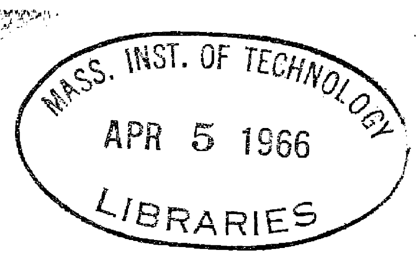


THE ON-LINE GRAPHICAL SPECIFICATION OF COMPUTER PROCEDURES

by

WILLIAM ROBERT SUTHERLAND

B.E.E., Rensselaer Polytechnic Institute

(1957)

M.S., Massachusetts Institute of Technology

(1963)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January, 1966

Signature of Author...
Department of Electrical Engineering, January 10, 1966

Certified by...
Thesis Supervisor

Accepted by..
Chairman, Departmental Committee on Graduate Students

THE ON-LINE GRAPHICAL SPECIFICATION OF COMPUTER PROCEDURES

by

WILLIAM ROBERT SUTHERLAND

Submitted to the Department of Electrical Engineering on January 10, 1966, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ABSTRACT

An interactive computer graphics system may be used for describing procedures in a two-dimensional programming language as well as for the more usual task of manipulating graphical data. The work described here is concerned with the graphical specification of procedures which then may be applied to problem data obtained from any source. The notation used to form a two-dimensional description of a procedure and the conventions used to control the procedure's execution are described.

An experimental graphical programming system has been created for the TX-2 Computer. With this system one may draw an arbitrary symbol and give it a meaning. The system has a "macro" capability enabling a new symbol to be defined as a combination of operators. A procedure may be executed after initial values are assigned. A number of debugging features are included in the system.

The conventions described make a graphical procedure description a natural way of expressing parallel operations. The standard notions of flow must be substantially modified. Explicit flow control is generally omitted but may be included at the users option. Some aspects of compiling from a graphical program are also discussed.

Thesis Supervisor: Claude E. Shannon

Title: Donner Professor of Science

ACKNOWLEDGMENTS

I would like to thank Professors C. E. Shannon, S. A. Coons, and T. G. Stockham, Jr. for their help and advice. Their comments and enthusiasm provided me with a great deal of stimulation and direction.

The interest and criticism of L. G. Roberts and J. A. Feldman of Lincoln Laboratory was especially valuable. They clarified a number of subtle points during many hours of discussion. My brother, Ivan, provided timely advice as well as much of the earlier computer graphics experience upon which this work rests.

The support of the M. I. T. Lincoln Laboratory made my research possible. J. I. Raffel, J. L. Mitchell and Group 23 provided the equipment, assistance, and TX-2 computer time required. Thanks are due to P. Rovner for his summer assistance on the project and to Ellin Foreman who retyped my writing many times.

Finally, this work would not have been possible without the patience and support of Linda.

TABLE OF CONTENTS

Abstract		2
Acknowledgements		3
Table of Contents		4
List of Figures		5
Chapter I.	Introduction	7
Chapter II.	Graphical Language	17
Chapter III.	Graphical Procedure Description	34
Chapter IV.	Graphical Procedure Conventions	43
Chapter V.	The Experimental System	59
Chapter VI.	A Two-Dimensional Language Scheme	77
Chapter VII.	Considerations in Compiling Machine Code	86
Chapter VIII.	Conclusions and Recommendations	95
Appendix A.	System Control Commands	102
B.	Experimental System Primitive Operations	107
C.	The CORAL Language and Data Structure	111
Bibliography		124
Biographical Note		127

LIST OF FIGURES

Figure	1.1.	Basic Symbols	12
	1.2.	Connected Program	12
	1.3.	Initial Values	14
	1.4.	Initial Values	14
	1.5.	Procedure After Execution	15
	1.6.	Variable Values Shown	15
Figure	2.1.	Man-Machine Graphical Communication	19
	2.2.	System Block Diagram	21
	2.3.	Graphical Symbol Connection	27
	2.4.	Positional Connection	28
	2.5.	Results of Moving Symbols	30
Figure	3.1.	Graphical Arithmetic Example	35
	3.2.	Another Arithmetic Example	37
	3.3.	Graphical Syntax Error	39
	3.4.	New Symbols for Transfer Function	40
Figure	4.1.	Flow Forks and Joins	44
	4.2.	Example Procedure	46
	4.3.	Corrected Example	49
	4.4.	Example With Flow Control	50
	4.5.	Connection Examples	53
	4.6.	Flow Path Pieces	54
	4.7.	Required-Data Join	56

Figure 5.1. Control Responses 66

5.2. Automatically Connected Symbols 68

5.3. Debugging Features 73

5.4. Trap and Data Probe Operators 75

Figure 6.1. Graphical Associative Programming Language 81

Figure 7.1. Parallel Operations 91

7.2. Independent Operations 92

Appendix C

Figure 1. Basic Ring Element Structure 113

2. Example of Block Form 114

CHAPTER I

INTRODUCTION

Men find pictures useful for communicating with computers as well as for conversing with each other. The possibilities for man-machine graphical communication range from the output of a simple graph to a complicated two-way conversation. Many of the answers which machines find for us are most conveniently presented in a pictorial form; much input data is too. Data such as the shape of a car body, a graph of rising sales, the predicted motions of an orbiting satellite, or an ordinary weather map can be depicted on a computer display or automatically drawn chart. Problem data in graphical form has been the concern of many computer graphics efforts.

The program given to a computer for solving a problem need not be in a written format. One might graphically define a procedure for handling data obtained from a radar set, from a ticket reservation office, or from some other source. This paper describes a graphical form of procedure specification and a working experimental system based on this technique.

BACKGROUND

The advantages of graphical input and output of data are obvious. The system developed by Roberts [24] illustrates capabilities which are in current use. A facsimile device attached to the computer scans a photograph of a set of solid objects. From this information the computer creates a perspective

line-drawing on the output display. The system user may rotate and move the view creating the impression of the observer moving about in real space. The computer makes appropriate assumptions about hidden portions of the objects so that in effect one may go around behind the photograph and take a look.

To accomplish the input and output of graphic data, a number of hardware devices have been developed. Cathode ray tube displays and automatic plotters exist in great variety. Graphic input devices include flying-spot scanners, TV cameras, and the relatively new RAND tablet [3]. This device consists of a writing surface with an embedded grid of wires and a stylus used to write on the surface. Electronic circuits provide the computer with continuous stylus coordinate information. Input of graphic data may also be accomplished with a light-pen and cathode-ray display.

Another aspect of computer graphics involves a close coordination between man and machine, on-line in real time. They interact through pictures rather than through the written languages generally used for communication with computers. The most widely known interactive graphics system is SKETCHPAD [28]. It successfully demonstrates the desirability of two-way graphical communication between man and computer. Using a light-pen and computer-driven display, the operator is able to draw a picture on the display face. He may view, move, modify, and erase the picture and its parts in a number of convenient ways. In addition, he can apply various constraints to the picture parts to maintain desired geometrical relationships. The operator need not be a skilled computer expert to use the system. The SKETCHPAD report has been widely distributed and has influenced many computer graphics

efforts including this one.

A three-dimensional SKETCHPAD has been developed [11]. It allows a user to manipulate top, front, side, and perspective views of wire frame objects. The General Motors Research Laboratory is developing a comprehensive system intended to automate a large portion of the blueprint paperwork associated with automobile design [33]. The system there has facilities for scanning-in graphical information, changing the resulting displayed picture in an interactive mode, and then making hard copy prints of the new results. Similar systems are being developed elsewhere in industry and at universities [26].

All of the work mentioned so far has been concerned with graphical data. Very little has been accomplished toward the graphical specification of procedures. Systems designed to produce flow-charts from written computer programs have been reported [8, 16], but these efforts deal with the problems of program analysis. In the realm of program synthesis, a preliminary start on a flow-chart compiler has been reported from M.I.T. [31]. This compiler is designed to accept MAD language statements placed in flow-chart boxes. Graphical programming work, unpublished as yet,* is also proceeding at the RAND Corporation. Otherwise, it appears that on-line graphical procedure specification has been a neglected field.

AN INTRODUCTORY EXAMPLE

The following example is presented to introduce the basic notions of graphical programming described in succeeding chapters. It illustrates how

* Publication of the details of GRAIL by T. Ellis and W. Sibley of RAND is expected soon.

one uses the graphical programming system that has been created on the TX-2 Computer at the M.I.T. Lincoln Laboratory. The chosen problem is to accept a series of numbers typed in on a keyboard and to type out their running sum. In addition, we will require that the computer display on the scope face the largest of the input numbers. The problem is trivial, but it is specifically chosen to illustrate a graphically formatted procedure working on non-graphical data.

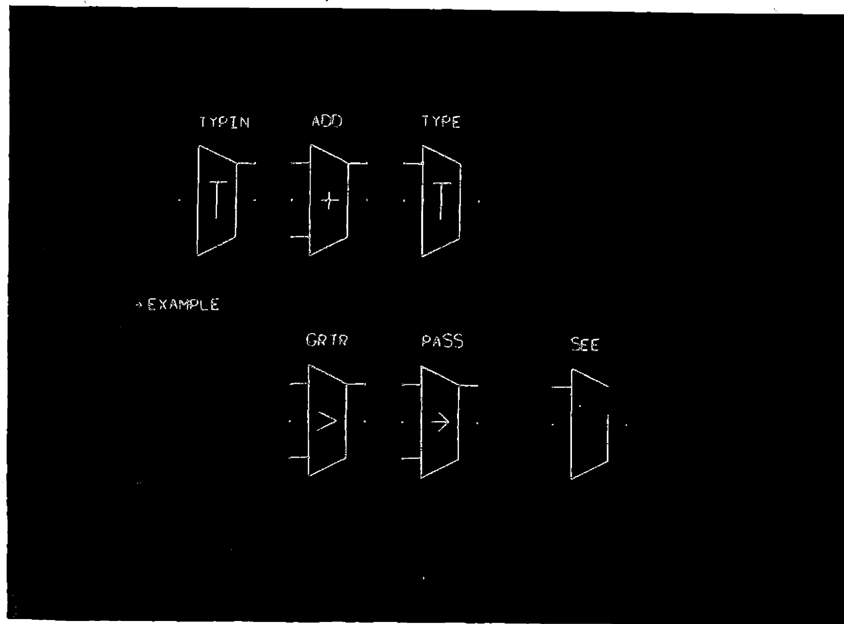
We will program this task by creating a pictorial procedure description on the computer display. The drawing facility provided within the experimental graphical programming system is very similar to that of SKETCHPAD. The reader should envision the system operating as follows: By giving commands with push buttons, foot pedals, and a keyboard, and by manipulating the light-pen as a pencil on the display face, the user creates a picture on the display. To draw a line a user places the pen at a starting location, presses a button labelled DRAW, and then moves the pen to a terminal position. The computer will display a line connecting the initial and terminal locations. One may delete a graphical entity by pointing at it with the light-pen and pressing a DELETE button. A detailed description of the controls of the experimental system will be found in Chapter V and Appendix A.

Any programming language, written or graphical, must include some primitive operations. To accomplish the defined task one needs actions which permit typing in and out, addition, and comparison. The experimental system provides these actions as primitives, but makes no restriction on the shape of the symbol used to represent each. Thus, the first programming step is to construct a symbol for each action needed in the example. This is done by

drawing a master picture for each symbol and designating terminals for connection to other symbols. The computer must be told which primitive action each master symbol represents. Replicas of the master symbols will then be used for constructing the pictorial program. Figure 1.1 shows the six symbols created for this example with their names directly above each symbol. The slanting tops and bottoms of the elements are intended to convey a sense of direction, indicating that inputs are on the left and outputs on the right. The detached dots to the left and right of each symbol are flow terminals. All operations have provision for flow in (to activate the operation) and flow out (to activate a succeeding operation). The flow terminals are optional and may be deleted when not needed.

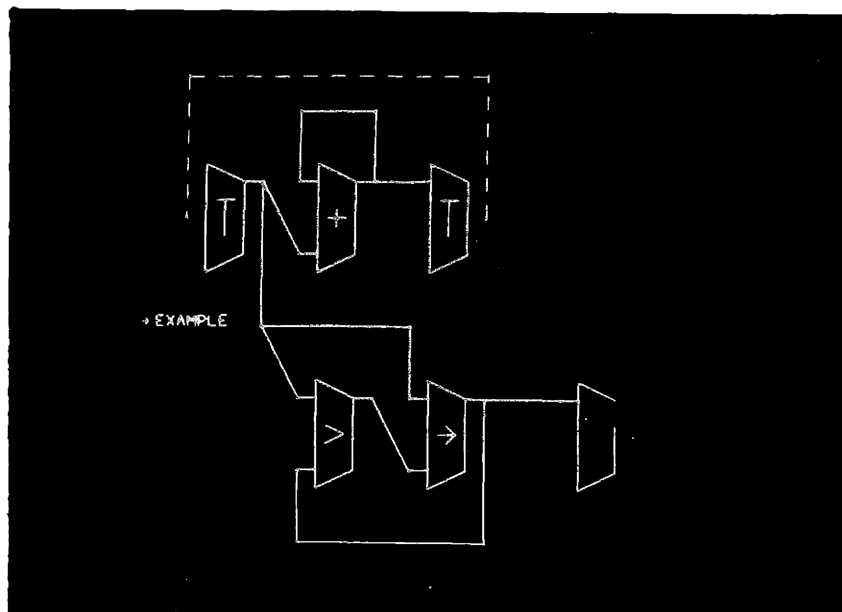
The meaning of the addition operator should be clear. The type-in and type-out operators at run-time are a data source and sink and represent the actions of the keyboard and printer. The comparison operator, "GRTR", produces a boolean output of true if the upper data value is larger than the lower. The "SEE" operation converts its one data input into visible text representing the data value. The last operation, "PASS", does not have a direct counterpart in written programming. This operator serves as a valve controlling the passage of the upper data variable through the operator. The lower data input is a boolean control; true means pass the data and false, block it.

Figure 1.2 shows the six operators connected together to make the program. This program was made by calling up replicas of the master symbols and then connecting terminals with lines. Note that the only explicit flow lines (dashed lines) are those used to reactivate the type-in operator after



Basic Symbols

Figure 1.1



Connected Program

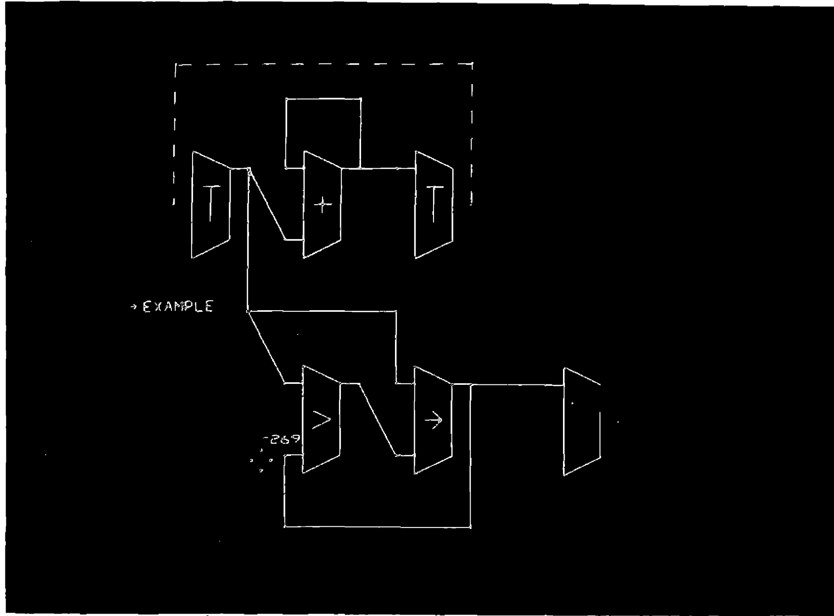
Figure 1.2

each type-out. Everywhere else the flow terminals have been deleted, for the data connections serve to indicate how the procedure should operate. Before the program can be run, initial values must be assigned by placing the desired values on the data terminals as indicated in Figures 1.3 and 1.4. The numerical values are obtained from the keyboard used by the operator. The operator keys in a number, picks up the light-pen, and places the number on the desired variable in the program. The particular numbers shown in Figures 1.3 and 1.4 were chosen at random.

When the procedure is activated the user may type in numbers which constitute the problem data. For each input number he obtains the running sum on the output printer. The greatest input number is displayed as shown in Figure 1.5. Except for some side issues, this is the end of the example since the desired results have been accomplished.

The system provides information about program execution to the user. The symbols of active operators blink so that one can follow the procedure's execution and obtain a useful visualization of its operation. The blinking of symbols is most useful in a slow step-by-step mode of operation in which the computer does a set of operations and then waits for the push of a button before moving on to the next set. If we operate the example in this mode, we will find out that several things go on at once. After the type-in operator has produced a new data value, the addition and comparison operations will be performed simultaneously as indicated by the data connections. The specification of parallel processing operations is accomplished implicitly by removing flow from the program where not needed. Little thought was given to an order for activating the process pieces during the construction of the program.

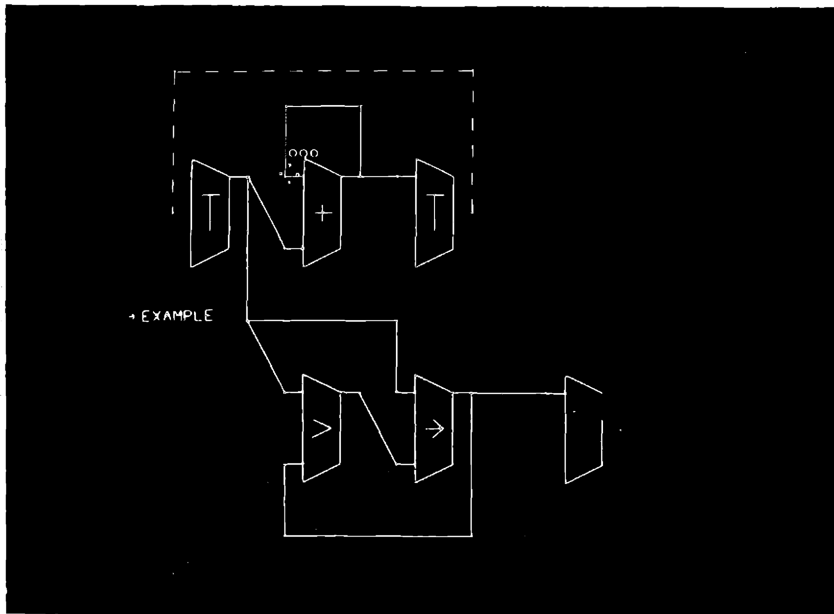
21



NOTE: The 4 dots are the pen tracking cross

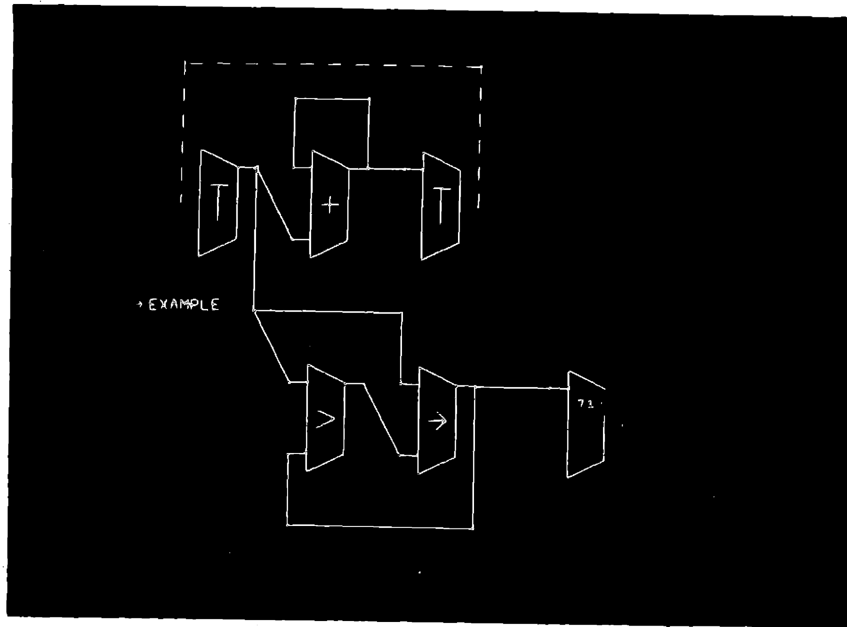
Initial Value Assignment

Figure 1.3



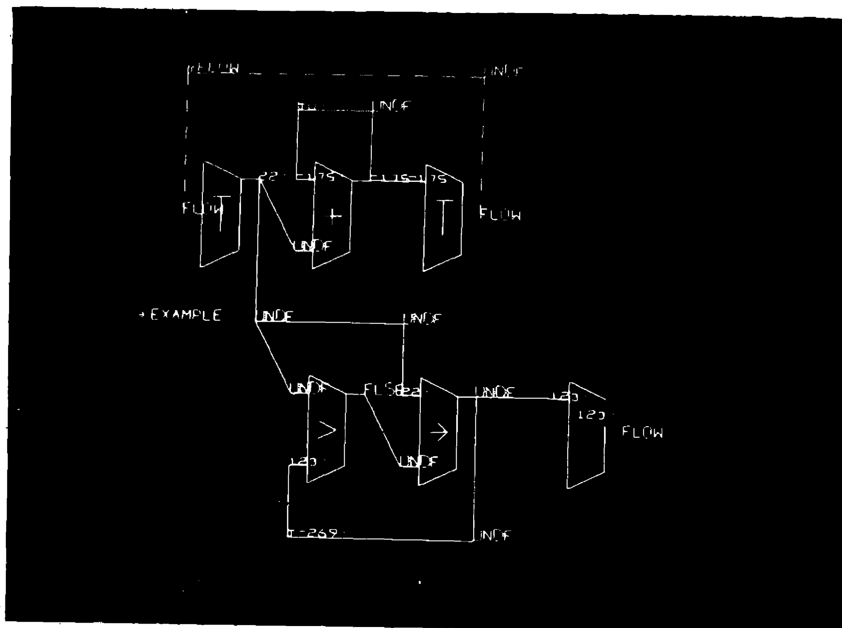
Initial Value Assignment

Figure 1.4



Procedure After Execution

Figure 1.5



Variable Values Shown

Figure 1.6

This freedom from explicit ordering is one of the most valuable results of this work.

There is a discrepancy in the graphical program presented for this example. Since the type-in operator triggers the addition and comparison operators, why does not the addition operator continuously trigger itself? As the program is drawn this would indeed happen, and the program would operate incorrectly. In Chapter IV we show how this discrepancy is overcome and it is mentioned here only for reassurance about this potentially worrysome difficulty.

People make mistakes and therefore debugging features are an important part of any programming system. The experimental graphical programming system incorporates a number of such features. In Figure 1.6 for example, the values of all variables in the graphical program are displayed for examination. The procedure execution has been interrupted, and the system has been placed in a special mode. The running sum is now -175, and the most recent number typed in was 22. The system's debugging features are covered in Chapter V.

There are two aspects of this introductory example which should be carefully scrutinized for fundamental principles. First, what are the basic factors involved in this kind of two-way graphical communication between man and machine; and second, what are the underlying concepts involved in the form of graphical procedure description presented here? A consideration of these two points will occupy the bulk of the following chapters.

CHAPTER II

GRAPHICAL LANGUAGE

Before we discuss the details of a "Graphical Programming Language" we should examine the general term "Graphical Language"; there are at least three different ways in which it has been used.

1. A written programming language used for implementing computer graphics; i.e., a source language to be fed into a compiler, thus creating the machine code programs required for graphical communication with a computer.
   EXAMPLE: CORAL*, LISP

2. A control language used for running an interactive computer graphics system. The language used by the man to communicate with the machine.
   EXAMPLE: The push button and light-pen language of SKETCHPAD.

3. A two-dimensional pictorial language used to communicate anything that is convenient to represent that way. The communication may be man-to-man, or as in SKETCHPAD, machine-to-man.
   EXAMPLE: The language of circuit diagrams or architectural blueprints.

The written language defined in (1) above is a tool used to create the programs for a computer graphics system. The latter two kinds of graphical language have been badly confused and misused in the past. A clear appreciation of the difference between them is essential. The term "Pictorial

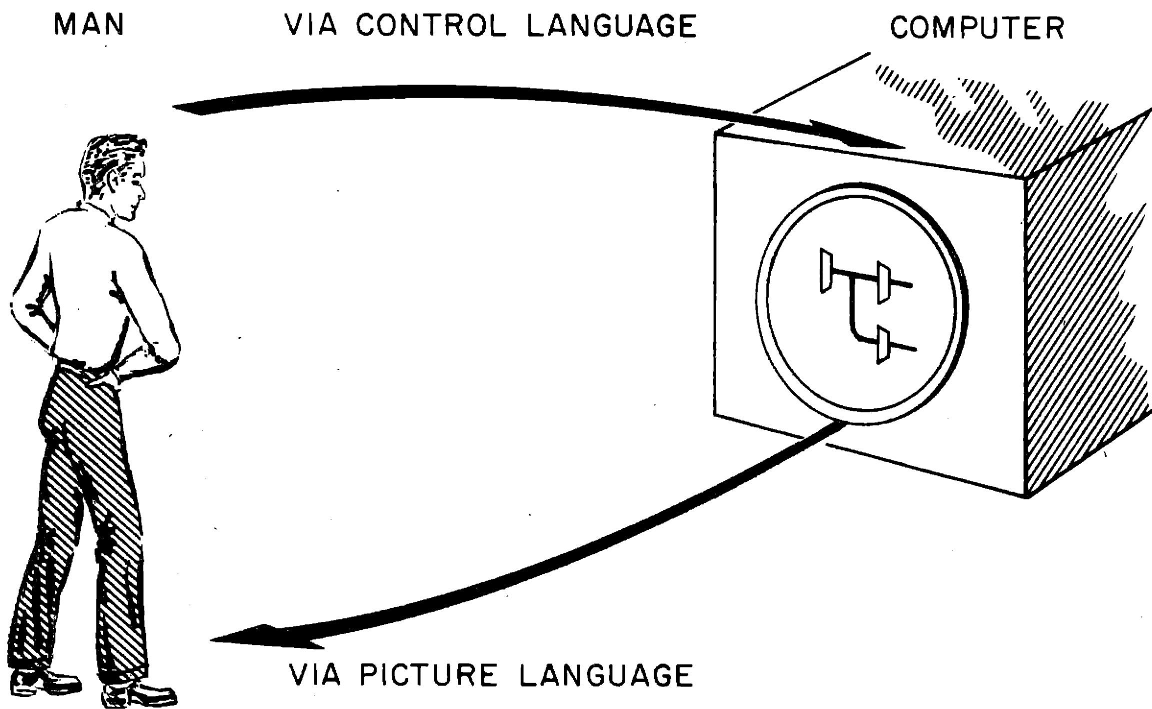
* CORAL is the written language used to create the experimental graphical programming system reported here. See Appendix C.

Graphical Language" (3 above) is meant to apply to a picture language; a two-dimensional representation of something on a piece of paper, on a photograph, or on a computer display. The picture may have been created by a man with a pencil, by light striking a photographic emulsion, or by some device under computer control. The "Control Graphical Language" (2 above) is very different. It is the language used to tell the computer where to place and how to connect the picture parts on the computer display. Consider a man with only a wooden pointer verbally telling an intelligent robot with a piece of chalk how to draw a picture on a blackboard. The verbal stream of instructions and the pointer motions are the control language, and the resulting picture on the blackboard is the pictorial graphical language. They are two different things.

The man-machine graphical communication in a SKETCHPAD-like system takes place as illustrated in Figure 2.1. The man communicates his desires and instructions to the machine via one graphical language (a control language), and the machine verifies back its "understanding" of the man's wishes via another graphical language (a picture language). To illustrate this point let us look at the functional parts of some unspecified graphical communication system. The illustrative system described will be a general one, and not one oriented toward any particular subject.

THE FUNCTIONAL PARTS OF AN UNSPECIFIED SYSTEM

By examining the parts of an interactive graphical system we hope to gain a better understanding of how the system operates and of the roles played by the control and graphical languages. In addition, we will see how



Man-Machine Graphical Communication

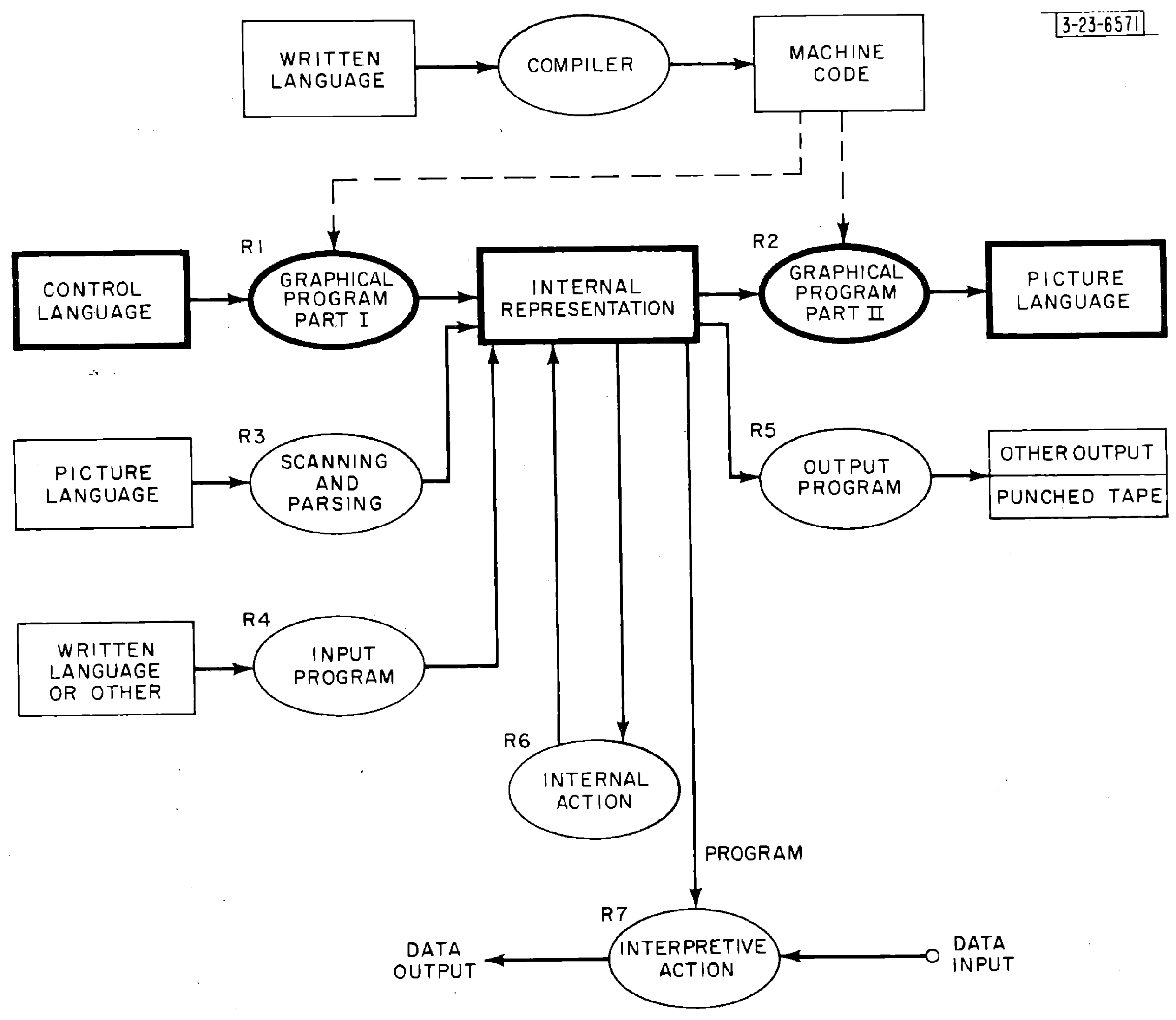
Figure 2.1

the parts affect the system's capabilities. Figure 2.2 shows the basic system parts with dark borders. Rectangles indicate data and oval shapes represent programs obtained from the written language and compiler shown at the top. The "Internal Representation" is some specialized arrangement of computer memory containing the problem information in a form useful to the system. Historically, the internal representation has been a list structure which the computer could examine or manipulate as required.

The first system part to be considered is R1 which takes control language statements as input and builds up the internal representation step by step. In this way the user's actions with the light-pen and buttons are converted into a computer memory version of the picture he desires. The display program (R2) converts the internal representation into a pictorial output for the man to view. These two actions go on together. The remaining parts in Figure 2.2 represent additions which might be made to the basic system of R1 and R2.

Adding an extra program (R3) and suitable scanning equipment would allow the direct input of graphical information. Instead of building up the picture step by step with a control language, the user might draw a picture on a piece of paper and use that as input data for the machine. Other inputs are also possible. An additional program (R4) would allow the system to accept written input data. This data might be a written version of a program and a flow chart of the written program might be the desired output.

Combinations of the three kinds of input (written, control, and pictorial) will be useful. After a written program has been accepted as data and a flow chart produced on the computer display, one might wish to use the



System Block Diagram

Figure 2.2

interactive graphical control language to make changes to the flow chart. Similarly, one might wish to update a blueprint which has been scanned into the computer. It should be possible to obtain a written version of the corrected program, or perhaps to get a punched paper tape of machine tool instructions for making the part drawn on the blueprint [19]. The program part labelled R5 accomplishes this sort of action.

The internal representation may also contain information which specifies how changes should be made to other data represented inside the machine. This situation is found in the constraint features of SKETCHPAD*. Consider the simple case where a constraint has been created in the internal representation. This constraint is associated with the two end points of a line and indicates that the line is to be kept horizontal. Whenever the line is moved off the horizontal, some program (R6) must make the line horizontal again.

The utility of a graphical communication system is determined by the variety of programs provided within the system. A system with only minimal R1 and R2 provides a simple drafting capability. To improve that capability, R1 and R2 must be expanded and other programs (R3 – 6) added. It should be quite clear in each case how to obtain the additional programs required; use a written language and compiler to make the necessary machine code.

Let us use the functional system to work on some problem. So far we have subtly assumed that the graphical communication will involve only the problem data, and that instructions on how to solve the problem will be

---

* Refer to SKETCHPAD report for information on constraints.

provided by explicit programs. The system is useful only with those problems for which a solution has been programmed and provided. Any man-machine communication about how to solve a particular class of problem must be carried out in advance by a system expert using a written programming language and compiler. What is missing is the ability for man and machine to converse graphically about how to solve problems, as distinguished from the ability to converse graphically about the details of a problem to be solved.

With a general-purpose drawing capability we can draw any picture, even one representing a procedure. Difficulties occur only when one wishes the computer to carry out the graphical procedure and actually do something. Then the internal representation of the picture used to describe the procedure must serve as a program to an interpretive action. The result is an active process which can act on other data. In Figure 2.2 an interpretive action (R7) uses as its program the same data structure which represents the picture of the graphically formatted procedure. The system's capabilities are expanded since graphical communication is no longer restricted to problem data.

The distinction between the system control language and the picture language being used to represent something (be it program or data) is most important. In weighing the merits of a particular system one might ask the three-way question, "Is the system's graphical language natural and easy to use?". Is the control language natural and easy to use, or is the picture itself a convenient and clarifying way to represent whatever we have in mind, or does the written programming language make adding to the system easy? Confusion is imminent when the term "graphical language" is used carelessly. This brief examination of system parts has served its purpose, and we will

next consider some of the fundamental properties of the various kinds of graphical language.

GRAPHICAL LANGUAGE FUNDAMENTALS

A written graphical language is a one-dimensional (linear) language. At present virtually all communication with computers is accomplished via written one-dimensional languages. Consequently the theoretical properties of linear languages have received much attention. All that one need say about a written graphical language is that it is a standard written computer programming language designed to make creating graphics programs easy. Several such languages have been implemented by the various groups doing graphic research [23, 26]. The particular written language used to implement a graphical communication system has little effect other than programming convenience on the form of the system's control or pictorial languages.

The control languages used in graphical communication systems are also linear (i.e., sequential) languages. The time sequence of button-pushes and pen-actions is a time-ordered string of operators and operands. For instance, a command might be: "Make <u>this</u> parallel to <u>that</u> and perpendicular to <u>that</u>". The meaning of "this" and "that" is defined when the user points with the light-pen to some picture part, presses a button called "PARALLEL" and then points to some other picture part. Since the control and written languages are both one-dimensional, it may well be that a single general-purpose compiler-translator could be used for both. At least one research group is actively pursuing this hypothesis [26]. It is also convenient to provide a written language capability within the system's controls. For example, a

typewriter is very useful when one wishes to assign a name to a picture.

The two similar kinds of graphical language discussed so far are both used as man-to-machine communication media. The similarity may be partially ascribed to the fact that present-day computers and computer input devices are sequential in operation. It is therefore difficult for any man-to-machine communication to be accomplished other than sequentially. However, in his sense of vision man has a considerable parallel input capability. He can accept a non-sequential language statement with his eyes. Machine-to-man communication may also be accomplished via a non-sequential language provided that the sequential nature of the computer output is hidden from the man. The speed of a computer-driven display masks the time sequence of its outputs, and even though it works slowly, a computer plotting device draws an entire picture before the man takes time to look at the results. Thus communication from machine to man is possible with a non-sequential pictorial language. In trying to understand the fundamentals of a picture language, the guidance provided by linear languages is very valuable. For both kinds of language we may make a standard breakdown into symbol set, syntax, and semantics.

SYMBOL SET

The symbol set of a pictorial language, or any language for that matter, is a clear-cut axiomatic concept. By definition both parties to the communication must have an a-priori common knowledge of the legitimate symbols of the language. The symbols can be arbitrary but both sides must know what they are. In addition, given a language statement, the parties to the communication must be able to identify the individual symbols which make up

the statement. If two symbols can be placed together and made to look like a third, confusion may result. Then it may be impossible to decide which symbol or combination of symbols the author of the statement meant. As long as the symbols remain within the bounds of clarity, the symbol set may be any arbitrary mutually-agreeable one.

SYNTAX

To create a language statement one uses appropriate symbols and connects them together. In a linear language there is only one method of connecting symbols; that of sequential ordering. Each symbol in a string has a left and right neighbor. A statement in a pictorial language will similarly have connected symbols, but it will use conventions other than simple sequence for indicating symbol connection. There are two ways of indicating graphical connection of symbols; positional connection and explicit connection. The distinction is not clear cut and depends on one's point of view. Positional connection implies that two symbols are related or connected together by virtue of their relative positions. Explicit connection requires that some direct indication run between the two symbols to show the connection. A line is a commonly used indicator. Examples of both modes are shown in Figure 2.3.

That the distinction between these two connection modes is one of point of view may be argued as follows: Include the explicit connection indicator in the symbol set. Now compare the picture of Figure 2.4 with the one of Figure 2.3. They both look the same. However, in Figure 2.3 there are two symbols and an explicit connection indicator, while Figure 2.4 has

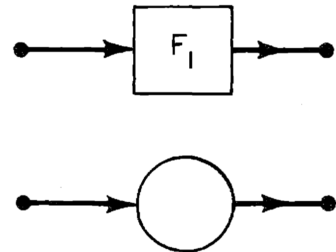
SYMBOL SET

3-23-6572

POSITIONAL CONNECTION

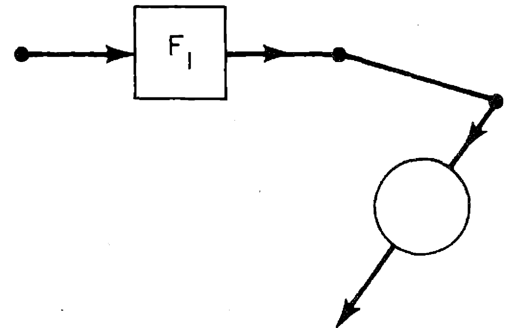
$= \; dx \; 3 \; y \; + \; \int \; x \; z$

EXPLICIT CONNECTION



STATEMENT

$$\int_{3}^{y+3} x \, dx = z$$

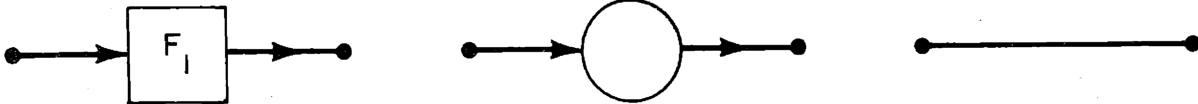


Graphical Symbol Connection

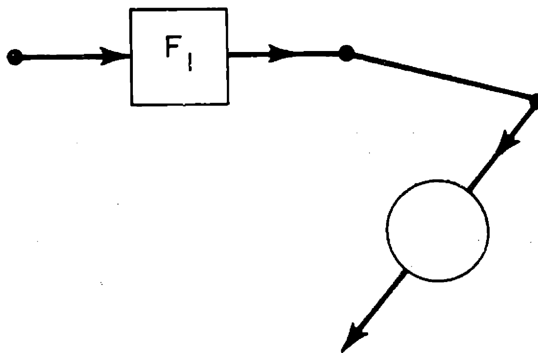
Figure 2.3

SYMBOL SET

3-23-6573



STATEMENT



Positional Connection

Figure 2.4

three symbols connected by positional conventions at the ends of the line. The distinction depends on what assumptions we make about the symbol set. A second argument refutes the first; this argument reemphasizes the importance of separating the two connection modes. Consider what happens when one of the symbols is moved. If they are explicitly connected, position is of no importance and the connection is maintained. As seen in Figure 2.5, moving a positionally connected symbol may remove the connection.

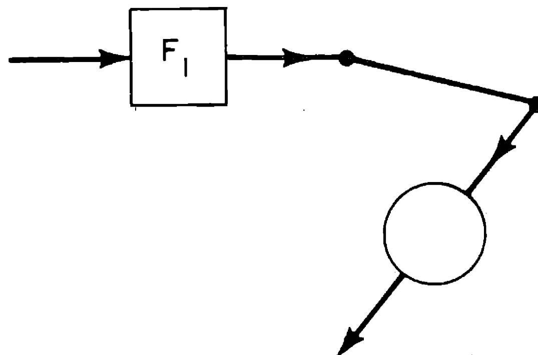
Symbol connection by labelling is also possible. Labels are used as a substitute for an explicit connection between two symbols. The label is itself a symbol connected to the original symbol by positional conventions. Positional and explicit connection conventions both have a place in graphical communication, but it is important to realize which one is being used in any particular case.

The symbols used by a pictorial language are affected by the mode of connection. Symbols which are to be connected by positional relations must have "sensitive regions" designated as part of the symbol definition [15, 20]. A connection is established only if a region of one symbol overlaps a region of another symbol. The integral sign in Figure 2.3 has four regions where it expects other symbols to appear; upper and lower limits, integrand, and differential. In addition, the integral sign and connected arguments are positionally connected to another symbol, the equal sign. Since there can be only a finite number of different relationships between symbols, the "sensitive regions" must be discrete. An infinite continuum of allowable positional relationships is impossible.

Symbols which are to be explicitly connected must include as part of

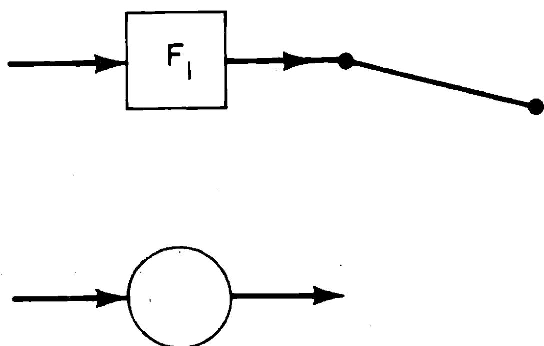
BEFORE MOVING

3-23-6574

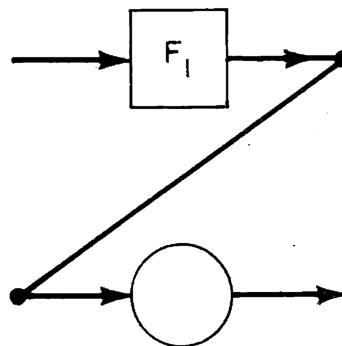


AFTER MOVING

POSITIONAL CONNECTION



EXPLICIT CONNECTION



Results of Moving Symbols

Figure 2.5

their definition the allowable terminals for connectors. Just as the "sensitive regions" above were quantized and finite, there must be a finite number of attachment points to represent the inter-symbol relations. The term "attachment point" is not meant to imply only a geometrical point although that is often the case. If the symbol were a circle the entire circumference of the circle could represent one attachment entity. This symbol could then stand in only one relationship (via explicit connection) to any other symbol.

It is not immediately obvious how picture syntax affects the kind of man-machine graphical communication being considered here. SKETCHPAD for instance, has very successfully avoided syntax problems. However, its function was drawing pictures with geometrical relations between the picture parts rather than the connection of symbols into a procedure description. The exact geometry depicting the procedure is of little interest. Since the syntax of a language is concerned with a formal description of how the language's symbols connect together, picture syntax is more directly relevant to a graphical procedure description than to a SKETCHPAD drawing. Even so, there are varying degrees of syntactic complexity which one faces.

Consider a particular graphical system which is designed to accept a complete picture as input. The pictorial description of a procedure is first drawn on a piece of paper. The paper is placed in a scanner and the picture is fed into the computer as a raster of dots. The computer is then faced with the task of progressing from a knowledge of the dot raster to a full understanding of the procedure described by the input picture. A full scale of pattern recognition and picture parsing problems must be handled. In

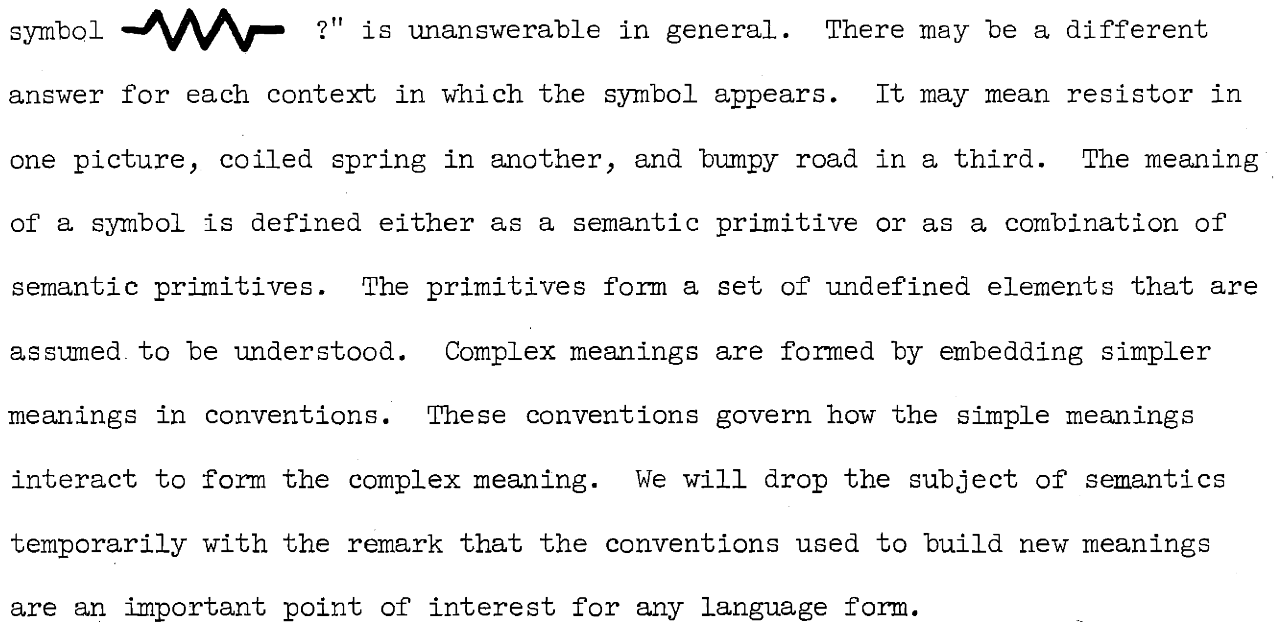
programming the computer to accomplish these tasks, a full and detailed knowledge of picture syntax is necessary. Some work has been done in formalizing picture syntax [13, 21, 22], but there is as yet no commonly accepted "Backus Normal Form" for multi-dimensional languages.

Luckily, the interactive kind of graphical communication described here places very little demand on our knowledge of picture syntax. The control language instructions which build up a picture are very explicit and leave little for the computer to interpret on its own. An input statement which says explicitly "Connect this terminal to that terminal" has done whatever picture parsing is necessary. It is most important to note that the connecting line which then appears is a result of and not the cause of the computer's knowledge that the two terminals should be connected! This is a great simplification. We do not force the computer to figure out that six lines placed appropriately are a box with a plus in it, and that this means addition. Instead, we call up by name (say ADD) a symbol which the computer knows means addition even though it may have any shape.

The subject of syntax has been raised to help clarify some of the basic concepts underlying the pictorial language used in graphical communication systems. Examples where picture syntax plays an important role in the graphical programming language are presented in the next chapter.

SEMANTICS

The last and most difficult subject in the three part breakdown of language is semantics. The subject of semantics cannot be considered in absolute terms out of context. The question "What are the semantics of the

symbol  ?" is unanswerable in general. There may be a different answer for each context in which the symbol appears. It may mean resistor in one picture, coiled spring in another, and bumpy road in a third. The meaning of a symbol is defined either as a semantic primitive or as a combination of semantic primitives. The primitives form a set of undefined elements that are assumed to be understood. Complex meanings are formed by embedding simpler meanings in conventions. These conventions govern how the simple meanings interact to form the complex meaning. We will drop the subject of semantics temporarily with the remark that the conventions used to build new meanings are an important point of interest for any language form.

CHAPTER III

GRAPHICAL PROCEDURE DESCRIPTION

We must next determine a reasonable graphical format and associated conventions for representing procedures during an interactive graphical conversation. In doing so we should expect to draw heavily on the notational forms which experience has proved valuable. The block diagram is an initial form to follow. For constructing a pictorial procedure description we will choose box-like symbols which have input and output terminals. These terminals will be used for connecting symbols together by either explicit or positional conventions. The terminals are portals through which variable values are introduced to and obtained from a "transfer function" within the box. Figure 3.1 shows written and graphical forms of an arithmetic computation. Note that the graphical statement appears very clumsy in comparison with the written form. There is a well-developed, efficient, compact notation for written arithmetic which is universally used and understood.

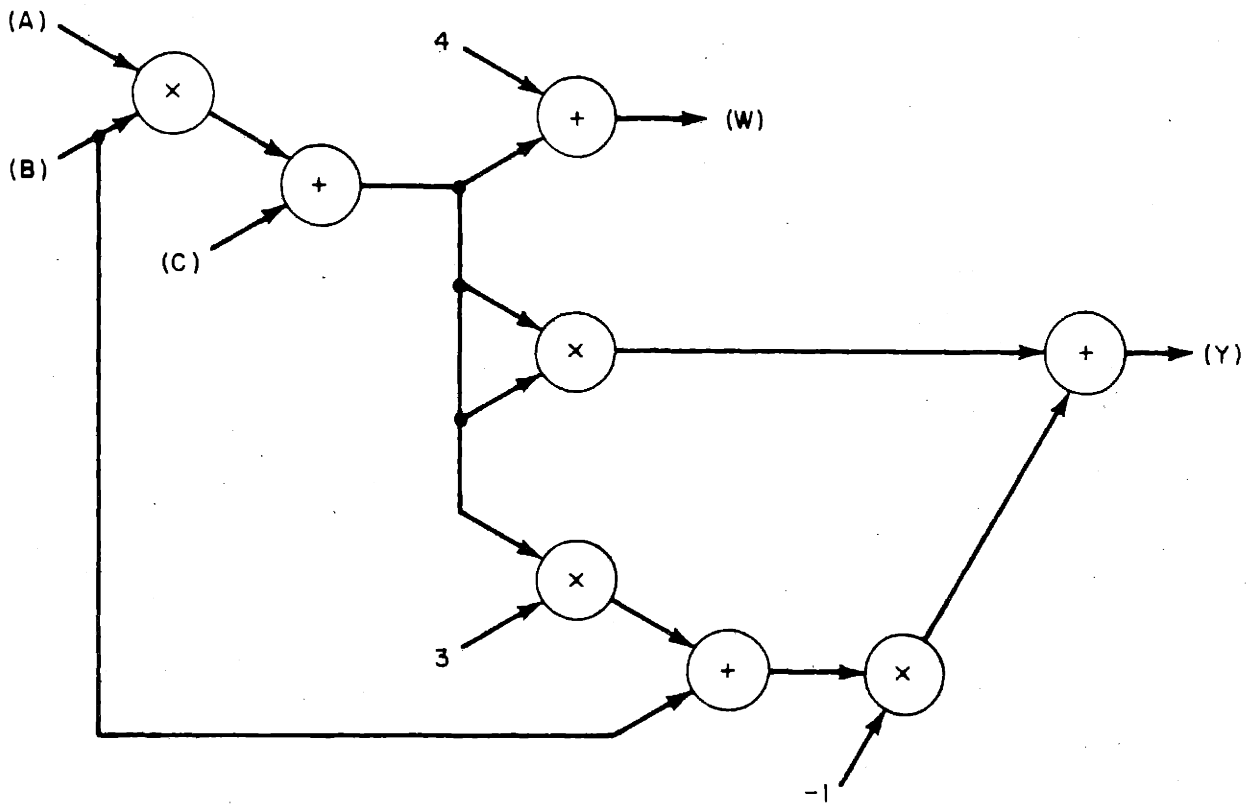
There are, however, some important lessons to be gleaned from the graphical portion of Figure 3.1. In the graphical statement an explicit name for the intermediate variable "Z" of the written form is not required. One can "run a wire" to connect directly to any needed variable. Note also that "B" is graphically connected to two places corresponding to the two appearances of "B" in the written form. In a linear language any particular instance of a symbol can appear in only one place in the symbol string and

3-23-6575

WRITTEN STATEMENT

$$Z = A \times B + C$$
$$W = Z + 4$$
$$Y = Z^2 - (3Z + B)$$

GRAPHICAL STATEMENT



NOTE: THE LETTERS IN PARENTHESES CORRESPOND TO THE WRITTEN FORM ABOVE

Graphical Arithmetic Example

Figure 3.1

can be directly connected only to a left and right neighbor. Many conventions such as labels, names, parentheses, etc., are used to overcome this inherent restriction of linear languages. A multi-dimensional pictorial language does not require such devices.

Another distinction between written and graphical languages has a different importance. There is no satisfactory way of indicating the total implications of multiple-valued outputs for a linear language. Linear language notation often leads one into neglecting the whole story. A simple example, again from arithmetic, involves the operation of square root. (See Figure 3.2) As before, the graphical statement is clumsy, but it is virtually impossible to ignore the fact that there are two resulting answers.

SEMANTICS AGAIN

Let us assume that the individual arithmetic operations in Figure 3.1 are primitive; i.e., we assume that they can be properly performed. If the reader were told that A = 4 and B = -5 and C = 12 he could determine X and Y from either notational form. There is a process implied by the graphical statement which the reader intuitively understands, or he would not be able to carry through to an answer. Furthermore, this process does not involve a unique sequence for performing the individual operations. The concept of a unique sequence involves the unwarranted assumption that the operations must be performed sequentially, one at a time. Figure 3.1 vividly indicates that there are three operations which could be performed simultaneously. That is not so obvious from the written form.

WRITTEN STATEMENT

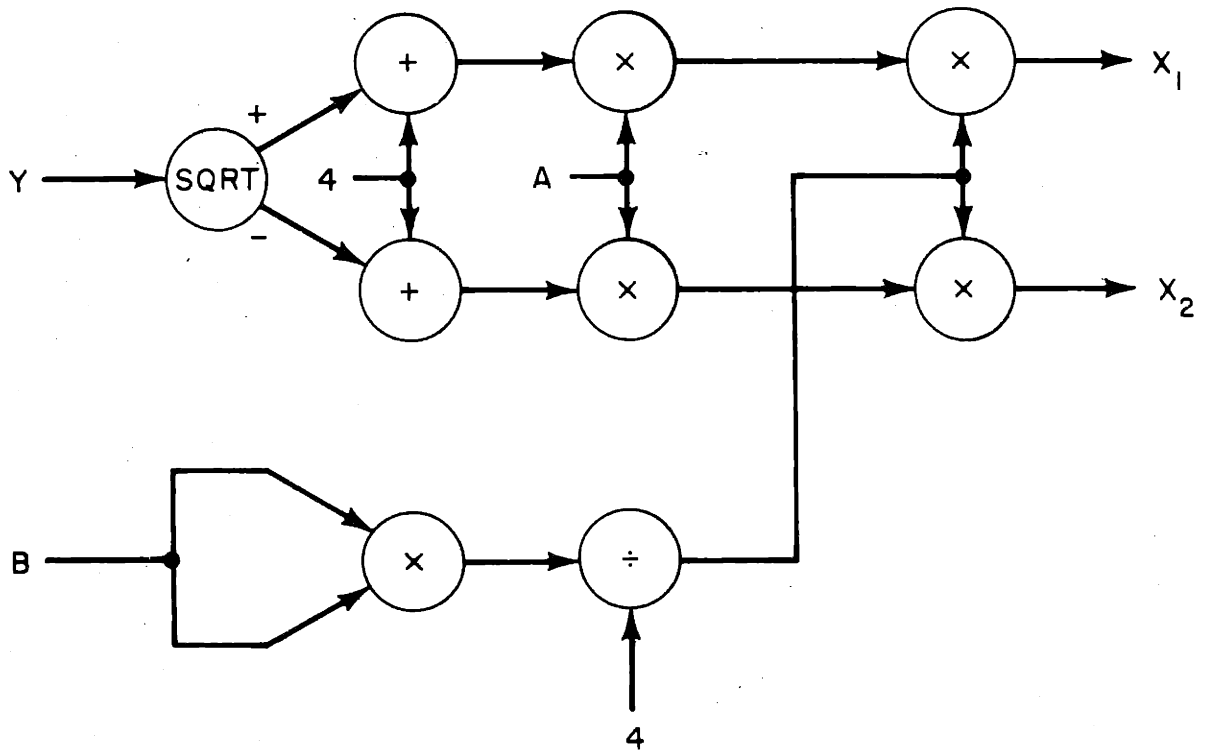
3-23-6576

$$Z = A \times \left[4 + SQRT(Y)\right]$$

$$X = Z \times 4/B^2$$

THE FACT THAT THERE ARE TWO VALUES OF X IS EASY TO NEGLECT

GRAPHICAL STATEMENT



Another Arithmetic Example

Figure 3.2

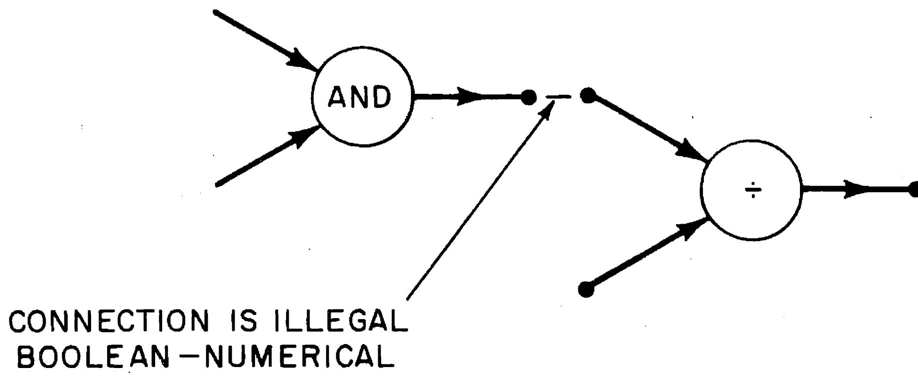
If the computer is to carry out graphical program statements like those in Figure 3.1 and 3.2 the interpreter of the graphical program must execute the pictorial program in a manner which agrees with what the man would do. More examples will be necessary before we can attempt to specify what should be included in the program-execution interpreter. It is clear, however, that the combination of primitive meanings and program activation conventions serve to define a more complex meaning.

USEFUL SYMBOL SYNTAX

Symbol portals may be classified by type, in effect giving each portal a "pipe size". When told to connect two terminals a graphical programming system may first check to see that the connection is allowable. The user may thus be prevented from connecting together terminals having incompatible types. The classification by type parallels the declaration of data types used in conventional written languages and gives the computer the ability to detect certain graphical syntax errors. Figure 3.3 illustrates a typical disallowed connection.

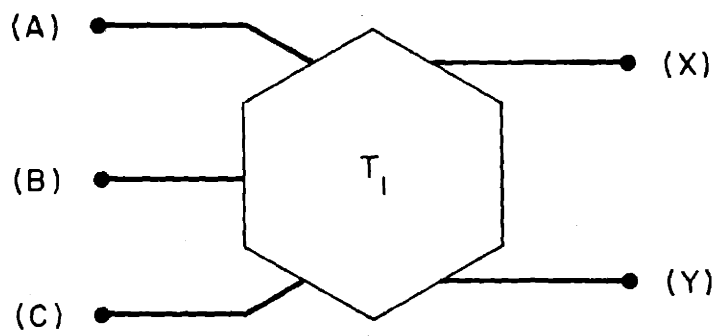
NEW SYMBOLS WITH MEANING

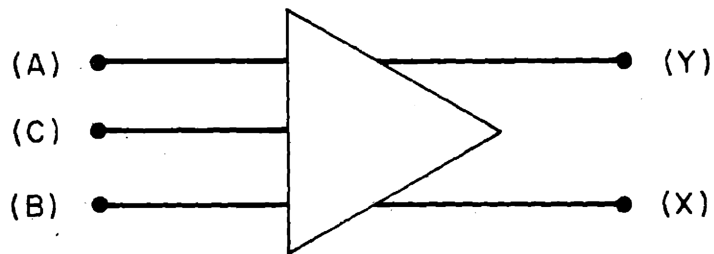
Let us define a new symbol to represent the entire pictorial program of Figure 3.1. This program has three inputs and two outputs, and can itself be considered a transfer function. The new symbol must also have three input portals and two outputs to match its meaning. Figure 3.4 shows several possible new symbols that could be constructed. After the shape of the symbol is defined we must declare that this new symbol is to mean the transfer

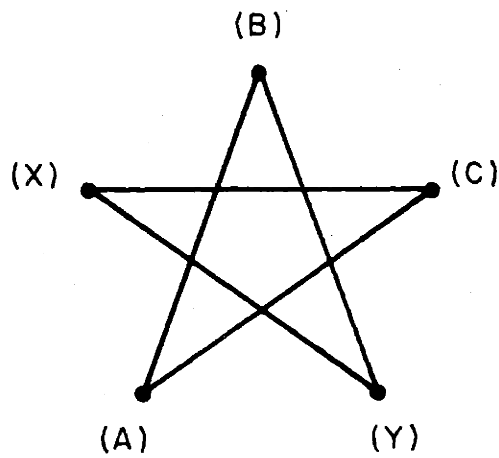


Graphical Syntax Error

Figure 3.3







New Symbols For Transfer Function
Figure 3.4

function of the original picture (Figure 3.1). We must also indicate how the designated terminals on the new symbol match up with the inputs and outputs of the semantic definition picture. The symbol should be constructed to remind the user of which terminal is which; the star is a poor symbol in this respect.

The defining process is analagous to the definition of a subroutine or macro in a conventional written language. The new symbol can be used in another graphical program, and another new symbol can be defined to represent the resulting, more complicated process. The definition layering can continue indefinitely. The user need not be aware of which operations are primitive and which are defined in terms of other graphical programs. If proper care is taken with regard to variable storage, it is even possible to use a process recursively in its own semantic definition. Such a recursion does not involve the pictorial representation of an operator, but only the definition of the operator's meaning.

Providing symbol terminals with data types, affects the actions required to define new symbols. Besides indicating which parts of a new symbol are to be attachment terminals, a user must indicate the kind of attachment terminal desired. This is very conveniently combined with the matching of terminals between a new symbol and semantic definition picture. Usually the terminal on the symbol should have the same data type as the corresponding terminal in the defining picture.

SUMMARY

Let us review the notation chosen for the graphical representation of procedures. The basic symbols are considered as block diagram elements, although they need not look like blocks. The shape of a symbol is quite arbitrary. Symbols have input and output terminals which may be connected to other symbol terminals. Symbols represent operations which transform data arriving on the input terminals into output values which are sent on to other operators. Symbols may be assigned a primitive meaning, or they may have a meaning defined graphically by a picture containing other symbols connected together. The functioning of a connected network of operations is controlled by a set of process conventions unspecified as yet. There remain uncompleted two basic considerations: the first involving the details of process conventions suitable for use with graphical procedure descriptions, and the second involving the question of where and how the graphical procedure gets its data. These topics are covered in the next two chapters.

CHAPTER IV

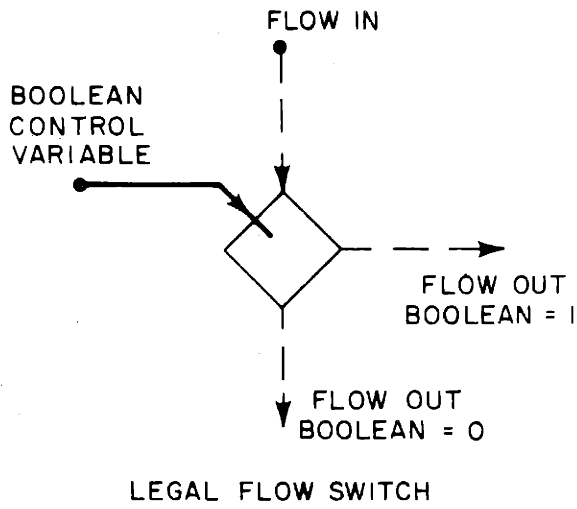
GRAPHICAL PROCEDURE CONVENTIONS

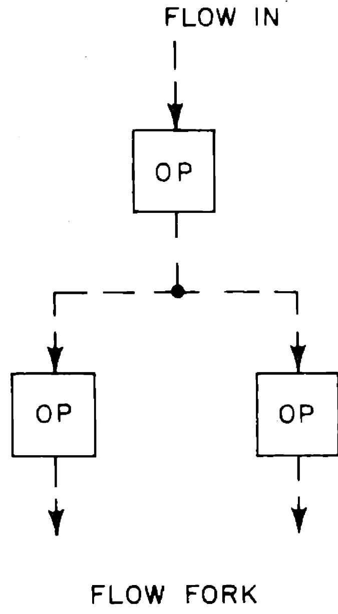
The conventions which control how a graphical program operates are the next topic of discussion. We cannot in general expect that one particular set of conventions will suffice for activating every possible form of graphical program. In a written programming language the normal convention is to execute each statement in sequential order, unless a control transfer specifies otherwise, and it is not usually possible to execute the statements in some strange order depending on the number of symbols in each line. Similarly, we are looking for a useful process convention to use with procedure specifying pictures.

FLOW CONVENTION

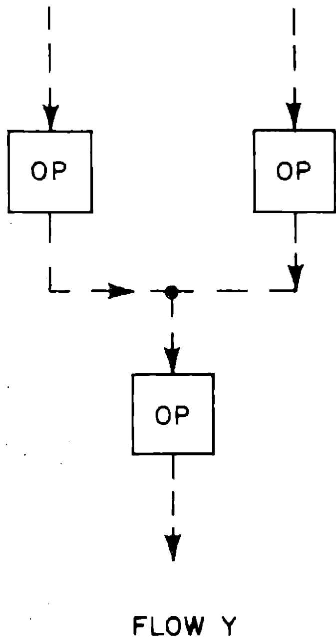
One simple convention directly indicates how the process is to proceed. Each operation must be provided with a flow input and a flow output terminal. These flow terminals must be connected together, and the process activity will traverse the flow pathways activating each operator in the explicitly indicated order. If we assume that only one processor is available, then the flow path must be a single thread through the operators. A flow switch is a legal operator, but a flow fork is obviously illegal. (See Figure 4.1 A) If many processors can be at work simultaneously then the flow fork is perfectly correct [2]. If we also permit flow forks and many active operators

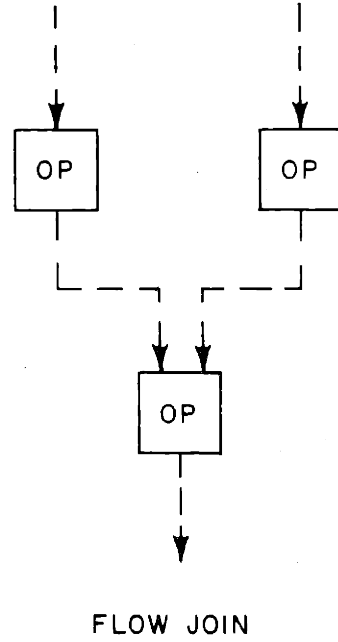
3-23-6579





(a)





(b)

Flow Forks and Joins

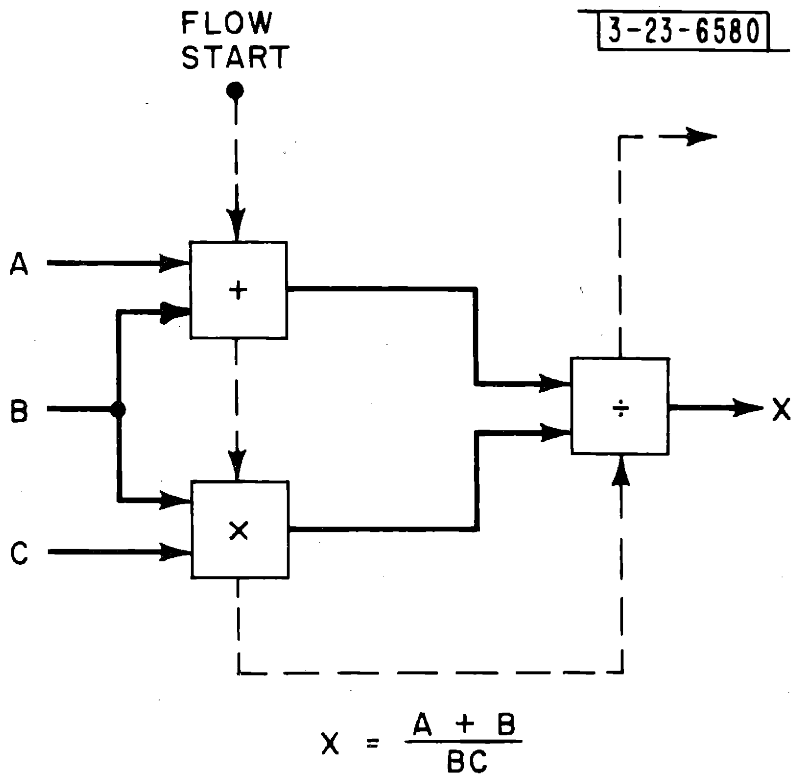
Figure 4.1

we must also allow the possibility of merging flow paths. Figure 4.1B shows two possible representations; a discussion of their meaning will be temporarily postponed. Explicitly controlling the flow of activity through the program is one way to insure correct operation.

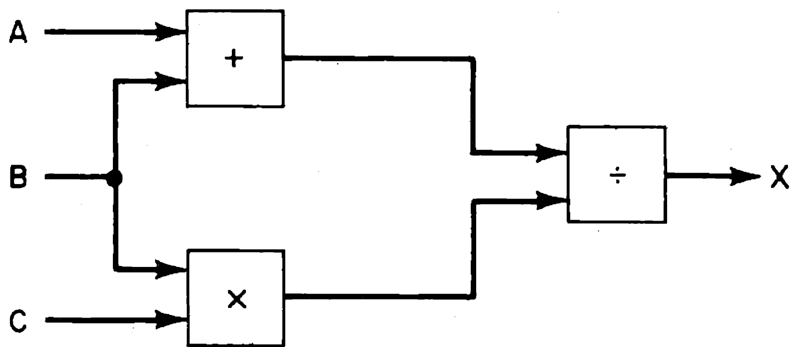
What happens if the network of Figure 4.2A is activated without specifying any values for A, B, and C? These variables might then have random values, or perhaps by convention all unspecified values are zero. Such conventions ignore the fact that a variable may sometimes have the legitimate value of "UNDEFINED". It is meaningless to talk about an actual numerical value for the outputs of the operators in Figure 4.2A before the operations are performed. Under normal programming conventions one assumes that variables have random or "garbage" values until properly assigned a value. The programmer has the responsibility of arranging for a correct flow ordering to insure that variables have proper values assigned before their values are used. An incorrect flow order may result in the use of bad data and lead to an incorrect answer. The variable with a possible value of UNDEFINED is valuable because a convention may be chosen whereby an undefined input will prevent an operator from being activated.

DATA-FLOW CONVENTION

Let us abandon the concept of explicit flow control and instead use the data connections to guide the process activity through the procedure description. Instead of thinking of activating the procedure parts in a certain order, it is useful to think of repeatedly scanning the total procedure to determine which parts are ready to be activated next. The variable with a



(a)



(b)

Example Procedure

Figure 4.2

value UNDEFINED is important since usually an operation with an undefined input should not be performed. It is thus possible to obtain control over how a procedure is executed from the data connections present in the procedure. From this point of view the flow path in Figure 4.2A is not needed, and Figure 4.2B is an adequate description of the process. After the values of A, B, and C are assigned, the addition and multiplication can be done. The answer is obtained by doing the division after its inputs are defined.

We shall adopt the basic convention that data values sit forever at terminals unless deliberately changed. Under this convention additional criteria for determining when an operator should be activated are required. Some provision must be made for activating the operator only when appropriate and not continuously. Two things can be passed down a data line from one operator to another. The first is the data value, and the second is some indication of change. The arrival of a new data value at any input will trigger an operator provided that all of its inputs have defined values. This convention applied to Figure 4.2B will produce just the desired result. Modifications to this convention are desirable. A two input operator might require that new data arrive on both inputs before it can be reactivated. We have assumed that data values are present forever at operator input terminals until written over by new values. An alternative treatment is possible. When an operator is activated it can reset its input variables to a value of undefined. Then all its inputs must receive new data values before the operator will be activated again. To provide maximum flexibility each individual input terminal should have the option of keeping or destroying its data value when the operator is activated. There should be a visible difference

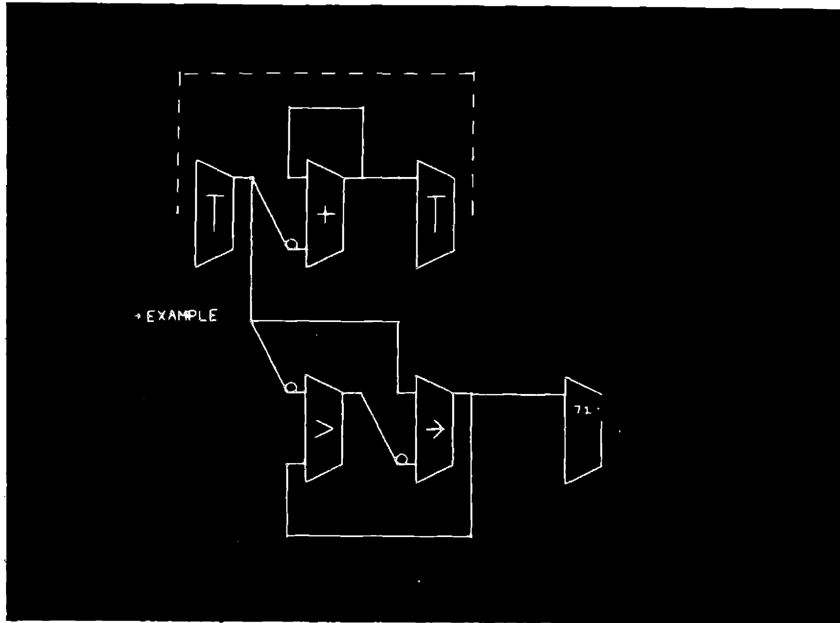
which informs a user which option he has designated at each terminal.

Another useful option is possible for any input terminal. Some operations may not care if an input is undefined, and will operate correctly anyway. This possibility should be available for the user to designate if appropriate. When this option has been chosen for an input terminal, that terminal will be excluded from the decision of when to activate its operator.

To illustrate the application of the data-flow conventions let us return to the example contained in Chapter I. That example was incorrect and would not work as shown. Figure 4.3A is a corrected version where the reset option has been chosen for three data inputs. The three circles at the inputs indicate that the special option is in effect at those terminals. In each of the three cases, data arriving at the other input terminal would cause the operator to activate at the wrong time unless prevented from doing so by an undefined value at the marked input.

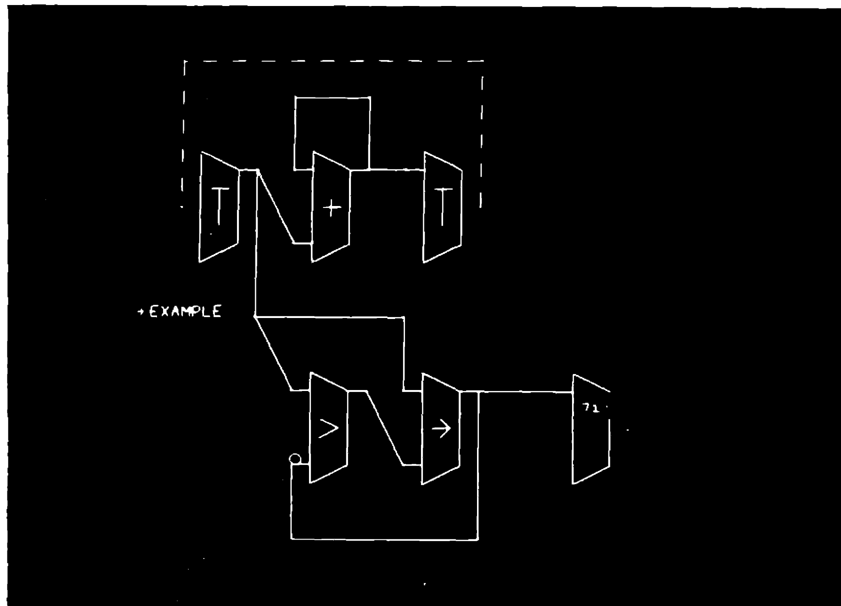
In the corrected version of this example normal input terminals retain their data values. The three terminals which reset their values to undefined are special cases. It is of course possible to reverse the basic assumption. Terminals which retain their input values are then special cases, specially marked. Figure 4.3B shows the example drawn under these reversed assumptions. Under either convention it is necessary to make a special designation only infrequently; the majority of the input terminals can have either convention applied. In all future examples unmarked input terminals are assumed to retain their data values. The possibility of reversing the convention was mentioned only for illustration.

Figure 4.4 is another version of the example procedure but with



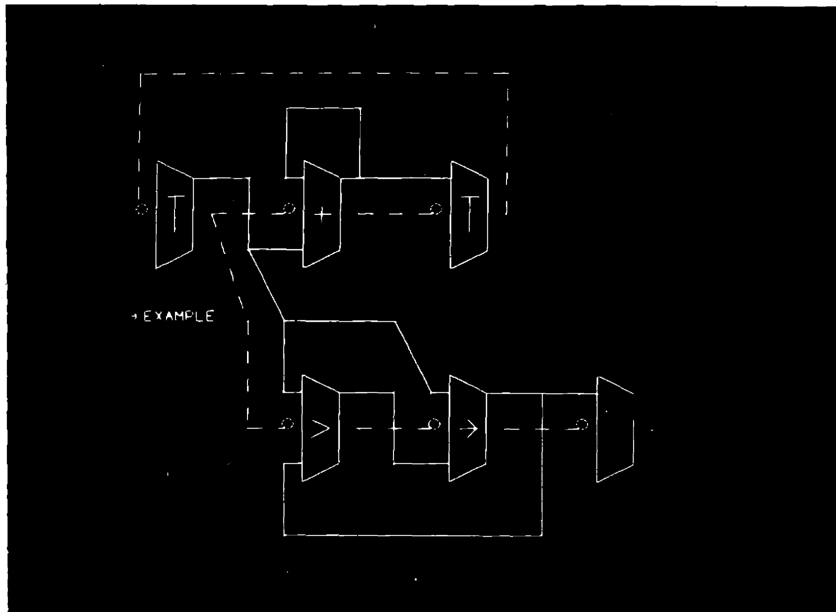
Corrected Example

Figure 4.3A



Example With Conventions Reversed

Figure 4.3B



Example With Flow Control
Figure 4.4

parallel flow paths present to provide an explicit designation of control. Note that the flow inputs now have the reset option instead of the data terminals.

FLOW WITHIN THE DATA CONTROL CONVENTIONS

It will next be shown that the "normal" concept of flow through a graphical program is optionally available if a user so desires. Let us allow an operator to have any number of dummy input variables. The values of these dummy inputs do not enter in any way into the calculation of operator output values. The dummy inputs serve only to help control the activation of the operator under the previously defined conventions. A flow input to an operator is precisely such a dummy input. A flow input always has the "reset data to undefined" option chosen. The actual data value that arrives on a flow terminal is unimportant since only the distinction between undefined and defined-as-something matters in deciding whether to trigger an operator. Thus any operator output could be used as a flow output. In practice, it has proved convenient to provide distinct flow output and input terminals to aid the user in keeping track of what he is doing.

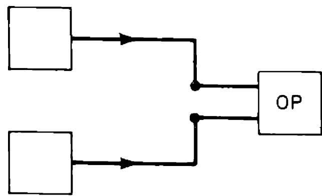
When a program is to be drawn with explicit flow control only the flow input terminals should have the reset option designated. All other data input terminals should retain their data values. Flow will then be the sole cause of operator activation, and each operator will be triggered without any undefined data inputs complicating its activation criteria. In actual practice explicit flow control is very seldom used. The data input

terminals are used with the appropriate options to control operator activation.

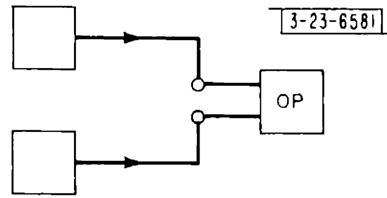
The same basic procedure activation conventions will satisfactorily handle any combination of explicit and non-explicit flow control. Explicit flow is treated as just another variable, although the user may choose to think of a flow path guiding the process activity. A flow line between two operators already connected by a data line would be redundant. All the necessary information can be passed down the data line. Some possible activation options may require that the data line connect to two separate input terminals, one a dummy. Figure 4.5 illustrates several examples of flow and data connection. Flow paths may start and stop as required to indicate the ordering and priority a user desires in a procedure description; Figure 4.6 illustrates this. It also illustrates the confusion which results from poorly constructed symbols. How are the "switch" outputs and the "greater" inputs assigned? This figure is not a specific procedure; it is a collection of connected operators. The output "Y" will be either undefined or equal to F(A + B)/CB depending on the results of the greater and switch operations. The reader must decide which way he will assign the terminals of these two operators.

JOINS AND Y'S

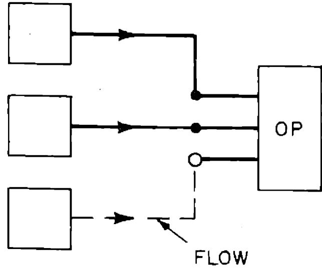
The "Y" connection where many flow paths converge indicates that a flow input may be defined from any one of a number of sources. The special operator shown in Figure 4.1B must be used to represent a parallel-processing "flow-join". When two (or more if so drawn) flow variables have accumulated



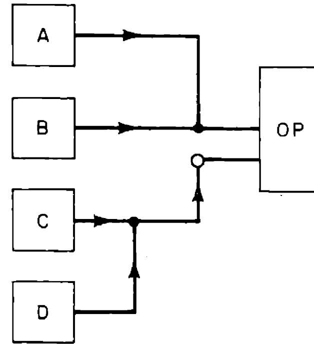
EITHER INPUT ACTIVATES OP



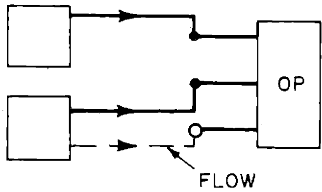
BOTH INPUTS REQUIRED TO ACTIVATE OP



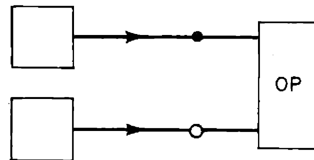
FLOW ACTIVATES OP



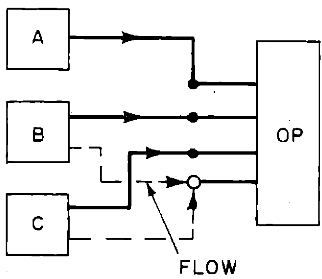
C OR D ACTIVATES OP



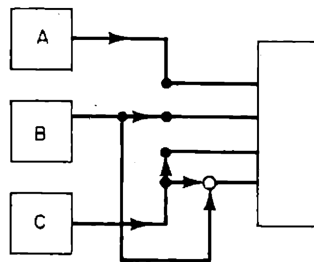
FLOW LINE UNNECESSARY SEE →



EQUIVALENT



B OR C TRIGGERS OP

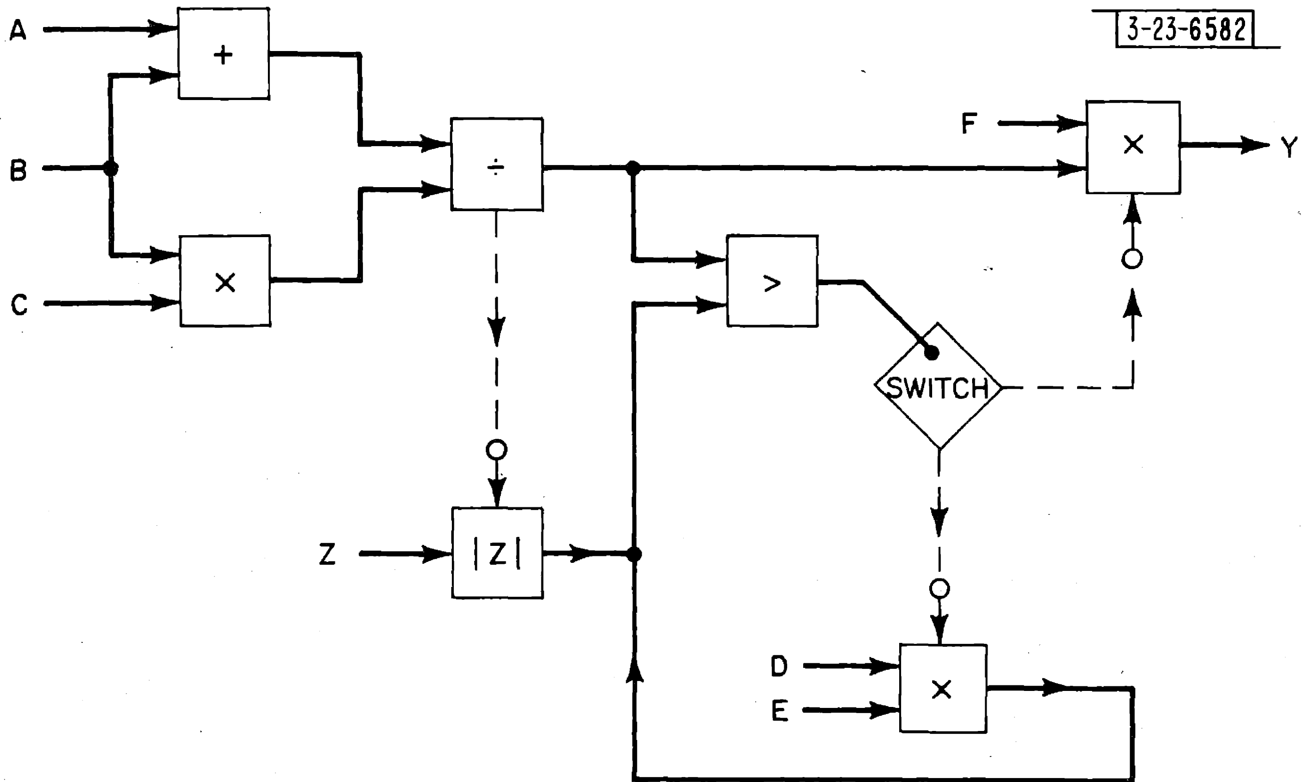


EQUIVALENT

●—— INDICATES "RETAIN DATA" OPTION
○—— INDICATES "RESET DATA" OPTION

Connection Examples

Figure 4.5



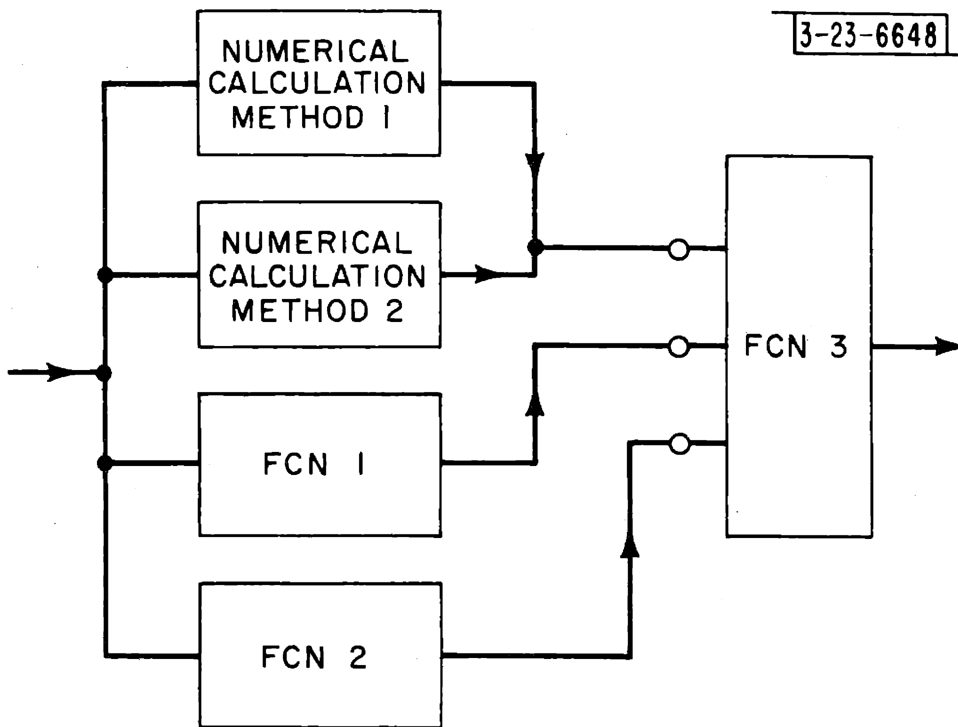
Flow Path Pieces

Figure 4.6

at the operator inputs, the join operator is executed and provides a defined flow variable as an output. Flow inputs are always reset to undefined. Flow must then arrive on all input terminals before the join operator can be executed again.

In a graphical program using the conventions described, a flow join as such will rarely be required. The requisite action may be provided for any operator by making appropriate choices of reset or retain data options at its input terminals. Furthermore, realizing the flow join action by means of data terminals provides a somewhat different kind of join action than has been suggested for written languages. The purpose of a join operation is to insure that when a number of processors are working on a problem together, all data necessary has been generated before the computation proceeds past the join. The standard method of keeping track of flow forks and joins as suggested by Conway [2] involves keeping a count of the number of preliminary tasks still unfinished. As each required action is completed, the count is decreased. The last action to finish will find a zero count in the join barrier and will then proceed on past the join. A fork operation setting up a new parallel path will of course increment the count in the appropriate join operation.

It is not at all obvious that the number of parallel paths provided and the number of data elements required must coincide. Figure 4.7 provides an illustration of this fact. There are four parallel paths and only three required pieces of data. This kind of situation could conceivably occur in a real time application. To obtain the top data input to FCN 3 in the minimum possible time, we are willing to use two processors running together.



Required-Data Join

Figure 4.7

Two different methods (perhaps one iterative, the other recursive) are used to calculate the answer. Presumably the time required differs depending on the particular data of the problem. The first answer obtained will be used. To require that four processes finish before executing FCN 3 is not what we have in mind; yet to have only three actions finish is not right either. It would then be possible to have the wrong three actions cause the operation of FCN 3. Keeping a count is not always satisfactory.

PROCEDURE OPERATION

Some means of matching a computer to the activation conventions just discussed must be provided since present computers do not operate under the graphical program conventions. Two solutions to this matching problem are possible. First, we may change the graphical procedure description into an equivalent form which may be directly executed by the computer. Chapter VII contains a discussion of this translation process (compiling). Second, we may create a programmed interpreter which will obey the activation conventions for executing a graphical program.

The first step in interpreting a graphical program is to initialize the program properly by setting to undefined all variables in the program which do not have an externally assigned specific value. Then all the operators in the program are checked and an initial list is made of those operations which can be performed. From then on the program interpretation is a cycle of the following three steps:

1. Do all the operations on the list in parallel. For each operation reset its input variables to undefined for

those inputs so designated. If the list is empty, program execution terminates.

2. Send the calculated answers on to their receiving operators and make a new list of these receiving operators.

3. Check each operator on the new list, and retain only those operations whose inputs indicate that the operations can actually be done; i.e., have no undefined inputs except those designated as ignored. Go back to step 1.

Similar methods have been used in logic circuit simulators [30] and in an executive system [7].

The fundamental activation process is best thought of as a continual scanning of all operators in the program to find those which should be activated next. The interpreter does not follow a flow path or paths and execute a predetermined sequence of operations. The values of the operator input variables, including flow, determine whether any particular operator is to be activated. Since flow is only a dummy data variable, there is nothing but data flow through the process description.

CHAPTER V

THE EXPERIMENTAL SYSTEM

To test the various ideas developed about graphical programming an experimental system has been created for the TX-2 Computer at the M.I.T. Lincoln Laboratory. At the start of this work the possibility of using the original SKETCHPAD programs was considered and rejected. Projects involving coupling a circuit simulator and a program flow-chart maker to SKETCHPAD had been tried and proved unsuccessful. SKETCHPAD's internal data structure and programs are so rigid that it is inconvenient to make a geometrical entity have non-geometric meaning. Furthermore, a new written language for creating graphics programs had been developed and made a fresh start particularly attractive. The CORAL language and data storage system greatly facilitated the development of the new graphical programming system. CORAL was created at Lincoln Laboratory as a direct result of experience gained with SKETCHPAD, SKETCHPAD III, and the work of Roberts [24] on three-dimensional representations and hidden line removal. This experience indicated a need for a general-purpose method of handling inter-related data in a computer memory and for a useful language to program actions upon the data. Appendix C contains a brief description of the data structure and language features of CORAL. Since its development CORAL has been used in a translator, a time-sharing executive, and a text editor as well as in graphics programs. CORAL gives one the benefit of the best features of both list and table data

structures.

In a facility for graphical communication about procedures an exotic drawing capability is not needed. The whole intent is to allow the placement and connection of symbols in a two-dimensional pictorial language which will describe some process. We will be concerned more with the connection and topology of the picture than with its geometrical layout. Consequently, as implemented, the system has no constraint facilities, no curve capabilities, and a relatively simple control language. The new system is not intended for designing mechanical parts or showing three-dimensional drawings of solid objects with hidden lines removed. One can draw an arbitrary straight line symbol and assign a meaning to it. Defined symbols may then be used as part of a graphical procedure description. After the assignment of appropriate variable values, the graphically described procedure may be executed.

BASIC GRAPHIC CAPABILITIES

In constructing the new system many SKETCHPAD features were adopted directly and many omitted. Only straight lines can be drawn since circles or curves could only make a pictorial program look better and would not add anything fundamental. One draws a line (just as with SKETCHPAD) by positioning the light-pen, giving a DRAW command, and then moving the light-pen and attached "rubber band" line to the desired terminal location. There the line may be terminated and, if desired, another line started by giving another DRAW command. A symbol can be created in any picture by making a reference to another picture causing a replica of the reference picture to appear in the picture under construction. This replica is treated as a unit

and may be moved, rotated, and deleted as a whole. This recursive sub-picture capability adopted from SKETCHPAD permits us to define whatever shape of symbol we like for use in a graphical program.

Geometrical constraints are not included in the new system. Instead, a "grid feature" makes picture parts lie neatly on a grid of discrete positions. The size of the grid can be changed as necessary. One can easily draw a very neat symbol on the "squared paper" which the grid feature provides. Since the system will only copy and use a replica of the symbol in a pictorial program, the dynamic action of SKETCHPAD's constraints is not required.

One of SKETCHPAD's limitations is that only one picture can be seen at a time. The new system allows any number of pictures to be seen together on the computer display. In fact, the computer display area may be broken up into any number of rectangular "viewing windows" each of which may have any number of pictures assigned. To create a new viewing window one draws a rectangle at the desired location on the screen and then orders that it be made into a window. Thereafter, until deleted, this window acts as a boundary for whatever pictures are "placed behind it". The entire scope face is considered as merely one of many possible windows. The viewing transformations which control how a picture is positioned under a window are separate for each picture-to-window assignment. Thus it is possible to have an overall view of a picture in one window and a zoomed-in closeup of a portion of that picture in another window.

The presence of a variety of views on the display face introduces a new complexity. If we say DRAW to the system when there are two windows on the display — each showing two pictures — in which picture and with which

transformation do we wish to draw? It is necessary to designate which picture and transformation should be used. In actual practice explicit boundaries for the pictures are rarely needed since usually only the total scope window is used. Pictures in it are separated by putting them in different corners.

A dictionary facility is provided which makes it possible to name pictures and windows; the name of a picture is used when calling up a replica symbol for use in another picture. In addition, a small number of system controls work through typed inputs and the dictionary facility. There is, for instance, a typed command which says "Delete the entity named".

SYSTEM CONTROL LANGUAGE

The control language is the medium via which a user communicates his commands to the computer. Such a language should be as easy as possible to use. The discussion here will be confined to a general description of the control features and their method of implementation. Appendix A contains the details of the experimental system's controls.

One of the difficulties with SKETCHPAD is the fact that its control functions are selected by a large number of buttons. This forces a user to look away from the displayed picture to find the appropriate button he wishes to push. The new system uses "light buttons" on the scope face itself to replace a large number of function buttons which must be pushed. There are only two physical buttons to push, and these are foot-operated. Thus the user need put down the light-pen only when he wishes to use the typewriter keyboard. In all other cases the light-pen and foot pedals are used to control what action will be taken. Instead of pressing a DRAW button, a user

of the new system points to a DRAW light button and presses one pedal. He thus places the system into a draw mode and then he can draw any number of lines by moving the light-pen and pressing the pedal. The pedal acts as a universal button, while the light buttons select what label and function the universal pedal shall have. The second pedal destroys the label assigned to the first pedal and requires another choice of function to be made.

It is very useful to think of the control language inputs causing transitions between system states [9]. Each state causes an associated action to be performed whenever the system is in that state. Some states are unstable and give way to another state after their actions have been done only once; other states are stable and have repetitive actions. As an illustration, let us consider the state changes involved in drawing a line. When the DRAW light button is selected, the system is set to a waiting state with a null action. Pen tracking is required before a line can be drawn. Consequently, the initiation of tracking causes a transition to another state, again with a null action. A push of the action pedal now causes a state change to an unstable one. The action of creating new end points and a line is done only once before the unstable state changes to a state which moves the new line as the light-pen is moved. The action for this state is done repeatedly until the termination of tracking returns the system to the previous waiting state.

It took a long time to realize that some kind of structure, no matter how simple, is required for the system's controller. The controller has a state table which describes the transitions that may be caused by the inputs of the control language. When this kind of control was adopted the

system factored neatly into the state controller, a large number of action subroutines, and an input collector which provides inputs to the controller from the pedals, light-pen, and keyboard. Until this simple order was imposed the system controls were a patched jumble of special features. With a state transition controller it is very easy to specify that a certain sequence of events is required before a series of actions is performed. Experience has shown that the control language should be treated by means of some formalized scheme of appropriate complexity.

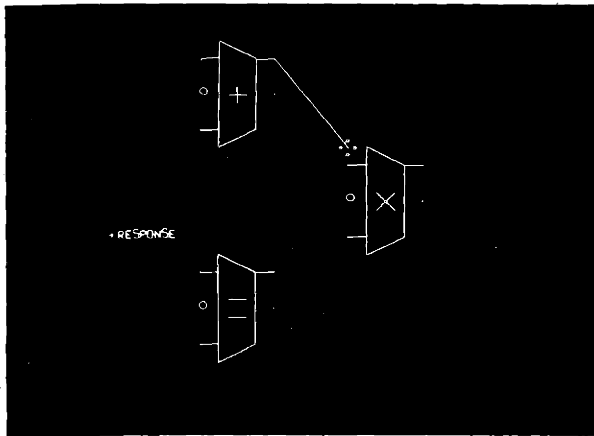
SPECIAL FEATURES

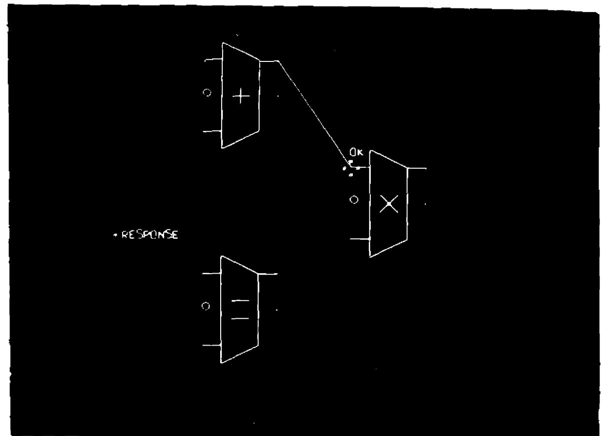
Not unexpectedly, a number of special capabilities had to be provided in the experimental system. One of these (the declaration of symbol terminals and data types) has already been mentioned. After a master picture for a new symbol has been drawn the input and output terminals for that symbol must be designated. This is accomplished by a set of declaration statements within the control language. If the symbol is to have a primitive meaning the data type for each terminal is chosen (by means of the light-pen) from a list of allowed data types. If the new symbol is to have a graphically defined meaning then the data types for the new symbol's terminals are automatically determined when the correspondence to terminals in the definition picture is established. Thus a new symbol which represents a graphical definition of "average" will automatically have numerical terminals since its definition picture contains the numerical operations of add and divide.

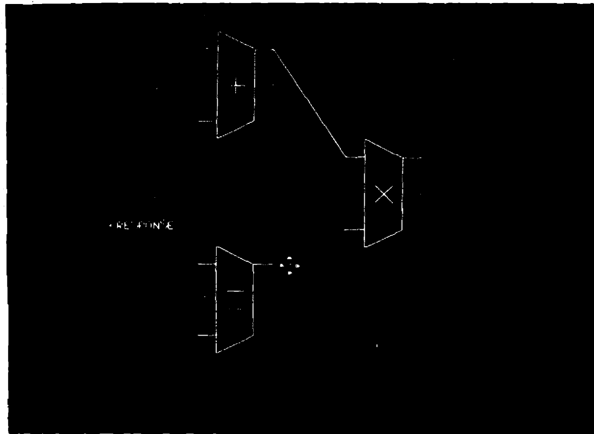
Because the picture of a procedure represents something non-geometrical there are some operations which would be legitimate from an exclusively

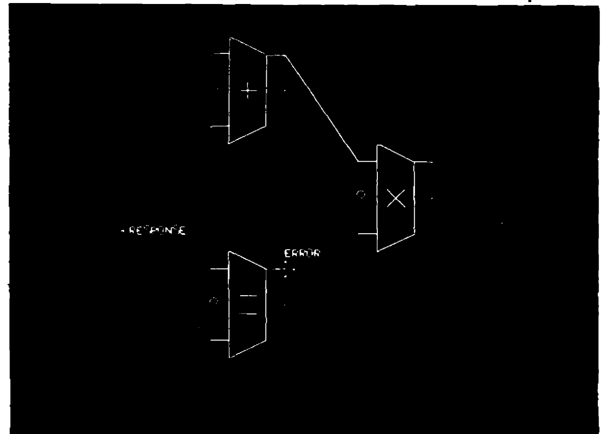
graphical standpoint which are really incorrect. The connection of dissimilar kinds of terminals is an example which has already been mentioned. The system will not permit certain incorrect actions or actions for which insufficient information is available. For instance, suppose we try to connect a BOOLEAN terminal to an INTEGER terminal. We position the light-pen over the first terminal and give a DRAW command. A line is created and stretches between the first terminal and the pen position. When the pen is positioned over the second terminal and the stop drawing signal (pen flick to terminate tracking) is given, the line will be deleted and for a short time the word "ERROR" will appear at the location of the second terminal. A user is also prevented from asking for a symbol replica without indicating the name of the desired symbol. The positive response of "ERROR" immediately reminds one that something was forgotten. The system also responds to errors in assigning terminal types and variable values.

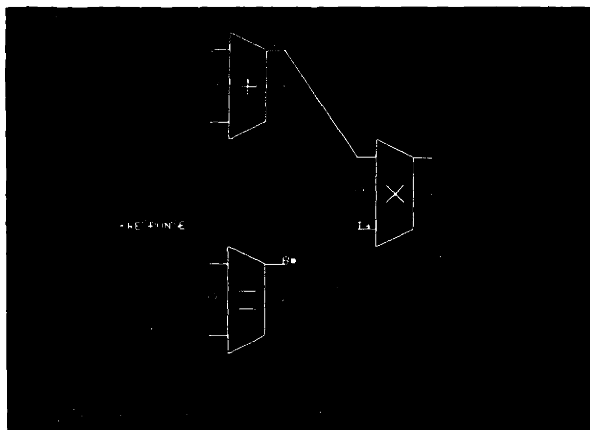
A positive response provides an indication that the system has complied with a correct command. If one connects two terminals and the terminal types are compatible, the connection will be made, and a reassuring "OK" will appear momentarily. Figure 5.1 shows how this response takes place. The top two pictures show a correct connection being drawn between two numerical terminals. The next two pictures show how a connection is rejected between two incompatible terminals. The fifth picture shows how a debugging feature can show that the two terminals in question are BOOLEAN-OUTPUT (B*) and INTEGER-INPUT (I →) and hence incompatible. This and other debugging features will be covered later. It is quite clear that a rapid response to errors and reinforcing response to correct actions is valuable. A more











Control Responses

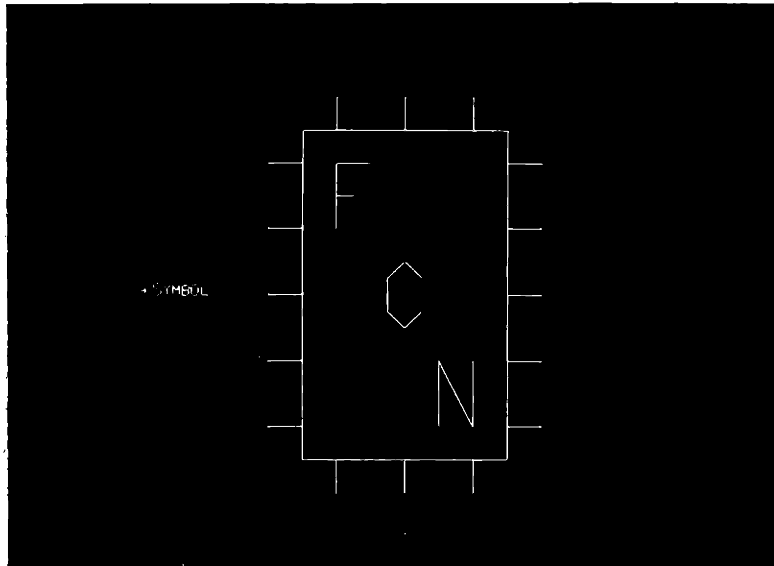
Figure 5.1

detailed diagnostic reply to incorrect actions is desirable and should be easy to provide when time permits.

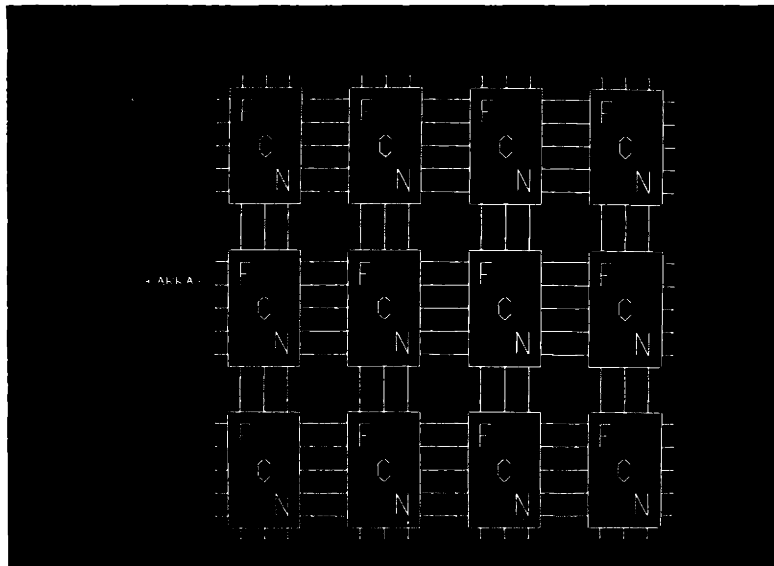
Terminal data types play an important role in the limited positional connection capability provided within the system. Often it is awkward to draw explicit connecting lines between symbol terminals. It would be much more convenient to place appropriate symbol terminals near each other by correctly positioning their symbols, and then to let the system create the explicit connections between terminals. Figure 5.2 illustrates a typical case. All of the connections in the array of symbols were made automatically by the machine. A symbol being moved on the display surface may momentarily have its terminals placed into a positional connection mode. At that time if any compatible terminal on another symbol is within a certain distance of a terminal of the moving symbol, the terminals will be explicitly connected. The assignment of data types to terminals makes this kind of automatic connection reasonable since only compatible terminals will be connected and not every possible pair of terminals. Because of data types the allowable distance separating terminals may be fairly large and yet the automatic connection feature will still work well with only a rare undesired connection occurring.

Variable values are entered by keyboard and appear on the scope. The user may move the text of the value with the light-pen. When pen tracking is terminated over a variable, it is assigned the value being moved around. The value text disappears and the system responds with a momentary "OK". An "ERROR" response occurs for an incorrect assignment.

The final special feature of the experimental system permits one to



Basic Symbol



Connected Symbols

Automatically Connected Symbols

Figure 5.2

choose the input terminal options for each symbol. All input terminals, except flow input terminals, are created with the data retaining option when a symbol replica is called up for use. Flow inputs are created with the data resetting option. There are declaration commands in the control language which allow a user to point to an input terminal and change it to whatever option he desires.

USING A PICTURE AS A PROGRAM

We will now shift attention away from the purely graphic aspects of the experimental system and consider using a picture as a program. The graphic elements of the system now must have a dual identity. A point is more than just a geometrical entity since it may represent a variable which has a value. Thus the representation of a point must contain some indication of the value assigned to the variable as well as the position of the point. Similarly, a complex symbol used in a picture must provide a semantic reference to that symbols meaning as well as a geometric reference for defining the symbol's shape. The line element serves as a connector both of points in geometric contexts and of variables in program contexts.

The internal representation of a pictorially described program is used in two principal ways:

1. As data for a display procedure written in machine-code. The resulting output is the picture presented on the display face.

2. As a process description which will guide and control a programmed interpreter. The action which results from this guiding of an interpreter will act on other, quite disjoint, data.

The internal data structure must be suitable for both of its uses.

Using the picture program's internal representation as instructions for an interpreter is not a difficult task. The interpreter must know how to find the program information in the data structure just as the display program must know how to find display information. Beyond that, the interpreter must determine that an operator should be activated, obtain the appropriate data values from the data structure, perform the operation, and put the answers back into the data structure. The mixing of pictorial and program information does not matter to either the interpreter or drawing system. Since picture and program are combined in one structure whatever is done to the picture is automatically done to the program. Deleting a line removes the indicated program connection as well. The picture is a visual model used for constructing and changing the program.

Sometimes picture and program concepts merge. For example, we would normally think of displaying a spot at a location specified by the graphic coordinates of a point. For display purposes the value of the variable need never be considered. It may be convenient, however, to substitute a text display of the variable's value for the spot. The graphic coordinates still determine the location of the point-variable's display be it text or spot. Similarly, the syntactic properties (data type) may be shown as text. In this way information about the variable represented by the point can be presented to the user. SKETCHPAD had a rather rigid separation of graphical and non-graphical concepts which made it unsuitable for this kind of use.

DEBUGGING

Mistakes in graphical as in written programming are likely to occur. The system provides some help in preventing obvious errors but cannot detect a logical error in a procedure. However, just having a two-dimensional version of a procedure spread out for examination has proved very convenient. When a graphical program on the screen is operating the elements which are actively being performed blink. It is most instructive to watch the activity progress through a program along many parallel paths. One can display a graphical sub-program as well as higher level programs which use it. When the activity in the upper program reaches the sub-program symbol it will flash. The sub-program will then be activated, and its symbols will flash showing the path of activity. Observation of activity in programs may be conducted to whatever depth of definition layering can be seen on the display.

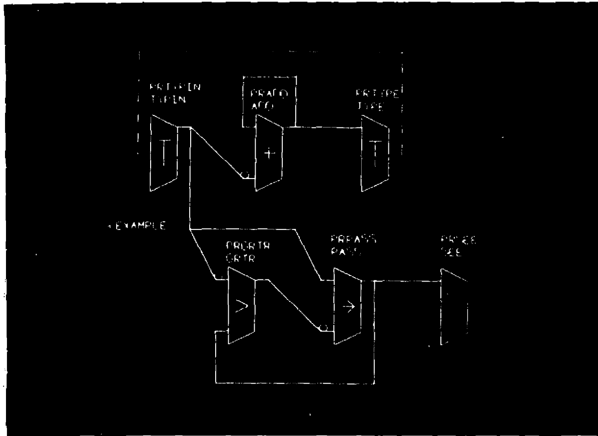
Program activity may be interrupted and various actions taken before resuming program execution. It is possible to examine intermediate results, to assign new values to variables, and even to change the program structure within reasonable limits. Execution of the graphical program may then be resumed and will continue, using whatever changes were made. To examine the variables and other items of interest in a graphical program, a special set of controls has been included in the experimental system.

In one of the special debugging modes it is possible to see on the display the values of any or all of the visible variables. As mentioned previously the geometrical point display is replaced with text which shows the value of the variable. Figure 1.6 in the introductory example of Chapter I showed this feature in operation. In another mode it is similarly

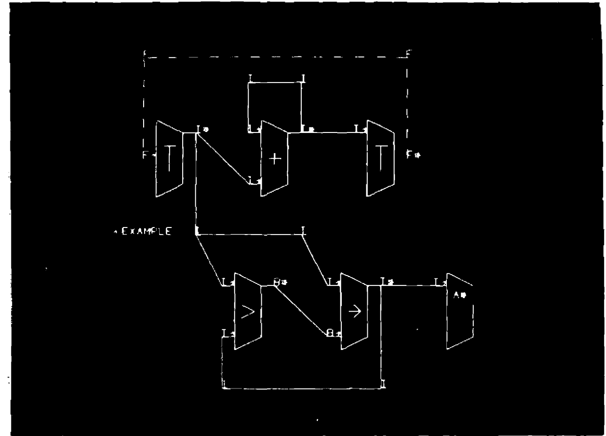
possible to see the syntactic properties (data type and input or output terminal designation) of any point on the display. In yet another mode, the name of the master picture of any complex symbol may be determined as well as the name of the picture which defines the meaning of the symbol. These two features are shown in the top illustrations of Figure 5.3. The bottom two views in that figure show how the corresponding points of symbol and definition picture may be matched. If a symbol and the picture defining its meaning are both visible on the display, the computer can be ordered to show a dashed line between corresponding points. The dashed line in this case does not represent flow. It is possible to analyze an unknown program with the examination capabilities just described. The names of all symbols and their defining pictures may be obtained, and what each terminal on a symbol means may be found by comparison to the definition picture.

One can run an individual definition picture to check its operation. Input data values may be assigned easily and output answers may be obtained and examined visually. Watching activity progress through a program is very useful in verifying correct operation. Thus a user may be cautious and check out program pieces as they are defined, or he may be bold and only look for errors when they occur.

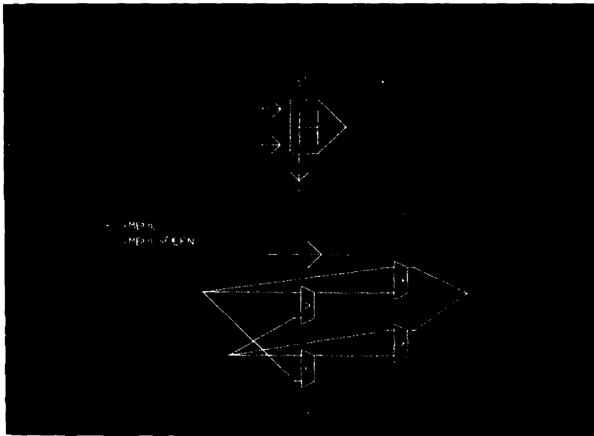
It is not necessary to define a meaning for an operator symbol before that symbol can be used to draw a procedure. Thus one can program from the "top down". An operator which is to perform some function can be drawn and used while leaving the problem of specifying its meaning to a later convenient time.

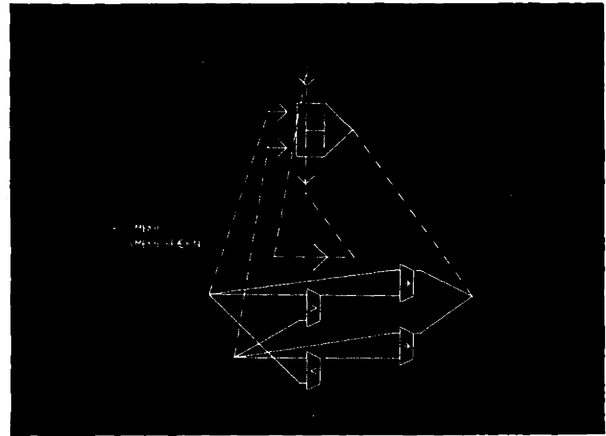


Symbol Definition Names



Terminal Syntactic Properties





Corresponding Points On A Symbol And Its Definition

Debugging Features
Figure 5.3

Since the graphical program is laid out in two-dimensions and since data terminals are readily available, it seems reasonable to provide "data probes" which may be attached to data terminals of interest. These probes provide a means of examining variable values. When a user is checking over a program he may attach a data probe much as he would attach an oscilloscope lead to a circuit. No special effort is required to provide this kind of attachable data probe since the user need only call up a suitable primitive operation symbol and graphically connect it to whatever data terminal he chooses. The added operation then becomes an integral part of the program. The information necessary to trigger the probe operator can be passed down the data lead and no flow connection is needed. The standard system primitives of SEE and TYPE have been used for just this purpose. Other operators which plot data values can be defined as necessary.*

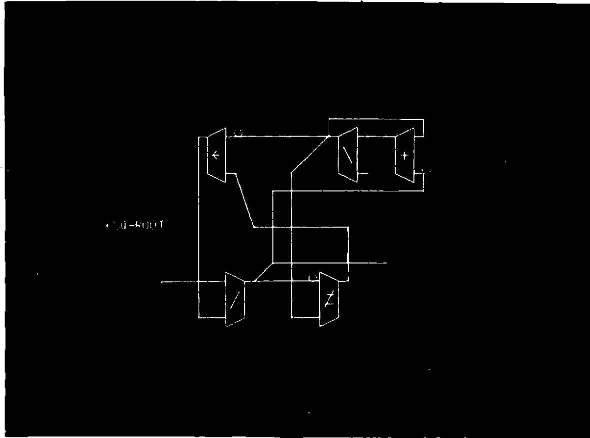
The first picture of Figure 5.4 shows a program for finding the square root of a number. In the second picture a data probe has been added to examine the value of a variable. The third picture shows a value obtained sometime during program operation. A different attached operator is shown in the fourth picture. This operator is a break point, and when activated it interrupts the operation of the interpreter thus allowing one to examine the program with the debugging features already covered.

GENERAL REMARKS

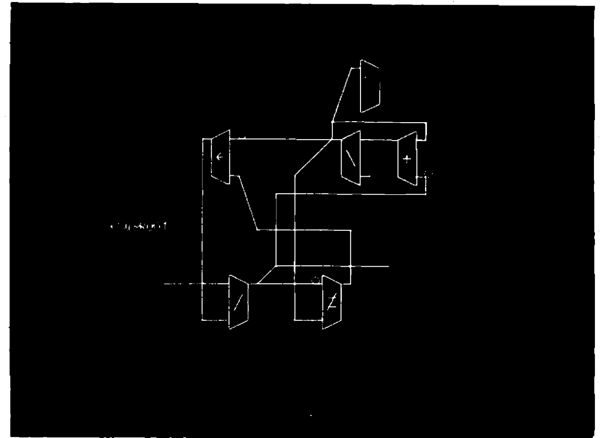
It is unreasonable to expect that graphical programming will be really

---

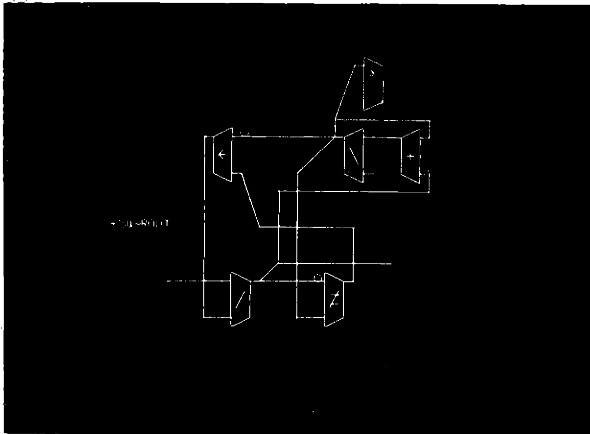
* Data probes were suggested by T. G. Stockham, Jr. [27].



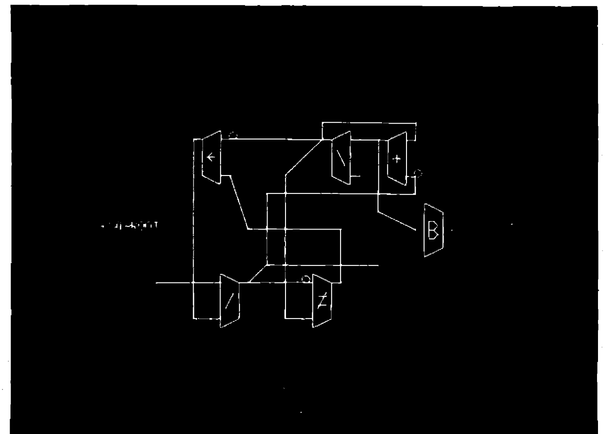
Basic Program



Probe Attached



Probe Operating



Trap Operator Attached

Data Probe And Trap Operators
Figure 5.4

practical until many man-years of effort are expended. This experimental system has only scratched the surface. The indications are, however, that graphical programming should be worthwhile. The details of the experimental system are less important than some general results which have become clear. These comments are applicable to interactive graphical systems of any kind.

First, the importance of a close mix of geometric and non-geometric properties has been shown. Whenever picture parts are to have meanings other than purely geometric ones, the abstractions which the picture parts represent may in some degree affect the graphic behavior of the parts. Second, the importance of a good written programming language for creating the system programs cannot be overemphasized. Without CORAL the task of creating the system would have been immeasurably harder. Third, some formalized scheme of control is a great asset. The experimental system uses a kind of finite state machine as its simple control scheme. With the control form fixed, the system factored into managable pieces; without any formalized control scheme, the controls and system were chaotic.

CHAPTER VI

A TWO-DIMENSIONAL LANGUAGE SCHEME

Before we apply the term "Graphical Programming Language" to pictorial procedures, let us examine what we mean by a programming language. ALGOL and LISP may be classified as two specific members of a class known as "Written Programming Languages". Every language in this class will share some common features with other members and will differ in some way. Written programming languages usually include the basic notion of executing statements in sequential order and share the syntactic restrictions of a one-dimensional source string. The pictorial programs described in this report belong to a different class, "Graphical Programming Languages". Two graphical programming languages might have radically different symbols with different meanings and yet share a common two-dimensional form, basic syntax rules, and operating conventions.

The experimental system just described provides a graphical programming language scheme or framework which can host a variety of specific languages, each of which will have certain definite symbols and operations. The experimental system allows any symbol set the user creates and also may have its primitive operations changed. The examples presented so far have involved numerical calculations. The next section shows how different non-numerical operations may be fitted into the two-dimensional language scheme.

GRAPHICAL ASSOCIATIVE PROGRAMMING LANGUAGE

The basic graphical programming scheme was combined with a special set of primitive operations which were developed to do associative processing as part of another project. The result is a graphical means of connecting together associative operations which create, erase, and search for pieces of information in a data base. Let us first consider the associative primitives and written associative language developed by Feldman [6].

Feldman was concerned with how to make use of an associative memory system should a large and practical one be developed. He created a written language for programming actions in an associative memory. He assumes that the information of interest can be divided into objects, attributes, values, and associations. An association is always a triplet of the form "ATTRIBUTE of OBJECT is VALUE". Statements in the associative programming language are made up of command words and associations. For example, the statement

SET Brother(Mary) = John

will cause the relation that Mary's brother is John to be entered into the associative memory. This information may be retrieved by asking a number of different questions. The statement

TYPE Brother(Mary) = X

will search memory and type out all of Mary's brothers, including John. The written programming language allows dummy variables to be used in a triplet as in the example directly above. The statement

ERASE Brother(X) = Y

will destroy all information in the associative memory that uses brother as an attribute.

The written associative language contains other control words which allow complex statements to be constructed. For example,

FOR Cousins(Mary) = X WHERE Sex(X) = Male TYPE Home Address(X) = Y;

will produce the home addresses of all of Mary's male cousins if that information is contained in the data base. In an associative memory which contains information about several generations of a large family we may insert the relationship "UNCLE" which is not now directly included as follows:

FOR Father(X) = Y WHERE Father(Y) = Z AND Father(W) = Z AND
Sex(W) = Male SET Uncle(X) = W; FOR Father(X) = Y AND
Uncle(X) = Y ERASE Uncle(X) = Y;

To create a working system which could use this language on a reasonably large data base, a simulated associative memory system was developed. Hash-coding provides quick entry to the data base and list features are included to make answering reverse questions possible. Primitive operations were constructed as machine code routines which search for, set, and erase the data triplets in the simulated associative memory. A compiler takes statements in the written associative programming language and creates a program of calls on the primitive operations. Executing this program then performs the desired actions in the associative memory.

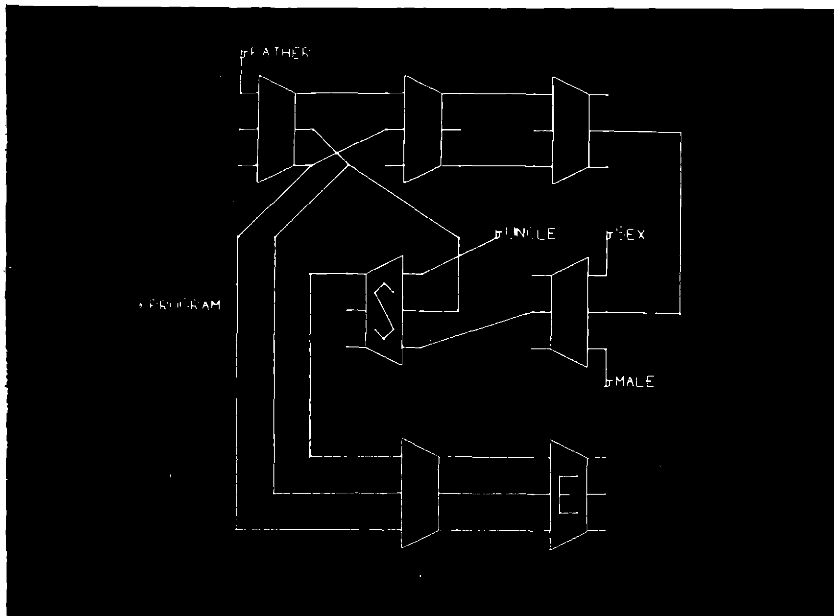
To unite the two systems (graphic and associative) was easy; it took a good summer assistant less than two months. This includes the time required to learn the TX-2 Computer, to learn about both the graphic and associative systems, and to produce a working merged system. Symbols may be drawn and defined to mean the special primitives. The standard graphic activation conventions work correctly to control the operation of the connected

associative operators. Figure 6.1 inserts the relationship UNCLE into the associative memory and is the graphical equivalent of the earlier written example. The variables represented by the data connections are not numerical, but instead are sets of named attributes, objects, and values.

It is not immediately apparent that the graphical program format is any better or clearer than the written one. One of the purposes in merging the two systems was to permit a side-by-side evaluation of the two language forms. In the figure the set and erase operators are indicated with letters while the search operations are unlabelled. Input terminals which have no data value shown correspond to the dummy variables of the written form. After a search in the memory, each unlabelled operator's output variables are defined by the set of associations in the memory which match the specified inputs. The first operator has inputs vertically corresponding to Father(X) = Y. Its outputs are derived from all associations containing Father as an attribute. The middle terminals on the second and third operators represent people who have a common father represented by the connecting bottom line. The fourth operator removes females and its output is used with the one representing the original niece or nephew to insert explicit associations about uncles. The last two operators remove an uncle relationship whenever the two people are really father and child.

GRAPHICAL LANGUAGE SCHEME

The relevant results of the preceding discussion are more than the specific details of one particular graphically formatted associative programming language. The example is intended to illustrate that a basic



Graphical Associative Programming Language

Figure 6.1

framework for two-dimensional procedure-oriented languages has been developed. Throughout the previous discussions of activation conventions and symbol connections, the actual primitive operations used were irrelevant. The scheme of a procedure description is concerned with the operation of some arbitrary actions and how data is passed between them. The framework of symbols, symbol connections, and activation conventions is quite independent of the meanings of the individual parts. This framework provides the formalism by which complex meanings are created from the simpler meanings of the component parts. The conventions chosen and presented here are not the only ones possible just as the usual sequential conventions are not the only ones possible for written languages. They are, however, a simple and useful set which may be used for many different tasks.

Languages created within the experimental graphical programming system will all use explicit connection of symbols. The symbols may be arbitrarily drawn, but they must have terminals for connection to other symbols. The particular syntax properties chosen for any terminal must conform to the basic syntax rules allowed. In creating a specific language the primitive operations for a particular application must be chosen and placed into the general language framework.

Providing primitive actions for the experimental system has been a slow process of adding a new one every now and then. Much of the early experimentation was done with boolean operations. These simple operators allowed full concentration on the graphical procedure frame of symbols and conventions. When operation of the basic graphic and interpretive features was satisfactory, more attention was given to a richer set of primitive

430

operations. To add a new primitive one must declare the written name by which it is to be known in the system, and provide a subroutine to do the required action. To use a new primitive one must draw a symbol for it.

Adding a new primitive operation is not to be confused with providing a new symbol with a semantic definition graphically constructed out of already defined parts. Adding a new primitive requires action by someone relatively expert in the inner working of the graphical programming system. He must place a new subroutine in the system's permanent records. On the other hand, creating a new symbol and giving it a graphically defined meaning can be done on-line by any user.

Appendix B contains a description of the primitive operations that have been inserted into the experimental system at the time of this writing. The list is by no means complete for it appears likely that the addition of primitives may go on indefinitely. In one sense the only primitives needed are boolean since with them it is theoretically possible to build up on-line whatever else might be desired. Practical efficiency considerations make many other primitive operations necessary if any useful applications of graphical programming are to be made.

LANGUAGE SCHEME APPLICATIONS

There are three direct characteristics of an interactive graphical programming system which indicate potential applications: the two-dimensional nature of the pictorial programming language, the ease with which parallel processing may be specified, and the fact that an interactive graphical system is available.

Many simulation problems fit easily into a two-dimensional source language. Rather than trying to describe a collection of filter elements, amplifiers, and modulators with a written language, it makes good sense to draw a schematic of the desired arrangement. When a picture describing the desired connection of active elements is complete the picture may be activated and the operation of the network simulated. Inserting already existing simulation primitives into a graphical programming framework is particularly attractive because little effort is involved. Thus we can envision a graphically programmed form of a language like BLODI [12]. The inclusion of an existing set of TX-2 primitives for a BLODI-like language is planned for the near future.

The kind of application just described may also make use of parallel processing. If the source picture describing the problem network so indicates, some of the simulation operations may be performed together. Other applications involving parallel processes should prove suited to a graphical description. For example, we might use a graphical format for programming the actions of a remotely controlled vehicle. Assume that we must specify how an automatic forklift cart is to behave in an automated warehouse. The cart has a number of actions which it can perform simultaneously; i.e., move, turn, raise forklift, tilt back. The two-dimensional source language could be used for specifying that a 90 degree turn and forklift lowering should be accomplished together. After both are completed the cart should move forward and engage a pallet. A simultaneous backing, lifting, and tilting can then be accomplished. Any procedure which involves parallel operations is a candidate for a graphical description.

On-line graphical programming suggests another broad area of application. Since an interactive drawing system is provided, let us use it to draw pictures representing data as well as program. In the previous example of the forklift cart, besides drawing a program of operations for the cart to perform, it might be useful to have a map of the warehouse available on the computer display. The graphical symbol representing a move operation could then be connected to graphical data (the warehouse map) to indicate where the cart should move. A great deal of care was taken in the previous chapters to separate pictures into those representing procedure and those to be used as data. However, a graphical program may work on data of any sort.

Some experience with combinations of pictorial data and pictorial procedures has been obtained. The experimental system contains primitives which act on graphic data. It is possible to draw a picture to be used as data, and to draw another picture which is a program specifying how the data picture shall change. Moving symbols and walking stick figures, for example, have been programmed. When using the graphical programming system with graphic data we get many of the graphic primitives free. The same routines used by the system to make and move program pictures may be called by the interpreter to act on data pictures. Program and data can even be combined into the same picture; a program when executed may change its own pictorial description.

CHAPTER VII

CONSIDERATIONS IN COMPILING MACHINE CODE

Past experience with computers has shown that interpretive operation of a program may be quite inefficient. To avoid this difficulty we translate the source language statements into an equivalent sequence of machine code instructions which the computer can execute directly. Can and should this be done for a graphical source language of the kind under discussion?

Let us first consider the problem of creating machine code from a graphical program which has a single explicit flow path designated to control how the procedure operates. All of the standard compiler considerations apply to this case. The graphical program is analogous to a written program and a compiler for this kind of graphical program can be constructed without difficulty. Compiling code for a single processor machine from a sequentially operating program description is a well understood task.

A graphical program, however, may represent a procedure in which a number of parts could be operating in parallel. This is one of the appealing characteristics of the two-dimensional language form. The difficulties in compiling code to improve the efficiency of a graphical program arise from parallelism in the program and in the computer. Depending on the particular circumstances, it may be desirable to keep the parallel paths in the machine code or it may be desirable to remove some or all parallelism and make the procedure operate sequentially.

The actions taken by the compiler will depend in part on the number of processors expected to be available at run-time. The code resulting from compiling some parallel graphical program should be different for the three cases where one, two, or twenty processors were predicted available for object code execution. Parallelism with only one processor available is useless, whereas with twenty processors available a high degree of parallelism is valuable. Multiple processors are a factor which compilers have not yet had to face.

A parallel program in machine code form may be expected to fit a sequential pattern of operation. At a machine instruction level computers almost universally use sequential control. Program instructions are executed from consecutive memory locations unless special jump commands order otherwise. The object code will have a number of parallel segments each consisting of a normal sequential machine code program. How much machine code each segment should contain is a matter of efficiency and depends on the particular circumstances. If the actions involved in assigning processors are efficiently handled by hardware the segments could be as short as one command. If processor assignment is accomplished by a programmed supervisor then the parallel segments must be long enough so that the supervisory overhead time is a reasonably small portion of the total running time.

Other factors may also indirectly affect the length and number of parallel program segments. If a large number of processors are to be active simultaneously, their program segments must reside in the computer's high-speed storage. The size of this storage is a practical limitation on the size and number of parallel segments which can reasonably be expected to

operate together. A reduction in the number of parallel segments in a program could well be accompanied by an increase in segment length since the same total amount of processing must be done. Unfortunately, longer segments take more storage, yielding no obvious net gain. Unneeded parts of a segment may be kept in bulk storage and brought into core as needed. In this way memory swapping efficiencies could become relevant to the decisions required during compilation.

SPECIAL CASES

There is one action which should always be taken to improve the efficiency of a graphical program. Those portions of a graphical program which will operate sequentially should be consolidated and compiled into machine code segments. It should be clear that finding sequential portions of a graphical program is not a difficult task. Examining the interconnecting data lines between operators for branches and checking the input data activation conventions present at each operator will determine those operations which have a single guaranteed successor. Once the sequential portions of a program have been identified there is no difficulty in compiling the code for those portions. We now assume that this step has been accomplished and that the sequential portions have each been redrawn as a single operator in the graphical program.

There are two possible reasons why the remaining operators were not consolidated. First, the activation of an operator may be followed by the activation of more than one succeeding operator; i.e., a new parallel path is needed; or second, an operator's activation may not necessarily follow

from the activation of an earlier operator; i.e., parallel paths join together.

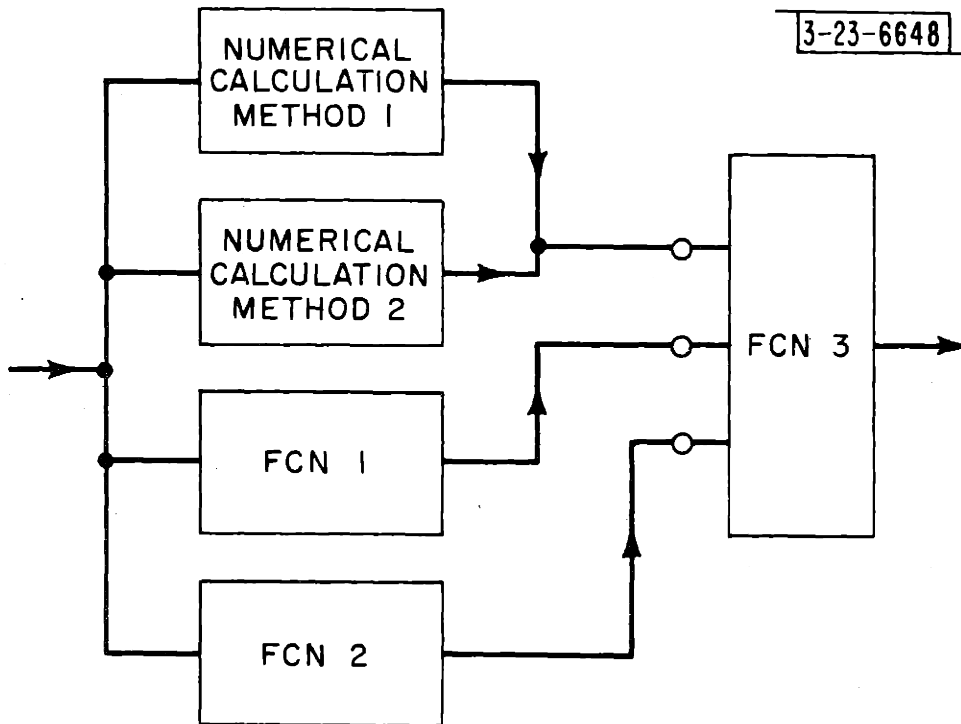
Let us first assume that more processors than will ever be required are available at run-time. The compiled code should retain all of the parallelism present in the graphical program. Compiling machine code for this case is a direct task. Each operation can be translated into a segment of sequential machine code instructions. A fork operation starting new parallel paths and activating new processors must be placed after each piece of code derived from an operator with a branching output. Machine code which performs the join and wait operation must be placed in front of the code for operators which have this input characteristic. The machine code program is a direct mapping of the pictorial program. There are no great difficulties inherent in doing this kind of compilation.

Let us next consider the other extreme of compiling machine code when it is known that only one processor will be available. The fully-parallel machine code just discussed can be used with only one processor, provided the supervisor is able to keep track of the parallelism. The supervisor must be able to simulate the operation of many parallel processors with only one. When simulating parallelism all the operations which would start at a particular time are done in any order and their results and completion times stored. Then the operations which would start at the next largest time are done, and so on. Alternatively, the parallelism may be removed by the compiler and the computation converted from a parallel into a sequential one. The advantage of converting to a sequential process is that no supervisory actions are required when a single processor follows a single explicit flow

path through the program. Unfortunately, it is not always possible to remove the parallelism from a program before it is run, and consequently the parallelism must sometimes be simulated at run-time.

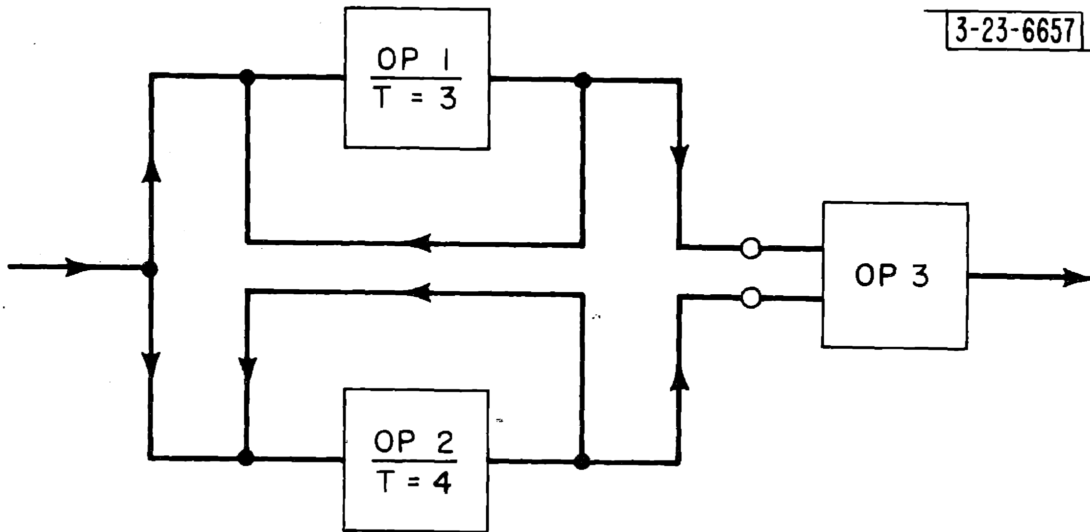
As a first example refer to Figure 7.1, the same figure that was used previously in discussing join operations. Let us assume, as before, that whether method one or method two for the numerical calculation finishes first varies and depends on the particular problem data. Furthermore, let us assume that there is some significant difference between the two answers provided. Without actually doing both operations and seeing how long they take (or doing something equivalent), it is not possible to know which resulting data value to use with FCN 3. Thus a machine with a single processor would have to simulate the parallelism and do both operations to obtain results equivalent to those obtainable with a multi-processor. The duration time of operator execution is crucial here. The fact that the execution durations are not determined until run-time makes compile-time decisions impossible.

Even when the duration of operations is known in advance, there are situations which will be difficult to compile. In Figure 7.2 two independent loops are feeding a third operator. It should be clear that OP 3 will operate every four time intervals. However, sometimes OP 1 will provide two data outputs before OP 3 is again activated. One of the OP 1 outputs will be lost. It would take a very sophisticated compiler to handle this situation without resorting to simulation of the parallelism.



Parallel Operations

Figure 7.1



Independent Operations

Figure 7.2

These two examples have been presented to show that the problem of removing parallelism is not a trivial one. However, we may expect that a large portion of the programs actually drawn will not involve such difficult cases. It is easy to see that a program consisting of a tree of operator connections with appropriate wait restrictions required for each operator's activation may easily be converted into a sequentially operating equivalent program. We hope that many of the programs we might wish to compile would fall into this category.

Let us consider the case where machine code has been created for a certain number of processors, but at run-time a smaller number of processors than planned is available. Operation of the program is unaffected until such time as only one unassigned processor remains available to the supervisor. Then if program operation is to continue, the supervisor must use that one remaining processor to simulate the action of the many missing processors. Changing over from the fairly simple actions of processor assignment to a full simulation of a parallel program is a large shift in supervisor operation. Machine code is usually compiled for a fixed hardware configuration, and changing the available hardware should require recompilation.

SUMMARY

To solve some problem, a man has created a procedure description in a graphical format. This procedure description has as one of its features a natural inclusion of parallelism. The program is to be executed for the user by a computer with certain capabilities. It is possible to take a brute force approach and interpret the graphical program directly on the available

machine. It may be better, however, to examine the program before it is run, and to modify it or change its form, thus providing for more efficient execution later on. How to modify the program before running it depends to a large extent on how it will be run. Therefore, the program which performs any efficiency improving modifications (the compiler) certainly depends on the means of controlling the program's operation (the supervisor or interpreter). Confusion may exist when the term "Compiling" is used out of its well-defined context, i.e., sequential program description to sequential code. Many new options occur when a parallel program is to be used on a parallel processing computer. Decisions about the alternative possibilities must be made before the source language is converted into object code.

CHAPTER VIII

CONCLUSIONS AND RECOMMENDATIONS

The results of the work reported here are a pictorial language form for specifying procedures and a working experimental system for graphical programming. Using the system, a user may draw a picture representing a procedure and then have the computer execute it. The major conclusions to be drawn from these results are as follows:

1. A pictorial program is a natural way of expressing parallel processes. The two-dimensional nature of the language helps in visualizing many things happening at once.

2. The ease of debugging programs, particularly parallel ones, will be enhanced by a pictorial language form. Being able to attach data probes and to see a program run gives one a grasp of detail that is hard to obtain in any other way.

3. A program's execution need not be controlled by the usual explicit sequential flow conventions. The movement of data through a program may determine its operation. A data controlled convention corresponds closely to our intuitive ideas of how a graphical program should operate and also allows parallel programming without explicit flow designations.

4. When a data controlled convention is used to activate a program, explicit flow may be treated as a new but

standard kind of variable. Flow may be introduced into a program as a dummy variable whenever special ordering considerations require it.

5. Choosing a general form for symbols, connection methods, and activation conventions does not determine a single pictorial language; it specifies a class of languages. A particular language is created by defining within the general rules a specific symbol set, specific connection rules, and specific meanings for each symbol.

Finally, the graphic properties of picture parts may be modified by the particular non-pictorial entities they represent. This last conclusion has great implications for the future. We have seen in the experimental system how non-pictorial properties modify the connection of points and the display of points and lines (connecting to a flow terminal makes a line dashed). In a circuit context, a node might display a voltage value, and a resistor symbol might be replaced with a current or dissipated power indication. Future graphic systems must allow a user to make statements about the non-graphic information which his picture represents. The experimental system requires special control commands for choosing the "retain data" or "reset data" options at input terminals. It also includes special programs to make each terminal's display match this non-pictorial information. Since people will use pictures for many different purposes, future control languages must be especially flexible and easily extended. The programmed actions available within a graphics system must be easily extendable too.

This research supports the stated views of others [26] that control language inputs to a graphics system should be processed by some formal scheme.

Having a known form for the control language and its processor makes extending the language a well-defined task. The experimental system has also shown that immediate system responses to correct as well as incorrect commands are very useful and should be included in future systems.

GRAPHICAL PROGRAMMING DRAWBACKS

There are a number of reasons why it might not be convenient to describe procedures graphically to a computer. One limitation is imposed by the physical size of a computer display screen. On even the largest of displays only a relatively small amount of program can be manipulated at one time. If really large programs are involved, it will be impossible to fit all of the program onto the display and may be inconvenient to break up the program into manageably-sized pieces. When a program must be broken into pieces the connections between a visible portion and other hidden portions cannot be explicitly handled. Labels will be needed, leading back to written language features which have so far been avoided.

The introduction of names and labels is a very desirable thing. We have used written languages for so long that we are accustomed to their requirements. So far, the distinctions between written and graphic programs have been emphasized. A combination of written and graphic forms will permit the best features of both to be used. Past experience with large blueprints of schematic wiring diagrams certainly indicates that we often like to use written and picture notations together.

Earlier it was stated that the parallel input capability of our eyes made a non-sequential form of language (pictures) useful. There is a limit

to how much information we can accept at once this way, and this limit may be related to how we use labels. When looking at a picture we are aware of an area only a few inches on a side. Within that area, explicit connections by lines are useful. When connections must extend beyond the boundary of our awareness, the parallel input of the details of the area must be broken and we then look at another area. Thus we observe a number of areas in sequence. Labels are a useful method of connecting between sequentially separated entities. Consequently, we would expect to use them for connections to some distant part of a drawing in preference to a long explicit connection which must be followed and traced out. Even if size restrictions of computer displays were removed, we would still wish to include labels as a method of connection.

The experimental system as presently implemented leaves much to be desired from an efficiency standpoint. Such considerations were relegated to a very minor role during the system's development. Operation of the interpreter is slow, and it consumes much overhead time in relation to short primitive operations. A pictorial program occupies a large amount of core storage in comparison with an equivalent machine code program. One must pay a price for the extra information involved in a picture. Similarly, if a blinking picture which shows how the graphical procedure is being activated is desired, the interpreter must steal the time necessary to do this from the program proper. The experimental system has accomplished its purpose of demonstrating some of the features of graphical programming, even if it does so slowly. However, the system is a long way from providing a realistic and practical capability for general-purpose graphical programming. Its efficiency is satisfactory only in special situations with large primitive operations.

Written notations have been developed for many purposes, and common usage has made these notations very familiar to us. When a useful written notation is available, we probably will not wish to use a graphical one instead. The earlier comparison between written and graphical notations for arithmetic is an example. Graphical languages for programming will find most use where a good written notation is not available or where the problem is not well-suited for a written description. The development of a graphical programming capability must not be considered in the light of replacing written programming. Rather we must extend the options available to a programmer so that he may use whichever method is most convenient for a particular application.

RECOMMENDATIONS FOR FURTHER WORK

In the general field of graphical communication with computers much remains to be done. Better control languages can be developed for interactive systems. New data structures for representing pictures and what they mean are needed. Written programming languages for making graphical system programs can be improved, and new hardware devices for graphics use can be devised. On the theoretical front, a solid foundation for picture syntax needs to be developed. Many picture parsing and pattern recognition problems remain to be solved. Computer graphics efforts everywhere will benefit from a solution to these problems, particularly in areas where input from an already existing picture is desired.

The initial exploratory work reported here may be continued on any number of fronts. Specific applications of the basic graphic programming capability developed may be pursued, and other conventions and graphic programming

notations for classes of languages may be explored. Combined written and graphical notations and systems which use them need to be developed. Compared to the amount of effort invested in written programming languages, virtually nothing has been done yet about graphical programming. It follows, therefore, that almost everything still remains to be done.

One of the claims made for a graphical program is that it is a natural way to express parallel processes. This claim needs to be tested on a computer which has many processors available. The TX-2 on which the experimental system runs has only a single processor and so must simulate the parallelism in a graphical program. All of the considerations of efficiency and compiling in Chapter VII need further development based on the actual requirements imposed by a specific multiple processor computer. Only when this is done will we obtain the ultimate verification of a system which actually works.

The graphical debugging features of the experimental system can be extended and improved. Data probe operators which make graphs or provide other useful kinds of information can be created. Recording probes which would not affect program operation but which would gather statistical information about a program's operation might be useful. A two-dimensional program displayed by the computer offers a new range of possibilities for methods of examining and debugging computer procedures.

The method of executing graphical programs reinforces previously suggested ways of building computer control elements [7]. Rather than having a control element which operates sequentially, it might be useful to consider making a control element based on an associative memory. A simplified model or map of the parallel program would be contained in the special memory. The

control element could then search this memory in parallel to determine which portions of the program were ready for activation and needed a processor assigned. The assignment of processors could be done in parallel also.

This investigation has uncovered very promising evidence that graphically describing procedures to computers will ultimately become useful. Pictorial programs are a natural way to describe parallel operations. Much more work by many people is needed to overcome some of the problems standing in the way of efficient, practical use of this method of programming.

APPENDIX A

SYSTEM CONTROL COMMANDS

TYPED COMMANDS

The following commands are given to the system via the input typewriter. The underlined words are replaced by specific names.

Command	Meaning
☛P PIC IN WDW	Put a picture in the named window. If there is no picture with that name, make one.
☛T PIC FROM WDW	Take the picture from the named window.
☛X NAME	Delete the thing named.
☛D PIC1 AS PIC2	Define picture 1 to mean picture 2.
☛U PIC1 FROM PIC2	Undo the definition of picture 1 as picture 2.
☛W NAME	Make a drawn rectangle into a window boundary and give it a name.
☛G MODE	Change the grid feature. Modes are ON, OFF, BIGGER(x2), SMALLER(x.5).
☛S PIC	Start named picture running; i.e. consider it as a program and execute it.
☛C	Continue executing the pictorial program after an interruption.

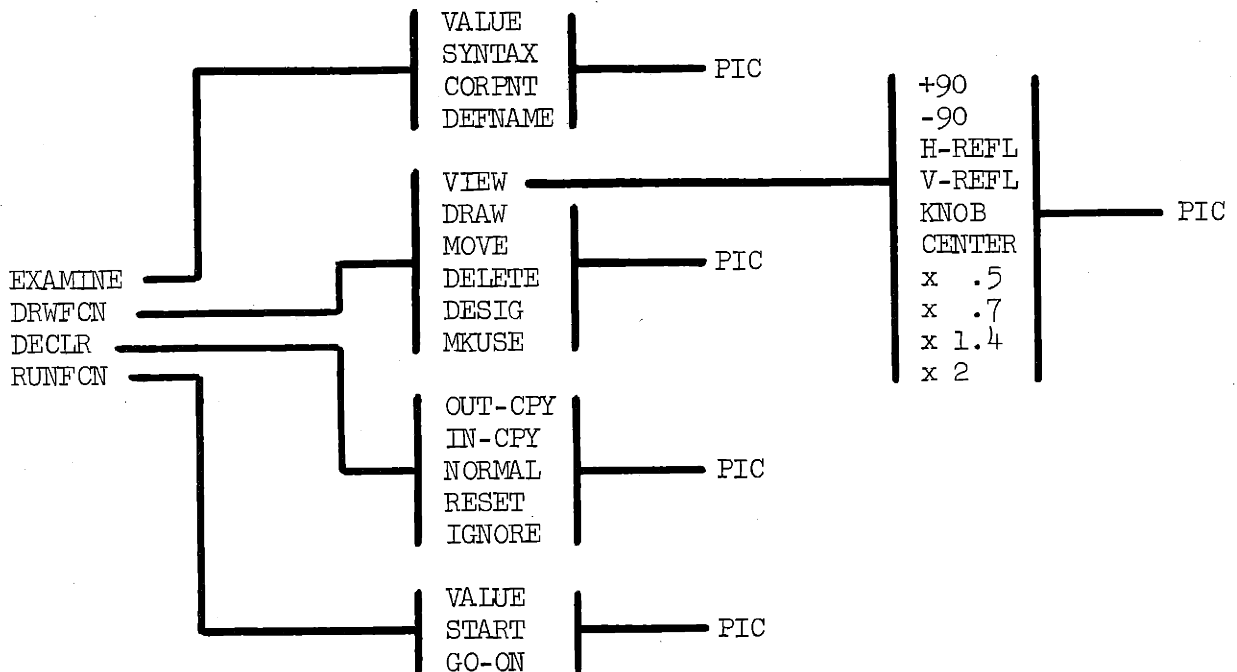
LIGHT BUTTON COMMANDS

The remaining system control features are selected by light buttons and foot pedals. Two pedals (or push buttons if desired) provide activation signals. One signal is for moving along the tree of selection options below and then actually doing the selected action. The other button is a stop button and returns control back down the tree. While the light buttons are displayed, the picture on the screen vanishes. When a picture is seen the system is in a particular state and the action pedal provides a signal causing the chosen action to be performed.

a. The action button selects functions → and then does the selected function.

b. The return button moves back ← .

c. PIC indicates that the picture is seen on the display, and the system is in the last selected function mode.



DESCRIPTION OF LIGHT BUTTON COMMANDS

A. EXAMINE

Select debugging set of commands.

1. VALUE

   Examine value of variables by making the value display on scope.

2. SYNTAX

   Examine syntax of variables by making syntax description display on scope.

3. CORPNT

   Connect a dashed line between a symbol terminal and the corresponding point in the symbol definition picture.

4. DEFNAME

   Show picture name and definition picture name of a symbol.

B. DRWFCN

Select drawing set of commands.

1. VIEW

   Select set of options for changing viewpoint of picture on the scope. If pen not tracking and nothing seen, view changes are centered in the designated window and occur to the designated picture. If the pen is tracking, view changes are centered on the pen and occur to the designated picture. If the pen is not tracking and a symbol is seen, the change occurs only to that symbol regardless of what picture it is in.

   a. +90 — Rotate ninety degrees clockwise.

   b. -90 — Rotate ninety degrees counterclockwise.

   c. H-REFL — Reflect horizontally about a vertical axis.

   d. V-REFL — Reflect vertically about a horizontal axis.

   e. KNOB — Obtain viewing transformation changes from knobs.

   f. CENTER — Make spot under the pen be centered in the designated window. Pen must be tracking.

   g. x .5 — Shrink by scale factor.

h. x .7 – Shrink by scale factor.

i. x 1.4 – Expand by scale factor.

j. x 2 – Expand by scale factor.

2. DRAW

Draw a rubber band line from initial pen location until terminated. If terminated on a point and syntax is OK, merge points.

3. MOVE

Move whatever is seen by pen when the action signal (from pedal) is given.

4. DELETE

Delete whatever is seen by the pen.

5. DESIG

Designate a picture and a window as the current ones. Picture parts then added will belong to the designated picture and the transformation associated with the designated window will be used.

6. MKUSE

Make a copy of the symbol seen, or if no symbol is seen make a new symbol (use) of the picture last named.

C. DECLR

Select the declaration set of commands.

1. OUT-CPY

Declare a point as an output terminal for a symbol. Copy its syntax properties from the corresponding point in the symbol definition picture. Alternatively, copy its syntax properties from a list of allowable data types.

2. IN-CPY

Declare a point as an input terminal for a symbol. Copy its syntax properties as above.

3. NORMAL

Select the normal data-retaining option for an input terminal.

4. RESET

Select the data reset-to-undefined option for an input terminal.

5. IGNORE

Select the ignore-an-undefined-input option for an input terminal.

D. RUNFCN

Select the run-a-program set of commands.

1. VALUE

Type in a variable value. Upon starting tracking the value will appear just above the pen tracking cross. Assign the value to a variable by terminating tracking while seeing a point.

2. START

Start a pictorial program running by pointing to its name on the scope.

3. GO-ON

Continue on with graphical program execution after an interruption.

Note: There are also typed commands to accomplish the START and GO-ON functions.

APPENDIX B

EXPERIMENTAL SYSTEM PRIMITIVE OPERATIONS
(As of November 1965)

The list which follows contains the primitive operations included in the experimental system up through November, 1965. This list is not complete since new primitives may easily be added and will be as the need arises. The symbols used to represent primitive operations are not fixed. A user may define any symbol he wishes and then make it represent any one of the primitives. Several different symbols may each be defined as the same primitive. One can then use whichever symbol fits best into the graphical context each time that operation is needed. The only restriction on symbols is that they have at least as many input terminals as the primitive operation expects. Extra input terminals are merely ignored.

The primitive operations are grouped into classes. The listing of operations provides the class, name, inputs, outputs, and a short description for the operation of each primitive. Each operation has optional flow input and output terminals. These optional terminals are not included in the listing descriptions. All the primitive operation written names start with "PR" to help avoid confusion with other names one might have present in the system. In describing the inputs and outputs of the operators, a number and a letter are used. The number indicates how many terminals of a particular kind each operator has, and the letter indicates the type of variable expected. A "?" in place of the number means that any number of terminals of that kind are

allowable. The letter designations of data types are as follows: A = any, B = boolean, C = cable, F = fraction, I = integer, N = numerical, and FL = flow.

LIST OF PRIMITIVE OPERATIONS

Class and Name	Inputs	Outputs	Description of Operation
Arithmetic			
PRADD PRSUB PRMUL PRDIV	2N	1N	Self explanatory. The operations check for data type (integer, floating, etc.) and act accordingly.
Boolean			
PRITA PRUNA PRDSA	2B	1B	Intersect, unite, and distinguish (exclusive or) operations.
Test			
PREQ PRUNEQ	2A	1B	Tests two variables of any kind. Output is TRUE if condition is met.
PRGRTR PRLESS	2N	1B	Tests numerical variables as above.
Control			
PRPASS PRSTOP	1A 1B	1A	Any variable is either passed through the operator or blocked. For PASS, TRUE control input means pass, FALSE means block. STOP is reversed from this.
PRFLSW (flow switch)	1I or B	?FL	All flow outputs are undefined except the n'th. n = integer input or 0 or 1 if input boolean.

Class and Name	Inputs	Outputs	Description of Operation
Data Source			
PRTYPIN	None	1A	Makes variable from typed input. Keyboard is used at run time.
PRCONT (contents)	1I 1A	1A	The contents of the memory location specified by the integer input are made into a variable of the same kind as the any-input. The actual value of the any-input is not used.
PRTIME	None	1I	Makes current system internal time into an integer output. Internal time is used to simulate parallelism.
PRUSAD (use address)	None	1I	Graphic primitive. Provides internal reference integer address of the symbol which caused activation of the picture containing a use of this primitive.
Data Sink			
PRTYPE	1A	None	Types out variable at run-time on printer.
PRSEE	1A	None	Shows variable on the screen as text in the operator symbol.
PRCINS (core insert)	1A 1I	None	Stores the variable bit pattern in the core address specified by the integer input.
Miscellaneous			
PRSUBR	3I	2I 1B	General-purpose subroutine call. One integer input is the absolute address of the routine to be called. Two integer inputs and outputs are the two halves of the accumulator. The boolean output indicates whether a return to the calling address plus one or plus two was made.

Class and Name	Inputs	Outputs	Description of Operation
Miscellaneous (Continued)			
PRMODU	1I 4F	None	Graphic primitive. Integer is address (perhaps from PRUSAD) of a symbol to be displayed. The fraction inputs are a Δx, Δy, $\Delta \Theta$, and size change to be applied.
PRTRAP	1A	None	When activated, this operator stops interpreter action so that the graphical program may be modified.
PRGPACT (General purpose action)	1I ?A	?A	Every other primitive has a number. The integer input controls which primitive this operation really is. The integer input is then removed and the operation done to the other inputs present.
Cable Class			
PRCABL	?A	1C	Gathers up any number of any kind of input into a cable bundle. This cable may then be treated as a single variable; i.e. a vector.
PRUNCABL	1C	?A	Unties the cable and fans out the individual outputs.
Associative Class			
PRFIND PRSET PRERASE PRTYPEOUT			The whole operation of this class is too complex for concise description.

APPENDIX C

THE CORAL LANGUAGE AND DATA STRUCTURE

The term "CORAL" stands for Class Oriented Ring Associative Language. The acronym is inexact as CORAL is really a service system consisting of a basic data structure form, a number of action subroutines, and a macro language for programming actions upon the data structure. CORAL was developed at the M. I. T. Lincoln Laboratory for the TX-2 Computer during 1964. The basic ideas of the data structure and language are not new and are applicable to any computer [1, 26]. However, the particular details of the structure and language are intimately related to the peculiarities of TX-2, its macro assembler "Mark IV", and the character set of the Lincoln Writer keyboards.

CORAL DATA STRUCTURE

The CORAL data structure consists of table-like blocks of list elements. CORAL list ties are always formed into rings. Each element in a ring requires one 36-bit word and contains a 17-bit pointer to the next element. One element of the ring is the start or head and all the other elements are subordinate to it. The ring start element contains, besides its forward pointer, 9 bits of data and a 9-bit identification code (-0) which marks it as a start point. Every other element of a ring has a second 17-bit pointer which either points to the ring start element or is used as a backward pointer. One bit marks which type of pointer is being used and rings are built with back pointers and

start pointers alternating. Back pointers point to the closest previous element with a back pointer so that they form a complete ring. Alternation of the less useful pointer types retains the advantages of both pointers in half the space (one word per element) and with only a small loss of time. (Figure 1 illustrates the basic ring structure.)

BLOCKS

A block of elements collects many ties together and thus allows the multi-dimensional associations required for graphical data structures. Although it is sufficient to use at most two ring elements per block, more are often used for increased efficiency. A block is formed from a sequence of registers of any length and contains a blockhead identifier at the top, a group of ring elements and any number of data registers. (Figure 2 illustrates an example of a block.) Blocks are used to represent items or entities and the rings form associations between blocks. Thus, an element may be reached by moving up or down in a block rather than going around its ring. To find out what group the element belongs to it is necessary to find the head element of its ring and this is made efficient through the use of the start ties. If it is necessary to delete an element or insert a new element before it, the back ties are used to find the previous element. Thus, the full set of ties are necessary in the list elements if deletion, insertion, and identifying the elements' group are to be accomplished efficiently.

CORAL PROGRAMMING LANGUAGE

The CORAL language provides a means of programming actions which will operate on a CORAL data structure. A CORAL program consists of a combination of operators and named variables of type "POINTER". A pointer has as its

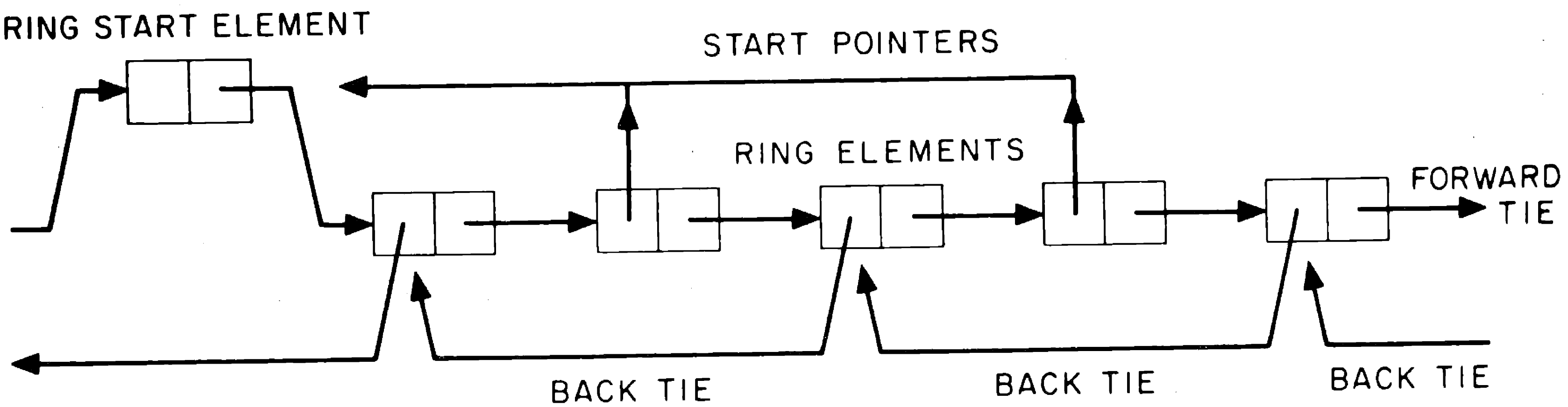


FIG 1 BASIC RING ELEMENT STRUCTURE

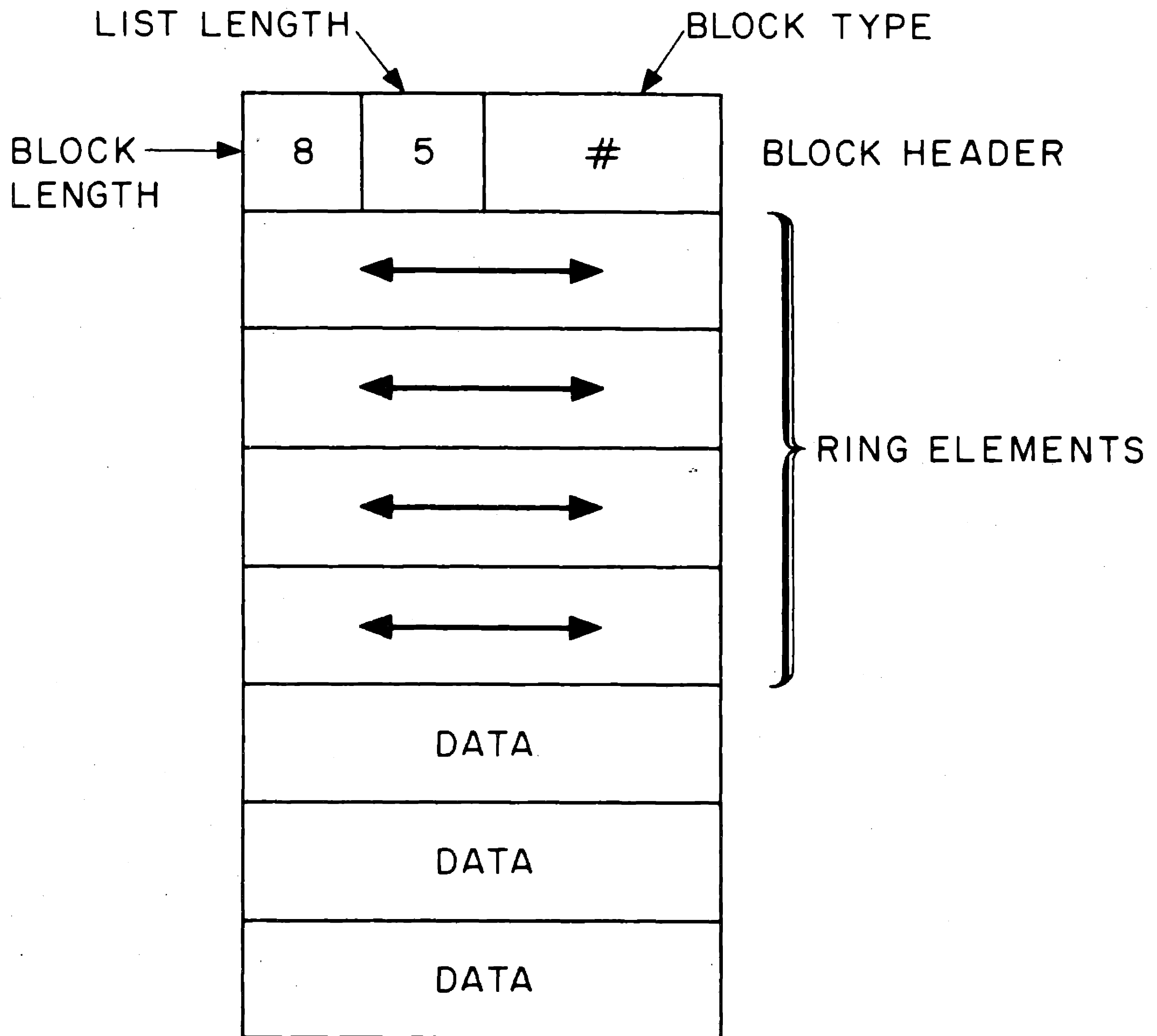


FIG 2 EXAMPLE OF BLOCK FORM

C23-357

value an integer address of a data structure register. Some operators will change the value of the pointer variables. For example, an operation for moving down two registers in a block would simply add two to the value of a pointer, thus making it point to a new location in the block.

Other operations will not change the value of the pointers, but will change the data structure pieces which are pointed to. Consider the operation of inserting a new tie element into a ring. This operation will have two arguments; the pointer to the element to be inserted, and a pointer to a reference element in an already existing ring. At the completion of the operation the pointers will be unchanged but the data structure will be changed since the two elements are now ring neighbors. It is useful to think of the pointers as named fingers keeping track of elements in the data structure. CORAL contains operators which move the fingers to new places in the structure and also operators which make the fingers manipulate whatever part of the structure they are grasping.

The various CORAL operators are used with pointer parameters and also with numerical parameters. The operator symbols are compound characters made from Lincoln Writer symbols. The list at the end of this appendix contains the principal operators and a description of how they work. The computer accumulator is used as an inter-operator communication register. Just as the result of an X + Y operation is normally available in the accumulator, so the result of a pointer moving operation is also available. CORAL operators may thus be nested in an almost arithmetic-like fashion. An operator parameter may be a named pointer variable, or it may be another operator expression. In this latter case the value resulting from the first operation on a pointer

is left in the accumulator and used directly as the input value for the second operation. Nesting to any depth may be accomplished.

CORAL OPERATORS

The list describing the principal CORAL operators uses a number of standard parameter symbols as follows:

Parameter Symbol	Meaning
A	Computer Accumulator
P	Pointer
Q	Reference Pointer
N	Numerical Value
S	Pointer
R	Control Transfer Label
TB	Numerical Value of Block Type

In a CORAL program actual named pointers or expressions will appear in place of the parameter symbols above. The appearance of "S" in macros containing "→ S" will cause A to be stored in pointer S at the end of the macro. "☛ R" at the end of a macro will cause a transfer of control to statement R. Omission of these parameters causes their actions to be omitted.

CORAL OPERATOR LISTING

MOVEMENT

These macros enable one to move from one part of the list structure to another. P points to a starting place which must be a tie register for left-right movement; a tie register or block head for up-down movement, except where indicated. N indicates how far to move. After any of the macros has been executed the accumulator points to the resulting location in the data structure.

Symbol	Format	Description
↑	P↑N→S☚R	Go up N memory locations from P. P may point to any register of block. If N is negative, goes down.
↓	P↓N→S☚R	Go down N memory locations from P. P may point to any register of block. If N is negative, goes up.
⤒	P⤒N→S☚R	Go to bottom of block P, then go up N-1 memory locations. If N is omitted goes to top. If N is 0, ends up at register below bottom. If N is negative, goes down from bottom to outside of block.
↧	P↧N→S☚R	Go to top of block P, then go down N memory locations. If N is negative, goes up above block, normally undesirable.

Symbol	Format	Description
⊁	P⊁N→S☛R	Go right from P around ring N places. If N is negative, does nothing.
⊀	P⊀N→S☛R	Go left from P around ring N places. If N is negative, does nothing.
⊤⊁	P⊤⊁N→S☛R	Go to ring start of P, then right N places around ring. If N is negative, stays at ring start.
⊤⊀	P⊤⊀N→S☛R	Go to ring start of P, then left N places around ring. If N is negative, stays at ring start.
[D]	P[D]N→S☛R	Go down to N+1th data register of block P.

STRUCTURE CHANGING

These macros enable one to build, delete, and arrange list structure elements. After any of the macros has been executed, the accumulator points to some part of the list structure just referenced.

Symbol	Format	Description
⦶	X⦶L→S☚R	*Make master master block* at location determined by X for list structure L long. A points to header of block.
⊜	TP⊜L×LL→S☚R	*Make master block* for block-type TP with length L and list length (non-data length) LL. A points to master block.
Ⓑ	ⒷTP∧L→S☚R	*Make block* of type TP and length L. If L is omitted, the length specified in the master block for type TP is used. A points to first tie register of block.
⧁	P⧁Q→S☚R	*Put* P *right* of Q. If P points to a ring start, error message results. For this and the operators listed below A has pointer P.
⧀	P⧀Q→S☚R	*Put* P *left* of Q. If P points to a ring start, error message results.
⧁̄	P⧁̄Q→S☚R	*Move ring* P to *right* of Q. P must point to a ring start which thereby becomes empty after its ring elements are moved.
⧀̄	P⧀̄Q→S☚R	*Move ring* P to *left* of Q. P must point to a ring start which thereby becomes empty after its ring elements are moved.
ⓣ	Pⓣ→S☚R	*Take* P out of ring it is in, making it an empty tie register (pointing to itself). If P points to a ring start, does nothing.

Symbol	Format	Description
⊗	P⊗→S☛R	Delete ring P. If P points to empty tie register, does nothing. If P points to a ring element, takes it. Then checks to determine if the block containing the ring start (of this ring element) thereby has all of its tie registers empty; if so, return block to free storage, otherwise, do nothing more. If P points to a ring start, does ⊗ to each block containing a ring element of the ring start. A has pointer P.
$\overline{\otimes}$	P$\overline{\otimes}$→S☛R	Delete Block P. This macro does ⊗ to each tie register of block P and then returns block P to free storage. A has pointer P.

TESTS

These macros are used for branching by "asking a question". If "answer" is yes, go to JYES; if "answer" is no, go to JNO. If JYES and JNO are expressions, do them as appropriate.

Symbol	Format	Description
[H]	P[H]⊃JYES | JNO☛R	P pointing to ring start ? NOTE: An empty tie register is considered a ring start. A has pointer P.
[■]	P[■]TP⊃JYES | JNO☛R	P pointing to block of type TP ? A has pointer P.
[E]	P[E]⊃JYES | JNO☛R	P pointing to empty tie register ? (i.e., tie register pointing to itself) A has pointer P.

DATA

These macros retrieve data into the accumulator.

Symbol	Format	Description
[K]	P[K]N→S☛R	Load the contents of the register whose address is N plus the pointer P into A.
[I]	P[I]→S☛R	Get length of block containing P. A has numerical answer.
[J]	P[J]→S☛R	Get list length (non-data length) of block containing P. A has numerical answer.
[T]	P[T]→S☛R	Get type number of block containing P. A has numerical answer.

GENERATORS

These macros cause an action to be performed for each ring element in a ring.

Symbol	Format	Description
[>]	P[>]ACTION^PDL→S☛R	<u>Go right around ring</u> P doing ACTION for each element except element P. Save pointer to each element at S before doing ACTION. Usually P points to ring start. "Current element" may be deleted by ACTION without hurting go-around. If an element is put right of current element by ACTION, it is not visited next. For recursive go-around include parameter PDL which specifies a push-list. Each time ACTION is called, A points to current element. At the end, A has the original pointer P.
	P[>]|ACTION^PDL→S☛R	<u>Go right insertable around ring P</u>. Same as above, except that "current element" must not be deleted by ACTION and, if an element is put right of current element by ACTION, it is visited next.
[<]	P[<]ACTION^PDL→S☛R	<u>Go left around ring P</u>. Same as [>], except goes left around ring.
	P[<]|ACTION^PDL→S☛R	<u>Go left insertable</u> around ring P. Same as [>], except goes left around ring.

CORAL PROGRAM EXAMPLE

The following two statement lines are typical of CORAL programs.

```
(ⒷBLK1→NEWBLK)⧁((OLDBLK↓3)↚(OLDBLK[K]5))

((⍖)⍖NAMRING)[E]⊃((OLDBLK↓1)[>]SUBR1◤LABEL1)
```

The main operator in the first line is ⧁ (put right). The element to be inserted is the first tie register of a newly created block of type BLK1. BLK1 is a symbolic name for some integer type number. The pointer named NEWBLK is assigned the address of the inserted ring element. The new tie element will be inserted next to an element determined by what follows the ⧁ operator. To find the reference element, the program will move the pointer OLDBLK down three registers and then left (backwards) around that ring. The number of steps moved is determined by the contents of the data structure register found by moving the pointer OLDBLK down five. The accumulator's contents are identical with those of NEWBLK after the statement has been executed.

The second line takes the accumulator pointer to the newly inserted tie element, moves to its ring start, then to the top of the block containing the ring start, and down a distance equal to NAMRING. If the resulting tie register is not empty, control goes on to the next line. If the register is empty then the subroutine SUBR1 is performed for each member of the ring found by moving the pointer OLDBLK down one. After all the members of the ring have been treated, control transfers to LABEL1.

BIBLIOGRAPHY

1. Comfort, W. T. "Multiword List Items," Comm. of the ACM, Vol. 7, No. 6, pp. 357-362, June 1964.

2. Conway, M. "A Multiprocessor System Design," AFIPS FJCC Conference Proceedings, Vol. 24, pp. 139-146, Spartan Books, Baltimore, 1963.

3. Davis, M. R., and T. O. Ellis. "The RAND Tablet: A Man-Machine Communication Device," AFIPS FJCC Conference Proceedings, Vol. 26, pp. 325-331, Spartan Books, Baltimore, 1964.

4. Dennis, J. B. "Segmentation and the Design of Multiprogrammed Computer Systems," Journal of the ACM, Vol. 12, No. 4, pp. 589-602, October 1965.

5. Feldman, J. A. "A Formal Semantics for Computer Oriented Languages," PhD Thesis, Carnegie Institute of Technology, June 1964.

6. Feldman, J. A. "Aspects of Associative Processing," M.I.T., Lincoln Laboratory, Technical Note 1965-13, April 1965.

7. Fitzwater, D. R., and E. J. Schweppe. "Consequent Procedures in Conventional Computers," AFIPS FJCC Conference Proceedings, Vol. 26, pp. 465-476, Spartan Books, Baltimore, 1964.

8. Haibt, L. A. "A Program to Draw Multi-Level Flow Charts," IBM Research Report RC-89, April 1, 1959.

9. Heistand, R. E. "An Executive System Implemented as a Finite State Automaton," Communications of the ACM, Vol. 7, pp. 669-677, November 1964.

10. Heller, J. "Sequencing Aspects of Multiprogramming," Journal of the ACM, Vol. 8, pp. 426-439, July 1961.

11. Johnson, T. "Sketchpad III, 3-D Graphical Communication with a Digital Computer," MS Thesis, Dept. of Mechanical Eng., Massachusetts Institute of Technology, June 1963.

12. Kelley, J. L., C. Lochbaum, and V. A. Vyssotsky. "A Block Diagram Compiler," Bell Systems Technical Journal, Vol. 40, pp. 669-676, May 1961.

13. Kirsch, R. A. "Computer Interpretation of English Text and Picture Patterns," IEEE Transactions on Electronic Computers, Vol. EC-13, pp. 363-376, August 1964.

14. Knowlton, K. C. "A Computer Technique for Producing Animated Movies," AFIPS SJCC Conference Proceedings, Vol. 25, pp. 67-87, Spartan Books, Baltimore, 1964.

15. Krakauer, L. J. "Syntax and Display of Printed Format Mathematical Formulas," MS Thesis, Massachusetts Institute of Technology, 1964.

16. Krider, Lee. "A Flow Analysis Algorithm," Journal of the ACM, Vol. 11, No. 4, pp. 429-436, October 1964.

17. Lass, S. E. "PERT Time Calculations Without Topological Ordering," Communications of the ACM, Vol. 8, No. 3, March 1965.

18. Lang, C. A., R. B. Polansky, and D. T. Ross. "Some Experiments With an Algorithmic Graphical Language," Massachusetts Institute of Technology ESL-TM-220, August 1965.

19. Ling, Tse-Sheng (Marvin). "The Logical and Analytical Structure of the Computer-Aided Design Process as Applied to a Class of Mechanical Design Problems," PhD Thesis, University of Michigan, October 1962.

20. Martin, William A. "Syntax and Display of Mathematical Expressions," Massachusetts Institute of Technology, Project MAC Memo MAC-M-257, July 29, 1965.

21. Narasimhan, R. "Labeling Schemata and Syntatic Descriptions of Pictures," Information and Control, Vol. 7, pp. 151-179, 1964.

22. Narasimhan, R. "Syntax Directed Interpretation of Classes of Pictures," ACM Workshop on Programming Languages, San Dimos, Calif., August 1965.

23. Roberts, L. G. "Graphical Communication and Control Languages," Proceedings of the Second Congress on Information System Sciences, pp. 211-217, Spartan Books, Baltimore, 1964.

24. Roberts, L. G. "Machine Perception of Three-Dimensional Solids," PhD Thesis, Massachusetts Institute of Technology, May 1963.

25. Roberts, L. G. "A Graphical Service System With Variable Syntax," ACM Programming Languages Conference, San Demos, California, August 1965.

26. Ross, D. T., and C. G. Feldman. "Verbal and Graphical Language for the AED System; A Progress Report," Massachusetts Institute of Technology, Project MAC Technical Report No. 4, May 1964.

27. Stockham, T. G. Jr. "Some Methods of Graphical Debugging," IBM Symposium on Man-Machine Communication, Yorktown Heights, New York, May 1965.

28. Sutherland, I. E. "Sketchpad: A Man-Machine Graphical Communication System," M. I. T., Lincoln Laboratory Technical Report No. 296, January 1963.

29. Teager, W. M. "Real Time Man-Machine Communication With Graphical Languages," First Congress on the Information System Sciences, ESD-TDR 63-474-5, January 1964.

30. Ulrich, E. G. "Time-Sequenced Logical Simulation Based on Circuit Delays and Selective Tracing of Active Network Paths," Proceedings of 20th National Conference of the ACM, Cleveland, Ohio, p. 437, August 1965.

31. Westerfeld, E. C. "Flowchart Compiler Using Teager Board Input," MS Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering, June 1965.

32. ______________. "Research on Advanced Dynamic Attribute Extraction," AF DOC, AFCRL-65-736, Prepared by Adams Associates under contract AF 19(628)-453.

33. Five Papers from the 1964 Fall Joint Computer Conference:

Allen, T. R., and J. C. Foote. "Input/Output Software Capability for a Man-Machine Communication and Image Processing System," AFIPS FJCC Conference Proceedings, Vol. 26, p. 387, Spartan Books, Baltimore, 1964.

Cole, M. P., et al. "Operational Software in a Disc Oriented System," AFIPS FJCC Conference Proceedings, Vol. 26, p. 351, Spartan Books, Baltimore, 1964.

Hargreaves, B., et al. "Image Processing Hardware for a Man-Machine Graphic Communication System," AFIPS FJCC Conference Proceedings, Vol. 26, p. 363, Spartan Books, Baltimore, 1964.

Jacks, E. L. "A Laboratory for the Study of Graphical Man-Machine Communication," AFIPS FJCC Conference Proceedings, Vol. 26, p. 343, Spartan Books, Baltimore, 1964.

Krull, F. N., and J. E. Foote. "A Line Scanning System Controlled from an On-Line Console," AFIPS FJCC Conference Proceedings, Vol. 26, p. 397, Spartan Books, Baltimore, 1964.

BIOGRAPHICAL NOTE

William Robert Sutherland was born on May 10, 1936 in Hastings, Nebraska. After an early childhood near Chicago, he moved to Scarsdale, New York where he graduated from the Scarsdale High School. Mr. Sutherland attended Rensselaer Polytechnic Institute, Troy, New York under the NROTC Holloway plan and received his Bachelor of Electrical Engineering degree in June 1957. While at Rensselaer he was a joint winner of the Ricketts prize in 1957. Mr. Sutherland entered the Navy in June 1957 and completed flight training as a Naval Aviator in January 1959. While on duty with a carrier anti-submarine aircraft squadron in Norfolk, Virginia, he was awarded the Legion of Merit. Upon release from the Navy in July of 1962, he entered M. I. T. as a full time graduate student under a National Science Foundation fellowship. He received the Master of Science Degree in Electrical Engineering in June 1963. During his two years as a NSF fellow Mr. Sutherland was associated with the Research Laboratory of Electronics at M. I. T. In June 1964 he became associated with the M. I. T. Lincoln Laboratory as a Staff Associate. He is a member of Tau Beta Pi, Eta Kappa Nu, and Sigma Xi.