Massachusetts Institute of Technology
Artificial Intelligence Laboratory

28 June 1978

AI MEMO • 482

DIRECTOR GUIDE

Kenneth M. Kahn

Abstract

Director is a programming language designed for dynamic graphics, artificial intelligence, and naive users. It is based upon the actor or object oriented approach to programming and resembles Act 1 and SmallTalk. Director extends MacLisp by adding a small set of primitive actors and the ability to create new ones. Its graphical features include an interface to the TV turtle, pseudo-parallelism, many animation primitives, and a primitive actor for making and recording "movies". For artificial intelligence programming Director provides a pattern-directed data base associated with each actor, an inheritance hierarchy, pseudo-parallelism, and a means of conveniently creating non-standard control structures. For use by relatively naive programmers Director is appropriate because of its stress upon very powerful, yet conceptually simple primitives and its verbose, simple syntax based upon pattern matching. Director code can be turned into optimized Lisp which in turn can be compiled into machine code.

The author of this work is currently supported by an IBM Fellowship. The research described herein is being conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program.

CONTENTS

I. The General Idea

Director is an actor-based extension of MacLisp and is described in AI Working Paper 120. [Kahn 1976] This document is intended to help you use it.

While much of the work described here is intended for a graphics audience (describing much simpler and more intuitive ways of thinking about graphics and animation), much of it should be of interest to anyone interested in actors. Graphics is an ideal domain to test out different styles of message passing in a way that is concrete. A face telling its mouth to smile is pedagogically a much better example than a number being told to multiply by the result of factorial being sent the result of that number being asked to subtract one.

The language is also usable as an AI language. Each actor has rather sophisticated abilities including inheritance, a relational database with demons, and a pseudo-parallel control structure. The language is currently grafted upon MacLisp, so that in addition to lists, atoms, numbers, lambda expressions and other Lisp entities, Director provides actors and a few primitives for manipulating them. This implementation strategy was one of necessity and many of Director's deficiencies would disappear were it built upon an Actor language such as Act 1.

II. An Introductory Example

To get a general impression of what Director is all about try the following the next time you are logged into AI on a TV (if you want to try it without graphics read the "Getting Started" section). Start up the system by typing

```
:direct <carriage return>
|Welcome to Director Version # 39| ;; at which point Director is ready for instructions

(ask poly make pent) ;; create a polygon named pent
(ask pent show) ;; a default polygon (a hexagon) should appear
(ask pent set your angle to 72) ;; it should now look like a pentagon

(ask poly make star) ;; make another poly named star
(ask star set your angle to 144) ;; set its angle to 144
(ask star show) ;; finally ask it to show
(ask star forward 200) ;; ask star to go forward 200 steps
(ask star grow 250) ;; ask the star to become 250 units larger
(ask star print) ;; if you are curious about what star knows try printing it
```

At this point, you might want to play around. If the typing is too much for you there are abbreviated versions of nearly all the messages. For example typing,

```
(ask star &sypt (200 -100)) ;; is the same as typing
(ask star set your position to (200 -100))
; which means go to the point 200 units over and 100 down from the center
```

A list of all the abbreviations is in the index at the end of this document.

To make a little movie type the following

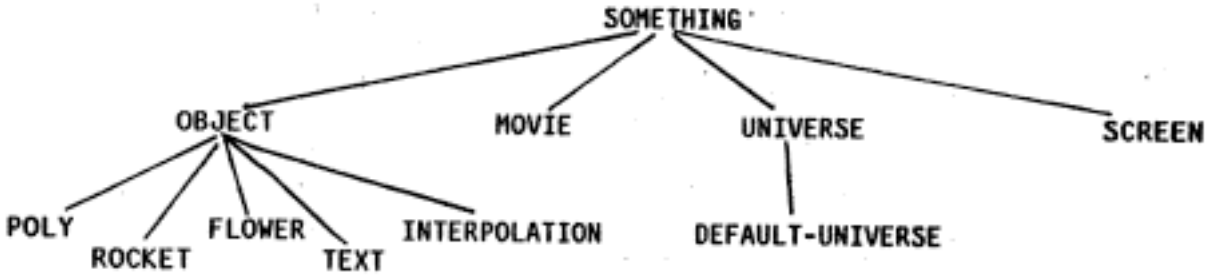
```
(ask star plan next gradually shrink 300)
;; start slowly shrinking beginning with the next clock tick
(ask star plan next gradually left 90) ;; plan to also slowly turn left
(ask pent plan next gradually grow 300) ;; now for the pentagon
(ask pent plan next gradually right 90)

(ask movie make my-first-film) ;; everything is all planned so lets make the movie
(ask my-first-film set your frames-per-second to 2)
;; if the computer were faster 30 might be nicer
(ask my-first-film film the next 6 ticks) ;; roll the cameras for the next 6 clock ticks
(ask my-first-film project) ;; you just saw the film being shot, now lets see it projected
```

III. Something

When you start up a new Director you initially have available to you only a few actors. The most important one is called **Something** and it does everything that every actor should be able to do. In other words, **Something** maintains a memory, accepts print messages, makes instances of itself, maintains plans for pseudo-parallelism (with the help of **Universe**), and can be told how to handle new messages. **Object**, **Movie**, and **Screen** do graphical things and are described later.

Every actor is an offspring of **Something** and therefore unless explicitly told otherwise will behave as **Something** does when receiving the messages is this section. The sections describing messages of **Object**, **Movie**, **Universe**, and **Screen** apply only to those actors and their descendants. The relationships of all the actors initially present in Director is depicted in the following diagram.



This guide is organized by the patterns of messages an actor can handle. Each section describes a primitive actor by listing those message patterns directly associated with it (i.e. those messages that the actor itself handles rather than passing the problem on up to its parents or more distance ancestors). Variables that the actor treats specially are also described.

A pattern is a list of words or patterns. If a word begins with a question mark (?) or a "%" then it is treated specially. Patterns are matched against messages. A question mark means that anything may be typed in the corresponding position of a message. If a question mark is followed by a word, then that word becomes the name of the what you typed in the corresponding position in the message. A "%" means that any number of sub-items (even zero) can be at the corresponding position of the message. For example, the message "(I eat pot stickers)" matches the pattern "(I

?action %things)" and as a result the name "action" temporarily gets the value "eat" and the word "things" is bound to (ie gets the value) "(pot stickers)". Currently the "%" can only be used at the end of a pattern.

You may type upper or lower case letters as you prefer. The patterns given in this paper use capital letters for required words and lower case for variable names in patterns. If there is an abbreviation for a pattern in the text then it is on the far right of the pattern.

Creation and Destruction of Actors

`(ASK ?anyone MAKE ?name)`

This causes an instance of the receiver of the message to be made and returned. The newly created actor will behave exactly as its parent does (except when asked for its name, offspring, or parent's name of course). You can tell it new things to remember or how to behave if it receives new kinds of messages. If there was already an actor around with the same name it will be destroyed and replaced by this new one.

Any time you want to ask an actor anything you type "(ask ", then its name, the message and end with a ")". So to create an actor named Sally just type,

`(ask something make Sally)`

`(ASK ?anyone MAKE INTERNED OFFSPRING)` **mio**

This behaves just like "make ?name" except that here you lets Director pick a name.[1]

`(ASK ?anyone MAKE UNINTERNED OFFSPRING)` **muo**

This behaves like the previous one only the name it picks can not be typed in. To talk to

---

1. Mio is an abbreviation for "make interned offspring" so you can just as well type (ask joe &mio). The "&" is needed to signal Director that what follows is an abbreviation.

the resulting actor you must either save the result of this message in a Lisp variable or have some other actor store it.[1]

(ASK ?anyone IF NEW MAKE ?name) inm ?

If there already exists an actor with the same name then nothing happens otherwise one is made. Name is returned in either case.

(ASK ?anyone MAKE COPY ?sibling) mc ?

An exact copy of the recipient in this message transmission is created. The only differences between the new actor "sibling" and the recipient are their names and that the newly created actor has no offspring. Of course, afterwards they may be told different facts and how to handle messages. This creates an identical twin while the other "Make" messages create children. One big difference is that if you change a twin its sibling is not affected but if you change a parent then its children potentially will be affected.

(ASK ?anyone MAKE SYNONYM ?name) ms ?

This does not create a new actor. Instead the recipient is given an alias, another name with which you can use to send it messages. Besides providing a way to give shorter names to any actor, this is often useful when the name is a list. For example, if you have an actor that is the comparison between A and B then you can define the comparison and give it equivalent name as follows,

```
(ask something make (comparison-of A B))
(ask (comparison-of A B) make synonym (comparison-of B A))
```

---

1. The only reason for putting up with this inconvenience is so that the actor is subject to garbage collection, i.e. will go away when you no longer can get to it to ask it anything.

Defining the Behavior of Actors

(ASK ?anyone DO WHEN RECEIVING ?pattern %action) **dwr ? %**

This very important message type expands the repertoire of an actor. The "%" indicates that "action" is to be the name of the part of the message that follows the "pattern".

Suppose you want to make an actor named "Sally" who will respond with "yes it is a nice day, morning or evening" when you say "good morning, evening or whatever". You just type:

```
(ask something make Sally)
(ask Sally do when receiving (good ?time-of-day)
            ;; if Sally receives the word good followed by another word
            ;; which we will call by the name time-of-day then
            (type '(yes it is a nice ,time-of-day)))
```

If the comma had not preceded the variable "time-of-day" then she would always type the message "yes it is a nice time-of-day". To test it out we again ask Sally something, this time to respond to "good morning".

```
(ask sally good morning)
; to which she responds
YES IT IS A NICE MORNING
```

(ASK ?anyone EXTEND BEHAVIOR WHEN RECEIVING ?pattern BY DOING %some-action) ebwr ? bd

This is useful for extending the behavior of an actor in response to a message that it already can respond to. If while running "some-action" a message is sent that matches the "pattern", then this new clause is ignored and the old action is taken.[1] Suppose you want Star to move forward and in addition grow whenever it is told to go forward.

---

1. The main reason the standard "receive" message is not defined to behave this way is to allow recursion. If each receiver clause were itself a full actor then extending their behavior could be done in a much cleaner way.

```
(ask star extend behavior when receiving (forward ?amount) by doing
        (script: (ask :self grow ,amount) ;; ask yourself to grow the same amount
                 (ask :self forward ,amount))) ;; send along the message to yourself
```

You could have used "star" instead of ":self" but :self will also work right if you make any offspring of Star and ask them to go forward, since :self always contains the name of the original recipient of the message. The word "script:" is used to protect the commas. Recall that they mean use the value of the following expression and so without the word "script:" Director would be confused and say that "amount" has no value when you type the above expression. Using the word "script:" means that the comma should get the value of the following expression when the script is being run. To learn how to remove extended behavior, if no longer wanted, see the "remove clause for..." message below.

If you are confused about how this message differs from the "do when receiving" message above try making some actor and tell it "do when receiving" with the same pattern and action as above. Then tell it to go forward some small amount. After you get the idea type control-g (while holding down the control button (ctrl) type g).

Printing

`(ASK ?anyone PRINT %option)` ps OR pm OR pv OR pdb

The possible "options" are "script", "memory", "variables", "database" or "all". If no option or "all" is given then all are assumed. They cause the script or the memory of the recipient to be printed. The memory is split up into "variables" and "database". If a second option is given it is assumed to be a file name to which to print. So to print the entire definition of Sue to a file type

```
(ask Sue print all Sue file)
```

This will cause Sue to be printed out in a form that is designed to be easiest to read. It can not be read back into a Director however. To do that use the following message

(ASK ?anyone SAVE %file-name)

This message causes the recipient to print itself out in Lisp onto the "file-name". If a file of that name already exists it will add itself to the end. You can call the Lisp compiler upon the file or just use this to save away an actor onto file. Using the Lisp function "Load" you can get the saved actor back into another Director. If no file-name is given then it will return the Lisp form instead. This does not work if the actor is compiled as is the case with all the actors available in a fresh Director. If you want to save away Sally and Sue for another time, for example, then type

```
(ask sally save flower file) ;; put sally in the file called flower file
(ask sue save flower file) ;; put sue there too
```

So another time if you want Sally and Sue just type (load '(flower file)).

(ASK ?anyone HELP %pattern)

This causes the recipient to print out comments about the different messages it can receive. If no pattern is given then all the different messages are described. If a pattern is gven then only those that match the pattern are printed. So to see all the messages that begin with the word "project" that a movie can handle just type

```
(ask movie help project %)
```

(ASK ?anyone RECALL CLAUSE FOR %sample-message) rcf ?

This will return the first clause that matches the "sample-message". NIL is returned if the clause is compiled or if nothing matches the message. For example, if you tell Sally,

```
(ask sally do when receiving (bye ?when) '(goodbye ,when)) ;; eg (ask sally bye now)
(ask sally recall clause for bye any-ol-time)
((BYE ?WHEN) '(GOODBYE ,WHEN)) ;; is returned
```

(ASK ?anyone REMOVE CLAUSE FOR %sample-message)

This removes the first clause to catch the "sample-message". You can even remove clauses from the compiled primitive actors such as Something and Object so use this message with care.

Tracing

`(ASK ?anyone TRACE ?pattern %action)`

If any transmission occurs that match the "pattern" then a comment that it happened is printed and the "action", if any, is taken. Finally the message is sent and a comment is printed describing the results. A transmission is the recipient combined with the message. This is so that you can ask a parent to trace some or all messages passed to it from its offspring. If you want to see all the messages for Sally that begin with either grow or shrink, you should ask her to trace by typing[1]

```
(ask sally trace (sally {or grow shrink} %))
```

or if you want a break point when she or any of her descendants receives a message that changes the value of the variable "size" then type

```
(ask sally trace (? {or set change} your size to ?) (break size-being-set))
```

If you want to trace every message that Sally and her descendants receives you can type

```
(ask sally trace) ;; same as (ask sally trace (?))
```

`(ASK ?anyone UNTRACE ?pattern)`

This removes any traces that match the "pattern". If no pattern is given (the message is simply untrace) then all traces on that actor are removed. Untrace removes traces from the actor receiving the message and all of its descendants. Therefore to remove all traces from everyone just type "(ask something untrace)".

---

1. Patterns in Director can be more complicated than described so far. A complete description of patterns can be found in a later section. The use of {} and or in the examples here mean that the pattern succeeds if it matches either of the subpatterns.

Memory Messages

Something (and since all actors are descendants of **Something** this is true of all actors in Director) has two kinds of memories, variables and a relational data base. The variables are known by the actor that was originally told to set them and indirectly by all of its offspring and more distant descendants. The data base is good for remembering any list structure and recalling it later with a pattern.

Variables

Each Director variable is associated with a particular actor. The value and the *name* of a variable may be either an atom or a list. So for example, an actor may have a variable called "(EYE COLOR)" and its value may be "BLUE".

```
(ASK ?anyone (or CHANGE SET) YOUR ?variable TO ?new-value)        sy ? to ? OR cy ? to ?
```

The part enclosed in {} means that either the word "change" or "set" can be typed, there is no difference. A message matching the pattern causes the value of "variable" to be changed to "new-value". If the actor had no such variable then one is created and its value set. New-value is returned. For example, if Sally just got a hair cut you might want to update her hair length as follows

```
(ask sally set your (hair length) to medium)
```

which causes Sally's variable "(hair length)" to be set to "medium".

```
(ASK ?anyone RECALL YOUR ?variable)                                ry ?
```

This message causes the value of "variable" to be returned. If there is no such variable associated with the actor, then its parent will be asked the same question, and so on until either a

value is found or finally NIL is returned.[1] So, if you ask Sally

```
(ask sally recall your (hair length))
```

the word "medium" will be returned.

```
(ASK ?anyone RECALL EACH OF YOUR ?variable-pattern)                reoy ?
```

Variable names may be either atoms or lists. If it is a list then one can refer to it by giving only part of its name and a "?" for each missing part. This message causes the return of the list the values of all the variables that match the "variable-pattern". Note that variable patterns are not as general as message patterns, only "?"s are permitted and only at the top level of the pattern. Using this feature an actor can have the equivalent of property lists, nested property lists, and arrays. For example,

```
(ask sally set your (color-of friend bob) to red)
(ask sally set your (color-of friend sam) to blue)
(ask sally set your (color-of stranger tom) to green)
;; then asking
(ask sally recall each of your (color-of friend ?))
(BLUE RED) ;; is returned
;; while asking
(ask sally recall each of your (color-of ? ?))
(GREEN BLUE RED) ;; returns all three
```

If you made the movie My-first-film described earlier and you want to see all the frames you can type

```
(ask my-first-film recall each of your (frame ?))
```

This pattern notation for variables also works for the "set or change" messages. For example, typing

```
(ask sally set your (color-of ? ?) to yellow)
```

---

1. The returning of NIL for unbound variables is very convenient default, especially for novices. This can be overridden so that it becomes an error by using the variable demons discussed below.

will set all the variables of Sally that are three long and begin with "color-of" to "yellow".

`(ASK ?anyone INCREMENT YOUR ?variable BY ?amount)` **iy ? by ?**

This causes the value of "variable" to be set to the sum of its old value plus "amount" in the message. If there is no such variable then one is created and set to the "amount". Unless both "amount" and the value of "variable" are numbers an error will result. The new value of "variable" is returned. So if you type

```
(ask sam increment your size by 10)
```

and his old size was 20 then his size is now 30. If Sam did not know what to do when he received a message asking him to increment a variable you could have told him by typing

```
(ask sam do when receiving (increment your ?variable by ?amount)
        (script: (ask :self change your ,variable to
                      ,(plus (or (ask :self recall your ,variable)
                                 0) ;; In case it has no value already
                             amount))))
```

Notice that this message and the following ones do not change the value of a variable of an ancestor. So if when asking Sam to increment his size, he did not know his size and had to ask his parent, then his parent's size is not affected by this change. Instead a variable for size is created for Sam and initialized to be the sum of his parent's size and "amount".

`(ASK ?anyone ADD ?new-item TO YOUR LIST OF ?list-name)` **add ? tylo ?**

If "new-item" is already a member of the contents of "list-name" then nothing happens, otherwise "new-item" is added onto the value of "list-name". The new value of list-name is returned. For example, typing

```
(ask sam set your neighbors to (fred bob sally))
(ask sam add sue to your list of neighbors)
```

results in Sam's variable "neighbors" to be set to (SUE FRED BOB SALLY).

(ASK ?anyone ADD ?new-item TO YOUR LIST OF ?list-name REGARDLESS) add ? tylo ? reg

This is the same as the previous "add ..." message only the "new-item" is added regardless of whether it is already a member of the value of "list-name".

(ASK ?anyone REMOVE ?old-item FROM YOUR LIST OF ?list-name) remove ? fylo ?

This removes all copies of "old-item" from the contents of "list-name". Suppose Bob moves away from Sally, then you can ask Sally

```
(ask sally remove bob from your list of neighbors)
```

(ASK ?anyone LIST ALL YOUR VARIABLE NAMES) layvn

Returns a list of the names of all the variables known directly by the recipient. Special variables like descendants and siblings are not included since they are computed when asked for and are not known otherwise.

(ASK ?anyone FORGET YOUR ?variable-pattern) fy ?

If "variable-pattern" is the name of a variable it is forgotten (the recipient is restored to the same condition as before the variable was created). If it is "?" then all variables are forgotten except for "parent", "offspring" and "name". If "variable-pattern" is a list containing question marks then all those variables which match are forgotten. So typing

```
(ask sally forget your (color-of friend ?))
```

will cause Sally to forget about the variables "(color-of friend sam)" and "(color-of friend bob)".

Demons for Variables

Sometimes you want some special action to take place when recalling or changing the value of a variable. Each special action is called a demon and the demons for a variable are kept in a variable whose name starts with the name of the variable to be watched followed by either "actions-if-recalling" or "actions-if-changing" depending on whether the action should take place when the variable is recalled or changed. Variables matching the following variable patterns are run when their corresponding variable is changed or recalled.

Variable pattern `(?variable-name ACTIONS-IF-RECALLING)` *is abbreviated as* `(? airx)`

Such variables can be created, modified and inspected using the normal variable messages. If the second name of a variable is "actions-if-recalling" then if the "variable-name" is ever recalled, then each member of the list of actions-if-recalling is evaluated. The value returned by the recall is the last form of the list evaluated. The forms can reference the Lisp variable ":old-value" which is the current value of the variable. So if you want Jack to always lie about his age, so that it is never greater than 39 you can type

```
(ask jack add (min :old-value 39) ;; return the minimum of current age and 39
     to your list of (age actions-if-recalling)) ;; if ever recalling his age
(ask jack set your age to 78) ;; now to test it out
78
(ask jack recall your age)
39 ;; to which he responds 39
```

This feature is used by objects (descendants of Object which is described later) which have a variable called "state" that is a list of the x coordinate, y coordinate and heading. When you ask an object for its heading it returns the third element of this "state" variable. This is done by

```
(ask object add (third (ask :self recall your state))
     to your list of (heading actions-if-recalling))
```

Variable pattern `(?variable-name ACTIONS-IF-CHANGING)` *is abbreviated as* `(? aic)`

When changing or setting (they are the same thing) the "variable-name" then the variable whose first name is "variable-name" and second name is "actions-if-changing" is recalled. Each element of that variable (if there is one) is then evaluated. The value that the variable has just been set to is available in the Lisp variable ":new-value". Suppose you know that Jack should never be over 39. Then to get a warning if his age is set to too high a value just enter

```
(ask jack add
     (cond ((> :new-value 39) ;; if the value is greater than 39
            (type '(warning jacks age set to ,:new-value which is too high))))
     to your list of (age actions-if-changing))
(ask jack change your age to 73) ;; try it out by setting his age to too high a value
WARNING JACKS AGE SET TO 73 WHICH IS TOO HIGH
73 ;; but his age is set to 73 anyway
```

If you wanted his age automatically set to 39 in this case then type

```
(ask jack add (cond ((> :new-value 39)
                     (ask jack set your age to 39)))
          to your list of (age actions-if-changing))
```

Relational Data Bases

Associated with each actor (since each actor is an offspring of Something) is a powerful data base. The patterns used to retrieve items from the data base have the same form as the message patterns, e.g. "?" is used for anything in that position, "%" for anything taking up any number of positions. A complete description of patterns can be found in a later section.

Memorizing

`(ASK ?anyone MEMORIZE ?item)` **mem ?**

Any list structure may be remembered. Once an actor is told to memorize an item it will never forget it unless you ask it to "forget" as described later. Some examples of memorizing are

```
(ask sally memorize (color sky blue))
(ask sally memorize (color ocean blue))
```

Retrieving

It doesn't do any good to have Sally remember these things if we can't ask her about them so we have the following messages.

(ASK ?anyone RECALL IF ANY ITEMS MATCH ?pattern) **raim ?**

Sally will answer "T" (for true) or "NIL" (for don't know anything about it) depending on whether she was ever told something that matches the "pattern". So if you asked Sally the following questions

```
(ask sally recall if any items match (color sky blue))
T ;; i.e. yup
(ask sally recall if any items match (color ?anything blue)) ;; same as (color ? blue)
T ;; of course
(ask sally recall if any items match (color ?anything green))
NIL ;; dont know of any green things
```

(ASK ?anyone RECALL AN ITEM MATCHING ?pattern THEN ?action) **raim ? then ?**

It would be nice if Sally would say more than yes or no. One way to do this is to use this message. If she remembers at least one thing that matches the "pattern" she will take whatever "action" you ask her to. So, if you type •

```
(ask sally recall an item matching (color ?thing blue)
           then (type '(i know that the ,thing is blue)))
I KNOW THAT THE SKY IS BLUE ;; to which she responds
```

If you hadn't told her about any blue things then she would have answered "NIL".

The first element of an item in a database is treated specially in order to speed up retrieval. Because of this you should always provide the first element completely without using any question

marks. If you don't like this you can always use some dummy word in the front of all the items.

`(ASK ?anyone COLLECT ITEMS MEMORIZED MATCHING ?pattern)` `cimm ?`

The previous messages aren't too helpful if the actor in question has remembered several items that match the pattern since you never know which item the actor will base its answers on. Director once had (and it could come back by popular demand) a series of messages for creating a stream of answers to questions that could be interrogated for its answers one by one. Instead of that, this simplier, less general, "collect items memorized matching..." message is provided which collects into a list and returns all the forms that were memorized and match the "pattern". So typing

```
(ask sally collect items memorized matching (color ? blue))
((COLOR OCEAN BLUE) (COLOR SKY BLUE)) ;; is returned
```

Demons and Virtual Items

You sometimes want special actions to occur during some database references. Similar to the demons for variables, you can create database demons to satisfy such needs. Variables whose name consists of the first element of the item in question followed by either "if-recalling-items" or "actions-if-memorizing" are evaluated when an item with the same first element is called for or memorized.

Variable pattern `(?first-element IF-RECALLING-ITEMS)` *is abbreviated as* `(? ir1)`

The constructors of virtual items (items that are not really in the actor's database but that pretend to be) are found on variables whose names match the above pattern. When needed by either a "recall if any items match", a "recall an item matching" or a "collect items memorized matching" message they get run.

Suppose you want to have a table actor that keeps track of blocks on top of it. You want it to remember which blocks are left and right of each other. Furthermore, you don't want to clutter

the table up with too many facts. One thing to do is to have the table memorize only items about which blocks are left of which others and to use "if-recalling-items" demons to fake it to look as if it also had memorized which are right of which others. To do this enter,

```
(ask table add
     ;; add to the table's list of demons for questions beginning with right-of
     (if-asked-about (right-of ?block-1 ?block-2)
      ;; if asked about a block being right of another then
      (script:
       (ask :self recall an item matching ;; then look for an item that says that
                 (left-of (and ?b2 ,block-2) (and ?b1 ,block-1))
                 ;; the second block is left-of the first one
                 then '(right-of ,b1 ,b2))))
                 ;; and if found pretend that a right-of item was there
          to your list of (right-of if-recalling-items))
```

Only those items constructed that match the original request are used in answers. To test out our table we can enter

```
(ask table memorize (left-of (green block) (red cube))) ;; tell it something
(ask table recall an item matching (right-of (red cube) ?some-other-block)
     then (type '(the ,some-other-block is right of the red cube)))
;; after asking it for something that is right of the red cube it responds
THE (GREEN BLOCK) IS RIGHT OF THE RED CUBE
```

Variable pattern `(?first-element ACTIONS-IF-MEMORIZING)` *is abbreviated as* `(? aimx)`

If, instead of pretending that some item is memorized, you want some special actions to occur when certain items are memorized then you should put the actions in a variable whose name is the first element of the items in question followed by "actions-if-memorizing". Suppose you rather have the table memorize both the left and right relationships of its blocks, but you don't want to have to enter both kinds of statements. Then using actions-if-memorizing you can have one kind automatically added after the other kind is. To get the table to memorize right-of statements after first hearing of the left-of version just type,

```
(ask table add
     ;; add to the list of actions to happen if memorizing an item beginning with left-of
     (if-told-about (left-of ?thing-1 ?thing-2)
      (script:
       (cond ((ask :self recall if any items match (right-of ,thing-2 ,thing-1)))
             ;; If I already have memorized the corresponding item then do nothing
             (t (ask :self memorize (right-of ,thing-2 ,thing-1))))))
             ;; Otherwise memorize the item right-of with the things switched around
     to your list of (left-of actions-if-memorizing))
```

The reason I first checked whether the new item is already known is so that the analogous demon can be placed upon the pattern "(right-of ? ?)" without the two activating each other forever.

Forgetting

`(ASK ?anyone FORGET ITEMS MATCHING ?pattern)` **fim ?**

This removes any previously memorized item matching the "pattern" from the data base of an actor. If you want Sally to forget about all the blue things she knows about you could type:

```
(ask sally forget items matching (color ? blue))
```

To have her forget everything (except her name, parent and receiving clauses) type

```
(ask sally forget items matching (%)) ;; forget every data base item
(ask sally forget your ?) ;; forget all your variables
```

Plans and Pseudo-Parallelism

Normally when an actor receives a message it responds as quickly as possible. In order to have several actors behave at what seems to be the same time this is not desirable. Director solves this by providing a "tick" mechanism which is described in greater detail in [Kahn 1978].

`(ASK ?anyone PLAN NEXT %action)` **pn %**

This indicates that the message called "action" should be sent after the next tick. Ticks are explained later in this section.

(ASK ?anyone PLAN AFTER ?number TICKS %action) **pa ? ticks %**

This requests that the message called "action" be sent after "number" ticks. Suppose you want to have Sally grow and after that to go forward and at the same time you want Sam to go forward on the next tick and to grow on the tick after that. You could type

```
(ask sally plan next grow 100)
(ask sally plan after 2 ticks forward 50)
(ask sam plan next forward 50)
(ask sam plan after 2 ticks grow 100)
```

Once all the plans have been made you can get any actor to do all planned on the next tick by asking it to (TICK). If you ask an actor named **Default-universe** to (TICK) then all the actors will do what they planned for the next tick. If you ask **Default-universe** to **"RUN FOR 3 TICKS"** then 3 ticks of action will happen.

(ASK ?anyone PLAN AFTER ?number SECONDS %message) **pa ? seconds %**

This is the same as the previous message except that the units are in *film* seconds. While making a movie there are variables for the number of frames per second and ticks per frame which are multiplied by "number" in the message to get the number of ticks.

(ASK ?anyone PLAN AFTER RECEIVING ?event-pattern TO ?message-form) **parx ? to ?**

If the recipient of this message type receives a message matching the "event-pattern" then the "message-form" is evaluated (in the environment extended by matching the event-pattern against the event message) and sent to the recipient. Suppose you want Sam to melt after colliding with a sun otherwise to explode. Then you could enter:

```
(ask sam plan after receiving (colliding with ?other) to
        ;; if a message matches (colliding with ?)
        (script: (cond ((ask ,other are you a sun) ;; if the other is a sun
                        '(melt)) ;; then the message to be sent is melt
                       (t '(explode))))) ;; otherwise it is explode
```

(ASK ?anyone PLAN AFTER RECEIVING ?event-pattern TO ALWAYS ?message-form) parx ? ta ?

This is just like the last message except that it will *always* do "message-form" after receiving a message that matches "event-pattern", while the other one only works the first time that a message matches. The affect of this message could easily be accomplished using the "extend behavior when receiving" message. Suppose you want Sally to grow 150 units 3 ticks after gradually going forward 200, then you could type

```
(ask sally plan after receiving (note that I finished gradually forward 200)
      to always '(plan after 3 ticks grow 150))
```

(ASK ?anyone REPEAT ?message ?number TIMES)

This causes the "message" to be sent now and the plan the messages the comma means the value instead of the name.

(ASK ?anyone REPEAT ?message ?number TIMES EVERY ?so-many TICKS) repeat ? ? te ? ticks

If beacons had not be told how to blink they could be told to using this message by entering,

```
(ask ,beacon repeat (show) 10 times every 2 ticks)
(ask ,beacon plan next repeat (hide) 10 times every 2 ticks)
```

So that the beacon will show and hide on alternate ticks.

(ASK ?anyone REPEAT ?message FOREVER)

This is similar to the previous message only the action is repeated *ad infinitum*. This is defined as follows:

```
(ask something ;; every actor inherits this behavior from Something
     do when receiving (repeat ?message forever)
     (script: (ask :self !,message) ;; send the message to yourself
              (ask :self plan next repeat ,message forever)))
              ;; and plan next to repeat the same thing again
```

The "!" in the first line of the script means that the list that follows should be inserted into the list without its parenthesises. So typing "(ask sally !(print memory))" is exactly the same as typing "(ask sally print memory)".

(ASK ?anyone TICK)

This message causes those messages planned for the next tick to be sent. These are kept on the variable called "things-to-do-next". Conceptually the transmissions planned for a tick happen in parallel as does the broadcasting of tick messages by either a movie or a universe. A tick is a quantum of time during which you should not care about the order of transmission of any planned messages. See the sections describing **Movie** and **Universe** below for more details.

Broadcasting Messages

Sometimes you want to have an actor send messages on along to others it knows about. The following messages are to help you do that.

(ASK ?anyone ASK YOUR ?variable TO %message) **ay ? to %**

This asks the recipient to recall its "variable" and then send to it the "message". For example,

```
(ask sally set your friend to bob)
(ask sally ask your friend to recall your offspring)
```

Hopefully Sally has more than one friend. In that case you can create a different variable ("friends" is a good name) and use the next message.

(ASK ?anyone ASK EACH OF YOUR ?variable TO %message) **aeoy ? to %**

This sends out the "message" to each of the members of the list that is the value of "variable". Nothing happens if there is no value for "variable". As an example,

```
(ask sally set your friends to (bob ted carol alice))
(ask sally ask each of your friends to print variables)
```

There are a few special variables that are useful together with this message. They are "descendants", "childless-descendants", "siblings" and "offspring". Offspring are all the children of an actor, siblings are all the offspring of an actor's parents except itself, descendants are all the children and their children's children and so on, and childless-descendants are those descendants that themselves have no children. If you wanted all the actors in existence (except **Something** and those actors created by "make uninterned offspring") to print, for example, you just type

```
(ask something ask each of your descendants to print)
```

```
(ASK ?anyone ASK ?another TO %message)
```

This one may seem kind of silly. Why ask someone else to ask another person something, why not just ask them yourself? This is handy mostly when you are planning or repeating something because they, by convention, deal with messages that are to be sent to the planner at some later time. The planner is typically the same actor as the one who executes plans but occasionally this is not the case. For example, if you want Sally to tell someone that he is a clutz because he just collided with her (rather then do something herself such as explode), then you could type

```
(ask sally plan after receiving (colliding with ?other)
     to always '(ask ,other to note that you are a clutz))
```

If Sally receives a message informing her that she is colliding with Bob then the following transmission occurs

```
(ask sally ask bob to note that you a clutz)
```

```
(ASK ?anyone DO THE FOLLOWING: %MESSAGES)                    dtf %
```

This too may seem silly at first, why send yourself many messages at once like this? Again

this message is handy when planning to do something later. Since you have no say, regarding the order of messages sent during a tick, this pattern provides you with some. The difference is subtle. Using this message you can plan to do a compound action at a particular time. The alternative is to plan to do several simple actions all at that time. Suppose you want Sue to draw a pentagram (a five pointed star), then you could type either

```
(ask sue pendown) ;; So that she leaves a trail behind her
(ask sue repeat (do the following: (forward 200) (right 144))
```

or if you don't care whether the star is drawn by going forward first or right first then the following is just as good

```
(ask sue repeat (forward 200) 5 times)
(ask sue repeat (right 144) 5 times)
```

Remember that the "repeat" message is defined to do the message and plan to repeat one less time the same message.

(ASK ?anyone KEEP DOING UNTIL ?predicate %message) **kdu ? %**

This keeps sending the "message" to itself until it results in something that satisfies the "predicate". Suppose you want Sam to grow 10 until he is at least 200 big. Then you could type

```
(ask sam keep doing until (lambda (result) (> result 200)) grow 10)
```

Variables that are Special to All Actors

There are a few variables that are treated specially. Among them are "siblings", "offspring", "descendants", and "childless-descendants". If there is no value for one of these variables instead of asking the recipient's parent for a value one is made up. It returns either the relatives indicated or NIL if there are none.

name *is a variable whose default value is* <name used when made>

Every actor has a name.

`parent` *is a variable whose default value is* `<the maker of recipient>`

Every actor has as its parent the actor that created it. It essential to have a parent to accept any message or recall any variable or database item that one does not know explicitly. You can change your mind as to who a parent of an actor is and reset it using "change your ..." but the previous parent's offspring (see below) will not be changed and the new parent will not automatically know of this new offspring.

`offspring` *is a variable whose default value is* `nil` *and is abbreviated as* `offs`

Every actor knows the names of all its offspring (except those that are not interned). This variable is not inherited of course.

`siblings` *is a variable whose default value is* `nil` *and is abbreviated as* `sib`

An actor's siblings are all of its parent's offspring except itself.

`descendants` *is a variable whose default value is* `nil` *and is abbreviated as* `des`

This is a list of the actor's offspring and their offspring and so on.

`childless-descendants` *is a variable whose default value is* `nil` *and is abbreviated as* `cd`

This is a list of all the actor's descendants that themselves have no offspring.

`universe` *is a variable whose default value is* `default-universe` *and is abbreviated as* `uni`

The variable "universe" is used in planning and its value should always be a descendant of Universe. Its value is informed whenever an actor plans anything.

`things-to-do-next` *is a variable whose default value is* `nil` *and is abbreviated as* `ttdn`

This variable contains the list of things that its owner plans to do upon receipt of the next tick message. You need not worry about this variable unless you want to write your own planning primitives.

IV. Object

Object is the top-level actor for those that can be seen on a display screen. Objects are much like Logo turtles only much more versatile. Many of the message patterns are intended to be compatible with the TV turtle.

Creation and Appearance

Objects are created using the "Make" message described above. In order to inform this newly created actor of its appearance it must be told the name of a Lisp program (made up of turtle commands) that will draw it on a display. If its appearance is not easily describable by turtle commands (such as text) then you can use the instant turtle or you can supply clauses for the reception of display and erase messages. This is all described later.

`(ASK ?an-object WHEN DRAWING USE ?draw-procedure OF %draw-args)` **wdu ? of %**

This informs "an-object" that it should draw itself using the "draw-procedure". The procedure may consist of **Forward, Back, Right** and **Left** commands but coordinate commands (for example, Setxy) should be avoided. Instead of **Penup** and **Pendown** use **Thingup** and **Thingdown**. "Draw-args" is a list of variables that must be "Recall-able", i.e. variables known (either directly or through inheritance) by the actor in question. If these variables are changed and the object is currently being displayed, then its its appearance will be updated. For example, to define an object that can appear as any polygon we could type

```
(define poly object
        (set your angle to 60) ;; set the default to be a hexagon
        (when drawing use draw-poly of size angle))
```

The macro "Define" is just short hand for:

```
(ask object make poly)
(ask poly set your angle to 60)
(ask poly when drawing use draw-poly of size angle)
```

Draw-poly in Lisp could be defined as follows:

```
(defun draw-poly (distance turnage)
 (do ((original-heading (heading)) ;; save away the original heading
      (first-time? t))
     ((and (= original-heading (heading)) ;; if back to the original heading
           (setq first-time? (not first-time?))))
           ;; and if this is not the first time through then it's finished
     (forward distance) ;; each time go forward the distance
     (right turnage))) ;; and turn right the turnage
```

When giving the arguments to the drawing procedure one may put any of the arguments in parenthesis. This declares to Director that the argument does not change the shape of the object. For example, if the drawing procedure has an argument for the texture then it should be in parenthesis otherwise whenever the texture is changed Director will go through much more work than necessary. If "draw-poly" had a third argument for the texture then you should type, "WHEN DRAWING USE DRAW-POLY OF SIZE ANGLE (TEXTURE)". Also, the variable "size" is treated specially by the system (in the definition of grow and shrink for example) so you should use that name when that is what you mean. All sizes are standardized so that any object of size, say, 100 will just fit inside a circle of radius 100 units. You needn't worry about centering the appearance, Director will do it for you. This can be overridden by setting the variable "center-offset" as described below.

Currently the initial environment includes this definition of **Poly** and two other objects, **Rocket** and **Flower**.

Showing and Hiding

`(ASK ?an-object SHOW)`

"An-object" is shown if not already being shown.

`(ASK ?an-object HIDE)`

"An-object" is hidden if currently visible.

Moving and Turning

Just as in Logo there are at least two ways of changing an object's position or orientation. You can ask it to go forward or to turn, or you can ask it to change either its "xcor" (the horizontal coordinate), "ycor" (the vertical coordinate), "heading", "position" (the xcor and ycor) or "state" (the xcor, ycor and heading).

Turtle Commands

There are two ways of moving or turning. An object can either disappear from where it is and appear in its new position or orientation. Another way of moving or turning is to do it gradually. "Gradually" messages to objects do *not* cause them to move slowly when the message is received. Instead they move only a tick's worth (see previous discussion of ticks) and plan to do the rest. In order to see the object move gradually one should ask an instance of **Universe** or **Movie** to run for a number of ticks (see below).

`(ASK ?an-object (and ?command (or FD FORWARD RT RIGHT LT LEFT BK BACK)) ?amount))`

If any of the words fd, forward, rt, right, lt, left, bk or back followed by any sort of number is received by an object then it will hide, do the "command", and reappear. So typing `(ask sally forward 200)` causes Sally to disappear and then to appear 200 steps forward from the way she

was facing (her heading). If she was hidden to start with she will move but you won't see her.

`(ASK ?an-object GRADUALLY ?command ?amount)` `grad ? ?`

This causes the actor to find its appropriate speed and send itself a message asking it to do "command" with either the speed or "amount" whichever is less. The name of the speed is found in the variable (,command speed-name). If there is anything left over to do, then it plans to do that next. So typing

```
(ask sally gradually forward 250)
;; is the same as typing (assuming her speed is less than 250)
(ask sally forward ,(ask sally recall your speed))
(ask sally plan next gradually forward ,(- 250 (ask sally recall your speed)))
```

The speed of change is the variable "speed" if the message is a "forward" or "back" message. It is "rotational-speed" if the message is "right" or "left". Object knows reasonable defaults for these variables but of course you can override any of them. When an object does something gradually such as `(GRADUALLY RIGHT 88)` then the message `(NOTE THAT I STARTED GRADUALLY RIGHT 88)` is sent to itself and later when it finished the message `(NOTE THAT I FINISHED GRADUALLY RIGHT 88)`. These are very handy for making complicated plans where only after some event has begun or finished should another start.

`(ASK ?an-object RIGHT-REVOLVE ?degrees)` `rr ?`

`(ASK ?an-object LEFT-REVOLVE ?degrees)` `lr ?`

These cause the recipient to revolve around its "revolution-center" if it has one, otherwise it acts just as if it received either "right" or "left". It travels in a circle around its "revolution-center" travelling "degrees" to the right or left.

`(ASK ?an-object GRADUALLY RIGHT-REVOLVE ?degrees)` `grr ?`

`(ASK ?an-object GRADUALLY LEFT-REVOLVE ?degrees)` `glr ?`

These causes "an-object" to revolve at the lesser of its "revolution-speed" and "degrees" and

then plan to do the rest next.

Coordinate Messages

Sometimes you might want to tell an object where to go by giving the distance up or down and the distance left or right from either the current position or the center of the screen. The up and down part is called the y coordinate and the left and right part is the x coordinate. The following are the coordinate-oriented messages

`(ASK ?an-object SETXY ?new-x ?new-y)`

Set x coordinate to new-x and y to new-y.

`(ASK ?an-object SETX ?new-x)`

`(ASK ?an-object SETY ?new-y)`

`(ASK ?an-object SETTURTLE (?new-x ?new-y ?new-heading))` **sett (? ? ?)**

This is the same as (change your state to ?new-state).

`(ASK ?an-object SETHEADING ?new-heading)`

Sets the heading same as (change your heading to ?new-heading).

`(ASK ?an-object DELXY ?delta-x ?delta-y)`

Add delta-x to current x coordinate and delta-y to y.

`(ASK ?an-object DELX ?delta-x)`

`(ASK ?an-object DELY ?delta-y)`

`(ASK ?an-object CHANGE YOUR STATE TO ?new-state)` **cyst ?**

Where state is (xcor ycor heading) same as setturtle new-state.

`(ASK ?an-object CHANGE YOUR POSITION TO ?new-position)` **cypt ?**

Position is just (xcor ycor).

`(ASK ?an-object CHANGE YOUR HEADING TO ?heading)` **cyht ?**

Sets the heading

`(ASK ?an-object CHANGE YOUR XCOR TO ?new-xcor)` **cyxt ?**

Synonymous with (setx ...).

`(ASK ?an-object CHANGE YOUR YCOR TO ?new-ycor)` **cyyt ?**

Same as (sety ...).

`(ASK ?an-object RECALL YOUR STATE)` **rys**

Returns the current state

`(ASK ?an-object RECALL YOUR POSITION)` **ryp**

Returns the list of the xcor and ycor

`(ASK ?an-object RECALL YOUR HEADING)` **ryh**

Returns the current heading

`(ASK ?an-object RECALL YOUR XCOR)` **ryx**

Returns the x coordinate

`(ASK ?an-object RECALL YOUR YCOR)` **ryy**

Returns the y coordinate

Just as with the turtle commands it is possible to have a coordinate message happen gradually. The relative ones (Delx Dely Delxy) use the variable "speed" just as "forward" and "back" do. The absolute ones (Setx Sety Setxy Setturtle Setheading) are more unusual. While the recipients of these messages also use "speed" and "rotational-speed" where appropriate, they cause odd behavior sometimes. An object is defined to move towards its goal (the desired coordinates and heading) as much as its speeds allow and then if the goal is not reached to plan to do the original message all over again on the next tick. This can cause some interesting conflicts if you have an object trying to get to a particular state and while that is in progress have it also receive some "forward" or "right" messages throwing it off course. The gradual coordinate messages are:

`(ASK ?an-object GRADUALLY DELX ?delta-x)` **gdelx ?**

`(ASK ?an-object GRADUALLY DELY ?delta-y)` **gdely ?**

`(ASK ?an-object GRADUALLY DELXY ?delta-x ?delta-y)` **gdelxy ?**

```
(ASK ?an-object GRADUALLY SETTURTLE ?new-state)                    gsett ?
(ASK ?an-object GRADUALLY SETHEADING ?new-heading)                  gsh ?
(ASK ?an-object GRADUALLY SETX ?new-x)                             gsetx ?
(ASK ?an-object GRADUALLY SETY ?new-y)                             gsety ?
(ASK ?an-object GRADUALLY SETXY ?new-x ?new-y)                     gsetxy ?
```

Growing and Shrinking

All objects can change their size. Just as with going forward or turning an object can grow gradually or all at once.

```
(ASK ?an-object GROW ?amount)
```

This is the same as the messages of the form `(INCREMENT YOUR SIZE BY ?amount)`. If the object is currently visible it will disappear and reappear bigger (if "amount" is positive).

```
(ASK ?an-object SHRINK ?amount)
```

This is the same as a grow message of the negative of the "amount".

```
(ASK ?an-object GRADUALLY GROW ?amount)                             gg ?
```

```
(ASK ?an-object GRADUALLY SHRINK ?amount)                           gs ?
```

This works a lot like the gradually messages for going forward, turning right and the like. The object will grow (shrink) at most the amount of its "growth-speed" and if anything is left will plan to do it later.

Gradually Changing the Value of a Variable

`(ASK ?an-object GRADUALLY CHANGE YOUR ?variable TO ?new-value)` `gcy ? to ?`

This will gradually change any "variable" to a "new-value". The speed of the change is given by the variable (,variable speed) or assumed be 1 if none is provided. If you want Sally to slowly grow to 300 at a rate of 30 units per second (by default there is one tick per second) then type

```
(ask sally set your (size speed) to 30)
(ask sally plan next gradually change your size to 300) ;; plan to change size
(ask sally keep doing until nothing-more-to-do tick) ;; keep sending ticks until finished
```

`(ASK ?an-object GRADUALLY INCREMENT YOUR ?variable BY ?addition)` `giy ? by ?`

This is like the previous one and is defined to gradually change the "variable" to its current value plus the "addition". The only difference between the following two ways of making Sally grow is that grow uses "growth-speed" while "gradually increment your ... " uses (size speed).

```
(ask sally gradually grow 200)
(ask sally gradually increment your size by 200)
```

Pens

When an object moves it can leave a trail behind itself. It does this by falling back upon the TV Turtle. The commands that are currently available are pu, penup, pd, pendown, erd, eraserdown, xd, xordown, xu or xorup. They have the same meaning as described in the LLogo memo [Goldstein 1975].

Special Variables

We have already seen a few variables that objects treat specially such as "heading", "state", and "position". Also the arguments for drawing given in the "when drawing use ..." message are also special variables. They are defined to cause the object in question to disappear if visible and then to reappear with a new appearance. This is accomplished using the "actions-if-changing" variable demons described earlier and explains why setting the variables "size" or "angle" of a Poly causes it to change. Suppose Bob is a square and you want him to become a triangle of size 200 then just type

```
(ask bob change your angle to 120) ;; become a triangle
(ask bob change your size to 200)
```

There are a few other special variables associated with objects that you might want to set sometimes.

`after-show-action` *is a variable whose default value is* `nil` *and is abbreviated as* `asa`

This contains the action to be performed after the object has finished drawing itself. The turtle is at the center so that shading is possible here. So to shade in Sally with a checker pattern (for more about shading see the LLogo memo [Goldstein 1975]) you type

```
(ask sally change your after-show-action to (shade 'checker))
```

`after-hide-action` *is a variable whose default value is* `nil` *and is abbreviated as* `aha`

This describes the action to be taken after the object has erased itself. If you want Bob to print a message every time he hides then

```
(ask bob change your after-hide-action to (type '(try and find me now)))
```

`erasability` *is a variable whose default value is* `t` *and is abbreviated as* `eras`

Director has two ways to erase an object, either redraw it with an eraser or erase everything in the region of the object. Only if the object's erasability is nil is the latter action performed.

This is necessary if the appearance is shaded for example and is often faster if the drawing is complex. Sally's problem now that she is checkered is that she can't be erased by redrawing with an eraser so you should type

```
(ask sally change your erasability to nil)
```

center-offset *is a variable whose default value is* nil *and is abbreviated as* co

Director assumes that an object should be shown, turn, grow, etc. around its center. The center-offset is evaluated before these actions are taken to allow you to change this. If you want Bob to be a flower that turns (or grows) not at his center but his base then try

```
(ask flower make bob)
(ask bob change your center-offset to (forward (ask bob recall your size)))
(ask bob show)
(ask bob right 30) ;; notice the difference if you create another flower and have it turn
```

Colors

If you want to change the colors of an object you use can the "change your ..." message. If colors were mentioned in the "When drawing use" message then this will work automatically. The list of possible colors is in the Lisp variable ":colors" and others can be made as described in the LLogo memo. If you want to see a smooth transition from one set of colors to another you can use the following message.

```
(ASK ?an-object CHANGE YOUR COLORS TO ?colors IN ?number TICKS)      cyct ? in ? ticks
```

The number of colors before should be equal to those after. Each color is slowly changed to the color in the corresponding position in "colors". Only 1/number of the change will occur, the rest will be planned for later. Of course, if you are not running the color system then these colors will all look white, but internal variables can be inspected to see that indeed the color is being "changed".

(ASK ?an-object PREPARE TO MIX COLORS WITH ?other-colors) ptmcw ?

The number of "other-colors" should be the same as the current colors. This message sets up a variable called "color-mix" that controls the mix of the old colors with the "other-colors". If "color-mix" is set to 0.0 then the old colors appear, if it 1.0 then the new-colors, if it is .5 then they are mixed 50-50. This message is used to define the previous message as follows

```
(ask object
     do when receiving (change your colors to ?colors in ?number ticks)
     (script: (ask :self prepare to mix colors with ,colors) ;; prepare the mix
              (ask :self set your (color-mix speed) to ,(//$ (float number)))
              ;; reciprocal of number is what should be done each tick
              (ask :self gradually increment your color-mix by 1.0)))
              ;; on each tick color-mix is incremented by the (color-mix speed)
```

For example,

```
(ask sally change your colors to (red white blue) in 5 ticks)
```

is the same as typing

```
(ask sally prepare to mix colors with (red white blue))
(ask sally set your (color-mix speed) to .2)
(ask sally gradually increment your color-mix by 1.0) ;; it starts off at 0
```

Interpolation

Sometimes you want an object's shape to slowly change to another shape. In Director you can create an actor that is the *interpolation* between the appearance of two other actors.

(ASK ?an-object MAKE ?name INTERPOLATION TO ?another-object)

This returns an actor named "name" that is the interpolation between "an-object" and "another-object". This actor is an object that you can tell to grow, turn or whatever. It has associated with it a special variable called "amount". Amount is initially .5 which means that the appearance should be exactly between the two appearances. 0.0 will make it look like "an-object"

and 1.0 the appearance of "another-object". Very interesting results occur if you try negative numbers or numbers greater than one.

```
(ASK ?an-object MAKE INTERPOLATION TO ?another-object)                    mitx ?
```

Same as the previous one only Director picks a name of the interpolation for you. To make a simple movie of a circle becoming a square try the following

```
(ask poly make circle)
(ask circle set your angle to 10) ;; will look like a circle but will really be a 36-agon
(ask poly make square)
(ask square set your angle to 90)
(ask circle make circle-to-square interpolation to square) ;; make the interpolation actor
(ask circle-to-square set your amount to 0.0) ;; start off looking like a circle
(ask circle-to-square set your (amount speed) to .05)
(ask circle-to-square gradually increment your amount by 1.0)
(ask circle-to-square show) ;; needs to be visible if we're going to make a movie of it
(ask movie make cts-movie) ;; movies are described in a later section
(ask cts-movie film the next 20 ticks) ;; send out 20 ticks recording as you go
(ask cts-movie project) ;; filming is over so project yourself
```

This transition will be linear. If you wanted the rate of change to increase just enter the following before running the movie.

```
(ask circle-to-square set your ((amount speed) speed) to .01)
; the amount speed itself has a speed, now (ie the acceleration)
(ask circle-to-square gradually change your (amount speed) to .25)
```

Treating Objects as Turtles

It is possible to have an object behave as a turtle and run turtle procedures. To do this use the following message pattern

```
(ASK ?an-object RUN %action)
```

"Action" can be any turtle command, procedure, or sequence of them beginning with the

word "script:". If you want an object named Sam to go forward 100, turn right 45, and then follow the course of a circle then you could type:

```
(ask sam pendown) ;; to see his trail
(ask sam run script: (forward 100) (right 45) (draw-poly 10 10))
```

where draw-poly is a Lisp TV Turtle program.

Appearance Definition Using Instant Turtle

Another way of creating appearances for objects is by using a mode called "instant turtle". It is entered by typing (instant) and is exited by either type "q" or control-g. The idea is to enable you to "draw" on the screen by having the turtle move to your every key stroke. Most single letters cause the turtle do something or to define the current image as either an actor, procedure or the definition of single character. Numbers are given to it by typing them before the letter command. If no numbers are given then the numbers last given to that letter are assumed (or 1 if it is the first time that letter is used). The mode is self-documenting just type

```
(instant)
Welcome to Instant Turtle ;; And it will will welcome you when it is ready
? : . ou type ? and then any letter or an * for help on all commands
```

and each character will describe itself. Note especially the "n" command for naming the picture as an actor.

Non Standard Appearances

The default appearance handler of objects assumes that the Lisp program given in the "When drawing use" message consists of turtle commands such as forward, right, left, and back. If you want a special kind of appearance you need to define an appearance handler that takes at least two messages "Display" and "Erase". If you want even less help from the system then you can define your actor to accept "Hide" and "Show" messages but this is not recommended. Suppose you want an actor that displays text then you could define it as follows:

```
(define text object
 (set your text to |No text given|) ;; provide a default text
 (set your font to tvfont) ;; default font is this one
 (set your appearance-variables to (text font));; needed only for movies
 (do when receiving
     ({or display erase}) ;; if the messages is either display or erase
     (penup)
     (setturtle (ask :self recall your state)) ;; move the turtle to state
     (fontprinc (ask :self recall your text) ;; print the text in the font
                (ask :self recall your font)))) ;; prints if blank, otherwise erases
```

This is the minimum needed to define a text of any font, or position. You might want the screen to reflect changes in the variables "text" and "font". The "actions-if-changing" variables (described in the earlier section on demons for variables) can be used for this as follows

```
(ask text add (script: (ask screen wipe) :new-value)
      ;; can't erase old appearance easily so erase everything and redraw it all
      to your list of (text actions-if-changing)) ;; if text is changed
(ask text add (script: (ask screen wipe) :new-value)
      to your list of (font actions-if-changing))
```

We can now use the new text definition.

```
(ask text make label)
(ask label show)
(ask label change your text to |Here I am|) ;; So it becomes the words here I am
(ask label forward 200) ;; to move it forward
```

The "text" actor in Director is defined as described here with the ability to display the text in various fonts. The fonts have to be made by "windowize" which is described in ai:libdoc;fwmake kenl. If you want you can use a font called tr18 (about 3 times bigger than normal) and it will automatically be loaded in. Fonts are loaded in automatically if the font name has a "font-autoload-file" property. So if you type

```
(ask label change your font to tr18)
```

the text will be displayed in that font.

V. The Screen

The Screen actor provides the interface between the world of actors and the TV Turtle display area. There is currently only one "screen" though the system could be extended to have multiple screens. To see them on different physical displays is another question. Much of what Screen does you need not bother with. The messages of some use follow.

(ASK SCREEN SILENT RUNNING) sr

This makes the Screen "pretend" to do what its told but not to show anything on the TV. This is useful mostly for making movies, but also if one wants to do many things and then see the final result. For example, if it takes a while to draw an object you may not want to see it erase and redraw as you tell it to go forward, grow and turn. You could always hide the object first but if there are many objects then it is easier to use this. If, while Director is running, you want the Screen to run silently you can type control a at any time.

(ASK SCREEN NORMAL RUNNING) nr

This tells the Screen to stop running silently and also to wipe as described next. This message can be sent to Screen by simply typing control r.

(ASK SCREEN WIPE)

This wipes off anything from the Screen and then redraws any objects that should be visible. This is useful if a message or something messed up your display area.

(ASK SCREEN CLEARSCREEN) cs

This tells all the actors to hide, so that the screen becomes blank.

`height` *is a variable whose default value is* 200

`width` *is a variable whose default value is* 550

The height and width of the screen is controlled by its variables "height" and "width". If you want a square screen 400 big then type

```
(ask screen set your height to 400)
(ask screen set your width to 400)
```

`visible-objects` *is a variable whose default value is* `nil` *and is abbreviated as* `vo`

This is a list that Screen keeps of actors it thinks should be visible. If you forget the name of someone on the screen then look at this variable.

VI. Universe

Universes (instances of **Universe**) are the actors responsible for knowing who wants ticks messages. Any actor that has planned anything has told its universe that it has something to do. (The "plan ..." messages handle this and all actors inherit these from **Something**.) The only thing that you need ask of a universe is to run when you want all the planned activities to occur. Each actor is asked for its "universe" when planning and unless told otherwise inherits from **Something** one called "default-universe". Telling a universe to run is not recommended when the display is involved instead **Movie** should be asked to "film". (Movies are described in the next section.)

`(ASK ?a-universe RUN FOR ?length TICKS)`

Send tick messages to each actor with something planned "length" number of times. So to run through a scene with Sally and Sue both growing you could type

```
(ask sally plan next gradually grow 300)
(ask sue plan after 2 ticks gradually grow 200)
(ask sally plan after 2 ticks print memory) ;; to see what's happening to Sally
(ask sally ask your universe to run for 4 ticks)
;; unless told otherwise Sue and Sally live in the same universe
```

`actors-to-run-next` *is a variable whose default value is* `nil` *and is abbreviated as* `atrn`

This variable is kept by a universe and should be a list of all actors in that universe with anything planned.

VII. Movie

Movie can also be told to run and its major difference from **Universe** is that movies manage to remember any changes to the screen. Movies can then be asked to play back at a speed that is typically much faster than when first created.

`(ASK ?a-movie FILM THE NEXT ?film-length TICKS)` `ftn ? ticks`

This is similar to the "run for ..." message for instances of **Universe**, however movies (instances of **Movie**) also record what's happening to the display. To create a movie named **Fantasia** 12 ticks long one need only type

```
(ask movie make fantasia)
(ask fantasia film the next 12 ticks)
```

All objects that are currently on the screen or plan to appear during the next 12 ticks will be in the movie.

`frames-per-second` *is a variable whose default value is* 1 *and is abbreviated as* `fps`

Sometimes you may want to see the same movie but with more frames and less change between frames. If you used "gradually" commands described above then you can control the number of frames per second. Remember that the speeds of gradually commands are in units per second. If the display were fast enough setting the number of frames per second to 20 or 30 and projecting at that rate would make the movement very smooth. To project many frames per second see the messages described later in this section. If you set Fantasia's frames-per-second to 4 and had it run for 48 ticks instead then every fourth frame will be the same as before and the others will show a smooth transition between the frames. You can think of frames-per-second as the speed with which the "camera" shoots the action. The default value for frames-per-second is 1.

`ticks-per-frame` *is a variable whose default value is* 1 *and is abbreviated as* `tpf`

If you want to film just every 5th tick then set the movie's "ticks-per-frame" to 5. This is

primarily useful if you want a tick to be a small unit for accuracy and yet don't want to see or record every tick.

(ASK ?a-movie FILM SECRETLY THE NEXT ?length TICKS) **fstn ? ticks**

This does the same as the previous message in that all changes to Screen are recorded except here they are not displayed. This is useful if you want to save the time of displaying changes on the screen or to free the terminal to do something else (e.g. edit a file) while the movie is being computed.

There is a wide selection of different messages asking a movie to display itself. They are:

(ASK ?a-movie PROJECT)

; *Show all the frames from the start not skipping any*

(ASK ?a-movie PROJECT FRAMES ?begin TO ?end SHOWING EVERY ?so-many) **pfs ? to se ?**

; *Show from frame number BEGIN to END skipping every SO-MANY frames.*

(ASK ?a-movie PROJECT FRAMES ?begin TO ?end) **pj ? to ?**

; *assume that no frames should be skipped*

(ASK ?a-movie PROJECT STARTING AT FRAME ?begin SHOWING EVERY ?so-many) **psat ? se ?**

; *show until the end of the movie from BEGIN showing every SO-MANY frames*

(ASK ?a-movie PROJECT SHOWING EVERY ?so-many) **pse ?**

; *starts at the beginning and goes to the end showing every SO-MANY frame*

(ASK ?a-movie PROJECT FRAME ?number) **pf ?**

; *just show that one frame*

speed *is a variable whose default value is* 99999

Movies have a speed which indicates how many frames per second should be displayed. Unfortunately the computer seldom can show more than a small number per second. The speed

may be less than one if you want very slow motion. If the machine can not display frames as fast as indicated (for example if the speed is the default of 99999) then it will just show them "as fast as the it can".

new-frame-action *is a variable whose default value is* (clearscreen) *and is abbreviated as* nfa

Another variable associated with movies is called the "new-frame-action". This provides instructions as to how to make the transition between frames. The default is (CLEARSCREEN) which just clears everything off the screen. The value of this atom is evaluated so if it is NIL then nothing will happen and you will see all the frames superimposed on the screen. One useful value is "erase-old" which redraws the frame with the eraserdown. Sometimes this is faster.

(ASK ?a-movie SMART COMPILE %file-name) **sc %**

This message will create a file of Lisp code that can then be compiled. The resulting movie projects the same as before but can run faster and be saved. To run the movie, load it into your Director and call the function "PROJECTED-COMPILED-MOVIE" (abbreviated PCM). If you call it with a '? it will print out a description of the arguments it expects and their defaults. So to save the movie My-first-film as ffilm >, compile it and then run it do the following:

```
(ask my-first-film smart compile ffilm >)
;; put the Lisp translation of my-first-film in the file ffilm > on your directory
^z  ;; Leave Director
:qcompl ffilm > ;; compile the film if you want it to run a little faster
direct^h ;; after compilation is finished return to Director
(load 'ffilm)  ;; load the compiled movie into your Lisp
(pcm '?) ;; to see the defaults
(pcm) ;; to project it using the defaults
```

I contemplate extending Movie so that while projecting you can have the "projector" zoom, pan, or other filmic effects. Type "(Ask Movie help)" to find out about this. Also Scripts are planned which will save the state of all the plans. This way you can save the script, run the movie, restore the script and then modify it.

VIII. A Big Example

Suppose we want to write a space war in Director. First we will want to define space ships, suns, and gravity. One way to do this is to associate with each physical object another object corresponding to its velocity. The velocity actors have their own position which corresponds to the magnitude and direction of the velocity. On every tick each object's position is updated by turning it in the direction of its velocity and going forward the magnitude of its velocity. Also the velocity itself may be updated in a similar manner by the thrust of the ship or by the gravitational pull of the suns. This use of a turtle's position to represent the velocity vector is similar to the approach presented in [Abelson 1975]. First we define physical objects that will include the space ships and the suns. Then we define the gravitational field to apply the forces between the objects to their velocities.

```
; this file is a test of Director for doing orbital physics
(include |ai:ken;declare >|) ;; this is needed only if I am compiling the following
(define physical-object object
 ;; make physical-object as a kind of object and send it the following messages
 (set your mass to 10) ;; the default mass
 (extend behavior when receiving (make ?instance) by doing
         ;; this enables me to extend the normal behavior of make ?
         (ask :self make ,instance) ;; create the object as normal
         (ask velocity make (velocity-of ,instance)) ;; make a velocity for object
         instance) ;; return the newly created instance
 (do when receiving (update your state)
    ;; when I get a message asking me to update my state
    (ask :self change your position to ;; I update my position by
              ;; by adding to my current position to the position of my velocity
              ,(position-sum (ask :self recall your position)
                             (ask (velocity-of ,:self) recall your position)))
    ;; I ask the gravitational field at my location to change my velocity
    (ask gravitational-field
         apply gravitational forces at
         ,(ask :self recall your position) to (velocity-of ,:self)))
(do when receiving (yield pull at ?place)
     ;; to determine the gravitational pull at the place (G=1 in our units)
     (quotient (ask :self recall your mass) ;; take my mass
               (square (ask :self yield distance to ,place))
               ;; divide by the square of my distance to the place to get force per second
               :frames-per-second ;; divide by this to get force per frame
               :ticks-per-frame))) ;; divide to get force per tick
```

```
(define gravitational-field something
 ;; I never move or appear on the screen so no need to be an object
 ;; make the field and send it the following messages
 (do when receiving (apply gravitational forces at ?place to ?velocity)
      ;; for me to apply the gravitational forces at a place to a velocity
      (ask :self exert pulls of ;; I exert the pulls of the masses not at the place
           ,(remove-any-at-place (ask :self recall your masses) place)
           on ,velocity at ,place)) ;; on the velocity
 (do when receiving
    (exert pulls of (?first-mass %rest-of-the-masses) on ?velocity at ?place)
     ;; to exert the gravitational pull at a point of some masses on a velocity
     (ask ,velocity move ,(ask ,first-mass yield pull at ,place) in direction
                    from ,place to ,(ask ,first-mass recall your position))
     ;; move towards the mass from the place by the pull (acceleration) at that place
     (ask :self exert pulls of ,rest-of-the-masses on ,velocity at ,place))
     ;; and let the rest of the masses exert themselves on the velocity
 (do when receiving (exert pulls of () on ? at ?)
     ;; when there are no more masses do nothing
     nil))

(define velocity object) ;; a velocity is an object so that is can move in velocity space

(define ship physical-object ;; now to define ships
 (do when receiving (thrust forward ?amount) ;; When I'm asked to thrust forward
     (ask (velocity-of ,:self)
          change your heading to ,(ask :self recall your heading))
     ;; I set the heading of my velocity to my own heading
     (ask (velocity-of ,:self) ;; and change my velocity by
          ;; having it go forward the quotient of the thrust and my mass
          forward ,(quotient amount (ask :self recall your mass))))
 (when drawing use draw-rocket of size))
 ;; and I am drawn by the Draw-rocket procedure applied to my size

(define sun physical-object ;; a sun is also a physical-object
 (set your angle to 10) ;; near enough to a circle (really a 36-agon)
 (set your mass to 100) ;; the default mass of a sun is 100
 (when drawing use draw-poly of size angle))
 ;; I am drawn using Draw-poly of my size and angle
```

```
(ask object ;; It is reasonable to give this ability to all Objects
     do when receiving (move ?amount towards ?position)
     (script:
      (let ((original-heading (ask :self recall your heading)))
           ;; save the original heading
           (ask :self change your heading to ;; change heading to face towards position
                ,(ask :self yield heading to ,position))
           (ask :self forward ,amount) ;; go forward the amount
           (ask :self set your heading to ,original-heading)))) ;; restore old heading

(ask object ;; to move parallel to a line between the positions given
     do when receiving
     (move ?amount in direction from ?begin-position to ?end-position)
     (script:
      (let ((original-heading (ask :self recall your heading)))
           (ask :self
                set your heading to ,(heading-from begin-position end-position))
           (ask :self forward ,amount)
           (ask :self set your heading to ,original-heading))))
```

Now to test out this program we make a short movie. One ship will pass by a double star system. We define this as follows

```
(define enterprise ship  ;; make a ship called the enterprise
   (set your state to (-1000 -400 90)) ;; put me at an interesting starting state
   (show) ;; show myself
   (plan next
         repeat (thrust forward 200) 5 times)) ;; turn on thrusters for the next 5 ticks

(define sun1 sun ;; make sun1
   (set your position to (0 200)) ;; start off 200 units above the screen center
   (ask (velocity-of sun1) to forward 25) ;; start me off with a velocity of 25 upwards
   (set your size to 100) ;; give it a size
   (set your mass to 7000000)) ;; and a big mass

(define sun2 sun ;; this one is a little smaller and less massive
   (ask (velocity-of sun2) to back 75)
   (set your position to (600 200 0)) ;; start off way to the right of Sun1
   (set your size to 60)
   (set your mass to 3000000))
```

```
(ask-each '(sun1 sun2 enterprise) plan next repeat (update your state) forever)
; on every tick send to each of the objects the message (update your state)

(ask sun1 set your after-show-action to (shade 'lighttexture))
 ;; so that it is shaded
(ask sun1 set your erasability to nil)
(ask sun1 show)

(ask sun2 set your after-show-action to (shade 'texture))
 ;; a darker texture for Sun2
(ask sun2 set your erasability to nil)
(ask sun2 show)
```

(ask gravitational-field set your masses to (sun1 sun2 enterprise));; *tell the field about the objects*

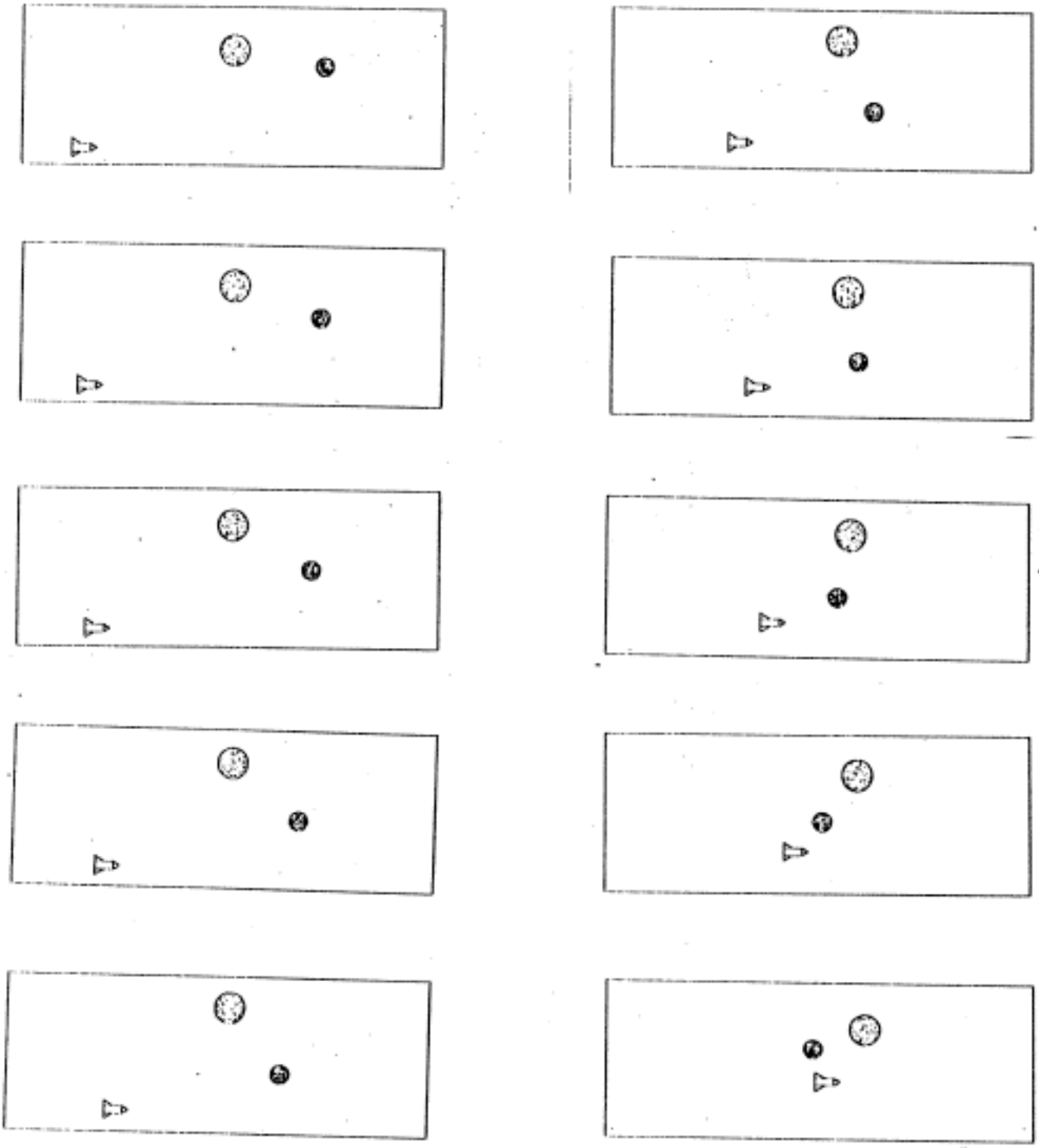
; *Everything is ready to go, so to test it we make a 10 tick movie. It can be seen in Figure 1.*

```
(define test-movie-1 movie
 (film the next 10 ticks);; finally make the movie
 (project)) ;; show the movie at default speed and order
```

I'll leave it as an exercise for you to finish the space war program. Unfortunately, it will be the slowest space war ever created. To speed things up very much[1] we could compile our code as described in the next section.

1. But probably still not enough to run on the AI machine.

IX. Figure - 1 A Test of the Space War Program



X. Compiling

There are some very fancy macros for compiling Director code into efficient Lisp so that it will run at a more reasonable speed. There are three major optimizations, one is to compile the patterns given in "do when receiving" messages and database inquiries. For example, the pattern

```
(BELIEVED: (COLOR-OF ?THING ?COLOR))
```

becomes something like the following

```
; this is the test part of the pattern
(LAMBDA (OBJECT)
 (AND OBJECT ;; it is not NIL
      (EQ (CAR OBJECT) 'BELIEVED:) ;; the first element is 'believed:
      (CADR OBJECT) ;; there is a second element
      (EQ (CAADR OBJECT) 'COLOR-OF) ;; the first element of the second is 'color-of
      (CDADR OBJECT) ;; there is a second element of the second element
      (CDDADR OBJECT) ;; there is a third element of the second element
      (NULL (CDR (CDDADR OBJECT))) ;; and there is no fourth element
      (NULL (CDDR OBJECT)))) ;; and no third element at all
; this is the part that finds the variables in the pattern (actually more complicated but...)
(LAMBDA (OBJECT) (LIST (CAR (CDDADR OBJECT)) (CADADR OBJECT)))
;; return a list of the color and the thing
```

This compilation of patterns happens throughout the system. The clause patterns and the data base inquiries are compiled. The clause patterns are also compiled into nested CONDs so that redundant tests are avoided. If you are curious about how this works create a simple actor and then ask it to save (see description of "save %file-name" earlier) and it will return the Lisp optimized version of itself. (Note that this does not make the actor become that more efficient. To do that you should type (eval (ask sue save)).)

A more important optimization is the compilation of message transmissions. The typical message will be tried by several different actors as the message gets passed along to each actor's parent. Also the predefined actors such as Something and Object have very many patterns to try out on the message. To skip all this computation a macro for transmission (Ask) figures out what action will be taken by the transmission and substitutes that action for the transmission. For

example, the transmission

```
(ask my-first-film project frames 3 to 6)
```

is replaced by

```
(LET ((:SELF '|MY-FIRST-FILM|)
      (:MESSAGE '(PROJECT FRAMES 3 TO 6)))
     (|(PROJECT FRAMES ?BEGIN TO ?END)MOVIE|))
```

where the Lisp function |(PROJECT FRAMES ?BEGIN TO ?END)MOVIE| corresponds to the action part of **Movie**'s clause whose pattern is "(project frames ?begin to ?end)". This optimization has its price in terms of flexibility. If **My-first-film** is told a new way to project this will have no affect upon the compiled code which contained the above transmission. In practice this is not too serious since changes to the way that **Movie** itself handles this kind of "project frames" message will affect compiled code. This scheme works fine even when the message and target contain commas indicating variables.

What has been described so far, is the default action for compilation when no other advice is available. It is possible to give advice as to how any particular message should be compiled.

```
(ASK ?anyone COMPILE MESSAGE ?message-form AS IN (ASK ?target-form %message)))
```

The compiler uses this message to compile all transmissions. The target-form and the message-form are from the actual transmission. The "message" is an example of the "message-form". This message is sent to the target when incrementally compiling (as described below). Otherwise the actor in which the transmission is embedded (or **Something** if there is none) is sent this message. The following is a simple example of this ability to give the compiler advice:

```
(ask (macros-of something) do when receiving
     ;; a parallel parent-offspring structure is kept beginning with the atom macros-of
     (compile message (recall your parent) as in (ask ?target-form %))
                      '(parent-of (actor-of ,target-form)))
```

Using this advice the transmission

```
(ask ,some-actor recall your parent)
```

will compile to

```
(PARENT-OF (ACTOR-OF SOME-ACTOR))
```

There are a few macros for defining "macro" receivers for an actor. Something and Object use them often so that messages that they receive are compiled efficiently. A corresponding normal receive is also created so that while running interpretively there is no overhead. For more details see me. The code produced in this manner is very close to the Lisp one might have written. For example the transmission

```
(ask something ask each of your offspring to print memory)
; is compiled into
(MAPCAR
 (FUNCTION (LAMBDA (ACTOR)
                   (FANCY-PRINT '(MEMORY)
                                (OR (GET-ACTOR ACTOR) (ACTOR-AUTOLOAD ACTOR NIL)))))
 (RECALL-VARIABLE 'SOMETHING 'OFFSPRING))
```

```
(ASK ?anyone COMPILE MESSAGE %message)
```

This provides an easy way to see how a message would be compiled. For example,

```
(ask sally compile message change your color to blue)
```

will return

```
(CHANGE-VALUE 'SALLY 'COLOR 'BLUE)
```

There are two ways in which you can use the compiler. One is incrementally and all you need do is to type (COMPILER-SWITCH T) and it is turned on. From then on all transmissions will be replaced by their expanded form and their old form prefaced by the atom "Expansion:". (COMPILER-SWITCH NIL) will not only prevent new forms from being made but will clean up old expansions.

To compile your Director code into machine language you need to include the Director

declaration file into your file. At the beginning of you file you should have:

```
(INCLUDE |AI:KEN;DECLARE >|)
```

Use the COMPLR when trying this. The "c" switch to the compiler has been added and means to "cleverly" compile the transmissions. If you create any actors without using DEFINE then to get them compiled you should have the form "(COMPILE-ACTORS)" at the end of your file.

XI. Odds and Ends

Debugging

For the most part debugging Director is like Lisp. The "trace" message described under the Something section is helpful. There are three kinds of break points. Lisp ones, "shouldnt-happen" ones which indicate a system bug or a "no-such-actor" or a "bad-message" break. These last two are often easy to recover from. The "no-such-actor" break can be returned from as follows:

```
(ask sue recall your parent)
;Warning from ASK that SUE who was asked (RECALL YOUR PARENT) is not an actor
;BKPT NO-SUCH-ACTOR
(?)  ;; a good way to get some help
(ask something make sue)
(return 'retry)
SOMETHING
```

If I had meant "sally" not "sue" then I could simply have replied (RETURN 'SALLY). Similarly the "bad-message" break point can be returned from. If you fix it so that the message is receivable then just return 'retry. If the message was wrong just return the right message. When you get an error try typing "(?)", it might be helpful.

Complete Description of Patterns

A pattern can be any list structure. If an atom in the pattern begins with a "?" then anything can be in the corresponding position in the message. In addition, the Lisp variable whose name follows the question mark becomes bound to the corresponding expression in the message. The character "%" is similar but will match the rest of the corresponding list structure, and so should be used only at the end of a list (which can be a sub-list of course). An expression surrounded by curly brackets {} is also treated specially. If it begins with the word "OR" then if any of the following expressions match the corresponding element in the message then the match

continues. If the word is "AND", then all the following expressions must match and all the bindings occur. Any other expression is evaluated and if it returns NIL the match fails. Such expressions can have an atom beginning with a "?" in it which becomes bound and then evaluated. For example, the pattern "{greaterp ?n 33}" will match any number greater than 33 and n will be bound to that number. An error will result if the corresponding element is not a number, however, so to be safe you should write the pattern as "{and {numberp ?} {greaterp ?n 33}}".

Global Variables

There are very few global variables, and even fewer worth knowing about. A few useful ones follow.

```
:SELF ;; this is bound to the actor who originally received the message
      ;; even if the message has been passed along to an ancestor to handle
:MESSAGE ;; the message in question

:TVRTLE-FILE-NAME
;; this is set to the normal tv turtle and should be reset if you want color for example

:PRINT-LOAD-MESSAGES ;; if NIL then no message is typed when a file is loaded

:MESSAGE-NOT-UNDERSTOOD
;; its value is a function of the target and message and it should
;; handles messages that are not understood. The default value puts you in a break-point

:ACTOR-NOT-DEFINED
;; a function of the target and message called when target is not defined

? ;; This a very handy variable, usually bound to some help so
  ;; try typing it whenever you need some help or (?) to print ? more nicely
```

Useful Lisp Functions and Macros

To make life a little easier there is the "define" macro for defining new actors. For example the definition of Poly is:

```
(DEFINE POLY OBJECT
        (SET YOUR ANGLE TO 60)
        (WHEN DRAWING USE DRAW-POLY OF (SIZE ANGLE)))
```

You type the name, its parent and then a list of messages to be sent to this newly created actor. There is a variant of define called "Define-or-add-to-actor" that differs only in that if the actor already exists it adds to it, while "Define" will clobber the old one.

The Lisp predicate "Exists?" of an actor returns NIL if the actor does not exist. "Actor-of" returns the internal representation of an actor if you are curious. There are also slight variants of the "ask" macro. They differ primarily in either how errors are handled or code is compiled. They are

```
ASK-IF-EXISTS  ;; just like ask except returns NIL if target does not exist
ASK-IF-UNDERSTOOD  ;; ask but returns NIL if message not understood
ASK-IF-CAN  ;; return NIL if either actor not defined or message not understood
UNCOMPILED-ASK  ;; normal ask but is not to be compiled
ASK-ALL  ;; ask all actors whose name matches the target
;; eg (ask-all (compiler-for ?) print memory) to get all the compilers to print
DO-AS-IF-YOU-WERE ;; has two targets, the first is the one to be used for compilation
;; for example, (do-as-if-you-were object ,body grow 100)
```

You can define your own abbreviations by using "define-abbreviation" which itself is abbreviated "da". If you are often asking Sally lots of things then to abbreviate "ask sally" just enter

```
(&da sal ask sally) ;; sal should be an abbreviation for ask sally
(&sal &pm) ;; Test it out, this is same as typing (ask sally print memory)
```

Discussion of Why Director is the Way it is

Director is the way it is mostly because I am a fan of object-oriented programming. For graphics its seems the most natural way of thinking about what happens on a display screen. For knowledge-based programming the association of a database with each actor and the inheritance mechanism are very handy. More importantly, the ability to arbitrarily mix data and procedure (and different forms of each) is a great convenience for representing complex knowledge. For knowledge-based programming the spectrum of flexibility is nicely spanned by the variables, the database, the variable demons, the database demons, and the message receiving -- all potentially usable by the same actor. These feature lead naturally to very modular programs with all the advantages that that brings.

The graphics in Director is very strongly influenced by turtles. Director's objects are generalized turtles that can change appearance and remember things in addition to the usual turtle actions. The pseudo-parallelism based upon ticks is in Director to ease the task of coordinating the actions of several different objects "at once". For more about ticks see [Kahn 1978] and among many [Papert 1971a], [Papert 1971b], [Goldstein 1975], [Goldstein 1976], and [Kahn 1977b] are good sources for learning more about turtles.

One design decision that may strike many as peculiar is the verbose style of programming in Director. The advantages are many. Programs need few comments since the code is itself close to English. Debugging is aided by the long, typically self-explanatory, messages that are traced or seen at error break points. While someone unfamiliar with Director could not write any code, compared to most languages there is a good chance such a person could read the code. The obvious objection to such long messages that it necessitates too much typing is just plain false. The abbreviation feature in Director cuts down drastically the amount of typing needed while retaining all the advantages since the abbreviation is expanded at read time. All of the

abbreviations are also available for use by Emacs's abbreviation package.[1] This has the added features of expanding as soon as a space or parenthesis is typed and that it requires no character prefix. (This is why all the abbreviations that would be an English word have an "x" at the end.) Another common objection to such English-looking syntax for programs is that users get confused and expect paraphrases that are valid in English to be valid in the computer language. This does not really apply to Director since the user learns about pattern matching very early on and that is the only mechanism for "parsing".

Director was built upon MacLisp so that I could fall back upon Lisp for memory management, a garbage collector, debuggers, readers, printers, an evaluator, and a compiler (to machine code). The running of Lisp at low levels of Director made it feasible to put in many costly features in "Ask", the basic transmission mechanism of Director. The overhead of a transmission is reasonable for events of the size typically dealt with by Director. A message-passing definition of factorial in Director, however, would be exceedingly slow (though it would compile pretty well). This inefficiency need not be the case with message-passing languages --- witness Smalltalk and Act 1. Ideally Director should have been built upon such a base to make things more consistent --- everything could then be an actor.

Discussion of Why One Might Want to Use Director

It's both fun and good for writing real programs (i.e. long complex movies or large AI programs).

---

1. Emacs is an excellent text editor developed at the MIT Artificial Intelligence Lab by Richard Stallman and others. The abbreviation package in Emacs was developed by Eugene C. Ciccarelli.

Getting Started

Couldn't be easier (well almost). You type:[1]

```
:DIRECT
```

After you see the message "Welcome to Director" you can type, for example,

```
(DEFINE STAR POLY (SET YOUR ANGLE TO 144) (SHOW))
```

and you should see a star appear on your TV. Type "(?)" for a little bit of help. If want to use only part of Director or are running it in color (in which case you should type (run-color) before doing anything on the screen) then type

```
:LISP KEN;DIRECT
```

and as you need things the appropriate files will be loaded. Have fun and report any problems to KEN@AI.

Acknowledgements

I wish to thank Carl Hewitt and Henry Lieberman for the help and support they have provided. Hal Abelson and Bill Kornfeld provided many important suggestions as to the form and content of this guide. Jerry Barber, as Director's first user, provided me with suggestions, discovered bugs and noticed missing features. Ira Goldstein provided many ideas when Director was first being developed. I wish I could acknowledge SmallTalk as a *direct* source of ideas, but unfortunately I was ignorant of its details and rediscovered many of their ideas on my own. The Learning Research Group at Xerox Parc pioneered in the development of object-oriented programming and graphics for which I am very indebted.

---

1. Don't do this if you are not on a TV or are planning to run Director in color.

XII. References

Bibliography

[Abelson 1975]
Abelson, H., DiSessa A., Rudolph L.
"Velocity Space and the Geometry of Planetary Orbits,"
American Journal of Physics, July 1975.

[Goldberg 1974]
Goldberg, A.
"Smalltalk and Kids -- Commentaries"
Learning Research Group, Xerox Palo Alto Research Center, 1974 Draft

[Goldberg 1976]
Goldberg, A., Kay A. editors
"Smalltalk-72 Instruction Manual"
The Learning Research Group, Xerox Palo Alto Research Center, March 1976

[Goldstein 1975]
Goldstein I., Lieberman H., Bochner H., Miller M.
"LLOGO: An Implementation of LOGO in LISP"
MIT-AI Memo 307, March 4, 1975

[Goldstein 1976a]
Godlstein, I., Abelson H., Bamberger J.,
"LOGO Progress Report 1973-1975"
MIT-AI Memo 356, March 1976

[Halas 1974]
ed. Halas, J.
Computer Animation
Hastings House, New York, 1974

[Hewitt 1975]
Hewitt C., Smith B.
"Towards a Programming Apprentice"
IEEE Transactions on Software Engineering SE-I, March 1975

[Kahn 1976]
Kahn, K.
"An Actor-Based Computer Animation Language", Proceedings of the SIGGRAPH/ACM Workshop on User-Oriented Design of Interactive Graphics Systems, October 14-15, 1976, ed. Treu, S., pp. 37-43
Revision of:
Kahn, K.
"An Actor-Based Computer Animation Language"
LOGO Working Paper 48, AI Working Paper 120, MIT, February 1976

[Kahn 1977a]
Kahn. K. "Three Interactions between AI and Education",
Machine Intelligence 8 Machine Representations of Knowledge,
eds. Elcock E. and Michie, D., Ellis Horwood Ltd. and John Wylie & Sons, 1977

[Kahn 1977b]
Kahn, K., Lieberman H.
"Computer Animation: Snow White's Dream Machine"
Technology Review, Volume 80, Number 1, October/November 1977, pp. 34-46

[Kahn 1978]
Kahn, K. Hewitt C.
"Dynamic Graphics using Quasi Parallelism"
MIT AI Memo 480, June 1978

[Kay 1977]
Kay, A., Goldberg A.
"Personal Dynamic Media"
Computer, IEEE, March 1977, v. 10, n. 3, pp 31-41

[Moon 1974]
Moon, D.
"MacLisp Reference Manual"
Project Mac MIT, April 1974

[Negroponte 1977]
Negroponte, N.
"The Return of the Sunday Painter or The Computer in the Visual Arts"
Future Impact of Computers and Information Processing
Dertouzos, M. and Moses J. ed.
in press 1978

[Newman 1973]
Newman W.
Principles of Interactive Computer Graphics
McGraw-Hill, New York, 1973

[Papert 1971a]
Papert S.
"Teaching Children Thinking"
MIT-AI Memo 247, October 1971

[Papert 1971b]
Papert S.
"Teaching Children To Be Mathematicians vs. Teaching About Mathematics"
MIT-AI Memo 249

[Reynolds 1978]
Reynolds C.
"Computer Animation in the World of Actors and Scripts"
Masters Thesis, MIT Department of Archecture, May 1978

[Smith 1975]
Smith B. and Hewitt C.
"A Plasma Primer"
MIT-AI Working Paper 92, October 1975

[Tilson 1976]
Tilson, M.
"Editing Computer Animated Film"
University of Toronto, Technical Report CSRG-66, January 1976

XIII. Index of Patterns

Patterns that **Object** *Handles*

Patterns that **Screen** *Handles*

Patterns that **Universe** *Handles*

Patterns that **Movie** *Handles*

XIV. Index of Special Variables

Variables Treated Specially by **Universe**

Variables Treated Specially by **Movie**