Best
Available
Copy

⑤ FG

ADA016811

PYGMALION:
A Creative Programming Environment

by

David Canfield Smith

D D C

RECEIVED

NOV 5 1975

B

COMPUTER SCIENCE DEPARTMENT
Stanford University

PYGMALION:
A Creative Programming Environment

by
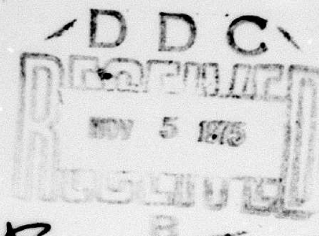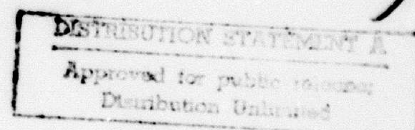
David Canfield Smith

ABSTRACT

PYGMALION is a two-dimensional, visual programming system implemented on an
interactive computer with graphics display. Communication between human being and
computer is by means of visual entities called "icons", subsuming the notions of "variable",
"reference", "data structure", "function" and "picture". The heart of the system is an interactive
"remembering" editor for icons, which executes and (optionally) saves operations for later re-
execution. The display screen is viewed as a document to be edited. Programming consists of
creating a sequence of display frames, the last of which contains the desired information.
Display frames are modified by editing operations. PYGMALION employs a powerful
paradigm that can be incorporated in virtually any other programming language:

Every operation has both visual (aesthetic) semantics and internal
(mechanical) semantics.

In fact, every operation in PYGMALION has three responsibilities:

(a) for accomplishing a given internal machine task — the machine
"semantics" of the operation;

-A-

(b) in display mode, for generating a representative visual action;

(c) in remember mode, for adding onto a code list the operation(s) necessary to reproduce itself.

Thus the system includes an incremental "iconic compiler". Since each operation has visual semantics, the display becomes a visual metaphor for computing. The programmer need deal with operations only on the display level; the corresponding machine semantics are managed automatically. The mechanical aspects of programming languages has been and is continuing to be well studied. The focus in this paper is on developing and interacting with an articulate visual presentation.

PYGMALION is a computational extension of the brain's short term memory. It is designed to relieve the load on the short term memory by providing alternative storage for mental images during thought. The display screen is seen as a "dynamic blackboard", on which ideas can be projected and animated. Instead of abstract symbols, the programer uses explicit display images. Considerable flexibility is provided for designing icons; the programmer may give them any shape that can be generated by a routine. This helps to reduce the translation distance between representations used in the mind in thinking about a problem and representations used in programming the problem.

The main innovations of PYGMALION are:

(1) a dynamic representation for programs — an emphasis on doing rather than telling;

(2) an iconic representation for parameters and data structures requiring less translation from mental representations;

(3) a "remembering" editor for icons;

(4) descriptions in terms of the concrete, which PYGMALION turns into the abstract.

The responsive, visual characteristics of PYGMALION permit it to play an active role in human problem solving. The principal application has been in assisting the design and simulation of algorithms.

This dissertation was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Preface

The following is a map of this document.

Chapters 1,2 -- A psychological model of creative thought, forming the basis for the PYGMALION design principles.

Chapter 3 -- Other projects which adhere to some of the same principles.

Chapters 4,5 -- The PYGMALION programming environment in detail.

Chapter 6 -- Examples of PYGMALION programs and data structures.

Chapter 7 -- Conclusions and suggestions for the future.

This paper places equal emphasis on presenting a psychological model of thought and using the model in a computer environment. Readers interested in aspects of creative thought which can be assisted by a computer should read chapters 1 and 2. Readers interested in how the PYGMALION system attempts to stimulate creative thought should look at chapter 6 (mostly pictures) to get the flavor, then read chapters 4 and 5. The works of others which deal with the same aspects are described in chapter 3. Chapter 7 suggests areas for future exploration. Thorough readers will read the chapters in order. Chapter 6 and 4-A through 4-D are a minimal set for readers in a hurry.

There are three parts to this report.

Part I examines a psychological model which contends that visual imagery is a powerful metaphor for thought. Mental images are derived from sense perceptions, but they may be abstracted and fragmented. Visual images are superior to linguistic descriptions in the *quantity* of information they can contain and in the *quality* of their portrayal of objects and concepts. Aspects of creativity are also discussed. Creative thinking involves the conjunction of two normally distinct thought contexts using images from both.

This part develops the criteria for articulate communication between a human being and a computer. A programming language can be more than just a passive medium for communication. The technology now exists to design systems which actively augment the user's ability to think and learn.

The goal is to develop a system whose representational and processing facilities correspond to and assist mental processes that occur during creative thought.

Part II derives a programming environment called PYGMALION from the model of thought in Part I. PYGMALION is a two-dimensional, visual programming system implemented on an interactive computer with graphics display. Communication between human being and computer is by means of visual entities called "icons", subsuming the notions of "variable", "reference", "data structure", "function" and "picture". The heart of the system is an *interactive "remembering" editor for icons*, which executes and (optionally) saves operations for later re-execution. The display screen is viewed as a document to be edited. Programming consists of creating a sequence of display frames, the last of which contains the desired information. Display frames are modified by editing operations. PYGMALION employs a powerful paradigm that can be incorporated in virtually any other programming language:

> Every operation has both visual (aesthetic) semantics and internal (mechanical) semantics.

In fact, every operation in PYGMALION has three responsibilities:

(a) for accomplishing a given internal machine task -- the machine "semantics" of the operation;

(b) in display mode, for generating a representative visual action;

(c) in remember mode, for adding onto a code list the operation(s) necessary to reproduce itself.

Thus the system includes an incremental "iconic compiler". Since each operation has visual semantics, the display becomes a visual metaphor for computing. The programmer need deal with operations only on the display level; the corresponding machine semantics are managed automatically. The mechanical aspects of programming languages has been and is continuing to be well studied. The focus in this paper is on developing and interacting with an articulate visual presentation.

PYGMALION is a computational extension of the brain's short term memory. It is designed to relieve the load on the short term memory by providing alternative storage for mental images during thought. The display screen is seen as a "dynamic blackboard", on which ideas can be projected and animated. Instead of abstract symbols, the programer uses explicit display images. Considerable flexibility is provided for designing icons; the programmer may give them any shape that can be generated by a routine. This helps to reduce the translation distance between representations used in the mind in thinking about a problem and representations used in programming the problem.

The main innovations of PYGMALION are:

(1) a dynamic representation for programs -- an emphasis on *doing* rather than *telling*;

(2) an iconic representation for parameters and data structures requiring less translation from mental representations;

(3) a "remembering" editor for icons;

(4) descriptions in terms of the *concrete*, which PYGMALION turns into the *abstract*.

The responsive, visual characteristics of PYGMALION permit it to play an active role in human problem solving. The principal application has been in assisting the design and simulation of algorithms. Part II describes the implementation in detail and presents numerous examples.

Part III summarizes the results, with suggestions for the future. The responsive, visual characteristics of PYGMALION permit it to play an active role in human problem solving. The principal application has been in assisting the design and simulation of algorithms.

The appendices include a listing of the initial implementation, notes and bibliography. In this paper, material that would normally be placed in footnotes has been assimilated directly into the text (possibly in parentheses) or dropped entirely. The notes designated by superscripts [eg 99] contain only specific article and page information and are found at the end of the paper. They do not contain any supplementary text. Superscripts having a number higher than one hundred [eg 1975] are dates.

I wish to express my sincere thanks to my advisers, Drs. Alan C. Kay of the Xerox Palo Alto Research Center and Kenneth M. Colby of Stanford University. Dr. Kay has inspired the most stimulating project of my career by demonstrating the power of an individual with a personal computer. In seven years of association with Dr. Colby, he has never been less than enlightening, motivating, and, in general, amazing. His encouragement and guidance have been invaluable. I also wish to thank Drs. Terry Winograd and Cordell Green for helpful suggestions in the preparation of this manuscript; Horace Enea for interesting and informative conversations too numerous to mention; Dr. John McCarthy, Lester Earnest and everyone at the Stanford Artificial Intelligence Laboratory for making the lab a good place to work; my wife Janet for all the midnight hours spent helping me with this thesis and with life; and finally the Greeks for the wonderful myth of Pygmalion, giving shape to the dreams of creative people everywhere.

Table of Contents

Introduction

In Greek mythology, Pygmalion was a king of Cyprus who fell in love with a statue of Aphrodite. Ovid, in the *Metamorphoses*, invents a more sophisticated version. Pygmalion, a sculptor, makes an ivory statue representing his ideal of womanhood, then falls in love with his own creation; Venus brings it to life in answer to his prayer.[1]

Artists down through the centuries have felt Pygmalion's lure: to play the role of creator. But outside the mythological world of gods and goddesses the struggle has been frustrating. Artists have consistently reported a feeling of excitement *during* the process of creation, followed by depression when the work is *finished*. Their best efforts always remained lifeless and, in the end, unsatisfying. Michelangelo is said to have struck with his mallet the knee of perhaps the most beautiful statue ever created, the Pieta, when it would not speak to him. The historian E.H.Gombrich discusses "the belief in the power of art to create rather than to portray" in his eloquent book *Art and Illusion*:

> Without the underlying promise of this myth, the secret hopes and fears that accompany the act of creation, there might be no art as we know it. One of the most original young painters of England, Lucien Freud, wrote very recently: "The moment of complete happiness never occurs in the creation of a work of art. The promise of it is felt in the act of creation, but disappears toward the completion of the work. For it is then that the painter realises that it is only a picture he is painting. Until then he had almost dared to hope that the picture might spring to life."

> "Only a picture," says Lucien Freud. It is a motif we find in the whole history of Western art; Vasari tells of Donatello at work on his *Zuccone*, looking at it suddenly and threatening the stone with a dreadful curse, "Speak, speak -- *favella, favella, che ti venga il cacasangue!*" And the greatest wizard of them all, Leonardo da Vinci, extolled the power of the artist to create. In that hymn of praise to painting, the "Paragone," he calls the painter "the Lord of all manner of people and of all things." "If the painter wishes to see beauties to fall in love with, it is in his power to bring them forth, and if he wants to see monstrous things that frighten or are foolish or laughable or indeed to be pitied, he is their Lord and God."

Leonardo Da Vinci: Left: Grotesque heads, c.1495; Right: Leda, c.1509 [Gombrich[2]]

Indeed the power of art to rouse the passions is to him a token of its magic ... And yet Leonardo, if anyone, knew that the artist's desire to create, to bring forth a second reality, finds its inexorable limits in the restrictions of his medium. I feel we catch an echo of the disillusionment with having created only a picture that we found in Lucien Freud when we read in Leonardo's notes: "Painters often fall into despair ... when they see that their paintings lack the roundness and the liveliness which we find in objects seen in the mirror ... but it is impossible for a painting to look as rounded as a mirror image ... except if you look at both with one eye only."

Perhaps the passage betrays the ultimate reason for Leonardo's deep dissatisfaction with his art, his reluctance to reach the fatal moment of completion: all the artist's knowledge and imagination are of no avail, it is only a picture that he has been painting, and it will look flat. Small wonder that contemporaries describe him in his later years

as most impatient of the brush and engrossed in mathematics. Mathematics was to help him to be the true maker. Today we read of Leonardo's project to build a "flying machine," but if we look into Leonardo's notes we will not find such an expression. What he wants to make is a bird that will fly, and once more there is an exultant tone in the master's famous prophecy that the bird *would* fly. It did not ... The claim to be a creator, a maker of things, passed from the painter to the engineer -- leaving to the artist only the small consolation of being a maker of dreams.[3]
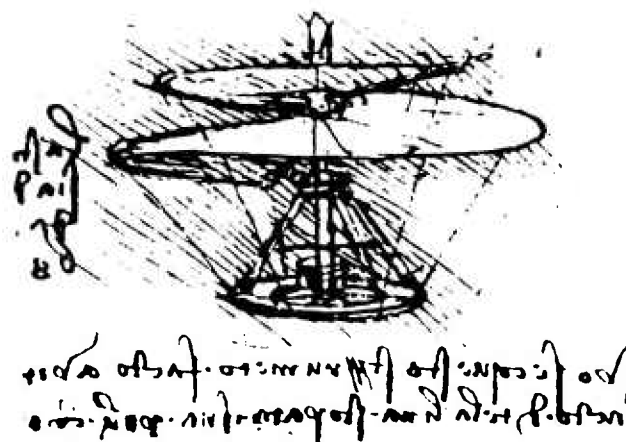
Leonardo Da Vinci: Airscrew, c.1488 [Wallace[4]]

Today, with the advent of inexpensive computers, for the first time the average man is presented with a powerful medium through which his ideas *can* attain a life of their own. A computer program is a *creation*: unusual, appropriate, in a very real sense a work of art, perhaps more so than computer "scientists" like to admit. Many programmers have felt a sense of creative excitement watching their programs execute on a computer without needing their intervention, particularly if they can watch the programs' progress with their own eyes. Surprise, stimulation, satisfaction and savoring are the rewards of creativity.

There is much to be learned from associating artistic concepts with computers. The potential use of computers in creative activity is the source of much of their attraction and mystique. "The computer is the world's best toy," Papert has said in discussing his project to use the computer to enhance education.[5] Arnheim advances a more serious reason: "There [is] much evidence that truly productive thinking, in whatever area of cognition, takes place in the realm of imagery. There has been a longstanding isolation and neglect of the arts in society and education."[6] Pye is succinct in his agreement: "If anyone thinks it important to a civilization that a common ground between art and science shall be found,

then he had better look for it in front of his nose; for it is ten to one that he will see there something which has been designed."[7] And Koestler contends that there is *no difference* between scientific and artistic creativity, or even between scientific and comic creativity: "Regardless of what scale of values you choose to apply, you will move across a continuum without sharp breaks; there are no frontiers where the realm of science ends and that of art begins."[8] This report investigates ways to exploit the creative potential of those using computers.

Durer: Drawing, c.1527 [Gombrich[9]]

PART I

Aspects of Creative Thinking

Chapter 1

Some Characteristics of Thought

Summary

(1) Visual imagery is a productive metaphor for thought. Visual images are concrete and easily manipulated, and they provide a powerful representational capability. The images are multi-dimensional and are analogous to the pertinent features of the subjects they represent.

(2) Words are "Fregean" and lack the representational power of images.

This part of the report is divided into three chapters. The first chapter presents a model of some types of thought processes in the brain. The second chapter focuses more particularly on the nature of creative thinking, thought processes that are both original and productive. The third chapter surveys several approaches to using machines which augment these processes. The model of thinking has been largely adopted from the theories of the psychologist Rudolf Arnheim, particularly from his superb book *Visual Thinking*, augmented with other results from current psychological research. Only those aspects of the model which provide guidance for the design of a computer interface have been emphasized. The goal of PYGMALION is to develop a computer system whose representational and processing facilities correspond to and assist mental processes that occur during creative thought. Part I serves as the raw material for the system.

Section A -- Words and Symbolic Thought

> The difference between words and pictures is the difference between
> telling and showing.
>
>   -- William Bowman[1]

> Often we have to get away from speech in order to think clearly.
>   -- Woodworth[2]

A traditional theory of psychology is that the mind manipulates words in the process
of thinking. As late as 1921 Edward Sapir in his book *Language* wrote, "Thought may be
a natural domain apart from the artificial one of speech, but speech would seem to be the
only road we know of that leads to it."[3] In other words, language is the only discernible
medium available for thought. Indeed, the capacity for symbolic reasoning has come to be
a determining factor in measuring "intelligence". Undeniably words and symbols *are*
important to thought, but their precise *role* in the mind has never been well defined or
understood. "[Psychologists] know a good deal about what thinking does but little about
what it is.... A principal question remains. What are the mental shapes of thought?"
[Arnheim[4]] (This question, and this entire thesis, deals with the highest levels of cognitive
activity. Obviously at the neuro-chemical level of individual brain cells, it makes little sense
to talk about "shapes of thought". We are investigating and attempting to aid high level
mental processes which are distinguishing characteristics of intelligent human beings.)

Suppose we remove words from their meanings and limit them to their actual
shapes, i.e. examine language as language. The perceptual dimensions of language -- its
sounds and written appearance -- are unstructured. Little information can be communicated
to someone who does not know a language just by presenting him with spoken or written
words. (Though some languages, such as hieroglyphic languages, do use pictures of objects
as words.) Language has meaning only insofar as individual words and phrases are
understood to have meaning, i.e. only if we understand to what the words refer, both
externally in the world and internally within ourselves. "Our ability to apply names and
descriptions to objects in the world has to be mediated by analogical representations. For
instance, one can define a word such as 'plank' in terms of other words, such as 'straight',
'parallel', 'wooden', etc., but eventually one has to say of some words, to a person who claims
not to understand them, 'You'll just have to learn how things of that sort *look*.'"[Sloman[5]]
Words are *"signs"*, i.e. they stand for a content without representing its structure. (This
terminology is defined in section G.) The mind must have something *structural* to
manipulate.

Hayes[1973] has presented evidence that images are used even in such non-pictorial
domains as arithmetic and algebra. Most mathematicians visualize their formulae and
manipulate them as structure. Martin[6] agrees that the ability to visualize a problem is a
useful skill in solving it, and that to mathematicians, notation is a form of graphical
visualization. In 1945 Jacques Hadamard conducted a systematic survey of the creative
methods of American mathematicians. He concluded: "My mental pictures are exclusively
visual.... About the mathematicians born or resident in America, whom I asked, phenomena
are mostly analogous to those which I have noticed in my own case. Practically all of them

... avoid not only the use of mental words but also, just as I do, the mental use of algebraic or any other precise signs ... The mental pictures of the mathematicians whose answers I have received are most frequently visual, but they may also be of another kind -- for instance, kinetic."[7]   Einstein, replying to the survey, felt compelled to emphasize in a personal letter to Hadamard:

> The words or the language, as they are written or spoken, do not seem to play any role in my mechanism of thought. The physical entities which seem to serve as elements in thought are certain signs and more or less clear images which can be 'voluntarily' reproduced and combined.... Taken from a psychological viewpoint, this combinatory play seems to be the essential feature in productive thought -- before there is any connection with logical construction in words or other kinds of signs which can be communicated to others. The above-mentioned elements are, in any case, of visual and some of muscular type. Conventional words or other signs have to be sought for laboriously only in a secondary stage, when the mentioned associative play is sufficiently established and can be reproduced at will.[8]

Arnheim finds in art parallels between visual perception and cognitive processes: "Artistic activity is a form of reasoning, in which perceiving and thinking are indivisibly intertwined. The person who paints, writes, composes, dances ... thinks with his senses. This union of perception and thought [is] not merely a specialty of the arts ... The remarkable mechanisms by which the senses understand the environment are all but identical with the operations described by the psychology of thinking. Inversely, there [is] much evidence that truly productive thinking, in whatever area of cognition, takes place in the realm of imagery."[9]

Images are derived from sense perceptions. They may be visual, auditory, kinesthetic, olfactory, and may even use taste. This thesis concentrates on visual images because (a) they have powerful representational capabilities; (b) operations on visual images are versatile, yet simple; and (c) computer technology is at a stage where visual processing can be effectively augmented. Arnheim summarizes the relative potential of words and images in productive thought:

> Concepts are perceptual images and ... thought operations are the handling of these images ... Images come at any level of abstractness. However, even the most abstract among them must meet one condition. *They must be structurally similar (isomorphic) to the pertinent features of the situations for which the thinking shall be valid.*[10] [emphasis mine]

> ... [While] language can supply information by what Kant calls analytical judgments, ... purely verbal thinking is the prototype of thoughtless thinking, the automatic recourse to connections retrieved from storage. It is useful but sterile. What makes language so valuable for thinking, then, cannot be thinking in words. It must be the help that words lend to thinking while it operates in a more appropriate medium, such as visual imagery.

> The visual medium is so enormously superior because it offers structural equivalents to all characteristics of objects, events, relations. The variety of available visual shapes is as great as that of possible speech sounds, but what matters is that they can be organized according to readily definable patterns, of which the geometrical shapes are the most tangible illustration. The principal virtue of the visual medium is that of representing shapes in two-dimensional and three-dimensional space, as compared with the one-dimensional sequence of verbal language. This polydimensional space not only yields good thought models of physical objects or events, it also represents isomorphically the dimensions needed for theoretical reasoning.[11]

Arnheim's notion of "isomorphic" here is not to be confused with the mathematical concept of "isomorphism". Rather it means that there is an element in the representation for each feature of the concept represented that the mind considers to be relevant. For complex concepts this requires considerable representational flexibility. Perceptual images, particularly visual images, provide more descriptive power than do words.

Section B -- Conscious Thought and Short Term Memory

Experiments in psychology indicate that the brain possesses at least two kinds of memory, "long term" and "short term". (There are apparently finer divisions of each, but this is the gross organization.) The "long term" memory (LTM) contains all the information that a person is able to recall, either voluntarily or involuntarily. We will make no assumptions in this paper as to how information is stored in the long term memory. The "short term" or "working" memory (STM) contains the objects currently being dealt with in conscious thought. Much psychological research is being done on the nature of such objects. Representative experiments are described later in this chapter which support Arnheim's theory that *visual imagery* is an exceptionally productive medium.

In my view, words normally play two roles in thinking:

(1) Words suppress detail, keeping the brain's limited-capacity short term memory from overflowing.

(2) Words provide access paths to more complex structures in the long term memory, which are then retrieved and manipulated directly in thought. Thus words are indices, pointers to concepts.

Psychologists are in substantial agreement that the information capacity of the short term memory is limited. Miller, in a classic paper[1967], proposed the "magic" number 7 +- 2 as the size of the STM. The STM can hold between 5 and 9 "chunks" of information (the number now appears to be as small as 4 or 5 [Chase and Simon[1973]]), but the *size* of each chunk can vary. More total information can be stored when an *organization* is imposed on

individual pieces of information than when the individual pieces are stored directly. Moran[1973] reports an experiment in which subjects were asked to remember a sequence of directions of the form: north, east, south, east, south, west, ... The experiment was designed so that only the short term memory was involved. The subjects were able to remember quite long sequences (two-to-three times the usual capacity of the STM) by *visualizing* a line drawing of the directions:
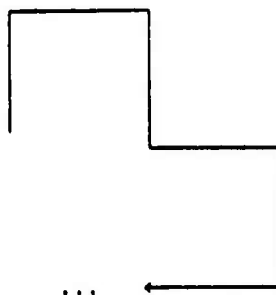
North, east, south, east, south, west, ...

An interesting corollary of Moran's experiments is that the STM seems to have *spatial* as well as *quantitative* limits. With one sequence, subjects complained that the path "keeps dragging out" to the east[12], causing space bounds to be exceeded.

Visual organization is an extremely efficient chunking method. A closed curve that can be remembered topologically contains an infinite amount of information about individual points. One thing experimenters usually do with data is plot it on a graph or in a table, to organize it visually. Concepts become clear only when the overall structure of individual pieces of information can be grasped.
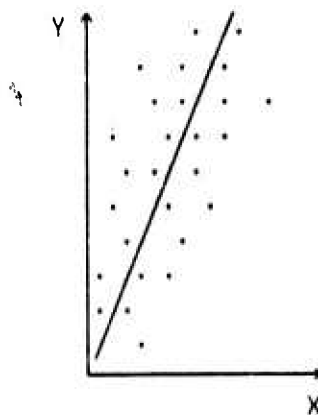
In computer programming, "structured" programming [Dijkstra[1972]] is an attempt to organize the process of programming by chunking related activities into separate stages. "Automatic" programming [Balzer[1972]] is similar to structured programming except that some of the stages are left for the computer to complete. These two concepts share an important characteristic with visual organization in the mind: they enable more information -- more of the structure of a problem -- to be contained in the conscious memory. We must be able to grasp the whole in all its fuzzy detail as well as to investigate parts of it as minutely as necessary. However it is ironic that "structured programming" has dealt with *unstructured* programming languages. They are "unstructured" in the sense that programs have no spatial organization, and data is designated abstractly, without displaying its structure. A "structured" programming language would use structure in its semantics. As Bowman points out, "spatial quality in the graphic figure is defined by the form that it organizes."[13] In other words, the spatial relationship *between* elements in a picture has as much to do with meaning as do the elements themselves. But "linear" programming languages have no spatial structure at all. (By "linear" is meant a verbal language such as English, consisting of a sequence of words.) One of the few characteristics of "linear" languages that even approaches spatial organization is indentation:

```
IF ... THEN
      WHILE ... DO
            BEGIN
              .

              .

            END
   ELSE ...
```

At best this is only an indirect indicator of meaning. The vast potential of multi-dimensional communication is simply not realized in linear languages because they are spatially unstructured.

This paper primarily deals with *conscious* thought, concentrating particularly on *visual thinking* as a basis for designing a graphical computer interface. The model of visual thinking presented so far may be summarized as follows.

(1) Conscious thought deals with concepts in the short term memory which are retrieved (in some fashion) from the long term memory.

(2) Concepts in the short term memory are metaphorical images derived from sense perceptions. Words are pointers to full concepts and are used to suppress unnecessary detail.

(3) Thought operations are transformations of images.

PYGMALION uses a computer display as a computational extension of short term memory. Operations are provided to retrieve and store images between the display screen and long term memory (secondary storage). No assumptions are made as to the structure of long term memory items. The critical restriction is to *concepts that enter and are manipulated in the short term memory as perceptual images.*
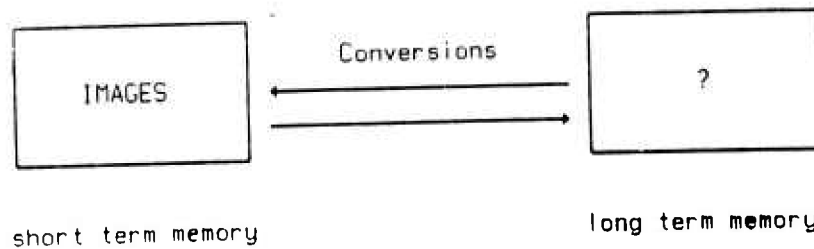
By patterning itself after this model, PYGMALION hopes to stimulate and assist such thinking processes in people.

Section C -- Analogical versus Fregean

To put the preceeding sections into different terms, Sloman distinguishes two kinds of systems: "analogical" and "Fregean" (after G.Frege, the inventor of predicate calculus). Analogical systems are *analogous* in structure to the things they describe; Fregean systems bear no such resemblance. Natural language, for example, is Fregean for most tasks, whereas architectural diagrams are analogous (at some level) to the buildings they describe.

> In an analogical system ... the structure of the representation gives information about the structure of what is represented. As two-dimensional pictures of three-dimensional scenes illustrate, the correspondence need not be simple.... The interpretation of an analogical representation may involve very complex procedures, including the generation of large numbers of *locally* possible interpretations of parts of the representation and then searching for a *globally* possible combination.... By contrast, in a Fregean system there is basically only *one* type of "expressive" relation between parts of a configuration, namely the relation between "function-signs" and "argument-signs"....
> The structure of such a configuration need not correspond to the structure of what it represents or denotes. At most, it corresponds to the structure of the *procedure* by which the object is identified, such as the structure of the route through a complex "data structure".[14]

Fregean representations have the advantage that concepts which are difficult to represent (such as universals and non-ostensive concepts) can still be manipulated, since the representation is always the same. However, as discussed in the last section this is not as powerful a medium for conscious thought as is visual imagery. Concepts which cannot be represented visually are more difficult to "think about" than visual concepts, since the transformations are more abstract. Occasionally the words themselves become images for

non-visual concepts like "time", "space", "infinity", "love", "justice". "Infinity" is not an eight-letter concept; it has many dimensions, each of which can be retrieved and dealt with on demand. The collection is represented in the image INFINITY, or perhaps ∞. Even the mathematician G.Polya, who claims he thinks with words, "does not use words as equivalents of ideas, since he uses *one* word or one or two letters to symbolize a whole line of thought; his psychological process would be in agreement with Stanley's statement that 'language, as an indicator, can only indicate by suggesting to our consciousness what is indicated, as object, thought or feeling...'"[Hadamard[15]] Hadamard said he never met another man who responded as did Polya.

One of the advantages of analogical representations over Fregean ones is that structures and actions on structures in the metaphorical context have a functional similarity to structures and actions in the represented context. It is less likely that operations will be applied to analogical representations which would be illegal in the other context. "When a representation is analogical, small changes in the representation (syntactic changes) are likely to correspond to small changes in what is represented (semantic changes). Changes all in a certain *direction* or *dimension* in the representation correspond to similarly related changes in the configuration represented, and constraints in the problem situation ... are easily represented by constraints in the types of transformations applied to the representation, so that large numbers of impossible strategies don't have to be explicitly considered, and rejected. Hence 'search spaces' can be efficiently organised. By contrast, the sorts of changes which can be made to a Fregean, or other linguistic, description, such as replacing one name with another, ... are not so usefully related to changes in the structure of the configuration described." [Sloman[16]]

The notion of an analogical representation is task dependent; descriptions analogical in one domain may be Fregean in another. Even FORTRAN is analogical when dealing with mathematical formulae:

$$f(x) = a\, x^2 + b\, x + c$$

$$F = A * X**2 + B*X + C$$

But FORTRAN is Fregean and obscure for specifying change, the flow of control, or data structures other than numbers.

14

Section D -- Mental Images

Without a presentation, intellectual activity is impossible.

-- Aristotle[17]

Thinking can deal with objects and events only if they are available to the mind in some fashion.

-- Rudolf Arnheim[18]

What are mental images like? The branch of psychology known as eidetics holds that they are faithful replicas (eidola) of the actual scene. People possessing eidetic recall can examine their mental images and discover information of which they had not originally been aware, almost like examining a photograph. The neurosurgeon Wilder Penfield has performed operations in which he used an electric probe to stimulate certain areas in the temporal lobes of the brains of his patients. The patients reported the recreation of events in their past:

> All patients agreed that the experience is more vivid than anything they could recollect voluntarily; it is not remembering but reliving. The experienced episode proceeds at its natural speed as long as the electrode is held in place; it can neither be stopped nor turned back by the patient's will. At the same time it is not like a dream or hallucination. The person knows that he is lying on the operation table and is not tempted to talk to people he sees in his vision. Such images seem to approach the completeness of scenes directly perceived in the physical environment; like that outer visual world, they seem to have the character of something objectively given, which can be explored by active perception the way one scrutinizes a painted or real landscape. [Arnheim[19]]

In addition to this involuntary re-creation, the psychologist Erich Jaensch has estimated that 40% of all children and some adults possess voluntary eidetic recall,[20] though eidetic recall seems to deteriorate as children acquire language. Similar results have been obtained for subjects under hypnosis: they are able to recreate events far more accurately than they could voluntarily.

Some indication of the enormous capacity of the brain is furnished by the experiments of Bela Julesz[1971]. Julesz used "random-dot stereograms" to study the physiology of perception. These are arrays of computer-generated random dots, typically 100 x 100. When viewed superimposed, the arrays appear to contain a figure such as a diamond raised above a random background. (The reader may try crossing his eyes to superimpose the following stereogram, or he can refer to Julesz's book.)
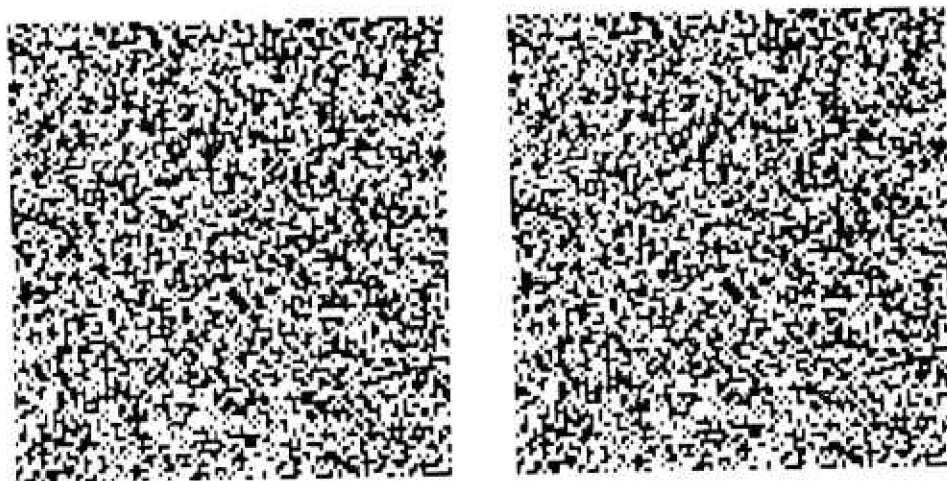
A Random-Dot Stereogram [Julesz[21]]

The figure will appear only if the superposition is exact; small displacements or inaccurate reproduction of one of the arrays destroys the effect. In a classic sequence of experiments, one stereogram was shown for a few seconds to a subject possessing eidetic recall. After a ten minute interval the subject was shown the other image and correctly superimposed them to detect an inverted T-shaped area. Furthermore the area was sharply defined, which is remarkable since blurring of either array causes the corners to appear rounded off. The time between images was then extended to 24 hours with another set of stereograms. Again the eidetiker correctly detected a raised square area. "That 10,000 picture elements of a random-dot texture could be precisely stored for 24 hours without loss of detail argues that at least a select few of Homo sapiens have a detailed texture memory."[Julesz[22]] The subject was then shown a series of stereograms to determine the ultimate capacity of her eidetic memory; *she was able to reproduce with complete accuracy four 100 x 100 random-dot arrays.*

Experiments like these indicate that the mind is capable of storing sensory data in great detail. Furthermore, they suggest that the precise temporal order and rate of events is preserved. Under the proper conditions this information can be regenerated, "read out" like a detailed computer dump.

This is the full "data base" of the mind. However it is not the data of thought. "The kind of 'mental image' needed for thought is unlikely to be a complete, colorful, and faithful replica of some visual scene. But memory can take things out of their contexts and show them in isolation."[Arnheim[23]] Thought deals with fragments, transformations, abstractions and combinations of incomplete scenes. The discerning mind selects only those traits which are relevant to its present goals. In his article "What People Dream About," Calvin Hall[195] wrote that in ten thousand dreams reported from both men and women,

21% of the characters in the dreams were not identified as to sex. Though obviously a characteristic of every human being, sex was not a relevant dimension in the dream report, so it was ignored by the mind. Something quite different is going on here from what Penfield reported. The psychologist Edward Titchener summed up the difference (in 1909): "The incompleteness of the mental image is not simply a matter of fragmentation or insufficient apprehension but a positive quality, which distinguishes the mental grasp of an object from the physical nature of that object itself. [It is an error to assume] that the mind's account of a thing is identical with all or some of the thing's objective properties."[24] In fact, as Colby points out[25], "intelligence" might be defined as the ability to make appropriate selections leading to desired goals achieved under varying circumstances.

Plato first advanced the concept that mental images are built up out of successive sense impressions, like transparent photographs stacked one on top of the other. Gradually the common elements from many different horses will emerge in an "ideal" horse. Philosophers for two thousand years have struggled with the problems created by "ideal" images. How can we have a concrete image of a horse without assigning it some attributes of actual horses? How detailed can an "ideal" image be? Is a three-legged horse still a horse? A two-legged horse? A horse with no head? "None of us, I believe, carries in his head such schematic pictures of bodies, horses, or lizards as [this] theory postulates. What these words conjure up will be different for all of us, but it will always be an elusive welter of fleeting events which can never be communicated in full."[Gombrich[25]] Colby[26] speculates that perhaps memory is generative: specific memories are processes which generate the item remembered. For example, we remember how to swim by generating the actions involved.

Mental images are not limited to concrete physical objects. Many fascinating books have been written on the range of concepts that can be expressed visually.[27]

A Visual Representation of "Rhythm"

Recently several experimenters have asked their subjects to represent such abstract concepts as Past, Present and Future, Democracy, Good and Bad Marriage, and Youth in drawings. Most drawings successfully captured some essential dimension of the concept.
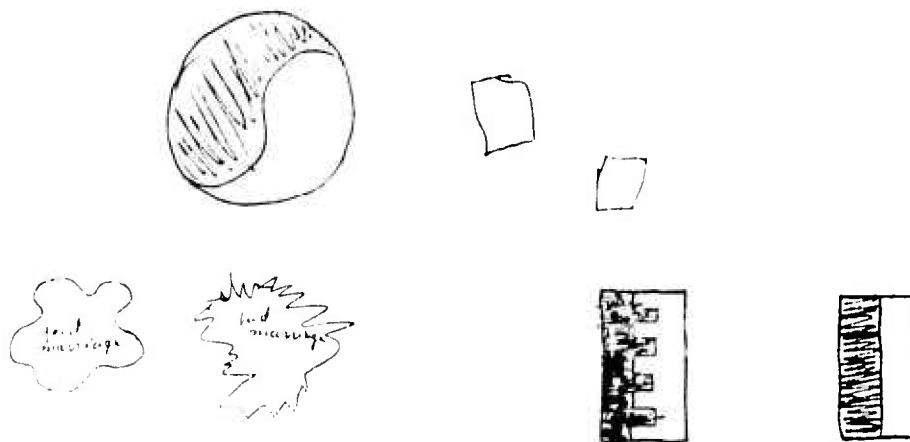
Left: Good Marriage; Right: Bad Marriage [Arnheim[28]]

Even for concrete objects, images need not look like the objects. Images may be classified as "mimetic" or "non-mimetic" depending on whether or not they resemble the objects or concepts in physical appearance.
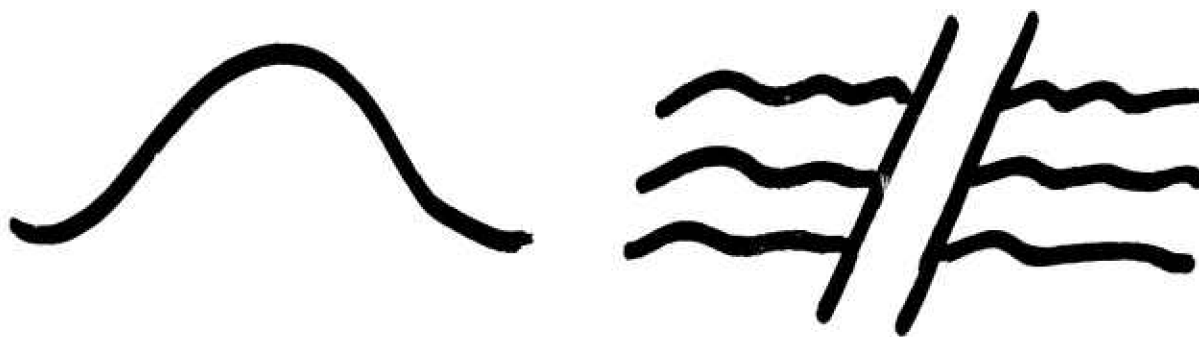
Left: Mimetic Image of "Hill"; Right: Non-mimetic Image Stressing Dimension of "Barrier"

Current research in psychology (see, for example, the Eighth Carnegie Symposium on Cognition) is beginning to gather quantitative experimental evidence on the *nature* of visual images, rather than merely trying to demonstrate their *existence*. Arnheim suggests that mental processes involved in thinking with images are similar to those involved in the

perception of images. Cooper and Shepard suggest that perhaps the *same mechanism* is involved, i.e. the same functional units in the brain. They note selective interference experiments in which a subject is asked to form a mental visual image of an object. While he is doing this, his ability to detect and recognize other externally-presented visual images is reduced. The same thing happens with auditory images: while thinking of a tune, the subject is less able to detect and recognize other sounds. However, visual images interfere *less* in the detection of audio signals than do audio images, and vice versa. [Cooper and Shepard[1973]] This indicates that the image processing units in the brain are discrete according to the type of image.

As opposed to interference, *reinforcement* occurs if the external signal is "compatible" with the internal image. The subjects show *improved* detecting ability when external and internal images are similar in form. In fact, in that situation subjects forming mental images do better than subjects forming no images at all. "Performance should then be best when the imagery is in a form (modality, structure, or system) that is most -- not least -- similar to the form of the externally presented material." [Cooper[29]] *The quickest mental response occurs when the representation in the medium is closest to the representation in the mind.*

Section E -- The Problem of an 'Articulate' Representation

I consider it a heresy to think that any painting as such records a sense impression or a feeling. All human communication is through symbols, through the medium of a language, and the more articulate that language the greater the chance for the message to get through.

-- E.H.Gombrich[30]

"What is this?"

"A steamboat."

"And that scribble over there?"

"That is art."[31]

The fundamental problem for PYGMALION is to design an ARTICULATE LANGUAGE for communication.

Webster defines "articulate" as "expressing or expressed readily, clearly, or effectively."[32] The most readily, clearly and effectively expressed information is that which requires the least translation between the internal representation in the mind and the

external representation in the media.   An articulate form for a problem is one which corresponds closely to the form used in the mind in thinking about the problem. (This is problem dependent, the mind uses different representations in different situations.) Let us define the *"translation distance"* between two representations as the number of transformations that one representation must be put through to make it identical to the second representation.   Then we can summarize the results in the previous section in a "minimum translation principle".

> **MINIMUM TRANSLATION PRINCIPLE:** The most articulate representation for a subject is at a minimum translation distance from the mind's representation.

Like Arnheim, Cooper and Shepard report that their experiments "establish that the internal representations and mental operations upon these representations are to some degree analogous or structurally isomorphic to corresponding objects and spatial transformations in the external world."[33] This finding is restated in the following corollary.

> **COROLLARY:** Analogical systems are more articulate than Fregean systems for extensional problems.

Programming languages and environments have almost all been Fregean systems. Computers normally accept problem solutions only in a static, linear form. The programmer must translate his mind's representation into this static, linear representation in a text file. The translation distance is large. The *inarticulateness* of a text representation explains the following phenomena: Most programmers have difficulty understanding someone else's program given just a listing of the source code. They have less trouble if they can talk to the programmer directly and get the program explained to them. They usually have even less trouble if a blackboard or other multi-dimensional medium can be used as part of the explanation. (Baecker at Toronto is presently engaged in *animating* program semantics. His computer animations communicate the meaning of programs more effectively than anything else I have seen. Cf. Chapter 3.) Really articulate communication can "open our eyes" to meaning. "Salvador Dali's way of letting each form represent several things at the same time may focus our attention on the many possible meanings of each colour and form -- much in the way in which a successful pun may make us aware of the function of words and their meaning." [Gombrich[34]] An articulate representation may illuminate new and hitherto unrealized aspects of a problem and lead to a creative solution of it. With a suitable representation, the problem-solving abilities of a person using a computer can be vastly increased.

ABSTRACT   CONCRETE

10.5
3.6
9.3
14.7
6.2

X

Two Ways of Designating a REAL ARRAY 5 Long

What *is* an "articulate form" for communicating with computers? It is generally agreed that machine language programs are less clear than programs in higher-level langauages like FORTRAN or ALGOL. It is also true that flow charts of FORTRAN programs are (usually) more articulate than the programs themselves. IBM requires its programmers to document their programs with flow charts in addition to the normal write-ups. Some languages have attempted to use flow charts themselves as program notation (e.g. GRAIL -- see Chapter 3-B). But no one would claim that the limit of representational clarity has been achieved.

As everything so far suggests, I lean toward higher-dimensional representations. From an information processing standpoint the superiority of two-dimensional communication over one-dimensional has been well documented through studies such as Miller's. From an educational standpoint educators like Piaget and Dewey have long known that the concrete is easier for children to understand than the abstract. Much concept learning proceeds from the concrete to the abstract and from particulars to the general. Scientists since the Egyptians have used diagrams as an aid to solving problems and for communicating results to fellow scientists. Communication is concrete; understanding involves abstraction. In computer science an increasing use is being made of graphical (even 3-D) presentation of data. And finally the model of thinking we are using asserts that regardless of how abstract mental images are, they "must meet one condition. They must be structurally similar (isomorphic) to the pertinent features of the situations for which the thinking shall be valid." [Arnheim35] This requires more than one dimension for all but the simplest problems.

Section F -- The 'Innocent Eye' and Other Myths

> The forms of art, ancient and modern, are not duplications of what the artist has in mind any more than they are duplications of what he sees in the outer world. In both cases they are renderings within an acquired medium, a medium grown up through tradition and skill -- that of the artist and that of the beholder.[36]

> ... Any representation must of necessity allow of an infinite number of interpretations and ... the selection of a reading consistent with our anticipations must always be the beholder's share.[37]

> ... The interpretation of all images [is] a philosophical problem.[38]
> -- E.H.Gombrich

Instead of dealing with complete, finely-detailed information, thought processes tend to *abstract* information down to a skeleton of what the mind considers (at the time) to be relevant features. This is necessary to prevent the mind from being overwhelmed by a mass of details. "It is a quality invaluable for abstract thought in that it offers the possibility of reducing a theme visually to a skeleton of essential dynamic features, none of which is a tangible part of the actual object." [Arnheim[39]] In art, caricaturists since the sixteenth century have been aware that a single stroke of the brush could significantly alter the information content of a picture. A single line is capable of being seen as a complete scene, indeed a complete event. "The humble suitor is abstracted to the flash of a bent figure. And this perceptual abstraction takes place without removal from the concrete experience, since the humble bend is not only understood to be that of the humble suitor but seen as the suitor himself." [Arnheim[40]]

The impressionists elevated this principle to high art. As Gombrich tells us, the beholder of an image always tries to *project* his internal preconceptions onto the external world. He interprets an image not only in terms of its *content* but also in terms of the *context* in which the image occurs and in terms of his *expectations* of what the content will be. Gombrich notes "a well-known experiment that a familiar shape will induce the expected color; if we cut out the shape of a leaf and of a donkey from identical material and ask observers to match their exact shade from a color wheel, they will tend to select a greener shade of felt for the leaf and a grayer one for the donkey." [Gombrich[41]] The reason impressionist paintings often seem more accurate to modern viewers than medieval or renaissance paintings is that the spare, suggestive lines of the impressionists leave more latitude for the viewer to fill in missing detail by projecting his preconceptions.
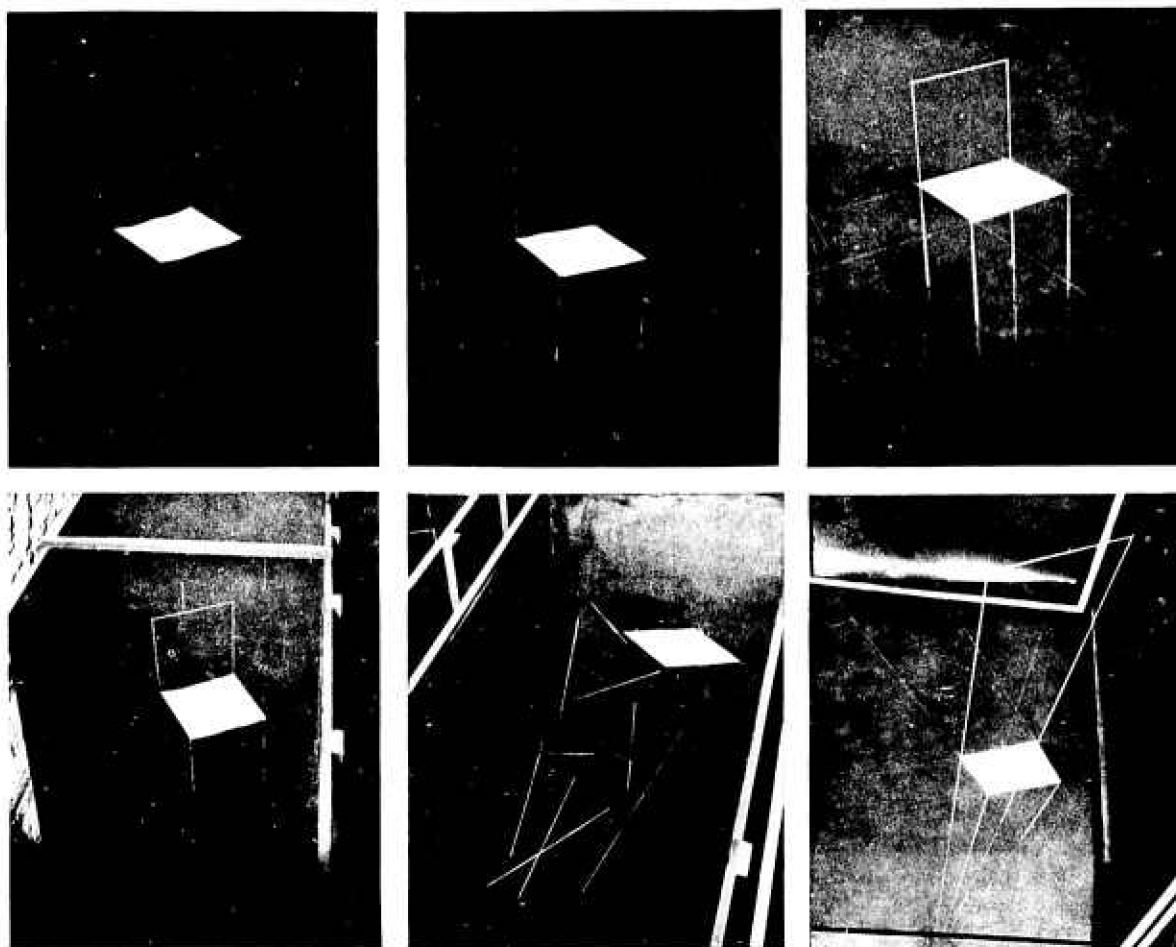
The Ames Chair Demonstrations [Gombrich[42]]

The psychologist Adelbert Ames has devised a set of clever *trompe l'oeil* demonstrations that show very clearly this process of *projection* in the perception of images. The picture above shows three of the demonstrations. When viewed through strategically-placed peepholes, each room appears to contain a chair. But when viewed from above, two of the rooms actually contain only a skewed collection of lines. "What is hard to imagine is the tenacity of the illusion, the hold it maintains on us even after we have been undeceived. We return to the three peepholes and, whether we want it or not, the illusion is there." [Gombrich[43]]

The perception of any given picture is heavily dependent on the context in which it is viewed. This is the reason that art has evolved during its history.

Section G -- The Role of Mental Images in Thought

Now assuming the existence of mental images in thought, the next question is: what roles do they play? What functions can they assume? In Arnheim's terms, there are three functions performed by images: (1) *signs*, (2) *symbols*, and (3) *pictures*, in order of decreasing abstractness. (1) An image is a *sign* if it is Fregean, i.e. "it stands for a particular content without reflecting its characteristics ... To the extent to which images are signs they can serve only as indirect media, for they operate as mere references to the things for which they stand. They are not analogues, and therefore they cannot serve as media for thought in their own right ... Numerals and verbal languages ... are the sign media *par excellence*." The other two functions of images are analogic. (2) "An image acts as a *symbol* to the extent to which it portrays things which are at a higher level of abstractness than is the symbol itself." (3) "Images are *pictures* to the extent to which they portray things located at a lower level of abstractness than they are themselves ... A particular image may be used for each of these functions and will often serve more than one at the same time. As a rule, the image itself does not tell which function is intended. A triangle may be a sign of danger or a picture of a mountain or a symbol of hierarchy." [Arnheim44]
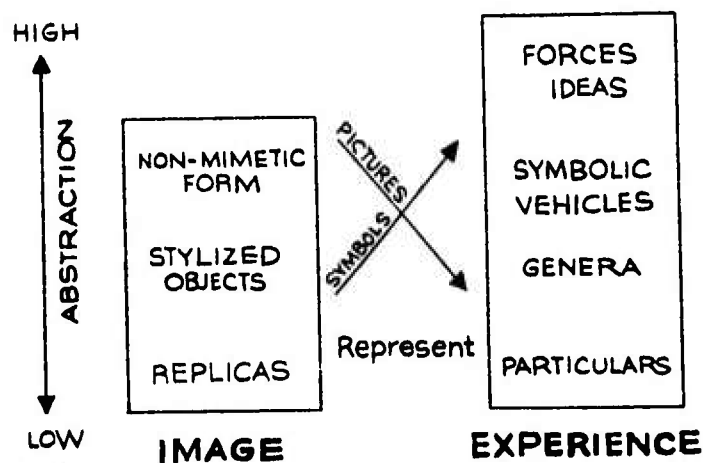
Images as Pictures and Symbols [Arnheim45]

Images are *metaphors* for concepts. They provide an *alternate reality* which is simultaneously concrete in structure and analogic in representation. Koestler wrote of Michael Faraday, whom he termed a 'metaphorical visionary': "He saw the stresses surrounding magnets and electric currents as curves in space, for which he coined the name 'lines of forces', and which, in his imagination, were as real as if they consisted of solid matter."46 The visual medium is an extremely useful metaphorical tool not only because it

has powerful representational capabilities but also because it has a rich set of topological transformations within its own domain. Two- and higher-dimensional media possess far more versatile structural operations than do one-dimensional media.

A system designed to manipulate images should allow them to assume different roles in a program. For example, the programming language LISP permits lists to be used both as data and as program to be evaluated, depending on the context. This provides a formalism for proceeding from the concrete to the abstract. A particular image may first be used as a constant data object, standing for itself, and then it may be used to represent something else, as a variable. The variables in almost all programming languages today are *signs*; their representations (e.g. X) bear no resemblance to the structure of their values. The representation of a problem in most programming languages bears little resemblance to the thought processes that occured in its solution. For this reason most programming languages actually *impede* communication. In a multi-dimensional language the variables can be analogical, *pictures* of their values.

Section H -- Examples of Purely Iconic Reasoning

> Geometrical demonstration must start from the direct visual awareness
> of the fact to be proven.
>
> -- Schopenhauer[47]

The earliest use of analogical reasoning on a computer was Gelernter's "Geometry-Theorem Proving Machine"[1963]. His program employed properties of the representation to guide the proof of theorems. His simple heruristic "Reject as false any statement that is not valid in the diagram" enabled his system to vastly reduce the search space of possible proofs.

Geometry has historically provided a fertile ground for analogical reasoning. In ancient India a branch of geometry developed which used in proofs *only diagrams* and one theorem, the square of the hypotenuse. "Every proposition is presented as a self-contained fact, relying on its own intrinsic evidence. Instead of presenting a sequence of steps, the Indian mathematician shows the relevant figure, completed, if necessary, with auxiliary lines and offered with no comment other than the word "Behold!" The proof consists of the evidence visible within the given figure."[48] This is analogical, visual thinking in its purest form. Examples of such proofs follow.

The triangle based on the diameter of a circle is always right-angled. [Arnheim[48]]
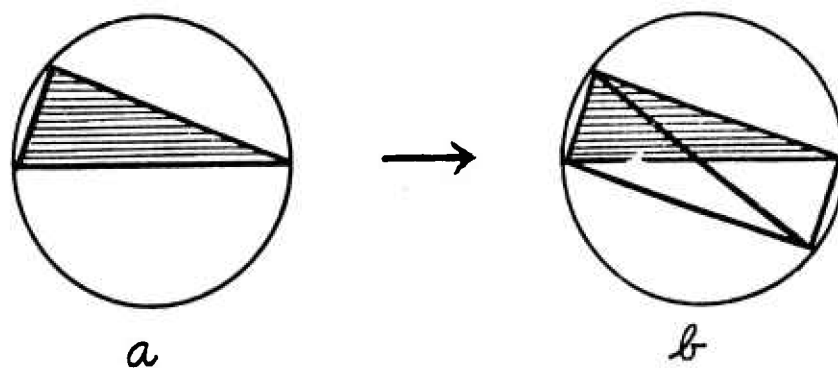
BEHOLD !

$(A + B)^2 = A^2 + 2AB + B^2$ [Arnheim[49]]
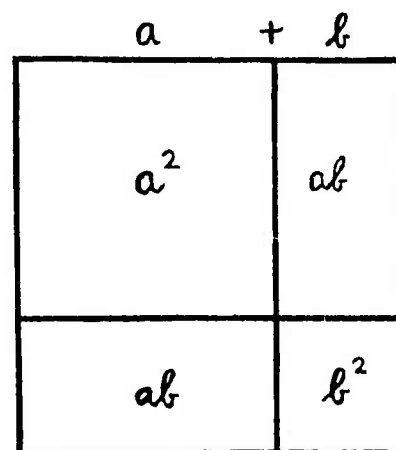
BEHOLD !

Pythagoras' original proof of the Pythagorean Theorem:

For all right triangles A B C, $A^2 + B^2 = C^2$ [Courtesy of Alan Kay]
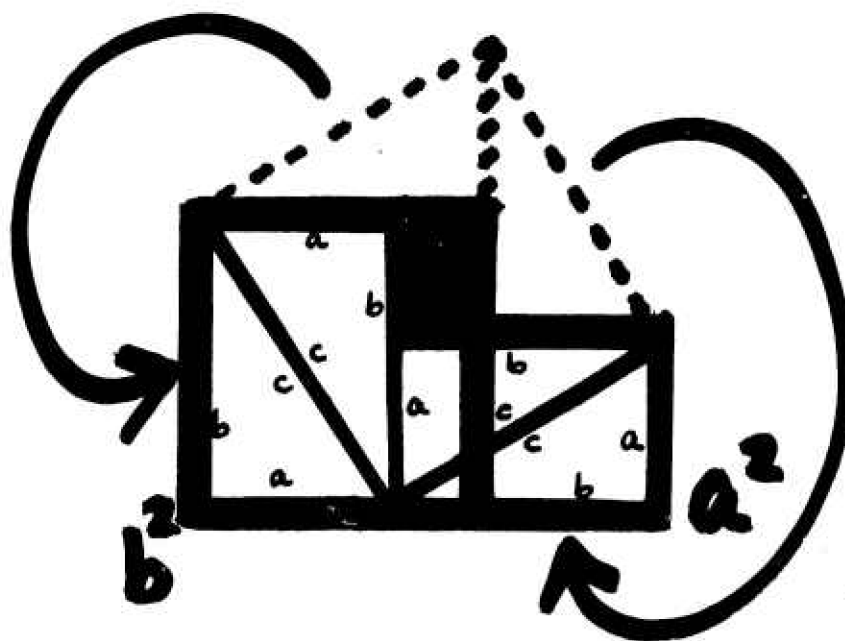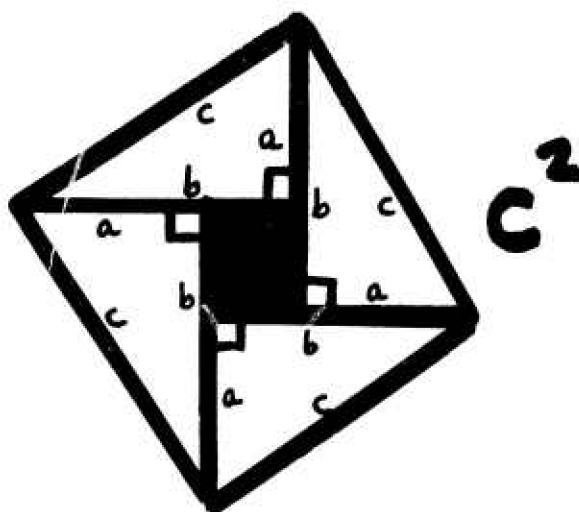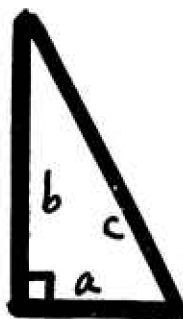
BEHOLD !

As Arnheim points out, mathematics so firmly related to perceptual evidence "can arouse keen interest in unspoiled people. This is observed in the response of young children to structural algebra and arithmetic. It is equally true for the person of mature mind. If he is forced to perform at a level at which the task can only be solved by memorized routines, his reasoning will protest or dry up. If instead he can operate in such a way that *perception invites comprehension*, he will realize by his own experience why [Berthold] Brecht makes his Galileo say: 'Thinking is among the greatest pleasures of the human race.'"[50]

Section I -- Concrete and Acti e iedia in Education

The ideas first in the mind, it is evident, are those of particular things, from whence, by slow degrees, the understanding proceeds to some few general ones; which being taken from the ordinary and familiar objects of sense, are settled in the mind, with general names to them. Thus particular ideas are first received and distinguished, and so knowledge got about them; and next to them, the less general or specific, which are next to particular. For abstract ideas are not so obvious or easy to children, or the yet unexercised mind, as particular ones. If they seem so to grown men, it is only because by constant and familiar use they are made so.

-- John Locke[51]

In an abstract setting these concepts are difficult. Concretized in suitable projects in [a computer] laboratory they are perfectly accessible.

-- Seymour Papert[52]

Further foundation for the theory that communication is concrete and that understanding involves the abstraction of concrete information is supplied in some current theories of education. Educators such as Dewey and Piaget have stressed that *doing* is better than *telling*. Children learn more from direct experiences than from vicarious ones. Direct experiences are those that a child actually does or that actually happen to him. Vicarious experiences are those in which he participates indirectly, such as by reading about them or being told about them. With the advent of television, a vast amount of vicarious experience has been opened to every child. "it has shifted the balance between direct and vicarious experience towards vicarious experience for all of us and has done so most strongly for the young. Instead of information poverty, they now experience information richness. Schools as they now exist were designed for an information-poor society, in part to provide a child with vicarious experience from books and contact with the teacher. Obviously that function was radically altered by television, radio, and other media outside the school." [Coleman[53]] Some teaching methods, such as the Montessori method, emphasize direct experience and individual guidance. The child is encouraged to

actively interact with his environment. "Although the school is no longer necessary to provide information, it is more important than ever for developing skills for the management of information."[Coleman[54]] This applies as much to computer science as to education. Computers must provide their users with effective techniques for managing the ever growing quantity of information.

The difference between direct and vicarious experience is the difference between an active and a passive medium. In a passive medium such as television, the user merely observes. He has no control over the events. In an attempt to supply an active theater, the Czechoslovakian exhibit at the 1967 Montreal Expo allowed a movie audience to vote at certain times on the direction the movie could take. But this provides only partial relief from passivity. It is difficult for conventional media to permit participation. To counter passivity, *active* media are needed -- ones that permit the viewer to interact with and influence events. The computer is an obvious candidate because its output is *generative*, as opposed to being fixed or static as in a filmstrip. A computer's output can be changed according to the input. If the user is to establish a cause-effect relationship, the medium must be interactive, so that changes are immediately observable. It must give the user the feeling of *participating* in an experience.

An active medium must also be inherently positive. If a computer is to act as a laboratory for experimenting with ideas, it must not give negative feedback of the form: "You did that incorrectly." Instead something consistent with the situation should happen. If the result is not what was anticipated, it is up to the user to figure out *why*. Contrast this with conventional computer-aided instruction, in which a child is asked to answer questions and is told that he is either right or wrong. There is little potential in such instruction for a child to experiment with a subject in depth until he is finally satisfied he understands it.

Simulation languages, such as SIMULA [Dahl[1966]], embody many of the laboratory-like capabilities of active media. Whenever a model is constructed of a given situation and then systematically changed to investigate different effects, the experimental paradigm is in effect. But two difficulties with most simulations are: (1) They have not been immediately interactive. When a parameter is changed, the effect cannot be observed until some time later. (2) The representation of the parameters and of changes to the parameters have not been in a natural notation. A translation of the parameters is required into some form, say a series of numbers, that the computer can accept. The PYGMALION laboratory attempts to overcome these shortcomings by immediate interactive feedback and by flexibility in the definition of representations. Fubini admirably expresses the delights of a true computer laboratory when he says, "Perhaps the most moving and impressive show I have seen is the simulation of the universe, and it is still in my eyes. A random population of uniform bodies obeying Newton's Law was injected in a simulated space. Then in a few seconds right in front of me, the moving bodies in apparently random motion acquired shape. It was thrilling to see spiral, nebular, globular galaxies appear in completely unexpected fashion. I did not truly realize the shape of the universe was defined in as gross morphology by Newton's Law alone. The ability to simulate physical and sociological structures and to examine in detail the effects of changes is one of the most powerful pedagogical tools I have seen. It can be used to teach in concrete and immediate form the ultimate consequences of a law or set of relations, and it can be used to teach a student to find the relations himself."[54] This is an admirable statement of the goals of an active medium. How often with ordinary programs and languages has such enthusiasm been displayed?

The concreteness and immediacy of computers is beginning to be appreciated by contemporary educators. Computers are a tool for *managing information* and *conducting experiments*, as such they can *help people to think and learn*. Several educators state the case for computers. Robert Filep: "Children can see immediate payoff on a computer in even the most minimal demonstration of their skills. I believe these factors have much to do with the students' excitement about using computers." Mark Greenberger: "The computer is very concrete. It is real -- something students can get their hands on and see the effects of directly. It is a refreshing change from abstract discussion in the classroom." William Huggins: "Students today, at least at the college level, live in a completely symbolic world: a world of symbols, mathematics, and words. They do not get their hands on active experiences that give symbols meaning and physical definition that produce intuitions that I find missing."[55]

In the future education will shift toward the teaching of strategies for learning and managing information. Computer interaction will shift toward strategies for actively assisting thinking.

Chapter 2

On Creativity

We may proceed to define the creative thinking process as the forming of associative elements into new combinations which either meet specified requirements or are in some way useful. The more mutually remote the elements of the new combination, the more creative the process or solution.

-- Sarnoff Mednick[1]

Summary

(1) Creative thinking involves the juxtaposition of two normally-distinct contexts, using elements from both in a new and productive way.

(2) The creative product is novel, appropriate to the situation, a transformation of contexts, a condensation having summary power.

(3) Creativity is less logical than deductive reasoning. It involves emotion and may even involve chance.

Section A -- The Nature of Creativity

> We cannot judge expression without an awareness of the choice
> situation, without a knowledge of the organon ... Where we have no
> matrix, no keyboard, we cannot assess the meaning of an individual
> feature.
>
> -- E.H.Gombrich[2]

Creativity has only been examined quantitatively in the 20th Century. Psychologists have long been uncertain of its nature. Galton thought that creativity involved a collection of qualities, among them what he called "fluency", i.e. "an unusual and spontaneous flow of images and ideas." The creative mind is "always pullulating with new notions." Other qualities are "receptivity" and "intuition of insight". James called the latter quality "sagacity", and T.S.Eliot named it "sense of fact". McDougall added the quality of "deviant association". But what these *really are* is a source of mystery. The word "creativity" was not even defined in the Oxford English Dictionary until its later editions.[3]

One school of thought has held that creativity is not the gift of the individual but the spirit of the age. "An idea whose time has come," we often hear. Goethe called it *Zeitgeist*. For example, had Copernicus, Kepler, and Newton died early in life, their contemporaries eventually would still have discovered the laws of gravity and motion. Indeed environment does play a part in the creative act. Had Kepler not formulated his theories on planetary motion, Newton would not have discovered the law of gravity. (It is interesting to speculate what he might have discovered instead.) But to believe that environment is the whole story is to believe that "if William Shakespeare, like his elder sisters, had died in the cradle, some other mother in Stratford-upon-Avon or Stratford atte-Bow would have engendered his duplicate before the Elizabethan era ended."[4] No one really wants to deny the genius of a Shakespeare or Bach or Picasso, and yet to make a distinction between artists and scientists is to make a distinction between art and science. If there is one thing that psychologists such as Arnheim, Gombrich and Koestler have shown, it is that *the same mental processes are used in science as are used in art.*

Arthur Koestler has written an excellent book on the nature of creativity, *The Act of Creation*. Many of the principles in this chapter are derived from his enlightening theories. Koestler contends that all creative activities have a basic pattern in common, and that comic inspiration, scientific discovery and artistic originality all share the same pattern. "The first is intended to make us laugh; the second to make us understand; the third to make us marvel. The logical pattern of the creative process is the same in all three cases. It consists of the discovery of hidden similarities. But the emotional climate is different in the three panels: the comic simile has a touch of aggressiveness; the scientist's reasoning by analogy is emotionally detached, i.e. neutral; the poetic image is sympathetic or admiring, inspired by a positive kind of emotion.... When two independent matrices of perception or reasoning interact with each other, the result is either a *collision*, ending in laughter, or their *fusion* in a new intellectual synthesis, or their *confrontation* in an aesthetic experience."[5] The same interaction may produce any of these results, depending on the emotional climate of the interaction. The myth of Pygmalion itself is an example. The artistic or lyric aspects of creation as symbolized in the myth are represented in literature by Ovid and in painting by Burne-Jones, who brought Pygmalion's statue Galatea to life in a mystical atmosphere. The

comic attributes of this same event are represented in literature by Shaw and in painting by Daumier, who treated the event in an irreverent, satirical manner. The intellectual aspects have been pursued in a serious manner by Gombrich in his book *Art and Illusion*, and in computer science by this report.

Left: Burne-Jones, Pygmalion, 1878; Right: Daumier, Pygmalion, 1842 [Gombrich[6]]

It is interesting to try to place various phenomena on the continuum between science, humor and art. In mathematics a popular way to refute a proposition is by *reducio ab absurdum*, which extrapolates the consequences of a theorem in a logically valid sequence of steps to a result that is so absurd as to be almost humorous. And frequently mathematicians or computer scientists have criticized solutions to problems not because they were incorrect but because they lacked *elegance*.

> The pattern underlying a creative event is "the perceiving of a situation or idea in two self-consistent but habitually incompatible frames of reference ... While this unusual situation lasts, the idea is not merely linked to one associative context, but bisociated with two."[Koestler[7]]
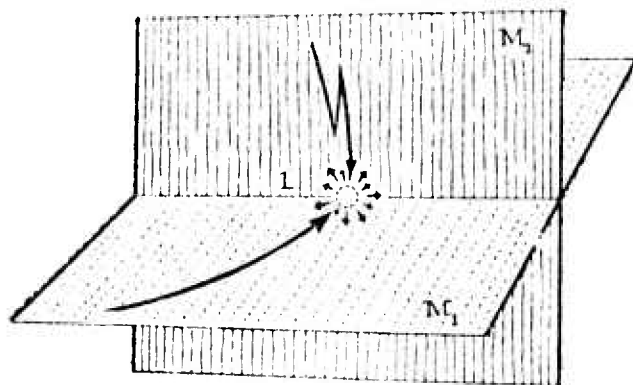
Intersection of Thought Contexts   [Koestler[8]]

This is the ultimate source of the history of art. When faced with a new situation, the artist applies a schema developed in handling other situations, producing a juxtaposition of the two. He brings an established way of looking at things into a new context. The visual contexts of artists are different from those of laymen. Artists have created a code of rules which they obey in looking at pictures. The innovations in art have been modifications of this code. Artists before Constable represented landscapes in "the browns of an old violin" because their code of rules said that distance should be represented by varying shades of brown. Constable brought a new perceptual context to landscape painting -- that of the non-artist, who sees landscapes in shades of green. This led to the codification of a new rule, namely: distance may equally be represented in shades of green as in shades of brown. The drive toward realism has been the attempt to develop a code of rules among artists which is the same as the perceptual code of the viewer. Modern painting has developed a new goal: create codes that are deliberately different from those of the layman viewer, forcing the viewer to adopt new rules for perception.

This juxtaposition of thought contexts, Koestler believes, is transitory and unstable. Sometimes it happens that the thread of thought oscillates rapidly between two frames of reference. More likely though, the established context, the schema, produces an "Einstellung effect". The Einstellung effect is the following: After once thinking about a problem in one way, it often becomes very difficult to think about it in other (even simpler) ways. A simple suggestion at this point may produce a powerful effect. Once the initial suggestion is made, the receptive mind immediately begins exploring the new context. Immediately a host of comparisons present themselves. Conscious thought may remain in the schema, the metaphor, until the initial rush of discovery wanes. Thereafter it may return to the schema periodically for further comparisons and inspiration.
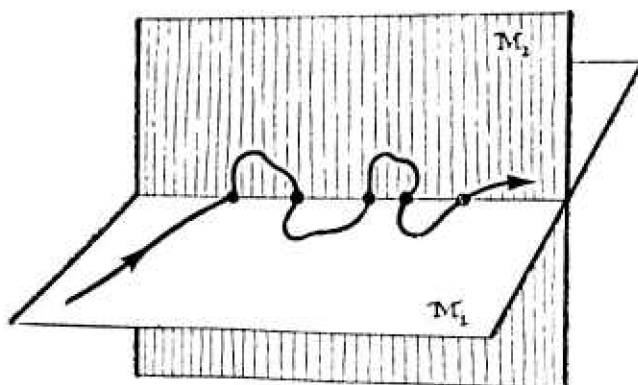
Oscillations Between Two Thought Contexts [Koesler[9]]

"All coherent thinking is equivalent to playing a game according to a set of rules.... In the routines of disciplined thinking only one matrix [set of rules] is active at a time."[10] Therefore it seems plausible that a computer which knows the rules for a particular domain should be able to assist and even stimulate thinking in that domain.

As an example of the way certain images   trigger a whole "train of thought", a chess board with a single piece on it, say a knight, may immediately start one questioning whether a knight can legally touch every square on the board. This leads "naturally" into an entire set of sub-problems. Can the knight get to an adjacent square? Can it get to an adjacent square on the side of the board? Can it get to an adjacent square from a corner? Can it get into a corner? From the results in Chapter 1 we can deduce that such trains of thought are more likely to be initiated by analogical pictures of the chess board than by Fregean words like "chess board" and "knight". Then we might reformulate our definition of "articulate" in operational terms:

> A medium is "articulate" to the degree in which it elicits a productive Einstellung effect in its users.

The process of creative thinking has similarities with the process of perception. The mathematician Poincaré felt that the most productive ideas "are harmonious, and, consequently, at once useful and beautiful."[11] Gombrich applies the metric of "simplicity" to measure the productivity of ideas: "It might be said, therefore, that the very process of perception is based on the same rhythm that we found governing the process of representation: the rhythm of schema and correction. It is a rhythm which presupposes constant activity on our part in making guesses and modifying them in the light of our experience.... In looking for regularities, for a framework or *schema* on which we can at least provisionally rely (though we may have to modify it for ever), the only possible strategy is

2-A   The Nature of Creativity

to proceed from simple assumptions.... This is not due to the fact that a simple assumption is more probably right but because it is most easily refuted and modified.... The simplicity hypothesis cannot be learned. It is, indeed, the only condition under which we could learn at all. To probe the visible world we use the assumption that things are simple until they prove to be otherwise."[12] However, to say that a concept is "simple" is not to say that it is "familiar" or "natural"; many inventions seem bizarre at first.

Section B -- The Great 'Whale Ears' Scandal

(The Use of Schemata in Thought)

> You cannot create a faithful image out of nothing. You must have learned the trick if only from other pictures you have seen.
> -- E.H.Gombrich[13]

> To copy others is necessary, but to copy oneself is pathetic.
> -- Pablo Picasso[14]

Gombrich asserts that artists always begin a picture with a "schema", a framework to serve as a foundation. They "classify the unfamiliar with the familiar."[15] A Dutch artist in the sixteenth century drew a picture of a whale that had washed up on the coast. Whales in the Netherlands are rare indeed, and the artist had undoubtedly never seen one before, particularly not lying on a beach. His picture shows that the whale had ears! "The draftsman probably mistook one of the whale's flippers for an ear and therefore placed it far too close to the eye. He, too, was misled by a familiar schema, the schema of a typical head."[Gombrich[16]] Though unintentional, in a sense the Dutch artist was actually being creative; he created a new kind of animal -- a whale with ear lobes. Who would ever have thought of that?

After Goltzius: Whale Washed Ashore in Holland, 1598 [Gombrich[17]]

On a more practical level the same type of reasoning was done by Johannes ·Kepler in 1609 when he applied ·the schema of Euclidian proportion to the new situation of planetary motion. Kepler's laws state that (1) the orbits of all planets are in the shape of an ellipse with the sun at one focus, and (2) a line between a planet and the sun sweeps out equal areas of the ellipse in equal times. Wilson, in his article "How Did Kepler Discover His First Two Laws?", investigates the question: which came first, the data or the model? "What emerges from Kepler's own account is that he goes on his journey laden with theory, and that he manages to arrive at the two laws only because he approached the problem with a preconception. It is an initial hunch, a physical hypothesis, that guides him throughout."[18] In other words, the ellipse came first. Kepler knew, through Tycho Brahe's observations, that Copernicus' circle model was not correct. The observations together with his area hypothesis suggested a more egg-shaped oval orbit. However since the calculations were so "horrendous" with the oval, Kepler simplified the shape to an ellipse. Why an ellipse? Because the oval resembled an ellipse in shape and because the ellipse is the next simplest closed figure to a circle! Newton said later, "Kepler knew ye Orb to be not circular but oval, & guest it to be Elliptical."[19] Actually Kepler did not start specifically with an ellipse in mind; rather he firmly believed that the eventual solution *would be simple, like an ellipse*. Throughout his investigations Kepler relied on Euclidean geometry to provide a schema for organizing the confusing world of Renaissance astronomy.

Kepler's   investigation   embodies   the   essence   of   the   scientific   method   of

understanding: hypothesis formation, experimentation and observation, hypothesis refutation, followed by hypothesis formation again, until finally an hypothesis remains unrefuted and becomes a "law". What Gombrich and Koestler tell us we can learn from art is that hypothesis formation (the creative link in the chain) involves schemata. We form hypotheses in the same way we paint pictures, not out of nothing but by applying a preconceived schema to the new situation. The mathematical discipline is a schema which has proved invaluable for solving problems in physics. Mathematical formulae are attractive because they have many well-understood properties which can be manipulated independently of their referents, and then mapped back onto their referents transformed. In this sense they are Fregean; their structure does not conform to the structure of the things represented. But nevertheless, as was pointed out in Chapter 1, mathematicians perceive formulae as structured, two-dimensional images. With practice mathematicians are able to attain great skill in manipulating these images. Therefore formulating a fact in mathematical terms is tantamount to "understanding" it at a certain level. For example, the formula

$$cost = \sum_{i=1}^{n} core(t_i) \ (t_{i+1} - t_i)$$

might mean "the cost of executing a program is directly related to the amount of core required at time t_i times the length of time $(t_{i+1} - t_i)$ the core is required." This is an illustration of Koestler's theory of intersecting thought contexts: if one can map a problem into another context that he understands better, he may be able to get a solution more easily. It is the basic idea of linear systems theory: find a mapping from the problem space into a (mostly) separable one.[Kay20]

Transforming a problem *after* one becomes familiar with it may indeed lead to a creative solution, but the question remains: Why do people deliberately *approach* problems with preconceived schemata or biases? Artists are well aware of the quandary of starting "from scratch": "To draw an unfamiliar sight presents greater difficulties than is usually realized."[Gombrich21]. Picasso said that "the most awful thing for a painter is the white canvas."[Wertenbaker22] To create something in an unknown domain is a superhuman task. True genius is able to make a large creative leap; we cannot overestimate the creative contribution of Picasso in inventing cubism or of Buxtehude in inventing the organ style which culminated in Bach. But most creative achievements are transformations on the schemata of their predecessors. In computer science, programmers usually find it far easier to "optimize" an algorithm, to produce a more efficient or elegant algorithm that computes the same result, than to write the original. The use of schema in creativity and the use of projection in analysis are important aspects of the process of creative thinking.

Picasso: The empty canvas (detail from "The Studio at Cannes", 1956) [Wertenbaker[23]]

Section C -- The Role of Emotions in Creativity

How is it that each of the thousands of facts, images and perceptions that we experience every day does not lead to a creative idea? Why are only certain images and percepts used in creative thought? What is it about those images that others do not have? Several noted thinkers have suggested an answer that STAR TREK's unemotional Mr. Spock would scarcely be able to comprehend. The answer suggested is that the *aesthetic* qualities of ideas react with our *emotions* on a subconscious level, and those ideas having a *favorable reaction* (in terms of the individual's personality) are raised to the level of conscious thought.

Poincaré, in his famous address in 1913 on mathematical creativity, made a strong case for the aesthetic.

Among the great numbers of combinations blindly formed by the subliminal self, almost all are without interest and without utility; but just for that reason they are also without effect upon the esthetic sensibility. Consciousness will never know them; only certain ones are harmonious, and, consequently, at once useful and beautiful. They will be capable of touching this special sensibility [that all mathematicians know], ... and which, once aroused, will call our attention to them, and thus give them occasion to become conscious.

... When a sudden illumination seizes upon the mind of the mathematician, it usually happens that it does not deceive him, but it also sometimes happens ... that it does not stand the test of verification; well, we almost always notice that this false idea, had it been true, would have gratified our natural feeling for mathematical elegance.

Thus it is this special esthetic sensibility which plays the role of the delicate sieve of which I spoke, and that sufficiently explains why the one lacking it will never be a real creator.[24]

Wilson says of Kepler's four-year, 900 page calculation that led to his planetary theory: "Kepler's discoveries were a kind of miracle of chance and love, and also of inventive hypothesizing and detective logic."[25] Nothing short of his love for the quest and his "belief in the possibility of understanding" could have sustained such a prodigious effort.

Section D -- Galileo and The Accidental Nature of Discovery

> Elias Howe had been frustrated for many years by his failure to perfect
> the sewing machine. One night he dreamed he had been captured by
> savages and dragged before their king. The king issued a royal
> ultimatum. If Howe did not produce a machine that would sew within
> 24 hours, he would die by the spear. Howe failed to meet the deadline
> and saw the savages approaching. The spears slowly rose and then
> started to descend. Howe forgot his fear as he noticed that the spears
> all had eye-shaped holes in their tips....
>
> -- Krippner and Hughes[26]

How does creation happen? What leads to a new idea or discovery? One might think it is an eminently rational process: after a body of representative facts is assembled, a careful process of logical deduction leads to a new invention. While this may have happened, it is by no means necessary. In fact, Taylor states that "the rules of logic and scientific method are a psychological straight jacket for creative thought."[27] Often creativity is emotional, imprecise, illogical and just plain wrong. It may even involve chance. Drake[1973] has presented evidence that Galileo obtained his first formulation of the law of accelerated motion by accident, through an error. Galileo's creative step was enormous, but what was its nature? Like Kepler, Galileo applied a *predefined schema* -- the Euclidean theory of proportion -- to his (hypothetical) data. This led directly to his synthesis of the law of motion. But Drake points out there was an *error* in Galileo's hypothetical data which contributed directly to the result. In fact, there appear to be *two* accidents in his derivation; the second was his use of 4 and 9 for his hypothetical distances, both square numbers. These accidents appear to have been standard fare for Galileo. Another schema -- "nature always acts in the simplest way" -- he used *all his life.* Though we might be tempted to regard Galileo less highly because of his "lucky" mistakes, as James Joyce has said, "A man of genius makes no mistakes; his errors are portals of discovery."[28] And Hadamard[1945] points out that while there is a distinction between invention and discovery, both involve creativity. Some people have even carried the process of making errors to the extreme of making it their working method. Mednick tells of a physicist who places "in a fishbowl large numbers of slips of paper, each inscribed with a physical fact. He regularly devotes some time to randomly drawing pairs of these facts from the fishbowl, looking for new and useful combinations."[29] Many creative ideas have even been the result of dreams, as the story of Elias Howe illustrates.

Ernst Mach, a 19th century thinker about thinking, has a memorable introspective description of how mental "luck" leads to discovery:

> After the repeated survey of a field has afforded opportunity for the
> interposition of advantageous accidents, has rendered all the traits that
> suit with the word or the dominant thought more vivid, and has
> gradually relegated to the background all things that are inappropriate,
> making their future appearance impossible; then from the teeming,
> swelling host of fancies which a free and high-flown imagination calls
> forth, suddenly that particular form arises to the light which harmonizes
> perfectly with the ruling idea, mood, or design. Then it is that that

which has resulted slowly as the result of a gradual selection, appears as
if it were the outcome of a deliberate act of creation. Thus are to be
explained the statements of Newton, Mozart, Richard Wagner, and
others, when they say that thoughts, melodies, and harmonies had
poured in upon them, and that they had simply retained the right
ones.[30]

"Harmonizes with the mood..." There is a critical principle hidden in these emotional terms.
Unless a person really *likes* a task, has an *affection* for it, he is seldom if ever creative in it.

What is the mechanism of creativity? Sarnoff Mednick has clearly outlined one
answer. In general, "any ability or tendency which serves to bring otherwise mutually
remote ideas into contiguity will facilitate a creative solution; any ability or tendency which
serves to keep remote ideas from contiguous evocation will inhibit the creative solution."[31]
In particular, there are three ways to arrive at a creative solution:

(1) *Serendipity* -- two stimuli may occur together by accident or "luck" and evoke an
associated response. This is what the physicist above was trying. It is irreverently
recognized in the saying, "Give enough monkeys enough typewriters and they will
eventually produce the complete works of Shakespeare." The stories of Newton and the
apple and of Fleming and penicillin are other popular anecdotes about serendipity.

(2) *Similarity* -- two stimuli may evoke a response due to the similarity of the
elements evoked by the stimuli. An example is the use in literature of homonyms and
rhyme. Mednick feels this is important in areas which are not dependent on the
manipulation of symbols (e.g. painting, sculpture, music, poetry).

(3) *Mediation* -- two stimuli may evoke a response because they have some element(s)
in common. "For example, in psychology, the idea of relating reactive inhibition and
cortical satiation may have been mediated by the common associates 'tiredness' or
'fatigue'."[32] Mednick feels this is particularly important for creativity in areas involving
heavy use of symbols (e.g. mathematics, chemistry).

'Mednick distinguishes at least two distinct cognitive properties or dimensions that
govern an individual's creativity: the "associative strength" dimension and the "visualizer-
verbalizer" dimension.

The *"associative strength"* dimension is the number of associations that an
individual has between ideas and the strength between these associations. For example, we
might ask a person what concepts "table" suggest. If he is limited to just a few stereotyped
responses or if he has difficulty generating many responses, he is said to have an associative
hierarchy with a *steep* slope.

Number of Responses
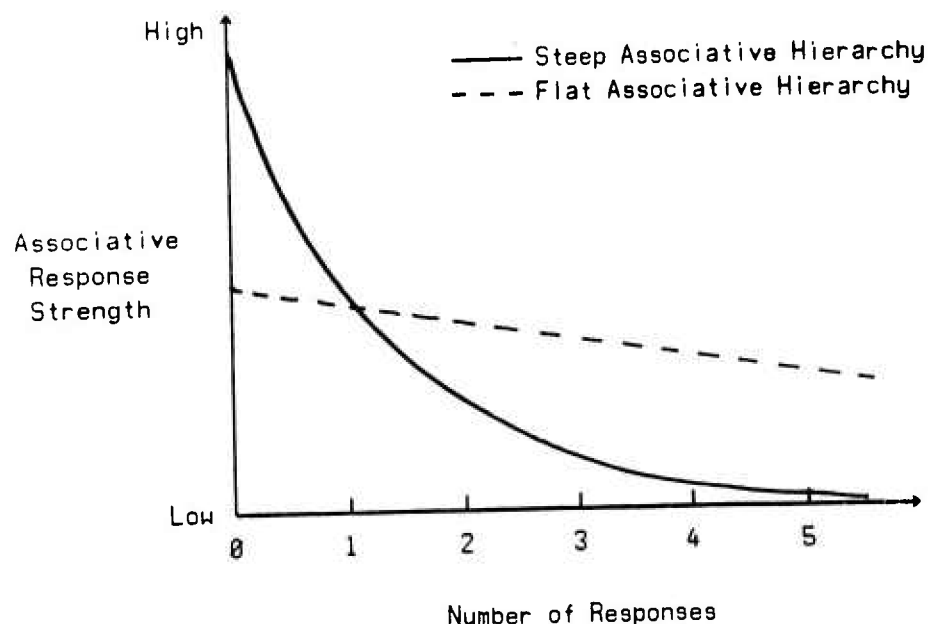
If the person is easily able to generate more remote associations, he is said to have a *flat* slope to his associative hierarchy. This is important because a person obviously is not going to be able to use elements in a creative way if he doesn't have the elements or if it is very difficult for him to generate them. Mednick concludes:

> It would be predicted that the greater the concentration of associative strength in a small number of stereotyped associative responses (steep hierarchy) the less probable it is that the individual will attain the creative solution. Thus, the word association behavior of the high creative individual should be characterized by less stereotypy and commonality.
>
> ... The prediction suggesting an expectation of less creativity from an individual with a high concentration of associative strength in a few responses leads to another prediction. The greater the number of instances in which an individual has solved problems with given materials in a certain manner, the less is the likelihood of his attaining a creative solution using these materials.[33]

There is a rather frightening implication for computer science: The more "skilled" or experienced a person is in using computers, the less likely it is that he will be able to use the computer in a creative way! This induces the argument in Part II for a widening of the concept of "computer scientist" from a person skilled in the use of computers to anyone who knows how to do something and wants to use the computer as a tool in doing it. Such a widening requires not only an increased access to computers for the common man (Kay's "personal dynamic medium", for example), but it also requires an improved method for communicating with computers since it is presently too technical.

The "*visualizer-verbalizer*" dimension is a restatement of the notions of analogical and Fregean. The visualizer is one who tends to call up relatively complete memorial sensory representations of the relevant concrete aspects of problems. If the problem deals with horses, he tends to picture a horse in terms of its sensory qualities. On the other hand, the verbalizer explores the problem by associating with words around the word "horse". If the requisite elements are high in his verbal associative hierarchy to the word horse, the verbalizer will be more likely to attain a creative solution; the visualizer may be thrown off or at least delayed by many false leads. On the other hand, if a requisite verbal associative response to the word horse is very low, or not present in the verbalizer's hierarchy, then the visualizer will be more likely to attain the creative solution. It is therefore clear that some types of problems will be solved more easily by the visualizer and some by the verbalizer.[Mednick[34]] Conventional programming languages almost uniformly require a person to verbalize his solution if he wants to implement it on a computer. PYGMALION is an attempt to get further to the visualization end of the scale.

Chapter 3

The Relevance of Computers

[Computers'] especial talent in the direction of intelligence is the ability to make elaborate models and fiddle with them, to answer in detail questions that begin "What if ...?" In this they parallel (and can help) the acquiring of intelligence by children.... The human mind ... can tolerate and even thrive on inconsistency.

-- Stewart Brand[1]

Summary

(1) The computer has characteristics that can be used to assist a person's thinking and learning processes.

(2) Some of these characteristics are being exploited successfully today.

Programming is often tedious work. It simply takes too much time, effort and irrelevant detail to implement an idea on a computer. The programmer must know the capabilities of his programming language, the efficiencies of its compiler, and even the organization of the machine. There are syntax conventions, semantic conventions and conceptual conventions to which he must faithfully adhere. Computers are intolerant of even the tiniest mistake; a single wrong bit may kill a program. As Stewart Brand irreverently says, "The basic fact of computer use is 'Garbage In, Garbage Out' -- if you feed the computer nonsense, it will dutifully convert your mistake into insanity-cubed and feed it back to you."[2] Few laymen feel competent to undertake large programs. Even among computer scientists, there is a notable reluctance to do the actual coding. They prefer to hire people to implement their ideas. The process of formulating a solution is interesting, but the implementation is not.

This thesis addresses the questions:

(1) Why is programming a tedious process? Is it *necessarily* tedious?

(2) What are the relationships between creating a solution to a problem and creating a problem to find a solution?

(3) Do programming languages stimulate or inhibit creative solutions?

(4) Does creativity in art and mathematics provide any guidelines for creative activity on a computer?

(5) Can a programming environment be constructed to stimulate creative thought? What would be its characteristics?

Programming need not be tedious. The rest of this paper is devoted to computer systems which make programming *fun*. As we have seen, creativity is an emotional process, and joy is one of the strongest emotions. There is playfulness in creativity. Given the groundwork of the first two chapters, we can begin to answer questions (1) through (4). Question (4) in particular now warrants a resounding "yes!" This chapter bridges the gap between the abstract model of thought of Part I and the concrete computer environment of Part II. In so doing, we attempt to tie together the threads represented by the first four questions.

The main result of this paper is that the answer to the first part of question (5) is "yes". The answer to the second part of (5) forms Part II. In fact there are already several systems existing today which successfully aid some aspects of creative problem solving. This chapter contains a brief discussion of them and of the problems in interfacing a computer with a creative human being.

Section A -- Dynamic Programming

The Unimate Robot

The Unimate robot consists of a mechanical arm with 6 degrees of freedom mounted above a large base containing electronics. It is a programmable manipulator designed for industrial applications.

Rotary motion

Vertical motion

Radial motion

Wrist bend

Wrist yaw

Wrist swivel
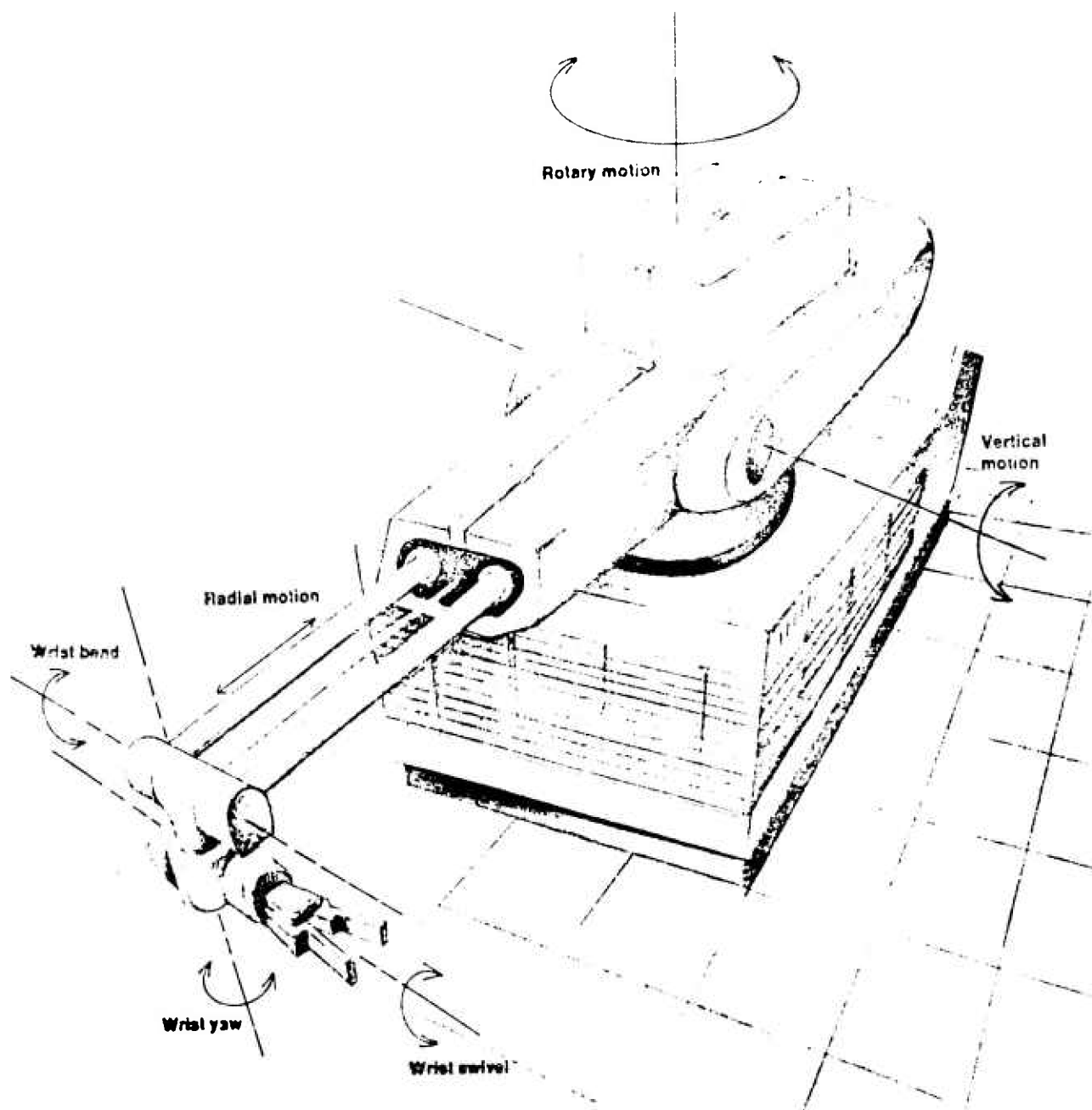
The Unimate Robot

3-A   Dynamic Programming

The robot may be operated in either of two modes: training mode or production mode. In training mode, the robot's arm is *guided through the steps necessary to perform a task* by a human "trainer". The robot has a digital electronic memory in which it can "remember" up to 1024 operations and their timing. Typical operations are "rotate a joint", "move to (x,y)", "close the hand", etc. After the training phase the robot can operate in production mode, automatically repeating the operations in its memory. It will repeat the operations indefinitely, until stopped or until a pathological condition occurs.

The robot has been particularly successful on assembly lines. General Motors is presently using 26 Unimate robots to do 80% of the final welding on its Vega assembly line. The primary practical deficiencies of the robot are the absence of (a) conditional branching and (b) force or visual feedback. From a conceptual standpoint, the robot is a relatively uninteresting computer since its only data structures are (X,Y) coordinates.

However the Unimate demonstrates the potential of dynamic, analogical programming. Rather than writing an algorithm whose form bears no resemblance to the task to be done, programming occurs by actually *doing* the task. Advantages:

(1) The *act* of programming is analogous to the *function* of the program being written. It is *learning by doing*. Programming is exceptionally clear and easy.

(2) An untrained operator can program the robot, "untrained" in the sense that he need have little knowledge of computer programming -- he need only be familiar with the task the robot is to perform. This makes the robot accessible to a large class of users.

(3) Bug-free programs can be written the first time. Since programming involves doing the task once, *successful completion* of the task means that a correct program has been written (modulo mechanical and/or timing inaccuracies).

If there is a programming language which can make these claims, it should come forward and be honored. (Radia Perlman's "TORTIS", described in section E, is an excellent first attempt.)

"TV" Editor

The Stanford Artificial Intelligence Laboratory has installed a network of about 50 raster-scan video displays driven by a Data Disc video disk. Several excellent programs have been written at the lab to exploit the displays' potential for visual interaction. A text editor called "TV" has been developed by Dan Swinehart[1971] (since written as "ETV" by Wright and Samuel). TV was inspired by other on-line display editors, particularly Engelbart's "NLS" system for "augmenting human intellect".[1962,1968] TV is organized around logical "pages" and physical "windows" of text. A "window" is the text -- 33 lines -- that can be physically shown on the display screen at one time. TV always shows a screen full of text *as it currently exists*; editing changes are made in place on the display screen as

well as in the document. This makes it extremely easy to create and modify text, and to comprehend the current state of the document.

TV has several dynamic operations that have the same flavor as the Unimate operations. Replacing characters in a line of text is accomplished by positioning a cursor beneath the first character to be replaced and then simply typing the new text. The updated state of the line is always displayed. It is easy to see if the change has been made correctly by just looking at the line. But the most analogical feature is "attach mode", a method of designating text (first implemented in NLS). A group of consecutive lines can be "attached" and manipulated as a unit. The attached lines can be moved to another part of the document or even to a different document. Searches and substitutions can be limited to just the attached lines. This gives the user excellent control over the scope of operations and over the structuring of text. Since he can always see the context above and below the attached lines, the user has virtually no difficulty positioning a body of text exactly where he wants it. Editing with "TV" is far more error-free than with batch oriented or non-display oriented editors. As with the Unimate, the *appearance* of each operation is analogous to its *effect*.

HP-65 Pocket Calculator

A recent addition to the ranks of analogical programming systems is the Hewlett-Packard HP-65 pocket calculator. The HP-65 differs from other hand-held calculators in that it can accept magnetic strips containing up to 100-step programs.

The HP-65 Pocket Calculator

The relevant aspect is the way in which programs are written. The calculator is simply put in "program" mode, and then the desired sequence of keys are pushed, *just as if one were doing a calculation.* The keys are remembered on a magnetic strip. In "run" mode the program can be executed, or the calculator can be operated manually. However a major defect in the design is that while in "program" mode the display does not show the current state of the calculation. Rather it shows a numerical representation for the last key pushed. So typically programs have to be worked out in "run" mode, written down, and then entered in "program" mode. Going straight to "program" mode is too abstract. While programmable desk-top calculators have been available for a number of years, the size and portability of the HP-65 give it many of the characteristics of Kay's "personal dynamic media"[1975]. The calculator comes to be viewed as an extension of the self.

"RAID"

If one *must* program in machine language, the best debugger for it that I have seen is "RAID"[1970], developed at the Stanford AI Lab for the PDP-10. RAID is the culmination of a line of debuggers going back to the TX-2 at Lincoln Labs. It represents the state of the art in debuggers. Using the excellent display facilities at Stanford, RAID is capable of

(a) dynamically displaying the contents of memory locations, including locations which are referenced "indirectly" through other locations;

(b) stepping through an instruction sequence, during which the user can observe changes to his displayed locations;

(c) initiating execution at any instruction;

(d) replacing instructions with other instructions and then re-executing the sequence with the new code;

(e) replacing the contents of displayed memory cells with values entered from the keyboard; and

(f) other more standard debugging behavior.

The result is that the user can *directly observe the effects of instructions on actual information.* It is concrete. The consequences of a sequence of even such primitive operations as machine instructions are frequently difficult to comprehend in the abstract. With RAID, it usually requires no more than one pass through a routine, replacing incorrect instructions as you go, to completely debug it. Occasionally users even write routines in debug mode. Interaction, concreteness, and a visual display are the keys to RAID's success. The chief deficiency of RAID, aside from the fact that one must deal with machine language to use it, is that it displays only machine words. It does not display the structure of data.

High-level Debuggers

It is more difficult to design a good debugger for higher-level languages. Since most such languages are compiled before they are executed, less information is available at run time than at compile time -- information such as the symbol table, the source code corresponding to an instruction sequence, the logical structure of data types, etc. Kay[1969] addressed the problem of writing, displaying, monitoring and debugging high-level programs through multiple processes. His "FLEX" machine divided its display screen into multiple "windows" and "viewports". Each window in Kay's system shows a 1024 x 1024 section of a virtual 16384 x 16384 display. Viewports are subdivisions of windows. Windows and viewports are the communication links to processes. Swinehart[1974] implemented these and other ideas using the Stanford display facilities. Each of his windows can show the same computation in a different representation. One window might be a RAID window showing the machine instructions being executed. Another might show the source text that corresponds to those instructions. Still others might show the state of variables used in this part of the program, an image of the stack, program counters, or debugger options. Unimplemented but possible are dynamic, graphic displays of changing data structures. An unusual notion is that each window-process can control the execution of other processes. For example, one can single-step a source language statement in a text window, which causes the machine language window to execute a sequence of instructions and then pause. Or we can change variable values in a variable window, which will modify subsequent execution in other windows. This powerful concept of "floating control" has also been successfully used in time-sharing systems such as the SDS-940 system [Lampson[1968,1969]] and simulation languages such as SIMULA [Dahl[1966]] and SMALLTALK [Kay[1972]] (a FLEX derivative).

Interpreters and Compilers

Some of the difficulty of telling a user what is going on in compiled programs has been resolved simply by not compiling them. Some languages, notably LISP, APL and SMALLTALK, are based on interpreters. Binding of tokens to semantics is delayed until values are actually required. For example, the identifier X might be bound to a variable value or a function name; in either case evaluating it will cause it to return a value. In the first case a simple fetch is done; in the second, an arbitrary amount of work may be performed before the value is obtained. The languages are not analogical, as with the Unimate, but the *environment* in which they execute is dynamic and concrete. "X" may be ambiguous at the time it is written, but at run time it will be bound to something which knows how to produce a value. Using interpreters, the programmer deals with concepts like "value producer" and "generic operation", instead of "cell which can contain a real number" or "integer addition". Interpreters permit the kind of "selective abstraction" that we noticed in creative thinking (Chapters 1 and 2). Entities such as "X" can exist on a variety of conceptual levels: as a (Fregean) representation for information, as a cell which can contain information, or as a routine which can generate information. This ability to defer instantiation until necessary, so characteristic of the mind, usually makes it easier to program in interpreted languages than in compiled languages.

Mitchell[1970] merged the concepts of interpreter and compiler. He notes that

compilers and interpreters do essentially the same work. Each must analyze program text for syntactic and semantic correctness. Interpreters then execute code based on the analysis, while compilers emit code for later execution. Mitchell suggested that a way to delay instantiation in a compiler is to execute programs interpretively the first time. The compiler can then *emit the same code that the interpreter executes!* The execution of the interpreter provides a "semantic cache"[3] which the compiler can use to emit code with little extra effort. The successful interpretation of a program (a) indicates that it is at least syntactically correct; (b) gives a certain confidence that it is semantically correct as well; and (c) gives a high degree of confidence in the correctness of the compiled code, since it is identical with the interpreter's code. It is even possible to declare the types of variables at "interpreter time", by indicating to the compiler that the data type bound to a variable the first time will be the type in all successive bindings. This "dynamic declaration" is a promising concept.

The logical extrapolation of interpreters in a computer environment is the concept of *hardware interpreters*. The Burroughs Corporation has developed interpreting machines in its B-5000 (ca. 1961) and B-6000 series. These machines provide instructions to load descriptors on top of an internal stack. The hardware interprets each descriptor and executes the necessary operations to produce a value, whether it is by a simple literal call, an array access (which will pick up indices from the stack and check them against bounds), a procedure call (which will set up a procedure frame and transfer control), or some other type of access. The compiler can invoke any of these cases by emitting a single type of instruction: "fetch descriptor". Needless to say, this substantially simplifies writing compilers for Burroughs machines. Today, with the availability of fast, inexpensive, microcodable computers, hardware interpreters are becoming an increasingly viable technique.

Programming by Example

Sussman[1973] used some of these ideas in an automatic programming context. His system "HACKER" has the ability to extend general cases to handle unforeseen exceptions. Using HACKER the programmer writes an algorithm for the situations which he knows will occur, or which he explicitly wants to handle. If during execution the algorithm encounters an exceptional situation, the system can (sometimes) automatically modify the algorithm to handle the new case. There is a fixed set of strategies for modifying algorithms, which Sussman has observed being used by "hackers" at MIT (programmers who enjoy squeezing the last ounce of performance out of a machine). This is a variation on the "learning by doing" concept. The user programs a general strategy, and the final algorithm evolves in the course of trying various examples. It has similarities with the Unimate/HP-65 systems discussed above, in that programs are written by *doing* specific tasks.

Winston[1970] addressed the related task of learning structural descriptions from examples. He investigated the possibilities (and pitfalls) of generalization. For instance, his system was able to develop the concept of "arch" from examples of specific arches.

Section B -- Graphical Descriptions of Algorithms

"Sketchpad"

There have been and are numerous graphical application programs (cf. Wells[1972]). These are programs written in a linear language, e.g. FORTRAN or machine language, that are designed to incorporate graphical interaction in specific application domains, such as architectural or engineering drawing systems. They are special purpose programs. However their operations are often cleverly chosen and provide significant assistance to workers in their domains.

The first and still among the most elegant graphical application program is Ivan Sutherland's "Sketchpad"[1963]. Sutherland's goal was to use the computer to help people visualize things. He pioneered methods of drawing on display screens. While Sketchpad is basically a drawing system, it does not just mimic paper. Instead it takes advantage of the computer's special abilities in such features as multiple "windows" and scaling. The user draws on the display screen with a light pen. Sketchpad was the first to employ the "rubber band" line (used in PYGMALION) in which one end of a line remains fixed on the display while the other end follows the pen. The user may create and modify pictures and their parts. Pictures may be constructed out of "instances" of other pictures.

Instance   Derived Picture
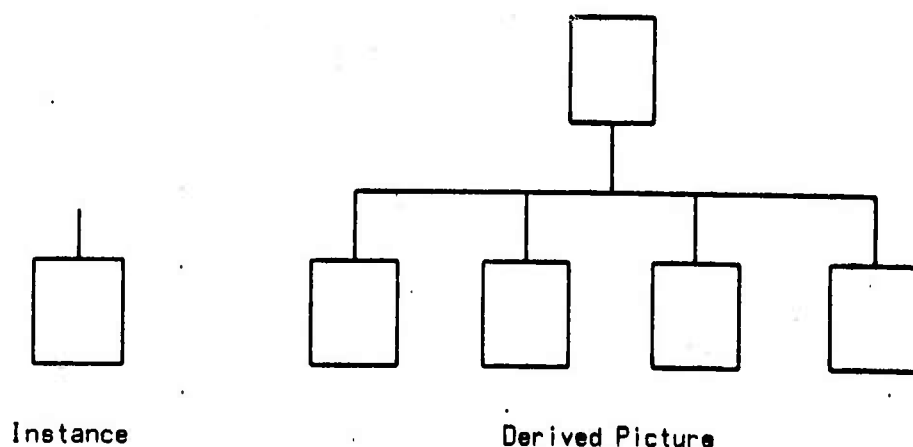
Sketchpad's vocabulary deals with properties of geometric drawings. Sutherland developed a system for specifying pictorial "constraints", such as two lines are to remain: (a) parallel to each other, (b) at a fixed angle to each other, or (c) connected at one end. The constraints are specifiable pictorially, in an abstract diagram which can itself be given constraints and manipulated iconically. Sketchpad was used in several applications, including an engineering application to compute the stresses on a bridge and an animation application in which placing constraints on a diagram caused it to move.

Sketchpad deals with images as *pictures*, to use Arnheim's terminology. The diagrams are manipulated as diagrams, not as representations of something else. "Sketchpad's internal data structure and programs are so rigid that it is inconvenient to make a geometrical entity have non-geometric meaning."[W.R.Sutherland[4]] As with most

graphical application programs, Sketchpad is not a general-purpose programming language. But it is an articulate medium for manipulating pictures, and the first to make effective use of a light pen.

Graphical Procedures

Ivan's brother, W.R.Sutherland, subsequently developed a Sketchpad-like system giving the geometric diagrams symbolic capabilities. His system is based on an electronic circuit metaphor. His program elements are arithmetic, logical and control units with input/output leads that operate like the "level" (versus "pulse") outputs of flip-flops.
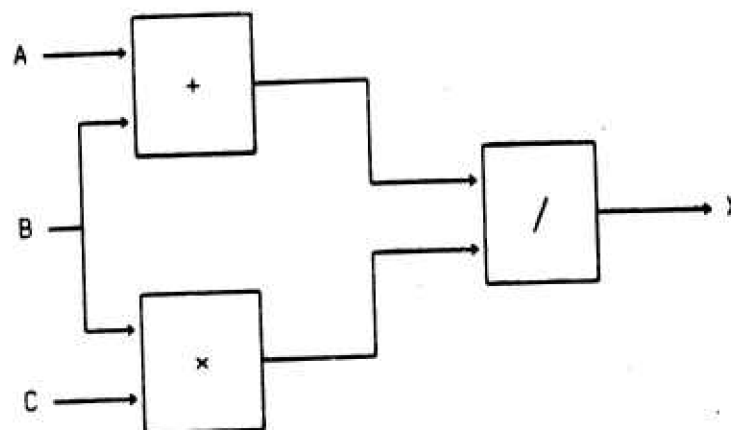
$$X = (A + B) / (C * D)$$

Using a light pen, the user can select elements from a menu, connect them, assign values, and watch the network execute, all in real time. The language is very good for expressing parallel control, which is, of course, standard operating procedure in circuit diagrams. For example, a value placed on the lead labelled "B" above is shunted to the "+" and "x" boxes in parallel. A menu of circuit elements provides functions for type-in, type-out, "seeing" values on certain leads, arithmetic, logic, conditional branching, and some diode-like elements for controlling data flow. A problem with the system, the same problem experienced by AMBIT (see below), is that the circuit diagrams are often so complex as to be difficult to comprehend. Consider the following program, which computes the square root function.

$$X = SQRT(A)$$

The angled boxes are used to indicate more clearly the direction of data flow. Races in networks like this are prevented by the "__ notation above input leads. This specifies that after use, the input is to go to *undefined*. Thereafter, the operation will not execute until a new non-*undefined* value arrives.

W.R.Sutherland recognizes the potential of this medium of expression when he writes: "The two-dimensional nature of the language helps in visualizing many things happening at once.... Being able ... to see a program run gives one a grasp of detail that is hard to obtain in any other way."[5]

The GRAIL language [Ellis[1969]] developed at RAND was a similar attempt to describe programs visually. The GRAIL philosophy was that since flowcharts are usually more articulate than the statements they describe, why not program in the flowchart notation to begin with? GRAIL incorporated several graphical elements for representing programming operations, such as conditional branching, functional elements, and loops. The contents of each box, however, consisted of ordinary machine language statements. Just the interaction of functional units was described graphically. GRAIL did include a "zooming" control for dynamically scaling program elements. Complexity could be managed by enlarging boxes to be examined and then shrinking them again to see the overall structure.

"AMBIT"

Another approach to graphical programming is provided by the AMBIT family of languages.[Christensen[6]] Ivan Sutherland's Sketchpad operated on pictorial data but without assigning machine semantics to the pictures. W.R.Sutherland's system assigned semantics to pictures, but the data was non-pictorial (e.g. numbers). AMBIT/G (for graphs) and AMBIT/L (for lists) are attempts to combine pictorial data and procedures. Processing in AMBIT is by two-dimensional pattern matching. A procedure is actually an image of the data to be passed to it. The image simultaneously specifies an input pattern to be matched and a transformation to occur if the match is successful. This dual role is accomplished by having arrows made up of single lines represent links that exist in the input data, and arrows made up of double lines represent new links to be formed in the output data.

An interesting feature is that each procedure-image contains a control link specifying what to do when the pattern match succeeds (S) or fails (F). In the image above, success causes the procedure to be called recursively. Some quite large programs have been written in AMBIT, including several garbage collection algorithms. It is enlightening to see the AMBIT versions of certain algorithms. AMBIT is a step toward an effective metaphorical context for working out problems.
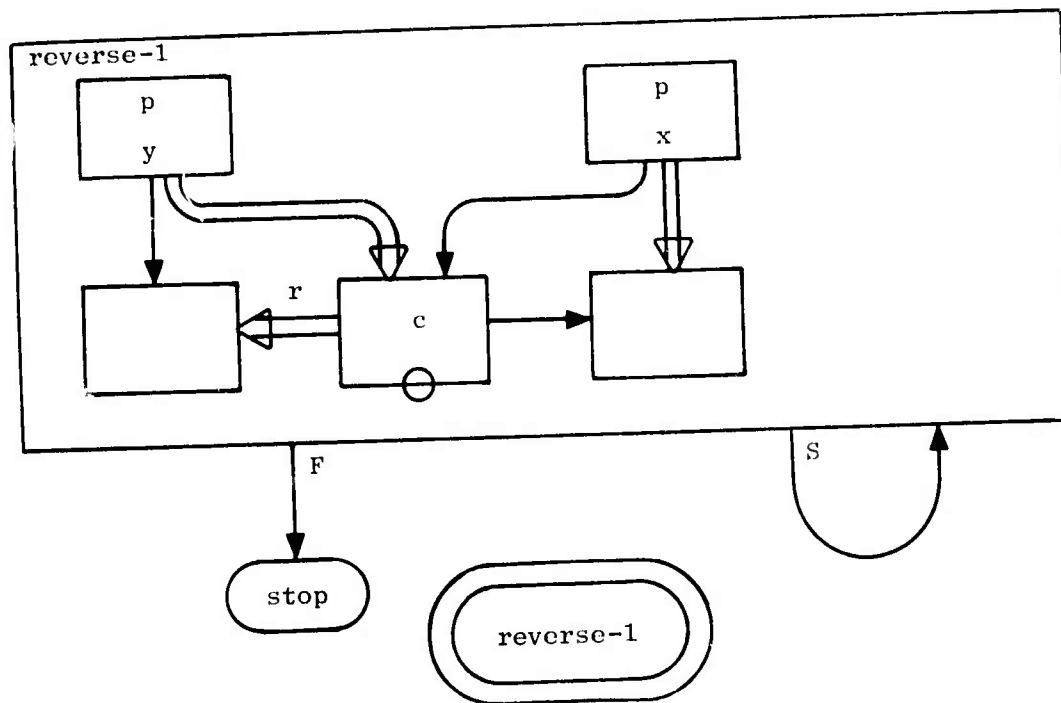
Reversing a List

However AMBIT is a good illustration of the deficiencies inflicting all current graphical programming languages. A practical deficiency is that implementations are not yet efficient enough to make the languages usable on a large scale. But a more basic, theoretical deficiency is their philosophical orientation. Every graphical language suffers from one or more of the following:

(1) They use *static* representations for dynamic processes.

(2) They lack detail suppression mechanisms. Consequently, pictures quickly increase in *complexity* beyond the ability of the eye and short term memory to assimilate. (GRAIL is an exception.)

(3) They operate on *formal* representations of data, one level removed from actual information.

(4) They lack image-defining capabilities. The programmer cannot draw his own images; he must use the images designed by others. This limits the effectiveness of the metaphor.

Anyone who has looked at a large AMBIT program knows how hard it is to understand. The maze of lines soon becomes overwhelming. Even small algorithms sometimes take a lot of parsing by the eye before their operation can be discerned. This is probably due to unfamiliarity with the languages, but articulate languages should not need much explaining!

Much of the complexity can be traced to AMBIT's representation of both the present state and a future state in a single frame. This is an example of *telling* instead of *doing.* Environments for *describing* computations are inherently more abstract and require a greater translation from mental representations than environments for *doing* computations. The program frames become more and more complex and full of notational dogma. They violate one of the cardinal rules of animation:

> What happens *between* each frame of a sequence of pictures is more important than what exists *on* each frame. The actual graphical information at any given instant is relatively slight. The source of information is picture *change*.[Baecker[7]]

It is the business of animators to communicate effectively with human beings. The next example discusses a computer animation system.

Animated Graphical Descriptions
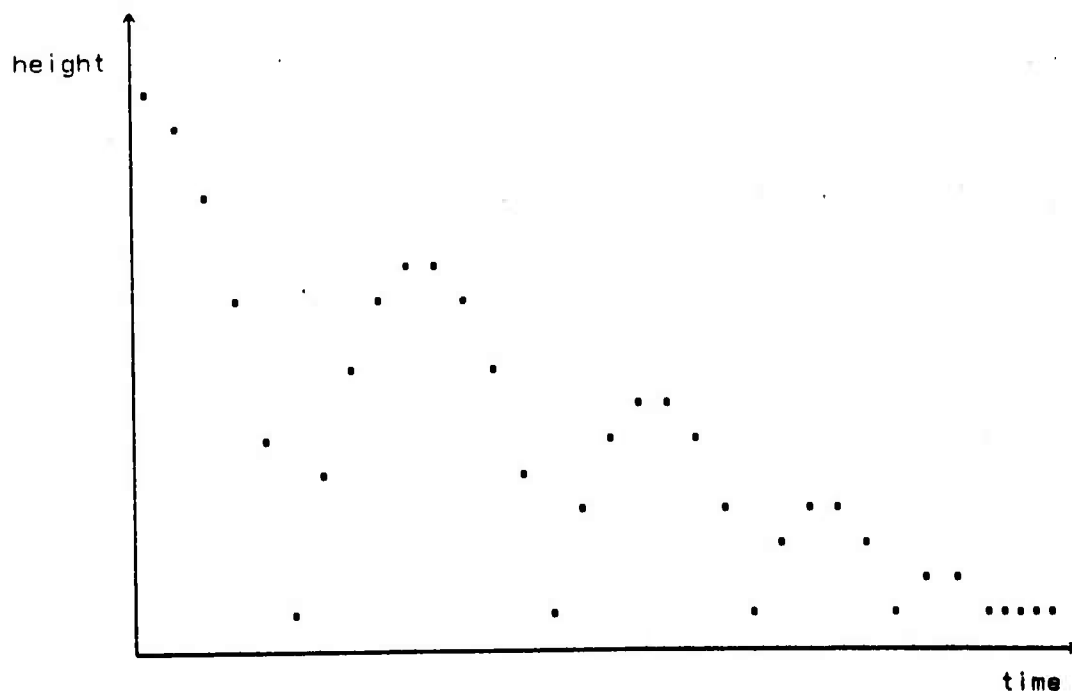
Ron Baecker[1969] in his GENESYS system investigated ways to describe animation sequences using pictorial control. For example, the graph above might be used to drive a movie of a ball bouncing. Recently he and his students at Toronto have made some animations of program semantics, such as (a) recursive functions in LOGO [Papert[1970]] and SMALLTALK [Kay[1972]], dynamically showing recursive calls until a terminating case is reached; (b) the operation of the railway switching algorithm for parsing; (c) the execution of various simulations, in particular the simulation of the movement of people in a subway system. These animations are the most effective presentation of mechanical semantics that I have seen. They begin to instill the level of insight and intuition so necessary for creative thought. The critical aspect of their success is the *dynamic, visual, movie-like presentation of information.* A descendant called SHAZAM has been implemented in Kay's Learning Research Group at Xerox PARC (described in section D). SHAZAM enables the computer to dynamically interact with the animator, permitting him to create and modify movies in real time. These animations are at the core of PYGMALION's model of articulate communication.

Visual media like the ones in this section provide the user with a rich working environment. If a concept can be placed in one of these metaphors, extensive processing can be done on it before the limitations of the medium become too procrustean. With Fregean media such as verbal languages or one-dimensional programming languages, the limitations immediately impose constraints. In Koestler's terms, visual media provide alternative "matrices" or "thought contexts" which are powerful enough to be of use in creative thinking; linear media do not.

Section C -- The Computer as an Artistic Resource

Art enters when we labor thoughtfully with some ideal in view -- that is, as soon as we cut loose from action that is purely mechanical.
                                            -- Clifton Johnson, photographer[8]

A good example of the reasons why the aesthetic and technical elements in design must be considered together is implicit in the term 'fair', used to describe fluid dynamic surfaces in products such as air-frames, aero-engines and ships' hulls. It is required that some surfaces be 'fair' in order that they should work properly, that the air or water should flow over them in the required way. But fairness can only be judged by inspection of the surfaces, by aesthetic judgement. Thus we have in its clearest form the need to keep the data structure and mathematical methods of design, and the visual or aesthetic aspects going hand in hand.

The use of 'fairing' is not a reason for ignoring precise technical methods in so far as they have been developed. One might perhaps say

that the use of fairing represents the use of that part of knowledge which comes from experience and has not yet been precisely formulated. But there will always be knowledge that has not been precisely formulated.

-- Anthony Hyman[9]

PYGMALION brings *art* into *computer science.* Rather than providing a computer resource which artists can use to create (paint, compose music, etc.), PYGMALION is a first attempt to provide an *artistic resource* which *computer scientists* can use to create. In fact, I hope PYGMALION will contribute to a re-evaluation of what a "computer scientist" is. In my view, a computer scientist is anyone who knows how to do something and wants to use the computer in doing it. The view that only highly-trained programmers can implement tasks on computers is intellectual snobbery of the worst kind. Anthony Hyman describes such conventional "computer scientists" as "computer specialists":

> The idea that computing is necessarily difficult and to be reserved for senior grades or forms at school is surely wrong, corresponding to a rather early stage in the development of computers: experimental studies with seven-year-old children [and even younger] have shown that children will take to the use of display terminals with ease if they are given the opportunity young enough. Today computers are still expensive and computing languages by and large still rather clumsy. (It is curious that after working with computers for many years some people have deluded themselves into thinking that FORTRAN, the most commonly used scientific computing language, resembles a natural language.)[10]

Such views are merely historical, an outgrowth of the anachronistic programming languages that were first provided. As Kay says, we should have a more optimistic opinion of people; people *are* smart, and incredibly versatile.

> The skilled programmer is necessary only because the distance between the computer implementation of a task and a person's mental conception of it is too great.

Symbiotic systems like architectural design systems have eliminated the skilled programmer altogether, and substituted the architect, a real "computer scientist". But architectural design systems are special-purpose programs written (by a "skilled programmer") in some general-purpose language for a specific application. What is needed is a symbiotic general-purpose language. As Hyman notes,

> A good case can be made that to achieve satisfactory results the ultimate user must have a say in the development of the design systems themselves ... The designer can no more rely on the computer specialist to develop design systems appropriate to his needs than a wood carver can allow other people to sharpen his tools. The computer specialist does not have the knowledge and cannot acquire it without becoming a designer himself.[11]

Computer science does employ some general principles that are beneficial for every programmer to know, such as *subgoals and subroutines* (subdividing a problem into smaller units), *heuristics and partial algorithms* (it is not necessary to insure that a program will work on all possible inputs or to specify all possible paths), *recursion* (parts may exhibit the same characteristics as the whole), *processes* (the computer may remember state), *tracing* (following the execution of a process step by step), *debugging and editing* (parts of a process may contain errors and be corrected independently of other parts), and others. But *these* are the concepts that should be taught to a person who is learning to use the computer, for as Papert says, these are really *teaching him about thinking*. Too much time is spent teaching "integer variables must begin with the letters I, J, K, L, M and N and real variables with other letters."

What is the potential of using computers artistically? "In Renaissance Florence men such as Brunelleschi and Da Vinci were at once artist and engineer. They were deeply involved in extending both new ideas and new techniques ... [Today the sheer volume of information makes this difficult.] Recently computers, having acquired effective methods of display and considerable visual processing power, are beginning to be used for design in earnest. With this development the possibility is raised of re-establishing the unity of Renaissance Florence." [Hyman[12]]

Section D -- The Computer as a Creative Resource

Artists create using a medium. The *process* of creation is incremental. Painters dab, dab, dab at their canvasses until they find just the right expression of their feelings. Sculptors chip away at the stone until first a knee, then a nose, slowly a whole figure is perfected. Though they begin with an outline, a "grand idea", artists realize the idea in a series of small steps ("subgoals"). There are a few exceptions. Chinese brush painting emphasizes single strokes to define figures; the technique of placing the brush on the paper has been highly developed. Painters using watercolors are prohibited by the nature of the medium from touching up their pictures, but just for this reason watercolors are difficult to use. Photographers seem to be most completely spared the incremental task. (In fact, Henri Cartier-Bresson refuses to allow his pictures to be cropped or modified during printing, declaring "The picture is good or not from the moment it was caught in the camera."[13]) But photographic creativity is ill-defined, and it is certainly more than the mechanics of camera, film and darkroom. "The only microscopic part of the photographic process which can be remotely described as 'creative' is that fractional instant when the photographer feels: yes, now."[14] It may require years of patient experiments before a photographer develops the instinct for "the decisive moment". Though photographers do have a more immediate medium, the process of *learning to use it* is still incremental.

The value of incrementality has not been ignored in computer science. Time-sharing is an attempt to provide incremental computing, among its other goals. Many programming languages are incremental, most prominently LISP, SMALLTALK and APL. Some text editors are incremental; anyone who has tried to use a batch-oriented text editor

will appreciate the value of incremental ones! The advantage of incremental computing is that the consequences of an operation are *immediately* discernible by the user. He can isolate the consequences of individual operations and debug them one at a time. The similarity with artists is inescapable: painters dab on a bit of paint and then step back to look at the result, etc.

A computer will be of the greatest value to creativity if computing done with it is incremental.

O.K.Moore stresses the need to take the *work* out of learning, making it seem in all respects like *play*, if we want to develop creative thinkers. With his "Talking Typewriter" and his notion of an "autotelic environment", Moore concentrated on combining enjoyment with substance. The computer alone is not sufficient to interest children; it must be used in an imaginative way. Moore defines an activity as *autotelic* if "engaging in it is done for its own sake rather than for obtaining rewards or avoiding punishments that have no inherent connection with the activity itself."[15]

The most imaginative work involving computers and children is going on today at MIT under Seymour Papert and at Xerox PARC under Alan Kay. Papert and Kay have abandoned the traditional approach to computer-aided instruction, which they view as using the computer to teach the same old concepts in the same old way. Their goal is to develop new ways to teach concepts and, more importantly, new concepts themselves. Papert has developed a "turtle" controlled by a computer, and a simple command language (LOGO) for manipulating it.


Children Using a Turtle

them that "complete happiness" in creation of which Lucien Freud wrote. For the children, through the computer, *make the turtle draw* the geometric figures. They create, as it were, brief life histories for the turtle. Not since Pygmalion's statue Galatea stepped off her pedestal has a man-made object been brought to life to a comparable degree.

Kay has expanded on Papert's work. He and others at PARC have developed a small, powerful, stand-alone computer with an excellent graphics display and audio output. He relies on the display and a clean simulation language called SMALLTALK to teach children how to think. The computer serves as a laboratory for experimenting with representations. Kay's eventual goal is to develop an inexpensive, portable "personal computer" the size of a notebook. People will use it instead of paper. It will have an interchangeable tape cassette which can hold books, newspapers, letters and memos. It will have the ability to use multiple type fonts, so that the owner can dynamically display text in his favorite font (dynamic publishing). It will have a plug for connecting with remote sources of information, such as libraries, newspaper offices, stores and banks. With its high-resolution display screen, it can even be used to watch television and video tapes. Of course, in addition the owner will be able to write programs with it. Its flexible, personal nature makes the computer a "user-moldable" medium. PYGMALION was begun to provide a two-dimensional way to communicate with this computer.

A true genius is creative and stimulates the creativity in others. He causes an atmosphere so charged with excitement that good things are bound to happen. The computer also, with the correct interface, can cause such an atmosphere. In Koestler's terms, it can serve as the representational "matrix" whose intersection with the mind's "matrix" provides the "creative spark". At Xerox PARC this small computer is being used to stimulate people's enthusiasm, inventiveness and artistry.

Section E -- Radia Perlman and Her Magical Button Box

Radia Perlman, a visitor at PARC from Papert's group at MIT, has developed a unique instrument for teaching very young children (ages 3 to 6) about programming. The instrument is a "button box" and is used as an input device for a Papert turtle. The button box is connected to a small computer. As the picture below shows, some of the buttons have an iconic description of their action on top. For example, pushing the button marked with a vertical arrow (which we will call "UP") causes the turtle to move forward one unit (about half an inch). Other buttons make the turtle back up, turn to the left or right, honk its horn, put its pen up or down, turn its light on or off, and stop. Some buttons have numbers on them and can be used as repeat factors: pushing 5 UP makes the turtle go five units forward. Since the turtle responds immediately, the children have little trouble understanding what the buttons do. With these few commands, children quickly learn to draw pictures and geometric shapes. Radia calls her button language "TORTIS".
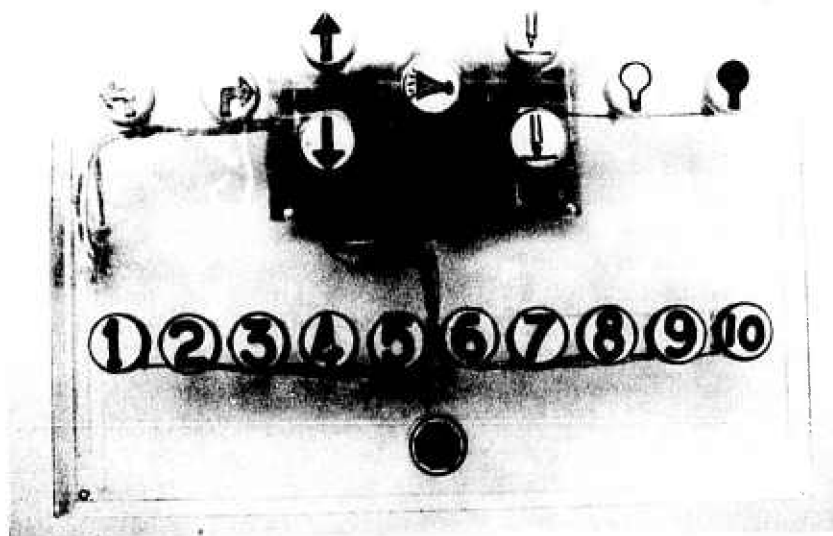
The Basic Button Box

Now the plot thickens. In addition to this basic button box, Radia has developed another box which can be plugged into it: a "procedure" box. The procedure box, shown below, has four buttons labelled with iconic descriptions of the actions "start remembering", "stop remembering", "do it", and "forget it". (I leave it for the reader to figure out which button represents which action.) Pushing the "start remembering" button causes the computer to keep track of the buttons that are pushed from then on. While the computer is remembering the buttons, the turtle is also doing them, so the child can always see what he has done. Pushing **DOIT** causes the computer to execute its remembered sequence. **DOIT** may be preceded by a repeat factor: 5 **DOIT** causes the sequence of remembered buttons to be executed five times. In addition to these four buttons, the procedure box also has four other buttons labeled with a color (red, blue, green and orange). These are "naming" buttons and can be used to give a sequence of remembered button-pushes a name. "Start remembering" **RED** starts remembering button pushes and names the sequence **RED**. **DOIT RED** executes the red sequence. 5 **DOIT RED** executes the red sequence five times. The **DOIT** button may itself be remembered, so that a command to execute the blue sequence (i.e. a subroutine) can be included in the red sequence. In fact the red sequence can **DOIT RED** (i.e. execute itself) so that recursion is possible. However there is no way to terminate recursion once started, since there is no "conditional" button.
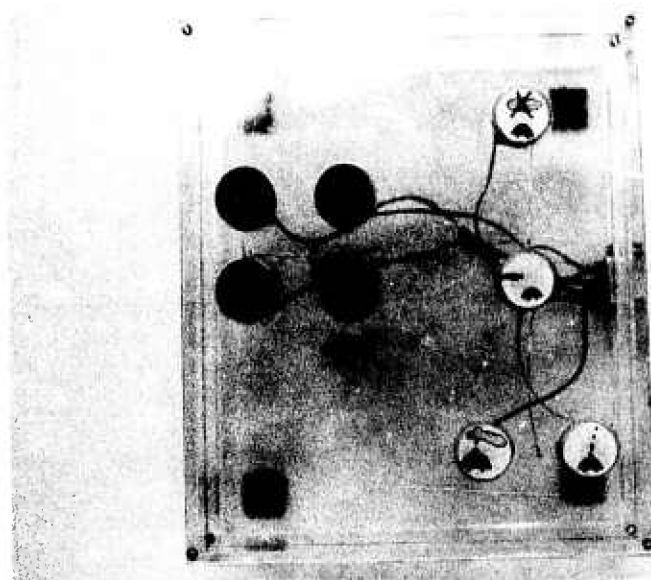
The Procedure Box

Radia has also constructed two simpler button boxes to introduce children gradually to these concepts. The simplest box is the basic box without numbers, i.e. without the concept "repeat". The other box is the procedure box without the colored "naming" buttons.

What can we learn from "Radia's Magical Button Boxes"? Surprisingly, these simple boxes outperform the most sophisticated programming languages in several aspects!

(1) *Children as young as four or five years old can learn to program with these boxes.* These children have great difficulty learning to program with "adult" languages. And this is not entirely because the concepts are simpler, since the notions of subroutine, recursion, iteration and (turtle) state are all incorporated in the system. Children can use the button boxes because the action associated with each button happens *immediately* upon pushing it, and that action is *visually concrete*. Each button has on its top an easily understood iconic description of its action. It is very easy to "see what is going on" and to establish cause/effect relationships. In other words, it is easy for the children to *form a model of the semantics* of each button and of the system as a whole. This model-building is the key to understanding. What children can understand, they can use.

(2) *Children frequently write bug-free programs the first time.* How many programming languages can claim this, even with adult programmers? The reason for this is that programming with the buttons is *incremental* with *continual visual feedback*. To see

how this can lead to bug-free programs, suppose the child wants to write a program to make the turtle draw a square. The simplest way is to push "start remembering" and then to push the "forward" and "turn" buttons as many times as necessary to produce the desired square. Finally, pushing "stop remembering" finishes off the routine. Score: 1 pass, 1 procedure, 0 bugs. True, these programs are *very* simple. But PYGMALION shows that these same error-free characteristics can be built into a general-purpose programming language.

PART II

PYGMALION

Chapter 4

Principles of Iconic Programming

Graphic communication draws upon the natural resources of its own language, and refers to visual experience as a source of principles and values for designing more articulate form ... [It] is a conceptual logic rather than a technical method; a way of seeing the graphic figure as a visual statement.

-- William Bowman[1]

Summary

The main innovations of PYGMALION are:

(1) a dynamic representation for programs -- an emphasis on *doing* rather than *telling*;

(2) an iconic representation for parameters and data structures requiring less translation from mental representations;

(3) a "remembering" editor for icons;

(4) descriptions in terms of the *concrete*, which PYGMALION turns into the *abstract*.

Part I discussed a model of creative thought, emphasizing visual thinking. The model serves as the basis for the design principles in Part II. Part I may be summarized as follows.

(1) Visual thought processes deal with images that are structurally similar to the features of the concepts being represented. Images are a powerful, flexible and effective metaphor for thought.

(2) Creativity involves the conjunction of two normally-distinct thought contexts.

(3) Communication is concrete. Abstraction (i.e. understanding) occurs in the mind from concrete information.

(4) Creativity and understanding are incremental; large discoveries usually derive from a bisociation of smaller ones.

This chapter utilizes the model in a computer environment. It presents the general form and goals of PYGMALION. Most of the specific implementation details are deferred until the next chapter.

Section A -- Introduction

The main goal of PYGMALION is to *develop a system whose representational and processing facilities correspond to and assist the mental processes that occur during creative thought*. It attempts to make Pygmalions out of people, to provide the average person with a medium for creativity without requiring a substantial recasting of his ideas into terms different from his normal way of thinking. The medium is an environment for writing computer programs. Non-numeric programming primarily involves the manipulation and transformation of structure. In PYGMALION sufficient flexibility is incorporated to permit the programmer to design structures patterned after images in his mind. Part I provides two concrete guidelines for the implementation:

(1) Multi-dimensional representations are superior to one-dimensional for communicating some types of information to a human being. Since the intent is to provide as articulate an interface as possible, the system is founded on visual communication using a graphics display.

(2) Interactive feedback is essential in a creative environment. Interactive text editors are a case in point: they are far easier to use than batch-oriented editors. A text editor is similar to a programming language operating on the restricted domain of text strings. If we generalize the domain to arbitrary data structures, the similarity becomes more apparent. If the editor remembers the editing commands and re-executes them on demand, the two concepts become virtually identical.

The heart of PYGMALION is an interactive, "remembering" editor for iconic data structures exhibited graphically on a display screen. PYGMALION is a visual metaphor for computing. Instead of symbols and abstract concepts, the programmer uses concrete display images, called "*icons*". The system maps the visual characteristics of icons into corresponding machine semantics. The display screen is equivalent to a document to be edited. Programming involves creating a sequence of display frames, the last of which

contains the desired information. Display frames are modified by editing operations. When in "remember mode", the system records the operations as they are done.

PYGMALION is a direct descendant of Kay's FLEX machine[1969]. Many of the features of the implementation are derived from FLEX and from conversations with Kay. Perhaps the simplest description of PYGMALION is to say it is just Radia Perlman's button box "grown up" (cf. Chapter 3-E). Though developed independently, they are similar in design. Instead of the elemental turtle operations like "go forward", PYGMALION incorporates general-purpose programming operations like "store value". But the instant-response, visually-oriented, "teaching," error-free characteristics and philosophy are the same.

I want to emphasize that PYGMALION is not a graphical programming language in the traditional sense. Graphical programming languages have all attempted to find two-dimensional ways to *tell* programs what to do. This inherently involves the manipulation of formal representations of data. PYGMALION has no representation for *telling* a program anything; PYGMALION is an environment for *doing* computations. If the system happens to remember what is done, then a program is constructed as a side effect. But the goal of the programmer is to *do a computation once*. This is helpful for understanding in any case: *a good way to understand a complicated algorithm in any language is to work through it with representative values*. Instead of using the medium of paper or blackboard, the PYGMALION programmer uses the display screen.

Section B -- The PYGMALION Philosophy

The overall philosophy of PYGMALION is summarized in the following list. These are the design principles guiding the implementation. The individual principles derive from the model of thought in Part I.

(1) VISUAL: The system is visually oriented.

(2) CONCRETE: Since for many subjects concrete information is easier to deal with than abstract concepts, the form of information manipulated is explicitly represented, rather than implicitly described. Arguments and values of functions are analogical.

(3) PARTIAL: Since people deal with selected, incomplete fragments of memory images, data structures may be left partially instantiated and routines partially specified, with traps on incomplete paths.

(4) LEVELS OF DETAIL: Since the quantity of information a person can handle comfortably (his short term memory) is limited, the system incorporates a detail suppression control. A structure may be displayed at any level of detail, including a symbolic (i.e. string) representation.

(5) MULTIPLE ROLES: Icons are capable of assuming different roles, corresponding to the functions which images serve in thought. Variables may be signs, symbols or pictures of their values. An icon representing a program may simultaneously be a part of a picture or data structure.

(6) SCHEMATA: In order to provide schemata for problem solving, generic prototypes of common operations are provided, such as conditionals, subroutines, iteration, recursion, sequentiality, subgoal hierarchies, and classes and subclasses. In addition, a rectangular shape representing a cell for storing information is provided as the default iconic shape.

(7) INCREMENTAL: Since creativity is incremental, programming proceeds in a step-by-step, interactive fashion, much as one uses an editor to change a body of text.

(8) TIME DEPENDENT: Information is capable of a time-dependent readout, since it is stored sequentially. The proper representation of a PYGMALION program is a *movie*.

(9) CONTEXT DEPENDENT: Since a person *projects* his internal models during perception, the system does likewise. It forms expectations about its input and interprets the input in light of those expectations. For example, the mouse buttons are context dependent.

(10) COMPUTABILITY: The system is a general purpose programming language, capable of computing anything computable (i.e. equivalent to a Turing machine).

Section C -- Iconology

> The mysterious way in which shapes and marks can be made to signify
> and suggest other things beyond themselves ...
>
> -- E.H.Gombrich[2]

Webster defines *"icon"* as "a pictorial representation, a vivid or graphic representation or description, something introduced to represent something else that it strikingly resembles or suggests, a reproduction or imitation of the form of a thing."[3] PYGMALION icons are two-dimensional, visual, analogical, concrete descriptions of concepts. They can be used to represent anything that can be drawn on a blackboard. (This is not suggesting that icons may only represent concrete concepts or that they must look like the concepts. Icons, like mental images, may be classified as "mimetic" or "non-mimetic" depending on whether they resemble objects or concepts in physical appearance.) Visual images are a powerful medium for portrayal in the mind. Except for the restriction to two dimensions, PYGMALION icons retain all the expressive power of mental images. Icons form the communication interface: person ↔ PYGMALION ↔ computer.

The primary entity used for computing in PYGMALION is the ICON.
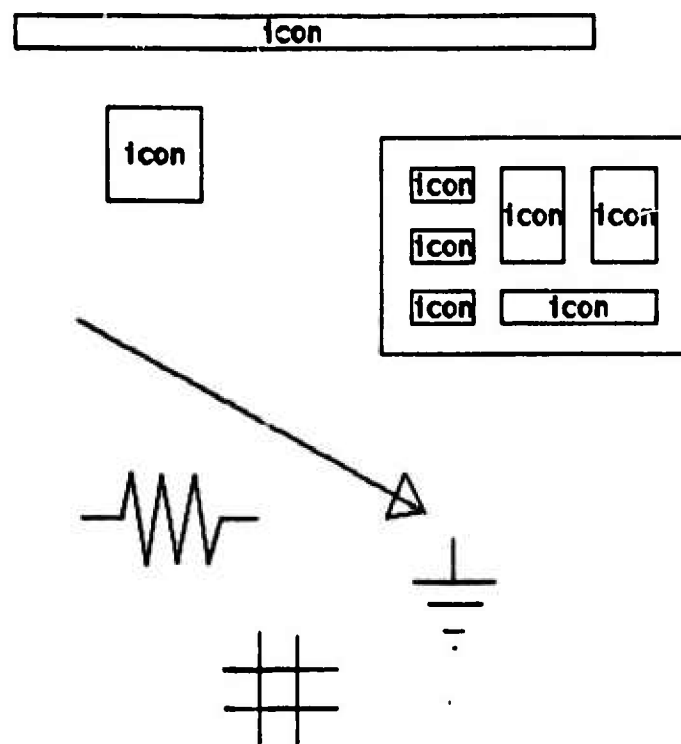
Examples of Icons

Programming in PYGMALION is a process of *designing and editing icons*. The PYGMALION programmer is an *"iconographer"* -- "a maker or designer of figures or drawings."[Webster[4]] The programmer of the future will be as well skilled in design as architects and artists today. Icons define the "PYGMALION machine". Everything that can be done to information is done through icons. Icons provide the mechanism for storing and retrieving information and for representing procedures. Icons exist on a variety of conceptual levels. On the most primitive level, an icon may simply be a picture, a piece of data consisting of line drawings. On a symbolic level, an icon may represent a single machine bit, and the contents of that icon represent the state of the bit, either 0 or 1. On higher levels an icon may represent a machine word, a sequence of words, an arbitrary data structure, the state of the computation, all of memory, the entire computer, or indeed anything which can be simulated. Icons may also represent dynamic processes: functions, coroutines, interrupts. The virtue of PYGMALION lies in being able to use icons as metaphors for the objects to be manipulated. Every operation on icons affects the display state as well as the internal machine state. The programmer need interact with the system only on the display level, with the images he has created. The artist Alexander Cozens

taught his students to project their ideas onto random blots of ink, much as Rorschach did a century later The inkblots became a *source* of ideas to the artists. PYGMALION as an iconic programming language attempts to fulfill the same role. Icons provide an alternative representation which stimulates creative thought in the programmer.

Suppose, as an example, we want to design a controller for "spacewar" space ships. The display screen might contain the following:
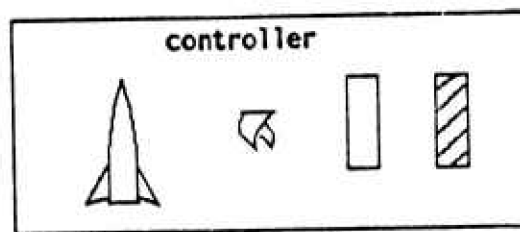
A Controller for Space War

This indicates there are four arguments to "controller": an object of type "ship", an object of type "thrust", an object of type "empty torpedo tube", and an object of type "full torpedo tube". This is far more descriptive than:

PROCEDURE CONTROLLER (SHIP S; THRUST T;
        EMPTY TORPEDO TUBE T1; FULL TORPEDO TUBE T2),

even assuming the programming language used permitted the above data types to be defined, which most languages do not. Some extensible languages, like SMALLTALK or LISP70 [Tesler,Enea,Smith[1973]], permit conceptual entities like "ship" to be easily defined and manipulated. But imagine trying to manipulate a ship represented as an array of numbers in FORTRAN. The amount of translation required from the mind's representation is enormous!

Icons have several "properties" or "attributes", some governing the visual (display) state and some governing the internal (machine) state. The total collection of attributes constitutes the "state" of an icon. Each attribute will be discussed in detail later in the paper where it is relevant. A complete listing follows.

Attributes of Icons

NAME   -- a string

VALUE   -- any object

SHAPE   -- code to generate the shape

BODY   -- code representing the functional semantics

DISPLAYED -- one of (FALSE, NAME, VALUE, SHAPE)

CONTAINER -- an icon

RUNCODE   -- code executed when the icon is told to "RUN"

FRAME   -- a display frame (a Smalltalk class); the
                 entire rectangular boundary of the icon

IX   -- a number, the X coordinate of the upper left
                 corner of the boundary

IY   -- a number, the Y coordinate of the upper left
                 corner of the boundary

IWD   -- a number, the width of the boundary

IHT   -- a number, the height of the boundary

FETCHER   -- code to retrieve an attribute of an icon

STORER   -- code to store a value into an attribute of
                 an icon

Icons also respond to several queries and commands. (In Smalltalk parlance, they respond to "messages".) The most important messages are *"has"*, *"run"*, *"fetch"* and *"store"*:

<icon> has x y

<icon> run

<icon> fetch <attribute>

<icon> store <attribute> <value>

"Has" and "run" are discussed in Chapter 7-F. "Fetch" and "store" are generic access paths to the iconic state. Every icon responds to the messages "fetch" and "store", but the actions taken are icon-specific. In fact, the semantics of "fetch" ("store") are: (a) bind some internal variables to the rest of the message, then (b) execute the code in the FETCHER (STORER) attribute. This has great potential power, but it has been largely unexploited in the initial implementation of PYGMALION. See Chapter 5-C for the chief application.

Section D -- Text Editors as Programming Languages

The actual implementation of PYGMALION is founded upon the following observation: People using interactive text editors on a computer, even untrained people, rarely make permanent-type mistakes in which they attempt to correct a line, fail to do so, and exit from the editor with the line still wrong. Since the editor is interactive, the user (if he is paying attention) will see that the line is still wrong and will simply edit it again until he corrects it. Thousands of people have used interactive text editors, many of them untrained in computer programming. Secretaries and office workers are beginning to use them because it is so easy to create documents, correct mistakes, and change words. But many of these same people shy away from programming because it is "too difficult".

Text editors begin to lose their error-free characteristics when they become batch oriented or when they employ complicated macros (for example, string-substitution macros) that operate on large sections of the text before they show the results. With these types of editors, changes are sometimes made that were undesired and are not detected until later.

Let's examine the nature of text editors a little more closely. Without too much difficulty, we should be able to see that *text editors resemble programming languages operating on a restricted domain*. Their operations, though restricted to text strings, have a functional similarity to operations in programming languages. For example, changing a character in a text string is similar to changing the value of a variable. I'm not going to belabor this point; the reader will be able to find numerous similarities. Instead, note that one difference between the two is that text editors *forget* their operations as soon as they have executed them. A typical cycle is (1) prompt for a command, (2) execute the operation, (3) display the result, (4) go to 1. In a program one wants the operations remembered, so that the program may be run as many times as desired. If, instead of forgetting, text editors *remembered* the operations as they were done, then the similarity with programming languages would become even closer. With such a "remembering" editor, we would not only be editing a body of text; we would also be *writing a program* that, given the same body of text containing the same errors, would automatically correct it. Furthermore, we could be sure that if we correctly edited the text once, every subsequent processing of the same text would also be correct! Of course this is seldom necessary, but some editors do maintain transaction files in case of machine crashes. At the heart of such editors is a powerful idea that may radically change the way software is written. For suppose that instead of limiting our editor to text, we expand its domain to include *arbitrary data structures*. And suppose we also expand its set of operations to include arithmetic, conditionals, subroutines, etc. Then we find our once-meek editor assuming all the capabilities of a general-purpose programming language. And the motivation for doing this is our claim that *it is a far easier programming language to use*.

> BASIC PYGMALION METAPHOR: a program is a series of EDITING CHANGES to a DISPLAY DOCUMENT. Input to a program is an initial display document, i.e. a display screen containing images. Programming consists of editing the document. The result of a computation is a modified document containing the desired information.

PYGMALION uses a remembering editor operating on iconic data structures. PYGMALION differs from all other programming languages in that there is *no static representation* for a program. One programs in PYGMALION by *doing* the operations on data structures directly, rather than by *telling* the program how to do them. When I first began this project, I spent a great deal of time trying to decide what is a good representation for communicating with a machine. Should it be at machine-language level, or higher level? Linear or multi-dimensional? Procedural or descriptive? Pattern-matching or imperative? Because of the reasons in Part I, I chose a two-dimensional representation for objects. But all two-dimensional languages I considered suffer from the same deficiency: they are too complex when representing the dynamic aspects of programming -- the semantics of operations and the flow of control. The representation had to be *articulate*. It had to correspond to representations in the mind. However I don't believe the current state of knowledge permits one to claim that some fixed notation is *the* mind's representation for *any* problem, let alone for *all* problems. In fact I believe that a myriad of representations are used in everyone's mind. The question then arises: why bother to have a predefined representation at all? The important thing is that the *mind* have a representation for a problem. Why not just solve the problem and let the computer "take notes"? The answer, which rather surprised me, is that *there is no need for an intermediate, predefined level of representation between the mind and the computer*. Any intermediate level is just extra work. The reason interactive editors like Engelbart's "NLS" and Swinehart's "TV" are so easy to use is that they don't interpose an intermediate level of work between a person's intention to do a task and the task itself.

Section E -- The PYGMALION Machine

In computer-aided design these relationships, between the aesthetic and technical side, must be made explicit, must be clearly and precisely formulated. There are two aspects of design to be considered: the visual and physical aspects of design as understood by the designer; and the mathematical representation of the design and associated information which is the form it takes inside the computer.

-- Anthony Hyman[5]

Hardware
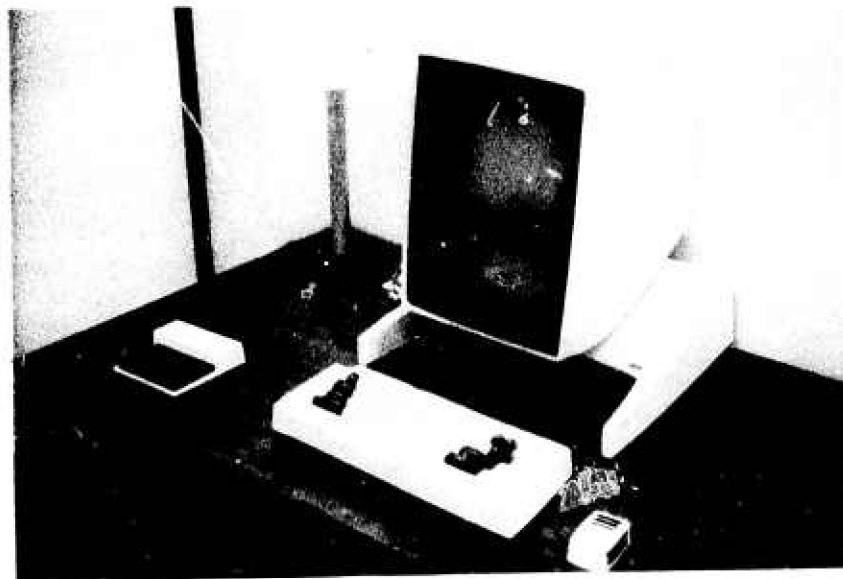
Display with Keyboard and Mouse

To use PYGMALION, the designer sits in front of a television-like *display screen*. The one shown is capable of displaying both text and graphics. In front of the display is a typewriter-like *keyboard* and a *"mouse"*. A mouse is a small device originally developed at SRI which has an X-Y tracking capability and three buttons on top that can be read under program control as three binary digits. Pressing a mouse button changes the corresponding binary digit from 0 to 1; releasing it changes the digit back to 0. Associated with the mouse is a *cursor* on the display screen; moving the mouse on the table moves the cursor correspondingly on the display. The cursor instantaneously follows mouse movements; mouse tracking is done in machine language. An interactive computer, a graphics display and a pointing device such as a mouse or light pen are the essential hardware elements of PYGMALION.

```
menu

icons
 create
 change
 delete
 copy
 refresh
 show
 name
 value
 shape
 body

opcodes
 +    -
 *    /
 =
 <    >
 and
 or
 not

control
 if
 repeat
 done
 eval
 return

others
 memory
 disk
 next
 display
 remember
 draw
 text
 trace
 constant
 plot
 exit
```

mouse value

mouse

remembered

smalltalk

PYGMALION Design Environment

4-E   The PYGMALION Machine

Software

When a programmer starts to use the system, the display screen shows the basic PYGMALION design environment. This provides a visual *schema*, an initial design metaphor. The basic environment consists of six icons and an empty area called the "design space". The initial icons are named (1) "world", (2) "menu", (3) "mouse", (4) "mouse value", (5) "remembered", and (6) "smalltalk".

(1) Icon "world"

Every icon is an instance of the Smalltalk class ICON. Two of the attributes possessed by every icon are a BOUNDARY and a CONTAINER.

BOUNDARY

The BOUNDARY of an icon is a rectangular area of the display screen. An icon is capable of detecting when the mouse cursor is within its boundary. The boundary of the "world" encompasses the entire display screen; therefore *all icons shown on the display lie within the boundary of the "world" icon*. This sets up a correspondence between the physical and logical characteristics of the display. The *upper left corner* of the boundary is called the *"origin"*. All coordinates used by icons are relative to this point. When a PYGMALION operation positions an icon on the display, it asks for a location for the origin. The origin of the "world" icon is location (0,0) on the display.

CONTAINER

"Containment" is a natural characteristic of images. The rule is:

> The container of an icon "I" is the most recent icon created and currently being displayed whose boundary physically encloses I's origin.

The reason for this particular convention is related to the notion of "iconic context" and is explained in Chapter 5-F. Ordinarily physical containment is the same as logical containment; i.e. if the boundary of icon A encloses icon B on the display screen, then icon A is B's CONTAINER. In the top two pictures below, icon A contains icon B; in the bottom two pictures neither icon contains the other, because neither contains the other's upper left corner.
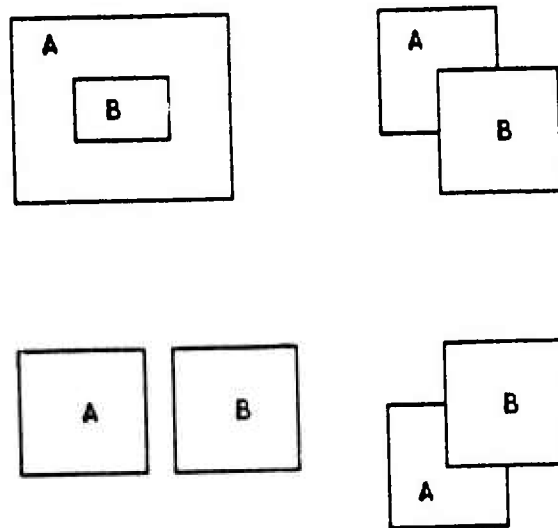
Iconic Containment

An exception to the containment rule is provided by the "change container" command. (Individual operations are explained in detail in the next chapter.) It is occasionally useful for icons to be logically connected while physically separate. The change container command explicitly sets the containment relationship between two icons. In the example above, icon A could explicitly be made the container of B, or vice versa, by using this command.

Iconic Structures

The container attribute is an integral part of the internal semantics of icons. It is used to organize icons into hierarchical structures. The container of an icon is itself an icon. Every icon has a unique container, and every icon may, but need not, contain other icons. If an icon does contain other icons, its VALUE attribute (discussed later) is said to be an *"iconic structure"*. The VALUE of the "world" is an iconic structure containing all the top level icons.
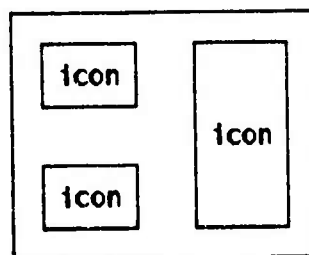
An Icon Containing Three Sub-Icons

An *iconic structure* is a collection of icons. The purpose of iconic structures is to provide a symbolic way to access all the icons on the display. Every icon is a member of some iconic structure. Internally iconic structures are organized as queues. New icons may be added only to the back of the queue, but icons can be deleted from any position. Internal descriptions of icons are in the form of *index lists*, in which each index is an offset from the front of a queue. Since the boundary of the "world" icon encompasses the entire display screen, all index lists begin with an offset in the "world's" queue. Icons are recursively organized. In addition to being a member of some iconic structure, every icon may contain its own iconic structure. The "world" is the root of the tree; it is the top-level structure that provides access to all the other icons. The index list (6 3 5) refers to the sixth icon (call it "A") in the "world's" iconic structure, then to the third icon (call it "B") in A's iconic structure, and finally to the fifth icon in B's iconic structure. While (6 3 5) is a Fregean representation internally, it is created analogically by pointing to the icon with the mouse cursor (see icon "mouse" below).

Since each icon has exactly one container, it follows that each icon must be unique. Making a COPY of an icon creates a new icon, with its own unique container. The container attribute may be changed by

(a) CREATEing an icon;

(b) FETCHing an icon from memory (core or disk).

(c) COPYing an icon;

(d) CHANGEing an icon's position;

(e) CHANGEing an icon's container explicitly;

(2) Icon "menu"

```
menu

icons
 create
 change
 delete
 copy
 refresh
 show
 name
 value
 shape
 body

opcodes
 +   -
 *   /
 =
 <   >
 and
 or
 not

control
 if
 repeat
 done
 eval
 return

others
 memory
 disk
 next
 display
 remember
 draw
 text
 trace
 constant
 plot
 exit
```

As mentioned, PYGMALION uses an interactive remembering editor operating on iconic representations of data. On the left of the display screen is a list called the "menu" of the editing operations and submodes that are currently available. An interesting aspect of PYGMALION is the scarcity and simplicity of its operations. The trend among high-level languages is to include more and more esoteric operations. PYGMALION runs counter to this trend. There are only a few basic operations, and they are grouped into four categories:

> *structure* -- operations which create and edit icons themselves

> *opcodes* -- arithmetic and boolean operations on values

> *control* -- operations which affect the flow of control in a program (conditional, iteration, subroutine)

> *others* -- miscellaneous operations to save icons in memory, turn remember mode on or off, draw a shape, etc.

The principle *structure* operations are CREATE, DELETE and COPY icons; CHANGE the size or position of icons; SHOW and REFRESH different levels of iconic structure; and fetch from and store into the NAME, VALUE, SHAPE or BODY attributes of icons. Fetching and storing attributes are comparable to fetching and storing the contents of a cell in memory -- the most primitive machine operations. The *opcodes* provide the standard arithmetic and boolean functions on values. The *control* operations provide conditional branching (IF), iteration (REPEAT, DONE), and subroutine invocation (EVAL, RETURN). The *miscellaneous* operations provide various other facilities that have been added from time to time: operations to save icons in memory (MEMORY, DISK), make line drawings (DRAW, TEXT), step through iconic structures (NEXT), turn display and remember modes on and off (DISPLAY, REMEMBER), trace the execution of iconic functions (TRACE), fetch constant values (CONSTANT), make a hard copy of the display screen (PLOT), and leave the PYGMALION environment (EXIT). No claim is made that this is a necessary, complete or even the most useful set of operations. PYGMALION is intended to demonstrate how a set of operations (a display metaphor) can be implemented iconically. The reader should be able to define his own set given the formalism presented in this report. Additional operations can be added to the menu at any time. In fact, the only difference between menu operations and iconic functions is that the names of the menu operations appear in the menu. The reason names (a Fregean representation) are used in the menu is that there is not enough space to display all the operations graphically. However iconically-displayed operations *can* be created, and Chapter 5-B has an example of an iconic menu.

Hyman has a perceptive comment on the use of menus:

> The presentation of a menu of choice is a powerful working tool. With a well designed system a teaching manual is largely redundant: the system incorporates a sort of programmed learning as a part of the design system. It may be held that the rhetoric of a subject is not to be learned that way, but it will serve well enough for the three Rs. It would also be of help to a designer transferring from one automated design to another. When the designer is freed from the burden of attention to detail, which grows enormously in a real production system, his attention is freed for creative design.[6]

A menu provides constant reinforcement about the capabilities of the system. It relieves the short term memory from constantly having to recall the available operations.

The programmer should quickly attain a high degree of confidence in the system because

(a) each operation is simple and easy to understand;

(b) its effect is immediately displayed;

(c) mistakes can usually be undone without harmful side effects.

A menu operation may be "*executed*" by pointing to it with the mouse cursor and pressing the top (DOIT) mouse button.
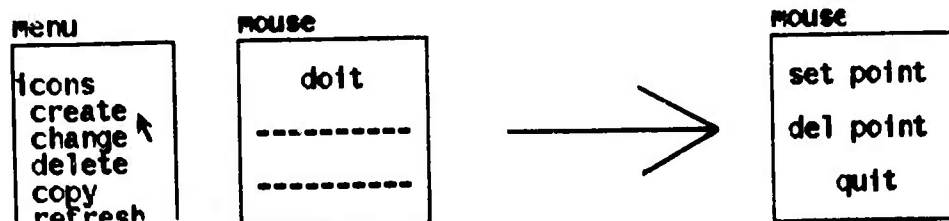

Executing a Menu Operation (note change in mouse state)

This typically sets up a new context in which subsequent commands are interpreted. One effect is to *project* an interpretation onto the mouse buttons.

(3) Icon "mouse"

mouse

The mouse is the primary input and control device in PYGMALION. The two main uses of the mouse are to:

(a) designate an icon

(b) accept button commands.
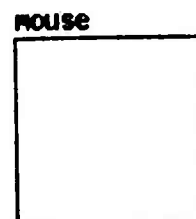
An icon is said to be "*designated*" when the mouse cursor is within its boundary.

Designating an Icon with the Mouse Cursor

Only icons currently on the display screen can be designated, and only designated icons can be affected by menu operations. What you see is what you get. This is a simulation of the principle that only images in the brain's short term memory are operated on by conscious thought. The display screen corresponds to short term memory. Icons not on the display must first be brought to display level by creating them or retrieving them from storage. Core and disk memory simulate the brain's long term memory. There is a fairly accurate correspondence:

The Mind		The Computer
short term memory	↔	display screen
long term memory	↔	core and disk memory
conscious thought	↔	operations on icons

(The simulation breaks down at the interface between short and long term memory. In PYGMALION icons are retrieved from memory by referring to their symbolic names, e.g. "factorial" or "resistor", under the control of a (conscious) menu operation. In the mind that would correspond to bringing everything into short term memory under the control of conscious thought, which is certainly not the case. As pointed out in Chapter 2, creative retrieval is often subconscious and involuntary. However, the rest of the simulation is as faithful as I can make it.)

Actions are initiated by pointing to an icon with the mouse and pressing one of the

mouse buttons. The mouse is context sensitive; the mouse buttons have different interpretations depending on which icon the cursor is in and which operation is currently being executed. The "mouse" icon acts as a prompt, displaying the current interpretation. Descriptions of the interpretations are represented in this paper by the following picture.

Explanation of the buttons

mouse
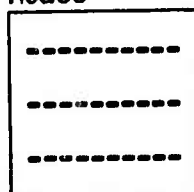
top button

middle button

bottom button

For the menu, the interpretation is:

mouse

doit
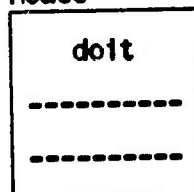
Execute the operation pointed to

(4) Icon "mouse value"

mouse value

In PYGMALION, the mouse has a *value* associated with it, in addition to its cursor and its buttons. Several operations like NAME and VALUE use the mouse value. Fetching an attribute from an icon may be thought of as *attaching* it to the mouse. The value may then be *deposited* in some other icon. The "mouse value" icon shows the current value attached to the mouse.

remembered

(5) Icon "remembered"

In "remember" mode (explained later) the system keeps track of operations as they are done and inserts them in a code list. The last operation or two remembered (i.e. in the current code list) is displayed in the "remembered" icon.

(6) Icon "smalltalk"   **smalltalk**

This icon is for communicating with Smalltalk. When the mouse cursor is in this icon, Smalltalk expressions may be typed and evaluated. In addition, other information about the state of the system is periodically displayed in this icon.

The Design Space

The rest of the screen is a large area called the "design space", available for displaying iconic data and program. The design space lies within the "world's" boundary, of course. The programmer perfor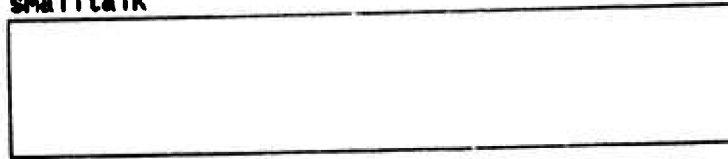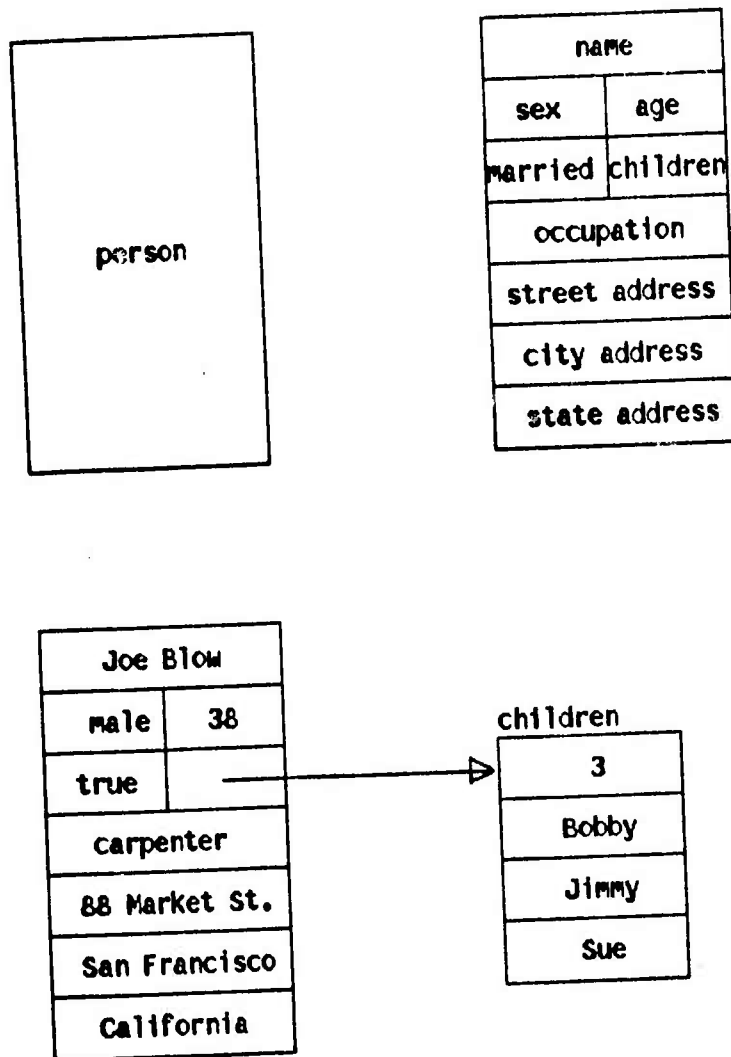ms computations in this environment by pointing with the mouse cursor to operations in the menu and then selecting operands from the data structures in the design space. At times an operation may request linear data, e.g. a number, or string, which may be input from the keyboard. The result of every operation is immediately displayed; if it is not what was intended, other operations can be executed until the desired state is achieved.

Routines are written by editing actual data structures and performing operations on actual operands. It is programming by example. Instead of trying to *imagine* what data objects are being passed around, the PYGMALION designer manipulates the *actual* objects. And instead of *telling* the machine the sequence of operations to perform by putting them down on paper, the PYGMALION designer *does* them himself, and the machine records them. With very few exceptions, programming languages have required programs to be written with formal arguments. This additional level of abstractness obscures the meaning of routines and is the single most important source of programming errors. In PYGMALION even formal arguments can be visual images of structures. For example, the structure of a data object can be concretely described while the values of the fields are formally represented. The following is an example of an iconic structure that might be used to represent the concept "person" in varying degrees of tangibility. Even more concrete information could, of course, be shown; the degree is up to the designer. The child "Bobby" below may itself be a reference to an instance of PERSON, and it could be displayed to the same level of detail if desired.

name	
sex	age
married	children
occupation	
street address	
city address	
state address	

person

Joe Blow	
male	38
true	
carpenter	
88 Market St.	
San Francisco	
California	

children
3
Bobby
Jimmy
Sue

Degrees of Representation of the Concept "PERSON"

Modes of Execution

There are two modes of execution in PYGMALION:

(1) Display mode

(2) Remember mode.

Every operation has a well-defined meaning in each of the modes, but not every operation does something in each mode.

(1) *Display mode* provides a means for communicating the semantics of operations visually. In display mode each operation shows iconically the results of its execution. This permits the user to remain in the display metaphor without having to deal with the internal semantics of operations. Display mode is usually on while a program is being written. It is also turned on when an iconic trace of a program is desired. Display mode may be turned off to gain additional speed of execution, since the display code in each operation will then not be executed.

(2) *Remember mode* is for writing programs. In this mode the designer *does* the computation and the machine *remembers* each operation as it is done. The operations place themselves in a *"code list"*. When the computation is complete, the designer executes "stop remembering" and assigns the code list to the BODY or SHAPE attribute of an icon. Programming in remember mode corresponds closely to the debugging process in most languages. Remember mode is similar to an interactive, display-oriented debugger that permits operations to be inserted and deleted in the code and that permits execution to be initiated from any point.

PYGMALION routines may be partially written and run while only partially specified; the specification has to be completed only when the code is about to be executed. This is frequently done with conditionals. When a program is written using actual data, only one branch of a conditional will be taken, with the other branch being left until (and if) it is entered. Every code list ends with a trap to the user asking for more instructions. If the list is completely specified (i.e. terminated with "stop remembering"), the trap is removed. Only if the trap on a partially specified branch is encountered will the programmer have to give additional instructions.

An unusual aspect of PYGMALION's remember mode is that it does not involve a compiler from the display-level actions to the machine-level actions. Rather *each operation is its own compiler*. Each operation is responsible

(a) for accomplishing a given internal machine task -- the machine "semantics" of the operation;

(b) in display mode, for generating a representative visual action;

(c) in remember mode, for adding onto the current code list the operation(s) necessary to reproduce itself.

The operations added are "primitive" in the sense that they are not built up out of other

operations available to the program designer. (An exception might be a macro-operation added by the designer to do some commonly-repeated task.) Thus the product of remember mode is a code sequence of the lowest-level operations possible. Ideally all or parts of these operations would be coded in machine language or microcode. In the current implementation they are coded in Smalltalk.

This concludes the basic hardware and software elements in the "PYGMALION machine".

Section F -- Characteristics of Programming in PYGMALION

Using PYGMALION causes several changes in the way software is constructed:

(1) *One may expect to write error-free programs the first time.* This is not the usual case with programming languages, where a great amount of theory and effort has been spent on the debugging process. It *is* the case with interactive text editors. The relevant characteristics are: (a) The system is highly interactive. *The results of operations are immediately visible, and mistakes immediately correctable.* (Therefore PYGMALION must be implemented on a computer that permits interaction, for example on a time-sharing system or mini-computer.) (b) *Operations are primarily concrete rather than abstract.* Some operations are inherently abstract, such as evaluating a subroutine or iterating over the "ith" element of a structure. But most operations are explicitly represented and their consequences immediately displayed. This concreteness simplifies the model-building (understanding) of the programmer. (c) *Data and program are visually represented.* Programs are display documents that can be visually examined and changed until they contain the desired information.

(2) *The standard mode for writing programs is what other systems consider to be debugging mode.* In the traditional writing mode, one works out the logic of a program and organizes the operations that will implement that logic. In debugging mode, one observes the step-by-step execution of operations on actual data, interrogates key values at certain times, and corrects (or at least notes) those operations that are in error. In PYGMALION there is no difference between the two modes. Designers of large systems, such as operating systems and programming languages, are beginning to realize that debugging tools should be designed and implemented *first*, as an integral part of the project. The more complex a system is, the stronger is the need to *see* what is going on. PYGMALION extracts the useful features of debuggers and integrates them directly into the programming process.

(3) *The human programmer is considered to be part of the execution of the program.* A program need not be completely specified in machine-executable terms. At certain points the human may be called upon to guide the machine through more operations, with the machine remembering what the human does. Thus there is a distinction between two kinds of programs in PYGMALION: *"open"* programs and *"closed"* programs. An *open* program consists of one or more machine-executable operations, of which one or more are requests

for human intervention; a *closed* program consists of one or more machine-executable operations, none of which are requests for human intervention. The program is "closed" in the sense that the human has told it everything it has to be told. With few exceptions, conventional programming languages require closed programs. Production software (i.e. intended for use by other people without the writer being present) should be closed. But software in development may, and in fact *should*, be "open", since such software is far easier to create.

(4) *No additional medium besides the computer is used in designing software*: no paper listings, no punched cards, no scratch paper. Routines are specified dynamically rather than statically, by *doing* rather than by *telling*. There has been no attempt to make a static representation for a dynamic process. A listing of a PYGMALION routine is a *movie*, a sequence of display frames. Examples of such movies are found in Chapter 8.

Section G -- PYGMALION versus Automatic Programming

> It is characteristic of computers that each time they intrude into a new subject they force practical decisions on questions which have hitherto been considered of a philosophical character, more suited to meandering discussions over cups of coffee far into the night than to the prosaic daylight world of industrial practice.
>
> -- Anthony Hyman[7]

The goals of PYGMALION are quite different from those of automatic programming. The general goal of automatic programming is to automate some of the programming process. In the ideal system, the human specifies the goal of the program in the form of input/output relations. The computer chooses internal representations, organizes the logic, maintains consistency, verifies correctness, and so on. Such tasks are problem-independent; they involve techniques that programmers learn from experience and apply to many different programs. By extracting problem-independent aspects, automatic programming hopes to make the process of programming easier. Balzer[1972] presents an example of a person communicating the semantics of an airline reservation system to a hypothetical automatic programming system. It is evident that such a system would require less superfluous detail than would conventional programming languages.

However there is a danger in this, if carried too far. By making the computer into a "black box" that does the actual programming, the user has to think less about the logical structure of the problem. Furthermore, the user's interaction with the computer is on a verbal level. Verbal communication is Fregean and inhibits understanding the developing program. The questions that Balzer has his hypothetical system ask often seem obscure, since the user does not know what logical relationships the computer is forming. The computer may even ignore the user's recommendations, for example on internal representation, if it thinks it knows better. If successful, automatic programming systems

will replace some fairly high-level thinking processes in humans. *Instead of encouraging humans to do more and better thinking, automatic programming may encourage humans to do less and poorer thinking.* At any rate, automatic programming is unlikely to stimulate the development of new problem-solving techniques in the user.

On the other hand, Kay and Papert have shown that learning to program can actually improve children's thinking and learning skills. The notion of debugging is particularly fruitful in giving children new learning strategies. Programming can serve as a catalyst for creativity; knowing how to program can provide the schemata necessary to solve problems, just as knowing how to play chess is frequently useful in real-world situations. With an articulate interface, computers can form a symbiotic relationship with a person which will vastly increase his thinking power. PYGMALION and systems such as those mentioned in Chapter 3 are attempts to provide such an interface.

92

Chapter 5

The Internal Structure of PYGMALION

Summary

(a) Icons can exist on a variety of conceptual levels: as pictures, symbols or signs.

(a) PYGMALION is a remembering editor for icons. Operations are provided to deal with icons on each of their levels.

Section A -- Smalltalk

PYGMALION is implemented in Smalltalk, a language designed by Alan Kay and implemented at Xerox PARC by his Learning Research Group. I highly recommend Kay's "personal dynamic media" and Smalltalk papers[1972(a),1972(b),1975] for an enjoyable reading experience. Smalltalk is an interactive symbol-processing language with flexible display primitives. Smalltalk, like LISP, is an interpreted language, with an evaluator for implicitly and explicitly evaluating objects. The dominant characteristics of Smalltalk are its notions of *"class"* and *"intrinsic semantics"*. In Smalltalk every object is an instance of a class. Classes themselves are instances of the class CLASS. This provides an elegant and consistent internal structure, even more consistent than pure LISP.

Communication between instances is done with *"messages"*. A message is a sequence of symbols. The set of messages that a class can handle, together with the responses to those messages, constitute the *"semantics"* of the class. (Hewett[1973] has derived a similar concept called "actors" from these Smalltalk ideas.) Since the code for processing messages is internal

to the definition of each class, the semantics are said to be "*intrinsic*" to the class. Neither the evaluator nor other classes know about the internal workings of a particular class; they know only to what messages the class will respond and what form the responses will take. The class NUMBER responds to the message "+" by performing an addition, but how it manages to accomplish the addition is usually not of interest. This differs from most other systems in which the semantics of objects are represented "extrinsically". In compilers the semantics of data types are represented by the code the compiler produces for them. Data types themselves have no intrinsic meaning; their meaning is distributed among the code generators. Having the semantics localized in a single definition is a significant aid to understanding and modifying data types.

Classes and Subclasses

Smalltalk is a descendant of the original SIMULA [Dahl[1966]] and FLEX[Kay[1969]]. Unlike SIMULA-67[Dahl[1970]], however, Smalltalk classes normally exist on the same conceptual level; that is, one does not normally think of Smalltalk classes as being subclasses of other classes (except class CLASS). Here we will briefly discuss the concept of hierarchical classes, since iconic data structures (cf. section D) are hierarchically organized. Hierarchical classes are not currently in Smalltalk, though they can be simulated.

The key principle of hierarchical classes is that a subclass is an *instance of an instance*. Currently Smalltalk objects are classes, instances of classes, and activations of instances. The notion of "subclass" involves a new kind of object: an instance of an instance. One creates an instance of a class, as before, and gives it instance-specific state information. Then one makes a copy of it, preserving all the filled-in state. Additional information is then added to the new instance, making it a subclass of the original instance. Subclasses are recursive; a subclass can itself have subclasses, forming a tree structure. To implement hierarchical classes, Smalltalk would need the additional capability of *dynamically adding attributes to instances*. This is necessary if changes to attributes at a subclass level are not to alter its superclasses. PYGMALION's iconic classes provide this capability since additional icons can be added to iconic structures at any time.

There are two kinds of behavior that subclasses can exhibit:

(1) *Upward Mobility*

(2) *Downward Mobility*

These refer to the information (and hence control) paths between class and subclass.

(1) In "upward mobility", each subclass can access all of the knowledge of all of its superclasses. In addition each subclass has some information of its own which causes it to be distinguished from its immediate superclass. For example, the class HUMAN might be discriminated into subclasses MAN and WOMAN based on the additional information, SEX.

```
           HUMAN
           /\
          /  \
        MAN   WOMAN
```
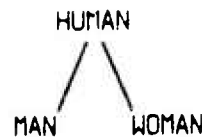
his additional state may involve changes in the values of superclass attributes, or it may
. entirely new attributes not present in a superclass. Changes made at the subclass level
do not affect its superclasses. For example, if subclass MAN changes the attribute HAIR-
LENGTH to SHORT, the change occurs at the subclass (MAN) level, not at the superclass
(HUMAN) level.

The key aspect of upward mobility is the *access method* that subclasses use to retrieve
information. In what we might call "*dynamic upward mobility*", a request for the value of an
attribute begins at the level making the request and searches up the tree of superclasses
until it find the first occurrence of the attribute. This is the most general form. SIMULA-
67 uses a more restricted form, which we might call "*static upward mobility*". SIMULA
copies superclass information into each subclass, so a subclass can access only the
information that exists in its superclass *at the time the subclass is created*. No new
knowledge can be added dynamically. (Actually creating subclasses is a compile-time action
in SIMULA; at run-time SIMULA does not even remember the class-subclass relationship.)

(2) In "downward mobility", any change in a superclass is reflected in all of its
subclasses. Again we might further discriminate this behavior into "*dynamic downward
mobility*" and "*static downward mobility*". Suppose the class HUMAN possesses an attribute
called NUMBER-OF-LEGS, and suppose its value suddenly changes from 2 to 1. In the
dynamic form, all the subclasses of HUMAN (i.e. MAN, WOMAN) are immediately
updated. In particular, any instance of HUMAN drawn on the display screen would now
be drawn with only one leg. In the static form, the change will be observed by the
subclasses if they ever ask for the value of the NUMBER-OF-LEGS attribute, but
otherwise they will never know. Static downward mobility is equivalent to dynamic upward
mobility.

Dynamic upward mobility requires that there be a link from each subclass to at least
its immediate superclass and that the access method for attributes search back through the
tree. Dynamic downward mobility requires that each superclass have links to all of its
subclasses and that the "change attribute" operation be monitored, so that changes in a
superclass can be broadcast to the subclasses. Static upward mobility as in SIMULA does
not require the presence of links, but it has the unfortunate trait that each subclass must be
larger than its superclass, since the superclass information is copied into it. By the time one
gets to the leaves of the tree, the subclasses might be quite large.

Smalltalk has proven to be an exceptionally flexible language in which to implement
PYGMALION, as well as a fascinating language in its own right. The SIMULA notions
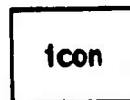of subclasses have also proven useful.

Section B -- Icons as Pictures

Icons derive their power of expression from the fact they are abstract in content but concrete in shape. The virtue of abstraction is that many different instances can be handled within one framework. The virtue of concreteness is that it provides a schema for organizing thought and reduces the possibility of mistakes. Arnheim points out that mental images can assume different functions: they can be signs, symbols or pictures of their values. PYGMALION icons can also assume any of these functions. This section deals with their most concrete use, as pictures. The operations described here all deal with the visual properties of icons themselves, as they appear on the display screen. The picture operations are:

CREATE   icon

COPY   icon   icon

CHANGE   icon   icon

DELETE   icon

REFRESH   icon icon   icon on

SHOW   NAME   resistor

             VALUE   10

             SHAPE   10 /\/\/\/\
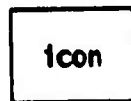
The user interface is basically the same for all of these, so we will only describe CREATE in detail. Executing the CREATE operation in the menu establishes a context for the mouse. The mouse icon will show that the user can now

mouse

set point
del point
quit

Set a point (a corner of an icon)

Delete the last point set

Quit

The user moves the mouse cursor into the design area and presses the top button. This fixes one corner of the rectangular *boundary* of an icon. As the mouse is moved around, the opposite corner follows the cursor, dynamically changing the shape of the rectangle. The middle button may be pressed to start over. Once an icon is started, pressing the top button a second time fixes the opposite corner. This completely defines the boundary of an icon. Each icon starts off with its boundary as its shape, but it can be given a fancier shape. The DELETE operation is the inverse of CREATE. The CHANGE operation can be used to change the position of an icon, the size of its boundary, or its CONTAINER attribute.

The CREATE, CHANGE and DELETE operations have internal semantics in addition to their display actions. As explained in Chapter 4-E, every icon is a member of exactly one iconic structure, and every icon can (but need not) contain its own iconic structure. The CREATE operation adds the created icon to the structure which contains its origin. If no icon encompasses its origin, it is added to the "world's" iconic structure. Similarly for DELETE; the deleted icon is removed from the structure in which it occurred. Normally physical containment and logical containment are the same: an icon logically contains another icon if it physically encloses the other icon's origin. The CHANGE operation can be used to explicitly change the container, and thus the structure containing, any icon. This enables icons to be logically linked while physically separate on the display screen.

(Aside: Most icons are pointed to only once, but multiple references to icons may be created if iconic structures are treated as values. For example, if the value of an icon is an iconic structure, and its value is "fetched" and then "stored" in another icon, there will now be two references to all of the icons in the structure. This is dangerous and not recommended. However it is sometimes done by the system in controlled circumstances.)

The COPY operation gives icons class/subclass characteristics. An icon can be constructed, which represents a template of a class. For example, the following is an iconic menu of classes of circuit elements. The elements are all icons which have been given various pictorial and internal properties. The COPY operation can be used to create instances of each class. All properties are copied by value, not by reference. Thus iconic subclasses display *static upward mobility* (cf. section A) in the current implementation. With the COPY operation items can be selected and positioned on the display without leaving the iconic metaphor -- that is, without having to refer to them symbolically.
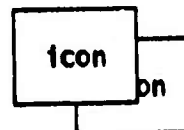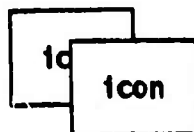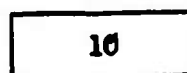
```
menu

icons
 create
 change
 delete
 copy
 refresh
 show
 name
 value
 shape
 body

opcodes
 +   -
 *   /
 =
 .   >
 and
 or
 not

control
 if
 repeat
 done
 eval
 return

others
 memory
 disk
 next
 display
 remember
 draw
 text
 trace
 constant
 plot
 exit
```
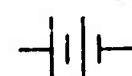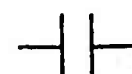
mouse value

remembered

smalltalk

mouse

copy icon

An Iconic Menu

The SHOW and REFRESH operations cause different levels of iconic representation to be displayed. The REFRESH operation simply redraws an icon at its current level. This is useful if it has been obscured by other drawings. The SHOW operation sets up the following mouse context.

```
mouse
┌──────────────┐
│ show name    │        Show the NAME of an icon
│              │
│ show value   │        Show the VALUE of an icon
│              │
│ show shape   │        Show the SHAPE of an icon
└──────────────┘
```

Usually pressing any of these buttons simply causes the corresponding attribute of the designated icon to appear on the display. However there are several special cases. If the NAME of an icon (see the next section) is already being displayed when the "show name" button is pushed, the effect is that (a) the name of the icon's *container* is displayed, and (b) all of the icons in the container's iconic structure are erased. If the VALUE of an icon (see the next section) is an iconic structure, then pressing the "show value" button causes all of the icons in the structure to display themselves. The "show name" and "show value" buttons can thus be used to move up and down in iconic structures, selectively displaying or suppressing structure. If an icon has a shape routine defined for it, then pushing the "show shape" button causes the shape routine to be evaluated.

SHAPE Attribute and Operation

The main pictorial attribute of icons is:

SHAPE -- code to draw the icon

The initial shape for an icon consists of the rectangular boundary with its name inside it. The default name is "icon".

```
┌──────────────┐
│              │
│     icon     │
│              │
└──────────────┘
```
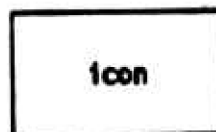
Default Iconic Shape

The default shape emphasizes the concept of "container": an icon is a cell which can hold information. It also indicates that the cell has a name and can be referenced symbolically

However icons are not limited to rectangular shapes. In general the value of an icon's SHAPE attribute is a user-definable routine which displays the icon's surface-level structure. A shape may be any display image that can be generated by a routine. This flexibility allows the iconographer to *design* his metaphors. He may give them whatever shape he considers appropriate to their meaning. This is another example of PYGMALION's attempt to program in the language of thought.
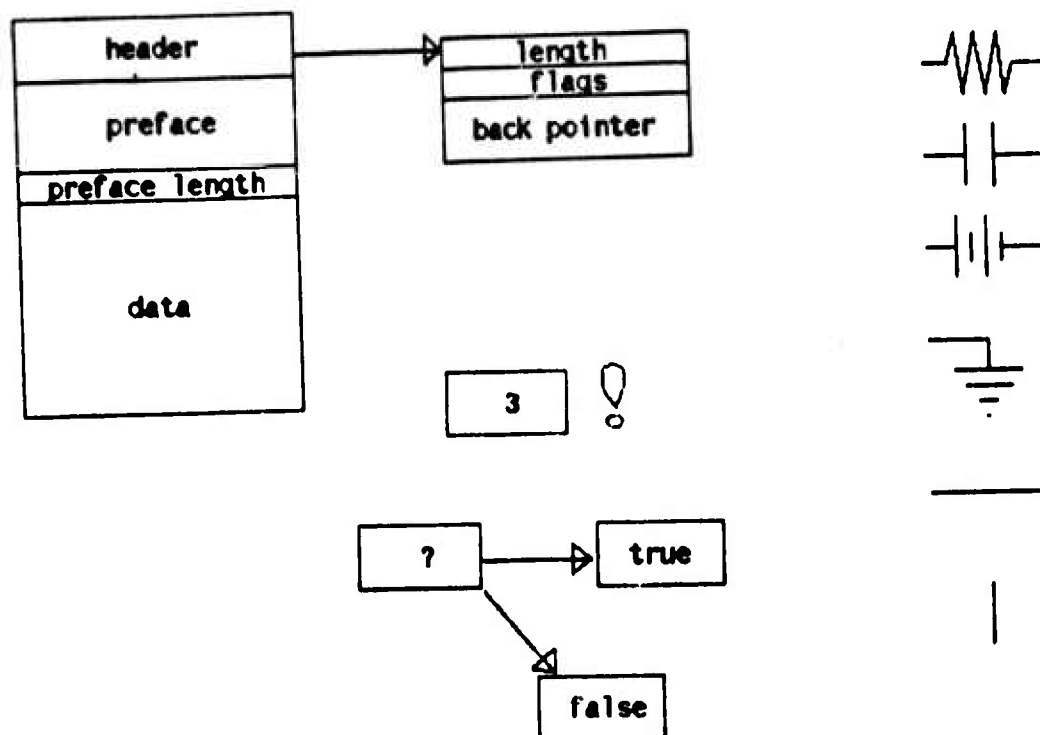
Some Icons with Fancy Shapes

Any icon can be given a fancy shape, regardless of its intended use: picture, variable, function, data type, etc. The value of the shape in communicating semantic intent is entirely dependent on the user's skill in designing images. It is the task faced by artists in communicating with the public. PYGMALION presents an empty canvas with only a few pre-defined icons and operations. A suggestion for the future (Chapter 7) is to develop within the PYGMALION paradigm a more extensive graphic vocabulary to serve as

schemata for the user. But the initial system presents just the tools and not the patterns for designing icons. (For those who can't draw, PYGMALION does provide tools for defining rectangular shapes: CREATE, COPY, CHANGE.) Defining a shape routine is exactly like defining an iconic function, except that the code assembled is put in the SHAPE attribute instead of the BODY attribute. Shape code is evaluated with the "show shape" button in the SHOW operation. (Function code is evaluated with the EVAL operation.) The SHAPE operation is used to define shape routines. The procedure is to put the system in "remember" mode, and then execute menu operations which draw the shape. This is discussed extensively in the section on icons as functions, section E. While any menu operation can be "remembered", the two most useful ones for shapes are DRAW and TEXT.

DRAW Operation

The DRAW operation is used for making line drawings. It sets up the following mouse context.

mouse

start line
stop line
quit

Start a new line

End the current line

Quit

Pressing the top mouse button causes a new line to be started at the current position of the mouse cursor. As the mouse is moved around, the other end of the line will dynamically follow the cursor (Sutherland's "rubber band" line). When the top button is pressed again, the other end point is fixed at the current cursor position and a new line begun from there. Pressing the middle button causes the end point to be fixed without starting a new line. The bottom button leaves the mouse context. The lines are all relative to the origin of an icon designated by the user; if the icon changes position, the lines move with it.
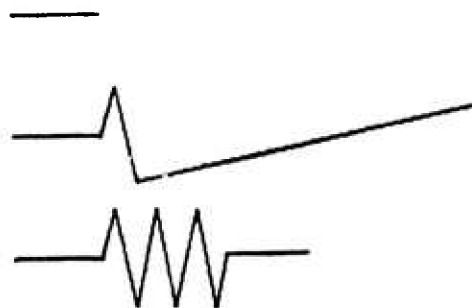
Drawing a Resistor

TEXT Operation

Text strings can be included in pictures with the TEXT operation. Executing it (a) first asks the user to type a string in the "smalltalk" icon, (b) then asks him to designate an icon to use as a base position, and (c) finally asks him to position the text on the screen. The text is positioned relative to the origin of the designated icon. As with line drawings, if the icon moves, the text moves also, maintaining the same relative position to the icon's origin

Section C -- Icons as Variables

The mathematical concept of "variable" has the attributes NAME and VALUE associated with it. These attributes are preserved in PYGMALION because icons can serve as variables

        NAME   -- a string

        VALUE -- any object; e.g. number, string, vector,
                    icon, iconic structure

But PYGMALION variables can have an additional attribute not usually possessed by variables, a SHAPE. For example, a variable which is to be bound to resistors might be represented by an icon having the following attributes:

NAME   —   "R"

VALUE   —   unbound

SHAPE   —   -WW-

NAME Attribute and Operation

The NAME attribute is an ordinary string of characters and is one way to refer to an icon. Names are used to suppress detail when complex structures are being manipulated. They can refer to icons without causing their shapes to be displayed. Names serve the same functions in PYGMALION that words serve in conscious thought: they provide a reference to other structures without exhibiting the full detail of the structures. Just as the *referents* of words, not words themselves, are used in thinking, the *referents* of names (i.e. icons) are used in computations in PYGMALION. This correspondence is another of the ways in which PYGMALION attempts to bring programming closer to thinking.

        Examples of names: "x"
                                  "alongname"
                                  "a multi-word name"
                                  "a strange @!12$3+._.& name".
                                  "UPPER AND lower case LETTERS"

The name of an icon can be changed by using the NAME operation in the menu. Executing the NAME operation sets up the following mouse context.

```
mouse
┌─────────────┐
│ fetch name  │        Fetch the name attribute
│             │
│ store name  │        Store the name attribute
│             │
│    quit     │        Quit
└─────────────┘
```

The top button attaches the name attribute of the designated icon to the mouse. The middle button deposits the mouse value in the name attribute of the designated icon. The

bottom button leaves this context. With these buttons the user can point to icons and manipulate their NAME attributes. The effect of the buttons depends on the icon designated; it is not the same for every icon. "Fetching" a name from the "smalltalk" icon causes a "read" to happen; the user may then type in a name or something which evaluates to a name. "Storing" into the "smalltalk" icon causes the value attached to the mouse to be printed. "Storing" into any of the other initial icons ("menu", "mouse", "mouse value", "remembered" or "world") is illegal.

VALUE Attribute and Operation

The VALUE attribute and VALUE operation are identical to the NAME attribute and operation, except that the VALUE attribute can be any object in the system, not just a string as with the NAME attribute. Since the VALUE of an icon can be any object, in particular it can be an iconic structure. (Recall that an iconic structure is an ordered collection -- a queue -- of icons.) Each of the icons in the structure can be used as a variable, so that any icon can effectively have more than one value. Indeed any sub-icon can have its own iconic structure as its VALUE, and so on indefinitely. In this way entire structures can be built up and manipulated.

Examples of values: atom
193
"a string"
(a vector)
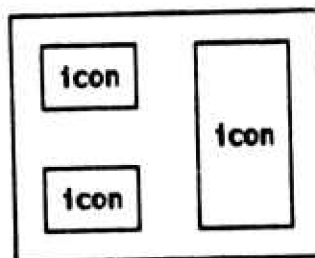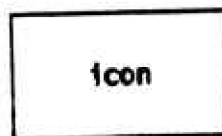
Note that the NAME and VALUE operations, like the picture operations described in the last section, are perceived by the user as *editing operations on display images*. They change the contents of the display screen as part of their execution. They also change the internal state, of course, but the mapping between this state and the display screen permits the user to deal with them on the display level, without bothering about the internal implementation. He perceives his task as *editing the display screen until it contains the desired information*.

Section D -- Icons as Data Structures

As mentioned in the previous section, an icon can have an iconic structure as its value. In that case we say the icon "contains" the iconic structure. That terminology derives from the fact that the iconic structure usually occurs physically within the boundary of the icon, as is the case with the example presented below:
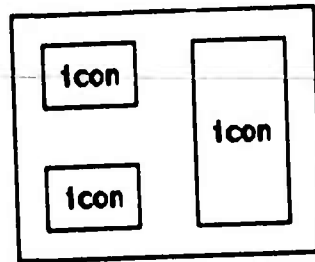
An Iconic Structure Consisting of Three Sub-Icons

Since sub-icons can themselves contain other icons, structures of any degree of complexity can be built. The structure may be a single level deep, as in the array below,
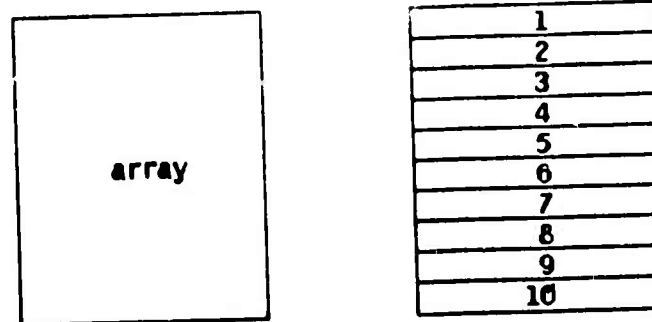
An Iconic Structure Representing an ARRAY

or the structure can have nested sub-levels.
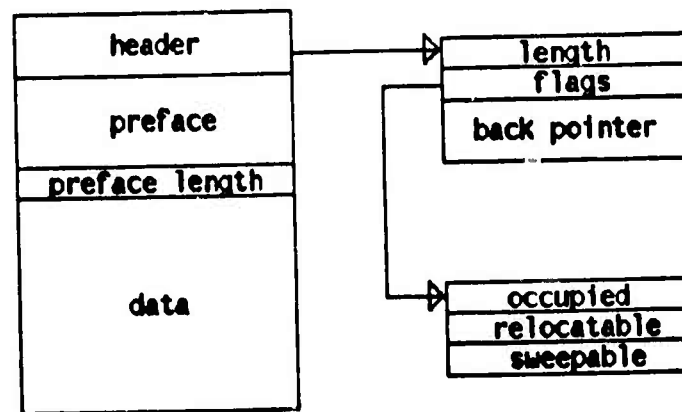
Nested Iconic Structure

Classes of objects can be constructed by creating an iconic structure containing the desired number of sub-fields, filling in the relevant sub-fields with type-dependent information, and then saving it in memory or on the disk. This provides a *visual template* for the data type. The following are steps defining a class of 10 ohm resistors.

Create an Icon

```
icon
```

Give it a NAME

```
resistor
```

Fill in the VALUE

```
10
```

Draw a SHAPE

10 $\bigwedge\!\!\bigwedge\!\!\bigwedge$

If we wanted arbitrary (instead of 10 ohm) resistors, we could simply leave the VALUE attribute unspecified. This is a very simple definition, involving no substructure. A more realistic simulation of resistors might have sub-icons containing current flow, connectivity information, and other model-specific details. The design flexibility and the visual concreteness of iconic structures permit the programmer to use them effectively to represent his ideas. They serve as *schemata in thought*.

Section E -- Icons as Functions

Another attribute possessed by every icon is:

```
BODY -- code for functional semantics
```

This permits icons to represent functions. The BODY of an icon can be evaluated at any time by executing the EVAL menu operation and then pointing to the icon to be evaluated.

Since the EVAL operation, like all operations, can be "remembered", subroutine calls to iconic functions can be included in a function's definition. An iconic function can EVAL itself, so that recursion is possible. This is similar to Radia Perlman's Button Box, but PYGMALION includes conditional and iteration operators for execution control. (Recall that Perlman's procedures had no way to terminate recursion.)

Defining Iconic Functions

Iconic functions can be defined in PYGMALION by executing the BODY operation in the menu. Basically this entails *putting the system in "remember" mode and then doing the calculation to be performed by the function.* This is precisely the "dynamic programming" philosophy of the Unimate robot, HP-65 calculator and Perlman Button Box: the programmer "teaches" the machine a procedure by *doing it once.* The following are the steps necessary to completely define a function.

(1) Create an icon. This is called the "function's icon" and serves as the framework for the definition. The code for the function will be put in the icon's BODY attribute.

(2) Give the icon a symbolic name. Symbolic names are used to invoke functions that are not on the display screen (i.e. that are in memory or on the disk). The function need not have a name if it will never be invoked symbolically, in which case it is like an unlabeled LAMBDA expression in LISP.

(3) Create sub-icons in the function's icon to hold individual arguments to the function. This makes the function's icon into a formal "argument icon". This need not be done if the function takes no arguments.

(4) START REMEMBERING.

(5) DO THE COMPUTATION ONCE.

(6) Create sub-icons in the function's icon to hold the values computed. This makes the function's icon into a "value icon". This need not be done if the function returns no values. Frequently the same structure can be used both as argument icon and as value icon.

(7) STOP REMEMBERING

The argument icon and the value icon provide the communications interface between routines. Anything in the argument icon when a function is EVALed may be used by the body of the function. Anything in the value icon when the function returns may be used by the caller. It is easily seen that functions in PYGMALION can take zero or more arguments (the contents of the argument icon) and return zero or more values (the contents of the value icon). So we get multi-valued functions easily. The argument and value icons

can be any size. In particular they can cover the entire design area. In that case everything on the display screen is passed to and returned from the function

As with iconic data structures, the argument and value icons correspond to the *schemata* used in thought. Once the argument icon is displayed, the designer fills it in with actual values. This eases the problem of interfacing procedures since the called procedure can display the structure of the arguments that it wants. It eliminates a major source of bugs: calling a procedure with the wrong number or type of arguments. Similarly the value icon helps to reduce confusion about the values functions return. A cleverly designed value icon can be of substantial assistance in keeping values straight. In a large program with many procedures, it is a difficult task to remember the calling/returning conventions. Fregean descriptions of parameters (e.g. "ARRAY X", "LIST L") provide little help. One of the aspects of PYGMALION I like best is the very real assistance provided by concrete, analogical argument/value icons.

Icons representing functions can, of course, have any of the other attributes of the class ICON, such as a SHAPE routine.

Remember Mode

"*Remember*" mode is one of the two modes in which PYGMALION can operate. (The other is "*display*" mode.) Remember mode is entered by evaluating either the BODY or the SHAPE operation in the menu, depending on whether the user wants to define a function body or an iconic shape. (Actually the difference between the two is purely a matter of convention: the SHAPE attribute could be used to hold a second function body if the programmer so desires. Both attributes use the same set of operations.) When the BODY or SHAPE operation is executed, an empty "*code list*" is placed in the corresponding attribute of the designated icon. A "code list" is a sequence of operations. A reference to the code list is also placed in the "remembered" icon. The code list in the "remembered" icon is called the "*current code list*". As mentioned in Chapter 4, each operation in the menu is capable of adding itself to the current code list. An internal flag called REMEMBERMODE controls this: when true, operations execute and add themselves to the current code list; when false, the operations simply execute. The BODY and SHAPE operations set REMEMBERMODE to true. The "stop remembering" option of the REMEMBER operation sets it to false when the computation is complete.

There are several display indications that the system is in remember mode. The notation "remembering..." appears at the top of the screen. The last operation or two done is shown in symbolic form in the "remembered" icon. The picture above shows that the user has just fetched the value 9999 from the fifth icon ("smalltalk") and stored it in the sixth icon (in the middle of the screen). Then the REMEMBER operation was executed to display the "remember" options available: remembermode can be temporarily suspended and then resumed, or stopped altogether. The suspend/resume option is useful to make adjustments to the display screen without having the operations remembered.

menu

remembering...

```
icons
 create
 change
 delete
 copy
 refresh
 show
 name
 value
 shape
 body

opcodes
 +    -
 *    /
 =
 <    >
 and
 or
 not

control
 if
 repeat
 done
 eval
 return

others
 memory
 disk
 next
 display
 remember
 draw
 text
 trace
 constant
 plot
 exit
```

```
9999
```

mouse value

9999

remembered

(fetch (5) value)
(store (0) value)

smalltalk

mouse

stop

suspend

resume

The PYGMALION Environment in Remember Mode

Calling Iconic Functions -- Iconic Contexts

Each iconic function carries with it an *"iconic context"*. An iconic context is an image of the display screen -- the state of the display that exists when the function is first defined. Internally an iconic context is just a copy of the "world's" iconic structure, in which the icons are copied into a new iconic structure *by reference*. Iconic contexts are necessary to insure that the display screen looks the same when a function is *evaluated* as when it was *defined*. (This is explained further at the end of the next section.) The following are the steps involved in calling an iconic function; the steps marked with an asterisk (⊛) are done by the user.

(1⊛) The argument icon for the function is brought to display level. Recall that the argument icon is an iconic structure containing cells for all the parameters to be passed to the function. The icon is displayed by retrieving it from memory, from the disk, or by making a copy of an icon already on the display screen. The parameter cells are then filled in with actual values.

(2⊛) When its argument icon is sufficiently (not necessarily completely) instantiated, the function is evaluated with the EVAL operation.

(3) The display screen is set to the function's iconic context. The display screen now contains the same number and types of icons as it did when the function was defined, except that the newly-instantiated argument icon replaces the argument icon used when the function was written.

(4) The BODY of the function is executed. Recall that the BODY attribute of an icon contains a list of menu operations. Some of the operations must place the values to be returned into the argument icon, turning it into a value icon.

(5) The display screen is restored to its state before step (3), except that the function's value icon now appears where its argument icon used to be.

(6⊛) The values in the value icon are attached with the mouse and deposited in other argument or value icons, for use in further computations.

Normally iconic functions should be defined with no superfluous icons on the screen. However some interesting effects can be acheived through the judicious choice of "permanent" icons, icons which remain on the screen across function boundaries. They can provide an alternative means of communicating between functions, similar to global variables in other languages. The following is an iconic context with a four-icon interrupt vector in one corner of the display screen. This can remain on the screen, and thus appear in iconic contexts, for a number of functions. Any of the functions can set or interrogate the icons. The icons themselves can be iconic functions that the user evaluates after he fetches or stores their values. This is an iconic version of *software interrupts*.

```
menu

icons
 create
 change
 delete
 copy
 refresh
 show
 name
 value
 shape
 body

opcodes
 +   -
 *   /
 =
 <   >
 and
 or
 not

control
 if
 repeat
 done
 eval
 return

others
 memory
 disk
 next
 display
 remember
 draw
 text
 trace
 constant
 plot
 exit
```

```
*interrupt*
 *error*
 *trap*
 *advise*
```

```
mouse value


```

```
mouse


```

```
remembered


```

```
smalltalk


```

A Shared Structure: An Iconic Interrupt Vector

Another possibility is to create a set of shareable cells for use as global variables. The picture below shows a four-cell structure similar to a FORTRAN "COMMON" area, of which three are taken by the (global) variables X, Y and Z. If more cells are needed, the user can always add additional icons to the iconic structure. The cells then be used to hold information for later access by other functions.

Local variables are easy to create and a very natural concept in this environment. A variable is just a cell for holding information. Since icons can have values, to create a new local variable the user has only to create a new icon. (The CREATE operation can be remembered.) The normal procedure is to position the icon in the design area so that it becomes part of the "world's" iconic structure, but it could be placed inside some other icon. The icon's VALUE attribute can then be set and retrieved using the mouse. Local variables can be created and deleted at any time as part of the definition of a function. The number of variables is limited only by the size of the display screen. Using the detail-suppression techniques of the last section, even physical size presents little constraint.

```
menu
┌──────────┐
│icons     │
│ create   │
│ change   │
│ delete   │
│ copy     │
│ refresh  │
│ show     │
│ name     │
│ value    │
│ shape    │
│ body     │
│          │
│opcodes   │
│ +    -   │
│ *    /   │
│ =        │
│ <    >   │
│ and      │
│ or       │
│ not      │
│          │
│control   │
│ if       │
│ repeat   │
│ done     │
│ eval     │
│ return   │
│          │
│others    │
│ memory   │
│ disk     │
│ next     │
│ display  │
│ remember │
│ draw     │
│ text     │
│ trace    │
│ constant │
│ plot     │
│ exit     │
│          │
│          │
└──────────┘
```

```
                            common
┌──────────────┐      ┌──────────────┐
│ *interrupt*  │      │      X       │
├──────────────┤      ├──────────────┤
│  *error*     │      │      Y       │
├──────────────┤      ├──────────────┤
│  *trap*      │      │      Z       │
├──────────────┤      ├──────────────┤
│ *advise*     │      │              │
└──────────────┘      └──────────────┘

mouse value                          mouse
┌─────────────────────────┐      ┌─────────────┐
│                         │      │             │
│                         │      │             │
└─────────────────────────┘      │             │
                                 │             │
remembered                       │             │
┌─────────────────────────┐      │             │
│                         │      │             │
│                         │      └─────────────┘
└─────────────────────────┘

smalltalk
┌──────────────────────────────────────┐
│                                      │
│                                      │
│                                      │
│                                      │
└──────────────────────────────────────┘
```

*Two Shared Structures: An Interrupt Vector and a Common Area*

OPCODES

We will illustrate the execution of iconic functions with some simple functions called "opcodes". (An illustration of the execution of FACTORIAL is presented in Chapter 6-B.) Opcodes are operations on values that are so frequently used that it is convenient to include their names in the menu. The semantics of opcodes are identical to those of iconic functions with one exception: since its name is in the menu, an opcode can be invoked -- that is, its argument icon brought to display level -- by simply executing its menu entry. Other functions are invoked by fetching them from storage using the MEMORY or DISK operation. As with all iconic functions, the body of the opcode may be evaluated as soon as the argument icon is sufficiently instantiated. The value icon replaces the argument icon when the execution is complete. The opcodes presently available are

Arithmetic opcodes: +   add
   -   subtract
                      *   multiply
                      /   divide

Boolean opcodes: =   equal
                      <   less than
                      >   greater than
                      and logical and
                      or logical or
                      not logical not

Since these all work the same way, we will only discuss "+" here. Executing the "+" operation in the menu initiates the following process.
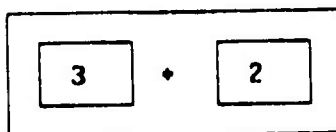
(1) The argument icon for "+" is brought to display level and positioned where designated by the mouse cursor. The "+" process then suspends.
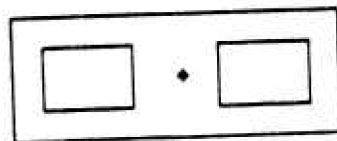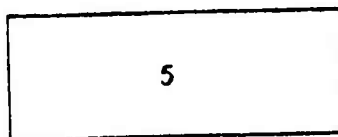

(2) "+" has two formal icons for holding the addend and augend. They may be given values.
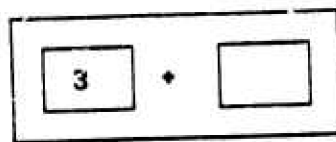
(3) As soon as both values are present, the body of "+" may be evaluated with EVAL. In this case the body simply computes the sum of the two values; i.e. 3 + 2.

(4) When the body finishes executing, the value icon is displayed. In this case it is simply a rectangle containing the number which is the sum.

Actually, the value returned by "+" need not be a number at all. To be completely acccurate, the value is the result returned by passing to the first argument the message "+" together with the second argument. The position taken by most extensible languages, Smalltalk included, is that symbols like "+" have no intrinsic meaning. Symbols are defined by the rules which use them. For example, the rule

$$<INTEGER> \; + \; <INTEGER> \; \rightarrow \; \ldots$$

might be used to define "+" to be integer addition, but another rule

$$<REAL> \; + \; <REAL> \; \rightarrow \; \ldots$$

might be used in the same translator to define "+" to be addition of real numbers. In Smalltalk, *classes* have intrinsic meaning, not symbols. The symbol "+" has meaning only insofar as classes know how to deal with it. For example, Smalltalk objects of class NUMBER (e.g. 3) contain code to handle messages containing "+", understanding it to mean ordinary addition. Objects of class STRING (e.g. "abc") can also handle "+", but in this case it is interpreted to mean "concatenate" The semantics of classes are intrinsic in Smalltalk, the semantics of symbols are extrinsic. This distinction is preserved in PYGMALION

The body of opcodes [step (3) above] may now be more accurately described as follows:

(3') As soon as both values are present, the body of "+" may be EVALed. The body passes to the first value the message "+" together with the second value; e.g.
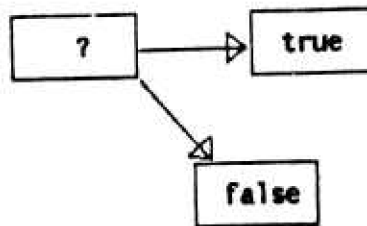
<value 1>   +   <value 2> .

The first value must be an object that knows how to handle the message "+"
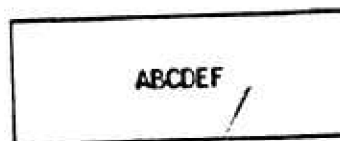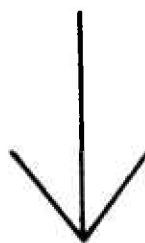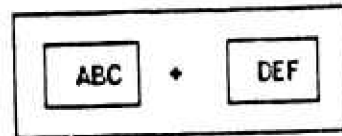
IF Operation

Conditionals are provided by the IF menu operation. Executing it brings the following argument icon to display level.

The "if" icon

The "if" icon contains three sub-icons. Only the value of the sub-icon named "?" is of interest. It must be assigned a value before the "if" icon is EVALed. The BODY of the "if" icon tests the value of the "?" icon and depending on the result evaluates the BODY of either the "true" icon or the "false" icon. Initially the BODYs of these icons consist of code lists containing only a trap to the user asking for more instructions. When the trap is encountered, the system is automatically put in remember mode. The BODY can then be defined in the usual way.

REPEAT Operation

Iteration is provided by the REPEAT menu operation. The "repeat" icon has the following shape.
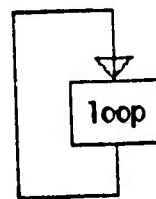
The "repeat" icon

The "repeat" icon has one sub-icon named "loop". It has the same flavor as the "if" icon, except that its semantics are: successively evaluate the BODY of the "loop" icon until the DONE operation is encountered. The DONE operation is for terminating repeat-loops. As with the "true" and "false" icons above, the "loop" icon's BODY initially consists only of a trap to the user asking for more instructions.

EVAL Operation

Subroutines are called with the EVAL menu operation. It may be used to evaluate the BODY of any icon. The icon must be displayed before it can be evaluated. This means that invoking an iconic function is a three stage process:

(1) Bring the function's icon to display level.

(2) Instantiate its argument icon with values.

(3) Evaluate its BODY with EVAL.

Section F -- Icons as Processes

The key to associating actions with icons on the display screen is a set of attributes possessed by icons:

```
DISPLAYED -- a data attribute

HAS       -- a procedure attribute

RUN       -- a procedure attribute
```

RUN

The RUN attribute is said to be a "procedure" attribute because a procedure, instead of a piece of data, is associated with it. (In precise Smalltalk terms, RUN is a *message* rather than an *attribute* of icons; but in this case there is no difference in the user's perception of the two.) PYGMALION is organized as a process structure. Each icon is an independent process. An icon *gains control* by being internally told to RUN and by the

mouse being within its boundary. It will retain control so long as the mouse remains within the boundary. The mouse, then, serves as a dynamic control manipulator. The user transfers control between icons by moving the mouse from one to the other. (An exception occurs if an icon, by error or design, does not give up control based on mouse movements. This is sometimes necessary to insure that a task gets completed before control is transferred.) The code associated with the RUN attribute for user-defined icons usually just invokes the VALUE menu operation. The "smalltalk" icon has run-code that waits for a character to be typed on the keyboard and then evaluates the subsequent Smalltalk expression. The "menu" icon's run-code executes menu operations. The "remembered" icon's run-code is an editor for code lists.

Process control derives from the "world" icon. The run-code for the "world" is: repetitively ask each icon if the mouse cursor is within its boundary; if it is, tell the icon to RUN. The "world" can access each icon because its value contains pointers to all of the top-level icons on the display screen. These in turn contain pointers to lower level icons.

DISPLAYED

The HAS and DISPLAYED attributes are the primitives used by PYGMALION to control the "aesthetic" and "technical" elements in design. The programmer/designer can manipulate only those icons that he can see, that are on the display screen. The DISPLAYED attribute can have any of the following values:

false - the icon is not currently being displayed.

name   - the name (and boundary) of the icon are
          currently being displayed.

value - the value of the icon is currently being
          displayed.

shape - the shape of the icon is currently being
          displayed.

Only those icons having a non-false DISPLAYED attribute can be designated with the mouse. And only designated icons can be the object of menu operations. The result is that only icons displayed in some form are accessible to the user.

HAS

Like RUN, the HAS attribute is said to be a "procedure" attribute because a procedure is associated with it. The procedure takes two parameters, an X coordinate and a Y coordinate. An icon can be asked if it "has" a particular value of X and Y; e.g.

<p align="center">i has mousex mousey .</p>

The procedure checks if the X-Y values given are within the boundary of the icon. In the example above, the icon "i" is being asked if the mouse is currently within its boundaries. HAS first checks if the DISPLAYED attribute is false, returning false if it is. Therefore, an icon will respond that it "has" an X-Y value only if it is currently being displayed in some form and the X-Y pair is within its boundaries.

Abstract Designation of Icons

There is no difficulty in designating icons when operations are first done. The programmer merely points to an icon with the mouse and executes a command (such as "fetch value") by pushing a mouse button. The command then operates on the icon to which the mouse is pointing. But suppose we are defining a program. With PYGMALION's "remembering editor" every operation done is added to a code list that eventually becomes the BODY or SHAPE of some icon. When the program is run, the operations will be re-executed, probably without the programmer even being present. It is impossible (and, in any case, undesirable) to require the programmer to point with the mouse to every icon to be operated on. He should have to designate the icons only while he is *writing* the program; thereafter the program must automatically act on corresponding icons. The problem is: how are "corresponding" icons to be identified without using the mouse? Three solutions:

(1) One solution might be to specify an icon by its X-Y coordinate location -- its two-dimensional "address" on the display. This is the two-dimensional analogue of absolute addressing, in which symbolic variables (like "X") in a program are mapped to absolute locations in memory during compilation. However, absolute addressing has several drawbacks, such as making recursion and relocation difficult. Recursive calls on a function would have to display the function's icons in the same place every time, overlaying and obscuring earlier calls. Furthermore, one might wish to rearrange the icons on the display screen for aesthetic reasons (permitted since PYGMALION's philosophy is that one *designs* a program, and designing involves aesthetics). It seems unreasonable either to prohibit rearranging icons or to have it invalidate code written earlier. (As Ivan Sutherland pointed out, the representations of images in human visual memory are invariant over size and position changes.)

A variation on absolute addressing might be to order the icons on the display topologically But again one might wish to rearrange them for aesthetic reasons, which should not invalidate code

(2) Another solution might be to specify icons by a unique label. We could give every icon in the system its own unique identifying symbol (such as a LISP "gensym") and use those symbols when dealing with icons. But again recursion causes difficulties. There is no way to tell at program-writing time what names will be used in recursive calls to a function at execution time. If recursion is permitted, the method of accessing icons must itself be recursive.

(3) The solution adopted by PYGMALION is a two-dimensional analogue of dynamic addressing. Some programming languages, e.g. ALGOL and LISP, use a stack to allocate space for variables. Every time a function is entered, a fresh cell is created on the stack for each local variable. This simplifies recursion at a slight cost in execution speed. As mentioned earlier, in PYGMALION iconic structures are actually *queues*. Whenever an icon is created on the display screen, it is added to the end of the queue of the iconic structure which contains it. There is a top-level icon called the "world" whose boundary encloses the entire display; so *every icon occurs within some iconic structure*. This queue structure establishes a *time sequential* access: icons which are put on the screen first occur "earlier" in an iconic structure than later entries; i.e. they are towards the front of the queue. Since each icon is contained in exactly one iconic structure, we can uniquely describe it by its queue index. The complete representation is a list of queue indices beginning with the outermost structure (the "world"). For example, the sixth icon on the display is represented by (6). This description is recursive. If the sixth icon on the display contains an iconic structure, then we may specify the second icon in the structure by (6 2). There is no theoretical limit to the depth of this nesting. This was discussed in Chapter 4-E.

Time sequential accessing is important to the "remembering editor" aspect of PYGMALION. Editing operations are done to actual icons on the display screen using the mouse; the icons are translated by the system into queue offsets when the operations are remembered. Concrete, aesthetic display images are transformed into abstract, technical queue offsets. However there is a difficulty. With the time sequential access method described, the same environment must exist when a function is *executed* as when it was *written*, or at least the same number and types of icons must be present on the display screen. Otherwise a queue representation like (6 2) may be invalid -- there may not even *be* a sixth icon on the screen. To permit the PYGMALION programmer to use functions written by others without requiring him to know in detail their environments, each iconic function carries with it an *iconic context* (cf. section E). When the function is evaluated, the screen is set to the state that existed when the function was defined, and restored after execution is complete. The function's argument/value icon is the primary means of communicating between the two contexts.

Chapter 6

# Examples of Purely Iconic Programming

This chapter imitates the branch of ancient Indian geometry that used only one word "BEHOLD" in its proofs (Chapter 1-G). The reasoning was done iconically. In this chapter sequences of display images are presented with a minimum of text accompanying them.

### Section A -- LISP70 Memory Organization

The first example is an iconic description of the LISP70 [Tesler,Enea,Smith[1973]] memory organization. Memory consists of a series of structured "blocks" of consecutive words of core. The block structure is presented here in various levels of detail. The purpose of this example is to demonstrate the expository capacity of icons and the detail suppression facilities in PYGMALION.
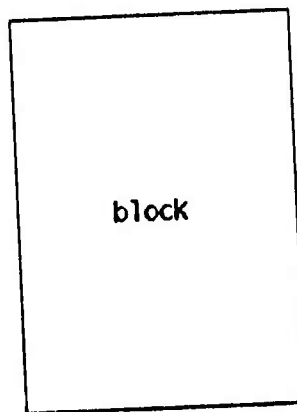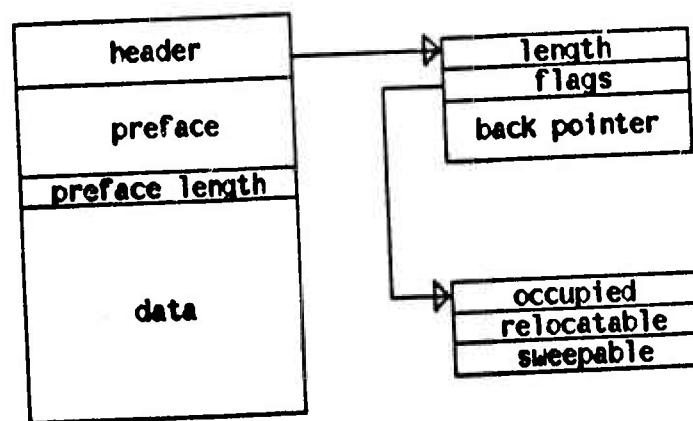
An Abstract BLOCK
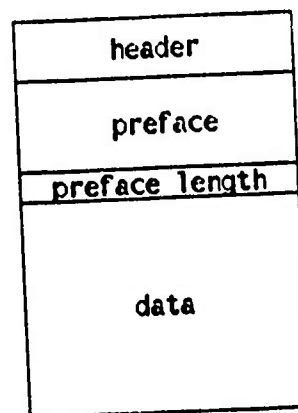
Gross Organization of a BLOCK
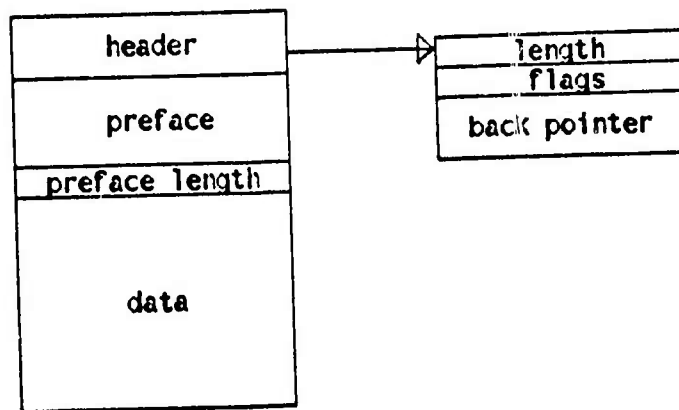
Structure of a BLOCK HEADER

Structure of the FLAGS field in a BLOCK HEADER

Section B -- Factorial

This example presents an iconic trace of the execution of the function FACTORIAL on the argument 6. On the display screen these frames appear in sequence, forming a movie of the execution. For completeness and to demonstrate that there is a static representation for PYGMALION functions, the code list assembled by these operations is included at the end. But this is not the most articulate notation. *The proper representation of a PYGMALION function is a movie.*

This example presents the iconic versions of these concepts in the context of a familiar function, FACTORIAL. While FACTORIAL is not a particularly iconic function, mapping numbers into numbers, it does exhibit several powerful programming concepts:

(a) the concept of "variable", and fetching from and storing into variables;

(b) the concept of "conditional";

(c) the concept of "recursion";

(d) arithmetic and boolean operations (+ - ⊗ =);

Note: the dashed lines in the following pictures are NOT part of the definition. They have been added by hand to clarify the movement of values. In actual use such clarification is unnecessary, as the movements of the mouse cursor adequately indicate which icons are affected.

menu

icons
 create
 change
 delete
 copy
 refresh
 show
 name
 value
 shape
 body

opcodes
 +
 -
 *
 /
 =
 <
 >
 and
 or
 not

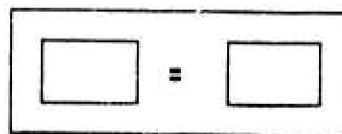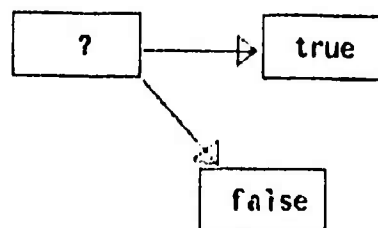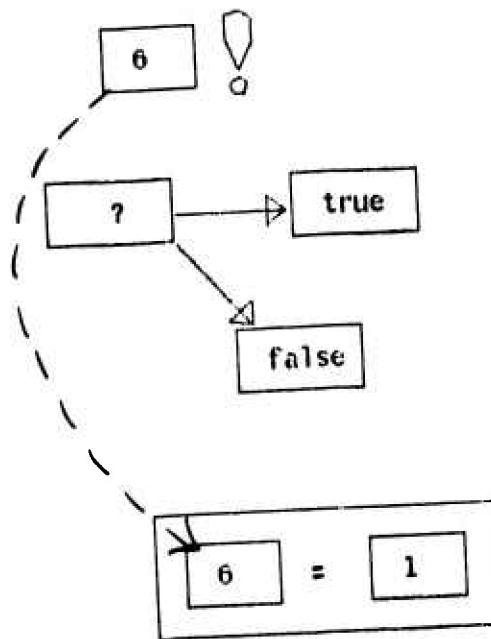control
 ?
 call
 return
 repeat
 done
 eval

others
 remember
 constant
 define
 display
 draw
 text
 break
 plot
 exit

6

mouse value
6

mouse

remembered

smalltalk

The BODY of FACTORIAL with the VALUE 6 is evaluated with the EVAL operation

```
menu

icons
 create
 change
 delete
 copy
 refresh
 show
 name
 value
 shape
 body

opcodes
 +
 -
 *
 /
 =
 <
 >
 and
 or
 not

control
 ?
 call
 return
 repeat
 done
 eval

others
 remember
 constant
 define
 display
 draw
 text
 break
 plot
 exit
```

```
   ┌──────┐    ╷
   │  0   │    ▽
   └──────┘    ○
```

```
mouse value                              mouse
┌──────────────────────────────────┐    ┌─────────────┐
│ 0                                 │    │             │
└──────────────────────────────────┘    │             │
                                         │             │
remembered                               │             │
┌──────────────────────────────────┐    │             │
│                                   │    │             │
└──────────────────────────────────┘    └─────────────┘

smalltalk
┌────────────────────────────────────────┐
│                                        │
│                                        │
│                                        │
└────────────────────────────────────────┘
```

The display screen is set to FACTORIAL's iconic context (no superfluous icons)

```
menu

icons
  create
  change
  delete
  copy
  refresh
  show
  name
  value
  shape
  body

opcodes
  +
  -
  *
  /
  =
  <
  >
  and
  or
  not

control
  ?
  call
  return
  repeat
  done
  eval

others
  remember
  constant
  define
  display
  draw
  text
  break
  plot
  exit
```

```
  +---+      (
  | 6 |       )
  +---+       O
```

```
  +-----+        +------+
  |  ?  |-----▷  | true |
  +-----+        +------+
       \
        ▷
     +-------+
     | false |
     +-------+
```

```
mouse value                              mouse
+------------------------+               +------------+
| 'If'                   |               |            |
|                        |               |            |
+------------------------+               |            |
                                         |            |
remembered                               |            |
+------------------------+               |            |
|                        |               +------------+
|                        |
+------------------------+

smalltalk
+------------------------+
|                        |
|                        |
|                        |
+------------------------+
```

A conditional icon is brought to display level

menu

```
icons
  create
  change
  delete
  copy
  refresh
  show
  name
  value
  shape
  body

opcodes
  +
  -
  *
  /
  =
  <
  >
  and
  or
  not

control
  ?
  call
  return
  repeat
  done
  eval

others
  remember
  constant
  define
  display
  draw
  text
  break
  plot
  exit
```
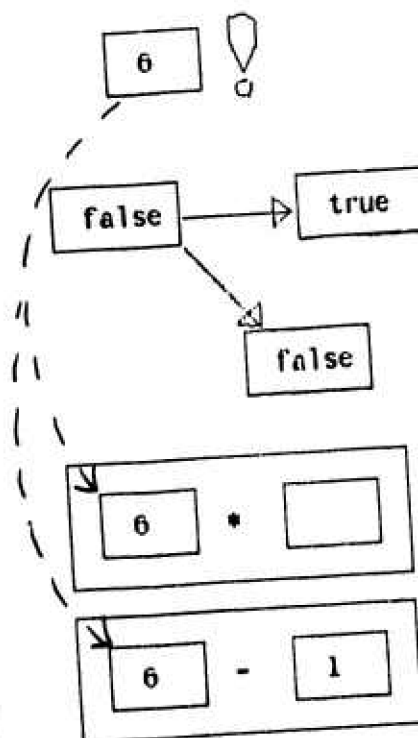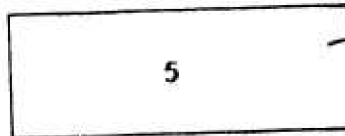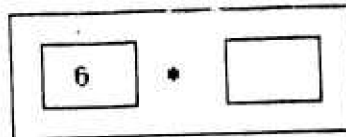
6

? → true

false

=

mouse value

'If'

remembered

mouse

smalltalk

An icon for testing equality is invoked

130

```
menu

icons
  create
  change
  delete
  copy
  refresh
  show
  name
  value
  shape
  body

opcodes
  +
  -
  *
  /
  =
  <
  >
  and
  or
  not

control
  ?
  call
  return
  repeat
  done
  eval

others
  remember
  constant
  define
  display
  draw
  text
  break
  plot
  exit
```
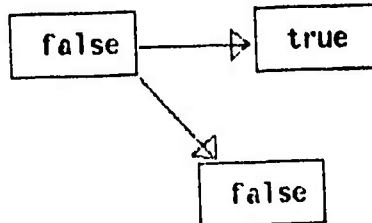
6

? → true

false

6 = 1

mouse value

1

remembered

mouse

smalltalk

The argument to FACTORIAL is tested against the constant 1

menu

icons
  create
  change
  delete
  copy
  refresh
  show
  name
  value
  shape
  body

opcodes
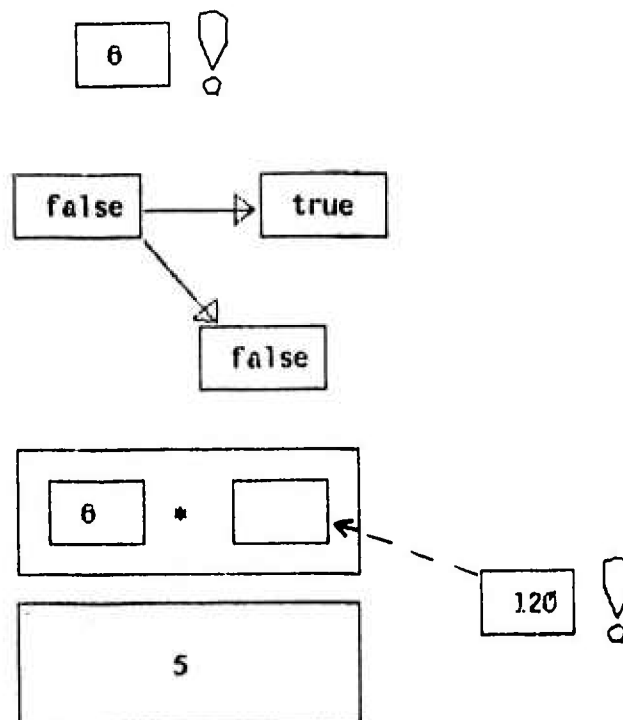 +
  -
  *
  /
  =
  <
  >
  and
  or
  not

control
  ?
  call
  return
  repeat
  done
  eval

others
  remember
  constant
  define
  display
  draw
  text
  break
  plot
  exit

6

? → true

false

false
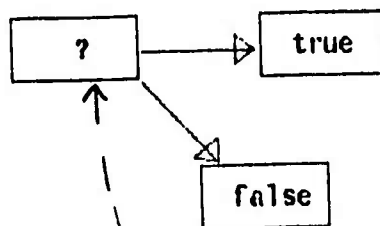
mouse value
1

mouse

remembered

smalltalk

132

```
menu

icons
  create
  change
  delete
  copy
  refresh
  show
  name
  value
  shape
  body

opcodes
  +
  -
  *
  /
  =
  <
  >
  and
  or
  not

control
  ?
  call
  return
  repeat
  done
  eval

others
  remember
  constant
  define
  display
  draw
  text
  break
  plot
  exit
```

6

false ➡ true

false

6 * [   ]

6 - 1

mouse value

1

remembered

mouse

smalltalk

The "false" icon causes 6 * FACTORIAL(6 - 1) to be computed

menu

icons
 create
 change
 delete
 copy
 refresh
 show
 name
 value
 shape
 body

opcodes
 +
 -
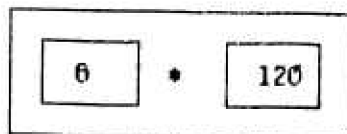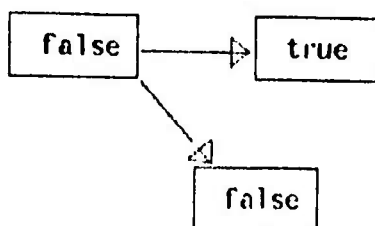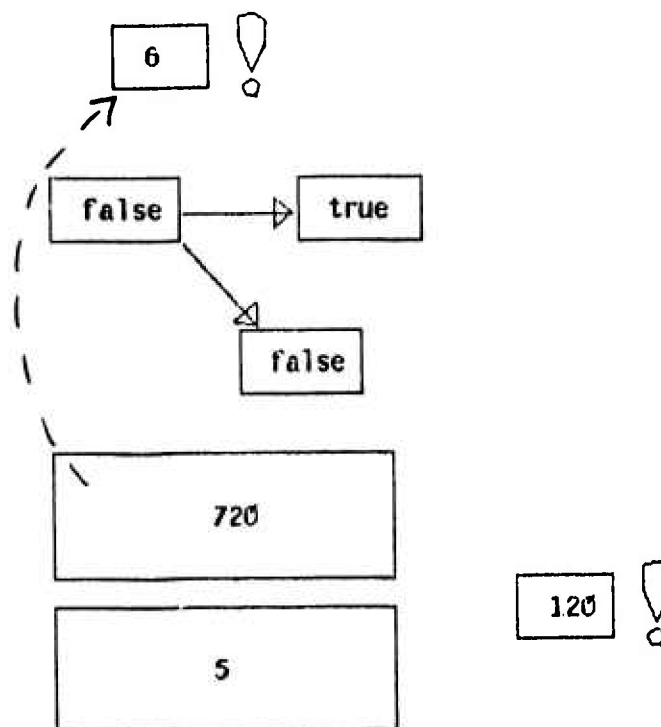 *
 /
 =
 <
 >
 and
 or
 not

control
 ?
 call
 return
 repeat
 done
 eval

others
 remember
 constant
 define
 display
 draw
 text
 break
 plot
 exit

```
         ┌──────┐    ◇
         │  6   │    ◇
         └──────┘    ◯

 ┌──────────┐        ┌──────────┐
 │  false   │───▷    │   true   │
 └──────────┘  ╲     └──────────┘
                ╲
                 ◁
              ┌──────────┐
              │  false   │
              └──────────┘

 ┌──────────────────────────────┐
 │   ┌──────┐        ┌──────┐    │
 │   │  6   │   *    │      │    │
 │   └──────┘        └──────┘    │
 └──────────────────────────────┘
                                        ┌──────┐   ◇
 ┌──────────────────────────────┐       │  5   │   ◇
 │                              │ - - ─▷ └──────┘   ◯
 │              5               │
 │                              │
 └──────────────────────────────┘
```

mouse value
5

mouse

remembered

smalltalk

Preparing for a recursive call on FACTORIAL

menu

icons
  create
  change
  delete
  copy
  refresh
  show
  name
  value
  shape
  body

opcodes
  +
  −
  *
  /
  =
  <
  >
  and
  or
  not
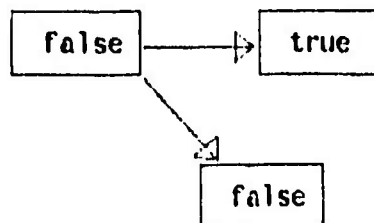
control
  ?
  call
  return
  repeat
  done
  eval

others
  remember
  constant
  define
  display
  draw
  text
  break
  plot
  exit

0

false ──▷ true

false

0 *

120

5

mouse value
5

mouse

remembered

smalltalk

FACTORIAL(5) = 120   (The intermediate steps have not been shown.)

menu

icons
  create
  change
  delete
  copy
  refresh
  show
  name
  value
  shape
  body

opcodes
  +
  -
  *
  /
  =
  <
  >
  and
  or
  not

control
  ?
  call
  return
  repeat
  done
  eval

others
  remember
  constant
  define
  display
  draw
  text
  break
  plot
  exit

6

false → true

false

6 * 120

5

120
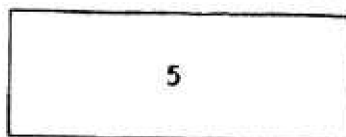
mouse value
120

remembered

mouse

smalltalk

The value 120 has been placed in the other half of the multiplication icon

menu

icons
  create
  change
  delete
  copy
  refresh
  show
  name
  value
  shape
  body

opcodes
  +
  -
  *
  /
  =
  <
  >
  and
  or
  not

control
  ?
  call
  return
  repeat
  done
  eval

others
  remember
  constant
  define
  display
  draw
  text
  break
  plot
  exit

6

false ▷ true

false

720

5

120

mouse value
120

remembered

mouse

smalltalk

6 * 120 = 720

menu

icons
 create
 change
 delete
 copy
 refresh
 show
 name
 value
 shape
 body

opcodes
 +
 -
 *
 /
 =
 <
 >
 and
 or
 not

control
 ?
 call
 return
 repeat
 done
 eval

others
 remember
 constant
 define
 display
 draw
 text
 break
 plot
 exit

720

false ⊳ true

false

720

5

120

mouse value
720

remembered

smalltalk

mouse

The value 720 is placed in FACTORIAL's argument icon, turning it into a value icon

menu

```
icons
 create
 change
 delete
 copy
 refresh
 show
 name
 value
 shape
 body

opcodes
 +
 -
 *
 /
 =
 <
 >
 and
 or
 not

control
 ?
 call
 return
 repeat
 done
 eval

others
 remember
 constant
 define
 display
 draw
 text
 break
 plot
 exit
```

720

mouse value

6

remembered

mouse

smalltalk

The display is restored to its state at the time of the call on **FACTORIAL**

```
((IF 192 128
    ((constant 1)
     (store (6) value))
    ((opcode • 160 272)
     (opcode - 160 352)
     (fetch (6) value)
     (store (8 1) value)
     (store (9 1) value)
     (constant 1)
     (store (9 2) value)
     (constant 'factorial')
     (memory fetch 384 320)
     (Eval (9))
     (store (10) value)
     (Eval (10))
     (fetch (10) value)
     (store (8 2) value)
     (Eval (8))
     (store (6) value)))
  (opcode = 208 320)
  (fetch (6) value)
  (store (8 1) value)
  (constant 1)
  (store (8 2) value)
  (Eval (8))
  (store (7 1) value)
  (delete (8))
  (Eval (7)))
```

Format of the instructions shown:

```
(IF <x coordinate> <y coordinate> (<true code>) (<false code>))

(opcode <name> <x coordinate> <y coordinate>)

(constant <value>)

(fetch (<icon>) <attribute>)

(store (<icon>) <attribute>)

(Eval (<icon>))

(memory [fetch,store] <x coordinate> <y coordinate>)

(delete (<icon>))
```

The code assembled as a side effect of computing FACTORIAL

Section C -- Circuit Simulator

Iconic menus are possible, and in fact encouraged. The initial menu is symbolic (contains names) to save display space. Below is a menu of electronic circuit elements. The COPY operation is used to select icons from the menu. Each icon has a semantically descriptive SHAPE attribute. In addition, each icon contains an iconic structure representing the electronic characteristics that are being modeled. The structure of the "power supply" icon is shown in one picture. The choice of structure is entirely up to the user.

The definition and execution of operations are not shown here; they are similar to the FACTORIAL example. The fact that the operations modify icons having fancy shapes instead of mere rectangles has little significance internally. Programming the semantics of a circuit simulation proceeds in the same way as programming any function. The difference between using PYGMALION and using a one-dimensional programming language is that no additional media (scratch paper, blackboard) need be employed. The display screen can be used to sketch out ideas.

menu

```
icons
 create
 change
 delete
 copy
 refresh
 show
 name
 value
 shape
 body

opcodes
 +    -
 *    /
 =
 <    >
 and
 or
 not

control
 if
 repeat
 done
 eval
 return

others
 memory
 disk
 next
 display
 remember
 draw
 text
 trace
 constant
 plot
 exit
```
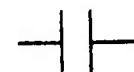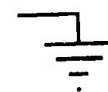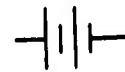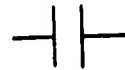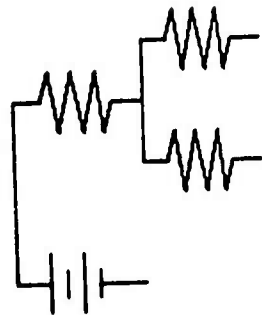
mouse value

remembered

mouse

copy icon

smalltalk

The COPY operation and a menu of electronic circuit elements

142
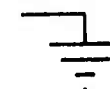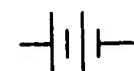
menu

icons
 create
 change
 delete
 copy
 refresh
 show
 name
 value
 shape
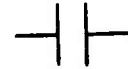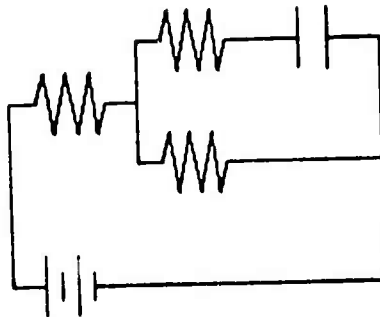 body

opcodes
 + -
 * /
 =
 < >
 and
 or
 not

control
 if
 repeat
 done
 eval
 return

others
 memory
 disk
 next
 display
 remember
 draw
 text
 trace
 constant
 plot
 exit

mouse value

remembered

smalltalk

mouse

copy icon

Several icons are copied from the menu and assembled into a circuit

menu

icons
 create
 change
 delete
 copy
 refresh
 show
 name
 value
 shape
 body
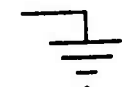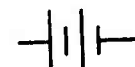
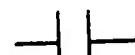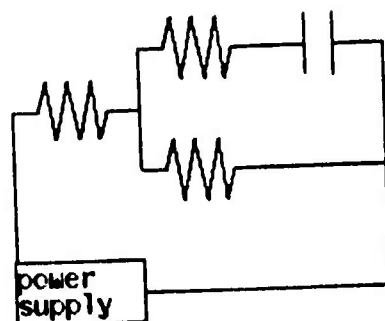opcodes
 +   -
 *   /
 =
 <   >
 and
 or
 not

control
 if
 repeat
 done
 eval
 return

others
 memory
 disk
 next
 display
 remember
 draw
 text
 trace
 constant
 plot
 exit

mouse value

remembered

smalltalk

mouse

The circuit is completed

```
menu

icons
  create
  change
  delete
  copy
  refresh
  show
  name
  value
  shape
  body

opcodes
  +    -
  *    /
  =
  <    >
  and
  or
  not

control
  if
  repeat
  done
  eval
  return

others
  memory
  disk
  next
  display
  remember
  draw
  text
  trace
  constant
  plot
  exit
```

power
supply

mouse value

mouse

remembered

smalltalk

The NAME and BOUNDARY of the "power supply" icon are shown

menu

icons
 create
 change
 delete
 copy
 refresh
 show
 name
 value
 shape
 body
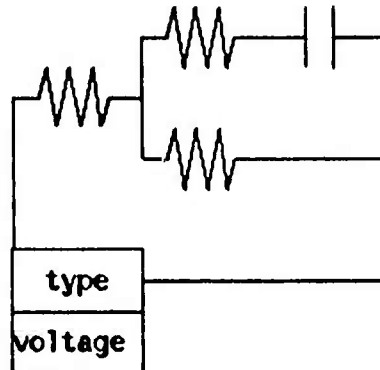
opcodes
 +   -
 *   /
 =
 <   >
 and
 or
 not

control
 if
 repeat
 done
 eval
 return

others
 memory
 disk
 next
 display
 remember
 draw
 text
 trace
 constant
 plot
 exit

type

voltage

mouse value

mouse

remembered

smalltalk

The iconic structure of the "power supply" icon is shown

menu

icons
 create
 change
 delete
 copy
 refresh
 show
 name
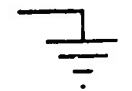 value
 shape
 body

opcodes
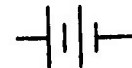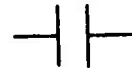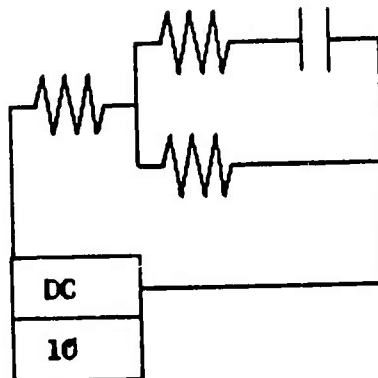 +   -
 *   /
 =
 <   >
 and
 or
 not

control
 if
 repeat
 done
 eval
 return

others
 memory
 disk
 next
 display
 remember
 draw
 text
 trace
 constant
 plot
 exit

DC

10

mouse value

10

remembered

mouse

smalltalk

The power supply is set to 10 volts, direct current

Section D -- Smalltalk Evaluator

This example contains part of a simulation of the Smalltalk evaluator. Several stages in the evaluation of a Smalltalk expression are displayed. The first two pictures present the principal memory structures involved: "activation records" (AREC) and "vector messengers" (VMESS). Subsequent frames show the creation and evolution of instances of these structures. The actual operations changing the contents of the fields are not shown, as they were in the FACTORIAL example; just the results of operations on the state of the structures are presented here.

```
to cons : hd tl
  (◀hd→
   (◀←→
     (⇑G⅋hd ← :)
   ⇑hd)
  ◀tl→
   (◀←→
     (⇑G⅋tl ← :)
   ⇑tl))
```

$$G⅋x ← cons\ 3\ 4$$

$$x\ hd + x\ tl$$

The Smalltalk statements involved are listed above in Smalltalk notation. Briefly, the symbols have the following meanings:

eyeball   peek ahead in the message stream for a literal
                    symbol

double right   conditional: evaluate the vector on the right
  arrow   if the expression on the left is true

double up   return the value of the expression on the right
  arrow

hand   quote the following atom

colon   fetch a message evaluated

The class presented simulates a LISP "cons" pair. It responds to the messages "hd" (head) and "tl" (tail). The variable "x" is assigned the cons pair (3 . 4). Then the expression "x hd + x tl" is evaluated. The display frames describe the execution from this point.
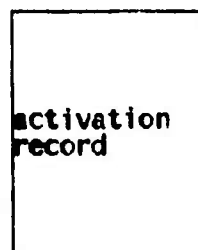
The principal memory structures used

The iconic structure of each element
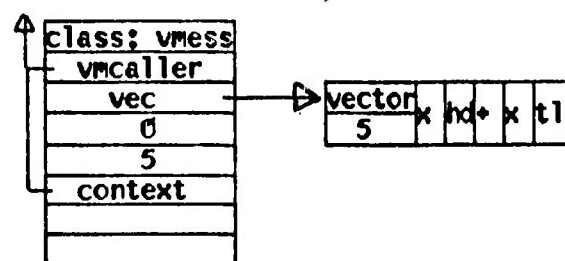
A VMESS is instantiated pointing to the vector (x hd + x tl)
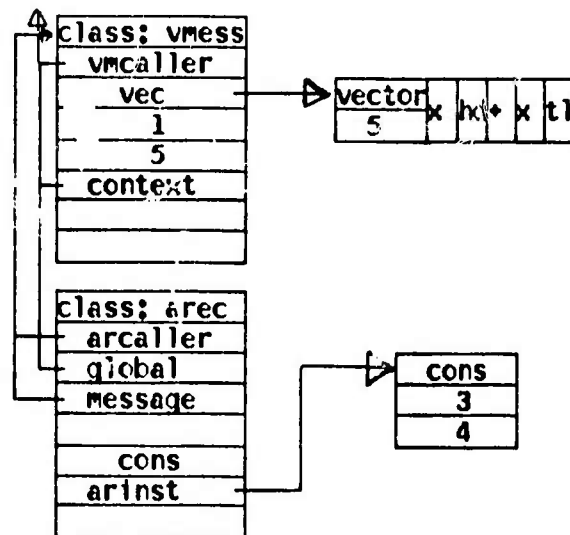
An AREC is instantiated pointing to the cons instance; note VMESS:pc → 1
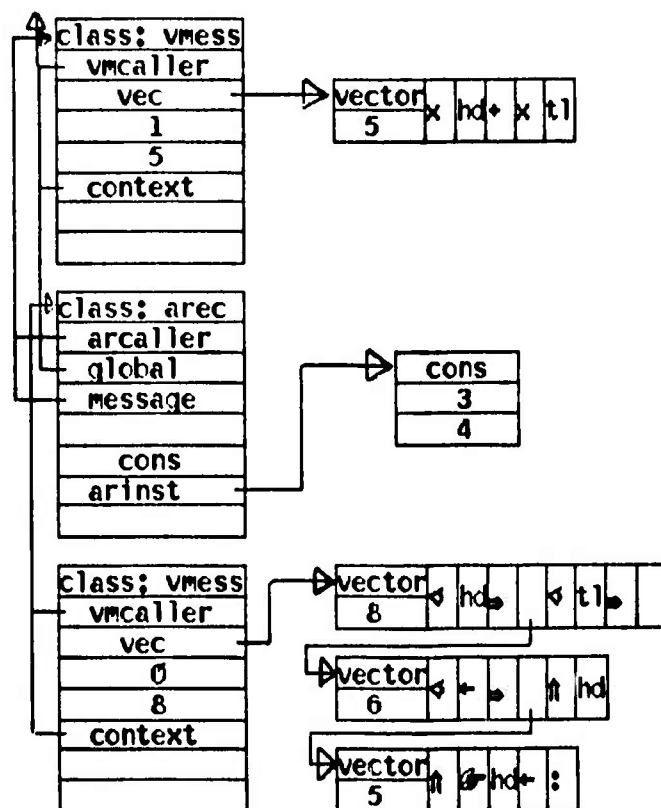
A second VMESS is instantiated pointing to the CONS message handler
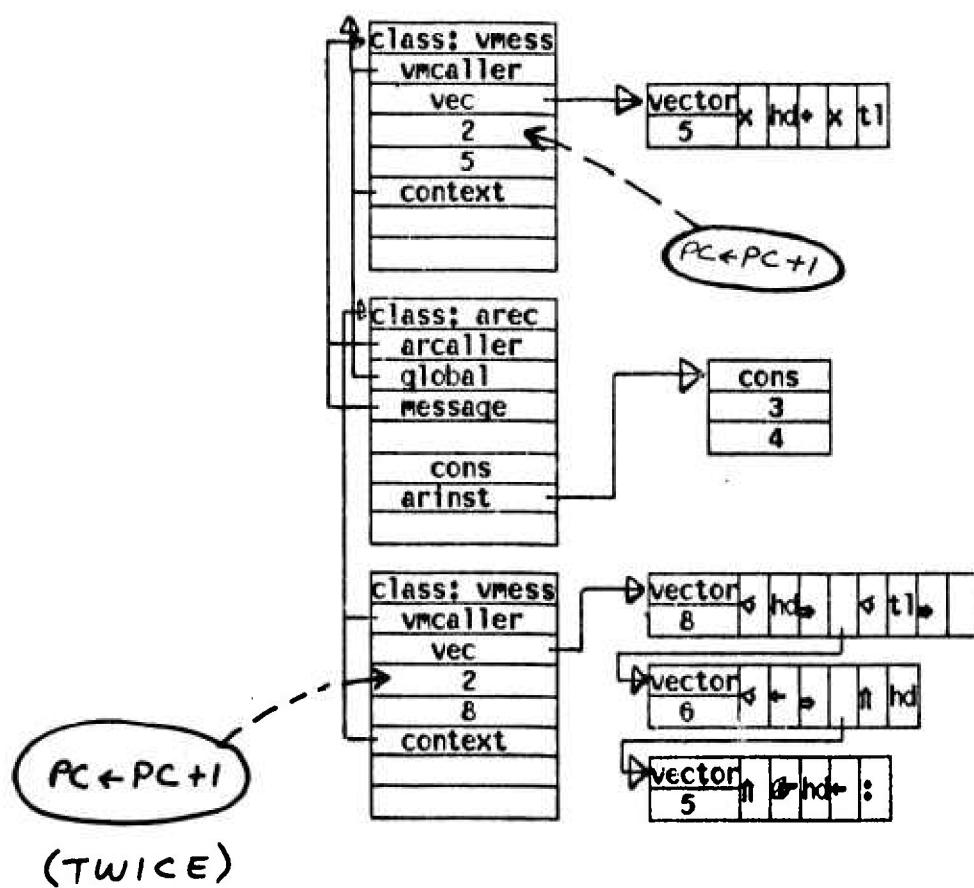
The message is scanned for the symbol "hd", and found
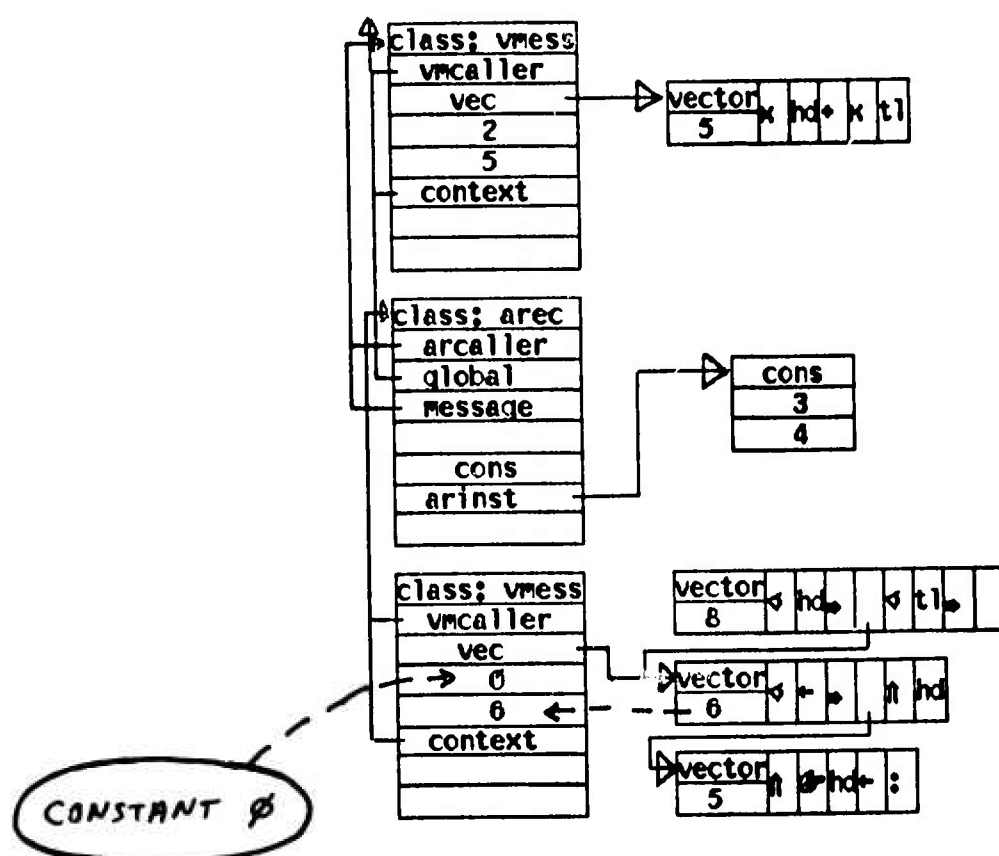
The second VMESS now points to the vector after the conditional arrow
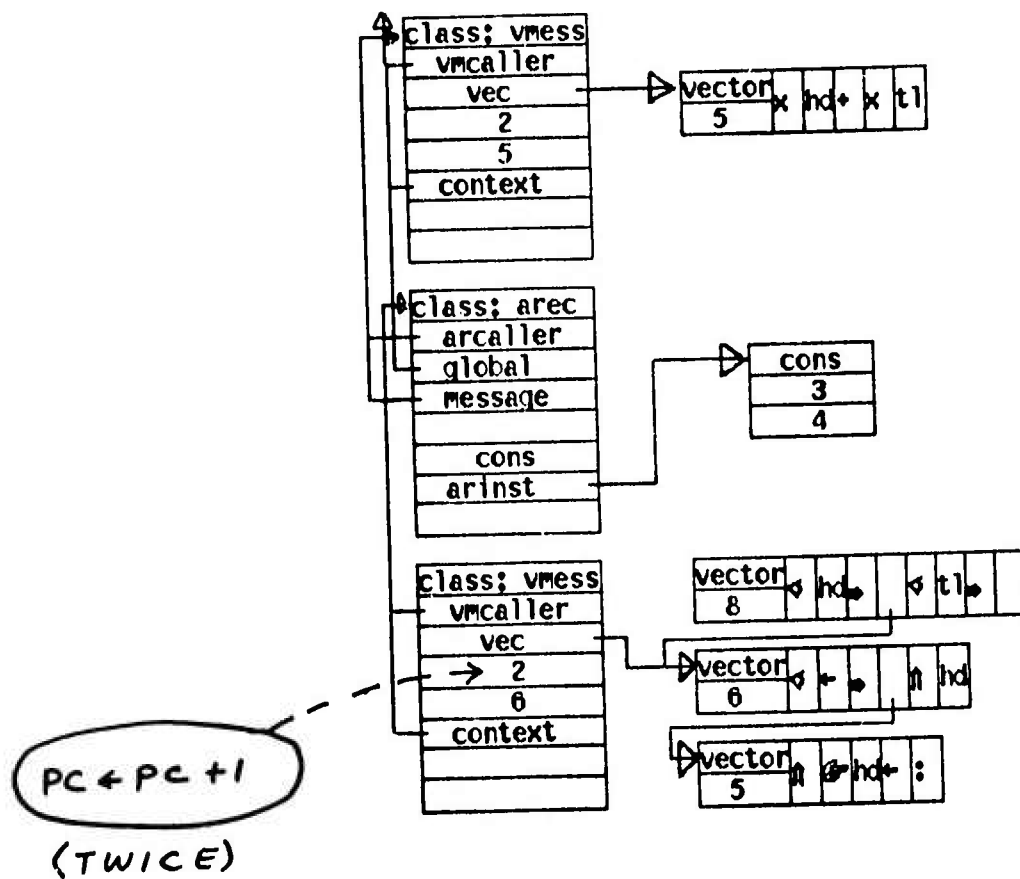
The message is scanned for the symbol "←", and not found
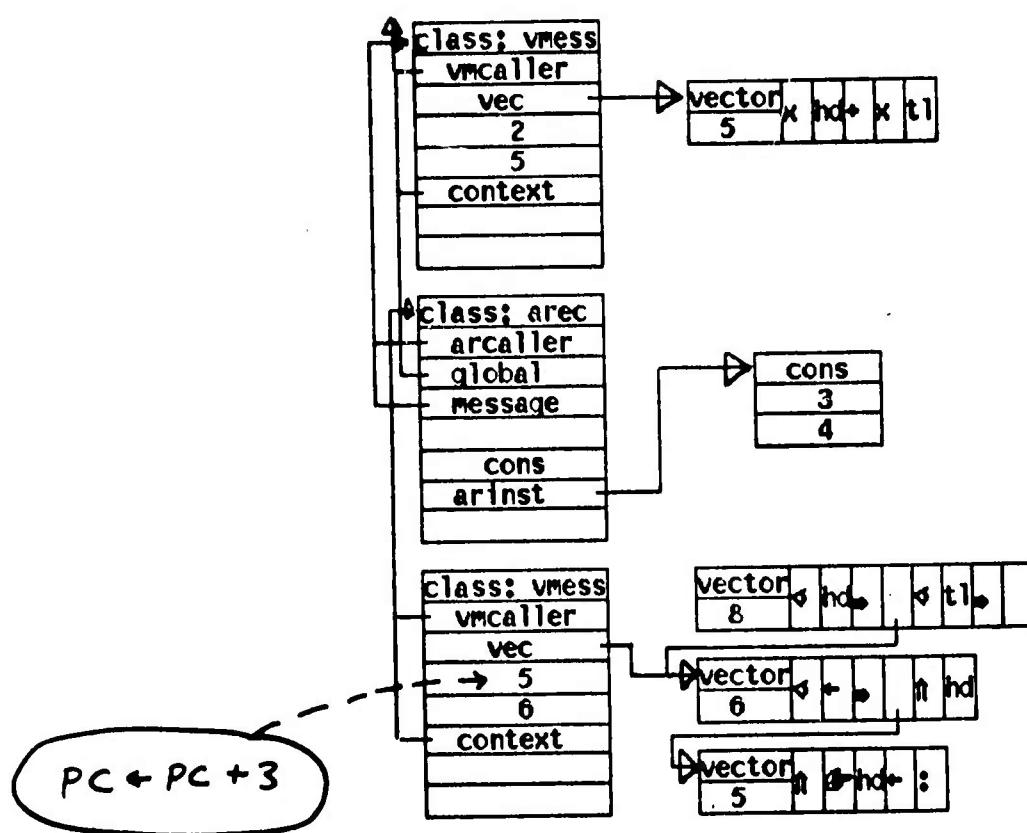
The return up-arrow is evaluated, which instantiates another AREC, etc.

PART III

The Once and Future PYG

Chapter 7

Conclusions and Suggestions for the Future

Summary

PYGMALION is useful today for the simulation of algorithms. It actively assists in the creation of programs, but current efficiency is not high enough for production software.

Part I of this paper enumerated capabilities that belong in an articulate medium of communication. Part II described ways in which the PYGMALION system incorporates many of these capabilities in a computer interface. This third part summarizes the results, presents some qualifications and suggests future work.

Section A -- Areas of Success

The characteristics of PYGMALION that seem to provide the greatest assistance to problem solving and creative programming are:

(1) The display screen assists thinking by relieving the load on the short term memory. This reduces mental fatigue and provides a convenient medium for visualizing and transforming concepts.

(2) Programming is in a "language of thought". The programmer can design many images in the form that is closest to his mental representations.

(3) The display serves as dynamic scratch paper. Ideas can be sketched out and

developed step by step. Programs need not be completely specified before
they can be executed.

(4) Icon: invocation of functions improves communication between routines.
The argument and value icons are "templates" or schemata of their
interfaces. They can be examined and instantiated visually by the
programmer. This reduces such errors as calling a function with the wrong
number or type of arguments.

(5) Aesthetic semantics are concrete; mechanical semantics are abstract. The
user has to deal only with the aesthetic attributes of images; the mechanical
aspects are managed automatically. The system maps particulars into
generals.

(6) PYGMALION provides a laboratory-like environment for experimenting
with ideas. Tests can be conducted, the results observed, changes made, and
new tests conducted, completely interactively.

Section B -- Efficiency and Other Problems

The current PYGMALION system is only a prototype implementation. It is not a
production programming language since it suffers from several defects.

Efficiency

Systems employing interpreters are usually versatile but slow. In that case,
PYGMALION must be extremelty versatile, because PYGMALION operations are *triply*
interpreted! Each level of interpretation slows the system down a bit more. The operations
in the menu are

(1) interpreted in Smalltalk, which is

(2) interpreted in Nova code, which is

(3) interpreted in microcode on the mini-computer, since it is not a Nova!

Because efficiency was not an immediate design goal, this was acceptable initially. The
correct way to remedy the situation is to build a "PYGMALION machine" in microcode, or
at least in machine code. This task is made easier by the simplicity of PYGMALION
operations. For example,

$$(FETCH <icon> VALUE)$$

$$(STORE <icon> NAME)$$

are primitive operations that could readily be written in machine code.

Internal Representation

Currently all icons are represented as instances of the Smalltalk class ICON. Thus even if an icon describes a single machine bit, it occupies an elaborate structure in memory. A mapping needs to be constructed between the display level structure of icons and their internal representations.

current:

needed:
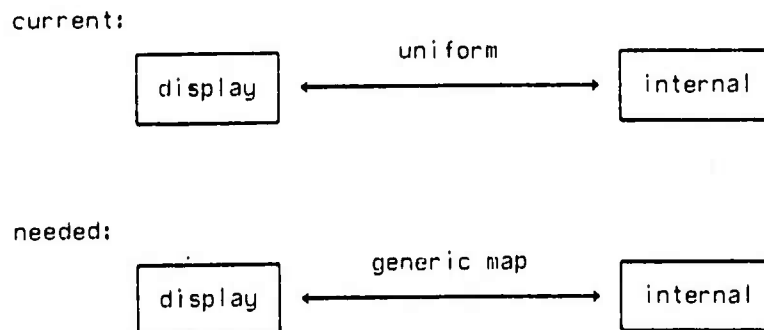
Operations now treat icons in a type-independent, uniform fashion. For example, the "fetch value" button works the same regardless of the icon designated (except for the "smalltalk" icon). There should be a provision for specifying an icon's internal representation in terms of structural primitives. Then operations on icons could be generic on the type of representation; e.g. "fetch value" could now have several generic cases:

Icon stored as	Implementation
bit	"load byte" instruction
word	"load word" instruction
part of structure	load ith element of structure
other	...

Section C -- For the Future

The preceeding section suggested improvements in efficiency as ways to extend PYGMALION. However there are more important directions which future research can take. PYGMALION has merely hinted at the potential of visual communication between man and machine.

Graphic Vocabulary

PYGMALION places minimum emphasis on pre-defined representations and structures. It is actually a non-language. PYGMALION is an *environment for working calculations*, a medium of communication with a machine. The stress is on *permitting the iconographer to design his own images*. But the complete absence of dogma is anarchy. Pre-defined schemata are helpful in the initial attack on a problem. An articulate *graphic vocabulary* can be of significant assistance to the designer. Several fascinating books have been written on expressing concepts visually. The most systematic in developing a graphic vocabulary is Bowman's *Graphic Communication*. He explores the potential of graphic expression, developing techniques for many areas. Below are some of the concepts for which his notation seems particularly appropriate.

PHYSICAL

FORMALIZED

SURFACE

ENVIRONMENTAL

TRANSPARENT

COMPOSITIONAL

Appearance [Bowman[1]]

REGULAR

DIVIDED

EMPHASIZED

COMPOUND

INTERRUPTED

QUANTIFIED

PHASED

System [Bowman^2]

ELEMENTARY

SYMBOLIC

RELATIONAL

CONTINUOUS

EMPHASIZED

COMPOUND

GROUPED

SUBDIVIDED

INTERNAL

DECLINING

QUANTIFIED

STRUCTURAL

SPATIAL

Organization [Bowman[3]]
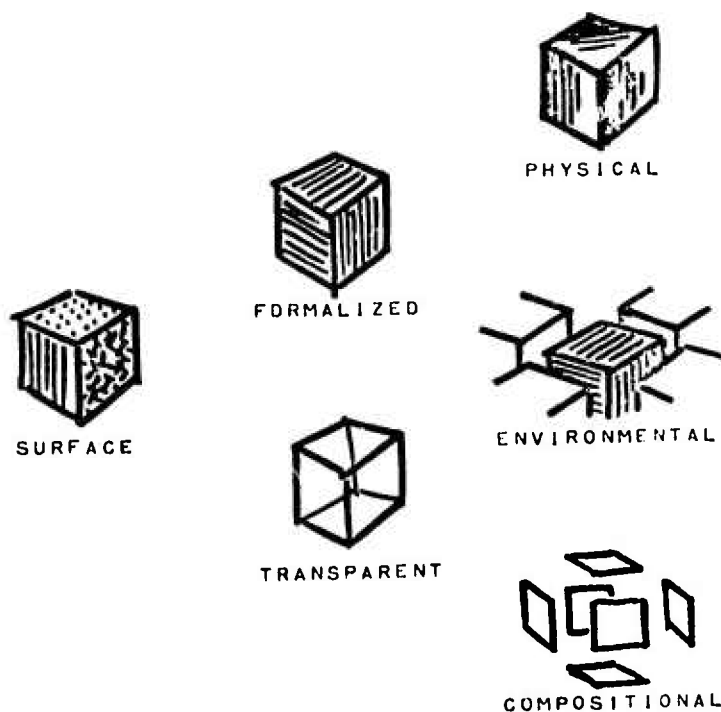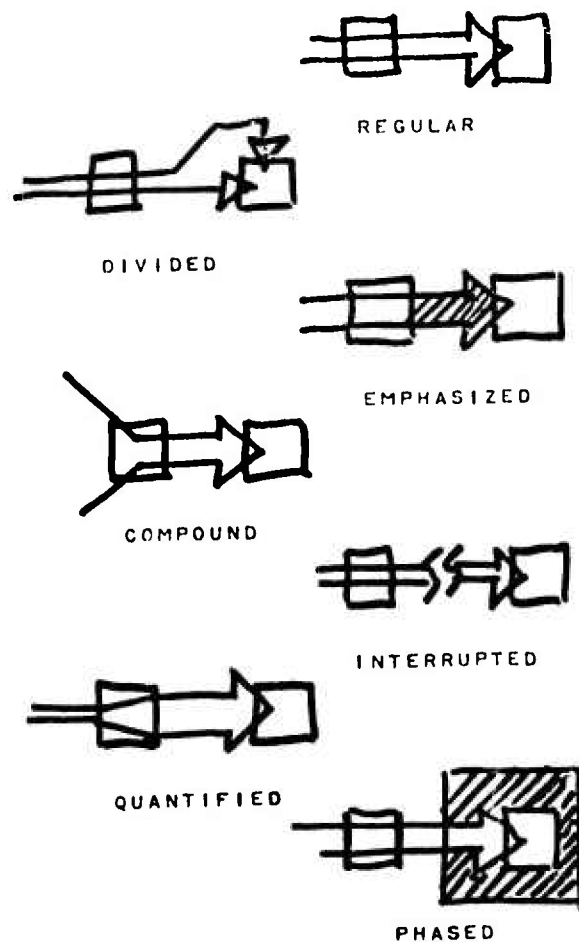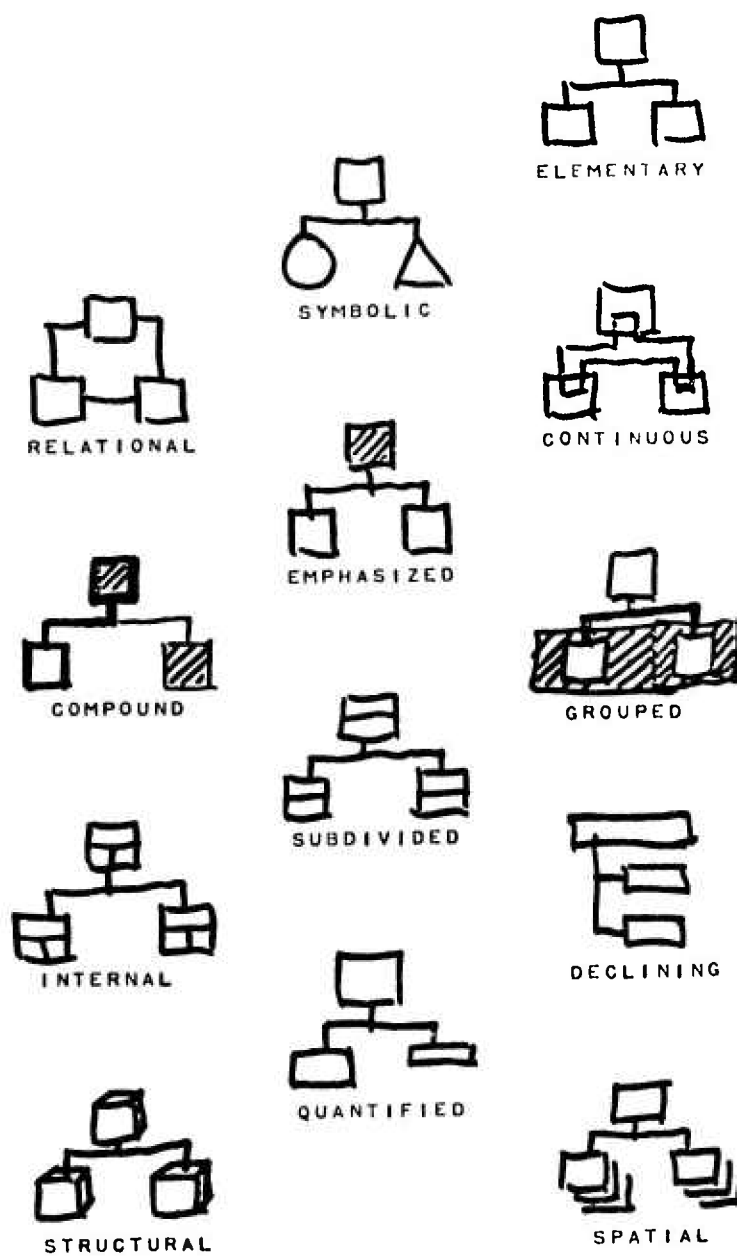
It seems clear that a good visual communications medium should find some common ground between a predefined graphic vocabulary and freedom of graphic expression. PYGMALION has concentrated on the freedom end of the scale, primarily because traditional languages have concentrated on dogma. A good project would be to systematically design sets of icons tailored to stimulating creativity in specific domains.

Dynamic Operations

A greater use should be made of dynamic operations. By this I mean operations whose execution inherently involve motion. My favorite example is Arnheim's dynamic geometry. The motivation arises from the difficulty of visualizing "ideal" concepts (in the Platonic sense). Suppose we want to prove a theorem about triangles. Aristotle defines the task: "Though we do not for the purpose of the proof make any use of the fact that the quantity in the triangle is determinate, we nevertheless draw it determinate in quantity."[4] This is undeniably difficult. We require a triangle which does not embody any unique characteristic (such as having a right angle) but which is nevertheless concrete in shape. Much of the difficulty children have in education can be traced to the problem of differentiating the generic from the specific.
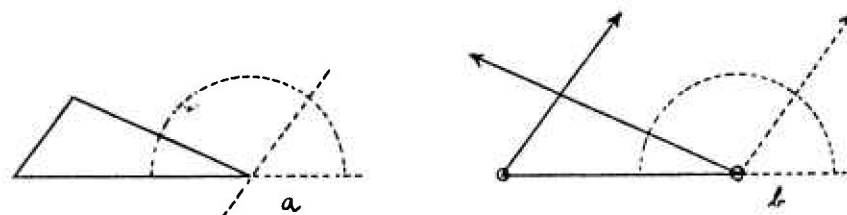
Euclid's Thirty-second Proposition [Arnheim[5]]

Something better is needed for the sake of true understanding. If I demonstrate Euclid's thirty-second proposition by drawing a parallel to one of the edges of a triangle (Figure a) and by showing that the equivalent of the three angles adds up to half a circle, I can point out, with Berkeley, that the size of the angles need not be referred to, and I thereby prove that the proposition holds for any triangle. To prove the correctness of a proposition is valuable practically; but what counts for thinking is that the range of the proposition be made evident. The figure I used shows, in fact, that the three angles add up to 180 degrees in this case. But *in order to truly understand that this is so in all*

triangles and for what reason, I must go beyond the particular figure to a full range of triangles. If I think of two of the edges as hands of indefinite length, hinged in such a way that they can sweep independently across the entire half circle (Figure b) I see that, whatever their positions, they will form three sectors adding up to the same semicircular whole. When one angle grows, its neighbor declines automatically by the same amount. In this way, the size of the angles is not ignored -- as Berkeley bids us to do, at the price of losing our visual grip on the situation -- but perceived in the sweep of its total range. *A static concept has been replaced with a dynamic one. Generality intended is now represented by generality perceived.*[Arnheim[6]] [emphasis mine]

Ivan Sutherland began the investigation of constraints on movable objects. For example, the sides of the triangles above must remain connected. Animators have also considered the problem. But much work remains to be done before a general specification of iconic movement is developed.

Section D -- Epilogue

If I had to sum up the lessons of this research in two sentences, I would say:

(1) Visual communication with a computer is a productive metaphor for assisting the thinking and learning processes of human beings.

(2) Research into articulate graphical communication has just begun to scratch the surface, but the paradigm of operations possessing both aesthetic and mechanical semantics is a powerful approach.

Appendix

The complete listing of the SMALLTALK code for PYGMALION follows. At the time of this writing, PYGMALION is the largest existing SMALLTALK program.

```
(GET obset ☞ DO)[4][14][6] ← ☞
  (☞vec ← vec[ 1 to ☞size ← size + 10].
   vec[☞end ← end + 1] ← input).

to icon x y z : name ix iwd iy iht frame CALLER value shape body runcode displayed fetcher sto
**rer container
  (◁'s ⇒
    (.☞x is vector ⇒
      (☞x ← x eval.
        ◁← ⇒ (⇑x ← :) ⇑x eval)
      ◁← ⇒ (⇑x ← :) ⇑x eval)
    ◁has ⇒
    (:x. :y.
     displayed is false ⇒ (⇑false)
     x < ix ⇒ (⇑false)
     y < iy ⇒ (⇑false)
     x > ix + iwd ⇒ (⇑false)
     y > iy + iht ⇒ (⇑false)
     ⇑true)
    ◁run ⇒ (SELF has mx my ⇒ (runcode eval))
    ◁display ⇒
    (◁name ⇒
      ((◁← ⇒ (☞name ← stringify :)).
       displaymode ⇒
        (SELF display erase.
         write SELF name.
         ☞displayed ← ☞name))
     ◁value ⇒
      ((◁← ⇒ (:value)).
       displaymode ⇒
        (☞displayed ← ☞value.
         value is iconstructure ⇒ (value map(xi display name))
         write SELF value))
     ◁shape ⇒
      ((◁← ⇒ (:shape)).
       displaymode ⇒
        ((null shape ⇒ (SELF display name)
          shape eval).
         ☞displayed ← ☞shape))
     ◁body ⇒ (◁← ⇒ (⇑:body) ⇑body)
     ◁erase ⇒
      (displaymode ⇒
        (eq displayed ☞name ⇒
          (frame fclear.
           frame frame 0.
           ☞displayed ← false)
         eq displayed ☞value ⇒
          ((value is iconstructure ⇒ (value map(xi display erase))).
           frame fclear.
           frame frame 0.
           ☞displayed ← false)
         eq displayed ☞shape ⇒
          ((null shape ⇒ () white shape eval black).
           ☞displayed ← ☞value.
           SELF display erase)))
     ◁delete ⇒
      ((displayed ⇒ (SELF display erase)).
       container's value delete CALLER.
       SELF map ☞('s (☞name ← ☞frame ← ☞CALLER ← ☞value ←
                  ☞shape ← ☞body ← ☞runcode ← ☞fetcher ←
                  ☞storer ← ☞container ← nil)))
     ◁← ⇒ (⇑:displayed) ⇑displayed)
    ◁fetch ⇒ (:x. ⇑fetcher eval)
    ◁store ⇒ (:x. :y. ⇑storer eval)
    ◁eval ⇒ (⇑body eval World)
    ◁map ⇒
    (:x.
     (value is iconstructure ⇒ (value map(xi map x))).
```

```
       apply SELF to x)
   ◁copy →
   (𝒢²x ← icon name ix iwd iy iht nil quick.
    x's runcode ← runcode.
    x's displayed ← displayed.
    x's fetcher ← fetcher.
    x's storer ← storer.
    ◁quick → (⇑x)
    x's value ← (value is iconstructure → (value copy) value).
    x's shape ← (shape is iconcontext → (shape copy) shape).
    x's body  ← (body is iconcontext → (body copy) body).
    ◁containerless → (⇑x)
    x's container ← container.
    ⇑x)
   ◁change →
   ((◁position →
      ((◁to →
         (𝒢²x ← - ix - :ix.
          𝒢²y ← - iy - :iy)
        𝒢²ix ← ix + :x.
        𝒢²iy ← iy + :y).
       value is iconstructure →
        (value map (xi change position x y)))
      ◁size →
      (𝒢²iwd ← max 16 ((:x * iwd) / 100) \ 16.
       𝒢²iht ← max 16 ((:y * iht) / 100) \ 16.
        (value is iconstructure →
         (value map (xi change size x y))).
       𝒢²ix ← (basex + (x * ix - basex) / 100) \ 16.
       𝒢²iy ← (basey + (y * iy - basey) / 100) \ 16)).
     frame param
     (𝒢²winx ← 𝒢²frmx ← ix.
      𝒢²winy ← 𝒢²frmy ← iy.
      𝒢²winwd ← 𝒢²frmwd ← iwd.
      𝒢²winht ← 𝒢²frmht ← iht))
   isnew →
   (𝒢²name ← stringify :.
    𝒢²frame ← dispframe :ix :iwd :iy :iht :.
    𝒢²CALLER ← SELF.
    ◁quick → ()
    (displaymode →
     (frame frame - 1.
      frame fclear.
      write SELF name.
      𝒢²displayed ← 𝒢²name)).
    𝒢²runcode ← iconrun.
    𝒢²fetcher ← iconfetch.
    𝒢²storer ← iconstore.
    change container absolute SELF ix iy.
    ⇑SELF)
   ◁print → (disp ← '<icon ' name print disp ← '>'))

to iconcontext j oldx oldy oldWorld : i x y quick code World
   (◁eval →
   ((World's value is vector →
      (𝒢²oldx ← World's value.
       World's value ← iconstructure 10.
       for j ← 1 to oldx length - 1 do
        (World's value push Icontable[oldx[j]]))).
    geticon i ← CALLER.
    quick → (code eval)
    :oldWorld.
    𝒢²oldx ← CALLER's ix.
    𝒢²oldy ← CALLER's iy.
    showicon CALLER x y World.
    code eval.
    SELF delete.
    showicon CALLER oldx oldy oldWorld)
   isnew →
```

pygmalion

```
(☞World ← Icontable['world'] copy quick.
 ◁initially ⇒
  (:i. :x. :y. :quick.
   ☞code ←
     (:j is supervector ⇒ (j) supervector initially j).
    World's value ←
     (:j is vector ⇒ (j) iconstructure initially j vector))
 ☞quick ← (◁quick ⇒ (true) false).
 :oldWorld. :i. :code.
 World's value ← iconstructure initially oldWorld's value vector.
 ☞x ← i's ix.
 ☞y ← i's iy.
 ☞i ← geticon x y index)
◁'s ⇒
 (:☞j.
  ◁← ⇒ (⇑j ← :) ⇑j eval)
◁copy ⇒ (⇑iconcontext initially i x y quick code World's value)
◁is ⇒ (ISIT eval)
◁print ⇒
 (disp ← 'iconcontext initially ☞'.
  i print sp.
  x print sp.
  y print sp.
  quick print.
  disp ← ' ☞' code print.
  disp ← ' ☞( '.
  World's value map(xi's name print sp).
  disp ← ')')
◁delete ⇒
 (☞oldx ← World's value.
  for j ← oldx length to 6 by - 1 do
   (eq oldx[j] CALLER ⇒ () oldx[j] display delete)))

to iconout i j x v c
 (:i.
  disp ← '☞iiii ← icon '.
  i's name print.
  disp ← ' basex+'.
  (i's ix - basex) print.
  sp i's iwd print.
  disp ← ' basey+'.
  (i's iy - basey) print.
  sp i's iht print.
  (null i's frame param(buf) ⇒ (disp ← ' nil.' cr cr)
   sp i's frame print disp ← '.' cr cr).
  (☞c ← i's container.
   c has i's ix i's iy ⇒ ()
   disp ← 'change container absolute iiii basex+'.
   (c's ix - basex) print.
   disp ← ' basey+'.
   (c's iy - basey) print.
   disp ← '.' cr cr).
  ☞v ← [☞shape nil ☞body nil ☞runcode iconrun ☞fetcher iconfetch ☞storer iconstore ☞value ni
**1].
  for j ← 1 to v length - 1 by 2 do
   (☞x ← i's (v[j]).
    eq x v[j+1] ⇒ ()
    x is iconstructure ⇒ (x map (iconout xi))
    disp ← 'iiii's ' v[j] print disp ← ' ← '.
    x is atom or x is vector ⇒
     (disp ← '☞' x print disp ← '.' cr cr)
    x print disp ← '.' cr cr).
   Icontable[i's name] ⇒
    (Icontable[i's name] delete))

to iconstructure ii xx : vv
 (◁is ⇒ (ISIT eval)
  ◁copy ⇒
   (☞xx ← vv vector.
```

```
        for ii ← 1 to xx length - 1 do
          (xx[ii] ← xx[ii] copy).
        ⇑iconstructure initially xx)
      isnew →
        (◁initially → (ⓖ'vv ← supervector initially :)
        ⓖ'vv ← supervector :)
      ◁print →
        (disp ← 'iconstructure initially ['.
         vv map(sp xi print).
         disp ← ' ]')
      eq vv ⓖ'xx ← apply vv → () ⇑xx)

to abort (disp ← ' ...aborted')

to and (⇑:)

to announce x
  (:x.
   within dispframe 192 320 16 16 string 40
     (disp clear.
      disp ← x.
      disp ← '...'))

to blink
  (disp ← 20.
   do 10().
   disp ← 8.
   do 10())

to box x y wd ht
  (penup. goto :x :y. pendn. up. :wd. :ht.
   do 2
     (right 90. go wd.
      right 90. go ht))

to change i j x y hasex basey
  (◁position →
     (ⓖ'i ← geticon :ⓖ.
      i display erase.
      i's container's value delete i.
      move i to :x :y)
   ◁size →
     (ⓖ'i ← geticon :ⓖ.
      i's (ⓖ'basex ← ix. ⓖ'basey ← iy).
      i display erase.
      i change size :x :y.
      i display shape)
   ◁container →
     ((◁absolute →
        (ⓖ'i ← :.
         ⓖ'j ← geticon :x+1 :y+1)
      ⓖ'i ← geticon :ⓖ.
      ⓖ'j ← geticon :ⓖ).
      ⓖ'x ← i's container.
      i's container ← j.
      (null x → ()
      x's value is iconstructure → (x's value delete i)).
      (eq j's displayed ⓖ'name → (j's displayed ← ⓖ'value)).
      eq i j → ()
      j's value is iconstructure → (j's value push i)
      eq j's value nil →
        (j's value ← iconstructure 2.
         j's value push i)
      sorry 'container is not an iconic structure: ' + stringify j's value.
      SELF display erase))

to constant
  (Mouse store ⓖ'value :)
```

```
to copy i x y
   (move geticon :☞i copy containerless to :x :y)

to create x wd y ht
   (icon 'icon' :x :wd :y :ht nil)

to delete i
   (☞i ← (◁icon ⇒ (:) geticon :☞).
    i store ☞display ☞delete)

to disk fil basex basey baseicon iiii : : showev
   (◁'s ⇒ (⇑:☞x eval)
    :fil.
    ◁fetch ⇒
     (:basex. :basey.
      filin fil.
      baseicon display shape.
      Icontable[ baseicon's name] ← baseicon copy containerless)
    ◁store ⇒
     (☞baseicon ← geticon :☞.
      ☞basex ← baseicon's ix.
      ☞basey ← baseicon's iy.
      filout fil ☞(baseicon)))

to drawline x y
   (goto :x + ix :y + iy)

to Eval
   ((geticon :☞) eval)

to extend x attrib
   (:☞attrib.
    ☞x ← CALLER's (attrib).
    cr disp ← 'need more instructions for ' + CALLER's name.
    (x is iconcontext ⇒ (☞x ← x's code)).
    (eq Remember's value x ⇒ (remember resume)
     remember start with x).
    repeat (eq Remember's value x ⇒ (World run) done))

to fetch i
   (☞i ← geticon :☞.
    Mouse store ☞value i fetch :☞)

to getbutton m n
   (☞n ← 0.
    repeat
     (0 = ☞m ← mouse 7 ⇒ (black ⇑ n)
      ☞mousex ← mouse 8.
      ☞mousey ← mouse 9.
      n = m ⇒ ()
      ☞n ← m.
      (m > 3 ⇒
       (☞m ← m - 4.
        Mouse's (black box ix iy + 12 iwd - 1 17))
       Mouse's (white box ix iy + 12 iwd - 1 17)).
      (m > 1 ⇒
       (☞m ← m - 2.
        Mouse's (black box ix iy + 68 iwd - 1 17))
       Mouse's (white box ix iy + 68 iwd - 1 17)).
      (m > 0 ⇒
       (Mouse's (black box ix iy + 40 iwd - 1 17))
       Mouse's (white box ix iy + 40 iwd - 1 17))))

to geticon x y v i
   (:x is vector ⇒
    (☞i ← World.
     ◁← ⇒
      (:v.
       for y ← 1 to x length - 2 do (☞i ← i's value[x[y]]).
```

```
        ↑i's value[x[x length - 1]] ← v)
      for y ← 1 to x length - 1 do (☞i ← i's value[x[y]]).
      ↑i)
   :y.
   ◁top →
    (◁index → (↑[World's value map until (xi has x y) index])
     ↑World's value map until(xi has x y))
   ◁index →
    (☞v ← supervector 5.
     getindex x y World's value.
     ↑v vector)
   ☞i ← get1 x y World's value → (↑i) ↑World)

to getindex x y z i xi
  (:x. :y.
   for i ← 1 to :z length do
    (☞xi ← z[i].
     xi's displayed is false → (↑false)
     v ← i.
     (xi's value is iconstructure →
       (getindex x y xi's value → (↑true))).
     xi has x y → (↑true) v pop).
    ↑false)

to get1 x y z i j xi
  (:x. :y.
   for i ← 1 to :z length do
    (☞xi ← z[i].
     xi's displayed is false → (↑false)
     (xi's value is iconstructure →
       (☞j ← get1 x y xi's value → (↑j))).
     xi has x y → (↑xi)).
    ↑false)

to IF i x y
  (Mouse store ☞value 'if'.
   ☞i ← memory fetch :x :y.
   :☞x.
   :☞y.
   i's value[2]'s body ← x.
   i's value[3]'s body ← y.
   ↑i)

to init
  (PUT USER ☞DO ☞(World run).
   ☞disp ← Smalltalk's frame.
   disp clear.
   World's frame fclear.
   World display shape.)

to makeline x y
  (penup. goto :x :y.
   pendn. goto :x :y)

to max x y
  (:x > :y → (↑x) ↑y)

to memory i x y
  (◁fetch →
     (:x. :y.
      ☞i ← Icontable[Mouse's value] → (↑move i copy to x y)
      sorry '<icon ' + stringify Mouse's value + '> is not in memory')
   ◁store →
     (☞i ← geticon :☞.
      Icontable[i's name] ← i copy containerless))

to memq x v
  (:x.
   ↑0 < :v[1 to v length] find x)
```

```
to min x y
  (:x < :y → (↑x) ↑y)

to move i x y
  (:i.
   ◁to.
   change container absolute i :x :y.
   i change position to x y.
   i display shape.
   ↑i)

to nameout i
  (:i's
    (G⁻displayed ← G⁻name.
     frame frame - 1.
     frame show.
     within dispframe ix iwd iy - 16 16 string 100(disp ← name)))

to neg x
  (:x < 0 → (↑[x]) ↑x)

to opcode op i x y
  (G⁻i ← icon :G⁻op :x 176 :y 64 nil.
   icon " i's ix + 16 48 i's iy + 16 32 nil.
   icon " i's ix + 112 48 i's iy + 16 32 nil.
   i display ← G⁻shape.
   i's shape ← G⁻
    (SELF display name.
     value[1] display value.
     value[2] display value).
   G⁻x ← G⁻(SELF display value ←
    Mouse store G⁻value value[1]'s value ☺ value[2]'s value).
   x[15] ← op.
   i's body ← x[1 to x length].
   ↑i)

to opcode1 op i j x y
  (:G⁻op.
   G⁻i ← icon " :x 112 :y 64 nil.
   G⁻j ← geticon x y index.
   [G⁻text, op, j, 16, 24] eval.
   icon " i's ix + 48 48 i's iy + 16 32 nil.
   i display ← G⁻shape.
   G⁻x ← G⁻
    (text name ☺ 16 24.
     value[1] display value).
   x[3] ← j.
   i's shape ← x[1 to x length].
   G⁻x ← G⁻(SELF display value ←
    Mouse store G⁻value ☺ value[1]'s value).
   x[9] ← op.
   i's body ← x[1 to x length].
   ↑i)

to or
  (:. ↑true)

to plot
  ((null GET xplot G⁻DO → (filin 'xplot')).
   disp fclear.
   xplot :.
   disp show)

to refresh i
  (G⁻i ← geticon :G⁻.
   i display erase.
   i display shape)
```

pygmalion

```
to remember i x CALLER
  (◁start ⇒
   (announce 'remembering'.
    ☞remembermode ← true.
    ◁with ⇒
     (Remember's value ← :x.
      Remember's frame clear.
      within Remember's frame
        (for i ← 1 to x length - 1 do (cr x[i] print)))
    :☞x.
    Remember's value ← supervector initially[[☞extend x]].
    Remember's frame clear)
   ◁stop ⇒
    (remembermode is false ⇒ ()
     Remember's value[end] ← nil.
     announce 'stopped remembering'.
     ☞remembermode ← false.
     Remember's frame clear)
   ◁suspend ⇒ (remembermode ⇒
     (announce 'temporarily stopped remembering'.
      ☞remembermode ← false))
   ◁resume ⇒
    (Remember's value ⇒
     (announce 'remembering'.
      ☞remembermode ← true))
   :x.(remembermode ⇒
     (☞i ← Remember's value pop.
      Remember's value push x.
      Remember's value push i.
      within Remember's frame(cr x print))).
   ◁doit ⇒ (⇑x eval) ⇑x)

to REPEAT i x y
  (Mouse store ☞value 'repeat'.
   ☞i ← memory fetch :x :y.
   :☞x.
   i's value[1]'s body ← x.
   ⇑i)

to setmouse x y z
  (:x. :y. :z.
   within Mouse's frame
    (disp clear.
     cr disp ← x cr.
     cr disp ← y cr.
     cr disp ← z))

to Show i
  (☞i ← geticon :☞.
   i store ☞display :☞)

to showicon i x y
  (:i change position to :x :y.
   within dispframe 112 400 32 432 nil(disp fclear).
   :i's value map from 6(xi display shape))

to sorry
  (cr disp ← 'sorry, ' disp ← :.
   cr disp ← 'last operation aborted'.
   cr disp ← 'read-eval-print loop -- type done to proceed:'.
   ev)

to startline x y
  (penup. goto :x + ix :y + iy. pendn)

to store i x
  (☞i ← geticon :☞.
   i store :☞x Mouse's value)
```

```
to stringify x disp
  (:x is string → (↑x)
   ⇧disp ← superstring 10.
   x print.
   ↑disp string)

to superstring xx : vec end
  (◁← →
    (:xx is string →
      (⇧vec ← vec[1 to end + xx length].
       vec[end + 1 to ⇧end ← end + xx length] ← xx.
       ↑xx)
     vec length < ⇧end ← end + 1 →
      (⇧vec ← vec[1 to 2 * end - 1].
       ↑vec[end] ← xx)
     ↑vec[end] ← xx)
   ◁string → (↑vec[1 to end])
   isnew →
    (⇧vec ← string :.
     ⇧end ← 0))

to supervector ii xx xi : vec end
  (◁← →
    (vec length > ⇧end ← end + 1 → (↑vec[end] ← :)
     ⇧vec ← vec[1 to 2 * end].
     ↑vec[end] ← :)
   ◁map →
    (◁until →
      (:⇧xx.
       for ii ← 1 to end do
         (⇧xi ← vec[ii].
          xx eval → (◁index → (↑ii) ↑xi)).
       ◁index → (↑0) ↑false)
     ⇧xi ← (◁from → (:) 1).
     :⇧xx.
     for ii ← xi to end do
       (⇧xi ← vec[ii].
        xx eval))
   ◁push → (↑SELF ← :)
   ◁pop →
    (end = 0 → (↑nil)
     ⇧xx ← vec[end].
     vec[end] ← nil.
     ⇧end ← end - 1.
     ↑xx)
   ◁length → (↑end)
   ◁vector → (↑vec[1 to end + 1])
   ◁eval → (↑vec eval)
   ◁delete →
    (⇧ii ← vec[1 to end] find :.
     ii = 0 → ()
     vec[ii to end - 1] ← vec[ii + 1 to end].
     vec[end] ← nil.
     ⇧end ← end - 1)
   ◁is → (ISIT eval)
   ◁print → (vec[1 to end + 1] print)
   isnew →
    (◁initially →
      (⇧end ← :vec length - 1.
       ↑SELF)
     ⇧vec ← vector :.
     ⇧end ← 0.
     ↑SELF)
   eq vec ⇧xx ← apply vec → () ↑xx)

to table i x : names values
  (◁[ →
    (:x. ◁].
     ⇧i ← names map until (x = xi) index.
```

```
          ⊲ ← ⇒
           (i = 0 ⇒
              (names ← x.
                ↑values ← :)
              :x.
             ↑values[i] ← x)
           i = 0 ⇒ (↑false)
           ⊲delete ⇒
         (names delete names[i].
          values delete values[i])
           ↑values[i])
        isnew ⇒
         (𝒢²names ← supervector :i.
          𝒢²values ← supervector i)
        ⊲print ⇒ (names print))

to text s i x y
   (:s.
    𝒢²i ← geticon :𝒢².
    𝒢²x ← i's ix + :.
    𝒢²y ← i's iy + :.
    within dispframe x 256 y 256 string 100(disp ← s))

to waitmouse n x y z
   (setmouse :x :y :z.
    𝒢²topbutton ← 𝒢²midbutton ← 𝒢²botbutton ← false.
    repeat
      (0 = mouse 7 ⇒ ()
       𝒢²n ← getbutton.
       n = 3 ⇒ (Mouse's frame show. opplot) done).
     Mouse's frame clear.
     n = 4 ⇒ (𝒢²topbutton ← true)
     n = 1 ⇒ (eq y dashes ⇒ (abort) 𝒢²midbutton ← true)
     n = 2 ⇒ (eq z dashes ⇒ (abort) 𝒢²botbutton ← true)
     abort)

to within disp
   (:disp.
    ↑(:𝒢²) eval)

to write i v x wd y ht
   (:i.
    𝒢²v ← stringify :.
    𝒢²wd ← min i's iwd 8 * v length.
    𝒢²ht ← min i's iht 16 *
      (1 + (v length - 1) / i's iwd / 8).
    𝒢²x ← i's ix + 4 + (i's iwd - wd) / 2.
    𝒢²y ← i's iy + 2 + (i's iht - ht) / 2.
    i's frame frame - 1.
    i's frame fclear.
    within dispframe x wd y ht string 100(disp ← v))

to [ v i
   (𝒢²v ← vector 10.
    𝒢²i ← 0.
    repeat
      (⊲] ⇒ (⇑v[1 to i + 1])
       v[𝒢²i ← i + 1] ← :.
       ⊲..
       i : v length ⇒ (𝒢²v ← v[1 to 2 * v length])))

to ⟨ i
   (⊲icon ⇒ (𝒢²i ← Icontable[:i]. ⊲⟩. ⇑i))

to op1 op
   (:𝒢²op.
    waitmouse ' position' dashes dashes.
    topbutton ⇒
      (remember[𝒢²opcode1, op, mousex \ 16, mousey \ 16] doit.))
```

```
to op2 op
  (:☞op.
   waitmouse ' position' dashes dashes.
    topbutton ⇒
     (remember[☞opcode, op, mousex \ 16, mousey \ 16] doit.))

to opbody i
  (waitmouse ' define body' ' fetch body' ' store body'.
    topbutton ⇒
     (☞i ← geticon mousex mousey.
      sp disp ← i's name.
      i's body is iconcontext ⇒
        (remember start with i's body's code)
      remember start body.
      i's body ← iconcontext World i Remember's value.
      Icontable[i's name] ⇒
        (Icontable[i's name]'s body ← i's body copy)
      cr disp ← 'do you want ' + i's name + ' saved in memory? (y or n)'.
      memq read[1] ☞(y Y) ⇒
        (Icontable[i's name] ← i copy containerless))
    midbutton ⇒
     (remember[☞fetch, geticon mousex mousey index, ☞body] doit)
    botbutton ⇒
     (remember[☞store, geticon mousex mousey index, ☞body] doit))

to opchange i j
  (waitmouse ' change pos' ' change size' ' change cont'.
    topbutton ⇒
     (☞i ← geticon mousex mousey.
      eq i World ⇒ (sorry 'cant change position of world')
      ☞j ← geticon mousex mousey index.
      waitmouse ' upper left' dashes dashes.
      topbutton ⇒
        (remember[☞change, ☞position, j, mousex\16, mousey\16] doit))
    midbutton ⇒
     (☞i ← geticon mousex mousey.
      eq i World ⇒ (sorry 'cant change size of world')
      ☞j ← geticon mousex mousey index.
      waitmouse ' lower right' dashes dashes.
      topbutton ⇒
        (remember[☞change, ☞size, j,
           (100 * mousex - i's ix) / i's iwd,
           (100 * mousey - i's iy) / i's iht] doit))
    botbutton ⇒
     (☞i ← geticon mousex mousey index.
      waitmouse ' container' dashes dashes.
      topbutton ⇒
        (remember[☞change, ☞container, i, geticon mousex mousey index] doit)))

to opconstant
  (cr disp ← 'value? '.
   remember[☞constant, read eval] doit)

to opcopy i
  (waitmouse ' copy icon' dashes dashes.
    topbutton ⇒
     (☞i ← geticon mousex mousey index.
      waitmouse ' position' dashes dashes.
      topbutton ⇒
        (remember[☞copy, i, mousex\16, mousey\16] doit)))

to opcreate n started x wd y ht
  (☞started ← false.
   setmouse ' set point' ' del point' '    quit'.
   repeat
    (☞n ← getbutton.
     n = 4 ⇒
       (Mouse's (white box ix iy + 12 iwd - 1 17 black).
```

```
              started is false →
                (G⃗x ← mousex \ 16.
                 G⃗y ← mousey \ 16.
                 G⃗wd ← G⃗ht ← 0.
                 G⃗started ← true).
              mem 65 ← 1 'XOR ink'.
              box x y wd ht.
              mem 65 ← 0.
              G⃗wd ← (mousex \ 16) - x.
              G⃗ht ← (mousey \ 16) - y.
              (wd < 0 →
                (G⃗wd ← - wd.
                 G⃗x ← mousex \ 16)).
              (ht < 0 →
                (G⃗ht ← - ht.
                 G⃗y ← mousey \ 16)).
              remember[G⃗create x wd y ht] doit.
              G⃗started ← false)
          n = 1 →
            (Mouse's (white box ix iy + 40 iwd - 1 17 black).
             started →
              (mem 65 ← 1.
               box x y wd ht.
               mem 65 ← 0.
               G⃗wd ← G⃗ht ← 0.
               G⃗started ← false))
          n = 2 →
            (done with Mouse's frame clear)
          started →
            (mem 65 ← 1.
             box x y wd ht.
             box x y G⃗wd ← (mx \ 16) - x G⃗ht ← (my \ 16) - y.
             mem 65 ← 0)))

to opdelete
  (waitmouse ' delete icon' dashes dashes.
   topbutton →
   (remember[G⃗delete, geticon mousex mousey irdex] doit))

to opdisk i j fil
  (waitmouse '  fetch' '  store' dashes.
   topbutton →
   (cr disp ← 'please type a file name: '.
    G⃗fil ← read eval.
    fil is string →
     (remember[G⃗disk, fil, G⃗fetch, mousex \ 16, mousey \ 16] doit)
    abort)
   midbutton →
   (G⃗i ← geticon mousex mousey.
    G⃗j ← geticon mousex mousey index.
     (i's name = 'icon' →
       (cr disp ← 'please type a name (a string) for the icon: '.
        i display name ← read eval)).
    cr disp ← 'please type a file name: '.
    G⃗fil ← read eval.
    fil is string →
     (sp disp ← i's name.
      remember [G⃗disk, fil, G⃗store, j] doit)
    abort))

to opdisplay
  (waitmouse ' on' '  off' dashes.
   topbutton → (remember G⃗(G⃗displaymode ← true) doit)
   midbutton → (remember G⃗(G⃗displaymode ← false) doit))

to opdraw i n started xstart ystart xstop ystop
  (G⃗started ← false.
   waitmouse ' relative to' dashes dashes.
   topbutton is false → ()
```

```
      ⌖i ← geticon mousex mousey.
      setmouse 'start line' 'stop line' '    quit'.
      repeat
        (⌖n ← getbutton.
         n = 4 ⇒
           (Mouse's (white box ix iy + 12 iwd - 1 17 black).
            started ⇒
              (remember[⌖drawline, xstop - i's ix, neg ystop - i's iy].
               ⌖xstart ← xstop.
               ⌖ystart ← ystop)
            ⌖xstart ← ⌖xstop ← mousex.
            ⌖ystart ← ⌖ystop ← mousey.
            remember[⌖startline, xstart - i's ix, neg ystart - i's iy].
            ⌖started ← true)
         n = 1 ⇒
           (Mouse's (white box ix iy + 40 iwd - 1 17 black).
            started ⇒
              (remember[⌖drawline, xstop - i's ix, neg ystop - i's iy].
               ⌖started ← false))
         n = 2 ⇒
           ((started ⇒
             (mem 65 ← 1 'XOR ink'.
              makeline xstart ystart xstop ystop.
              mem 65 ← 0)).
            done with Mouse's frame clear)
         started ⇒
           (mem 65 ← 1.
            makeline xstart ystart xstop ystop.
            makeline xstart ystart ⌖xstop ← mx ⌖ystop ← my.
            mem 65 ← 0)))

to opeval
  (waitmouse ' eval icon' dashes dashes.
   topbutton ⇒
   (remember[⌖Eval, geticon mousex mousey index] doit))

to opexit
  (PUT USER ⌖DO sysUSER.
   ⌖disp ← sysDISP.
   remember suspend.
   World's frame fclear.
   disp frame - 1.)

to opif
  (waitmouse '  position' dashes dashes.
   topbutton ⇒
   (remember[⌖IF, mousex \ 16, mousey \ 16,
      supervector initially ⌖((extend body)),
      supervector initially ⌖((extend body))] doit))

to opmemory i j
  (waitmouse '   fetch' '   store' ' fetch mouse'.
   topbutton ⇒
   (cr disp ← 'please type a name (a string) for the icon: '.
    remember[⌖constant, read eval] doit.
    remember[⌖memory, ⌖fetch, mousex \ 16, mousey \ 16] doit)
   midbutton ⇒
   (⌖i ← geticon mousex mousey.
    ⌖j ← geticon mousex mousey index.
     (i's name = 'icon' ⇒
       (cr disp ← 'please type a name (a string) for the icon: '.
        i store ⌖name read eval)).
     (Icontable[i's name] ⇒
       (cr i print disp ← ' is already in memory'.
        cr disp ← 'type y to replace: '.
        memq read[1] ⌖(y Y) ⇒ () ⇑abort)).
    sp disp ← i's name.
    remember [⌖memory, ⌖store, j] doit)
   botbutton ⇒
```

```
                (remember[G²memory, G²fetch, mousex \ 16, mousey \ 16] doit))

to opname (repeat
    (waitmouse ' fetch name' ' store name' '   quit'.
     topbutton →
      (remember[G²fetch, geticon mousex mousey index, G²name] doit)
     midbutton →
      (remember[G²store, geticon mousex mousey index, G²name] doit)
     botbutton → (done)))

to opnext ()

to opplot fil
   (er disp ← 'please type a file name: '.
    G²fil ← read eval.
    fil is string → (remember[G²plot, fil] doit) abort)

to oprefresh (repeat
    (waitmouse 'refresh icon' dashes '   quit'.
     topbutton →
      (remember[G²refresh, geticon mousex mousey index] doit)
     botbutton → (done)))

to opremember
   (waitmouse '   stop' '  suspend' '  resume'.
    topbutton → (remember stop)
    midbutton → (remember suspend)
    botbutton → (remember resume))

to oprepeat
   (waitmouse '  position' dashes dashes.
    topbutton →
     (remember[G²REPEAT, mousex \ 16, monsey \ 16,
        supervector initially G²((extend body))] doit))

to opshape i
   (waitmouse 'define shape' ' fetch shape' ' store shape'.
    topbutton →
    (G²i ← geticon mousex mousey.
     sp disp ← i's name.
     i's shape is iconcontext →
       (remember start with i's shape's code)
     remember start shape.
     i's shape ← iconcontext quick World i Remember's value.
     Icontable[ i's name] →
       (Icontable[i's name]'s shape ← i's shape copy))
    midbutton →
     (remember[G²fetch, geticon mousex mousey index, G²shape] doit)
    botbutton →
     (remember[G²store, geticon mousex mousey index, G²shape] doit))

to opshow i j
   (waitmouse ' show name' ' show value' ' show shape'.
    topbutton →
    (G²i ← geticon mousex mousey.
     G²j ← geticon mousex mousey index.
      (eq i display G²name →
        (eq i's container World → ()
         j[j length - 1] ← nil.
         G²j ← j[ 1 to j length - 1])).
      remember[G²Show, j, G²name] doit)
    midbutton →
     (remember[G²Show, geticon mousex mousey index, G²value] doit)
    botbutton →
     (G²j ← geticon mousex monsey index.
      remember[G²Show, j, G²shape] doit))

to optext i j s
   (waitmouse ' relative to' dashes dashes.
```

```
    topbutton →
    (℘i ← geticon mousex mousey.
     ℘j ← geticon mousex mousey index.
     cr disp ← 'please type the text (a string) to be displayed: '.
     ℘s ← stringify read eval.
     waitmouse ' position' dashes dashes.
     topbutton →
      (remember[℘text, s, j, mousex - i's ix, mousey - i's iy + 16] doit)))

to optrace
  (waitmouse ' trace icon' 'untrace icon' dashes.
   topbutton →
    (remember[℘trace, geticon mousex mousey index, ℘on] doit)
   midbutton →
    (remember[℘trace, geticon mousex mousey index, ℘off] doit))

to opvalue n
  (setmouse ' fetch value' ' store value' ' ----------'.
   repeat
    (℘n ← getbutton.
     n = 4 →
      (Mouse's frame clear.
       remember[℘fetch, geticon mousex mousey index, ℘value] doit.
       done)
     n = 1 →
      (Mouse's frame clear.
       remember[℘store, geticon mousex mousey index, ℘value] doit.
       done)
     CALLER has mx my → ()
     done with Mouse's frame clear))

to initall
  (disk's
    (to showev x
      (disp ← '℘baseicon ← '.
       iconout :x eval)).
    (menq ℘\ GET number ℘DO → ()
     addto number ℘(◁\ → (⇑:x * (SELF + x / 2) / x)).
     addto dispframe ℘(◁print → (buf print)).
     (GET dispframe ℘DO)[59][50] ← 0.
     (GET pshow ℘DO)[10][4] ← 6.
     (GET pshow ℘DO)[22][34] ← ℘(: ℘ # ⇑ [ ◁ → ▣ ).
     (GET file ℘DO)[75][24][4] ← ℘
      (dp0 evals filesopen map ℘
       (vec[i] evals ℘(℘dirinst ← nil))).
     ℘sysUSER ← GET USER ℘DO.
     ℘sysDISP ← disp).
   ℘dashes ← ' ----------'.
   ℘run ← ℘run.
   ℘Icontable ← table 10.
   Icontable['world'] ← nil.
   ℘iconrun ← ℘(opvalue).
   ℘iconfetch ← ℘([CALLER ℘'s x] eval).
   ℘iconstore ← ℘
    (eq x ℘display → ([CALLER ℘display y] eval)
     [CALLER ℘display x ℘← ℘℘ y] eval).
   ℘specialstore ← ℘(sorry 'cant change ' + stringify CALLER).
   ℘displaymode ← true.
   ℘remembermode ← false.
   initworld initmenu initmouse initremember initsmalltalk initicons.
   World's frame clear.
   disp frame -1.
   ℘initworld ← ℘initmenu ← ℘initmouse ← ℘initremember ← ℘initsmalltalk ← ℘initicons ← ℘inita
**ll ← nil)

to initicons i baseicon basex basey
  (℘basex ← 192.
   ℘basey ← 192.
```

Ⓖ'baseicon ← Ⓖ'i ← icon 'if' basex+0 192 basey+0 112 nil.
i's shape ← Ⓖ'(value[1] display name. value[2] display value. value[3] display value. startline
**66 18. drawline 124 18. drawline 114 10. drawline 114 26. drawline 121 18. startline 66 34.
**drawline 104 77. drawline 92 75. drawline 104 65. drawline 104 77).
i's body ← Ⓖ'(value [1]'s value → (value [2] eval) value [3] eval).

Ⓖ'i ← icon '?' basex+0 64 basey+0 32 ' '.
i's runcode ← Ⓖ'(opvalue).

Ⓖ'i ← icon 'true branch' basex+128 64 basey+0 32 ' '.
i's shape ← Ⓖ'(SELF display value).
i's value   true.

Ⓖ'i ← icon 'false branch' basex+96 64 basey+80 32 ' '.
i's shape ← Ⓖ'(SELF display value).
i's value ← false.

    Icontable['if'] ← baseicon copy containerless.
    baseicon display delete.

Ⓖ'baseicon ← Ⓖ'i ← icon 'repeat' basex+0 48 basey+0 32 nil.
i's shape ← Ⓖ'(CALLER display value. startline 23 33. drawline 23 59. drawline -31 60. drawline
**-31 (-38). drawline 23 (-38). drawline 23 (-2). drawline 13 (-13). drawline 33 (-13). drawli
**ne 23 (-2)).
i's value ← ' loop'.

    Icontable['repeat'] ← baseicon copy containerless.
    baseicon display delete)

to initmenu i
  (Ⓖ'Menu ← Icontable['menu'] ← icon 'menu' 16 80 32 647 string 300.
  Menu's shape ← Ⓖ'(nameout SELF).
  Menu's runcode ← Ⓖ'
    (setmouse '   doit' da   es dashes.
      repeat
        (getbutton = 4 →
          (Mouse's frame clear.
          Ⓖ'x ← (Menu's frame mfindt mousex mousey) / 2.
          0 < x < menuops length + 1 →
            (sp menunames[x] print.
            Ⓖ'operation ← menuops[x].
            eq operation Ⓖ'undefined →
              (done with disp ← ' undefined')
            World's (operation eval).
            disp ← ' ok'.
            done))
        SELF has mx my → ()
        done with Mouse's frame clear)).
  Menu's fetcher ← iconfetch.
  Menu's storer ← specialstore.
  Ⓖ'menunames ← supervector 50.
  Ⓖ'menuops ← supervector 50.
  within Menu's frame
    (supervector initially Ⓖ
      ((icons undefined create opcreate change opchange delete opdelete copy opcopy refresh
** oprefresh show opshow name opname value opvalue shape opshape body opbody)
        (opcodes undefined
        + (op2 +) - (op2 -) * (op2 *) / (op2 /)
        = (op2 =) < (op2 <) > (op2 >)
        and (op2 and) or (op2 or) not (op1 not))
      (control undefined if opif repeat oprepeat done opdone eval opeval return opreturn
**)
      (others undefined memory opmemory disk opdisk next opnext display opdisplay rememb
**er opremember draw opdraw text optext trace optrace constant opconstant plot opplot exit ope
**xit))
    map
      (menunames ← xi[1].
      menuops ← xi[2].
      cr xi[1] print cr.

```
                 for i ← 3 to xi length - 1 by 2 do
                   (mennames ← xi[i].
                    mennops ← xi[i + 1].
                    memq xi[i] ☞(+ * <) ⇒ (sp xi[i] print sp)
                    sp xi[i] print cr))))

to initmouse
   (☞Mouse ← lcontable['mouse'] ← icon 'mouse' 400 96 480 96 string 50.
    Mouse's shape ← ☞(nameout SELF).
    Mouse's runcode ← nil.
    Mouse's fetcher ← iconfetch.
    Mouse's storer ← ☞
     (eq x ☞display ⇒ (nil)
      within Mousevalue's frame
       (disp clear.
        Mouse's value ← y print)).
    ☞Mousevalue ← lcontable['mouse value'] ← icon 'mouse value' 112 272 480 32 string 100.
    Mousevalue's value ← supervector initially ☞(" " ").
    Mousevalue's shape ← Mouse's shape.
    Mousevalue's runcode ← nil.
    Mousevalue's fetcher ← iconfetch.
    Mousevalue's storer ← specialstore)

to initremember
   (☞Remember ← lcontable['remembered'] ← icon 'remembered' 112 272 544 32 string 100.
    Remember's value ← false.
    Remember's shape ← ☞(nameout SELF).
    Remember's runcode ← ☞
     (setmouse ' insert' ' delete' ' scroll'.
      repeat
       (☞x ← getbutton.
    x = 4 ⇒ ('...')
        x = 1 ⇒ ('...')
        x = 2 ⇒ ('...')
        SELF has mx my ⇒ ()
        done with Mouse's frame clear)).
    Remember's fetcher ← iconfetch.
    Remember's storer ← specialstore)

to initsmalltalk
   (☞Smalltalk ← lcontable['smalltalk'] ← icon 'smalltalk' 112 384 608 71 string 300.
    Smalltalk's shape ← ☞(nameout SELF).
    Smalltalk's body ← ☞
     (cr disp ← 'input? '.
      Mouse store ☞value read eval).
    Smalltalk's runcode ← ☞(repeat
       (kbck ⇒ (cr read eval print)
        SELF has mx my ⇒ (blink)
        done)).
    Smalltalk's fetcher ← ☞
     (cr x print disp ← '? '.
      read eval).
    Smalltalk's storer ← ☞(Mouse's value print))

to initworld
   (☞World ← lcontable['world'] ← icon 'world' 0 512 0 680 nil quick.
    World's value ← iconstructure 10.
    World's shape ← ☞(value map(xi display shape)).
    World's runcode ← ☞
     (Mouse's frame clear.
      value map(xi run)).
    World's fetcher ← iconfetch.
    World's storer ← specialstore.
    World's container ← World)

( initall )
```

Notes

Introduction
1. Encyclopaedia Britannica, Encyclopaedia Britannica Inc., 18, 1973, p.891.
2. Gombrich, p.95
3. Gombrich, pp.94-97 (reprinted by permission of Princeton University Press)
4. Wallace, p.116
5. Papert, unrecorded talk, but see Papert 1970, 1971(a), 1971(b), 1971(c), 1973 for ways to make the computer fun for children.
6. Arnheim, p.v
7. David Pye, in Hyman, p.61
8. Koestler, p.28
9. Gombrich, p.306

Chapter 1
1. Bowman, p.vii
2. Koestler, p.173
3. Quoted in Arnheim, p.228
4. Arnheim, p.227
5. Sloman, p.276
6. William A. Martin, p.78
7. Hadamard, p.83-5
8. Hadamard, p.142-3
9. Arnheim, p.v
10. Arnheim, p.277
11. Arnheim, pp.231-232
12. Moran, p.475
13. Bowman, p.16
14. Sloman, p.273
15. Hadamard, p.85
16. Sloman, p.275
17. Aristotle, *On Memory and Reminiscence*, p.449.
18. Arnheim, p.98
19. Arnheim, pp.103-104
20. Paraphrased in Arnheim, p.102
21. Julesz, p.xi
22. Julesz, p. 240
23. Arnheim, p.104
24. Arnheim, p.107
25. Gombrich, p.23
26. Colby, private communication
27. See, for example, Bowman (op.cit.) and *Learning to See*.

28. Arnheim, pp.125-126 (reprinted by permission of the University of California Press)
29. Cooper & Shepard, p.80
30. Gombrich, p.385
31. Gombrich, p.119
32. Webster's dictionary, p.50
33. Cooper & Shepard, p.85
34. Gombrich, p.395
35. Arnheim, p.277
36. Gombrich, p.370
37. Gombrich, p.393
38. Arnheim, p.108
39. Gombrich, p.x
40. Arnheim, p.108
41. Gombrich, p.226
42. Gombrich, p.248
43. Gombrich, pp.248-249
44. Arnheim, pp.136-138
45. Arnheim, p.151 (reprinted by permission of the University of California Press)
46. Koestler, p.170
47. Paraphrased in Arnheim, p.224
48. Arnheim, p.223
49. Arnheim, p.221
50. Arnheim, p.225
51. John Locke, *An Essay Concerning Human Understanding*, Introduction, sect.8.
52. Papert 1971(a), p.14
53. Coleman, pp.117-118
54. Coleman, p.125
55. Quoted in Coleman, p.132

Chapter 2
1. Mednick, p.221
2. Gombrich, p.376
3. Quoted in Koestler, pp.14-15
4. Koestler, p.16
5. Koestler, pp.27,45
6. Gombrich, p.94
7. Koestler, p.35
8. Koestler, p.35
9. Koestler, p.37
10. Koestler, p.39
11. Campbell, p.387
12. Gombrich, pp.271-272
13. Gombrich, p.83
14. Wertenbaker, p.74
15. Gombrich, pp.79-80
16. Gombrich, pp.80-81
17. Gombrich, p.80
18. Wilson, p.93
19. Wilson, p.106
20. Alan Kay, private note
21. Gombrich, p.81

22. Wertenbaker, p.162
23. Wertenbaker, p.162
24. Campbell, pp.387-388
25. Wilson, p.106
26. Krippner, p.42
27. Golann, p.550
28. Drake, p.89
29. Mednick, pp.221-222
30. Campbell, p.387
31. Mednick, p.222
32. Mednick, p. 222
33. Mednick, p.223
34. Mednick, p.224

Chapter 3
1. Brand, p.77
2. Brand, p.77
3. Term suggested by Alan Kay
4. W.R.Sutherland, p.59
5. W.R.Sutherland, p.89
6. Christensen 1966,1968,1971(a),1971(b) and Rovner 1969
7. Paraphrased from Baecker (1969)
8. *Saturday Evening Post, 173,* 21 (June 1900).
9. Hyman, p.62
10. Hyman, p.30
11. Hyman, pp.88,107,110
12. Hyman, p.7
13. Cartier-Bresson, 1963 (no page numbers)
14. Lincoln Kirstein in Cartier-Bresson, 1963 (no page numbers)
15. Moore (1964), p.170
16. Quoted in Koestler, p.170

Chapter 4
1. Bowman, p.1
2. Gombrich, p.vii
3. Webster, pp.412,415
4. Webster, p.412
5. Hyman, p.61
6. Hyman, pp.104-105
7. Hyman, p.61

Chapter 7
1. Bowman, p.62 (reprinted by permission of the author)
2. Bowman, p.124 (reprinted by permission of the author)
3. Bowman, p.94 (reprinted by permission of the author)
4. Quoted in Arnheim, p.180
5. Arnheim, p.180 (reprinted by permission of the University of California Press)
6. Arnheim, p.180

Bibliography

(Note: "IJCAI" = "International Joint Conference on Artificial Intelligence"; "ACM" = "Association of Computing Machinery")

Arnheim, Rudolf, *Visual Thinking*, (copyright (c) 1969 by the Regents of the University of California), University of California Press, Berkeley, 1971.

Baecker, Ronald, *Interactive Computer-Mediated Animation*, Ph.D. thesis, MIT Project MAC TR-61, Cambridge, Mass., 1969.

Balzer, Robert, *Automatic Programming*, Information Sciences Institute, USC Technical Memorandum, Los Angeles, 1972.

Bowman, William, *Graphic Communication*, John Wiley and Sons, New York, 1968.

Brand, Stewart, *II Cybernetic Frontiers*, Random House, New York, 1974.

Campbell, Donald, "Blind Variation and Selective Retention in Creative Thought as in Other Knowledge Processes", *Psychological Review*, 67, 6 (November 1960), pp.380-400.

Cartier-Bresson, Henri, *Photographs*, Grossman Publishers, New York, 1963.

Chase, William, (Ed.), *Visual Information Processing*, Academic Press, New York, 1973.

---, and Simon, Herbert, "The Mind's Eye in Chess" in Chase, 1973, pp.215-281.

Christensen, Carlos, "On the Implementation of AMBIT, A Language for Symbol Manipulation", *Comm. ACM*, 9, 8 (August 1966), pp.570-573.

---, "An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language" in *Interactive Systems for Experimental and Applied Mathematics*, M.Klerer and J.Reinfelds (Eds.), Academic Press, New York, 1968.

---, "An Introduction to AMBIT/L, a Diagrammatic Language for List Processing", *Proc. Second Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, 1971(a).

---, Wolfberg, Michael, and Fischer, Michael, *A Report on AMBIT/G*, Massachusetts Computer Associates, Wakefield, Mass., 1971(b).

Coleman, James, "Education in Modern Society" in *Computers, Communications and the Public Interest*, M.Greenberger (Ed.), Johns Hopkins Press, 1971.

Cooper, Lynn, and Shepard, Roger, "Chronometric Studies of the Rotation of Mental Images" in Chase, 1973, pp.75-176.

Dahl, Ole-Johan, and Nygaard, Kristen, "SIMULA -- an ALGOL-Based Simulation Language", *Comm. ACM*, 9, 9 (September 1966), pp.671-678.

---, and Myhrhaug, Bjorn, *SIMULA 67 Common Base Language*, Norwegian Computing Center, Forskningsveien 1b, Oslo, Norway, 1970.

Davis, M.R., and Ellis, T.O., "The RAND Tablet: A Man-Machine Graphical Communication Device", *Proc. AFIPS Fall Joint Computer Conference*, 1964, pp.325-331.

Dijkstra, E., "Notes on Structured Programming" in *Structured Programming*, O.Dahl, E.Dijkstra and C.Hoare, Academic Press, New York, 1972, pp.1-82.

Drake, Stillman, "Galileo's Discovery of the Law of Free Fall", *Scientific American*, 228, 5 (May 1973), pp.84-92.

Ellis, T.O., Heafner, J.F., and Sibley, W.L., *The GRAIL Project: An Experiment in Man-Machine Communications*, RAND Report RM-5999-ARPA, 1969.

Engelbart, D.E. *Augmenting Human Intellect: a Conceptual Framework*, Summary Report AD 289-565, SRI Project 3578, Stanford Research Institute, Menlo Park, Calif., 1962.

---, and English, W.K., "A Research Center for Augmenting Human Intellect", *Proc. AFIPS 1968 Fall Joint Computer Conference*, pp.395-410.

Gelernter, H., "Realization of a Geometry-Theorem Proving Machine" in *Computers and Thought*, E.Feigenbaum and J.Feldman (Eds.), McGraw-Hill Book Co., New York, 1963, pp.134-152.

Golann, Stuart, "Psychological Study of Creativity", *Psychological Bulletin*, 60, 6 (November 1963), pp.548-565.

Gombrich, E.H., *Art and Illusion*, No. 5 in the A.W.Mellon Lectures in the Fine Arts, Bollingen Series XXXV (copyright (c) 1960, 1961 and 1969 by The Trustees of the National Gallery of Art, Washington, D.C.), Princeton University Press, Princeton, New Jersey, 1960.

Hadamard, Jacques, *The Psychology of Invention in the Mathematical Field*, Dover Publications, New York, 1945.

Hall, Calvin, "What People Dream About", *Scientific American*, 184, 5 (May 1951), pp.60-63.

Hays, John, "On the Function of Visual Imagery in Elementary Mathematics" in Chase, 1973, pp.177-214.

Hyman, Anthony, *The Computer in Design*, Studio Vista, London, 1973.

Hewett, C., Bishop, P., and Steiger, R., "A Universal Modular *ACTOR* Formalism for Artificial Intelligence", *Proc. Third IJCAI*, Stanford, 1973, pp.235-245.

Jacks, Edwin, "A Laboratory for the Study of Graphical Man-Machine Communication", *Proc. AFIPS Fall Joint Computer Conference*, 1964, pp.343-350.

Julesz, Bela, *Foundations of Cyclopean Perception*, University of Chicago Press, Chicago, 1971.

Kammerahl, Hanna, *Picture Algorithm Language (PAL)*, NSF application, University of Illinois, Urbana, 1974.

Kay, Alan, *The Reactive Engine*, Ph.D. thesis, University of Utah, Salt Lake City, 1969.

---, "A Personal Computer for Children of All Ages", *Proc. ACM National Conference*, Boston, 1972(a).

---, "A Dynamic Medium for Creative Thought", *Proc. National Council of Teachers of English Conference*, Minneapolis, 1972(b).

---, *Personal Dynamic Media*, (internal document in press), Xerox Palo Alto Research Center, Palo Alto, Calif., 1975.

Koestler, Arthur, *The Act of Creation*, Dell Publishing Co., New York, 1964.

Krippner, Stanley, and Hughes, William, "Genius at ZZZZ Work ZZZZZZZ", *Psychology Today*, 4, 1 (June 1970), pp.40-43.

Lampson, Butler, "A Scheduling Philosophy for Multiprocessing Systems", *Comm. ACM*, 11, 5 (May 1968), pp.347-365.

---, "Dynamic Protection Structures", *Proc. AFIPS Fall Joint Computer Conference*, 31, 1969, pp.27-38.

Mednick, Sarnoff, "The Associative Basis of the Creative Process", *Psychological Review*, 69, 3 (May 1962), pp.220-232.

Miller, George, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information" in *The Psychology of Communication*, G.Miller, Basic Books, New York, 1967.

Mitchell, James, *The Design and Construction of Flexible and Efficient Interactive Programming Systems*, Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, 1970.

Moore, O.K., "Autotelic Responsive Environments and Exceptional Children" in *The Special Child in Century 21*, J.Hellmuth (Ed.), Seattle: Special Child Publications of the Sequin School, Seattle, 1964.

---, and Anderson, A.R., "Some Principles for the Design of Clarifying Educational Environments" in *Handbook of Socialization Theory and Research*, Goslin (Ed.), Rand McNally and Co., 1969.

Moran, Thomas, "The Symbolic Nature of Visual Imagery", *Proc. Third IJCAI*, Stanford, 1973, pp.472-477.

Negroponte, Nicholas, *The Architecture Machine*, The MIT Press, Cambridge, Mass., 1970.

Pager, David, "On the Problem of Communicating Complex Information", *Information Processing 71*, North-Holland Publishing Co., 1972.

Palme, Jacob, *SIMULA 67 -- an Advanced Programming and Simulation Language*, Norwegian Computing Center, Forskningsveien 1b, Oslo, Norway, 1970.

---, "SIMULA as a Tool for Extensible Program Products", *SIGPLAN Notices*, 9, 2 (February 1974), pp.24-40.

Papert, Seymour, "LOGO Book Notes", AI Laboratory, MIT, Cambridge, Mass., 1970.

---, *A Computer Laboratory for Elementary Schools*, Artificial Intelligence Memo 246, MIT, Cambridge, Mass., 1971(a)

---, *Teaching Children Thinking*, Artificial Intelligence Memo 247, MIT, Cambridge, Mass., 1971(b).

---, *Teaching Children to be Mathematicians vs. Teaching About Mathematics*, Artificial Intelligence Memo 249, MIT, Cambridge, Mass., 1971(c).

---, *Uses of Technology to Enhance Education*, Artificial Intelligence Memo 298, MIT, Cambridge, Mass., 1973

Petit, Philip, *RAID*, Stanford Artificial Intelligence Laboratory Operating Note 58.1, Stanford University, Stanford 1970.

Roberts, L.G., "Graphical Communication and Control Languages", *Proc. Second Congress on Information System Sciences*, 1964, pp.211-217.

Rovner, Paul, and Henderson, D.Austin, "On the Implementation of AMBIT/G: a Graphical Programming Language", *Proc. First IJCAI*, Washington, D.C., 1969.

Rowland, Kurt, *Learning to See*, Van Nostrand Reinhold Co., New York, 1971.

Sloman, Aaron, "Interactions Between Philosophy and Artificial Intelligence: The Role of Intuition and Non-logical Reasoning in Intelligence", *Proc. Second IJCAI*, London, 1971, pp.270-278.

Sussman, Gerald, *A Computational Model of Skill Acquisition*, Ph.D. thesis, MIT Project MAC TR-297, Cambridge, Mass., 1973.

Sutherland, Ivan, *Sketchpad, A Man-Machine Graphical Communication System*, Ph.D. thesis, MIT, Cambridge, Mass., 1963, also in *Proc. AFIPS Fall Joint Computer Conference*, 23, 1963, pp.329-346.

Sutherland, William, *On-Line Graphical Specifications of Computer Procedures*, Ph.D. thesis, MIT, Cambridge, Mass., 1966.

Swinehart, Daniel, *COPILOT: A Multiple Process Approach to Interactive Programming Systems*, Stanford Artificial Intelligence Laboratory Memo AIM-230, Stanford University, Stanford, 1974.

---, *TV, A Display Text Editor*, (internal document), Stanford Artificial Intelligence Laboratory, Stanford University, Stanford, 1971.

Tesler, Lawrence, Enea, Horace, and Smith, David, "The LISP70 Pattern Matching System", *Proc. Third IJCAI*, Stanford, 1973, pp.671-676.

Wallace, Robert, *The World of Leonardo*, Time-Life Books, New York, 1966.

Webster's Seventh New Collegiate Dictionary, G. and C. Merriam Co., Springfield, Mass., 1967.

Wells, Mark, and Morris, James, *Proc. Symposium on Two-Dimensional Man-Machine Communication*, SIGPLAN Notices, 7, 10 (October 1972)

Wertenbaker, Lael, *The World of Picasso*, Time-Life Books, New York, 1967.

Whitfield, Stephen, and Roddenberry, Gene, *The Making of Star Trek*, Ballantine Books, New York, 1968.

Wilson, Curtis, "How Did Kepler Discover His First Two Laws?", *Scientific American*, 226, 3 (March 1972), pp 92-106.

Winston, Patrick, *Learning Structural Descriptions from Examples*, Ph.D. thesis, MIT Project MAC TR-231, Cambridge, Mass, 1970.

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
STAN-CS-75-499, AIM-260		

4. TITLE (and Subtitle)	5. TYPE OF REPORT & PERIOD COVERED
PYGMALION: A CREATIVE PROGRAMMING ENVIRONMENT.	technical, June 1975
	6. PERFORMING ORG. REPORT NUMBER
	STAN-CS-75-499 (also AIM 260)

7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s)
David Canfield Smith	DAHC 15-73-C-0435,

9. PERFORMING ORGANIZATION NAME AND ADDRESS	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
Computer Science Department Stanford University Stanford, California 94305	ARPA Order 2494

11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE
Col. D. Russell, Deputy Director ARPA/IPT ARPA Headquarters 1400 Wilson Blvd., Arlington, Va. 22209	June 1975
	13. NUMBER OF PAGES
	195

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report)
ONR Representative: Philipp Surra Durand Aeronautics Bldg., Rm. 165 Stanford University Stanford, Ca. 94305	UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Releasable without limitations on dissemination.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

see reverse

DD FORM 1473 1 JAN 73   EDITION OF 1 NOV 65 IS OBSOLETE

#20

PYGMALION is a two-dimensional, visual programming system implemented on an interactive computer with graphics display. Communication between human being and computer is by means of visual entities called "icons", subsuming the notions of "variable", "reference", "data structure", "function" and "picture". The heart of the system is an interactive "remembering" editor for icons, which executes and (optionally) saves operations for later re-execution. The display screen is viewed as a document to be edited. Programming consists of creating a sequence of display frames, the last of which contains the desired information. Display frames are modified by editing operations. PYGMALION employs a powerful paradigm that can be incorporated in virtually any other programming language:

> Every operation has both visual (aesthetic) semantics and internal (mechanical) semantics.

In fact, every operation in PYGMALION has three responsibilities:

> (a) for accomplishing a given internal machine task - the machine "semantics" of the operation;

> (b) in display mode, for generating a representative visual action;

> (c) in remember mode, for adding onto a code list the operation(s) necessary to reproduce itself.

Thus the system includes an incremental "iconic compiler". Since each operation has visual semantics, the display becomes a visual metaphor for computing. The programmer need deal with operations only on the display level; the corresponding machine semantics are managed automatically. The mechanical aspects of programming languages has been and is continuing to be well studied. The focus in this paper is on developing and interacting with an articulate visual presentation.

PYGMALION is a computational extension of the brain's short term memory. It is designed to relieve the load on the short term memory by providing alternative storage for mental images during thought. The display screen is seen as a "dynamic blackboard", on which ideas can be projected and animated. Instead of abstract symbols, the programmer uses explicit display images. Considerable flexibility is provided for designing icons; the programmer may give them any shape that can be generated by a routine. This helps to reduce the translation distance between representations used in the mind in thinking about a problem and representations used in programming the problem.

> The main innovations of PYGMALION are:

> (1) a dynamic representation for programs- an emphasis on doing rather than telling;

> (2) an iconic representation for parameters and data structures requiring less translation from mental representations;

> (3) a "remembering" editor for icons;

> (4) descriptions in terms of the concrete, which PYGMALION turns into the abstract.

The responsive, visual characteristics of PYGMALION permit it to play an active role in human problem solving. The principal application has been in assisting the design and simulation of algorithms.