assignments, 4-7) is used to store the value of an expression into the DATUM of an item. The DATUM of a declared ARRAY ITEM is loaded automatically when the block in which the ARRAY ITEM is declared is entered. The DATUM of an item may also be referenced during evaluation of expressions (see Datums, 10-6). Examples:

```
INTEGER ITEM father,joe;
INTEGER ARRAY ITEM ages [1:20];
INTEGER a,b,c;

DATUM (father) ← 21 ;
DATUM (ages) [b] ← b / 33 ;
c ← DATUM (joe) - 12 ;
```

The DATUM operator is intended to link the powerful associative processing routines developed for manipulation of items with the algebraic facilities of ALGOL. This link is made as efficient as possible -- only two machine instructions are required to access the DATUM of an item.

7-4. Items or information about items may be stored in a variety of ways. The simple entity ITEM does not it itself occupy storage. Instead, instances of ITEMS are stored in ITEMVARS, SETS, or associations. The simplest of these forms is the ITEMVAR: an item may be "stored" in an ITEMVAR. Evaluation of that ITEMVAR will then yield the item stored into it. ITEMVARS are thus roughly analagous to simple arithmetic variables. SAIL also allows arrays of ITEMVARs, with the obvious interpretation. A typical declaration would be "ITEMVAR ARRAY x[1:22,0:1]", or "INTEGER ITEMVAR ARRAY y[1:20]".

7-5. Instances of items may also be stored as unordered collections, or SETS. Facilities are provided for common set operations (see Set Expressions, 9-2). The SAIL system uses one word of storage for each item in a set. A set will contain at most one instance of a specific item: if an instance of item X is already in set S, then any subsequent attempts to put an instance of X in S will have no effect. This is in keeping with the standard mathematical notion of set.

7-6. The third, and perhaps most important, form of storage of item instances is the association, or triple. Ordered triples of item instances may be written into or retrieved from a special store, the associative store. The method of storage of these triples is designed to facilitate fast and flexible retrieval. SAIL uses approximately two words of storage for each triple in the associative store. There is at most one copy of a triple in the store at any time. Once a triple has been stored in the associative memory, its component item instances may not be changed. In the examples which follow, a triple is represented by:

A • O ≡ V

where A, O, and V are items or itemvars. A, O, and V are mnemonics for the three components of a triple: attribute, object, and value. The exact syntactic rules for describing triples are discussed in SEMANTICS, 9-2.

General Restrictions

7-7. The implementation of the associative store and other forms of item storage imposes several limitations on the LEAP capability. The maximum number of items (as represented by their unique numbers) is 4090. This arises from an overwhelming desire to store a triple in one word of storage, and hence the requirement that an item number be describable in 12 bits.

Construction - Retrieval Distinction

7-8. There are two basic operations which are performed on the three types of item stores -- construction of a new element in that store, and retrieval of some existing element in the store. For some purposes, it is necessary to distinguish the operations being performed. This distinction manages to find its way to the syntax. In the discussion of associative expressions (Item Constructs, 9-4), the syntactic forms <construction_item_primary> and <retrieval_item_primary> are discussed. The ascent from primary level to associative expressions preserves these distinctions. Thus, one speaks of a <construction_item_expression>, or of a <retrieval_item_expression>. Often the BNF productions speak of <λ item_expressions>. This is merely a shorthand to denote that two separate sets of productions exist, one in which λ means "construction", and one in which λ means "retrieval".

PUT and REMOVE

7-9. The verbs PUT and REMOVE are provided for easily altering sets. After initialization, all sets are empty. They may be altered either by PUTting item instances into them or by explicit set assignment statements. The PUT statement is executed as follows: the construction item expression is evaluated, and must yield a single item. An instance of this item is then recorded in the set specified by the set variable. REMOVE operates in an analagous fashion. If an instance of the item to be REMOVEd does not occur in the set, an error message issues forth.

DELETE

7-10. DELETE releases an item from the universe of current items. Some small amount of storage is reclaimed in this process, as well as the unique number associated with the item DELETEd. Since there is an upper limit on the number of items, the DELETE statement can be used to free item numbers for other uses. The DELETE statement in no way alters the instances of the DELETEd item which are present in sets or associations. The user should be sure that there are no instances of the DELETEd item occurring in sets, itemvars or associations. Attempts to reference a DELETEd item in any way will result in confusion.

MAKE

7-11. Associations may be added to the associative memory with the MAKE statement. If the association already exists in the store, no alterations are made. The argument to the MAKE statement is a construction triple; that is, a triple composed of construction associative expressions. Every construct in these expressions is interpreted in a construction sense. The component associative expressions in this triple are evaluated left to right. Some constructs in these expressions (e.g. NEW, see NEW Items, 9-6 or in the case of bracketed triples) require that new unique item numbers be created. Examples:

```
MAKE item1 • item2 ≡ item3
MAKE item1 • itemvar1 ≡ NEW
MAKE item1•[item2•itemvar1≡item3]≡itemvararray[23]
```

7-12. The last example involves the use of a BRACKETED TRIPLE. The bracketed triple "[item2 • itemvar1 ≡ item]" which is used as an associative expression is inserted in the associative store. A new unique item number is generated, which refers to that association. Various functions (ISTRIPLE, FIRST, SECOND, THIRD -- see Item Selectors, 9-5) may use an instance of this new item as their argument. Consider the following statements:

```
MAKE number • [part • hand ≡ finger] ≡ new (5);
FOREACH x,y SUCH THAT number • x ≡ y AND
  (ISTRIPLE (x) AND FIRST (x) = part) DO
        count ← count + DATUM (y) ;
```

ERASE

7-13. The ERASE statement is provided to undo the damage done by the MAKE statement. The same general class of arguments must be provided. ERASE requires a retrieval triple as its argument, thus eliminating such questionable constructs as NEW from said triples. However, the construct ANY may appear in a triple specification to ERASE. This allows a whole slew of appropriate associations to be erased in one statement. (Restriction: ERASE ANY • ANY ≡ ANY is considered bad form, and is as a direct result, forbidden). Sample ERASE statements are:

```
ERASE Item1 • Item2 ≡ Item3
ERASE Item1 • Itemvar1 ≡ Item2
ERASE Itemvar1 • ANY ≡ Item1
```

FOREACH Statement

7-14. Flexible searching and retrieval are the main motivations for using the set and associative stores. The FOREACH statement provides this retrieval facility. The FOREACH statement is essentially a looping statement: the <statement> after the DO is executed for each group of item instances in the store which satisfies the FOREACH specification. If there are no such groups present in the store, the body of the statement is never executed. The <binding_list> specifies the itemvars which will contain results of the search. For instance, the simple construct FOREACH x SUCH THAT x IN set1 DO procedure(x) causes the body of the statement to be executed once for each item instance in the set set1. During execution of the body of the statement, the itemvar x evaluates to the item retrieved from the set set1. Consider, however, the FOREACH Statement

FOREACH x SUCH THAT x IN set1 AND x IN set2 DO statement

This specification may appear ambiguous, and indeed it is, unless we define the concept of BINDING the itemvars in a FOREACH specification. In an associative context, an itemvar which appears in the <binding_list> is said to be FREE until a search specification has determined the first requirement on the value of the itemvar (in a left-to-right scan of the <associative_context>). After the first requirement, it is said to be BOUND. Thus the <element> in the above example which reads "x IN set1" specifies a search in which x is free. The fact that x is free implies the searching operation. In the second element, "x IN set2", x is bound. Thus no search is conducted here. Instead, the question "Does an instance of the item I am considering for x appear in the set set2?" is evaluated. The answer must be TRUE in order that the statement be executed with x evaluating to that item. In summary, then, the FOREACH statement above specifies one search (x IN set1) and one additional requirement (x IN set2).

7-15. An element of a FOREACH specification may also be a parenthesized boolean expression. It is of course requisite that all itemvars appearing in the boolean expression must be bound, i.e. no searching of the associative store will be accomplished during the evaluation of the boolean expression. Example:

```
FOREACH x SUCH THAT x IN set1 AND ( DATUM (x) < 21 ) DO ...
```

Only members of set1 with DATUMs less than 21 will be selected by this specification. In the example above (FOREACH Statement, 7-14), the second <element> could also have been written in its boolean form: (x IN set2).

7-16. The most powerful <element> construct is a retrieval triple. Such specifications make searches (for any FREE itemvars) or verifications (in the case of completely BOUND elements) in the store of associations. For example:

```
1.    FOREACH x SUCH THAT a ⊛ o ≡ x DO PUT x IN set ;
2.    FOREACH x SUCH THAT a ⊛ o ≡ x AND b ⊛ g ≡ x DO ...
```

The aim of statement 1 is clear -- a search is conducted through the associative store for all associations with attribute "a" and object "o". If k such associations are discovered, then the body of the statement is executed k times, with x taking on successive values each time. The second example is similar, but places an additional constraint on the values of x which should be returned. Since the second element (b ⊛ g ≡ x) is completely BOUND, no search is conducted, but a test is made to verify that the association b ⊛ g ≡ x′ is in the store, where x′ is some item retrieved during the search for a ⊛ o ≡ x.

7-17. In general, an <associative context> is satisfied by some assignment of item instances to the itemvars in the <binding list> if all of the <element>s are satisfied under that assignment. A <boolean expression> is satisfied if it evaluates to TRUE. A <retrieval triple> containing no <set expression> is satisfied by an assignment if the association it specifies is in the universe of associations. A <retrieval triple> containing a <set expression> (or ANY) is satisfied if there are, in the universe of associations, any of the associations formed by substituting elements of the set (or arbitrary items) in the position occupied by the <set expression>.

7-18. With this concept of SATISFIERS, we proceed to the more general case with more than one itemvar cited in the binding list. Suppose there are α such itemvars. Then the <statement> is executed once for each permutation of the universe of items among the α itemvars which SATISFY the associative context. During the execution of the <statement>, the α itemvars will evaluate to the particular permutation which SATISFIED the associative context.

7-19. The above description for several itemvars is sound but slightly misleading. The SAIL implementation makes no effort to avoid duplicating a particular permutation of values which satisfies the associative context. Thus the <statement> will be executed one OR MORE times for every permutation which satisfies the associative context. (See Restrictions and Caveats, 7-21).

7-20. Examples of FOREACH statements with several free itemvars specified are:

1. FOREACH x,y,z SUCH THAT father•x≡y AND father•y≡z DO ...
2. FOREACH x,z SUCH THAT father• (father • x) ≡ z DO ...
3. FOREACH x,y SUCH THAT x IN set AND father • x ≡ y DO ...
4. FOREACH x,y SUCH THAT father • x ≡ y and x IN set DO ...

As it happens, 1 and 2 are equivalent. The compiler actually reduces 2 to 1 by including a dummy itemvar to be analagous to the use of "y" in the first example. Examples 3 and 4 are precisely equivalent, that is, the statement will be executed with x and y evaluating to all the ordered pairs of items which satisfy the (clearly equivalent) requirements. There is, however, a considerable difference in the execution efficiency of these two examples. Example 3 is more efficient since the "set" is probably quite small, and since the search of the associative memory with only one free itemvar in the search specification is rather fast. The second example, however, makes a search through the associative memory for all the (x,y) pairs and then discards those pairs for which an instance of x does not occur in the "set". Listed below in order of decreasing efficiency are the various basic forms of <element>s that are legal. The effect of a statement such as 2 above should be calculated by reducing it to the form of 1. In the list below, x, y, and z represent free itemvars, whereas A, O, and V represent either bound itemvars or fixed items.

A • O ≡ V	Verification that the triple is in the store.
A IN S	Verification that item A is in set S.
x IN S	All items x in the set S.
A • O ≡ x	Only the value is free.
x • y ≡ V	Attribute and object are free.
A • x ≡ V	Only the object is free.
x • O ≡ V	Only the attribute is free.
A • x ≡ y	Object and value are free.
x • O ≡ y	Attribute and value are free.
x • y ≡ z	PROHIBITED

Restrictions and Caveats

7-21. The SAIL implementation differs in fundamental ways from the implementation described by Feldman and Rovner in the CACM article. Their FOREACH statement builds a record of all the permutations which satisfy the associative context, being careful to include only one copy of each such permutation. Then the <statement> is executed once for each permutation that was stored during the retrieval operation. The SAIL implementation uses the associative context as a generator of satisfiers. Thus one group of satisfiers is found, <statement> is executed for those satisfiers, then another found, etc. until all groups of satisfiers have been found. The implications of this method are startling:

1. There is absolutely no way to guarantee that a particular group of satisfiers is not repeated. There are methods of coding around this problem. The user can stuff itemvar arrays with results of a FOREACH and avoid duplications. In many search specifications the nature of the searches (e.g. sets, where only one copy of an item instance can occur in the set) avoids duplicate satisfiers.
2. Operations within <statement> which change the associative data store may affect the subsequent satisfier groups retrieved. Note the difficulty in the following:

```
FOREACH x,y | link • x ≡ y DO MAKE link • x ≡ newlink
```

7-22. During and after the execution of a FOREACH statement, the values of the bound itemvars are in general well-defined. They evaluate to the permutation which last satisfied the FOREACH context. If a GO TO is executed within the <statement>, the values are correct in that they correspond to the group of satisfiers for which the <statement> was being executed. The only case in which the itemvars are undefined is when the search specified has been exhausted and the associative context contains a boolean expression. The explanation of this restriction is quite simple -- prior to the evaluation of a boolean expression, the core locations reserved for the itemvars in the <binding_list> are stuffed with the current satisfiers so that the evaluation of the boolean expression may reference them.

7-23. Expression case statements, conditional expressions, and procedure calls are all valid within an associative context specification, provided that all itemvars used in these constructs are BOUND.

SECTION 8

ALGEBRAIC EXPRESSIONS

SYNTAX

8-1.

```
<expression>                ::= <simple_expression>
                            ::= <conditional_expression>
                            ::= <assignment_expression>
                            ::= <case_expression>

<conditional_expression>    ::= IF <boolean_expression> THEN <expression>
                                  ELSE <expression>

<assignment_expression>     ::= <assignment_statement>

<case_expression>           ::= CASE <algebraic_expression> OF (
                                  <expression_list> )

<expression_list>           ::= <expression>
                            ::= <expression_list> , <expression>

<simple_expression>         ::= <algebraic_expression>
                            ::= <boolean_expression>
                            ::= <string_expression>
                            ::= <set_expression>
                            ::= <associative_expression>

<boolean_expression>        ::= <algebraic_expression>

<string_expression>         ::= <algebraic_expression>

<algebraic_expression>      ::= <disjuntive_expression>
                            ::= <algebraic_expression> v
                                  <disjunctive_expression>

<disjunctive_expression>    ::= <relational_expression>
                            ::= <disjunctive_expression> ^
                                  <relational_expression>
```

```
<relational_expression>  ::= <algebraic_relational>
                         ::= <leap_relational>

<algebraic_relational>   ::= <adding_expression>
                         ::= <relational_expression>
                                <relational_operator>
                                <adding_expression>

<leap_relational>        ::= <retrieval_item_expression> ∈
                                <retrieval_set_expression>
                         ::= <retrieval_set_expression>
                                <relational_operator>
                                <retrieval_set_expression>
                         ::= <retrieval_triple>

<relational_operator>    ::= <
                         ::= >
                         ::= =
                         ::= ≤
                         ::= ≥
                         ::= ≠

<adding_expression>      ::= <term>
                         ::= <adding_expression> <add_operator> <term>

<adding_operator>        ::= +
                         ::= -
                         ::= LAND
                         ::= LOR
                         ::= EQV
                         ::= XOR

<term>                   ::= <factor>
                         ::= <term> <mult_operator> <factor>

<mult_operator>          ::= *
                         ::= /
                         ::= %
                         ::= LSH
                         ::= ROT
                         ::= MOD
                         ::= DIV
                         ::= &

<factor>                 ::= <primary>
                         ::= <primary> ↑ <primary>
```

```
<primary>                ::= <algebraic_variable>
                         ::= - <primary>
                         ::= ¬ <primary>
                         ::= LNOT <primary>
                         ::= ABS <primary>
                         ::= <string_variable> [ <substring_spec> ]
                         ::= <constant>
                         ::= <function_designator>
                         ::= ( <algebraic_expression> )
                         ::= LENGTH ( <retrieval_set_expression> )
                         ::= LENGTH ( <string_expression> )
                         ::= CVN ( <item_primary> )
                         ::= LOP ( <string_variable> )
                         ::= ISTRIPLE ( <item_expression> )

<substring_spec>         ::= <algebraic_expression> TO
                                <algebraic_expression>
                         ::= <algebraic_expression> TO -
                         ::= <algebraic_expression> FOR
                                <algebraic_expression>

<function_designator>    ::= <procedure_identifier>
                         ::= <procedure_identifier> (
                                <actual_parameter_list> )

<actual_parameter_list>  ::= <actual_parameter>
                         ::= <actual_parameter_list> ,
                                <actual_parameter>

<actual_paremeter>       ::= <expression>
                         ::= <array_identifier>
                         ::= <procedure_identifier>

<algebraic_variable>     ::= <variable>

<string_variable>        ::= <variable>
```

SEMANTICS

Conditional Expressions

8-2. A conditional expression returns one of two possible values depending on the logical truth value of the Boolean expression. For the rules on evaluation of this truth value see Simple Expressions, 8-9 and following. If the Boolean expression (BE) is true, the value of the conditional expression is the value of the expression following the delimiter THEN. If BE is false, the other value is used. If both expressions are of an algebraic type, the precise type of the entire conditional expression is that of the "THEN part". Otherwise, both expressions must be of precisely the same type (Set, Item, etc.). Unlike the nested If statement problem, there can be no ambiguity for conditional expressions, since there is an ELSE part in every such expression.

Example

8-3.

```
FOURTHDOWN(YARDSTOGO,YARDLINE,IF YARDLINE < 70 THEN PUNT
      ELSE IF YARDLINE < 90 THEN FIELDGOAL
      ELSE RUNFORIT)
```

Assignment Expressions

8-4. The somewhat weird syntax for an assignment expression (it is equivalent to that for an assignment statement) is nonetheless accurate: the two function identically as far as the new value of the left part variable is concerned. The difference is that the value of this left part variable is also retained as the value of the entire expression. Assuming that the assignment itself is legal (following the rules given in Assignment Statements, 4-3 above), the type of the expression is that of the left part variable. This variable may now participate in any surrounding expressions as if it had been given its new value in a separate statement on the previous line. Only the ← operator is valid in assignment expressions. The ↔ operator is valid only at statement level.

Example

8-5.

```
IF (I←I+1) < 3Ø THEN I←Ø ELSE I←I+1;
```

Case Expressions

8-6. The expression

CASE AE OF (EØ, E1, E2, ... , En) is equivalent to

```
IF          AE=Ø THEN EØ
    ELSE IF AE=1 THEN E1
    ELSE IF AE=2 THEN E2
    ...
    ELSE IF AE=n THEN En
ELSE ERROR
```

8-7. The type of the entire expression is therefore that of EØ, If any of the expressions E1 ,,, En cannot be fit into this mold an error message is issued by the compiler.

Example

8-8.

```
OUT(TTY,CASE ERRNO OF("BAD DIRECTORY",
                      "IMPROPER DATA MODE",
                      "UNKNOWN I/O ERROR",
                      ...
                      "COMPUTER IN BAD MOOD"));
```

Simple Expressions

8-9. Simple expressions are simple only in that they are not conditional, case, or assignment expressions. There are in fact some exciting complexities to be discussed with respect to simple expressions, Set, Item, and Associative expressions are discussed in the next section. Before continuing with a description of algebraic expressions in the following paragraphs, an explanation of what is meant by a Boolean expression is in order.

The Boolean Expression Anomaly

8-10. You will notice that in the syntax a Boolean expression is said to be equivalent to an algebraic expression. This is simply a way of expressing syntactically that there are automatically invoked rules, 1) for obtaining a logical truth value from an expression which does not contain any logical operators or logical connectives, and 2) for obtaining an algebraic (Integer) value from one which does. The rules are very simple:

Integer, Real, or String to "Boolean"

8-11. The logical truth value of an expression 'X' which is of type Integer, Real, or String is the same as the truth value of the expression 'X≠Ø'. A String expression will be converted to an Integer one (see String-Arithmetic Conversions, 8-27) before the comparison is made. This need not be done for a Real expression, of course, since the Integer and Real representations for Ø are the same. This means you can write expressions of the form

```
IF I+3 THEN E1 ELSE E2        when you really mean
IF I+3≠Ø THEN E1 ELSE E2
```

One application of this rule can be found in several of the execution time routines (ENTER, LOOKUP, etc.) where an error flag is returned which is zero (FALSE) if the operation was successful and non-zero (TRUE) if an error occurred. This flag may be tested as a Boolean variable (IF FLAG THEN ERROR("LOOKUP FAILED")) or to determine exactly what went wrong by examining its actual value.

"Boolean" to Integer

8-12. The truth value of an expression containing logical operators and/or connectives may be determined by rules given below (see Algebraic Expressions, 8-16, Disjunctive Expressions, 8-19, Logical Expressions, 8-30). If this value is needed to determine which part to execute in a conditional statement, while statement, or conditional expression no actual numerical value need be created for the expression -- the tests which determine the truth value lead directly to the correct program branch. However, if this expression is combined with other algebraic expressions using some numeric operator, or if it is assigned to an algebraic variable, some actual value must be returned for the expression. If the expression is false, a zero is returned. A non-zero value indicates that the expression is true. The actual value returned for true expressions may differ from time to time, but it is guaranteed non-zero.

8-13.

Precedence of Algebraic Operators

8-14. The binary operators in SAIL generally follow "normal" precedence rules. That is, exponentiations are performed before multiplications or divisions, which in turn are performed before additions and subtractions, etc. The logical connectives ^ and v, when they occur, are performed last (^ before v). The exact precedence of operators is described in the syntax above. The order of operation can be changed by including parentheses at appropriate points (see Primaries, 8-39).

8-15. In an expression where several operators of the same precedence occur at the same level, the operations are performed from left to right. See Algebraic Expressions, 8-16, Disjunctive Expressions, 8-19 for special evaluation rules for logical connectives.

Algebraic Expressions

8-16. If an algebraic expression has as its major connective the logical connective "v", the expression has the logical value TRUE (arithmetic value some non-zero integer) if either of its conjuncts (the expressions surrounding the "v") is true; FALSE otherwise.

8-17. AvB does NOT produce the bit-wise Or of A and B if they are algebraic expressions. Truth values combined by numeric operators will in general be meaningless (use the operators LOR and LAND for bit operations).

8-18. The user should be warned that in an expression containing logical connectives, only enough of the expression is evaluated (from left to right) to uniquely determine its truth value. Thus in the expression

(J<3 v (K←K+1) > 0),

K will not be incremented if J is less than 3 since the entire expression is already known to be true. Conversely in the expression

(X ≥0 ^ SQRT(X)>2) (see Disjunctive Expressions, 8-19),

there is never any danger of attempting to extract the square root of a negative X, since the failure of the first test testifies to the falsity of the entire expression -- the SQRT routine is not even called in this case.

Disjunctive Expressions

8-19. If a disjunctive expression has as its major connective the logical connective "^", the expression has the logical value TRUE if both of its disjuncts are TRUE; FALSE otherwise. Again, if the first disjunct is FALSE a logical value of FALSE is obtained for the entire expression without further evaluation.

Relational Expressions

8-20. If any of the binary relational operators is encountered, code is produced to convert any String arguments to Integer numbers. Then type conversion is done as it is for + operations (see Arithmetic Type Conversions, 8-22). The values thus obtained are compared for the indicated condition. A Boolean value TRUE or FALSE is returned as the value of the expression. Of course, if this expression is used in subsequent arithmetic operations, a conversion to Integer (see "Boolean" to Integer, 8-12 above) is performed to obtain an Integer value.

8-21. Leap relational operators are discussed in depth in a later section.

Arithmetic Type Conversions

8-22. The binary arithmetic, logical, and String operations which follow will accept combinations of arguments of any algebraic types. The type of the result of such an operation is sometimes dependent on the type of its arguments and sometimes fixed. An argument may be converted to a different algebraic type before the operation is performed. The following table describes the results of the arithmetic and logical operations given various combinations of Real and Integer inputs. ARG1 and ARG2 represent the types of the actual arguments. ARG1* and ARG2* represent the types of the arguments after any necessary conversions have been made.

8-23.

OPERATION	ARG1	ARG2	ARG1*	ARG2*	RESULT
+ -	INT	INT	INT	INT	INT*
* ↑ %	REAL	INT	REAL	REAL	REAL
	INT	REAL	REAL	REAL	REAL
	REAL	REAL	REAL	REAL	REAL
LAND LOR	INT	INT	INT	INT	INT
EQV XOR	REAL	INT	REAL	INT	REAL
	INT	REAL	INT	REAL	INT
	REAL	REAL	REAL	REAL	REAL
LSH ROT	INT	INT	INT	INT	INT
	REAL	INT	REAL	INT	REAL
	INT	REAL	INT	INT	INT
	REAL	REAL	REAL	INT	REAL
/	INT	INT	REAL	REAL	REAL
	REAL	INT	REAL	REAL	REAL
	INT	REAL	REAL	REAL	REAL
	REAL	REAL	REAL	REAL	REAL
MOD DIV	INT	INT	INT	INT	INT
	REAL	INT	INT	INT	INT
	INT	REAL	INT	INT	INT
	REAL	REAL	INT	INT	INT

* Unless ARG2 is <0 for the operator ↑

8-24. An Integer is converted to a Real number in such a way that if this Real number is converted back to an Integer, the same Integer value will result. This is true unless the absolute value of the number is greater than 134217728. Some low-order significance will be lost for integers greater than this magnitude.

8-25. A Real number is converted to an Integer using the following formula:

Integer ← SIGN(Real)*(largest integer I such that I≤ABS(Real)).

This function will produce invalid results for Real numbers with a magnitude greater than 134217728.

8-26. If a String is presented as an argument to any of these operations, it is converted to an Integer. If an Integer or Real argument is presented to the concatenation operator (&), it is converted to a one-character string. Here are the rules:

String-Arithmetic Conversions

8-27. If a String is presented as an argument to an arithmetic operator, as a (value) parameter to a procedure which expects a Real or Integer value, or as an expression to be stored by an assignment statement into a Real or Integer variable, an Integer value is created for it as follows:

If the string is the null string (length=Ø), a Ø is returned as its 'Integer value'. Otherwise a word which has its lefthand 29 bits Ø, the rightmost 7 bits containing the first character of the String, is returned is its 'Integer value'. For instance, the String "ABCDE" has as its 'Integer value' '1Ø1, the octal representation of the letter 'A'. This Integer will then be converted to a Real number, if necessary.

8-28. If an Integer or Real number is presented where a String is expected, a one character String will be created whose character consists of bits 29-35 (the rightmost seven bits) of the numeric value. A Real number is not converted to an Integer before the conversion. For instance, the expression

"STRING"& '15 & '12

will result in a String which is 8 characters long. The last two characters are the ASCII codes for carriage return and line feed, respectively.

Adding Expressions

8-29. All the operators grouped in the semantic class <add_operator> all operate at the same precedence level. The user must sometimes provide parentheses in order to make the meaning of such expressions absolutely unambiguous. The + and - operators will do integer addition (subtraction) if both arguments are integers (or converted to integers from strings); otherwise, rounded Real addition or subtraction, after necessary conversions, is done.

8-30. LAND, LOR, XOR, and EQV carry out bit-wise And, Or, Exclusive Or, and Equivalence operations on their arguments. No type conversions are done for these functions. The logical connectives ∧ and ∨ do not have this effect -- they simply cause tests and jumps to be compiled. The type of the result is that of the first operand. This allows expressions of the form X LAND '777777777, where X is Real, if they are really desired.

8-31. Currently the values of the various overflow flags produced by these operators (and those which follow) are not available to the user.

Terms

Arithmetic Multiplicative Operators

8-32. The operation * (multiplication), like + and -, represents Integer multiplication only if both arguments are Integers; Real otherwise. Integer multiplication uses the IMUL machine instruction -- no double-length result is available.

8-33. The / operator (division) always does rounded Real division, after converting any Integer arguments to Real.

8-34. The % operator has the same type table as +, -, and *. It performs whatever division is appropriate.

8-35. LSH and ROT provide logical shift operations on their first arguments. If the value of the second argument is positive, a shift or rotation of that many bits to the left is performed. If it is negative, a right-shift or rotate is done. To obtain an arithmetic shift (ASH) operation, multiply or divide by the appropriate power of 2; the compiler will change this operation to a shift operation.

8-36. DIV and MOD force both arguments to be Integers before dividing. X MOD Y is the remainder after X DIV Y is performed (X MOD Y = X - (X DIV Y)*Y);

Concatenation Operator

8-37. This operator produces a result of type String. It is the String with length the sum of the lengths of its arguments, containing all the characters of the second string concatenated to the end of all the characters of the first. The operands will first be converted to strings if necessary as described in String-Arithmetic Conversions, 8-27 above. The normal use of the & operator is to collect lines of text, from several other string sources, which will subsequently be sent to an output device. Numbers can be converted to strings representing their external forms (and vice-versa) through explicit calls on execution time routines like CVS and CVD (see Execution Routines, 11-1 below).

Factors

8-38. A factor is either a primary or a primary raised to a power represented by another primary. As usual, evaluation is from left to right, so that A↑B↑C is evaluated as (A↑B)↑C. In the factor X↑Y, a suitable number of multiplications and additions is performed to produce an "exact" answer if Y is a positive integer. Otherwise a routine is called to approximate ANTILOG(Y LOG X). The result has the type of X in the former case. It is always of type Real in the latter.

Primaries

8-39. A primary represents an arithmetic or String value which always acts as a unit in any binary operation. It is either an expression surrounded by parentheses which indicate that all internal operations should be performed before combining it with other things, or one of myriad other constructs which will be considered separately.

Variables and Constants

8-40. These are clearly primary objects. They are values contained in specific core locations, or in parameter stacks, or in the case of some numeric constants, they are immediate operands.

Substrings

8-41. A String variable name which is qualified by a substring specification represents a part of the named string. ST[X FOR Y] represents the Xth through the (X + Y - 1)th characters of the String ST, ST[X TO Y] represents the Xth through Yth characters of ST, ST[X TO -] represents the Xth through LENGTH(X)th characters of ST. If at any time an attempt is made to compute a substring with a negative length, or with X<1, or with length L such that X+L-1 > LENGTH(ST), the job will be terminated with an error message. ST[X FOR Ø] is the null String (length = Ø, no characters).

Function Designators

8-42. A function designator defines a single value. This value is produced by the execution of a typed user procedure or of a typed execution-time routine (Execution Routines, 11-1). For a function designator to be an algebraic primary, its procedure must be declared to have an algebraic type. Untyped procedures may only be called from procedure statements (see Procedure Statements, 6-2). The value obtained from a user-defined procedure is that provided by a Return Statement within that procedure. If the procedure does not execute a Return Statement, the value might be anything at all. A Return Statement in a typed procedure must mention a value (see Return Statement, 5-19).

8-43. The rules for supplying actual parameters in a function designator are identical to those for supplying parameters in a procedure statement (see Procedure Statements, 6-2).

8-44. Several of the constructs given here as primaries have the form of function designators. However, the operations necessary to obtain the values of these constructs are generally compiled directly into the program. Descriptions of these functions follow:

Length

8-45. LENGTH is always an integer-valued function. If its argument is a set expression, the result is the number of items in the set. If the argument is a string, its length is the number of characters in the string. The length of an algebraic expression is always 1 (see String-Arithmetic Conversions, 8-27).

Lop

8-46. The LOP operator applied to a String variable removes the first character from the String and returns it in the form given in String-Arithmetic Conversions, 8-27 above. The String no longer contains this character. LOP applied to a null String has a zero value. If the argument is a Set expression the result is an item. This case is described below (Item Constructs, 9-4).

Cvn

8-47. CVN has as its value the Integer which is the internal representation of its Item argument. This function is highly implementation-dependent, and should only be used by people who are willing to follow the compiler writers around a lot. Its inverse function is CVI, described in Item Constructs, 9-4 below.

Lnot

8-48. The unary operator Lnot produces the bitwise complement of its (algebraic) argument. No type conversions (except strings to integers) are performed on the argument. The type of the result (meaningful or not) is the type of the argument.

Abs

8-49. The unary operator ABS is valid only for algebraic quantities. It returns the absolute value of its argument.

Unary Minus

8-50. -X is equivalent to (0-X). No type conversions are performed.

Boolean Primaries

8-51. The unary Boolean operator ¬ applied to an argument BE has the value TRUE if BE is false, and FALSE if BE is true. Notice that ¬A is not the bitwise complement of A, if A is an algebraic value. If used as an algebraic value, ¬A is simply Ø if A≠Ø (see "Boolean" to Integer, 8-12), some non-zero Integer otherwise.

8-52. Istriple (IE) is TRUE if IE is an Item which describes a bracketed triple. It is FALSE otherwise. If IE is not an Item expression, the compiler will complain bitterly.

ISTRIPLE ([A•B≡V]) is true.

ISTRIPLE (<declared item>) is false.

SECTION 9

SET AND ASSOCIATIVE EXPRESSIONS

SYNTAX

9-1.

```
<set_expression>          ::= <λ_set_expression>

<λ set_expression>        ::= <λ_set_term>
                          ::= <λ_set_expression> u <λ_set_term>

<λ set_term>              ::= <λ_set_factor>
                          ::= <λ_set_term> n <λ_set_factor>

<λ_set_factor>            ::= <λ_set_primary>
                          ::= <λ_set_factor> - <λ_set_primary>

<λ set_primary>           ::= PHI
                          ::= <set_variable>
                          ::= (λ_item_expr_list)
                          ::= ( <λ_set_expression> )
                          ::= <λ_derived_set>

<set_variable>            ::= <variable>

<λ item_expr_list>        ::= <λ_item_expression>
                          ::= <λ_item_expr_list> , <λ_item_expression>

<λ derived_set>           ::= <λ_associative_expr>
                                 <associative_operator>
                                 <λ_associative_expr>

<associative_operator>    ::= •
                          ::= `
                          ::= *


<associative_expression>  ::= <λ_associative_expr>

<λ associative_expr>      ::= <λ_item_expression>
                          ::= <λ_set_expression>
```

```
<λ_item_expression>          ::= <λ_item_primary>
                             ::= <selector> ( <λ_item_primary> )
                             ::= [ <λ_item_primary> • <λ_item_primary> ≡
                                    <λ_item_primary> ]

<construction_item_prim>     ::= <item_primary>
                             ::= NEW
                             ::= NEW ( <algebraic_expression> )
                             ::= NEW ( <array_name> )

<retrieval_item_prim>        ::= <item_primary>
                             ::= ANY

<item_primary>               ::= <item_identifier>
                             ::= <itemvar_variable>
                             ::= CVI ( <algebraic_expression> )
                             ::= COP ( <set_variable> )
                             ::= LOP ( <set_variable> )

<λ_triple>                   ::= <λ_derived_set> ≡
                                    <λ_associative_expression>

<selector>                   ::= FIRST
                             ::= SECOND
                             ::= THIRD

<itemvar_variable>           ::= <variable>

<set_variable>               ::= <variable>

<leap_relational>            ::= <retrieval_associative_expression> IN
                                    <retrieval_set_expression>
                             ::= <retrieval_associative_expression>
                                    <relational_operator>
                                    <retrieval_associatve_expression>
                             ::= <retrieval_triple>
```

SEMANTICS

Set Expressions

9-2. Three rather standard operators are implemented for use with sets. These are union (∪), intersection (∩), and subtraction (-). These operators have the standard mathematical interpretations. The only possible confusion pertains to subtraction: if we perform the set operation set1 - set2, and if there is an instance of an item x in set2 but not in set1, the subtraction proceeds and no error message is given.

Set Primaries

9-3. In addition to the <set_variable>, there are three set primaries: the empty set PHI, a set composed of a list of item expressions, and derived sets. The empty set is the set with a LENGTH of Ø. Its use is unrestricted. A set primary which results from a list of item expressions is put together as each item expression is evaluated. Derived sets are really sets of answers to questions which search the associative memory. The conventions are:

```
a • b          -- all x such that a • b ≡ x
a ` b          -- all x such that a • x ≡ b
a * b          -- (a • b) ∪ (a ` b)
```

Examples of set primaries:

```
PHI
( item1 , item2 , itemprocedure1 )
(item1 • itemvar1)
```

Item Constructs

9-4. There are several SAIL functions which yield items when evaluated. This is actually a rather ambiguous statement, since items as such have no real existence as entities to pass around in the breeze. But, of course, their unique identifier numbers may be passed about freely and indeed are, since the identifier number is sufficient to specify an item. As explained earlier, an itemvar evaluates to the item last "stored" in that itemvar. There are two functions provided for removing item instances from sets. The first of these is COP, which evaluates the <set_expression> argument and returns an instance of the first item in the set. The "first" item in a set is not well defined, since the sets are unordered. The value of the <set_expression> is unchanged. The function LOP is similar to COP in that its value is an instance of the first item in the set argument, but the item returned will be removed from the set if LOP is used. The set argument to LOP must be a <variable> for the simple reason that the set descriptor must be changed to reflect the removed item.

Item Selectors

9-5. The operators FIRST, SECOND, and THIRD are provided for decomposing bracketed triples (see Bracketed Triples, 7-12. The <item_primary> argument is assumed to be an instance of an item which was created for the bracketed association when the MAKE was executed. Examples:

FIRST ([a•o≡v]) evaluates to a.
SECOND ([a•o≡v]) evaluates to o.
THIRD ([a•o≡v]) evaluates to v.

NEW Items

9-6. The function NEW calls upon the associative store to refurbish a dusty old DELETEd item or to generate a new one. These new items become a part of the universe of existing items, and may be accessed and handled in precisely the same fashion as declared items. If NEW is used in an item expression, that expression is then constrained to be a construction item expression. NEW may also take an argument. In this case, the datum of the created item is preloaded with the value passed as argument. If this argument is algebraic (real or integer), then the datum will be of the same type. No type conversions are done when passing the algebraic argument. NEW will also accept an array name as argument. In this case, the created item will be of the type array. In fact, the array cited as argument will be copied into the newly created array. The new array will have the same bounds and number of dimensions as the array cited as argument. This array will not disappear until the OUTER block is exited.

ANY Construct

9-7. Some associative searches may need only partial specification -- particular portions of a foreach specification may be unimportant. The ANY construct is used to specify exactly which parts of the specification are "don't care"'s. Examples are:

```
FOREACH x SUCH THAT father • x ≡ ANY DO PUT x IN sons
```

CVI

9-8. The function CVI is provided for those people who insist on having the world at their disposal. The argument is an integer and the result is an instance of the item which uses that integer as its unique identifier. Absolutely no error checking is done. CVI is for daring men.

LEAP Booleans

9-9. Several boolean primaries are implemented for comparing sets and items. In the following discussion, "Ix" means item expression, and "se" means set expression. These are:
1. Set Membership. The boolean "Ix IN se" evaluates the set expression, and returns TRUE if the item value specified by the item expression is a member of the set.
2. Association Existence. The boolean " Ix • Ix ≡ Ix " returns TRUE if the association exists in the associative store. Examples:

```
IF father • x ≡ Joe THEN ...
IF father • Joe ≡ ANY THEN MAKE type • joe ≡ legitimate
```

3. Relations. The use of the third kind of boolean is more restricted than the syntax implies. Only the following relations are valid:

```
Ix = Ix          -- obvious interpretation
Ix ≠ Ix          -- obvious interpretation
se1 < se2        -- true if se1 is a proper subset of se2
se1 ≤ se2        -- true if se1 is identical to se2 or
                        if se1 is a proper subset of se2
se1 = se2        -- obvious interpretation
se1 ≠ se2        -- obvious interpretation
se1 > se2        -- equivalent to se2 < se1
se1 ≥ se2        -- equivalent to se2 ≤ se1
```

SECTION 10

BASIC CONSTRUCTS

SYNTAX

10-1.

```
<variable>              ::= <identifier>
                        ::= <identifier> [ <subscript_list> ]
                        ::= DATUM ( <item_identifier> )
                        ::= DATUM ( <item_identifier> ) [
                              <subscript_list> ]

<subscript_list>        ::= <algebraic_expression>
                        ::= <subscript_list> , <algebraic_expression>
```

SEMANTICS

Variables

10-2. If a variable is simply an identifier, it represents a single value of the type given in its declaration.

10-3. If it is an identifier qualified by a subscript list it represents an element from the array bearing the name of the identifier.

10-4. The array should contain as many dimensions as there are elements in the subscript list. A[I] represents the I+1th element of the vector A (if the vector has a lower bound of 0). B[I,J] is the element from the I+1th row and J+1th column of the two-dimensional array B. To explain the indexing scheme precisely, all arrays behave as if each dimension had its origin at 0, with (integral) indices extending infinitely far in either direction. However, only the part of an array between (and including) the lower and upper bounds given in the declaration are available for use (and in fact, these are the only parts allocated). If the array is not declared SAFE, each subscript is tested against the bounds for its dimension. If it is outside its range, a fatal message is printed identifying the array and subscript position at fault. SAFE arrays are not bounds-checked. Users must take the consequences of the journeys of errant subscripts for SAFE arrays. The bounds checking causes at least three extra machine instructions (two of which are always executed for valid subscripts) to be added for each subscript in each array reference. The algebraic expressions for lower and upper bounds in array declarations, and for subscripts in subscripted variables, are always converted to Integer values (see Arithmetic Type Conversions, 8-22) before use.

10-5. For more information about the implementation of SAIL arrays, see ARRAY IMPLEMENTATION, 16-33.

Datums

10-6. If the Item argument of DATUM has an algebraic datum, this value is returned. Otherwise the result is representative of some other data type and the value returned will have very little meaning as an algebraic value; it will probably be some internal pointer or something. This is mentioned here because there are times when the compiler will not be able to tell that such a type mismatch has occurred. Then it will be up to the user to interpret the strange results. If a Set is desired here, of course, the result is a Set primary and may be used as such.

Identifiers

10-7. You will notice that no syntax was included for the non-terminal symbols <identifier> or <constant>. It is far easier to explain these constructs in an informal manner.

10-8. A SAIL letter is any of the upper or lower case letters A through Z, or the underline character (_). Lower case letters are mapped into the corresponding upper case letters for purposes of symbol table comparisons (SCHLUFF is the same symbol as Schluff). A digit is any of the characters 0 through 9. An identifier is a string of characters consisting of a letter followed by any number of letters and digits (try us -- most text editors will give up before SAIL will). There must be a character which is neither a letter nor a digit (nor either of the characters "." or "$") both before and after every identifier. In other words, if YOU can't determine where one identifier ends and another begins in a program you have never seen before, well, neither can SAIL.

10-9. There is a set of identifiers which are used as SAIL delimiters (in the Algol sense -- that is, BEGIN is treated by Algol as if it were a single character. Such an approach is not practical, so a reserved identifier is used). These identifiers are called Reserved Words and may not be used for any purpose other than those given explicitly in the syntax. Another set of identifiers have preset declarations -- these are the execution time functions. These latter identifiers may be redefined by the user; they behave as if they were declared in a block surrounding the outer block. A list of reserved and predeclared identifiers follows:

Sail Reserved Words

10-10.

ABS AND ANY ARRAY ARRAY_PDL BEGIN BOOLEAN CASE COMMENT COMPLEX COP CVI CVN DATUM DEFINE DELETE DO DONE ELSE END ENTRY EQV ERASE EXTERNAL FALSE FIRST FOR FOREACH FORTRAN FORWARD FROM GLOBAL GO GOTO IF IN INTEGER INTERNAL ISTRIPLE ITEM ITEMVAR LABEL LAND LENGTH LIBRARY LOAD_MODULE LNOT LOP LOR LSH MAKE MOD NEEDNEXT NEXT NEW NEW_ITEMS NOT NULL OF OR OWN PHI PNAMES PRELOAD_WITH PROCEDURE PUT REAL RECURSIVE REFERENCE REMOVE REQUIRE RETURN ROT SAFE SECOND SET STEP STRING STRING_PDL STRING_SPACE SUCH SYSTEM_PDL THAT THEN THIRD TO TRIPLE TRUE UNTIL VALUE WHILE XOR

Sail Predeclared Identifiers

10-11,

ARRBLT ARRINFO ARRTRAN ARRYIN ARRYOUT BREAKSET CALL CLOSE CLOSIN CLOSOUT CLRBUF CODE CVASC CVD CVE CVF CVFIL CVG CVIS CVO CVOS CVS CVSI CVSIX CVSTR CVXSTR ENTER EQU GETCHAN GETFORMAT INCHRW INCHRL INCHRS INCHSL INCHWL INSTR INSTRL INSTRS INPUT INTIN LENGTH LINOUT LOOKUP MTAPE OPEN OUT OUTCHR OUTSTR REALIN RELEASE RENAME SCAN SETBREAK SETFORMAT STRBRK TTYIN TTYINL TTYINS WORDIN WORDOUT USERERR USETI USETO

10-12, Some of the reserved words are equivalent to certain special characters. These equivalences are:

CHARACTER	RESERVED WORD (s)
∧	AND
≡	EQV
¬	NOT
∨	OR
⊕	XOR
∞	INF
∈	IN
|	SUCH THAT

Arithmetic Constants

10-13,

12369 is an Integer with decimal value 12369
'12357 is an Integer constant with octal value 12357
123, is a Real constant with floating point value 123,Ø
Ø123,Ø is a Real constant with floating point value 123,Ø
,524 is a Real constant with floating point value Ø,524
5,3@4 is a Real constant with floating point value 53ØØ,Ø
5,342@-3 is a Real constant with value Ø,ØØ5342

10-14. If a . or a @ appears in a numeric constant, the type of the constant is returned as Real (even if it has an integral value). Otherwise it is an integer. Type conversions are made at compile time to make the type of a constant commensurate with that required by a given operation. Expressions involving only constants are evaluated by the compiler and the resultant values are substituted for the expressions.

10-15. The reserved word TRUE is equivalent to the Integer (Boolean) constant -1; FALSE is equivalent to the constant 0.

String Constants

10-16. A String constant is a string of ASCII characters (any which you can get into a text file) delimited at each end by the character ". If the " character is desired in the string, insert two " characters (after the initial delimiting " character, of course).

10-17. A String constant behaves like any other (algebraic) primary. It is originally of type String, but may be converted to Integer by extracting the first character if necessary (see String-Arithmetic Conversions, 8-27).

10-18. The reserved word NULL represents a String constant containing no characters (length=0).

Examples

10-19. The left hand column in the table that follows gives the required input format to obtain the strings given in the right-hand column:

INPUT	RESULT	LENGTH
"THIS IS A STRING"	THIS IS A STRING	16
"WHAT DOES ""FERNDOK"" MEAN?"	WHAT DOES "FERNDOK" MEAN?	25
"THIS IS HOW YOU TYPE A """	THIS IS HOW YOU TYPE A "	24
"""THIS IS A QUOTED STRING"""	"THIS IS A QUOTED STRING"	25
""		0
NULL		0

10-20. The scanning algorithm is altered somewhat if the String is being used as a macro body definition (see USE OF DEFINE, 12-0).

Comments

10-21. If the scanner detects the identifier COMMENT, all characters up to and including the next semicolon (;) will be ignored. A comment may appear anywhere as long as the word COMMENT is properly delimited (not in a String constant, of course);

10-22. A string constant appearing just before a statement also has the effect of a comment.

SECTION 11

EXECUTION TIME ROUTINES

GENERAL

Scope

11-1. A large set of pre-declared, built-in procedures and functions have been compiled into a library permanently resident on the system disk area (LIBSAI.REL[1,3]). The library also contains programs for managing storage allocation and initialization, and for certain String functions. If a user calls one of these procedures a request is automatically made to the loader to include the procedure, and any other routines it might need, in the core image. These routines provide input/output (I/O) facilities, arithmetic-String conversion facilities, array-handling procedures and miscellaneous other interesting functions.

11-2. The remainder of this section describes the calling sequences and functions of these routines.

Notational Conventions

11-3. A short-hand is used in these descriptions for specifying the types (if any) of the execution-time routines and of their parameters. Before the description of each routine there is a sample call of the form

```
VALUE ← FUNCTION ( ARG1, ARG2, ... ARGn )
```

If VALUE is omitted, the procedure is an untyped one, and may only be called at statement level (Procedure Statements, 6-2).

11-4. The types of VALUE and the arguments may be determined using the following scheme:

1) If " characters surround the sample identifier (which is usually mnemonic in nature) a String argument is expected. Otherwise the argument is Integer or Real. If it is important which of the types Integer or Real must be presented, it will be made clear in the description of the function. Otherwise the compiler assumes Integer arguments (for those functions which are predeclared). The user may pass Real arguments to these routines (WORDOUT, for example) by re-declaring them in the blocks in which the Real arguments are desired.

2) If the @ character precedes the sample identifier, the argument will be called by reference. Otherwise it is a value parameter.

Example

11-5.

```
"RESULT" ← SCAN ( @"SOURCE", BREAK_TABLE, @BRCHAR)
```

is a predeclared procedure with the implicit declaration:

```
EXTERNAL STRING PROCEDURE SCAN (REFERENCE STRING SOURCE;
                                INTEGER BREAK_TABLE;
                                REFERENCE INTEGER BRCHAR);
```

I/O ROUTINES

Open

Form:

11-6. OPEN(CHANNEL,"DEVICE",MODE,NUMBER_OF_INPUT_BUFFERS, NUMBER_OF_OUTPUT_BUFFERS,@COUNT,@BRCHAR,@EOF);

Function:

11-7. SAIL Input/output operates at a very low level In the following sense: the operations necessary to obtain devices, open and close files, etc., are almost direct translations into a functional notation of the system calls used in assembly language. OPEN is used to associate a channel number (Ø to '17) with a device, to determine the data mode of the I/O to occur on this channel (character mode, binary mode, dump mode, etc.), to specify storage requirements for the data buffers used in the operations, and to provide the system with information to be used for input operations.

CHANNEL is a user-provided channel number which will be used in subsequent I/O operations to identify the device, CHANNEL may range from Ø to 15 ('17). Needless to say, only one device may be active on a given channel at one time.

DEVICE must be a String (i.e. "TTY", "DATA") which is recognizable by the system as a physical or logical device name. The TTY will be opened only once. The effect of subsequent OPENs (on different channels) is to equate all channels mentioned by the user to that mentioned in the first OPEN for the TTY. Be sure to release this first channel last.

MODE is the data mode for the I/O operation, MODE Ø will always work for characters (see Input, 11-41 and Out, 11-46). Modes 8 ('1Ø) and 15 ('17) are applicable for binary and dump-mode operations using the functions WORDIN, WORDOUT, ARRYIN, or ARRYOUT (see Wordin, 11-49 and following). For other data modes, see [Moorer].

NUMBER_OF(INPUT/OUTPUT)_BUFFERS specifies the number of buffers to be reserved for the I/O operations (see [Moorer] for details). At least one buffer must be specified for input if any input is to be done in modes other than '17; similarly for output. If data is only going one direction, the other buffer specification should be Ø. Two buffers give reasonable performance for most devices (1 is sufficient for a TTY, more are required for DSK if rapid operation is desired).

11-8. The remaining arguments are applicable only for INPUT (String Input). They will be ignored for any other operations (although their values may be changed by the Open function).

COUNT designates a variable which will contain the maximum number of characters to be read from "DEVICE" in a given INPUT call (see Input, 11-41, Breakset, 11-23). Fewer characters may be read if a break character is encountered or if an end of file is detected. The count should be a variable or constant (not an expression), since its address is stored, and the temporary storage for an expression may be re-used.

BRCHAR designates a variable into which the break character (see INPUT and BREAKSET again) will be stored. This variable can be tested to determine which of many possible characters terminated the read operation.

EOF designates a variable to be used for two purposes:

1) If EOF is Ø when OPEN is called, a SAIL error message will be invoked if the device is not available or the channel is already open. The user will be given the options of retrying or terminating the operation. If EOF is non-zero when OPEN is called, it will be set to Ø if the OPEN is successful. Otherwise it will not be changed. In this case (EOF non-zero on entry) control will be returned to the user. This flag may then be tested.

2) EOF will be made non-zero (TRUE) if an end of file condition is detected during any SAIL input operation. It will be Ø (FALSE) on return to the user otherwise. Subsequent inputs after an EOF return will return non-zero values in EOF and a null String result for INPUT. For ARRYIN , a Ø is returned as the value of the call after end of file is detected.

Assembly Language Approximation to OPEN

11-9,

```
        INIT    CHANNEL,MODE
        SIXBIT  /DEVICE/
        XWD     OHED,IHED
        JRST    <handle error condition>
        JUMPE   <NUMBER_OF_OUTPUT_BUFFERS>,GETIN
        <allocate buffer space>
        OUTBUF  CHANNEL,NUMBER_OF_OUTPUT_BUFFERS
GETIN:  JUMPE   <NUMBER_OF_INPUT_BUFFERS>,DONE
        <allocate buffer space>
        INBUF   CHANNEL,NUMBER_OF_INPUT_BUFFERS
DONE:   <mark channel open -- internal bookkeeping>
        <return>

OHED:   BLOCK   3
IHED:   BLOCK   3
```

CLOSE, CLOSIN, CLOSO

Form:

11-10,

```
CLOSE ( CHANNEL )
CLOSIN ( CHANNEL )
CLOSO ( CHANNEL )
```

Function:

11-10, The input (CLOSIN) or output (CLOSO) side of the specified channel is closed; all output is forced out (CLOSO); the current file name is forgotten. However the device is still active; no OPEN need be done again before the next input/output operation. No INPUT, OUT, etc. may be given to a directory device until an ENTER, LOOKUP, or RENAME has been issued for the channel.

11-11, CLOSE is equivalent to the execution of both CLOSIN and CLOSO for the channel.

Getchan

Form:

11-12. VALUE ← GETCHAN;

Function:

11-13. The number of some channel not currently open is returned. -1 is returned is all channels are busy.

Release

Form:

11-14. RELEASE (CHANNEL);

Function:

11-15. If an OPEN has been executed for this channel, a CLOSE is now executed for it. The device is dissociated from the channel and returned to the resource pool (unless it has been assigned by the monitor ASSIGN command). No I/O operation may refer to this channel until another OPEN denoting it has been executed.

11-16. If you have opened more than one channel for the device "TTY", be sure to perform RELEASE's for these channels in the inverse order from that in which they were opened (see Open, 11-6).

11-17. Release is always valid. If the channel mentioned is not currently open, the command is simply ignored.

Lookup_Enter

Form:

11-18.
LOOKUP (CHANNEL , "FILE" , @FLAG);
ENTER (CHANNEL , "FILE" , @FLAG);

Function:

11-19. Before input or output operations may be performed for a directory device (DECtape or DSK) a file name must be associated with the channel on which the device has been opened (see Open, 11-6). LOOKUP names a file which is to be read, ENTER names a file which is to be created or extended (see [Moorer]). Both operations are valid even if no filename is really necessary. It is recommended that an ENTER be performed after every OPEN of an output device so that output not normally directed to the DSK can be directed there for later processing if desired. The format for a file name string is

NAME ,
NAME,EXT ,
NAME[P,PN] ,
or NAME,EXT[P,PN] (see [Moorer] for the meaning of these things if you do not immediately understand),

All characters are converted to SIXBIT by subtracting octal '40 from them. Lower case letters are first converted to upper case. SAIL is not as choosy about the characters it allows as PIP and other processors are. Any character which is not ",", ",", "[", or "]" will be converted and passed on. Up to 6 characters from NAME, 3 from EXT, P, or PN will be converted -- the rest are ignored.

11-20. If the LOOKUP or ENTER operation fails (see [Moorer]) then variable FLAG may be examined to determine the cause. The left half of FLAG will be set to '777777 (Flag has the logical value TRUE). The right half will contain the code returned by the system giving the cause of the failure.

11-21. If the LOOKUP or ENTER succeeds, FLAG will be set to zero (FALSE).

Rename

Form:

11-22. RENAME (CHANNEL , "FILE-SPEC" , PROTECTION , @FLAG);

Function:

11-23. The file open on CHANNEL is renamed to FILE_SPEC (a NULL file-name will delete the file) with read/write protection as specified in PROTECTION (nine bits, described in the time-sharing manual). FLAG is set as in LOOKUP and ENTER.

Breakset

Form:

11-23. BREAKSET(TABLE, "BREAK_CHARS" , MODE);

Function:

11-24. Character input/output is done using the String features of SAIL. In fact, I/O is the chief justification for the existence of strings in the language.

String input presents a problem not present in String output. The length of an output String can be used to determine the number of characters written. However it is often awkward to require an absolute count for input. Quite often one would like to terminate input, or "break", when one of a specified set of characters is encountered in the input stream. In SAIL, this capability is implemented by means of the BREAKSET, INPUT, TTYIN, and SCAN functions.

11-25. The value of TABLE may range from 1 to 18. Thus up to 18 different sets of break specifications may exist at once, which set will be used is determined by the TABLE parameter in an INPUT or SCAN function call.

11-26. The function of a given BREAKSET command depends on the MODE, an integer which is interpreted as a right-justified ASCII character whose value is intended to be vaguely mnemonic. BREAKSET commands can be partitioned into 3 groups according to mode:

GROUP 1 -- Break character specifications

11-27.

MODE FUNCTION

"I" (by Inclusion) The characters in the BREAK_CHARS String comprise the set of characters which will terminate an INPUT (or SCAN).

"X" (by eXclusion) Only those characters (of the possible 128 ASCII characters) which are NOT contained in the String BREAK_CHARS will terminate an input when using this table.

"O" (Omit) The characters in "BREAK_CHARS" will be omitted (deleted) from the input string.

11-28. Any "I" or "X" command completely specifies the break character set for its table (i.e., the table is reset before these characters are stored in it). Neither will destroy the omitted character set currently specified for this table. Any "O" command completely specifies the set of omitted characters, without altering the break characters for the table in question. If a character is a break-character, any role it might play as an omitted character is sacrificed.

11-29. The second group of MODEs determines the disposition of break characters in the input stream. The "BREAK_CHARS" argument is ignored in these commands, and may in fact be NULL:

GROUP 2 -- Break character disposition

11-30.

MODE	FUNCTION
"S"	(Skip -- default mode) After execution of an "S" command the break character will not appear either in the resultant String or in subsequent INPUTs or SCANs-- the character is "skipped". Its value may be determined after the INPUT by examination of the break character variable (see Open, 11-6).
"A"	(Append) The break character (if there is one -- see Open, 11-6 and Input, 11-41) is appended, or concatenated to the end of the input string. It will not appear again in subsequent inputs.
"R"	(Retain) The break character does not appear in the resultant INPUT or SCAN String, but will be the first character processed in the next operation referring to this input source (file or SCAN String).

11-31. For disk and tape files using the standard editor format, line numbers present a special problem. A line number is a word containing 5 ASCII characters representing the number in bits 0-34, with a "1" in bit 35. No other words in the file contain 1's in bit 35. Since String manipulations provide no way for distinguishing line numbers from other characters, there must be a way to warn the user that line numbers are present, or to allow him to ignore them entirely.

11-32. The third group of MODEs determines the disposition of these line numbers. Again, the "BREAK_CHARS" argument is ignored:

Group 3 -- Line number disposition

11-33.

MODE	FUNCTION
"P"	(Pass -- default) Line numbers are treated as any other characters. Their identity is lost; they simply appear in the result string.
"N"	(No numbers) No line number (or the TAB which always follows it in standard files) will appear in the result string. They are simply discarded.
"L"	(Line no. break) The result String will be terminated early if a line number is encountered. The characters comprising the line number and the associated TAB will appear as the next 6 characters read or scanned from this character source. The user's break character variable (see Open, 11-6 and Input, 11-41) will be set to -1 to indicate a line number break.
"E"	(Lee Erman's very own mode) The result String is terminated on a line number as with "L", but neither the line number nor the TAB following it will appear in subsequent inputs. The line number word, negated, is returned in the user's (integer) BRCHAR variable.
"D"	(Display) If the TTY is a DPY, each line number from any input file will be displayed (along with a page number) on the right-hand side of the screen. This mode really applies to all input operations after the "D" operand appears in any Breakset call. There is no way to turn it off.

11-34. Once a break table is set up, it may be referenced in an INPUT, TTYIN or SCAN call to control the scanning operation.

Example:

11-35. To delimit a "word" a program might wish to input characters until a blank, a TAB, a line feed, a comma, or a semicolon is encountered, ignoring line numbers. Assume also that carriage returns are to be ignored, and that the break character is to be retained in the character source for the next scanning operation:

```
BREAKSET(DELIMS," ,;"&TAB&LF,"I"); Comment break on any of these;
BREAKSET(DELIMS,'15,"O"); Comment ignore carriage return;
BREAKSET(DELIMS,NULL,"N"); Comment ignore line numbers;
BREAKSET(DELIMS,NULL,"R"); Comment save break char for next time;
```

Setbreak

Form:

11-36,

```
SETBREAK ( TABLE , "BREAK_CHARS" , "OMIT_CHARS" , "MODES" )
```

Function:

11-37. SETBREAK is logically equivalent to the SAIL statement:

```
BEGIN "SETBREAK"
 INTEGER I;

 IF LENGTH(OMIT_CHARS) > Ø THEN
        BREAKSET(TABLE,OMIT_CHARS,"O");

 FOR I←1 STEP 1 UNTIL LENGTH(MODES) DO
        BREAKSET(TABLE,BREAK_CHARS,MODES[I FOR 1])

END "SETBREAK"
```

STDBRK

Form:

11-38. STDBRK (CHANNEL);

Function:

11-39. Eighteen breakset tables have been selected as representative of the more common input scanning operations. The function STDBRK initializes the breakset tables by opening the file BKTBL.BKT[1,3] on CHANNEL and reading in these tables. The user may then reset those tables which he does not like to something he does like.

11-40. The eighteen tables are described here by giving the SETBREAKs which would be required for the user to initialize them:

```
DELIMS ← '15 & '12 & '40 & '11 & '14;
 Comment carriage return, line feed, space, tab, form feed;
LETTS ← "ABC ... Zabc ... z_";
DIGS ← "0123456789";
SAILID ← LETTS&DIGS;

SETBREAK ( 1, '12, '15, "INS" );
SETBREAK ( 2, '12, NULL, "INA" );
SETBREAK ( 3, DELIMS, NULL, "XNR" );
SETBREAK ( 4, SAILID, NULL, "INS" );
SETBREAK ( 5, SAILID, NULL, "INR" );
SETBREAK ( 6, LETTS, NULL, "XNR" );
SETBREAK ( 7, DIGS, NULL, "XNR" );
SETBREAK ( 8, DIGS, NULL, "INS" );
SETBREAK ( 9, DIGS, NULL, "INR" );
SETBREAK (10, DIGS&"+-@.", NULL, "XNR" );
SETBREAK (11, DIGS&"+-@.", NULL, "INS" );
SETBREAK (12, DIGS&"+-@.", NULL, "INR" );
SETBREAK (13-18, NULL, NULL, NULL );
```

Input

Form:

11-41. "RESULT" ← INPUT(CHANNEL, BREAK_TABLE);

Function:

11-42. A string of characters is obtained for the file open on CHANNEL, and is returned as the result. The INPUT operation is controlled by BREAK_TABLE (see Breakset, 11-23) and the reference variables BRCHAR, EOF, and COUNT which are provided by the user in the OPEN function for this channel (see Open, 11-6). Input may be terminated in several ways. The exact reason for termination can be obtained by examining BRCHAR and EOF:

EOF	BRCHAR	
-1	Ø	End of file occurred while reading. The result is a String containing all non-omitted characters which remained in the file when INPUT was called.
Ø	Ø	No break characters were encountered. The result is a String of length equal to the current COUNT specifications for the CHANNEL (see Open, 11-6).
Ø	<Ø	A line number was encountered and the break table specified that someone wanted to know. The result String contains all characters up to the line number. If mode "L" was specified in the Breakset setting up this table, bit 35 is turned off in the line number word so that it will be input next time. -1 is placed in BRCHAR. If mode "E" was specified, the line number will not appear in the next input String, but its negated value, complete with low-order line number bit, will be found in BRCHAR.
Ø	>Ø	A break character was encountered. The break character is stored in BRCHAR (an INTEGER reference variable, see Open, 11-6) as a right-justified 7-bit ASCII value. It may also be tacked on to the end of the result String or saved for next time, depending on the BREAKSET mode (see Breakset, 11-23).

11-43. If break table Ø is specified, the only criteria for termination are end of file or COUNT exhaustion. The routine is somewhat faster operating in this mode.

Scan

Form:

11-44. "RESULT" ← SCAN (@"SOURCE" , BREAK_TABLE , @BRCHAR)

Function:

11-45. SCAN functions identically to INPUT with the following exceptions:

1. The source is not a data file but the String SOURCE, called by reference. The String SOURCE is truncated from the left to produce the same effect as one would obtain if SOURCE were a data file. The disposition of the break character is the same as it is for INPUT.

2. BRCHAR is directly specified as a parameter. INPUT gets its break character variable from a table set up by Open, 11-6.

3. Line number considerations are irrelevant.

Out

Form:

11-46. OUT(CHANNEL,"STRING")

Function:

11-46. STRING is output to the file open on CHANNEL. If the device is a TTY, the string will be typed immediately. Buffered mode text output is employed for this operation. The data mode specified in the OPEN for this channel must be 0 or 1.

Linout

Form:

11-47. LINOUT (CHANNEL , NUMBER);

Function:

11-48. ABS(NUMBER) mod 100,000 is converted to a 5 character ASCII string. These characters are placed in a single word in the output file designated by CHANNEL with the low-order bit (line-number bit) turned on. A tab is inserted after the line number. Mode 0 or 1 must have been specified in the OPEN (Open, 11-6) for the results to be anywhere near satisfactory.

WordIn

Form:

11-49. VALUE ← WORDIN (CHANNEL)

Function:

11-50. The next word from the file open on CHANNEL is returned. A 0 is returned, and END_FILE_FLAG (see Open, 11-6) set, when end of file is encountered. This operation is performed in buffered mode or dump mode, depending on the mode specification in the OPEN.

ArryIn

Form:

11-51. ARRYIN (CHANNEL , @LOC , HOW_MANY);

Function:

11-52. HOW_MANY words are read from the device and file open on CHANNEL, and deposited in memory starting at location LOC. Buffered-mode input is done if MODE (see Open, 11-6) is '10 or '14. Dump-mode input is done if MODE is '16 or '17. Other modes are illegal.

11-53. If an end of file condition occurs before HOW_MANY words are read, the EOF variable (see Open, 11-6) is set to '777777 in its left half. Its right half contains the number of words actually read. EOF will be 0 if the full request is satisfied.

Wordout

Form:

11-54. WORDOUT (CHANNEL , VALUE);

Function:

11-55. VALUE is placed in the output buffer for CHANNEL. An OUTPUT is done when the buffer is full or when a CLOSE or RELEASE is executed for this channel. Dump mode output will be done if dump mode is specified in the OPEN (see Open, 11-6).

Arryout

Form:

11-56. ARRYOUT (CHANNEL , @LOC , HOW_MANY);

Function:

11-57. HOW_MANY words are written from memory, starting at location LOC, onto the device and file open on channel CHANNEL. The valid modes are again '10, '14, '16, and '17. The EOF variable is, of course, unaffected.

Mtape

Form:

11-58. MTAPE (CHANNEL , MODE);

Function:

11-59, MTAPE is ignored unless the device associated with CHANNEL is a magnetic tape drive. It performs tape actions as follows:

MODE	FUNCTION
"A"	Advance past one tape mark (or file)
"B"	Backspace past one tape mark
"F"	Advance one record
"R"	Backspace one record
"W"	Rewind tape
"E"	Write tape mark
"U"	Rewind and unload

Useti, Useto

Form:

11-60,
```
USETI ( CHANNEL , VALUE );
USETO (CHANNEL , VALUE );
```

Function:

11-61, The corresponding system function is carried out (see [Moorer]).

Realin, Intin

Form:

11-62,
```
VALUE ← REALIN ( CHANNEL );
VALUE ← INTIN ( CHANNEL );
```

Function:

11-63. Number input may be obtained using the functions REALIN or INTIN, depending on whether a Real number or an Integer is required. Both functions use the same free field scanner, and take as argument a channel number.

11-64. Free field scanning works as follows: characters are scanned one at a time from the input channel. Nulls, line numbers,and carriage returns are ignored. When a digit is scanned it is assumed that this is a number and the following syntax is used:

```
<number>          ::=   <sign><real number>

<real number>     ::=   <decimal number>|<decimal number><exponent>|
                        <exponent>

<decimal number>  ::=   <integer>|<integer>.|<integer>.<integer>|
                        .<integer>

<integer>         ::=   <digit>|<integer><digit>

<exponent>        ::=   @<sign><integer>

<digit>           ::=   0|1|2|3|4|5|6|7|8|9

<sign>            ::=   +|-|<empty>
```

11-65. If the digit is not part of a number an error message will be printed and the program will halt. Typing a carriage return will cause the input function to return zero. On input, leading zeros are ignored. The ten most significant digits are used to form the number. A check for overflow and underflow is made and an error message printed if this occurs. When using INTIN any exponent is removed by scaling the Integer number. Rounding is used in this process. All numbers are accurate to one half of the least siginificant bit.

11-66. After scanning the number the last delimiter is replaced on the input string and is returned as the break character for the channel. If no number is found, a zero is returned, and the break variable is set to -1; If an end of file is sensed this is also returned in the appropiate channel variable. The maximum character count appearing in the OPEN call is ignored.

Realscan, Intscan

Form:

11-67.
```
VALUE ← REALSCAN ( @"NUMBER_STRING" , @BRCHAR ) ; VALUE ← INTSCAN ( @"NUMBER_STRING" , @BRCHAR );
```

Function:

11-68. These functions are identical in function to REALIN and INTIN. Their inputs, however, are obtained from their NUMBER_STRING arguments. These routines replace NUMBER_STRING by a string containing all characters left over after the number has been removed from the front.

Teletype I/O Functions

Form:

11-69.
```
CHAR ← INCHRW;
CHAR ← INCHRS;
"STR" ← INCHWL;
"STR" ← INCHSL ( @FLAG );
"STR" ← INSTR ( BRCHAR );
"STR" ← INSTRL ( BRCHAR );
"STR" ← INSTRS ( @FLAG , BRCHAR );
"STR" ← TTYIN ( TABLE , @BRCHAR );

"STR" ← TTYINL ( TABLE , @BRCHAR );
"STR" ← TTYINS ( TABLE , @BRCHAR );
OUTCHR ( CHAR );
OUTSTR ( "STR" );
CLRBUF;
```

Function:

11-70. Each of the I/O functions uses the TTCALL UUO's to do direct TTY I/O.

INCHRW waits for a character to be typed and returns that character.

INCHRS returns -1 if no characters have been typed; otherwise it is INCHRW.

INCHWL waits for a line, terminated by a carriage-return and line feed (CR-LF) to be typed. It returns as a string all characters up to (not including) the CR. The LF is lost.

INCHSL returns NULL with FLAG = -1 if no lines have been typed. Otherwise it sets FLAG to 0 and performs INCHWL.

INSTR returns as a string all characters up to, but not including, the first instance of BRCHAR. The BRCHAR instance is lost.

INSTRL waits for a line to be typed, then performs INSTR.

INSTRS is INCHSL if no lines are waiting; INSTRL otherwise.

TTYIN uses the break table features described in (BRKS) and Input, 11-41 to return a string and break character. Mode "R" is illegal; line number modes are irrelevant. The input count (see Open, 11-6) is set at 100.

TTYINL waits for a line to be typed, then does TTYIN.

TTYINS sets BRCHAR to -1 and returns NULL if no lines are waiting. Otherwise it is TTYINL.

OUTCHR types its character argument (right-justified in an integer variable).

OUTSTR types its string argument.

CLRBUF flushes the input buffer.

STRING MANIPULATION ROUTINES

Length

Form:

11-71. VALUE ← LENGTH ("STRING");

Function:

11-72. The number of 7-bit characters in STRING is returned. This function is normally compiled into SAIL programs. The function is provided for other programs if they need it.

Equ

Form:

11-73. VALUE ← EQU ("STR1", "STR2");

Function:

11-74. The value of this function is TRUE if STR1 and STR2 are equal in length and have identically the same characters in them (in the same order). The value of EQU is FALSE otherwise.

TYPE CONVERSION ROUTINES

Setformat

Form:

11-75. SETFORMAT (WIDTH , DIGITS) ;

Function:

11-76. This function allows specification of a minimum width for strings created by the functions CVS, CVOS, CVE, CVF, and CVG (see Cvs, 11-80 and following). If this number (WIDTH) is positive, enough blanks will be inserted in front of the resultant string to make the entire results at least WIDTH characters long. The sign, if any, will appear after the blanks. If WIDTH is negative, leading zeroes will be used in place of blanks, The sign, of course, will appear before the zeroes. This parameter is initialized by the system to 0.

11-77. In addition, the DIGITS parameter allows one to specify the number of digits to appear following the decimal point in strings created by CVE, CVF, and CVG. This number is initially 7. See the writeups on the functions Cve, Cvf, Cvg, 11-88 and following for details.

Getformat

Form:

11-78. GETFORMAT (@WIDTH , @DIGITS) ;

Function:

11-79. The WIDTH and DIGIT settings specified in the last SETFORMAT call are returned in the appropriate reference parameters.

Cvs

Form:

11-80. "ASCII_STRING" ← CVS (VALUE);

Function:

11-81. The decimal integer representation of VALUE is produced as an ASCII String with leading zeroes omitted (unless WIDTH has been set by Setformat, 11-75 to some negative value). "-" will be concatenated to the String representing the decimal absolute value of VALUE if VALUE is negative.

Cvos

Form:

11-82. "ASCII_STRING" ← CVOS (VALUE);

Function:

11-83. The octal integer representation of VALUE is produced as an ASCII String with leading zeroes omitted (unless WIDTH has been set to some negative value by Setformat, 11-75). No "-" will be used to indicate negative numbers. For instance, -5 will be represented as "777777777773".

Cvis

Form:

11-84. "STRING" ← CVIS (ITEM , @FLAG) ;

Function:

11-85. The print name of ITEM is returned as a string. An Item's print name is the identifier used to declare it. Print names are not provided for Itemvars. FLAG is set to FALSE (Ø) if the appropriate string is found. Otherwise it is set to TRUE (-1), and you should not place great faith in the string result.

Cvsi

Form:

11-86. ITEM ← CVSI ("PNAME" , @FLAG) ;

Function:

11-87. The Item whose identifier is the same as the string argument PNAME is returned and FLAG set to FALSE if such an Item exists. Otherwise, something very random is returned, and FLAG is set to TRUE.

Cve, Cvf, Cvg

Form:

11-88.
"STRING" ← CVE (VALUE); "STRING" ← CVF (VALUE); "STRING" ← CVG (VALUE);

Function:

11-89. Real number output is facilitated by means of one of three functions CVE,CVG, or CVF, corresponding to the E,G, and F formats of FORTRAN IV. Each of these functions takes as argument a real number and returns a string. The format of the string is controlled by another function SETFORMAT (WIDTH,DIGITS) (see Setformat, 11-75) which is used to change WIDTH from zero and DIGITS from 7, their initial values. WIDTH specifies the minimum string length. If WIDTH is positive leading blanks will be inserted and if negative leading zeros will be inserted.

11-90. The following table indicates the strings returned for some typical numbers. _ indicates a space and it is assumed that WIDTH←10 and DIGITS←3.

CVF	CVE	CVG
______.000	__.100@-3_	__.100@-3_
______.001	__.100@-2_	__.100@-2_
______.010	__.100@-1_	__.100@-1_
______.100	__.100____	__.100____
_____1.000	__.100@1__	__1.00____
____10.000	__.100@2__	__10.0____
___100.000	__.100@3__	__100.____
__1000.000	__.100@4__	__.100@4__
_10000.000	__.100@5__	__.100@5__
_100000.000	__.100@6__	__.100@6__
_1000000.000	__.100@7__	__.100@7__
-1000000.000	_-.100@7__	_-.100@7__

11-91. The first character ahead of the number is either a blank or a minus sign. With WIDTH←-10 plus and minus 1 would print as:

CVF	CVE	CVG
_00001.000	_0.100@1__	_01.00____
-00001.000	-0.100@1__	-01.00____

11-92. All numbers are accurate to one unit in the eighth digit. If DIGITS is greater than 8, trailing zeros are included; if less than eight, the number is rounded.

Cvstr

Form:

11-93. "STRING" ← CVSTR (VALUE) ;

Function:

11-94. VALUE is treated as a 5-character left-justified word full of ASCII, the result is a 5-character long String containing these characters. The low order bit of VALUE is ignored.

Cvxstr

Form:

11-95. "STRING" ← CVXSTR (VALUE) ;

Function:

11-96. VALUE is treated as a 6-character left-justified word full of SIXBIT. The result is a 6-character long String containing these characters, converted to ASCII.

Cvd

Form:

11-97. VALUE ← CVD ("ASCII_STRING");

Function:

11-98. ASCII_STRING should be a String of decimal ASCII characters perhaps preceded by plus and/or minus signs. Characters with ASCII values ≤ SPACE ('40) are ignored preceding the number. Any character not a digit will terminate the conversion (with no error indication). The result is the internal (signed) 36-bit binary representation of the number.

Cvo

Form:

11-99. VALUE ← CVO ("ASCII_STRING");

Function:

11-100, This function is the same as CVD except that the input characters are deemed to represent Octal values,

Cvasc

Form:

11-101, VALUE ← CVASC ("STRING");

Function:

11-102, This is the inverse function for CVSTR, Up to five ASCII characters will be fetched from the beginning of STRING and placed left-justified in VALUE, If the String is less than five characters long, the right characters will be padded with null (Ø) characters.

Cvsix

Form:

11-103, VALUE ← CVSIX ("STRING");

Function:

11-104, The inverse for CVXSTR, this function works the same as CVASC except that up to six SIXBIT characters are placed in VALUE, The characters from STRING are converted from ASCII to SIXBIT before depositing them in VALUE,

Cvfil

Form:

11-105, VALUE ← CVFIL ("FILE_SPEC" , @EXTEN , @PPN) ;

Function:

11-106, FILE_SPEC has the same form as a file name specification for LOOKUP or ENTER. The SIXBIT for the file name is returned in VALUE. SIXBIT values for the extension and project-programmer numbers are returned in the respective reference parameters. Any unspecified portions of the FILE_SPEC will result in zero values.

ARRAY MANIPULATION ROUTINES

ArrInfo

Form:

11-107, VALUE ← ARRINFO (ARRAY , PARAMETER);

Function:

11-108,

ARRINFO(ARRAY,-1) returns the number of dimensions for the array. This number is negative for String arrays.

ARRINFO(ARRAY,0) returns the total size of the array in words.

ARRINFO(ARRAY,1) returns the lower bound for the first dimension.

ARRINFO(ARRAY,2) returns the upper bound for the first dimension.

ARRINFO(ARRAY,3) returns the lower bound for the second dimension.

ArrBlt

Form:

11-109, ARRBLT (@LOC1 , @LOC2 , NUM);

Function:

11-110. NUM words are transferred from consecutive locations starting at LOC2 to consecutive locations starting at LOC1.

ARRTRAN

Form:

11-111. ARRTRAN (ARRAY1, ARRAY2);

Function:

11-112. This function copies information from ARRAY2 to ARRAY1. The transfer starts at the first data word of each array. The minimum of the sizes of ARRAY1 and ARRAY2 is the number of words transferred.

LIBERATION-FROM-SAIL ROUTINES

CODE

Form:

11-113. RESULT ← CODE (INSTR , @ADDR)

Function:

11-114. This function is equivalent to the FAIL statements:

```
        EXTERNAL ,SKIP,          ;DECLARE AS _SKIP_ IN SAIL
        SETOM    ,SKIP,          ;ASSUME SKIP
        MOVE     Ø,INSTR
        ADDI     Ø,@ADDR
        XCT      Ø
        SETZM    ,SKIP,          ;DIDN'T SKIP
        RETURN   (1)
```

In other words, it executes the instruction formed by adding the address of the ADDR variable (passed by reference) to the number INSTR. Before the operation is carried out, AC1 is loaded from a special cell (initially Ø). AC1 is returned as the result, and also stored back into the special cell after the instruction is executed. The global variable _SKIP_ (,SKIP, in DDT or FAIL) is FALSE (Ø) after the call if the executed instruction did not skip; TRUE (currently -1) if it did. Declare this variable as EXTERNAL INTEGER _SKIP_ if you want to use it.

Call

Form:

11-115. RESULT ← CALL (VALUE , "FUNCTION");

Function:

11-116. This function is equivalent to the FAIL statements:

```
        EXTERNAL ,SKIP,
        SETOM    ,SKIP,
        MOVE     1,VALUE
        CALL     1,[SIXBIT /FUNCTION/]
        SETZM    ,SKIP,          ;DID NOT SKIP
        RETURN   (REGISTER 1)
```

The ,SKIP, variable (_SKIP_ in SAIL) is set as described in the previous paragraph (CODE).

Usererr

Form:

11-117. USERERR (VALUE , CODE , "MSG");

Function

11-118. MSG is printed on the teletype. If CODE = 2, VALUE is printed in decimal on the same line. Then on the next line the "LAST SAIL CALL" message is typed which indicates where in the user program the error occurred. A "?" or "→" character is typed and the user may type a standard reply (see ERROR MESSAGES, 13-19). If CODE is 1 or 2, a "→" will be typed and execution will be allowed to continue. If it is Ø, a "?" is typed, and no continuation will be permitted.

SECTION 12

USE OF DEFINE

The SAIL DEFINE feature provides a limited macro capability with parameter substitution. The formal syntax for DEFINE declarations is given in DECLARATIONS, 3-1. Use of these macros is described below.

Defining Macros

12-1. When a macro of the form

```
DEFINE MAC(X,Y) = "FOR Y←1 STEP 1 UNTIL X DO"
```

is seen by the compiler (either at declaration level or statement level), it first associates with the "formal parameters" sequential indices (X=1, Y=2). Then it reads the String constant representing the macro body into String space, substituting for each occurrence of a formal parameter the character '177 followed by the character representing the index of this formal parameter. These special characters will be used to locate the actual parameters when the macro is expanded. The modified macro body is stored under the name of the macro, where it lies dormant until someone mentions it again.

12-2. In what follows, the character ¬ will represent the character ('177) used to identify parameter locations. The number following it will always be the parameter index. The above macro is stored as:

```
FOR ¬2←1 STEP 1 UNTIL ¬1 DO
```

12-3. A macro may be re-defined (at statement level) as many times as desired. The new macro body replaces the old one. Macro names follow block structure, so for a macro with the same name as some other macro to be a redefinition, it must appear at the same block level as that other definition.

String Constants in Macro Bodies

12-4. String constants may be represented in macro bodies, but two quote characters (") must be inserted for each one which would be necessary if the String constant appeared outside the macro body (which after all is itself a String constant, hence the problem).

Using Macros

12-5. When a macro name (ignore for the moment the possibility of parameters) is detected in a file, the body of that macro is retrieved and becomes the input to the SAIL scanner until the String is exhausted; the scanner then returns to the source file for its input. The macro name itself never makes it out of the scanner. If, while a macro body is providing input, another macro name is encountered, the original macro body is put aside until this new macro is exhausted. Nesting may occur to any level; however, it will be necessary to increase the size of the compiler's DEFINE push down stack if nesting gets extremely deep (see the D switch in Switches, 13-13).

Macro Parameters

12-6. If a macro body has been defined with formal parameters, the compiler will look for actual parameters to satisfy them when a macro is expanded. Actual parameters follow the macro name, are surrounded by parentheses and separated by commas.

12-7. A macro parameter is scanned as a String constant. However, for convenience, the following special rules apply to the scanning of a macro actual parameter:

1) All blank characters after the left parenthesis are ignored.

2) If the first non-blank character is not the " character, the parameter String will be terminated by a comma or a right parenthesis, which will not appear in the parameter. If the " character is found after the first one, it is treated as any other text character.

3) If the first non-blank character is the " character, the parameter is scanned using the normal rules for String constants.

Example

12-8.

```
MAC("I","J") is equivalent to MAC(I,J);
MAC("J+3" , "X&""A STRING""")
  is equivalent to MAC(J+3,X&"A STRING");

but   MAC("""A STRING""","PROC(I,J)")
```

may not be abbreviated, because the meaning of the " character would otherwise be ambiguous in the first argument, and the commas and parentheses need protection in the second.

Actual Parameter Expansion

12-9. The actual parameter strings are stored in an ordered list just before the input stream is switched to the macro body. When one of the -number pairs appears, the input stream is switched to the (number)th actual parameter. Other macros (with or without parameters) may appear in these actual parameters without confusing the scanner (sic).

12-10. For an actual parameter to be recognized eventually as a String constant, enough " characters must surround it to allow one to survive on each end when it passes through the scanner for the last time. To be sure, the implementation of this feature is so wondrous that even the authors must resort to trial and error methods when complicated things are done\enod\attempted.

Examples

12-11,

```
DEFINE TTY="1", SRC="2", BRK_ON_LFD="2";
        Comment for constant parameters for which
          it is desirable to include symbolic names,
          this is more efficient than assigning the
          parameter values to variables;

DEFINE TYPE(MSG)= "OUT(TTY,MSG)";
        Comment note inclusion of TTY macro in the
          body of the TYPE macro;

DEFINE TYPEC(MSG)="OUT(TTY,""MSG"")";
        Comment argument always to be made into
          a String constant;

DEFINE DEBUGGING = "TRUE", INP1(VBL,WHERE)=
 "BEGIN
   VBL←INPUT(SRC,BREAK_ON_LFD);
   IF DEBUGGING THEN
        TYPE(""""""INPUT TO VBL AT WHERE IS""""&VBL"");
  END"; Comment (probably);
```

Using these definitions,

```
        INP1(STR,INITIAL READ) expands to:

        BEGIN
         STR←INPUT(2,2);
         IF TRUE THEN
                OUT(1,"INPUT TO STR AT INITIAL READ IS "&STR);
        END;
```

SECTION 13

COMPILER OPERATION

COMMAND FORMAT

Syntax

13-1.

```
<command_line>          ::= <binary_name> <listing_name> ←
                              <source_list>
                        ::= <file_spec> @
                        ::= <file_spec> EXC

<binary_name>           ::= <file_spec>
                        ::= <empty>

<listing_name>          ::= , <file_spec>
                        ::= <empty>

<source_list>           ::= <file_spec>
                        ::= <source_list> , <file_spec>

<file_spec>             ::= <file_name> <file_ext> <proj_prog>
                        ::= <device_name> <file_spec> <switches>
                        ::= <device_name> <switches>

<file_name>             ::= <legal_sixbit_id>

<file_ext>              ::= . <legal_sixbit_id>
                        ::= <empty>

<proj_prog>             ::= [ <legal_sixbit_id> , <legal_sixbit_id> ]
                        ::= <empty>

<device_name>           ::= <legal_sixbit_id>

<switches>              ::= ( <unslashed_switch_list> )
                        ::= <slashed_switch_list>
                        ::= <empty>

<unslashed_switch_list> ::= <switch_spec>
                        ::= <unslashed_switch_list> <switch_spec>

<slashed_switch_list>   ::= / <switch_spec>
                        ::= <slashed_switch_list> / <switch_spec>
```

```
<switch_spec>            ::= <valid_switch_name>
                         ::= <signed_integer> <valid_switch_name>

<valid_switch_name>      ::= D
                         ::= L
                         ::= M
                         ::= P
                         ::= Q
                         ::= R
                         ::= S
```

Semantics

13-2. All this is by way of saying that SAIL accepts commands in essentially the same format accepted by DEC processors such as MACRO and FORTRAN. The binary file name is the name of the output device and file on which the ready to load object program will be written. The listing file, if included, will contain a copy of the source files with a header at the top of each page and an octal program counter entry at the head of each line (see Listing Features, 13-13). The listing file name is often omitted (no listing created). The source file list specifies a set of user-prepared files which, when concatenated, form a valid SAIL program (one outer block).

13-3. legal_sixbit_identifier is a name which is acceptable to the time sharing system as a valid file name, device name, extension, etc, when its first six (device, file) or three (extension, project-programmer number) are converted from ASCII to SIXBIT. For more information about file and device names, see [Moorer].

13-4. If file_ext is omitted from the binary_name, the extension for the output file will be .REL. The default extension for the listing file is .LST. SAIL will first try to find source files under the names given. If this fails, and the extension is omitted, the same file with a .SAI extension will be tried.

13-5. If device_name is omitted, DSK: is assumed. If proj_prog is omitted, the project-programmer number for the job is assumed.

13-6. Switches are parameters which affect the operation of the compiler. A list of switches may appear after any file name. The parameters specified are changed immediately after the file name associated with them is processed. The meanings of the switches are given below.

13-7. The binary, listing and (first) source file names are processed before compilation -- subsequent source names (and their switches) are processed whenever an end-of-file condition is detected in the current source file. Source files which appear after the one containing the outer block's END delimiter are ignored.

13-8. Each new line in the command file (or entered from the teletype) specifies a separate program compilation. Any number of programs can be compiled by the same SAIL core image.

13-9. The file_spec@ command causes the compiler to open the specified file as the command file. Subsequent commands will come from this file. If any of these commands is file_spec@, another switch will occur.

13-10. The file_spec! command will cause the specified file to be run as the next processor. This program will be started in "RPG mode". That is, it will look on the disk for its commmands if its standard command file is there -- otherwise, command control will revert to the TTY. The default option for this file name is .DMP. The default project-programmer number is [1,3]. The default device, of course, is DSK.

13-11. For information about logging in, running jobs, and so on, see [Moorer].

Rpg Mode

13-12. The COMPILE, DEBUG, LOAD, and EXECUTE set of system commands may be used to compile and run SAIL programs. See [Moorer] for details. A typical command String to the system (which will prepare commands of the form described above and pass them to SAIL (after starting it) might be:

```
DEBUG /SAIL RECOG(-2L5MRR)=BEG+PROCS+RECOG/LIST,CMDSCN[1,DCS]
```

This command will cause the following commands to be placed in a file on your area by the name of QQSAIL.RPG:

```
RECOG,REL,RECOG,LST(-2L5MRR)←BEG,PROCS,RECOG
CMDSCN,REL←CMDSCN[1,DCS]
LOADER!
```

The /SAIL entry may be omitted if all files have a .SAI extension. The loader will load the files with DDT or RAID and then start the specified debugging program.

Switches

13-13. The following table describes the SAIL parameter switches. If the switch letter is preceded in the table by the D character, a decimal number is expected as an argument, Ø is the default value. The character O indicates that an octal number is expected for this switch, Otherwise the argument is ignored.

ARG	SWITCH	FUNCTION
	D	For every occurrence of this switch in the command line, the amount of space for the push down stack used in expanding macros (see USE OF DEFINE, 12-Ø) is doubled, Use this switch if the compiler indicates to you that this stack has overflowed, This shouldn't happen unless you nest DEFINE calls extremely deeply.
O	L	In compiling a SAIL program, an internal variable called PCNT (for program counter) is incremented (by one) for each word of code generated, This value, initially Ø, represents the address of a word of code in the running program, relative to the load point for this program, The current octal value of PCNT plus the value of another internal variable called LSTOFFSET, is printed at the beginning of each output line in a listing file, For the first program compiled by a given SAIL core image, this value is initially Ø, If the L switch occurs in the command and the value O is non-negative, O replaces the current value of LSTOFFSET, If O is -1, the current size of DDT is put into LSTOFFSET, If O is -2, the current size of RAID is used, In "RPG mode" the final value of PCNT is added to LSTOFFSET after each compilation, Thus by deleting all .REL files produced by SAIL, and by compiling all SAIL programs which are to be loaded together with one RPG command which includes the L switch, you can obtain listing files such that each of these octal numbers represents the actual starting core address of the code produced by the line it precedes, At the time of this writing, RPG would not accept minus signs in switches to be sent to processors, Keep trying,
D	M	D is a number from 1 to 6, This parameter puts the compiler in one of several debugging modes, This switch is most useful to compiler fixers, but some of the modes are of general interest, The functions

represented by each of these modes are described in Debugging modes, 13-14 below,

P Each occurrence of this switch doubles the size of the system push down list, It has never been known to overflow,

Q Each occurrence doubles the size of the String push down list, No trouble has been encountered here, either,

R Each occurrence doubles the size of the compiler's parsing and semantic stacks, A long conditional statement of the form (IF ... THEN ... ELSE IF ... THEN ... ELSE IF ...) has been known to cause these stacks to overflow their normally allocated sizes,

D S The size of String space is Set to D words, String space usage is a function of the number of identifiers, especially macros, declared by the user, In the rare case of String space exhaustion, 5000 is a good first number to try,

Debugging modes

13-14, Certain versions of the SAIL compiler have a debugging facility built into the inner loop of the parser, It is willing to display information about the current state of the compilation at strategic times, This routine can be in one of several modes, A debugging mode is initially specified using the M switch described above, It can be changed by the user as the compilation progresses, The modes and their functions are as follows:

1) Just before each code-generator is called, its name is displayed on the TTY along with the top few elements of the parse and semantic stacks, If the TTY is a DPY, one also gets the current input line with an arrow underneath indicating the next element to be scanned, If you do not know what to look for in the stack, don't use this mode, Compilation may be continued by typing the character "P",

2) No information is displayed in this mode, However line breaks and asynchronous breaks (see below) can still occur,

3) Just before each parse production is compared to the parse stack, the name of the production and the other information mentioned above is presented, Proceed by typing "P", Compilation takes forever in this mode,

4) This mode does not cancel any of modes 1, 2,or 3. However, it puts the debugging routines in a mode wherein they will not wait for a user go-ahead before proceeding from the displays described for these modes, Line and asynchronous breaks are still enabled in this mode, and may be used to regain control of things.

5) This mode has no very useful application if the TTY is really a TTY, However if it is secretly a DPY, the current input line is continuously presented along with an arrow showing the compiler's progress through it, No user go-ahead is necessary after each presentation, All other modes are cancelled, Line and asynchronous breaks are enabled.

6) This is the default mode, No information is displayed, The debugging routines are completely detached from the compilation loop, Line and asynchronous breaks are disabled. The only way to get any of the information described above is to start over.

13-15, If you have the compiler in a position where it is willing to listen to a "P" to continue, you may also type some other things. The most interesting one is the "L" command, Typing "L", followed by a space, followed by a page number (decimal), followed by a space, followed by a 5 character line number, followed by yet another space, causes the compiler to remember this page and line number, and to stop with a Line Break message and the information described above just after the specified line has been read, At this point you may change modes (see below) or not, as you prefer, and type "P" to continue, This command is really not too useful unless you are a compiler fixer.

13-16, To change modes while compiling, type any number of parameter-M pairs to the debugging interpreter before typing "P" to go on.

13-17, To get the compiler's attention when it is operating in one of the modes 2, 4, or 5, simply type a carriage return, Very shortly the compiler will display an Asynchronous Break message, the print line, and some stack elements, Then you may change modes, set a line break, or simply proceed, This is often useful simply to convince yourself that your program is still being compiled If you are running in mode 2, If you are operating in mode 6, the compiler will not listen to your plea, Start the compiler in mode 2 If you want this feature, but be warned that things will slow down considerably (10%?).

13-18. Here is an example of a compile string which a user who just has to try every bell and whistle available to him might type to compile a file named NULL:

```
COMPILE /LIST /SAIL NULL(RR-2L1M4M5000S)
```

The switch information contained in parentheses will be sent unchanged to SAIL. Note the convention which allows one set of parentheses enclosing a myriad of switches to replace a "/" character inserted before each one. This string tells the compiler to compile NULL using parse and semantic stacks four times larger than usual (RR). A listing file is to be made which assumes that RAID will be loaded and NULL will be loaded right after RAID (-2L). The user wants to see the stack and input line just before every code generating routine is called (1M), but he does not want the compiler to stop after each display (4M). His program is big enough to need 5000 words of String space (5000S).

ERROR MESSAGES

13-19. If the compiler detects a syntax or semantic error while compiling a program it will provide the user with the following information:

1) The error message. These are English phrases or sentences which attempt to diagnose the problem. If a message is vague it is because no specific test for the error has been made and a catchall routine detected it. If the message begins with the word "DRYROT" it means that there is a bug in the compiler which some strangeness in your program was able to tickle. See a system programmer about this.

2) The current input line. Page and line number, along with the text of the line being scanned, are typed. If the console device is a TTY, a line feed will occur at the point in the line just following the last program element scanned. If the device is a DPY, the line will be displayed with a vertical arrow below the scan position. The absence of a position indicator means that a macro (DEFINE) body is being expanded.

3) "CALLED FROM xxxxx". This is a message of value to compiler debuggers only.

4) A question mark or right-arrow (→).

13-20, Respond to the question mark in any of the following ways:

CR Try to continue compilation, A message will be printed and the sequence reentered if recovery is impossible (if a "?" was typed instead of a "→").

LF Continue and don't stop from now on. The program will not stop if it can help it. Messages will fly by (at an unreadable rate on DPYs) until the compilation is complete or an error occurs from which no recovery is possible, In the latter case the question sequence is reentered.

S Restart, Sometimes useful if you are debugging the compiler (or if you were compiling the wrong file). The program is restarted, accepting compilation commands from the TTY.

X Exit, All files are closed in their current state, The program exits to the system.

L Look at stack, This enters a part of the debugging routine (see Debugging modes, 13-14 above) to allow examination of the parse and semantic stacks, The compiler will lead you by the hand through these procedures.

E Edit, This command must be followed by a carriage return, or a space, a filename (in standard format, assumes DSK) and a carriage return, If the filename is missing, the SOS editor (see [Savitzky]) is started, given instructions to edit the current source file and to move the editing pointer to the current page and line number, If a file name is present, that file is edited starting at the beginning,

D Enter DDT or RAID if one is loaded, Otherwise, type "NO DDT" and re-question,

13-21, Any other character will cause the error routines to spew forth a summary of this table and re-enter the question sequence.

STORAGE ALLOCATION

13-22. The compiler dynamically allocates working storage for its push down lists, symbol tables, string spaces, etc. It normally runs with a standard allocation adequate for most programs. Switch settings given above may be used to change these allocations. If desired, these allocations may also be changed by typing ↑C, followed by REE (reenter). The compiler will ask you if you want to allocate. Type Y to allocate, N to use the standard allocation, and any other character to use the standard allocations and print out what they are. All entries will be prompted. Numbers should be decimal. Typing alt-mode instead of CR will cause standard allocation to be used for the remaining values. The compiler will then start, awaiting command input from the teletype.

SECTION 14

PROGRAM OPERATION

LOADING AND STARTING SAIL PROGRAMS

Loading

14-1. Load the main program, any separately compiled procedure files (see Separately Compiled Procedures, 15-7), any assembly language (see PROCEDURE IMPLEMENTATION, 16-46) or Fortran procedures, and DDT or RAID if desired. This is all automatic if you use the LOAD or DEBUG or EXECUTE system commands (see [Moorer]). Any of the SAIL execution time routines requested by your program will be searched out and loaded automatically from LIBSAI,REL[1,3].

Space Allocation, Normal Operation

14-2. If you can run with standard space allocation, simply start your program. First the SAIL storage areas will be initialized. All strings (except constants) will be cleared to NULL. All compiled-in arrays will be cleared. Then execution will begin with the first statement in the outer block of your main program. As each block is entered, its arrays will be cleared as they are allocated. Variables are not cleared. The program will exit when it leaves this outer block.

14-3. If more push down stack space (string, system, array) or string space is needed, type REE to the monitor and answer allocation questions as described in STORAGE ALLOCATION, 13-22. You can find out what the standard allocations are by typing a space after the system types ALLOC? at you. Arrays, Leap spaces and I/O buffers are allocated dyncamically, obtaining more storage from the operating system if necessary. See Storage Allocation Routines, 16-5 and following for ways of cooperating with SAIL with respect to storage allocation if you write machine language subroutines.

ERROR MESSAGES

14-4. Error messages have nearly the same format as those from the compiler (ERROR MESSAGES, 13-19). They indicate that

1) an array subscript has overflowed;

2) a case index is out of range;

3) a stack has overflowed while allocating space for a recursive procedure; or

4) one of the execution time routines has detected an error.

14-5. The "CALLED FROM" address identifies, in the first 3 cases, the location in the user program where the error occurred ; the "LAST SAIL CALL AT" address gives the location of the faulty call on the SAIL routine for type 4 messages.

14-6. All the replies to error messages described in ERROR MESSAGES, 13-19 are valid except the "L" option. If no file name is typed with the "E" option, the editor re-opens the last file mentioned in the EDIT system command.

14-7. The function USERERR may be used to activate the SAIL error message mechanism. See Usererr, 11-117 for details.

DEBUGGING

14-8. The code output for SAIL programs is designed to be fairly easy to understand when examined using the DDT debugging language or Stanford's display oriented RAID program. A knowledge of the debugger you have chosen is required before this section will be comprehensible.

Symbols

14-9. Only those symbols which have been declared INTERNAL (see Separately Compiled Procedures, 15-7) and those declared in the currently open "program" are available at a given time. The name of a SAIL program as far as DDT or RAID (henceforth DDRAID) is concerned is the name of the outer block of that program. If no name is given for this block, the name M. will be the default.

14-10. Only the first six non-blank characters of a block name or identifier will be used in forming a DDRAID symbol. If two identifiers in the same block have the same first six characters the program using them will not get confused, but the user might when trying to locate these identifiers.

14-11. To obtain symbols for the execution time routines, load RUNTIM,REL[1,3] with your other files. The routines will be loaded from this file, which includes symbols, instead of from the LIBSAI library, which does not. Your program will be several thousand words longer when this file is used.

Blocks

14-12. All block names and identifiers used as variables, procedures or labels in a given (main or separate procedure) program are available for typout when that program is "open" (NAME$; has been typed). To refer to a symbol, type BLOCK_NAME&SYMBOL/ (; for RAID). The block name may be omitted if you have "opened" the block with BLOCK_NAME$&. The symbol table is block-structured only to the extent that block names have appeared in the source program. For instance, in the program

```
BEGIN "NAME1"
  INTEGER I,J;
  ...
  BEGIN
    INTEGER I,K;
    ...
  END;
  ...
END "NAME1"
```

the symbols J, K, and both symbols I are considered by DDRAID to belong in the same block. Therefore confusion can result with respect to I. This approach was taken to avoid the necessity of generating meaningless block names for DDRAID when none were given in the source program. A compound statement will be considered by DDRAID to be a block if it has a name.

Sail-Generated Symbols

14-13. Some extra symbols are generated by SAIL and will show up when you are using DDRAID. They are:

ACS — The accumulators P (system push down list pointer), SP (string push down pointer), and TEMP (commonly used temporary) are given symbolic names. Currently P='17, SP='16, TEMP='14.

OPS — The op codes for the UUOs ERR., ERROR., FIX, FLOAT, PDLOV, and ARERR (subscript overflow UUO) are included to make these easy to detect in the code.

ARRAYS — For each array declared in the outer block (built-in arrays), the fixed address of its first element is given a symbolic name. This name is constructed from the characters of the array name (up to the first 5) followed by a period. For instance, the first element of array CHT is CHT.; the first element of PDQARR is PDQAR.; The last semicolon was really a period. This dotted symbol points to the second word of the first descriptor for String Arrays (see STRINGS, 16-14, ARRAY IMPLEMENTATION, 16-33).

BLOCKS — The first word of the first executable statement of every block or compound statement which has been given a name is given a label created in the same way as those for arrays above. This label cannot be gone to in the source program. It causes no program inefficiency. This label points at the first word of the compound tail -- not the first word of code generated for the block (skips any procedure or array declaration code).

START — The first word of code generated for any given program is given the name "S.".

Warnings

Hanging Store

14-14. Quite often an assignment statement results only in the loading of a PDP-10 accumulator. This AC will not be stored into the core location identified with the name of the variable until it is necessary. Confusion can result if you set a breakpoint somewhere, then examine the core variables of interest without checking the immediately surrounding code to be sure none of the interesting variables are still in ACs.

Long Names

14-15. Since only the first 6 characters of an identifier are available, it is wise to declare symbols which will be examined by DDRAID in such a way that these six characters will uniquely identify them.

SECTION 15

PROGRAM STRUCTURE

THE SAIL CORE IMAGE (REQUIRED)

15-1. The following things must be present in a core image containing SAIL-compiled files:

Main Program

15-2. A SAIL "main program", or an assembly language program which looks an awful lot like a SAIL main program, must be present if any SAIL-compiled files are. A SAIL source program which has no entry-specification as its first element satisfies this requirement. The first statement executed after storage allocation is complete will be its first statement. There should be no more than one main program per core image.

15-3. The salient characteristics of a main program are:

1) Its .REL file has a starting address block (the loader will tell the time sharing system to start the program at this address).

2) Its first task is to determine whether the program was started in RPG mode. If so, the global variable RPGSW is set to TRUE; otherwise FALSE.

3) Its next task is to call the storage allocator with JSR SAILOR.

4) It should then proceed with the main control of the program.

4) It should execute a POPJ 17,0 when it is all done.

5) It may not execute any UUOs except SAIL UUOs (nor alter permanently the UUO locations 40 and 41) without great caution.

Storage Allocation, Basic Utilities

15-4. There is a set of routines which must always be loaded to establish the operating environment for SAIL programs. These routines allocate storage, set up push down pointers, and initialize some of SAIL's internal tables. Other routines included in this package are a String garbage collector (see STRINGS, 16-14) and several basic routines which many others call upon.

15-5. These programs will be loaded automatically from LIBSAI,REL if the JSR SAILOR instruction, where SAILOR is an external request, is present in the main program (this is automatic for SAIL-compiled main programs).

Other Execution-Time Routines

15-6. All I/O, String-handling, etc. is done by routines which understand about SAIL. Programs requiring these services should probably use these routines. SAIL-compiled files automatically request these blessed routines from LIBSAI,REL.

OPTIONAL ADDITIONS

Separately Compiled Procedures

15-7. When a program becomes extremely large it becomes useful to break the program up into several files which can be compiled separately. This can be done in SAIL by preparing one file as a main program, and one or more other files as programs each of which contains one or more procedures to be called by the main program. Such a file must have the following characteristics:

1) An entry specification (see Entry Specifications, 2-1) must be the first item in the program (preceding even the BEGIN for its outer block). The list of identifiers will be used to form an Entry Block for the loader. Therefore the file may be placed in a user library if desired. The format of libraries is described in [Welher]. The identifier(s) appearing in the entry list may be any valid identifiers, but usually they will be the names of the procedures contained in the file. No starting address will be issued for a program containing an Entry Specification. No checking is done to see if entry identifiers are ever really declared in the body of the program.

2) Since no starting address is present for this file, entry to code within it may only be to the procedures it contains; the statements in the outer block, if any, can never be executed. All procedures to be called from the main program (or procedures in other files) must be qualified with the INTERNAL attribute when they are declared. External procedure declarations with headings identical to those of the actual declarations must appear in all those programs which call these procedures.

3) These internal procedures must be uniquely identifiable by the first six characters of their identifiers. In general, any two internal procedure names (or any other internal variables in the same core image) with the same first six characters will cause incorrect linkages when the programs are loaded.

4) Any variables (simple or array) which appear in the outer block of a Separately Compiled Procedure program will be global to the procedures in this program, but not available to the main program (unless they are connected by Internal/External declarations -- see below). Arithmetic arrays in these outer blocks will always be zero when the program is first loaded, but will never be cleared as others are (see Space Allocation, Normal Operation, 14-2) -- String arrays are always cleared.

5) Any variable, procedure or label may contain the attribute INTERNAL or EXTERNAL in its declaration (ITEMS may not).

15-8. The INTERNAL attribute does not affect the storage assignment of the entity it represents, nor does it have any effect on the behavior of the entity (or the scope of its identifier) in the file wherein it appears. However, its address and (the first six characters of) its name are made available to the loader for satisfying External requests.

15-9. No space is ever allocated for an External declaration. Instead, a list of references to each External identifier is made by the compiler. This list is passed to the loader along with the first six characters of the identifier name. When an Internal name matching it is found during loading, its associated address is placed in each of the instructions mentioned on the list. No program inefficiency at all results from External/Internal linkages (belay that -- references to External arrays are sometimes more inefficient).

15-10. The entity finally represented by an External identifier is only accessible within the scope of the External declaration. Transfers to external labels are always allowed, but if things work correctly when this is done it is only by sheer luck that they do.

Fortran Procedures

15-11. For a program written in DEC FORTRAN IV to run in the SAIL environment, the following restrictions must be observed:

1) It must be a SUBROUTINE or FUNCTION, not a main program.

2) It must not execute any FORTRAN I/O calls. The UUO structures of the two languages are not compatable.

3) It must be declared as a Fortran Procedure (see Fortran Procedures, 6-12) in the SAIL program which calls it.

The type bits required in the argument addresses for Fortran arguments are passed correctly to these routines.

The SAIL compiler will not produce a procedure to be called from FORTRAN.

Assembly Language Procedures

15-12. The Implementation section contains the following paragraphs to aid in writing assembly language procedures: User Table, 16-1, STRINGS, 16-14, ARRAY IMPLEMENTATION, 16-33, Storage Allocation Routines, 16-5, and PROCEDURE IMPLEMENTATION, 16-46. In addition, the following rules should be observed:

1) The ENTRY, INTERNAL, and EXTERNAL pseudo-ops should be used to obtain linkages for procedure names and "global" identifiers (remember that only six characters are used for these linkage names.

2) Accumulators P (currently '17) and SP ('16) should be preserved over function calls. P may be used as a push-down pointer for arithmetic values and return addresses. SP is the string stack pointer. String results are returned on this stack. Arithmetic results are returned in AC 1 (see PROCEDURE IMPLEMENTATION, 16-46).

3) The UUO locations 40 and 41 should be preserved.

4) JOBFF must be set by the user to some free buffer area before OUTBUF or INBUF UUOs are executed. JOBFF is periodically set by SAIL to an invalid address.

5) The CORE UUO may be used to increase memory size, but never to decrease it. Never attempt to use directly any of the memory space currently assigned to the job (except that explicitly provided in the routine). Release all memory obtained in this way before returning to SAIL routines. See Storage Allocation Routines, 16-5 for instructions on obtaining core from the SAIL memory allocators (a much safer, and sometimes faster way).

Others

15-13. There are no other known processors which will produce SAIL-compatible programs. In particular, the LISP 1.6 system, by its very nature, contains storage allocation conflicts which are difficult to resolve. If a great need for this kind of compatibility develops it can be provided.

SECTION 16

IMPLEMENTATION INFORMATION

STORAGE LAYOUT

User Table

16-1. All working storage areas for a SAIL-generated program and its execution-time routines are dynamically allocated -- some just once when the program is first started, some as more space is needed.

16-2. The first area allocated is a several hundred word table which contains information about the remaining storage areas and global variables for the execution-time routines. A single internal variable, GOGTAB, will always contain a pointer to this table. The execution-time routines make all accesses to storage through this table or through user-supplied addresses. They would therefore be totally re-entrant if the GOGTAB variable were allowed to vary over several users.

16-3. A FAIL source file containing symbolic indices for the user table, as well as some useful MACROs, OPDEFs, and accumulator definitions is available to provide accessability to this table for assembly language routines. This file may be concatenated to a FAIL program before assembly.

16-4. Most execution-time routines load the address contained in GOGTAB into the accumulator USER (currently '15) in order to index the user table. Thus in what follows, symbolic index XX into this table will be listed as XX(USER).

Storage Allocation Routines

16-5. SAIL makes all requests for storage through the routines CORGET, CORREL, CORINC, and CANINC. These routines are described in the following paragraphs. The AC's THIS and SIZ are currently set to 2 and 3, respectively. All core routines are called with PUSHJ 17,routine.

Corget

16-6. Corget is called with the desired size of a block of storage in register SIZ. It returns the address of the new block in THIS. No other accumulators are altered. Normally the function skips on return. It does not skip if insufficient core is available to grant the request. The address returned is that of the first free data word (DATA below).

16-7. A SAIL core block has the following form:

```
HEAD:   →LAST,,→NEXT    ;when not in use (free list links)
        SIZE            ;END-HEAD+1, negated when block is in use
DATA:   BLOCK   SIZE-3  ;available to user -- sometimes a few more
                        ;   words than requested will be contained
                        ;   in the block
END:    USEBIT,,→HEAD   ;USEBIT is 400000 if block is in use; else 0
```

The first time CORGET is called, GOGTAB is 0. CORGET notices this and performs the following special actions:

1) Prepares to allocate storage just past the program and symbols (left half of JOBSA contains the relevant address).

2) Allocates the user table; puts pointer in GOGTAB.

3) Forms remaining free storage from the end of the user table to contents (JOBREL) [C(JOBREL)] into a single free SAIL block. Puts →HEAD in LO(USER), FRE(USER). Puts C(JOBREL)+1 in TOP(USER).

4) Performs the requested CORGET operation.

16-8. FRE(USER) is the header of a linked free storage list. Blocks are obtained from this list and the list is updated. CORREL releases blocks onto this list. If no currently free block will satisfy a CORGET request, the CORE UUO is executed to get more from the time sharing system.

16-9. Users are free to use the CORGET function if they will be careful of the two header words and the single trailer word associated with each block. Release these blocks as soon as possible to prevent undue checker-boarding of free storage.

Correl

16-10. Correl is called with the address obtained in the corresponding Corget call (the DATA address) in register THIS. The block is returned to the free storage list. If either of the two neighboring blocks is already free, the adjacent free blocks are merged with the one being released to form a bigger one. If the block being released is uppermost in core, and if it occupies more than about 2K, the core size of the program is contracted using the CORE UUO. About 2K of free storage is left in this case. No ACs are altered by CORREL.

Corinc

16-11. Corinc is called with the DATA address of a SAIL block in THIS and a desired increment in SIZ. If there is a free block directly above the THIS block with at least SIZ free words, or if the THIS block occupies the highest addresses of any block in use, the request is granted, the block is extended by SIZ words and the function executes a skip-return. Otherwise no skip occurs and no action is taken. No ACs will be altered.

Caninc

16-12. Caninc performs the same tests as Corinc and skips under the same conditions. It also uses the same calling sequence. If it does not skip, it returns with SIZ altered to show the number of words by which the DATA block can be increased. It is Ø if no increase is possible. This function never affects current core allocation.

16-13. These functions are not available to SAIL programs since core can be obtained by array declarations (which in turn use these functions).

STRINGS

String Descriptors

16-14. A SAIL String has two distinct parts: the descriptor and the text. The descriptor is unique and has the following format:

```
WORD1:  STRINGNO,,LENGTH
WORD2:  BYTP
```

1) STRINGNO. This entry is Ø if the String is a constant (the descriptor will not be altered, and the String text is not in String space, is therefore not subject to garbage collection). Every time a String is created via the concatenation operator, or Input function, or an Integer-String type conversion, it receives a new STRINGNO. Each new String receives a number one greater than the last, starting at 1 when the program is initialized. All strings formed as substrings of a given String have the String of the original (major) string. These numbers aid in increasing String garbage collection efficiency.

2) LENGTH. This number is zero for any null String; otherwise it is the number of text characters.

3) BYTP. If count is Ø, this byte pointer is never checked (it need not even be a valid byte pointer. Otherwise, an ILDB machine instruction pointed at the BYTP word will retrieve the first text character of the String. The text for a String may begin at any point in a word. The characters are stored as LENGTH contiguous characters.

16-15. A SAIL String variable contains the two word descriptor for that variable. The identifier naming it points to WORD1 of that descriptor. If a String is declared INTERNAL, a symbol is formed to reference WORD2 by taking all characters from the original name (up to 5) and concatenating a "," (OUTSTRING's second word would be labeled OUTST,).

16-16. When a String is passed by reference to a procedure, the address of WORD2 is placed in the P-stack (see PROCEDURE IMPLEMENTATION, 16-46). For VALUE Strings both descriptor words are pushed onto the SP stack.

16-17. A String array is a block of 2-word String descriptors. The array descriptor (see ARRAY IMPLEMENTATION, 16-33) points at the second word of the first descriptor in the array.

16-18. Information is generated by the compiler to allow the locations of all non-constant strings to be found for purposes of garbage-collection and initialization (see PROCEDURE IMPLEMENTATION, 16-46). All String variables and arrays are cleared to NULL whenever a SAIL program is started.

String Operations

16-19. The four basic String operations are concatenation (CAT), substrings (SUBSTR), String-Integer (GETCH), and Integer-string (PUTCH). Other functions producing or operating upon strings are described in Execution Routines, 11-1.

Cat

16-20. CAT forms a new String from two other strings (constants or otherwise). The calling sequence is:

```
PUSH    SP,WORD11       ;WORD1, FIRST ARGUMENT
PUSH    SP,WORD12       ;WORD2, FIRST ARGUMENT
PUSH    SP,WORD21       ;ETC.
PUSH    SP,WORD22
PUSHJ   P,CAT
```

The result is found as a new two-word descriptor on top of the SP (currently AC '16) stack. If either argument is the null String, the result is the other argument. If the first argument occupies the space directly preceding the first free character in string space, only the second argument is copied. Otherwise, both arguments are copied (in order) into free space to form the result. A new String number is created for this result. The LENGTH field is the sum of the LENGTHs of the two arguments.

Substr

16-21, SUBSTR returns a descriptor representing a part of its input argument, SUBSTR is really three routines, called as follows:

```
                PUSH  SP,WORD1
                PUSH  SP,WORD2

   SUBST           SUBSR              SUBSI
PUSH P,LASDX    PUSH   P,NUMCHR    PUSH P,FIRSDX
PUSH P,FIRSDX   PUSH   P,FIRSDX

                PUSHJ P,SUBS(T/R/I)
```

LASDX is the number of the last character to be included (starting with 1), FIRSDX is the number of the first character to be included, NUMCHR is the number of characters to be included,

16-22, The result is always a two-word descriptor in the SP stack describing the substring,

SUBST is used for the construct ST[X for Y],
SUBSR is used for ST[X to Y].
SUBSI is used for ST[X to -].

16-23, An error message is printed if the request can not be satisfied, This will result in job abortion,

16-24, The String number of the output is the same as the String number of the input,

Getch

16-25, Call with

```
        PUSH    SP,WORD1
        PUSH    SP,WORD2
        PUSHJ   P,GETCH
```

The first character if the String is returned in AC 1 unless the String is NULL; zero is returned in this case. The SP stack is adjusted to remove the parameter. An error message will be printed if some part of the requested substring does not exist.

Putch

16-26. Call with

```
PUSH    P,VALUE
PUSHJ   P,PUTCH
```

The result is a String descriptor with count of 1 on top of the SP stack. The P stack is adjusted to remove the parameter and return address. The String number is new. The low order 7 bits of VALUE form the single character in the string.

String Space

16-27. The normal or user-specified (see STORAGE ALLOCATION, 13-22) number of words required for strings is used to obtain a single SAIL block (see Storage Allocation Routines, 16-5) when the program is started. The limits of this area are placed in ST(USER) and STTOP(USER). Other parameters are set up as described below.

16-28. String text characters are placed contiguously in thes area as strings are created. When not enough storage remains for a contemplated String, the garbage collector (see String Garbage Collection, 16-30) is called to obtain more (by compacting the current space, if possible). If this fails, the program will restart and request more reasonable allocation.

Parameters Used by String Operations

16-29,

ST(USER)	Bottom (low address) of String space
STTOP(USER)	(Top+1) of String space
TOPBYTE(USER)	Byte pointer such that IDPB TOPBYTE(USER) will store into next character
REMCHR(USER)	Negated number of free characters remaining
TOPSTR(USER)	WORD1 for last created String (doesn't include substring operations), CAT uses this word to decide whether its first argument needs to be moved (see Cat, 16-20),

String Garbage Collection

16-30, The String garbage collector (STRNGC) is called whenever the (estimated or actual) size of a soon-to-be-created String is larger than -REMCHR(USER). By various devious means it finds all active (non-constant) String descriptors, sorting them in ascending address sequence by using the byte pointers, associating all substrings of a given active String (major String) ...ouch, Then it compacts String space by moving the text for all major strings to lower memory locations occupied by text no longer reachable from any descriptor, Finally it updates all String descriptors and the parameters described above, If there is still not enough room, it prints a frustrated message and restarts the program with the allocation sequence normally obtained by typing the REEnter system command (see STORAGE ALLOCATION, 13-22),

String-Oriented Machine Language Routines

16-31, If you must write a routine which operates on strings, please observe the following conventions:

1) See PROCEDURE IMPLEMENTATION, 16-46 for conventions concerning input parameters and value returning.

2) If you merely need to read a String, no particular care is required (don't change the descriptor of a reference String parameter by performing careless ILDBs),

16-32, If you need to create a new String, these are also applicable:

4) Estimate the number of characters if it is not known exactly. This estimate must be an upper bound; an unrealistically large estimate will cause the garbage collector to work more often than necessary. Place the estimate in register A (1).

4) Execute the following code before doing any String-munching:

```
MOVE    USER,GOGTAB       ;ESTABLISH ADDRESSABILITY
ADDM    A,REMCHR(USER)    ;UPDATE REMAINING COUNT
SKIPLE  REMCHR(USER)      ;TEST IMPENDING OVERFLOW
PUSHJ   P,STRNGC          ;COLLECT, WILL NOT RETURN IF
                          ; NEW REMCHR+C(A)>0.
```

5) TOPBYTE(USER) should be your WORD2 result. Save it now.

6) Do repeated IDPBs to TOPBYTE(USER) to store your string. This keeps TOPBYTE accurate. Count characters if your estimate was only an estimate.

7) Create WORD1 of your result. The left half is the left half of TOPSTR(USER) incremented by one. The right half is the length of your new string. This word is not only WORD1 of your result, but also should be placed in TOPSTR(USER).

8) Subtract (estimate - actual length) from REMCHR(USER) to keep it honest. This should make REMCHR if anything more negative.

9) Return String results on the top of the SP stack. If a result is to go in a reference parameter (see PROCEDURE IMPLEMENTATION, 16-46) remember that the address you have is that of the WORD2 (byte pointer) word of the descriptor.

ARRAY IMPLEMENTATION

Form

16-33. Let STRINGAR be 1 (TRUE) if the array in question is a String array, Ø (FALSE) otherwise. Then a SAIL array of n dimensions has the following format:

```
HEAD:   →DATAWD          ;→ MEANS "POINTS AT"
        HEAD-END-1
ARRHED: BASE_WORD        ;SEE BELOW
        LOWER_BD(n)
        UPPER_BD(n)
        MULT(n)
        ...
        LOWER_BD(1)
        UPPER_BD(1)
        MULT(1)
        NUM_DIMS,,TOTAL_SIZE
DATAWD: BLOCK   TOTAL_SIZE
        <sometimes a few extra words>
END:    400000,,→HEAD
```

Explanation

16-34.

HEAD — The first two words of each array, and the last, are control words for the Storage Allocation Routines, 16-5. These words are always present for an array. The array access code does not refer to them.

ARRHED — Each array is preceeded by a block of 3*n+2 control words. The BASE_WORD entry is explained later.

NUM_DIMS — This is the dimensionality of the array. If STRINGAR, this value is negated before storage in the left half.

TOTAL_SIZE — The total number of accessible elements (double if STRINGAR) in the array.

BOUNDS The lower bound and upper bound for each dimension are stored in this table, the left-hand index values occupying the higher addresses (closest to the array data). If they are constants, the compiler will remember them too and try for better code (i.e. immediate operands).

MULT This number, for dimension m, is the product of the total number of elements of dimensions m+1 through n. MULT for the last dimension is always 1.

BASE_WORD This is

DATAWD - the sum of (STRINGAR+1)*LOWER_BD(m)*MULT(m)

for all m from 1 to n. The formula for calculating the address of A[I,J,K] is:

```
address(A[I,J,K]) =
    address(DATAWD) +
      (I-LOWER_BD(1))*MULT(1) +
      (J-LOWER_BD(2))*MULT(2) +
      (K-LOWER_BD(3))
```

This expands to

```
address(A[I,J,K]) =
    address(DATAWD) +
     I*MULT(1) + J*MULT(2) + K
       -(LOWER_BD(1)*MULT(1) + LOWER_BD(2)*MULT(2) + LOWER_BD(3)
```

which is

BASE_WORD + I*MULT(1) + J*MULT(2) + K.

By pre-calculating the effects of the lower bounds, several instructions are saved for each array reference.

Array Allocation

Dynamic Arrays

16-35, When an array is declared in any block other than the outer one, the compiler generates code to call the function ARMAK with parameters describing the array. This routine calls CORGET (see Storage Allocation Routines, 16-5) to obtain enough storage, then sets up the control table and clears the data area to zeroes. The ARRHED address is saved in an array push-down list whose pointer is ARRPDP(USER). The addres of DATAWD+1 is returned for String arrays; the address of DATAWD is returned for all others. The compiler generates code to store this address in the core cell bearing the name of the array variable.

16-36, When all declarations for a block containing array declarations have been processed, the compiler issues a call to ARMRK which marks the array push-down stack (with a -1, as a matter of fact). On block exit (or when a GO TO transfers out of the block), the routine ARREL is called to remove this mark and return all arrays back to the previous mark to the SAIL free storage list.

16-37, The String garbage collector uses the array push-down stack to find dynamic String arrays which need attention.

Built-In Arrays

16-38, Outer-block arrays have constant bounds. The compiler simply emits a Jrst instruction, then compiles the control table into the block head of the object program. It leaves room for the array, then issues the END word. The Jrst instruction then finds its home in some code to clear the array to zeroes.

16-39, The core location bearing the name of the array has the address of DATAWD (DATAWD+1 if STRINGAR) compiled into it. This address is given the dotted name described in DEBUGGING, 14-8.

16-40, For built-in String arrays, a String link block (see PROCEDURE IMPLEMENTATION, 16-46) is issued following the space allocated for the array. The String garbage collector (see String Garbage Collection, 16-30) gains access to this array through this static link.

16-41, It can be seen from all this that all dynamic and built-in arrays are cleared when the blocks in which they are declared are entered, Since the outer block of a separately compiled procedure file (see Separately Compiled Procedures, 15-7) is never entered, its built-in arrays, although available for use, are never cleared, The loader clears them once as it loads,

Array Access Code

16-42, In the worst case (no fixed bounds, bounds checking, not built-in) the statement K←A[I,J] will be compiled as:

```
MOVE    1,A             ;→FIRST DATA WORD
MOVE    2,I             ;FIRST SUBSCRIPT
CAML    2,-4(1)         ;IF <LOWER BOUND OR
CAMLE   2,-3(1)         ; >UPPER BOUND THEN
ARERR   1,[ASCIZ /A/]   ;  ERROR IN INDEX 1
IMUL    2,-2(1)         ;I*MULT(1)
MOVE    3,J             ;CHECK DIMENSION 2
CAML    3,-7(1)
CAMLE   3,-6(1)
ARERR   2,[ASCIZ /A/]
ADD     3,2             ;NO MULT FOR LAST, COLLECT OFFSET
ADD     3,-10(1)        ;  + BASE_WORD
MOVE    4,(3)           ;DATA FROM A[I,J]
MOVEM   4,K
```

16-43, If A is, however, declared in the outer block as SAFE INTEGER ARRAY A[1:10,1:5], the code for A[I,J] is

```
MOVE    1,I
IMULI   1,5             ;I*MULT(1)
ADD     1,J             ;COLLECT OFFSET
MOVE    2,A,-5(1)       ;CONSTANT PART OF ADDRESS COMPILED IN
MOVEM   2,K
```

16-44, A[I,3] would be compiled as

```
        MOVE    1,I
        IMULI   1,5
        MOVE    2,A,-2(1)
        MOVEM   2,K
```

and J←A[2,3] would be

```
        MOVE    3,A,+7
        ...
        MOVEM   3,J
```

16-45, Various configurations of array declarations and accesses result in code which ranges between these degrees of efficiency.

PROCEDURE IMPLEMENTATION

16-46,

Procedure Body

16-47, To describe the main characteristics of SAIL procedures, a set of sample procedures are displayed here along with the code they produce. Some of the entries are discussed in more detail below. The notation [n] is placed in the comment field of the assembly instruction to refer to these discussions:

```
INTEGER PROCEDURE P1(INTEGER I,J; STRING A);
        P1:     AOS     P1PAC   ;[1] INCREMENT PROC ACTIVE COUNTER

BEGIN
 INTEGER Q; STRING A,B;
 INTEGER ARRAY X[0:5];
                PUSH    P,[0]
                PUSH    P,[5]
                PUSH    P,[1]
                PUSHJ   P,ARMAK ;ALLOCATE AND CLEAR
                MOVEM   1,X     ;STORE POINTER
                PUSHJ   P,ARMRK ;END OF ARRAYS FOR BLOCK
```

```
 <code for procedure>

 RETURN(Q);
                MOVE    1,Q       ;[2] RESULT IN 1
                PUSHJ   P,ARREL   ;[3] RELEASE ARRAYS FOR BLOCK
                JRST    P1EXIT    ;EXIT PROCEDURE

 <more code for procedure>


END "P1"
                PUSHJ   P,ARREL ;IF FALLS THROUGH, RELEASE ARRAYS
        P1EXIT: SOS     P1PAC   ;ONE TIME LESS ACTIVE
                SUB     SP,[XWD 2,2]    ;REMOVE STRING PARAMETER
                SUB     P,[XWD 3,3]     ;[4] NON-STRINGS, RETURN ADDR
                JRST    @3(P)           ;RETURN
        Q:      Ø               ;ROOM FOR VARIABLE
        X:      Ø               ;ARRAY POINTER
        TEMPØ7: Ø               ;[5] TEMPORARY STORAGE
        A:      BLOCK   2       ;TWO WORDS FOR EACH STRING
        B:      BLOCK   2
        P1PAC:  Ø               ;[6] PROCEDURE-ACTIVE COUNT
                XWD     2,A     ;STRING COUNT,→FIRST
        LNKWD:  Ø               ;[7] LINK PASSES THROUGH HERE
                LINK    1,LNKWD ;[7] CAUSES LOADER LINKAGE



PROCEDURE P2(INTEGER I,J; STRING A);
BEGIN
 INTEGER ARRAY X[Ø:1Ø];
 ...
 BEGIN
  INTEGER ARRAY Y[Ø:1Ø];
  ...
   RETURN;
                PUSHJ   P,ARREL         ;RELEASE ARRAYS FOR ALL
                PUSHJ   P,ARREL         ; BLOCKS IN PROCEDURE
                JRST    P2EXIT

  ...
 END;
END"P2";


STRING PROCEDURE P3(STRING A,B);
BEGIN STRING C;
```

```
 ...
 RETURN(C);
                SUB     SP,[XWD 4,4]    ;REMOVE PARAMS
                PUSH    SP,C
                PUSH    SP,C+1          ;RETURN STRING RESULT
                JRST    P3EXIT

 RETURN(B);
                SUB     SP,[XWD 4,4]
                PUSH    SP,3(SP)        ;FIRST WORD OF B
                PUSH    SP,3(SP)        ;SECOND WORD OF B
                JRST    P3EXIT          ;GO RETURN

 RETURN(C&"STR"); COMMENT ASSUME CAT ALREADY DONE;
                SUB     SP,[XWD 6,6]    ;REMOVE PARAMS, TEMP RESLT
                PUSH    SP,5(SP)        ;TEMP RESLT
                PUSH    SP,5(SP)        ;2D WORD
                JRST    P3EXIT

 ...
END "P3";
        P3EXIT: SOS     P3PAC
                SUB     SP,[XWD 4,4] ;NOT THIS TIME, BUT WOULD
                PUSH    SP,[Ø]       ;BE INCLUDED IF NO RETURNS
                PUSH    SP,[Ø]       ;DONE ABOVE (RETURN NULL STRING)


RECURSIVE INTEGER PROCEDURE P4(STRING A,B; INTEGER I,J);
        P4TEXT: AOS     P4PAC

BEGIN
 STRING C,D; INTEGER K,L;
 ...
END "P4";
        P4EXIT: SOS     P4PAC
                SUB     SP,[XWD =8,=8] ;[8]TAKE OFF LOCALS,PARAMS
                HRRI    TEMP,C          ;[8]
                HRLI    TEMP,5(SP)      ;[8]
                BLT     TEMP,D+1        ;[8] RESTORE LOCAL STRINGS

                SUB     P,[XWD 6,6]     ;[8] SAME FOR P-SIDE
                HRRI    TEMP,K          ; (ALSO RETURN ADDR REMOVED)
                HRLI    TEMP,4(P)
                BLT     TEMP,TEMPØ3     ;MUST EVEN SAVE TEMPS
                JRST    @3(P)           ;RETURN

        P4:     ADD     P,[XWD 3,3]     ;LEAVE ROOM FOR LOCALS
                SKIPL   P               ;CHECK PUSH-DOWN OVERFLOW
                PDLOV   P,              ;[9]UUO TO SIMULATE PDL OV
```

```
                HRRI    TEMP,-2(P)      ;[9]SAVE LOCALS
                HRLI    TEMP,TEMPØ3     ; AND TEMPS
                BLT     TEMP,(P)
                <similarly for SP (string stack)>
                JRST    P4TEXT          ;GO DO PROCEDURE
                <variables and such>


RECURSIVE STRING PROCEDURE P5(STRING A,B);
BEGIN
 STRING C,D;
 ...
 RETURN(C);
                PUSHJ   P,P5POP ;[1Ø]REMOVE STRING LOCALS,PARAMS
                PUSH    SP,C    ;STRING RESULT
                PUSH    SP,C+1
                PUSHJ   P,ARREL ;ENOUGH TIMES IF ANY ARRAYS
                JRST    P5EXIT

 RETURN(B);
                PUSHJ   P,P5POP
                PUSH    SP,3(SP)
                PUSH    SP,3(SP)        ;RETURN PARAMETER
                JRST    P5EXIT

 RETURN(A&"STR");
                POP     SP,1    ;[11]ASSUME CAT ALREADY DONE
                POP     SP,Ø
                PUSHJ   P,P5POP
                PUSH    SP,Ø    ;[11]RETURN VALUE
                PUSH    SP,1
                JRST    P5EXIT

 ...
END "P5";
        P5EXIT: SUB     P,[XWD 2,2] ;OR WHATEVER, SEE ABOVE
                ...                  ;RESTORE LOCALS, ADJUST
                JRST    @3(P)       ;RETURN

        P5:     <as above>
                ...

        P5POP:  SUB     SP,[XWD -8,-8]  ;[1Ø] REMOVE STRING LOCALS,PARAMS
                HRRI    TEMP,C
                HRLI    TEMP,5(SP)
                BLT     TEMP,D+1
                POPJ    P,              ;RETURN
```

The main program has the following format:

```
S,:     SKIPA           ;NOT STARTED IN RPG MODE
        SETOM   RPGSW   ;STARTED IN RPG MODE -- RPGSW A GLOBAL
        JSR     SAILOR  ;INIT -- RETURNS BY PUSHJ P,@SAILOR

  Comment • The main program looks like a non-recursive procedure
    from here on, except for built-in arrays  •

        POPJ    P,      ;RETURN TO INIT, WHO EXITS
        <global variables, linkages>
        <non-String constants>
        XWD     Ø,,=8   ;TYPICAL STRING CONSTANT
        POINT   7,,+1
        ASCII   /CONSTANT/
        <more String constants>

        END     S,      ;STARTING ADDRESS FOR MAIN PROGRAM
```

Discussion

16-48,

[1] There is for each procedure a word (PAC for Procedure Active Count) which is incremented on procedure entry and decremented on exit. At one time, the String garbage collector used this word. It may again some time in the future. At present the counter is useful for determining the depth of recursion (from DDT).

[2] Non-String procedures return their results in 1; Fortran returns things in Ø; String results are returned on the SP stack.

[3] An ARREL call is issued for each block (containing arrays) which must be left in order to exit. All arrays for these blocks are released at this time. The same sort of thing happens when a Go To statement leaves one or more blocks.

[4] Since the return address is on the top of the P-stack, with parameters buried beneath, a subtract and an indirect jump replace the POPJ. Procedures always adjust the stack before returning.

[5] String temporaries are kept in the SP stack. Others occassionally occupy core locations. These are grouped with the non-String variables to make saving and restoring easy in recursive procedures.

[6] This is the Procedure Active Count word (see [1]). It is placed in a fixed location with respect to the String-link block (below). The String garbage collector could, if it wished, see this count.

[7] A linked list, with its head in a reserved cell in the user table (see User Table, 16-1) gives the String garbage collector access to all String variables declared for each procedure; and to all built-in String arrays. Each entry on the list contains three words: a PAC counter (currently ignored), a word giving the location and extent of the String descriptors being described, and the pointer (LNKWD) to the next entry. A Ø entry ends the list. The LINK pseudo-op (or the equivalent code issued by SAIL) instructs the loader to create this list. The LINKEND pseudo-op is issued in the SAILOR routine to collect the address of the first list element. This is then transferred to the user table. See [Welher] for details

concerning the LINK block type.

[8] When a recursive procedure is called, all values for variables declared in blocks internal to this procedure are saved in the appropriate stack. These are added "on top of" the parameters and return address for the procedure. At procedure exit the stack pointer is adjusted to point below the first parameter. Then the proper BLT word is set up to restore all these locals from the stack. After the BLT is executed, that stack is ready for procedure exit.

[9] Since SAIL is a one-pass compiler, it does not know how many locals a procedure has until all blocks for that procedure have been processed. Therefore the entry code for recursive procedures is added last, followed by a jump to the procedure text.

[10] When a String procedure returns a value, the String parameters and locals must be removed from the stack before the value (result) can be pushed on. Since the total number of String locals is not yet known, a routine like P5POP is called to remove the unwanted values first. Recursive String procedures must contain Return Statements (see Return Statement, 5-19); otherwise improper code will result.

[11] Once P5POP or its equivalent has been executed, the previous top of stack location is not known; the temp value is therefore removed first and restored after the call.

Procedure Calling Sequences

16-49. Again a case study is presented. A procedure with several internal procedures is presented to demonstrate the ridiculous number of possibilites. Only the relevant code is described. Accumulator numbers in the code below are only examples -- other values are possible. This list is not complete; to describe all cases here would take more space than a copy of the code in SAIL which handles them. Item and Set parameters behave like Integer and Real parameters as far as argument passing is concerned:

```
PROCEDURE SUPER(REFERENCE STRING RPSTR;
                INTEGER PINT; REFERENCE INTEGER RPINT;
                REAL PROCEDURE PPAR;
                STRING PSTR1,PSTR2);
BEGIN
 INTEGER INT1,INT2; STRING STR1,STR2; REAL REL;
 SAFE INTEGER  ARRAY ARR[2:10]; SAFE STRING ARRAY SARR[2:10];
 INTEGER PROCEDURE INTP(INTEGER I,J);...;
 PROCEDURE RINTP(REFERENCE INTEGER I);...;
 PROCEDURE STRP(STRING A,B);...;
 PROCEDURE RSTRP(REFERENCE STRING A);...;
 PROCEDURE PROCP(PROCEDURE PARAM);...;
 PROCEDURE ARRP(STRING ARRAY X);...;


 INT1←PINT↑2 + RPINT↑2 - 3;
                MOVE    1,-3(P)         ;RELATIVE LOC OF PINT
                IMUL    1,1
                MOVE    2,@-2(P)        ;RPINT's ADDRESS IS IN STACK
                IMUL    2,2
                ADD     2,1             ;SUM
                SUBI    2,3             ;RESULT LEFT IN 2

REL←INTP(INT1,PINT);
                PUSH    P,2             ;INT1 STILL IN 2
                PUSH    P,-4(P)         ;[1]ADJUST FOR PREV PUSH
                MOVEM   2,INT2          ;[2]STORE CURRENT ACS BEFORE CALL
                PUSHJ   P,INTP          ;CALL PROCEDURE
                FLOAT   1,1             ;CONVERT TO REAL -- REL IS IN 1

RINTP(INT1);
                PUSH    P,[INT1]        ;ADDRESS OF INT1
                MOVEM   1,REL           ;PREVIOUS RESULT
                PUSHJ   P,RINTP

RINTP(PINT);
                MOVEI   3,-3(P)         ;ADDRESS OF PINT
                PUSH    P,3
                PUSHJ   P,RINTP

RINTP(RPINT);
                PUSH    P,-2(P)         ;PASS ON ADDR OF RPINT
                PUSHJ   P,RINTP

INT2←INTP(INT1,ARR[PINT]);
                PUSH    P,INT1
                MOVE    4,-4(P)         ;PINT
```

```
                MOVE    5,ARR
                ADD     4,-4(5)         ;BASE ADDR OF ARR
                PUSH    P,(4)
                PUSHJ   P,INTP          ;RESULT IN 1

RINTP(ARR[PINT]);
                MOVE    6,PINT
                MOVE    7,ARR
                ADD     6,-4(7)
                PUSH    P,6             ;ADDRESS
                MOVEM   1,INT2
                PUSHJ   P,RINTP

STRP(STR1&"CON",PSTR1);
                PUSH    SP,STR1
                PUSH    SP,STR1+1
                PUSH    SP,CONAD        ;ADDRESS OF DSCRPTR FOR "CON"
                PUSH    SP,CONAD+1
                PUSHJ   P,CAT           ;LEAVE CONCATENATE IN STACK
                PUSH    SP,-4(SP)       ;PUT STR1 ON TOP
                PUSH    SP,-4(SP)
                PUSHJ   P,STRP

RSTRP(STR1);
                PUSH    P,[STR1+1]      ;ALL REF PARAMS TO P-STACK
                PUSHJ   P,RSTRP

RSTRP(RPSTR);
                PUSH    P,-4(P)         ;PASS REFERENCE ALONG;
                PUSHJ   P,RSTRP

RSTRP(PSTR2);
                HRROI   10,(SP)         ;[3]RH→2D WORD OF PSTR2
                PUSH    P,10
                PUSHJ   P,RSTRP

PROCP(RINTP);
                PUSH    P,[RINTP]       ;PARAMETRIC PROCEDURE
                PUSHJ   P,PROCP

ARRP(SARR);
                PUSH    P,SARR          ;THIS IS EFFECTIVELY A REFEREN
                PUSHJ   P,ARRP
```

Discussion

16-50.

Counts are maintained of the current number of actual parameters (during a procedure call) on each stack. These counts must be added to the parameter indices to access parameters of the procedure doing the calling.

[2] Whenever a SAIL procedure is called, all accumulators except $P ('16) and P ('17) are available for its use.

[3] Some String operations require that the left half of pointers to descriptors be negative. Therefore any operation which obtains a String descriptor address does a HRRO or HRROI to accomplish this. In this case it is not necessary, but it won't hurt anything. String reference parameters always point to the second word of the String in question.

SECTION 17

APPENDIX -- USEFUL SUMMARIES

ARITHMETIC TYPE-CONVERSION TABLE

17-1.

OPERATION	ARG1	ARG2	ARG1*	ARG2*	RESULT
+ -	INT	INT	INT	INT	INT*
* ↑ %	REAL	INT	REAL	REAL	REAL
	INT	REAL	REAL	REAL	REAL
	REAL	REAL	REAL	REAL	REAL
LAND LOR	INT	INT	INT	INT	INT
EQV XOR	REAL	INT	REAL	INT	REAL
	INT	REAL	INT	REAL	INT
	REAL	REAL	REAL	REAL	REAL
LSH ROT	INT	INT	INT	INT	INT
	REAL	INT	REAL	INT	REAL
	INT	REAL	INT	INT	INT
	REAL	REAL	REAL	INT	REAL
/	INT	INT	REAL	REAL	REAL
	REAL	INT	REAL	REAL	REAL
	INT	REAL	REAL	REAL	REAL
	REAL	REAL	REAL	REAL	REAL
MOD DIV	INT	INT	INT	INT	INT
	REAL	INT	INT	INT	INT
	INT	REAL	INT	INT	INT
	REAL	REAL	INT	INT	INT

* Unless ARG2 is <Ø for the operator ↑

SAIL RESERVED WORDS

17-2.

ABS AND ANY ARRAY ARRAY_PDL BEGIN BOOLEAN CASE COMMENT COMPLEX COP CVI CVN DATUM DEFINE DELETE DO DONE ELSE END ENTRY EQV ERASE EXTERNAL FALSE FIRST FOR FOREACH FORTRAN FORWARD FROM GLOBAL GO GOTO IF IN INTEGER INTERNAL ISTRIPLE ITEM ITEMVAR LABEL LAND LENGTH LIBRARY LOAD_MODULE LNOT LOP LOR LSH MAKE MOD NEEDNEXT NEXT NEW NEW_ITEMS NOT NULL OF OR OWN PHI PNAMES PRELOAD_WITH PROCEDURE PUT REAL RECURSIVE REFERENCE REMOVE REQUIRE RETURN ROT SAFE SECOND SET STEP STRING STRING_PDL STRING_SPACE SUCH SYSTEM_PDL THAT THEN THIRD TO TRIPLE TRUE UNTIL VALUE WHILE XOR

SAIL PREDECLARED IDENTIFIERS

17-3.

ARRBLT ARRINFO ARRTRAN ARRYIN ARRYOUT BREAKSET CALL CLOSE CLOSIN CLOSOUT CLRBUF CODE CVASC CVD CVE CVF CVFIL CVG CVIS CVO CVOS CVS CVSI CVSIX CVSTR CVXSTR ENTER EQU GETCHAN GETFORMAT INCHRW INCHRL INCHRS INCHSL INCHWL INSTR INSTRL INSTRS INPUT INTIN INTSCAN LENGTH LINOUT LOOKUP MTAPE OPEN OUT OUTCHR OUTSTR REALIN REALSCAN RELEASE RENAME SCAN SETBREAK SETFORMAT STRBRK TTYIN TTYINL TTYINS WORDIN WORDOUT USERERR USETI USETO

CHARACTER-IDENTIFIER EQUIVALENCES

17-4.

CHARACTER	RESERVED WORD
∧	AND
≡	EQV
¬	NOT
∨	OR
●	XOR
∞	INF
∈	IN
|	SUCH THAT

PARAMETERS TO THE OPEN FUNCTION

17-5.

CHANNEL	System Data Channel, Ø-'17
DEVICE	string giving device name
MODE	data mode
INBUFS	number of input buffers
OUTBUFS	number of output buffers
COUNT	text input count (reference)
BRCHAR	break char variable (reference)
EOF	end-of-file flag (reference)

BREAKSET MODES

17-6.

I	(Inclusion) string is set of break chars
X	(eXclusion) string of all non-break chars
O	(Omit) string of characters to be omitted from result
S	(skip) break char appears only in BRCHAR variable
A	(Append) break char is last char of result string
R	(Retain) break char is first char of next string
P	(Pass) line numbers appear in input without warning
N	(No numbers) line numbers and the tabs that follow them are removed.
L	(Line no break) line numbers cause input break. BRCHAR is negative. Next input gets line no characters.
E	(Erman) line numbers cause input break. Negated line no returned in BRCHAR. Line no removed from input.
D	(Display) after this appears, each line no is listed on the display (if TTY is a DPY) as it is dealt with.

MTAPE COMMANDS

17-7.

MODE	FUNCTION
"A"	Advance past one tape mark (or file)
"B"	Backspace past one tape mark
"F"	Advance one record
"R"	Backspace one record
"W"	Rewind tape
"E"	Write tape mark
"U"	Rewind and unload

COMMAND SWITCHES

17-8.

D	double size of define pushdown stack
numL	listing control -- num>Ø becomes listing starting addr, num=-1 starts listing after current DDT size, num=-2 starts listing after current RAID size.
numM	initial debugging mode set to num
P	double size of system pushdown list
Q	double size of string pushdown list
R	double size of parse pushdown list
numS	set size of string space to num

DEBUGGING MODES

17-9.

1	display before executing each code generation routine
2	don't display, but remain enabled for asynchronous and line breaks
3	display before each production is compared
4	continue from type 1 and 3 modes automatically
5	just display input file as it goes past
6	disable debugging mechanism (started in this mode unless an M switch appears).

VALID RESPONSES TO ERROR MESSAGES

17-10.

CR	(carriage return) try to continue
LF	(line feed) continue automatically -- don't stop for user go-ahead after each message
S	restart
X	exit -- close all files, return to monitor
L	look at stacks -- of interest only to compiler fixers
E	edit. Follow by CR to get file the compiler is working on (or last thing edited, for runtime routines). Follow with <name> CR to edit <name>.
D	go to DDT or RAID

SECTION 18

BIBLIOGRAPHY

18-1.

REFERENCE	DESCRIPTION
Feldman	Feldman, J.A. and Rovner, P.D. An Algol-Based Associative Language, Comm. ACM 12, 8 (Aug. 1969), 439-449.
Moorer	Moorer, J.A. Stanford A-I Project Monitor Manual, Sailons 54 and 55 (Sep. 1969).
Welher	Welher, W.F. Loader Input Format, Sailon 46 (Oct. 1968).
Savitzky	Savitzky, S.R. Son of Stopgap, Sailon 50.1, (Sep. 1969), a revision of Stopgap, Sailon 50, by W.F. Welher.

INDEX