Massachusetts Institute of Technology
Artificial Intelligence Laboratory

6 December 1979

AI MEMO # 482B
(Revised)

DIRECTOR GUIDE

Kenneth M. Kahn

Abstract

Director is a programming language designed for dynamic graphics, artificial intelligence, and use by computer-naive people. 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, quasi-parallelism, many animation primitives, a parts/whole hierarchy 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, and a means of conveniently creating non-standard control structures. For use by 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 research described herein is being conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program.

Preface to the First Revision

Experimental languages like Director change at very fast speeds. In the year since the first version of this guide has come out, much of Director has been rewritten several times. The internal representation of actors and methods changed frequently, the "compilation" of actors even more so. Many new features were added for dealing with parts and wholes, constraints, method extensions, and activities. Other features such as "demons" have been eliminated, typically because they are now easy to obtain using newer more generic features. The patterns of many methods have changed and the names of many primitive actors have changed, typically to obtain a more consistent naming.

Preface to the Second Revision

Since the last version Director has not changed very much. The mechanism for extending methods has changed drastically. The workings of the Director compiler and the internal representation of actors has been simplified. A new experimental menu-oriented interface for Director has been implemented. This is expected to be very important for the ease of use of Director by children and computer-naive adults.

Director Guide

Section I. The General Idea 6

Section II. An Introductory Example 7

Section III. What Every Actor Can Do 9

- A. Creation and Destruction of Actors 10
- B. Defining the Behavior of Actors 13
- C. Printing 14
- D. Tracing 16
- E. Memory Messages 17
  - 1. Variables 17
  - 2. Relational Data Bases 22
- F. Plans and Pseudo-Parallelism 25
- G. Broadcasting Messages 35
- H. Variables that are Special to All Actors 37
- I. How to Extend the Behavior of a Method 39

Section IV. What Every Performer Can Do 45

- A. Creation and Appearance 45
- B. Showing and Hiding 48
- C. Moving and Turning 49
  - 1. Moving 49
  - 2. Turning 50
  - 3. Coordinate Messages 50
- D. Growing and Shrinking 51
- E. Parts and Wholes 52
- F. What The-Cast Does 59
- G. Logo Compatible Commands 60
- H. Pens 60
- I. Special Variables 61
- J. Colors 63
- K. Interpolation 64
- L. Appearance Definition Using Instant Turtle 65
- M. Non-Standard Appearances 66

Section V. What The Stage Does 68

Section VI. What Clocks Do 70

Section VII. What Movies Do 72

Section VIII. A Big Example 75

Section IX. Compiling 81

Section X. Odds and Ends 86

- A. Debugging 86
- B. Complete Description of Patterns 87
- C. Global Variables 88
- D. Useful Lisp Functions and Macros 89
- E. Why Director is the Way it is 90
- F. How Director Works 92
- G. Why One Might Want to Use Director 94
- H. Getting Started 94

Section XI. Bibliography 96

Section XII. Index of Patterns 99

Section XIII. Index of Special Variables 103

Table of Figures

Figure 1 Ask My-First-Film Project 8
Figure 2 Family Tree of Primitive Actors 9
Figure 3 Sally Doing It Sequentially 31
Figure 4 Sally Doing It Concurrently 31
Figure 5 A Face Looking Around 33
Figure 6 Moving Forward (Level 0) 42
Figure 7 Moving Forward (Level 1) 42
Figure 8 Moving Forward (Level 2) 42
Figure 9 Joe Drawing a Fifth Degree Snowflake 44
Figure 10 Ask Face Show All 53
Figure 11 Ask Face Grow 100 53
Figure 12 Ask Face Turn Right 15 54
Figure 13 Ask Mouth Grow by Factor 2 54
Figure 14 A Hexagon Transformed into 4 Hexagons 56
Figure 15 4 Hexagons Transformed into 16 58
Figure 16 64 Hexagons Transformed into 256 58
Figure 17 256 Hexagons Transformed into 1024 59
Figure 18 Ask Cts-Movie Project 65
Figure 19 A Test of the Space War Program 80

Section 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 designed to be an AI language. Each actor has rather sophisticated abilities including inheritance, a relational data base, method extension, a parts/whole hierarchy and a pseudo-parallel control and planning structure. The language is currently grafted upon MacLisp (or Lisp Machine Lisp), 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.

Section 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 on a Lisp Machine or on ITS without graphics read the "Getting Started" section). Start up the system by typing

```
:direct <carriage return>
|Welcome to Director Version # 89| ;; 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 pent turn left 90) ;; have it turn 90 degrees

(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 move 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 movie make my-first-film) ;; make a movie called "my-first-film"
(ask default-clock set your frames-per-second to 2) ;; have it run at 2 frames per second
;; if the computer were faster 30 might be nicer

(ask star plan next do at speed 25 move forward 100) ;; slowly move forward 100 steps
(ask star plan after 2 seconds grow 50) ;; grow 50 units 2 seconds into the movie
(ask star plan after 4 seconds turn right 18) ;; turn left 18 after 4 seconds
(ask pent plan next gradually grow 300) ;; now for the pentagon
(ask pent plan next do in 4 seconds move back 200);; take 4 seconds to move back 200 steps

(ask my-first-film film the next 4 seconds);; roll the cameras for the next 8 clock ticks
(ask my-first-film project);; you just saw the film being shot, now lets see it projected
```

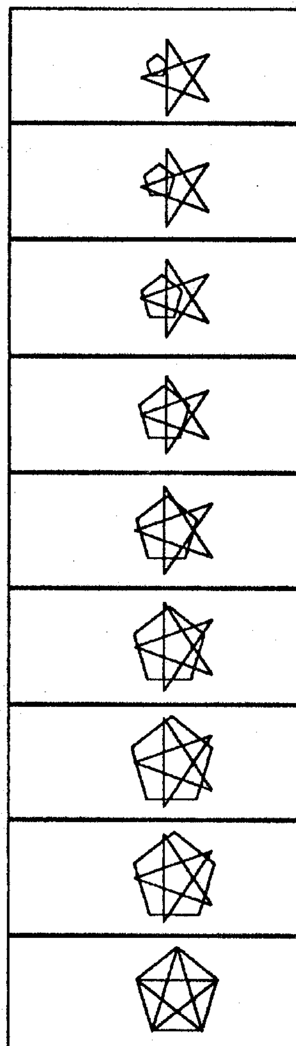

Figure 1 Ask My-First-Film Project

Section III What Every Actor Can Do

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 **Clock**), and can be told how to handle new messages. **Performer, Movie, The-Cast** and **Stage** 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 **Performer, Movie, Clock,** and **Stage** apply only to those actors and their descendants. The relationships of the actors initially present in Director is depicted in the following diagram.

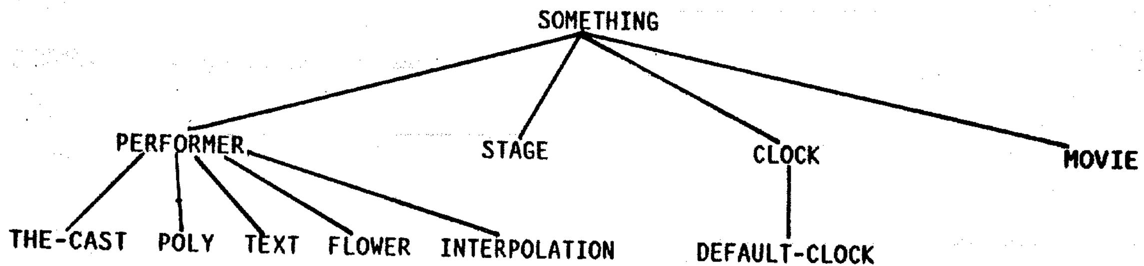

Figure 2 Family Tree of Primitive Actors

This guide is organized by the patterns of messages an actor can handle. Each section describes a primitive actor by listing the methods directly associated with it (i.e. those messages that the actor itself handles rather than passing the problem on up to its parent or more distant 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 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 (i.e. gets the value) "(pot stickers)". The "%" may be used only once at each level of the pattern.

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

Subsection A Creation and Destruction of Actors

```
(ASK ?anyone MAKE ?name)
```

This causes an instance of the receiver of the message to be made and given the "name". The name can be either a single word or a list. The newly created actor behaves 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 and 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)`

This method is also makes a new actor, but it does not have a name.[1] A new instance of the recipient is returned (while the previous methods returned the *name* of the new instance). Typically this message is used in such a way that the newly created actor is stored away. For example,

```
(ask sally set your secret-friend to ,(ask friend make))
```

The comma above means we want the *result* of "`(ask friend make)`" to be there. Without it the *text* "`(ask friend make)`" would be set to Sally's secret friend.

`(ASK ?anyone MAKE ?name IF ITS NOT ALREADY)`

make ? iina

If there already exists an actor with the same name then nothing happens otherwise one is made. Name is returned in either case.[2]

`(ASK ?anyone MAKE UP A NAME)`

muan

Sometimes you don't want to make up names for actors, but you want them to have names that you can refer to. This method is handy for this. If your program is making lots of flowers then you might want to make them as follows,

```
(ask flower make ,(ask flower make up a name))
```

1. One 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. Also there is no problem with name conflicts and nameless actors use up less memory.

2. "Iina" is an abbreviation for "if its not already" so you can just as well type (ask sam make samson •iina). The "•" is needed to signal Director that what follows is an abbreviation.

(ASK ?anyone MAKE UP AN UNINTERNED NAME) muaun

This behaves like the previous one only the name picked cannot be typed in. To talk to the resulting actor you must either save the result of this message in a Lisp variable or have some other actor store it (like with the previous (MAKE) message).

(ASK ?anyone CLONE)

An exact copy of the recipient in this message transmission is created and returned. The only differences between the newly created actor and the recipient is that the newly created actor has no name or offspring. Of course, afterwards they may be told different facts and how to handle messages. This creates an identical twin (a clone) while the "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 CLONE AND NAME IT ?name) cani ?

This is like the previous CLONE message except you provide a name for the newly created clone.

(ASK ?anyone MAKE SYNONYM ?name) ms ?

This does not create a new actor. Instead the recipient is given an alias, another name 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 an equivalent name as follows,

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

Subsection B Defining the Behavior of Actors

(ASK ?anyone DO WHEN RECEIVING ?pattern %actions) dwr ? %

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

Suppose you want an actor named "Sally" to type an answer to questions of the form "What is your <something-or-other>" Sally can be told how to handle such messages as follows,

```
(ask Sally do when receiving (what is your ?name-of-some-variable)
          ;; if Sally receives the words "what is your" followed by one other word
          ;; that last word we will temporarily call "name-of-some-variable"
          (type '(Oh its ,(ask myself recall your ,name-of-some-variable))))
```

If the comma had not preceded the variable "name-of-some-variable" then she would always type the message "oh its name-of-some-variable" To test it out we again ask Sally something, this time to respond to "good morning".

```
(ask sally what is your name)
; to which she responds
OH ITS SALLY
```

The variable "myself" is always bound to the actor that originally received the message. Using it in "(ask myself recall your ,name-of-some-variable)" is a good idea if you want the method to work ok for Sally's offspring as well as Sally herself. If you prefer you can use the word "I" for "ask myself" and "my" for "your" so the expression could have read "(I recall my ,name-of-some-variable)".

(ASK ?anyone DO ONCE WHEN RECEIVING ?pattern %actions) dowr ? %

This is the same as the previous method, except that the method created by this message self-destructs after its first use. In other words, the "actions" will only happen the *first* time a message comes in matching "pattern".

Subsection C Printing

(ASK ?anyone PRINT YOUR ?variable) py ?

This types out the value of the "variable" of the recipient. It is especially useful when planning as described later. For example, typing

```
(ask something print your clock) ;; will make
;SOMETHING's value of CLOCK is DEFAULT-CLOCK ;; appear on your console
```

(ASK ?anyone PRINT {or memory variables script database all}) ps OR pm OR pvs OR pdb

This causes the script or the memory of the recipient to be printed. The script is a description of each method of the actor. The memory is split up into "variables" and "database".

(ASK ?anyone PRINT {or memory variables script database all} ON FILE ?file-name)

This is like the previous message expect, in addition, you specify a file you want things printed out on. So to print the entire definition of Sue to a file called "sue file" type

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

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

(ASK ?anyone SAVE %file-names)

This message causes the recipient to print itself out in Lisp onto the "file-names". If any of the files already exists the actor 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 read the saved actors back into Director. If there are no "file-names", then it will return the Lisp form instead. 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
```

If another time 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 given then only those that match the pattern are printed. So, to learn about the messages that begin with the word "project" that a movie can handle just type

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

```
(ASK ?anyone RECALL METHOD FOR %sample-message)                    rmf %
```

This will return the first method that matches the "sample-message". For example, if you tell Sally,

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

```
(ASK ?anyone RECALL INTERNAL METHOD FOR %sample-message)           rimf %
```

Same as the previous message except the internal representation of the method is returned. You might use this if you are curious, but should not need to otherwise.

```
(ASK ?anyone REMOVE METHOD FOR %sample-message)
```

This removes the first method to catch the "sample-message" in the recipient. It will not affect its ancestors.

Subsection D Tracing

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

If any message is sent to the recipient that matches 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. If you want to see all the messages for Sally that begin with either grow or shrink, for example, you should ask her to trace by typing[1]

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

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

```
(ask sally trace ({or set recall} your size %) (break size-being-set-or-recalled))
```

If you want to trace every message that Sally 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. Actors that accomplish the tracing are all offspring of an actor named Tracer. To remove all traces from everyone just type "(ask tracer untrace all)".

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.

Subsection E 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 SET YOUR ?variable TO ?new-value)` **sy ? to ?**

A message matching this pattern causes the value of "variable" to be set (or 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

1. The returning of NIL for unbound variables is a very convenient default, especially for novices. This can be overridden so that it becomes an error as discussed in the section on extending methods. The default value is bound to "nothing-found" if you want to change it.

```
(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 returns the list of the values of all the variables that match the "variable-pattern". An actor using lists as variable names has 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
```

```
(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 (I set my ,variable to ;; "I" is just short-hand for "ask myself"
                   ,(plus (or (I recall my ,variable)
                              0) ;; In case it has no value yet
                          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 MULTIPLY YOUR ?variable BY ?factor) myx ? by ?

This message differs from the previous one only in that the current value of the "variable" is *multiplied* by the "factor".

(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 to 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 even if 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 Sam, then you can ask Sam

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

```
(ASK ?anyone CONSIDER ?list-of-names SYNONYMS)
```

This makes each of the names in the "list-of-names" equivalent for the recipient. All of the variable manipulating messages described above will behave the same if any of the names in the "list-of-names" are substituted for one another. One place where this is used is in the "gradually ..." message described later. For example, Performer considers the names (MOVE FORWARD SPEED) and (MOVE BACK SPEED) to be synonyms. For the descendants of the recipient, the names will be considered equivalent by default, but if they are explicitly set they will be different. For example,

```
(ask performer consider ((move forward speed) (move back speed)) synonyms)
(ask performer set your (move back speed) to 25)
(ask performer recall your (move forward speed))
25 ;; is returned
(ask performer make lobster) ;; make an offspring
(ask lobster set your (move forward speed) to 5)
(ask lobster recall your (move back speed))
25 ;; this is inherited from performer
;; this would not have been the case if lobster itself had the synonyms.
(ask lobster set your (move back speed) to 20)
(ask lobster recall your (move forward speed))
5 ;; still the same
```

```
(ASK ?anyone CONSIDER ?names COMPONENTS OF ?variable)
```

Sometimes you have a variable that is made up of "smaller" variables. At times you want to refer to the whole and at other times the parts. The parts/whole convention described in a later section is too expensive for such a simple use, especially since

Director uses Lisp's lists rather than have actors for them.[1] The turtle state of a performer, for example, is a list of its horizontal coordinate, its vertical coordinate, and its heading. The use of this message permits the following,

```
(ask Joe consider (x y heading) components of turtle-state)
(ask joe set your turtle-state to (10 20 90))
(ask joe recall your y)
20 ;; is returned
(ask joe set your heading to 45)
(ask joe recall your turtle-state)
(10 20 45) ;; is returned
```

(ASK ?anyone CONSTRAIN YOUR ?variable TO EQUAL ?function OF ?variable-2 OF ?other)

This method is for creating and maintaining a very limited class of constraints. It makes sure that if the "other's" value of the "variable-2" changes that the recipient's "variable" will become the "function" of the new value of the "variable-2". "Function" can be any Lisp function of one argument. Suppose that regardless of what happens, you want Sally to be twice as tall as Sam is wide. You could then type,

```
(ask sally constrain your height to equal double of width of sam)
(ask sam constrain your width to equal half of height of sally)
```

Where half and double are Lisp functions which could have been defined as follows,

```
(defun double (x)
       (plus x x))
(defun half (x)
       (times x .5))
```

Anticipated are new methods for describing more complex constraints. Use the "help" message to find out about them.

1. This is very important for efficiency, given the design of Director. Efficient lists implemented as actors are possible, however as demonstrated by Act 1.

`(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)` fy ?

If the recipient has such a "variable", it is forgotten (the actor is restored to the same condition as before the variable was created). Otherwise NIL is returned. This does not affect any variables whose value is inherited from an ancestor. "Forgetting" differs from setting the same variable to NIL in that it restores inheritance of the value from its ancestors.

`(ASK ?anyone FORGET EVERYTHING)` fe

Restores the recipient's memory to its time of birth. Methods and the data base associated with the actor are not affected.

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.

(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))
```

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 AN ITEM MATCHING ?pattern THEN %actions) raim ? then %

If an actor has memorized at least one thing that matches the "pattern" it will do whatever the "actions" are. So, if you type

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

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

(ASK ?anyone RECALL EACH ITEM MATCHING ?pattern THEN %actions) reim ? then %

The previous message isn'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 simpler, less general, "recall each item matching ..." message is provided. It does the "actions" for each item which matches the "pattern". If we changed the previous example to recall "each" item rather than "an" item we would get,

```
(ask sally recall each item matching (color ?thing blue)
          then (type '(i know that the ,thing is blue)))
I KNOW THAT THE OCEAN IS BLUE
I KNOW THAT THE SKY IS BLUE
```

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

This message collects into a list and returns all the forms that were memorized and match the "pattern". So if you type

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

It is defined in terms of the previous message as follows

```
(ask something do when receiving (collect items memorized matching ?pattern)
     (I recall each item matching (and ?the-item ,pattern)
        ;; recall items matching the "pattern" and return them
        then the-item))
```

This works because "recall each item matching" returns a list of the values returned by the "actions" taken on each matching item.

(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 methods) type

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

Subsection F 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 briefly described below and 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.

`(ASK ?anyone PLAN AFTER ?number ?units %action)` **pa ? ? %**

This requests that the message called "action" be sent after "number" "units". The units can be either ticks, seconds, or frames and are described below in the section on "clocks". 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 frames move forward 50),
(ask sam plan next move forward 50)
(ask sam plan after 2 frames grow 100)
```

Once all the plans have been made you can get an actor to do all that its planned for the next tick by asking it to `(TICK)`. If you ask an actor named **Default-clock** to `(TICK)` then all the actors with anything planned will be sent a tick. If you ask **Default-clock** to "`(RUN FOR 3 TICKS)`" then 3 ticks of action will happen.

(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 with 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 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 clock. 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 **Clock** below for more details.

(ASK ?anyone GRADUALLY %action ?amount) grad % ?

This method is the most crucial one for animation and simulation. Changes take place over time, and this method lets you express those changes that should take place gradually not abruptly. For example, to ask Sally to gradually increase her age type the following,

```
(ask Sally gradually increment your age by 20)
```

But what could this mean, unless you say how fast she should age? "Gradually" messages assume that the speed with which they do things is named (!,action speed), or in this

case (increment your age by speed).[1] The speed is in units per second, so if we wanted Sally to age 2 units per second then we should type,

```
(ask Sally set your (increment your age by speed) to 2)
```

When Sally gets a (TICK) message, she will do the following (assuming that, as is the default, there is one tick per second).

```
(ask Sally increment your age by 2) ;; Age by the speed
(ask Sally plan next gradually increment your age by 18) ;; plan next to age what's left
```

The scheme for "gradually" type messages works well when the change is additive, but what if its multiplicative? There are currently two kinds of "speeds" that an action can have: additive or multiplicative. The additive one is given by the list (ADD ?some-number) (or for convenience just a number) and the multiplicative one is given by (MULTIPLY-BY ?some-number). Suppose you wanted Sally to double in size, growing 5% per second, then you could type in the following,

```
(ask sally set your (multiply your size by speed) to (multiply-by 1.05))
;; grow at 5% per second
(ask sally gradually multiply your size by 2.0)
```

What if the action requires more than one number to control? How does this work then? Suppose you want Sally to gradually change her position by moving right 200 units and up 150. You could type

```
(ask sally plan next gradually move right 200)
(ask sally plan next gradually move up 150)
```

Alternatively if you think more in terms of "positions" (lists of horizontal and vertical coordinates), you could try the following.

```
(ask sally plan next gradually increment your position by (200 150))
```

The problem is how is Sally going to know enough to break it apart to add the list

---

1. The "!" in the description of the speed name means that the list that follows should be inserted into the list without its parenthesises. This applies generally to messages and quoted lists in Director. For example, typing "(let ((m '(print memory))) (ask sally !m))" is exactly the same as typing "(ask sally print memory)".

component-wise.[1] The solution is to tell Sally as follows

```
(ask Sally set your (components of increment your position by)
     to ((increment your xcor by) (increment your ycor by)))
```

In other words the variable (COMPONENTS OF !,action) is recalled and it should contain a list of actions which accomplish the original action.

(ASK ?anyone GRADUALLY SET YOUR ?variable to ?value) gsy ? to ?

The previous "gradually" message covers a large number of cases, but it is based upon the assumption that the action is the sort that you can do a little on a tick, and plan to do what's left over on the next tick until nothing is left over. If you want to gradually set the value of variable, however, there no clear notion of what "what's left over" means. What this method does is do a little (as determined by a speed associated with the actor) and then if that did not set the "variable" to the "value" it plans to try again on the next tick. The speed is called "(,variable speed)". For example,

```
(ask sally set your (size speed) to 50) ;; maximum change in size is 50 units per second
(ask sally gradually set your size to 300) ;; gradually go to a size of 300
```

This is equivalent to the following only if nothing else is happening concurrently to change Sally's size.

```
(ask sally set your (increment your size by speed) to 50);; increment at 50 units//second
(ask sally gradually
     increment your size by ,(difference 300 (ask sally recall your size)))
```

The first one will not give up until Sally's speed is 300, while the second way will stop when its added the difference of her current speed and 300.

1. If Director were thoroughly based upon actors, this would not be a problem because the actors representing lists of numbers would know how to add component-wise.

This message also works for variables whose value is a list, so long as the actor in question has a variable that indicates what the components of the variable are called. For example, if Sally's position is a list of her xcor and ycor then we could have her gradually move to the center as follows,

```
(ask sally set your (components of position) to (xcor ycor))
(ask sally gradually set your position to (0 0))
```

Interesting behavior results if concurrent with this you have Sally gradually moving around.

`(ASK ?anyone DO AT SPEED ?speed %action ?amount)` das ? % ?

The "gradually" methods described above look up the current value of the speed of the "action" on every tick. This enables you to change the speed as an actor's behaving for more complex behavior (such as acceleration). The disadvantages are that it is awkward to set and reset an actor's speed and occasionally an actor may be doing the same thing at different speeds (for example, the wind may be making the actor slowly turn back and forth while the actor is turning quickly chasing someone else). This method provides a means of overcoming these difficulties, at the price of being unable to change the speed. To ask Sally to move back 150 units at a speed of 20 units per second, just type

```
(ask Sally do at speed 20 move back 150)
```

`(ASK ?anyone DO IN ?number ?time-units %action ?amount)` do in ? ? % ?

This is like the previous method, where the speed is computed given the "amount" and the "number" of "time-units". The previous example, could equivalently been expressed as

```
(ask sally do in 30 frames move back 150)
```

assuming that there are 4 frames to a second (see the section on clocks).

(ASK ?anyone ASK ?another %message)

This one may seem kind of silly. Why ask someone to ask someone else something, like

```
(ask sally ask joe print memory)
```

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 instance, suppose there is an actor that is being influenced by two others. You anticipate that when one wins out the other's influence will disappear.

```
(ask fear ;; fear keeps trying to keep Marty at home
     gradually ask marty set your position to (0 0))
(ask desire ;; while "desire" tries to have him move forward 500 steps
     gradually ask marty move forward 500)
(ask marty ;; After 5 seconds Marty will overcome his fear
     plan after 5 seconds ask fear stop everything)
```

If after 5 seconds Marty has neither traveled the 500 steps nor returned back home, then "fear" will be stopped and Marty will travel whatever is left of his 500 step journey.

(ASK ?anyone SEQUENTIALLY %actions) se %

This does each of the "actions" one at a time waiting for them to finish if they are "gradually" or "plan" type messages. For example, to have Sally appear, slowly grow, wait a second and then fall over, type

```
(ask sally sequentially
      (show)
      (plan next gradually grow 300)
      (plan after 1 second gradually turn right 90))
```

```
(ASK ?anyone CONCURRENTLY %actions)                                    co %
```

This does each of the actions simultaneously.

The differences between the "concurrently" and the "sequentially" type messages are very important (unless all the actions are *instantaneous*, i.e. take less than 1 tick). If you are confused about the difference between them look at the difference between results of the previous example and the following one in the following figures

```
(ask sally concurrently ;; this is the only difference with the previous example
     (show)
     (plan next gradually grow 300)
     (plan after 1 second gradually turn right 90))
```

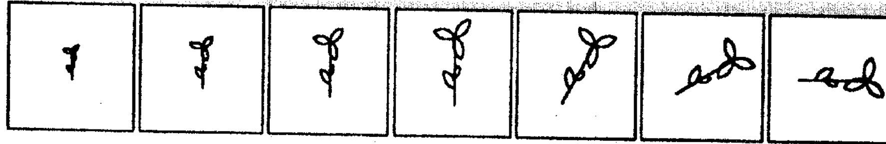

Figure 3 Sally Doing It Sequentially

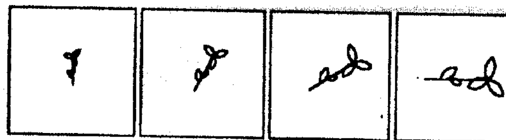

Figure 4 Sally Doing It Concurrently

Of course, the "concurrently" and "sequentially" messages can be included within each other. One of the test programs used to see if Director is working okay uses the "face" example included in the parts/whole section below. The entire action of the movie is described as follows.

```
(ask looking-around sequentially ;; look around by doing the following
     (ask looking-left ask eyes plan next gradually move left 50) ;; look left
     (plan after 1 second ask looking-right concurrently ;; a second after that
           ;; look right by doing the following concurrently
           (ask eyes gradually move right 100)
           (ask nose plan after .5 seconds gradually move right 40)
           (ask mouth plan after .5 seconds gradually move right 50))
     (ask the-surprise sequentially ;; when the above is finished do the following
          (ask scenery hide) ;; have the flower disappear
          (ask mouth plan next set your angle to 10))) ;; become a circle (its surprised)
```

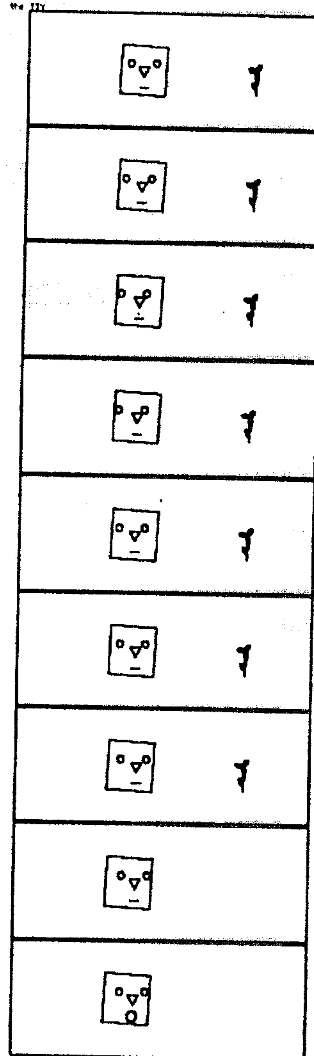

Figure 5 A Face Looking Around

The actors called "looking-around", "looking-left", "looking-right", and "the-surprise" are *activities* because they ask others to do things rather than do them themselves. This provides both structure and hooks to alter the events within an activity as a unit (for example, postpone their beginning). An actor that is planning to do something itself is its own activity.

(ASK ?anyone REPEAT ?message FOREVER)

This sends the message to the target immediately and on every subsequent tick. It is defined as follows:

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

(ASK ?anyone REPEAT ?message ?number TIMES)

If "number" is greater than 0, this method causes the "message" to be sent immediately. One tick after the "message" is finished the original message is sent again except that "number" is decremented by 1. If you wanted Sue to turn around every 5 seconds and do this 10 times just type

```
(ask sue repeat (plan after 5 seconds turn right 180) 10 times)
```

Suppose we want a "beacon" to blink on and off 10 times. We could enter the following

```
(ask beacon repeat (sequentially (show) (plan next hide)) 10 times)
```

(ASK ?anyone STOP EVERYTHING)

This will stop all the things that the recipient is doing, but it will not take affect until the beginning of the *next* tick. Since you have no control over the order of events within a tick this is not really a new restriction.

(ASK ?anyone WAIT FOR %signal) wf %

This is like the "stop everything" message above, except that the things that the recipient has planned to do are stored away and can be resumed by sending it the "signal". So to have Sally move in a circle, slowly growing, and to temporarily stop when her size is greater than 300 units type the following.

```
(ask sally  ;; first time size is set to a value over 300
     do once when receiving (set your size to {greaterp ?size 300})
     (continue-asking) ;; send the message on along to really set the size
     ;; see the section on extending methods
     (ask sally wait for go ahead) ;; signal will be "go ahead"
     size)
(ask sally concurrently ;; now have sally grow and move in a circle
     (gradually turn right 360)
     (gradually move forward 500)
     (gradually grow by factor 4))
```

You can send Sally the (GO AHEAD) message any time and she will finish growing and moving.

Subsection G 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 %message) ay ? %

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 recall your address)
```

Any message can optionally begin with the word "to" so if you prefer the last transmission is equivalent to

```
(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 %message) aeoy ? %

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 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 without names known to the reader) to print, for example, you just type

```
(ask something ask each of your descendants to print) ;; the "to" is optional
```

(ASK ?anyone BROADCAST TO YOUR ?others-name %message) bty ? %

This method causes the "message" to be sent to the recipient and the entire message (including the "broadcast to your...") is sent to each of the recipient's "others-name". We can use this, instead of "ask each of your descendants" described above, as follows,

```
(ask something broadcast to your offspring print)
```

The operational difference between this and the previous transmission is that this one will also send the print message to Something.

(ASK ?anyone DO OR BROADCAST TO YOUR ?others-name %message) **dobty ? %**

This method differs from the previous one in that it will send "message" only to those actors whose "others-name" is empty (i.e., value is NIL). The following two transmissions are equivalent (except for the order of broadcasting).

```
(ask something ask each of your childless-descendants to print memory)
(ask something do or broadcast to your offspring to print memory)
```

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

The recipient of this message 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 200.0)) grow 10)
```

Subsection H Variables that are Special to All Actors

There are a few variables that are treated specially. Among them are "siblings", "descendants", and "childless-descendants". There are methods for recalling them which generate the values on need. They return either the relatives indicated or NIL if there are none. A few variables are "private", i.e. do not follow the usual convention about being inheritable by the offspring. "Offspring", "name", "synonyms", "parts", and "things-to-do-next" are the common ones.

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

Only actors with names can be compiled or saved on file.

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 "set your parent to ...".

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 without names or that are not interned). This variable is not inherited of course.

private-variable-names *is a variable whose default value is* `(offspring things-to-do-next name synonyms whole parts)` *and is abbreviated as* `pvn`
This variable contains the list of names of variables that should be considered private, i.e. their value should *not* be inherited from the actor's ancestors. This variable is only inspected the first time a variable is set.

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.

clock *is a variable whose default value is* `default-clock`
The variable "clock" is used in planning and its value should always be a descendant of **Clock**. The clock is informed whenever an actor has anything planned.

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.

default-speed *is a variable whose default value is* `100.0` *and is abbreviated as* `ds`
The "gradually" type messages described above interrogates the actor in question for the speed of the action involved. If one is not provided, then a warning is typed and the

speed is assumed to be synonymous with the "default-speed". All speeds are in the units of the action per second.

Subsection I How to Extend the Behavior of a Method

Using the "do when receiving ..." messages you can extend the behavior of an actor by adding new methods. To extend the behavior of an existing method, you could just go and edit the text of the method and then evaluate it (with the Lisp variable "*replace-old-methods" being non-nil). But what if the method is a system primitive? What if the extension only makes sense to only a subset of the descendants of the "owner" of the message? What if you want your programs to temporarily or permanently modify the behavior of an actor? Suppose the extension is only a special case of the currently existing method, how would you extend the method then?

Over the history of Director, I have struggled with various "solutions" to this problem. At times there were transmission primitives for sending a message to someone but to start searching for methods somewhere other than the recipient. This was neither general enough nor very modular. Another idea was to have a new type of method that was invisible to itself when running, so the search for methods would skip it and find the "older" methods of its ancestors. One problem with this was that sometimes the body of this method would cause messages to be sent to the recipient's descendants to do things and they would not see this method. Another problem is that it made those methods non-recursive, since by definition if they invoked themselves they would be skipped. The most recent experiment was with a primitive called ASK-OLD which made the currently running method temporarily invisible only to the recipient of the message. This scheme was expensive to implement and I found it difficult to explain to people what it did.

The current Director primitive for extending methods is called CONTINUE-ASKING. It simply continues the search for an applicable method immediately following the current one. You can optionally change who the recipient is and more usefully what the message was. The three forms of this macro are:

```
(continue-asking) ;; doesn't change "myself
(continue-asking joe) ;; changes "myself
(continue-asking myself set your size to ,(times old-size .5)) ;; change the message
```

For example, suppose Jack does not want anyone to know that he is older than 39, but wants to be honest until he gets that old. Here are two ways this could be done using "continue-asking".

```
(ask Jack do when receiving (set your age to ?number) ;;; when Jack is told his age
     (script (continue-asking myself set your age to ,(min 39 number))))
; Another way to do this is
(ask Jack do when receiving (recall your age) ;; Jack will lie when asked for his age and
     (min 39 (continue-asking)));; answer with minimum of 39 and real age
```

Jack will even lie when asked to print.

There are many less frivolous uses of "continue-asking". It can be used as the basic building block of database demons (i.e. actions that should happen if an item is added, removed or needed from an actor's data base). It is important in implementing constraints and for helping the Stage maintain images of the latest appearances of the actors. It is also used in implementing many of the more sophisticated "planning" type messages.

In the first version of this report, there was a long section on "demons", actions that place when a variable is set or recalled, or an item added or taken from the data base. One problem that demons were used for was generating "virtual" items. We had a "table" with blocks on it and we wanted to be able to say that blocks were left of one another and for it to know which blocks were right of which others. We can use "continue-asking" instead of demons as follows,

```
(ask table do when receiving (recall all possible items matching (right-of ? ?))
  (union ;; combine the two lists, eliminating copies
   (I recall each item matching ;; then look for items that says that
      (left-of ?block-2 ?block-1) ;; the second block is left-of the first one
      then '(right-of ,block-1 ,block-2)) ;; if found pretend right-of item was there
   (continue-asking))) ;; and also include those that would have normally been found
```

To test this out type the following,

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

The unpleasantness of this scheme is that one needs to know about the message "recall all possible items matching ..." which Director always uses to get items which *might* match the pattern. If this bothers you just define a method for defining demons as follows,

```
(ask something do when receiving (if-needed ?pattern ?virtual-items)
 (script (I do when receiving (recall all possible items matching ,pattern)
          (union ,virtual-items (continue-asking)))))
```

and then type

```
(ask table if-needed (right-of ? ?)
     (I recall each item matching (left-of ?block-2 ?block-1)
        then '(right-of ,block-1 ,block-2)))
```

If you haven't guessed, making demons that go off when an item is added is quite easy. To make Table add a "right-of" item when it learns of a "left-of" item just type,

```
(ask table do when receiving (memorize (left-of ?block-1 ?block-2))
  (script
   (I memorize (right-of ,block-2 ,block-1) ;; add the "right-of" item
      if its not already) ;; memorize only if you don't know this already
   (continue-asking)))
```

The second transmission must be an "continue-asking" sort, otherwise the method would recurse infinitely. The first one has the suffix "if its not already" added in case you added the corresponding "demon" for memorizing "right-of" items, so the two methods do not invoke each other forever. It could be defined as follows, (ask something

```
do when receiving (memorize ?item if its not already)
 (script
  (cond ((not (I recall an item matching ,item then t))
         (I memorize ,item)))))
```

"Continue-asking" is used for many different things. One last example of its use follows. The problem is to draw "snowflake" or Koch curves [Mandelbrot 1977]. We want to replace each line with four line segments as follows.

Figure 6 Moving Forward (Level 0)

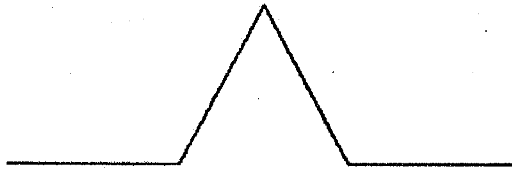

Figure 7 Moving Forward (Level 1)

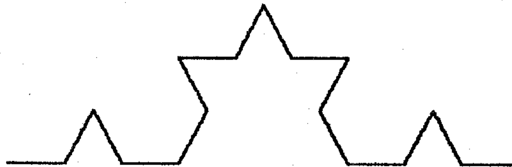

Figure 8 Moving Forward (Level 2)

Then we want to do that all over again any number of times. We can do this by repeatedly giving a performer a method for moving forward that uses the previous such method for drawing its lines. A program for this follows.

```
(ask performer make snowflake) ;; define a performer named "snowflake"

(ask snowflake do when receiving (snow ?size) snow
     ;; when asked to move in snowflake curve
 (script (repeat 3 ;; repeat the following three times
                 (I move forward ,size) ;; move forward whatever size called for
                 (I turn right 120)))) ;; turn right 120 degrees

(ask snowflake do when receiving (add ?level levels) ;; to add more levels of "recursion"
 (repeat level ;; repeat the number of levels the following
          (I do when receiving (move forward ?distance)
             ;; add methods for moving forward a distance
             (script
              (let ((distance (quotient distance 3.0))) ;; divide the distance by 3
                   (continue-asking myself move forward ,distance) ;; go forward the older way
                   (I turn left 60)
                   (continue-asking myself move forward ,distance)
                   (I turn right 120)
                   (continue-asking myself move forward ,distance)
                   (I turn left 60)
                   (continue-asking myself move forward ,distance))))))
```

Now to test it out we type the following,

```
(ask snowflake make joe)
(ask joe pen down) ;; so he leaves a trail as he moves
(ask joe add 5 levels) ;; make it a fifth degree snowflake
(ask joe snow 400) ;; draw it (see the result in following figure)
```

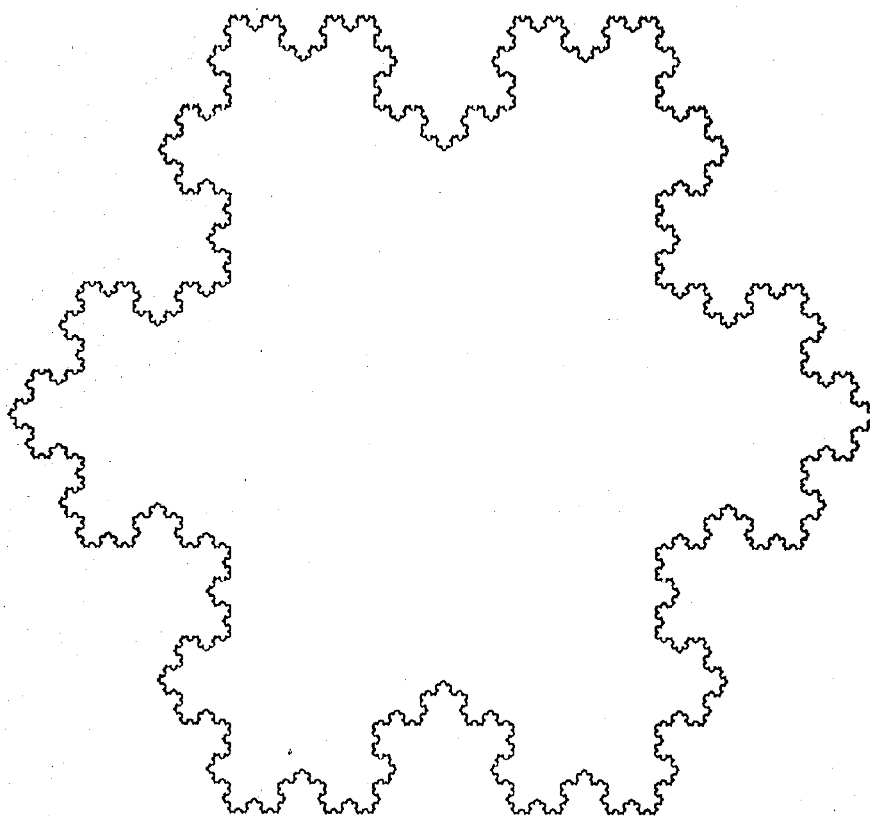

Figure 9 Joe Drawing a Fifth Degree Snowflake

Section IV What Every Performer Can Do

Performer is the top-level actor for those that can be seen on a display screen. Performers are much like Logo turtles only much more versatile. They handle messages to move, turn, show, hide, grow, change colors, and so on. A subset of its methods are compatible with the TV turtle.

Subsection A Creation and Appearance

Performers 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. You can do this explicitly using the "when drawing use" message described below or let Director do it for you using the "when drawing do" message below. Alternatively you can build up the appearance as a combination of simpler parts. Or you can use the "instant turtle" to procedurally "sketch" its appearance as described later. If the desired appearance is not easily describable by turtle commands (such as text) then you can supply methods for the reception of display and erase messages. This is described in the section on "non-standard" appearances.

(ASK ?a-performer WHEN DRAWING DO %messages) wdd %

Performers can be asked to move around. As you'll see later they can put down a "pen" so that they leave trails as they move. You can turn these trails into appearances of performers using this message. The trail that would have resulted from doing the "messages" becomes the appearance of the recipient. This method sometimes takes a while, but the appearance is remembered so that as the performer moves, turns and grows none of the "messages" need be sent.

Suppose you want to define squares this way, then you could type the following

```
(ask performer make square)
(ask square when drawing do
     (repeat (sequentially (move forward 100) ;; repeat 4 times moving forward and
                           (turn right 90)) ;; turning right 90 degrees
             4 times))
(ask square show) ;; to see if it worked
```

You needn't worry about the performer's pen unless you want "white" space in the appearance. If you wanted to define "half squares" whose sides are only half drawn then you could do it as follows

```
(define half-square performer ;; make a performer named "half-square"
        ;; and send it the following message
        (when drawing do
              (repeat (sequentially (pen up) ;; the first part will be invisible
                                    (move forward 100)
                                    (pen down) ;; the rest will be seen
                                    (move forward 100)
                                    (turn right 90))
                      4 times)))
```

If you remember how we got a performer named "Joe" to leave a "snowflake" trail then you should be able to guess how we could make Joe look like a snowflake. Suppose as part of a movie you want Joe to look like a snowflake as he draws one with one more level of detail.

```
(ask joe when drawing do (snow 100)) ;; look like a snowflake
(ask joe show) ;; show yourself
(ask joe add 1 levels) ;; add another level of detail (another "move forward" method)
(ask joe snow 400) ;; draw a that flake
```

(ASK ?a-performer WHEN DRAWING USE ?draw-procedure OF %draw-args) wdu ? of %

This informs "a-performer" 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.[1] "Draw-args" is a list of variables that are "Recall-able", i.e. variables known (either directly or through inheritance) by the actor in question. If these variables are changed and the Performer is currently being displayed, then its appearance will be updated. For example, to define a Performer that can appear as any polygon we could type

```
(define poly performer
       (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 performer 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 ((number-of-sides (sides-in-poly turnage) (1- number-of-sides)))
     ((= number-of-sides 0) no-value);; if back to the original heading, then return
     (forward distance) ;; each time go forward the distance
     (right turnage))) ;; and turn right the turnage

(defun sides-in-poly (turnage)
 ;; the number of sides in a poly is 360 divided by the gcd of the angle and 360
 (// 360 (gcd (round (float turnage)) 360)))
```

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 performer. 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

1. Thingup and Thingdown work even if the pen is an eraser or an "xor" pen.

are standardized so that any performer 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 as described under the "fuse your parts" method.

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

Subsection B Showing and Hiding

`(ASK ?a-performer SHOW)`

"A-performer" is shown if not already being shown. If it is made of parts each of its "visible-parts" are asked to display.

`(ASK ?a-performer HIDE)`

"A-performer" is hidden if currently visible. If it is made of parts then each of its "visible-parts" are asked to erase.

`(ASK ?a-performer SHOW ALL)`

"A-performer" asks each of its "parts" to show and notes that it is now visible.

`(ASK ?a-performer HIDE ALL)`

Asks all of its parts to hide and notes that it not visible.

Subsection C Moving and Turning

Just as in Logo there are at least two ways of changing a Performer'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).

There are two ways of moving or turning. A performer 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 performers do *not* cause them to move slowly when the message is received. Instead they move only a tick's worth (see previous discussion of "gradually") and plan to do the rest. In order to see the performer move gradually one should ask an instance of **Clock** or **Movie** to run for a number of ticks (see below).

Moving

`(ASK ?a-performer MOVE FORWARD ?amount)` **mf ?**

`(ASK ?a-performer MOVE BACK ?amount)` **mb ?**

Every performer has a "heading". If one receives either of these messages it will hide (if currently visible), move forward or back "amount" in the direction of its heading, and then reappear. So typing `(ask sally move forward 200)` causes Sally to disappear and then to appear 200 steps forward from the way she was facing (her heading). By default, there are 1000 steps from the top to the bottom of your screen. If she was hidden to start with she will move but you won't see her.

(ASK ?a-performer MOVE RIGHT ?amount) ml ?

(ASK ?a-performer MOVE LEFT ?amount) mr ?

These messages cause the recipient to move to *your* left or right the "amount". They ignore the recipient's heading.

(ASK ?a-performer MOVE UP ?amount) mu ?

(ASK ?a-performer MOVE DOWN ?amount) md ?

These make the recipient move up or down the "amount".

Turning

(ASK ?a-performer TURN RIGHT ?degrees) tr ?

This message causes "a-performer" to add "degrees" to its current heading. If it is visible, it will hide first, turn, and then reappear.

(ASK ?a-performer TURN LEFT ?degrees) tl ?

This is the same as the previous message except that it subtracts "degrees" from the current heading.

Coordinate Messages

Sometimes you might want to tell a performer where to go by giving the distance up or down and the distance left or right from 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. You can change the position of a performer by setting it directly using the normal "set your ?variable to ?value" message. The variable "position" is a list of the horizontal and vertical coordinates. "State" also contains the performer's "heading". The messages are

(ASK ?a-performer SET YOUR STATE TO (?xcor ?ycor ?heading)) **syst ?**

(ASK ?a-performer SET YOUR XCOR to ?value) **syxt ?**

(ASK ?a-performer SET YOUR YCOR to ?value) **syyt ?**

(ASK ?a-performer SET YOUR POSITION TO (?xcor ?ycor)) **sypt ?**

(ASK ?a-performer SET YOUR HEADING TO ?direction) **syht ?**

To change the "heading" of a performer you can set it directly using messages like "set your heading to 27". Heading is in degrees and a heading of 0 is straight up.

Subsection D Growing and Shrinking

All performers whose appearance was defined by a "when drawing use ..." message, a "when drawing do ..." message, or the "instant" mode can change their size.

(ASK ?a-performer GROW ?amount)

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

(ASK ?a-performer SHRINK ?amount)

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

(ASK ?a-performer {or SHRINK GROW} BY FACTOR ?number) **sbf OR gbf**

This is like the previous two messages except that the current size is *multiplied* by "factor". It is no different from messages of the form MULTIPLY YOUR SIZE BY ?number.

Subsection E Parts and Wholes

In addition to the child/parent hierarchy, Director maintains an optional parts/whole hierarchy. Currently many of the methods for dealing with parts and wholes are part of **Something**, but since only **Performer** make any use of them they are presented here as a feature of performers.

Groups of actors can be referred to as a whole while individual parts can still be addressed. A face, for example, can be made up of a mouth, a nose, eyes, and a head (an outline). The eyes in turn can be made up of a left eye and a right eye.

`(ASK ?a-performer SET YOUR WHOLE TO ?whole)` synt ?

This message makes the recipient become a part of the "whole". Its size, heading, and position are recomputed to be relative to the "whole". The "whole" has a list of "parts" and the recipient is added to it. If the whole's size, position, or heading changes then the parts are informed of the fact. Parts have a default way of responding to messages telling them of changes to their whole that makes the whole respond as a coherent unit. For example, if the face is asked to turn, each of its parts will turn and revolve around the face's center.

An example of defining such a composite actor follows. (This same face was used in the example in the section on plans and pseudo-parallelism.)

```
(define face performer)

        (define eyes performer
         (set your whole to face))

(define left-eye circle            (define right-eye circle
 (set your size to 10)              (set your size to 10)
 (set your whole to eyes)           (set your whole to eyes)
 (move left 40))                    (move right 40))

(ask eyes go forward 20)

        (define nose triangle
         (set your size to 20)
         (set your whole to face)
         (turn left 30))

        (define mouth horizontal-line
         (set your size to 15)
         (set your whole to face)
         (move down 50))

(define head square
 (set your whole to face))
```

So we can ask the "face" to show and move a few parts as follows.

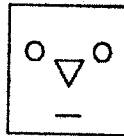

Figure 10 Ask Face Show All

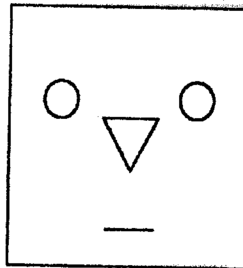

Figure 11 Ask Face Grow 100

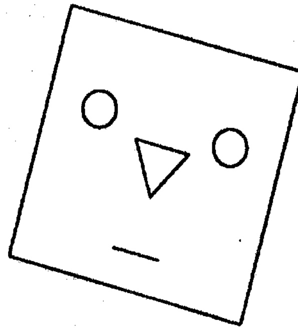

Figure 12 Ask Face Turn Right 15

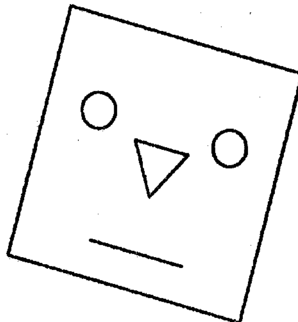

Figure 13 Ask Mouth Grow by Factor 2

(ASK ?a-performer ABSORB YOUR PARTS) ayp

This message makes all of the parts of the recipient nameless "disembodied" actors. They become accessible only as variables of the whole. If you "clone" a composite actor, how would Director know whose mouth you meant when you typed "(ask mouth grow by factor 2)"? This method is provided to avoid this problem. The "proper" way to clone a composite actor is to have it absorb its parts first as follows,

```
(ask face absorb your parts)
(ask face clone face-two)
```

The price you pay for using "absorb your parts" is the inconvenience of having to send messages via the "whole". To ask the left-eye to grow you have to type (ask face ask your eyes ask your left-eye grow by factor 2). This is the main reason that parts

aren't absorbed incrementally as they are declared part of the "whole".[1]

(ASK ?a-performer FUSE YOUR PARTS) fyp

After a performer has absorbed its parts, it can then "fuse" them together. Fusing destroys all the parts of the actor, but first changes the appearance of the actor to be all the appearances of the parts. The advantage of doing this is that often many few actors need exist. The disadvantage of it is that it both irreversible and you no longer can change any of the parts independently.

One use of this method is to create performers whose centers or orientations are not standard. Lines, for example, by default turn around their center and with this method you can construct one that turns around an end point or even a point far from the line. Most of these modifications could be (and indeed were in earlier versions of Director) accomplished by giving performers a few more special variables describing the center and orientation. This method is much more general, though admittedly more clumsy.

A simple example of the use of "fusing" is to make a hexagon which when pointed up has its top horizontal. This can be done by creating a normal hexagon (which has a vertex on top) turning it 30 degrees and making it part of another performer and then fusing it as follows.

```
(ask poly make hexagon) ;; make a normal hexagon
(ask hexagon show)
(ask hexagon turn right 30) ;; turn it so top is horizontal
(ask performer make flat-top-hexagon) ;; make a performer called "flat-top-hexagon"
(ask hexagon set your whole to flat-top-hexagon) ;; make the hexagon part of it
(ask flat-top-hexagon fuse your parts) ;; flat-top-hexagon's appearance is as desired
```

1. Perhaps this would be more acceptable if a "path name" syntax were incorporated so you could just type (ask left eye of eyes of face ...).

A problem suggested by Bill Gosper was to use such a hexagon to produce a complex fractal design. The hexagon is replicated three times and arranged as follows.

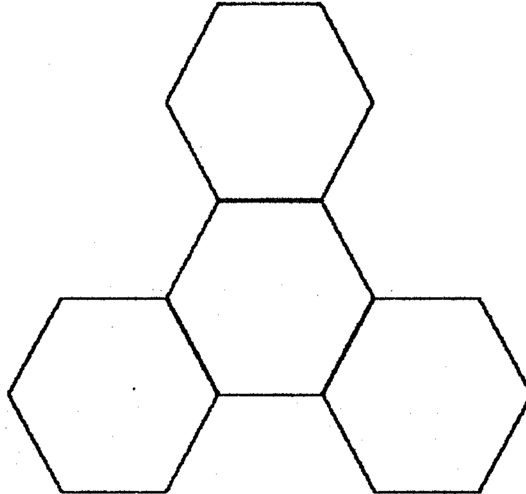

Figure 14 A Hexagon Transformed into 4 Hexagons

The result is then replicated and moved in the same manner producing the designs on the next page. A program for this follows.

```
(define four-hex performer) ;; define the performer that will do this

(ask four-hex do when receiving (clone and move)
 (script
  (I shrink by factor .5) ;; shrink down a half
  (let ((next-level (ask four-hex make ,(ask four-hex make up a name)))
         ;; create the performer that will be the next level of recursion
        (distance (*$ (I recall my size) #,(sqrt 3.0)))
        ;; figure out how far apart each hexagon should be
        (part-1 (I clone)) ;; make three copies
        (part-2 (I clone))
        (part-3 (I clone)))
       (ask ,part-1 move forward ,distance) ;; place the first one above the center
       (ask ,part-2 turn right 120)
       (ask ,part-2 move forward ,distance)
       (ask ,part-2 turn left 120) ;; put the second one to the right
       (ask ,part-3 turn left 120)
       (ask ,part-3 move forward ,distance)
       (ask ,part-3 turn right 120) ;; and the third one on the left
       (ask-each (,part-1 ,part-2 ,part-3 ,myself) set your whole to ,next-level)
       ;; make all the copies and the original part of the next level four-hex
       (ask ,next-level fuse your parts) ;; fuse all the part together
       next-level))) ;; return the name of the next level guy
```

Now we that we have defined "four-hex" we should use it. We could remake flat-top-hexagon the same as before but have its parent be "four-hex" or we can change it now and then clone it repeated as follows.

```
(ask flat-top-hexagon set your parent to four-hex) ;; always change parents with care
(ask flat-top-hexagon clone and move) ;; try it out
four-hex-1 ;; is returned
(ask four-hex-1 clone and move)
four-hex-2
(ask ,(ask four-hex-2 clone and move) clone and move) ;; do two levels at once
four-hex-4
(ask four-hex-4 clone and move)
```

At this point we have created only a few actors. Had we left out the "fuse your parts" transmission from the "clone and move" method we would have over a thousand.

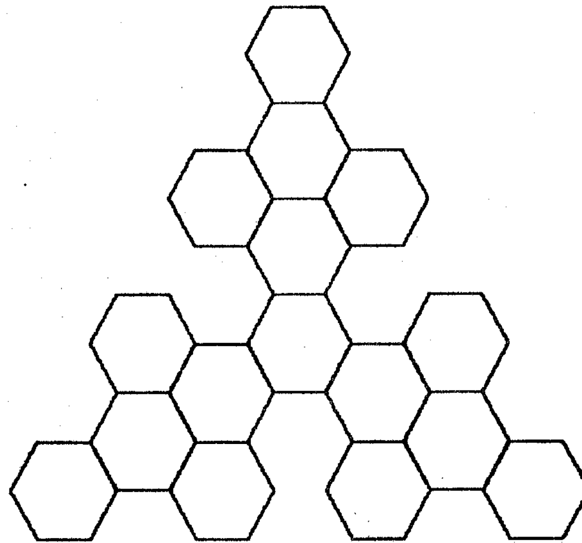

Figure 15 4 Hexagons Transformed into 16

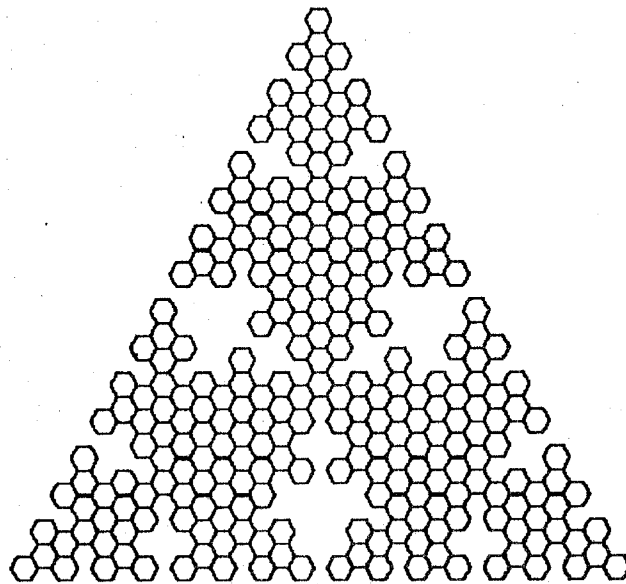

Figure 16 64 Hexagons Transformed into 256

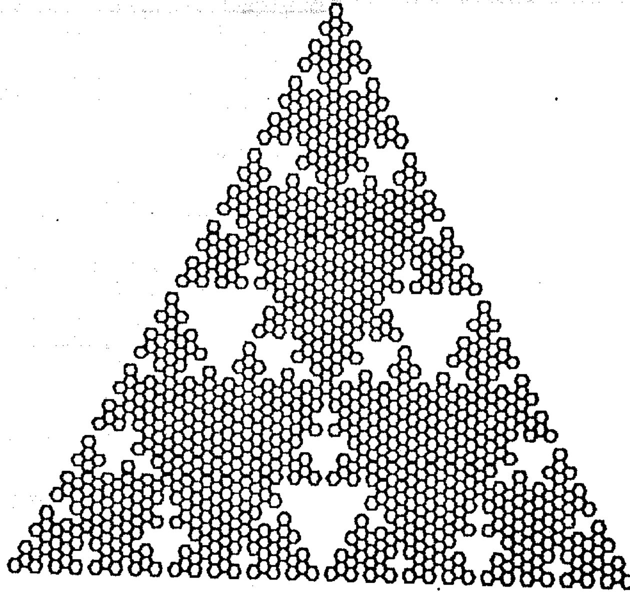

Figure 17 256 Hexagons Transformed into 1024

Subsection F What The-Cast Does

The-Cast is a performer with no special methods. It is however the default "whole" for all newly created performers. Unless you turn off this feature by setting Performer's default-whole to NIL (or some other actor), then every performer is either directly or indirectly a part of the The-Cast. This enables a simple means of achieving "cinemagraphic" effects such as zooms, pans, and the like. If you want to zoom and pan slowly to the left then just type,

```
(clip) ;; This is a TV Turtle command to make anyone who goes off the screen invisible
(ask The-Cast concurrently
     (gradually grow by factor 4.0)
     ;; zoom by a factor of 4 at The-Cast's "(grow by factor speed)"
     (do at speed 50 move right 500))
```

Subsection G Logo Compatible Commands

Director supports many of the Logo Turtle commands as described in [Goldstein 1975]. The Logo commands that performers can take are FORWARD, BACK, RIGHT, LEFT, SETXY, SETX, SETY, SETTURTLE, SETHEADING, DELXY, DELX, and DELY.

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

```
(ASK ?a-performer RUN ?action)
```

"Action" can be any turtle command, procedure, or sequence of them beginning with the word "script" (or "progn"). If you want a performer named Sam to go forward 100, turn right 45, and then follow the course of a circle then you could type:

```
(ask sam pen down) ;; 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.

Subsection H Pens

When a performer moves it can leave a trail behind itself. The type of trail depends upon the "pen-type" of the performer. The currently valid pen types are "normal" which draws regardless of whats underneath, "eraser" which erases what ever it passes over, and "xor" which draws if nothing is there already otherwise it erases what's there.

(ASK ?a-performer PEN DOWN) pd

This puts the pen of the performer's "pen-type" down.

(ASK ?a-performer PEN UP) pu

This moves the pen up, so the performer stops leaving a trail behind as it moves. This is the default for performers.

Subsection I Special Variables

We have already seen a few variables that performers 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 performer in question to disappear if visible and then to reappear with a new appearance. This is accomplished using the "continue-asking" type of transmission described in the section on extending methods. Suppose Bob is a square and you want him to become a triangle of size 200 then just type

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

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

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

Director has two ways to erase a performer, either redraw it with an eraser or erase everything in the region of the performer. Only if the performer'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.

default-whole *is a variable whose default value is* The-Cast

When a performer is made it is told that it is part of the "default-whole".

<u>standardize-size?</u> *is a variable whose default value is* t

The size of all performers whose appearance was defined by the "when drawing use ..." message is standardized to just fit within a circle of the same size. This feature can be turned off by setting a performer's "standardize-size?" to NIL.

<u>standardize-center?</u> *is a variable whose default value is* t

The default behavior is to center performers by averaging the positions of each vertex in its appearance.

<u>draw-mode</u> *is a variable whose default value is* (pendown)

<u>erase-mode</u> *is a variable whose default value is* (eraserdown)

Normally when the appearance of a performer is drawn on the screen it is done with the TV turtle's "pen" and erased with its "eraser". Sometimes, however, you want the performer to move over other stationary performers without having to redraw them as they become partially erased. A common solution to this is to both draw and erase using an "exclusive or" (xor) pen. You can do this by setting a performer's "draw-mode" and "erase-mode" to "(xordown)". This introduces a new problem however. If two performers overlap the overlapping areas will be white. The "draw-mode" variable can also be used for more esoteric purposes, e.g., to change the pen color frequently.

<u>pen-type</u> *is a variable whose default value is* normal

The "pen-type" describes the kind of pen to be used in the trail of a performer's movements, not in drawing the performer itself (see the previous description of "draw-mode" and "ease-mode"). The permissible types are "normal", "xor" and "eraser". The "pen-type" has no affect unless the pen is down (see the "(pen down)" message above).

<u>variables-to-copy-upon-creation</u> *is a variable whose default value is* (state visibility) *and is abbreviated as* vtcuc

This is really a special variable of Something. However, Something has it set to NIL so

it is inoperational by default. When an actor is made, the variables in the "variables-to-copy-upon-creation" list are "recalled" and then "set". These variables are not shared by an actor's offspring as is the usual case for variables.

Subsection J Colors

If you want to change the colors of a performer you use can the "set 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 using "Makecolor" as described in the LLogo memo [Goldstein 1975]. If you want to see a smooth transition from one set of colors to another you can use the following message.

(ASK ?a-performer DO IN ?number ?time-units SET YOUR COLORS TO ?colors) do in ? ? syct ?

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 ?a-performer PREPARE TO MIX COLORS WITH ?other-colors) ptmcw ?

The number of "other-colors" should be the same as the current value of 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 and so on. This message is used in the definition of the previous method as follows

```
(ask performer
     do when receiving (do in ?number ?units (or set change) your colors to ?colors)
     (script
      (I prepare to mix colors with ,colors) ;; prepare the mix
      (I set my color-mix to 0.0) ;; initialize color-mix
      (I do in ,number ,units increment your color-mix by 1.0)))
```

For example,

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

is the same as typing

```
(ask sally prepare to mix colors with (red white blue))
(ask sally set your color-mix to 0.0)
(ask sally do in 5 ticks increment your color-mix by 1.0)
```

Subsection K Interpolation

Sometimes you want a performer's shape to slowly change to another shape. In Director you can create an actor that is the *interpolation* of the appearances of two other performers.

`(ASK ?a-performer MAKE ?name INTERPOLATION TO ?another-performer)` `make ? itx ?`

This returns an actor named "name" that is the interpolation between "a-performer" and "another-performer". This actor is a performer 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 "a-performer" and 1.0 the appearance of "another-performer". Very interesting results occur if you try negative numbers or numbers greater than one. To make a simple movie of a circle becoming a star 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 star)
(ask star set your angle to 144)
(ask circle make circle-to-star interpolation to star) ;; make interpolation actor
(ask circle-to-star set your amount to 0.0) ;; start off looking like a circle
(ask circle-to-star set your (increment your amount by speed) to .1)
(ask circle-to-star gradually increment your amount by 1.0)
(ask circle-to-star 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 10 ticks) ;; send out 10 ticks recording as you go
(ask cts-movie project) ;; filming is over so project yourself
```

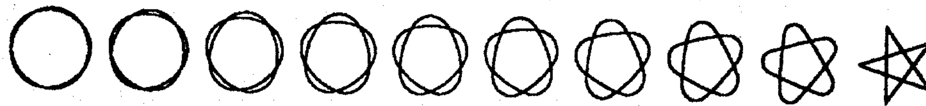

Figure 18 Ask Cts-Movie Project

This transition is linear. If you want the rate of change to increase just enter the following before running the movie.

```
(ask circle-to-star
     set your (increment your (increment your amount by speed) by speed) to .01)
; the amount speed itself has a speed now (ie the acceleration)
(ask circle-to-star gradually increment your (amount speed) by .25)
```

Subsection L Appearance Definition Using Instant Turtle

Another way of creating appearances for performers is by using a mode called "instant turtle". It is entered by typing (instant) and is exited by either typing "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 a performer to do something or to define the current image as either a performer or a procedure to be attached to a letter. Numerical arguments are given to it by typing them before the letter command. If no argument is given then the previous argument to that letter is assumed (or 1 if it is the first time that letter is used). It is useful for quickly and easily positioning performers as well as defining unusual appearances for performers. The mode is self-documenting.

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.

This is the minimum needed to define a text of any font, position, or orientation. You might want the screen to reflect changes in the variables "string" and "font". This can be accomplished as follows using the "continue-asking" type of transmission.

```
(define-method (set your (or string font) to ?new-value) text
 ;; this method works for both changes to the string or font
 (let ((visibility (I recall my visibility)))
      (cond (visibility (I hide)))
      (continue-asking) ;; this actually sets the variable
      (cond (visibility (I show))))
 new-value) ;;; we should still return the "new-value"
```

We can now use the new text definition.

```
(ask text make label)
(ask label show)
(ask label set your string to |Here I am|) ;; So it becomes the words "Here I am"
(ask label move 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 ken1. 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. If you type

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

the text will be displayed in that font.

Section V What The Stage Does

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

(ASK STAGE WIPE)

This wipes off anything from the Stage and then redraws any performers that should be visible. This is useful if a message or something messed up your display area. This can also be invoked by typing control-r (i.e., holding down the ctrl and r buttons).

(ASK STAGE CLEAR)

This tells all the actors to hide, so that the stage becomes empty.

The variables that are special to the Stage follow.

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

mode *is a variable whose default value is* normal

If the variable "mode" is set to "silent" then the Stage "pretends" to do what its told but does not 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 a performer 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 performer first but if there are many such performers then it is easier to set the stage's "mode". If, while Director is running, you want the Stage to run silently you can type "control a" at any time. "Control r" restores the mode and redisplays.

Section VI What Clocks Do

Clocks (instances of **Clock**) are the actors responsible for knowing who wants "tick" messages. Any actor that has planned anything has told its clock that it has something to do. (The "plan ..." messages handle this.) The only thing that you need ask of a clock is to run when you want all the planned activities to occur. Each actor is asked for its "clock" when planning and unless told otherwise inherits from **Something** one called "default-clock". Telling a clock 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-clock RUN FOR ?length ?units)` rf ! ?

Send tick messages to each actor with something planned for "length" "units". Units can be either "ticks", "frames", or "seconds". To run through a scene with, say, 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-each (sue sally) repeat (print your size) 5 times) ;; to see the values each time
(ask sally ask your clock to run for 4 ticks)
;; unless told otherwise Sue and Sally share the same clock
```

Most of the planning type messages accept units of time of either seconds, ticks, or frames. Seconds are most convenient for you to use, ticks are what Director uses, and frames are what movies (see below) use. The relationship between these different units of time is defined in the clock of the actor involved in the planning.

<u>frames-per-second</u> *is a variable whose default value is* 1 *and is abbreviated as* fps

If the display and computer 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. If you set the frames-per-second of the clock used in a movie called "Fantasia" to 4 then every fourth frame will be the same as if you had left the frames-per-second at the

default of 1. You can think of frames-per-second as the speed with which the "camera" shoots the action.

<u>`ticks-per-frame`</u> *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.

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

This variable is kept by each clock and should be a list of all actors sharing that clock with anything planned.

Section VII What Movies Do

Movies (instances of Movie) can also be told to run and they differ from clocks in that movies manage to remember any changes to the screen. Movies can then be asked to play back the changes at a speed that is typically much faster than when first created.

(ASK ?a-movie FILM THE NEXT ?film-length ?units) ftn ? ?

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

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

All performers that are currently on the screen or plan to appear during the next 12 seconds will be in the movie. Remember that "seconds" means film or animation time, not real or compute time.

(ASK ?a-movie FILM SECRETLY THE NEXT ?length ?units) fstn ? ?

This does the same as the previous message in that all changes to Stage are recorded except here they are not displayed (the stage's mode is "silent"). 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 SHOWING EVERY ?so-many FROM ?begin TO ?end) pse ? from ? to ?

; 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) **psaf ? 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 cannot 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 it can". If the speed is the same as the clock's frames-per-second, and if the machine is fast enough, then you will see the film at just the rate you planned things.

new-frame-action *is a variable whose default value is* erase-old *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 "erase-old" which erases by redrawing frames with the eraser down. If it is NIL then nothing will happen and you will see all the frames superimposed on the screen. If the value is "(erase-previous <number>)" then the "number"th previous frame is erased. For instance, if you set a movie's new-frame-action to (erase-previous 10) then the movie is projected so that you always see the 10 most recent frames. Any other value is EVALuated. One useful value is "(clearscreen)" which just clears everything off the screen. Sometimes this is faster.

<u>stage-mode</u> *is a variable whose default value is* `silent`

This variable specifies what mode the Stage should be in while filming. The default is "silent" so that all the intermediate changes that occur *within* a frame are not seen. This is desirable in conjunction with the default value of the "new-frame-handler" described below.

<u>new-frame-handler</u> *is a variable whose default value is* `frame-handler` *and is abbreviated as* `nfh`

The "new-frame-handler" of a movie is an actor that is sent a message whose pattern is

```
(just made frame number ?frame-number for ?movie)
```

whenever a frame is completed. What the default frame-handler does when receiving such messages is to show all the visible performers and then store the frame in the movie. When making a movie with "film secretly the next ..." the new-frame-handler is changed to one called "just-store-away-frame" which does not show you current state of the Stage. When Director is hooked up to a movie camera, then a "new-frame-handler" is defined to just show the visible performers and click the shutter and does not usually save away the frame in the movie.

`(ASK ?a-movie COMPILE ?file-name)`

This method creates 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 for another day. To run the movie, load it into a Director and ask the movie to project just as before. So to save the movie My-first-film in "ffilm >", compile it and then run it, do the following

```
(ask my-first-film compile (ffilm >))
;; put a Lisp translation of "my-first-film" in the file "ffilm >" on your directory
^z  ;; Leave Director
:complr 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
(ask my-first-film project)
```

Section 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 a performer 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 other objects. 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

(define physical-object performer
 ;; make physical-object as a kind of performer and send it the following messages
 (set your mass to 10) ;; the default mass
 (do when receiving (update your state)
    ;; when I get a message asking me to update my state
    (I set my position to ;; I update my position by
       ;; adding to my current position to the position of my velocity
       ,(position-sum (I recall my position)
                      (I ask my velocity recall your position)))
    ;; I ask the gravitational field at my location to change my velocity
    (ask gravitational-field
         apply gravitational forces at
         ,(I recall my position) to ,(I recall my velocity)))
 (do when receiving (yield pull at ?place)
     ;; to determine the gravitational pull at the place (G=1 in our units)
     (quotient (I recall my mass) ;; take my mass
               (square (I yield distance to ,place))
               ;; divide by the square of my distance to the place to get force per second
               (I ask my clock recall your frames-per-second)
               ;; divide by this to get force per frame
               (I ask my clock recall your ticks-per-frame)))
               ;; divide to get force per tick
 (do when receiving (recall your velocity) ;; if asked for my velocity
     (let ((velocity (continue-asking))) ;; find old one
          (cond ((null velocity) ;; if one does not exist then make one
                 (let ((velocity (ask velocity make)))
                      (ask ,velocity set your thing to ,myself)
                      (I set my velocity to ,velocity)))
                (t velocity)))))
```

```
(define gravitational-field something
 ;; I never move or appear on the screen so no need to be a performer
 ;; 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
     (I exert pulls of ;; I exert the pulls of the masses not at the place
        ,(remove-any-at-place (I recall my 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
    (compile-using performer
                    ;; this declares that the variable "velocity" is a performer
                    ;; without it, this transmission would compile less efficiently
     (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
     (I 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 ship physical-object ;; now to define ships
 (do when receiving (thrust forward ?amount) ;; When I'm asked to thrust forward
     (I ask my velocity set your heading to ,(I recall my heading))
     ;; I set the heading of my velocity to my own heading
     (I ask my velocity ;; and change my velocity by
        ;; having it go forward the quotient of the thrust and my mass
        move forward ,(quotient amount (I recall my 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 mass to 100) ;; the default mass of a sun is 100
 (when drawing do (repeat (sequentially (move forward 10) (turn right 10))
                          36 times))) ;; near enough to a circle (really a 36-agon)

; What follows is reasonable to give to every performer
(ask performer ;; 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 (I recall my heading)))
           (I set my heading to ,(heading-from begin-position end-position))
           (I move forward ,amount)
           (I set my 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 your velocity 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 your velocity to back 75)
   (set your position to (600 200)) ;; 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 do when receiving (display)
     (continue-asking) ;; display yourself as normal
     (ask the-turtle perform (shade 'lighttexture))) ;; so that it is shaded
(ask sun1 set your erasability to nil)
(ask sun1 show)

(ask sun2 do when receiving (display)
     (continue-asking) ;; display yourself as normal
     (ask the-turtle perform (shade 'texture))) ;; a darker texture for Sun2
(ask sun2 set your erasability to nil)
(ask sun2 show)

; tell the field about the objects

(ask gravitational-field set your masses to (sun1 sun2 enterprise))

; 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 11 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.

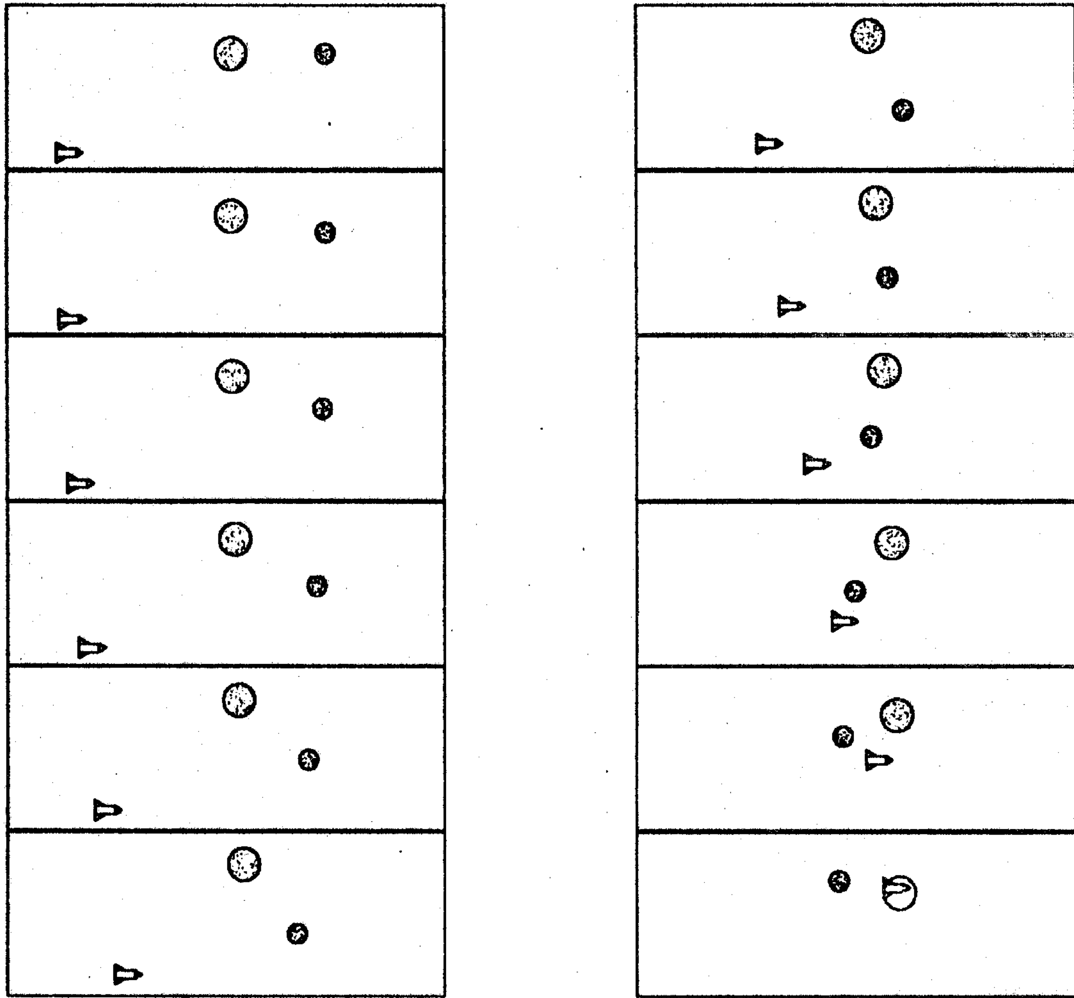

Figure 19 A Test of the Space War Program

Section IX Compiling

Director code can be translated 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. As an example of the pattern optimization, consider what happens to the pattern

```
(plan after ?number ?units %message)
```

It becomes something like the following

```
; this is the test part of the pattern
(AND (EQ (CAR MESSAGE) 'PLAN) ;; the first element of the message must equal 'plan
     (EQ (CADR MESSAGE) 'AFTER) ;; and the next element equal 'after
     (CDDDR MESSAGE)) ;; and the message must be at least four elements long
```

The binding of values happens only after the predicate indicates that there definitely is a match. In this case the body of the method becomes

```
((LAMBDA (MESSAGE UNITS NUMBER) ;; the variables in pattern are bound
  <body of the method is here>)
 (CDDDDR *MESSAGE) (CADDDR *MESSAGE) (CADDR *MESSAGE));; to parts of the message
```

This compilation of patterns happens throughout the system. The method patterns and the data base inquiries are compiled.

The list of method patterns owned by an actor are 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 optimized Lisp version of itself. (Note that this does not make the actor become more efficient. To do that you should eval the result, e.g. (eval (ask sue save)).)

The most important optimization is the compilation of message transmissions. Several different actors, some with many different methods, will typically try to match a message as it gets passed along to each actor's parent. To skip all this computation the macro for transmissions (ASK) figures out what methods will be involved by the transmission. For example, the transmission

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

is replaced by

```
(LET ((MYSELF (ACTOR-OF 'MY-FIRST-FILM))
      (*MESSAGE '(PROJECT FRAMES 3 TO 6)))
     (QUICK-COMPILED-ASK '#,(METHOD-AT-POSITION (|PROJECT FRAMES ? TO ?| 0) MOVIE)
                        ;; the "#," indicates that the method is found at load time
                        '#,(ACTOR-OF 'MY-FIRST-FILM))) ;; the compilation target
```

where the Lisp function QUICK-COMPILED-ASK will run the first method of **Movie** whose pattern is (PROJECT FRAMES ? TO ?).

This scheme works fine even when the message and target contain commas indicating variables. If the target is a variable, then the compiler tries to find a *compilation target*, an actor that either will receive the message or one of its descendants will. (If it can't deduce one, then Something is used.) At run time the message is sent to the actor and its ancestors stopping at the compilation target (the second argument to QUICK-COMPILED-ASK). If no one along the way from the actor to the compilation target claimed the message then the method indicated as the first argument to QUICK-COMPILED-ASK is run. If the message has "commas" in it indicating variables, then the compiler generates a list of methods that could possibly be applicable. Typically this list is short and the pattern predicates of each element is run against the message at run time until one succeeds. For this case the Lisp function COMPILED-ASK is used.

(ASK ?anyone MACRO EXPAND WHEN RECEIVING ?pattern %form)

This creates a "macro" method, which is like a normal method except that it is doubly evaluated. When compiling the first evaluation takes place, leaving the second evaluation for when the code is really running. For example, Something's "increment your ..." method is really a macro method defined as follows,

```
(ask something macro expand when receiving (increment your ?variable by ?amount)
 '(ask ,myself set your ,variable to
      `(plus (or (ask ,myself recall your ,variable)
                 0) ;;if variable is new for example
             ,amount)))
```

The backquote (') is a way of inserting commas, it becomes a comma upon evaluation. A transmission like the following one

```
(ask sally increment your height by 2)
```

expands to

```
(ask sally set your height to ,(plus (or (ask sally recall your height) 0) 2))
```

One advantage of making such a method a macro is that the expansion can then be compiled with a "compilation-target" that is typically lower in the parent/offspring hierarchy. In the previous case, we can compile the "set your" and "recall your" messages knowing they will go to Sally, while otherwise they would have to check at run time from Sally to Something to see if anyone has special methods for this. A disadvantage is that any special methods that Sally might have for "increment your" messages will not be noticed by compiled code, so if you use macro methods be prepared to recompile your code when you change them.

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 "Macroexpanded".[1] `(COMPILER-SWITCH NIL)` will not only prevent new forms from being made but will clean up old expansions.

---

1. This uses the macro expansion-use set to 'macroexpanded option of "defmacro" in Maclisp and a similar scheme on the Lisp Machine.

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

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

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

will return

```
(LET ((MYSELF (ACTOR-OF 'SALLY))
      (*MESSAGE '(SET YOUR COLOR TO BLUE)))
     (QUICK-COMPILED-ASK '#,(METHOD-AT-POSITION (|SET YOUR ? TO ?| 0) SOMETHING)
                         '#,(ACTOR-OF 'SALLY)))
```

To compile a file of Director code, you should run the function COMPILE-FILE from Director. It produces a file of Lisp (the second file name is "LISP") which you then hand off to the Lisp compiler. For example, to compile the file "ken;4hex >" do the following.

```
(compile-file '|ken;4hex|)
|done| ;; Its finished making the Lisp file
^z ;; leave Director
:complr ken;4hex lisp ;; start up a Lisp compiler and give it the Lisp file just created
direct^h ;; when its finished go back to Director
(load '|ken;4hex|) ;; load in the compiled version of the file
```

Besides compilation there is another mechanism for efficiency in Director. It allows a user to declare a more efficient representation for a subset of variables. The current feature is limited, but does point to a general scheme for getting some of the efficiency back this is lost in using very general mechanisms for implementing variables.

```
(ASK ?anyone WHEN REPRESENTING ?pattern-for-variables USE AN ARRAY OF SIZE ?size)
```

This adds two new methods to the recipient: one for recalling a variable that matches the "pattern-for-variables" and another for setting any such variables. Movies use this as follows,

```
(ask ,new-movie
     when representing (frame ?number) use an array of size ,(1+ max-frame-number))
```

Which sets up an array to hold the values of the variables whose names are (frame 0), (frame 1), and so on up to (frame <max-frame-number>).

Section X Odds and Ends

Subsection A Debugging

For the most part debugging Director is like debugging Lisp. The "trace" message described under the Something section is quite helpful. There are four kinds of break points. Lisp ones, "shouldnt-happen" ones which indicate a system bug, "no-such-actor", and "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 is not an actor.
The message is RECALL YOUR PARENT
For help type (?)
(?) ;; so you type "(?)"
SUE is not defined as an actor --- if it is misspelled then type
 (return '<correct-spelling>)
 if you want to define SUE do so now and then type
 (return 'retry)
 otherwise type "$p " and the transmission will not occur and NIL will be returned
(ask something make sue) ;; so then you make sue
(return 'retry) ;; have have it try again
SOMETHING ;; and the original question is answered
```

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. In general, when you get an error try typing "(?)", it might be helpful.

Subsection B 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 any number of elements in the corresponding list structure. It should be used only once per list (which can be a sublist of course).[1] An expression surrounded by curly brackets {} is treated specially. If it begins with the word "OR" then if any of the following sub-patterns match the corresponding element in the message, then the match continues. If the word is "AND", then all the following patterns must match and all the bindings are made. 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}}". An atom in a pattern can have a comma in it, in which case the value of the atom is looked up at run time.

1. If the "%" appears anywhere other than the end of a pattern, it should work but will be much slower than otherwise. The pattern matcher does no back-tracking so if you use more than one "%" at the same level it will match only some of the messages that it "could".

Subsection C Global Variables

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

```
MYSELF ;; 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 most current message

*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 messages are typed when a file is loaded

*MESSAGE-NOT-UNDERSTOOD
;; its value is a function of the target and message and it should
;; handle 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

*INSERT-METHODS-AT-END ;; Normally NIL
;; If T then new methods are added at the end of the actor.
;; T is the default when loading in a file using DIRECTOR-LOAD so the file
;; can have the methods in the same order as they will be in the actor

*REPLACE-OLD-METHODS ;; Normally NIL
;; if T then when defining methods will replace equivalent method
;; instead of adding the new method to the beginning or end of the list of methods of the actor

*REPLACE-OLD-ACTORS ;; Normally T
;; if T and you MAKE an actor that already exists, you replace it with this new one
;; otherwise it UNMAKEs the old actor and then makes a new one

*PROTECT-ALL-ACTORS ;; Normally T
;; asks for confirmation before any actor is UNMAKEd

*PROTECTED-ACTORS ;; Initially all the system's primitive actors
;; makes it hard to UNMAKE or replace any actor whose name is on this list

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

Subsection D 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 PERFORMER
        (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.

Another pleasant macro is called "I". It simply expands to "ask myself" which happens very frequently within the body of methods. For readability you can use "my" instead of "your" in any message since a method macro will convert it for you. (Therefore using "my" in interpreted code will be slower but there is no overhead once its compiled.)

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. It is typically printed specially, however (print (actor-of 'clock)), for example, should show it to you as it really is. The function "Script" is identical to the Lisp function "Progn", however it serves the important function of "protecting" any commas within from causing the next element to be evaluated immediately. Those elements preceded by commas will instead be evaluated when the "script" itself is being run.

There are many slight variants of the "ask" macro. They differ primarily in either how errors are handled or how the target is specified. They are

```
TELL ;;; same as ask, just not as polite
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
A ;; Same as above, useful at top level when incremental compiler is on
ASK-ALL ;; ask all actors whose name matches the target (works only with non-atomic names)
        ;; e.g. (ask-all (comparison-of sam ?) ...) to ask all the comparisons of sam
ASK-EACH ;; the target is a list of actors to send the message to
```

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)
```

Subsection E 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 data base 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 data base, the methods and the extended methods --- all potentially usable by the same actor. These features lead naturally to very modular programs with all the advantages that that brings.

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

[Papert 1971a], [Papert 1971b], [Goldstein 1975], [Goldstein 1976], and [Kahn 1977b].

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 without requiring a prefix character.[2] Another mechanism in Director that both drastically reduces the amount of typing necessary and makes clear at any point what all of the user's alternatives are is a menu-oriented interface describe in the section called "Getting Started".

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

---

1. Emacs is an excellent text editor developed at the MIT Artificial Intelligence Lab. [Stallman 1979] The abbreviation package in Emacs was developed by Eugene C. Ciccarelli.
2. This is why all the abbreviations that would be an English word have an "x" at the end.

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.

Subsection F How Director Works

You shouldn't need to know how Director works to use it effectively. If you are curious and know Lisp well then read on, otherwise skip to the next section.

Actors are represented as lists. The first element of the list is the atom "director-symbol-for-actor" which indicates that what follows is an actor. The rest of the list is a list of methods, except for the last element. The last element is the name of the actor's parent if it is named otherwise it is a pointer to the parent itself.

Each method is represented by a list. The first element indicates what kind of method it is. The second element is a function of no arguments that corresponds to the body (the "actions" given in "do when receiving ..." messages for example) embedded in some code that binds the variables of the method's pattern. Compiled actors just have the "subr" pointers of the functions here. The third element of the list is a predicate of one argument that returns NIL only if the pattern of the method does not match the argument. If the pattern does not contain any variables then the pattern itself is here and EQUAL is used. The fourth element is optional and is an alist for additional properties of the method.

There are four kinds of methods. The simplest is a "value-method" which just holds a constant and returns that when invoked. The most common method type is a "normal-method". When its predicate applied to the message returns a non-nil value its body is invoked. A "Macro-method" is similar except that the result of invoking its body is EVALed. The fourth type of method is a "method-selector". They are generated by

the compiler. When a method selector's "body" is invoked it returns a method that matches the message if one does. It also contains a pointer to the last method that it is a selector for. When Director is searching for a matching method this pointer is followed if the selector returns NIL thereby skipping all the methods in between.

The Lisp macro "Ask" just goes through the methods of the target of the message, invoking their method selectors and pattern predicates. If no method is found then it tries again with the parent of the actor. When it finds a method its body is invoked and the result returned.

Variables are implemented as a "value method" that contains two alists. The first is for variables with atomic names and the second for the others. The value method is always the first method of an actor. Data bases are implemented as variables with internal names.

Atomic names of actors are implemented both as a normal variable of the actor and by putting the actor on the property list of the name under the indicator "actor". Non-atomic names use a generalized property list scheme.

Message continuations are used for activities that take longer than a tick. The messages that need to be sent after the current activity completes are passed along until the activity is finished.

Subsection G 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).

Subsection H 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, type

```
(ask poly make star)
(ask star set your angle to 144)
(ask star show)
```

and you should see a star appear on your TV. Type "(?)" for a little bit of help.

Another way of interacting with Director is via a menu-oriented interface. Instead of typing in programs this subsystem incrementally puts together Director programs in response to answers to questions. Besides drastically reducing the amount of typing required this mode also makes clear at every point what your alternatives are. This mode can be entered by typing

```
(talk)
```

or if you want the system to keep a copy of the program being written on file then type

```
(talk '<file-name>)
```

If want to use only part of Director, are running it in color (in which case you should type (run-color) before doing anything) or want Director to control a 2500 graphical display then type

```
:LISP KEN;DIRECT
```

and as you need things the appropriate files will be loaded. Have fun and report any

1. Don't do this if you are not on a TV or are planning to run Director in color or with a 2500 display.

problems to KEN@AI.

There is a version on the Lisp Machine [Weinreb 1979] which is currently so volatile that you should ask me if you want to run it.

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. I am very indebted to the Learning Research Group at Xerox Parc for their pioneering the development of object-oriented programming and graphics.

Section XI Bibliography

Bibliography

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

[Borning 1979]
Borning, A.
ThingLab -- A Constraint-Oriented Simulation Laboratory,
Xerox Palo Alto Research Center report SSL-79-3, July 1979

[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-1, 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

[Kahn 1979]
Kahn, K.
Creation of Computer Animation from Story Descriptions,
MIT AI-TR 540, August 1979

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

[Kay 1977b]
Kay, A.
"Microelectronics and the Personal Computer"
Scientific American, September 1977

[Mandelbrot 1977]
Mandelbrot, B.
Fractals: Form, Chance, and Dimension, W.H. Freeman and Co., San Francisco, 1977

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

[Negroponte 1979]
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

[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

[Stallman 1979]
Stallman, R.
"EMACS --- The Extensible, Customizable, Self-Documenting Display Editor"
MIT-AI Memo 519

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

[Weinreb 1979]
Weinreb, D. and Moon, D.
"Lisp Machine Manual --- second preliminary version", January 1979, MIT AI Lab

Section XII Index of Patterns

Message Pattern .... Abbreviation Page

Patterns that Something *Handles*

Message Pattern	Abbreviation	Page
(MAKE ?name)		10
(MAKE)		11
(MAKE ?name IF ITS NOT ALREADY)	make ? iina	11
(MAKE UP A NAME)	muan	11
(MAKE UP AN UNINTERNED NAME)	muaun	12
(CLONE)		12
(CLONE AND NAME IT ?name)	cani ?	12
(MAKE SYNONYM ?name)	ms ?	12
(DO WHEN RECEIVING ?pattern %actions)	dwr ? %	13
(DO ONCE WHEN RECEIVING ?pattern %actions)	dowr ? %	13
(PRINT YOUR ?variable)	py ?	14
(PRINT {or memory variables script database all})	ps OR pm OR pvs OR pdb	14
(PRINT {or memory variables script database all} ON FILE ?file-name)		14
(SAVE %file-names)		14
(HELP %pattern)		15
(RECALL METHOD FOR %sample-message)	rmf %	15
(RECALL INTERNAL METHOD FOR %sample-message)	rimf %	15
(REMOVE METHOD FOR %sample-message)		15
(TRACE ?pattern %action)		16
(UNTRACE ?pattern)		16
(SET YOUR ?variable TO ?new-value)	sy ? to ?	17
(RECALL YOUR ?variable)	ry ?	17
(RECALL EACH OF YOUR ?variable-pattern)	reoy ?	18
(INCREMENT YOUR ?variable BY ?amount)	iy ? by ?	18
(MULTIPLY YOUR ?variable BY ?factor)	myx ? by ?	19

(ADD ?new-item TO YOUR LIST OF ?list-name) .... add ? tylo ? 19

(ADD ?new-item TO YOUR LIST OF ?list-name REGARDLESS) .... add ? tylo ? reg 19

(REMOVE ?old-item FROM YOUR LIST OF ?list-name) .... remove ? fylo ? 20

(CONSIDER ?list-of-names SYNONYMS) 20

(CONSIDER ?names COMPONENTS OF ?variable) 20

(CONSTRAIN YOUR ?variable TO EQUAL ?function OF ?variable-2 OF ?other) 21

(LIST ALL YOUR VARIABLE NAMES) .... layvn 22

(FORGET YOUR ?variable) .... fy ? 22

(FORGET EVERYTHING) .... fe 22

(MEMORIZE ?item) .... mem ? 23

(RECALL AN ITEM MATCHING ?pattern THEN %actions) .... raim ? then % 23

(RECALL EACH ITEM MATCHING ?pattern THEN %actions) .... reim ? then % 23

(COLLECT ITEMS MEMORIZED MATCHING ?pattern) .... cimm ? 24

(FORGET ITEMS MATCHING ?pattern) .... fim ? 24

(PLAN NEXT %action) .... pn % 25

(PLAN AFTER ?number ?units %action) .... pa ? ? % 25

(PLAN AFTER RECEIVING ?event-pattern TO ?message-form) .... parx ? to ? 26

(TICK) 26

(GRADUALLY %action ?amount) .... grad % ? 26

(GRADUALLY SET YOUR ?variable to ?value) .... gsy ? to ? 28

(DO AT SPEED ?speed %action ?amount) .... das ? % ? 29

(DO IN ?number ?time-units %action ?amount) .... do in ? ? % ? 29

(ASK ?another %message) 30

(SEQUENTIALLY %actions) .... se % 30

(CONCURRENTLY %actions) .... co % 31

(REPEAT ?message FOREVER) 34

(REPEAT ?message ?number TIMES) 34

(STOP EVERYTHING) 34

(WAIT FOR %signal) .... wf % 35

(ASK YOUR ?variable %message) .... ay ? % 35
(ASK EACH OF YOUR ?variable %message) .... aeoy ? % 36
(BROADCAST TO YOUR ?others-name %message) .... bty ? % 36
(DO OR BROADCAST TO YOUR ?others-name %message) .... dobty ? % 37
(KEEP DOING UNTIL ?predicate %message) .... kdu ? % 37

Patterns that Performer *Handles*

(WHEN DRAWING DO %messages) .... wdd % 45
(WHEN DRAWING USE ?draw-procedure OF %draw-args) .... wdu ? of % 46
(SHOW) 48
(HIDE) 48
(SHOW ALL) 48
(HIDE ALL) 48
(MOVE FORWARD ?amount) .... mf ? 49
(MOVE BACK ?amount) ....mb ? 49
(MOVE RIGHT ?amount) .... ml ? 50
(MOVE LEFT ?amount) ....mr ? 50
(MOVE UP ?amount) .... mu ? 50
(MOVE DOWN ?amount) ....md ? 50
(TURN RIGHT ?degrees) .... tr ? 50
(TURN LEFT ?degrees) .... tl ? 50
(SET YOUR STATE TO (?xcor ?ycor ?heading)) .... syst ? 51
(SET YOUR XCOR to ?value) ....syxt ? 51
(SET YOUR YCOR to ?value) ....syyt ? 51
(SET YOUR POSITION TO (?xcor ?ycor)) ....sypt ? 51
(SET YOUR HEADING TO ?direction) .... syht ? 51
(GROW ?amount) 51
(SHRINK ?amount) 51
({or SHRINK GROW} BY FACTOR ?number) .... sbf OR gbf 51

(SET YOUR WHOLE TO ?whole) .... sywt ? 52

(ABSORB YOUR PARTS) .... ayp 54

(FUSE YOUR PARTS) .... fyp 55

(RUN ?action) 60

(PEN DOWN) .... pd 61

(PEN UP) .... pu 61

(DO IN ?number ?time-units SET YOUR COLORS TO ?colors) ...... do in ? ? syct ? 63

(PREPARE TO MIX COLORS WITH ?other-colors) .... ptmcw ? 63

(MAKE ?name INTERPOLATION TO ?another-performer) .... make ? itx ? 64

(LET ME DRAW YOU) .... lmdy 66

Patterns that **Stage** *Handles*

(WIPE) 68

(CLEAR) 68

Patterns that **Clock** *Handles*

(RUN FOR ?length ?units) .... rf ? ? 70

Patterns that **Movie** *Handles*

(FILM THE NEXT ?film-length ?units) .... ftn ? ? 72

(FILM SECRETLY THE NEXT ?length ?units) .... fstn ? ? 72

(PROJECT) 72

(PROJECT SHOWING EVERY ?so-many FROM ?begin TO ?end) ....pse ? from ? to ? 72

(PROJECT FRAMES ?begin TO ?end) ....pj ? to ? 73

(PROJECT STARTING AT FRAME ?begin SHOWING EVERY ?so-many) ....psaf ? se ? 73

(PROJECT SHOWING EVERY ?so-many) .... pse ? 73

(PROJECT FRAME ?number) ....pf ? 73

(COMPILE ?file-name) 74

Section XIII Index of Special Variables

Variable Pattern	Abbreviation	Page

Variables Treated Specially by Something

Variable Pattern	Abbreviation	Page
name		37
parent		37
offspring	offs	38
private-variable-names	pvn	38
siblings	sib	38
descendants	des	38
childless-descendants	cd	38
clock		38
things-to-do-next	ttdn	38
default-speed	ds	38

Variables Treated Specially by Performer

Variable Pattern	Abbreviation	Page
erasability	eras	61
default-whole		61
standardize-size?		62
standardize-center?		62
draw-mode		62
erase-mode		62
pen-type		62
variables-to-copy-upon-creation	vtcuc	62

Variables Treated Specially by Stage

Variable Pattern	Abbreviation	Page
height		68
width		68
mode		68

Variables Treated Specially by Clock

frames-per-second .... fps 70

ticks-per-frame .... tpf 71

actors-to-run-next .... atrn 71

Variables Treated Specially by Movie

speed 73

new-frame-action .... nfa 73

stage-mode 74

new-frame-handler .... nfh 74