The Reactive Engine

A. C. Kay

1969

I wish to God these calculations were executed by steam... C. Babbage, The Analytical Engine

Many of the diagrams in the thesis were hand drawn. They have been redrawn hopefully without adding errors.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

This work could not have prospered without the challenging and pertinent criticisms of many people and the catalytic nature of the environment in the Computer Science Department of the University of Utah.

I would particularly like to thank Professors David C. Evans and Robert S. Barton for supplying the motivation, Patricia M. Madsen for providing the inspiration, and Edward C. Cheadle for making it all possible.

I also wish to thank C. Stephen Carr for many hours of fascinating discussion that certainly have had a lot to do with the present form of this work.

ABSTRACT

The design of machines which can participate in an interactive dialogue is the main topic of this thesis. To do this successfully, a language processor and a more general memory system will be incorporated as the lowest level of the hardware.

A higher level kernel language, FLEX, which is comprehensive enough to provide the basic dialogue, extend itself into more desirable forms, and describe other languages, yet is concise enough to be implemented directly as a hardware processor, is developed.

Several models of abstract interpreters for this reactive system, starting with a very simple textual model, are then discussed, followed by hardware representations of these schemes.

These considerations lead to the design and implementation of the *FLEX Machine*, a personal, reactive, minicomputer which

communicates in text and pictures by means of keyboard, line-drawing CRT, and tablet.

I. Preface and Reading Grade

I.a. Preface

Maybe Computer Science should be in the College of Theology! ...R. S. Barton, 1967

A rationale for writing a dissertation (besides the obvious one) should have something to do with the desire to shorten someone else's path to a given point so that he may press on to more interesting horizons. For that reason, an attempt has been made to write a somewhat verbose, but rather complete, account of the past two year's work. Since this paper involves the entire design of a computer system: *hard-*, *firm-*, and *software*, the considerations leading to some of the decisions and compromises must, of necessity, be somewhat involved if one is to *tell all*. Also, it is strongly felt that the crux points of decision making add more to the philosophical nature of a work than the rather concise engineering presentation to be found in Appendix A.

Granted, it is nice that a complete design, because of its inherent recursivity, may be boiled down to a few pages of symbols - for that is the nature and purpose of abstraction. Yet also, whenever a work such as this has been completed, it should be relegated immediately to the novice information scientist so that it need never again seem erudite. It is to this reader that *The Reactive Engine* is dedicated, not to the so-called Computer Science professional.

Thus, a low level (but accelerated) approach has been taken. It is impossible to present all of the introductory material that is necessary for the understanding of this thesis in one short chapter. Nevertheless, that chapter is most important to the novice reader, since it seeks to provide an intuitive feel for the environmental concepts and (more to the point) tries to instill motivation for consulting the basic references in the field.

Because there is a very real difference between the form and effects experienced by an interactive user of the FLEX environment and the quibblings and rationalizations about why things turned out as they did. This paper has been divided into a number of documents.

This portion contains the considerations and philosophy leading to the gross design of an interactive mechanism. An informal description of the external aspects, followed by a formal definition of the entire machine, is found in *The FLEX Handbook*, which can also serve as a user's manual. The other section consists of large charts which it is convenient to spread out separately.

There seems to be an interesting sociological connection between the rise and acceptance of the new dogma, as opposed to the old. The Pythoness at Delphi gagging on the fumes of insight has nothing on her modern counterparts, who are overwhelmed by a suffocating flow of bits. The quotations at the chapter heads, gathered from conversations (and certain sacred and profane documents), reflect the basic duality of Computer Science (and human nature).

Alan C. Kay

Salt Lake City

August, 1969

I.b. A Reader's Guide to Non-Periodic Literature

This work could be subtitled: *A Designer's View*. As such, its extreme length is somewhat intimidating to the casual reader, though it may be welcomed by the more serious student. This sub-preface will discuss a number of possible paths through the document.

Although each section builds on the preceding sections, the paper need not be read in a serial order. Sections II.1 and II.2 are included primarily as a philosophic base to the rest of the book and may be skimmed or skipped on first reading by the pragmatic reader. II.3 provides a reasonable starting place for the *Cooks Tour*. II.3h on the numerically-controlled tool system is a rather complete, somewhat independent paper which may be omitted on first reading.

Those readers familiar with mapping schemes and storage allocation problems may omit reading most of III.6. They may find III.6bii and III.6i to be somewhat novel.

III.6i has been assumed to mean III.6ai

Sections III.9 and III.10 are not central to the issues of this work and will not stunt the reader's growth if he passes over them.

Appendix A, *The FLEX Handbook*, may be read independently of the rest of the text. It is in the form of a user's manual and only presents *how's*, not *why's*.

The comprehensive reader will discover that, although all sections relate to the FLEX language and machine, the book itself may be regarded as a set of somewhat independent papers covering many criteria of machine design and, hence, may be approached in that fashion.

II. Part A: The Basic Dialogue

then in certain cases and for irresponsible men it may be that non-existent things can be described more easily and with less responsibility in words than the existent, and therefore the reverse applies for pious and scholarly historians: for nothing destroys description so much as words, and yet there is nothing more necessary than to place before the eyes of men certain things the existence of which is neither provable nor probable, but which, for this very reason, pious and scholarly men treat to a certain extent as existent in order that they may be led a step further towards their being and their becoming. --Joseph Knecht's holograph translation (M.L.)

1. Introduction

Systems programmers are high priests of a low cult. - R. S. Barton 1967

It is not the purpose of this book to determine *why* human beings build machines, nor is it to describe *how*. Yet a certain amount of metaphysics may creep into the discussion from time to time, since any investigation into the design of devices which are actually to be used by people must of necessity try to guess what the class of potential users will *want*. Of course, they do not know themselves if their needs are potentially interesting, and it is useless to poll the remainder (as previous experiments have shown) for they exhibit the well-known, *committee syndrome*: *...the only form of life with ten bellies and no brain*.

A good class project for undergraduates who have not become too tainted with either the commercial or research computing milieu, is to have them design a computer system for a *think tank* such as RAND or the Institute for Advanced Study at Princeton. It is a delightfully nebulous question, since they quickly realize it will be impossible for them to even discover what the majority of the thinkers are doing. Indeed, many of the researchers will not know themselves or be able to articulate that state of mind of *just feeling around*. It is at this point that a wide philosophical division appears in the students. Almost all of them agree that there is really nothing that they can do for the scientists. The more engineering-minded of the students, seeing no hard and fast solution, stop there. The rest, who are somewhat more fanciful in their thoughts, say *...maybe 'nothing' is exactly the right thing to deliver, providing it is served up in the proper package*. They have articulated an important thought. Not being able to solve any one scientist's problems, they nevertheless feel that they can provide tools in which the *thinker* can describe his own solutions and that these tools need not treat specifically any given area of discourse.

The latter group of students has come to look at a computing engine not as a device to solve differential equations, nor to process data in any given way, but rather as an abstraction of a well-defined universe which may resemble other well-known universes to any necessary degree. When viewed from this vantage point, it is seen that some models may be less interesting than the basic machine (payroll programs). Others may be more interesting (simulation of new designs, etc.). Finally, when they notice that the power of modeling can extend to simulate a communications network, an entirely new direction for providing a system is suggested.

While they may not know the jargon and models of an abstruse field, yet possibly enough in general of human communications is known for a *meta*-system to he created in which the specialist himself may describe the symbol system necessary to his work. In order for a tool such as this to be useful, several constraints must be true.

1. The communications device must be as available (in every way) as a slide rule.
2. The service must not be esoteric to use. (It must be learnable in private.)
3. The transactions must inspire confidence. (*Kindness* should be an integral part.)

The systems described in this book do not claim to completely satisfy these conditions, but they have definitely used these considerations as a global basis for the entire project.

First Principles

Machines which do one thing only are boring, yet exert a terrible fascination. The black box, whose only action when switched on is to exude a hand which turns the switch to *off*, is a popular symbol of the *futility of it all*. It was the creation of machines with exactly the opposite intent which got the whole business started.

In 1801, J. M. Jacquard decided that looms which could only weave one pattern were unbearably dull and devised a *parametric* drawloom which was *driven* by punched cards. Babbage owned what was probably the first instance of *computer graphics*, a woven portrait of Jacquard in silk done on the parametric loom whose resolution (1000 threads to the linear inch) approached that of the very best devices today. It was this notion of an *interpreter* basing its decisions on stored *parameters* that so fired Babbage's imagination. Unfortunately, neither the technology nor the mentality of his century were fertile ground for his ideas.

Of course, modeling systems have existed for as long as the human species. In fact, some anthropologists base their partition of primates into *sapient* and *non-sapient* beings solely on the existence of a well-developed language. It is well known that humans need to build a mental model of their environment before they can apprehend it in any way. Probably the two greatest discoveries to be made were the importance of position in a series of grunts, and that one grunt could abbreviate a series of grunts. Those two principles, called syntax (or form) and abstraction, are the essence of this work.

It is fascinating to note the relationship between the form of languages and the way a culture looks at the world.

The basic framework of the Indo-European languages is that of the verb inflected for tense, etc., which closely determines the way abstractions are partitioned into sounds. To this speaker, the world has objects which endure over time. Verbs connote actions and processes. Adjectives indicate permanent qualities, while transitory qualities (and qualities of qualities) are handled by several types of adverbs. Causality is tied to the ordering imposed by tense: producing *effect*. Things are discrete, etc.

This is a very digital world and much of it has counterparts in the basic architecture of present-day computers. However, much of mathematics describes another, more continuous world. *Cause* and *effect* are functional in nature, and time is viewed from a God-like vantage point. In this framework, there are relations between notions rather than discrete actions. Interestingly enough, there are human languages which reflect this very different way of apprehending the world. It was the discovery of Hopi that led Whorf (1) to the conclusion that the way a group of people see, hear and think about things is constrained by their language, since that is actually the medium through which they build and manipulate their internal models of the world.

Noting these thoughts leads one to a sobering situation when contemplating the design of a system to represent the form (syntax) and the modeling (semantics) that is necessary to represent a universe (albeit a small one). These weighty considerations will be treated in a very *engineering* and pragmatic way: they will be always remembered, adhered to when possible, and ignored when necessary!

Concepts and Vocabulary

In keeping with the principle of abstraction, it will be much simpler to define some terms which will be used frequently rather than to continually refer to these notions by circumlocutions. An intuitive (rather than axiomatic) approach is taken, with the belief that it is possible to be precise without being stuffy. Necessary formalities are to be found in the appendices and the references; they should not be sought here.

A few simple concepts will suffice to open up this investigation.

Sets

A set is thought to be a collection of *things* and may be represented by *{*, *}* surrounding the elements that are considered to be members.

Ordered Pairs

This fundamental notion will be symbolized by angle brackets with a dot separating the pair. <x • y> is an ordered pair where x and y may themselves be ordered pairs, etc.

The dot appears large, as shown here, in parts of the thesis but not always.

Strings

For now, a string is a set whose members are ordered. For example, the word, *string*, consists of the unordered set {"t", "i", "n", "s", "g", "r"} of letters and an ordering giving <"s", "t", "r", "i", "n", "g"> . An alternative definition defines *&* as the *concatenation* operator, such that *"a" & "b"* is the string *"ab"*. So *string* may be formed by *"s" & "t" & "r" & "i" & "n" & "g"* and will be assumed to be equivalent to the angle bracket notation until further elaboration.

Names

A name is a string starting with a letter and followed by an arbitrary sequence of letters or digits, or a string starting with a digit, followed by an arbitrary sequence of digits.

Free and Bound Names

A name as defined above consists solely of a sequence of characters. To paraphrase Humpty Dumpty: *It means just what you want it to mean, nothing more nor less*. As such, it is a very unsatisfactory creature and is roughly equivalent in semantic content to an inelegant simian belch. Consider now a sequence of digits: 1245. Ink marks on a page! It has the same vacuous extent as the name above. But now you say, *I know those marks mean a set with one thousand twelve hundred and forty-five members* Not so, for computer scientists are not to be trusted. The author may exert his *Dumptyrian* prerogatives to assert that those marks actually represent something in base_8; i.e., the set that has 677 (base_10) members. Sequences of characters about which only their textual representation is known are considered to be *free* (and slippery) names.

Names may be *bound* down by specifying some:

Property Names and Referents

The representation: base_8 • 1245 ≡ {1, 1, 1 ..., 1} relates two names and a set using two other marks for clarification. • may be read *of*. ≡ may be read *is defined to be*. However, these English *equivalents* do not clear the air. What truly matters is that associated with the name *1245* there is an ordered pair: <base_8 • {1, 1, 1 .... , 1}>

The first member may be called the *property name* or *attribute*, the second, the *referent* or the *value*. This pair represents a *binding* of the name 1245. Another binding would be:

<base_10 • {1, 1, 1 .... , 1}>

where the number of members of the referent is presumably different. Now it can be stated (with tongue in cheek) that the things that are *known* about 1245 can be expressed as the set of *ordered* pairs:

1245: { <"base_8" • {1, 1, ..., 1}> ,
<"base_10" • {1, 1, ..., 1}> }

In algebra, when it is wished for x to *stand for* the value 5, expressed as x←5, it is

value • x ≡ 5

that is meant.

This may seem like quibbling, but it is very convenient to be able to nail down the bindings of *x* when challenged. There may be other bindings which are apparent to the reader, such as

"style" • x ≡ © IBM Selectric: "Delegate"

Circumlocutions

The person who, through consummate artistry and careful application, seasons and arranges the products of those who toil on the soil and in the plains is a circumlocution for *chef*. Languages which do not have a ready mechanism for replacing the longer form with the short have grave difficulty in transmitting complicated constructions. A number of American Indian languages have this problem and, while it is quite possible to express any concept by means of a circumlocution, the system is not well suited for human communication because of the short attention span of the listener. The set with *677 (base_10)* numbers is a rather uncouth circumlocution for *base_8 • 1245*. Indeed, the author cheated by sneaking in an elipsis of three dots (...) to signify *lots* because he didn't feel that it was necessary to exhibit the entire set. And that, indeed, is the crux of this matter. It truly is rarely necessary to exhibit *all* of more complicated constructs to deal with them successfully. The Trobriand Islander gets along well with:

base_10 • 1 ≡ {1}
base_10 • 2 ≡ {1, 1}
base_10 • 3 ≡ {1, 1, 1}
base_10 • 4 ≡ "many"
.
.
.
base_10 • 1245 ≡ "many"

Other Ordered Pairs

So far the bindings of names has been expressed as sets of ordered pairs establishing relations. It should be noted that other pairs may be formed which can express equivalent *meanings*. The intension of the *attribute* may also be expressed as a set of ordered pairs. The following two representations may be considered to be equivalent:

≤ • 5 ≡ 4 gives 4 : < ≤ • 3 >
≤ • 4 ≡ 3 and 5 : < ≤ • {4, 3} >

≤ • 5 ≡ 3 or ≤ : { < 5 • 4 > , < 4 • 3 < }

Note: ≤ nearest symbol that is closest to the one actually used

The latter representation is the way relations are defined in mathematical logic: as the set of ordered pairs exhibiting all the possibilities.

The word *function* can be taken to mean a relation defined as a set of ordered pairs. If ≤ is thought to be a function and defined for 5: ≤ • 5, one would hope that the set {3, 4} would be exhibited as the possible right-hand members of <5 • 3> and <5 • 4> . Note that <5 • 3> and <5 • 4> are equivalent to <5 • {3, 4} > in this formulation.

Higher Order Relations

Given an ordered pair, an ordered triple may easily be defined as:

<1, 1, 1> ≡ <1 • <1 • 1> >

That is, the ordered pair consisting of an ordered pair and a member. Third order relations such as:

son: <Bob, John, Mary>

where John and Mary are father and mother, etc. There are a number of fourth order relations in human languages: one of them is *sell*, where

sell: < person_1, object, person_2, $_dollars >

Mappings, Functions, Transformations

These terms will be used somewhat interchangeably. A mapping, etc. is just considered to be a set of bindings which may be explicit - i.e., a table of bindings for the function tan:

```
tan: { <  0 ° •  0.00000 >,
     < 10 ° •  0.17633 >,
     < 20 ° •  0.36397 >,
     < 30 ° •  0.57735 >,
        .              .
        .              .
        .              .  }
```

or implicit: a rule for finding the bindings from other mappings:

tan • x = sin • x / cos • x

or, interchangeably

tan(x) = sin(x)/cos(x)

Abstraction and Elaboration

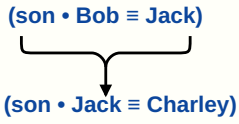
In the previous sections, abstractions which were formed by constructing bindings were put to good use. Lengthy circumlocutions were handily abbreviated and no longer needed to be used directly. Now elaboration, the inverse of abstraction, will be used to retrieve interesting bindings. If

son • Bob ≡ Jack (x)
son • Jack ≡ Charley (y)

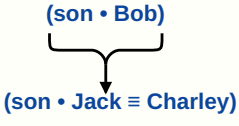
are bindings, then consider

son • (son • Bob ≡ Jack) ≡ Charley (1)
son • (son • Bob) ≡ Charley (2)

(1) uses *(,)* in the usual way to factor out an expression which is of interest (and which needs to be done first). In order for (1) to mean the same thing as the previous two bindings, the *value* (or what is *left behind*) in the *(,)*, must be the name Jack. Then the elaboration of (2) is



In (2) only the left side of the binding is given. This can be taken to mean that all bindings which have *son* as a left part - i.e., all *son • ?* will yield up their right parts. If (x) has been done, then the elaboration of (2) is



Notice that the rules of elaboration have been chosen in a highly arbitrary, yet somewhat intuitive manner.

Suppose that it is wished to abstract the following.

A relationship held in the population Bob, Jack and Charley is *son • Bob*. If it is stated

rel • population = son • Bob

the wrong thing will happen in that *son • Bill* will be elaborated to *Jack* by the previous rule. There is an obvious need for devices to both provoke and prevent the elaboration of bindings.

Primitives

These may be thought of as *axiomatic notions* which must be described in English, yet have the power to generate some interesting system. For example, the following collection of notions are all that is necessary to produce any computable function: (2, 3)

The empty sequence <> also called nil
The ordered pair: <X • Y>
Test for equality to nil: if X = <> then ... else ...

Function definitions by hd <X • Y> ≡ X
stating elaboration rule tl <X • Y> ≡ Y
ε <X, Y> ≡ <X • Y>

Integers are generated in a way very similar to the previous section by identifying <> with 0 and defining a successor function.

Suc(x) ≡ ε (<> • X)

	0	1	2
so	<>	<<> • <>>	<<> • <<> • <>>>
or	nil	<nil • nil>	<nil • <nil • nil>>

Numerals and names may be defined as functions.

0 ≡ nil
1 ≡ suc(0)
.....
4 ≡ suc(3)

Then

X ≡ 4 i.e. X elaborates as 4 which elaborates as <nil • <nil • <nil • <nil • nil>>>>

Addition may be defined as

plus(X,Y) ≡ if X = <> then Y else plus(tl(X), suc(Y))

X is *counted down* to 0 as Y is counted up in this unary number system. Notice that this function is defined as an *induction* on X. This is called a *recursive* definition and is perfectly straightforward. 3 + 4 is elaborated as:

plus(3, 4) is
if 3 = 0 then 4 else plus(2, 5)
plus(2, 5) is
if 2 = 0 then 5 else plus(1, 6)
plus(1, 6) is
if 1 = 0 then 6 else plus(0, 7)
plus(0, 7) is
if 0 = 0 then 7

The *7* becomes the value of each *else*, *on the way back*.

Arbitrary lists are formed by ε.

ε <X, Y> ≡ <X • Y>

The length of a list is

lg(X) ≡ if X = <> then 0 else suc(lg(tl(X)))

Equality of. two numbers is

equal(X, Y) ≡ if X = <> then if Y = <> then <> else <nil • nil>
else equal(tl(X), tl(Y))

X and Y are counted down until only *nils* are left. A *nil* (zero) is passed back for *true*, since that is all that can be tested for in the conditional. <nil•nil>(one) is passed back for false. This function may now be used as

if equal(X,Y) = nil then ... else ...

Equality of two arbitrary lists is

equal(X, Y) ≡ if X = <> then if Y = <> then nil else <nil • nil>
else if equal(hd(X), hd(Y) = <>
then equal(tl(X), tl(Y))
else <nil • nil>

This variation will *walk* an arbitrary list and, of course, will also work for numbers.

Of course, there are other primitive notions which will perform equivalent actions. The important point is that it is possible to start from a very simple collection of abstraction and elaboration rules and build structures of arbitrary power. This surely is one of the criteria for a successful meta-system.

But it alone is not enough, for the user quickly is buried under an avalanche of *( )* and < >. Some way must be found to extend the textual representation, along with the semantic intent.

Parameters

For this example, the elaboration rule has been simple. A function is elaborated when its operands become available. It has no control over the actual disposition of the formal parameters; they are always (and only) ordered pairs or *nil*, since they have been completely expanded before the function is elaborated. This is annoying, since many elaborations are really involved with *names*, rather than their *values*.

Lambda Notation

The preceding example also brings up the issue of function definition and composition. A tacit agreement was made between the author and the reader that the formal parameters (X, Y) in the definition

plus(X,Y) ≡ if X = 0 then Y else plus(tl(X), suc(Y))

and the actual parameters (3, 4) in the *call*

plus(3, 4)

form a correspondence. That is, the *3* replaces *X*, the *4* replaces *Y* everywhere in the definition before elaboration. Another way to look at this situation is to consider two extra bindings to have been made

X ≡ 3
Y ≡ 4

before the *if*. Now the *X* will elaborate to *3* during elaboration, not before.

In either case, the *order* of the formal and actual parameters determines the correspondence to be made. This would be indicated in the lambda notation of Church (4) as

plus ≡ λX, Y • if X = 0 then Y else plus (tl(X),suc(Y))

The list of names following the lambda (λ) explicitly states the order in which the function will expect to receive its parameters. In the above case, it made no difference to the final result whether replacement was done by copying before elaboration or by setting up a binding and allowing elaboration.

It is also obvious that

$f = \lambda x \bullet x^2 + 3$ and
$g = \lambda y \bullet y^2 + 3$

are the same functions. Lambda notation is implicitly used in almost all programming languages for declaring the correspondence between actual parameters. Unfortunately, this may not always be the most useful way to call a function, as will be seen later.

Another question that arises is what to do with F(A+B).

Should the addition be carried out?

There are no hard and fast rules for what the *default* case (the one that is done if nothing is present to prevent it) should be. In Fortran, the function call

F(A + B)

means F(5) if A = 3 and B = 2. In Algol, the very same call means F(A + B) until F can decide when and how A + B should be elaborated. (Clearly, + can also be represented as a set of bindings). On the *F* side of the call, the single parameter can be represented by the name X. F can decide to force elaboration before anything else happens by stating VALUE X. In this case, F(5) is what is meant, just as with Fortran. However, X could alternately be thought of as integer procedure X; A + B; which is elaborated *each* time X is mentioned. The bindings that are formed with X and the actual parameter are distinctly different. In the first case

value • x ≡ A + B which is the same as
value • x ≡ 5

The second case has A + B protected from elaboration by enclosing it in<, >. So

value • x ≡ <A + B>

Another problem arises immediately. What does

value • Z ≡ X

mean? It is ambiguous as it stands. It could mean

(a) value • Z = X (the name x) or
(b) value • Z = <A + B> (the binding of X)
(c) value • Z = A + B

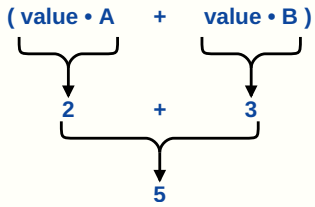
The first issue may be resolved by insisting that

value • Z = value • X be used

for the last two cases. value • Z now just relates the*name* x to the value • z. The last two cases are often resolved in languages by specifying that the elaboration of <, > involves stripping away one set of brackets as the binding is formed. So case (b) can be gotten by

value • x = A + B
value • z = value • x

The elaboration of value • Z will be A + B *unprotected* which will themselves elaborate into the value of the sum. Note that this discussion is cheating somewhat. A + B is actually ambiguous as it stands. If A and B are to mean anything as mathematical *variables*, then they must have bindings to numbers which, after all, are the only entities the + should know anything about. So A + B really means



This is a very unsatisfactory way to write things if the user wishes to evaluate formulas to numbers. On the other hand, if he wants to *manipulate* the *formula*, everything is fine. Clearly, some global way is needed to specify just what is desired. These mechanisms are called:

Coercions

The term first came to light in the ALGOL-68 documents.(5)

Coercive rules of elaboration are formed by keeping all parameters in their most abstract bindings while they are being passed and allowing the procedures (functions, operations, etc.) to decide their elaboration *after* the application of the *call* mapping. Primitive examples of this are seen in the various algorithmic languages.

Suppose in Algol:

```
X := 5          (which means value • x ≡ 5)
Y := 2          (which means value • y ≡ 2)
and x + y       (means (value • x)  + (value • y) is 7)
but x := x + y   (means value • x =  ( (value • x) + (value • y))
means
     X:= 7
not   5 := 7
```

The + wishes elaboration of its operands, while the *x* clearly demands elaboration on the right side, but not on the left. The distinction between a *value* call and a *name call* in ALGOL procedures is another case in point. But all these coercions, while eliminating a great deal of annoying punctuation (including the more grandiose schemes in ALGOL-68), have a great flaw: they are decided by the whims of the (usually long gone) language designers and implementers and, by one of Murphy's laws, are almost guaranteed to be what the user does *not* want.

Since a total lack of knowledge of the user's needs has been a postulate in this paper, any linguistic interpreters which are devised need to allow total control of all binding mechanisms by the user so that both global and local control of coercive processes is possible.

Syntax, Semantics and Pragmatics

It has already been stated that syntax has something to do with the allowable *textual* forms of some *semantic intent*. Although the syntax of the previous example language is quite spare, it could be trimmed even further by replacing the more readable *if X = <> then Y else Z* by a concise functional form *if(X, Y, Z)* where the value of Y will be taken if X elaborates to *nil* (otherwise Z). The semantic intent is the same, in that both instructions elaborate to the same list.

So there exists a mapping between allowable textual forms and the bindings of individual names. This is simply an extension of the binding concept as applied to names and now covers the generation of *intermediate* form by the elaboration of previously made bindings.

There also exists a mapping between semantic elaboration and pragmatic elaboration. This depends greatly on the device that is performing these expansions. As will be seen in succeeding sections of the introduction, some machines will elaborate *3 + 4* in essentially the same way as *plus (3, 4)*. Others will have a *fast adder* tucked away somewhere which can find *7* in a much quicker manner.

Just as there are many ways to syntactically express abstractions, there also exist equivalent *elaborators* for the same notion. This is the *pragmatic environment* for the interpretation. Smooth transformations from one equivalent representation to the next and back again are usually only effected by simultaneously considering all three areas when a language system is designed. Although this has partially been done in the design of the B-5000 (6), the SDS-940 (and one or two others); the FLEX machine represents a more comprehensive attempt than the previous efforts.

Interpretation

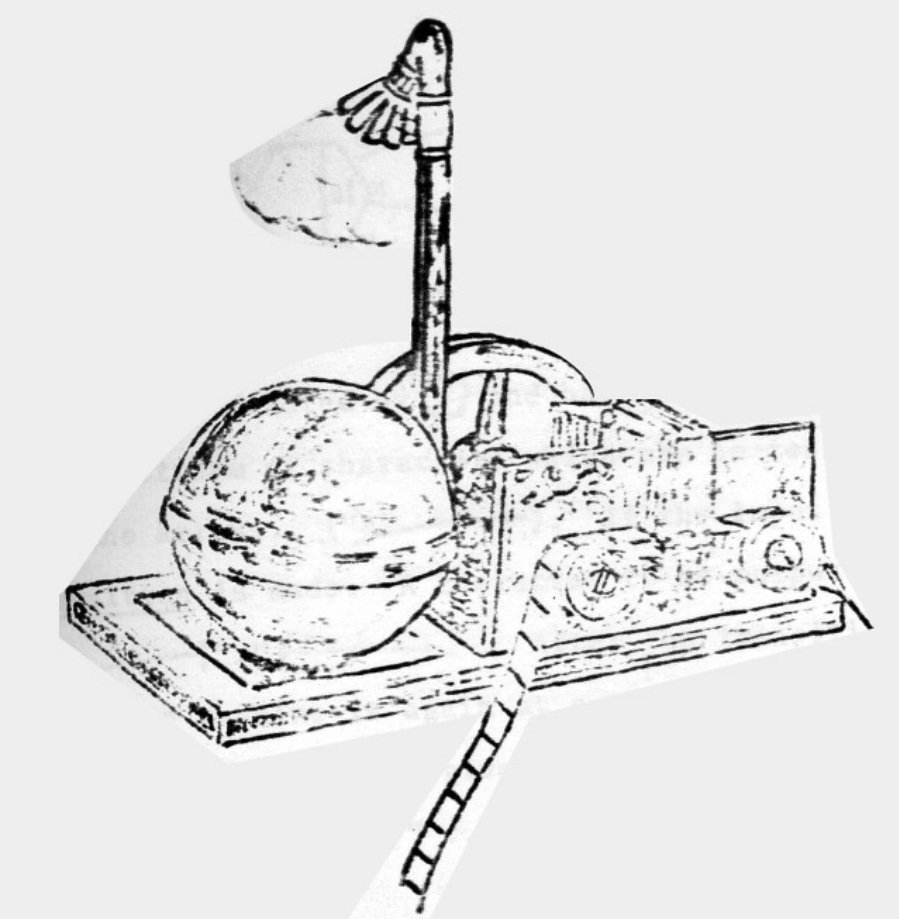
An *interpreter* is a machine that can decide its future actions by examining information held in a store. In a sequential interpreter, the information that is read is presented in the form of a *string*. Although various *abbreviations* may appear in the string (such as jumps, loops, etc.). the presumption is that, for a finite algorithm, there exists a single string that the interpreter may follow.

Turing Machines

How complicated does an interpreter have to be? One of the most important result in the theory of computation was obtained by Alan Turing in 1936 before *any* computing hardware had been built. The device he designed was very simple, yet had the ability to compute anything which any possible mechanical contrivance could compute. In particular, as will be seen, there are *Turing machines* which can simulate any other Turing machines that can possibly be devised.

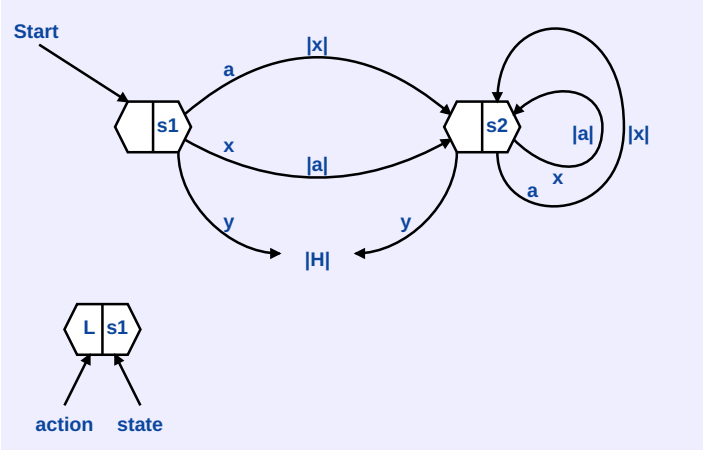
Although many of the results discovered in the branch of mathematics known as Automata Theory are not of interest in this discussion, it will be seen that the roots of sequential translation and production systems both spring from elementary operations on finite strings of characters and, thus, deeply influence the philosophy of machine design. More importantly, the *semantics* of simulation, if anywhere, have their basis in Post-Production Systems and the Universal Turing Machine.

An Intuitive Machine



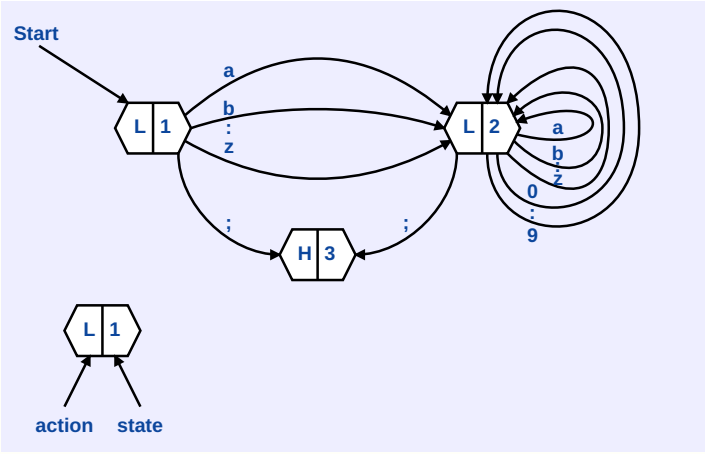
This rather simple-minded device is a morphization of a purely conceptual machine. A tape consisting of an indefinite number of *frames* which may be written and erased is moved right and left by two drives. A sensing device may examine one frame at a time and a pen may overwrite a character from some finite alphabet. The tape may then be positioned right or left, according to the interpretive whim of the logic box. The important question, of course, is the form of the logic box, which is a:

Finite State Machine



The input is a stream of characters which may be tested at the base of the arrows. If the input matches, the transition is made. A character to be output is denoted by |x|. After the transition is effected, the machine is again at a state (which may be the same). The above machine, given *aaaxaxaxxxy*, will transform it into *"xxxaxaxaaay* .. (i.e., the *complement* of the characters).

The input and output streams may be supplied by using the tape and only moving it to the left.

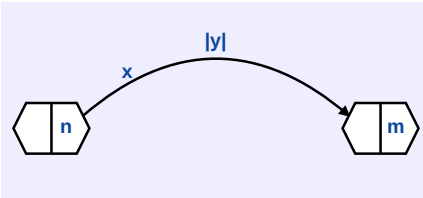


The figure shows a machine which tries to find a name (as defined previously) on the tape. Assume the machine is at *state* 1 when it starts. The quoted letters are tested against the frame of the tape under the TV camera. If a match is made on an alpha character, the machine follows the arrows to state 2, the tape is advanced left (L) one notch, and now is examined for *both* alphas and digits. This process will continue as each successful match returns the machine to state 2 until a *;* is found. This sends the machine to the *halting state* 3.

This machine is called *finite* state because only a finite number of states, members of the tape alphabet, and transitions, may be used. It must move the tape in the same direction each time a transition is made.

Another way of looking at these devices is to notice what information needs to be recorded to completely describe the actions that are taken.

A *transition* for a finite state machine may be described by stating four things.



name of source state	character to be recognised		character to be written	Destination state
n	"x"	||	"y"	m

So the above machine for recognizing names could be described by:

```
1     ";"     ||     ";"     H
1     "a"     ||     "a"     2
1     "b"     ||     "b"     2
.     .             .       .
.     .             .       .
.     .             .       .
1     "z"     ||     "z"     2
2     "a"     ||     "a"     2
.     .             .       .
.     .             .       .
.     .             .       .
2     "z"     ||     "z"     2
2     "1"     ||     "1"     2
.     .             .       .
.     .             .       .
.     .             .       .
2     "9"     ||     "9"     2
2     ";"     ||     ";"     H
```

In some sense, the *state diagram* as completely describes what a *name is* as the more verbose English definition. It is somewhat unwieldy, as it stands, because of the lengthy circumlocution chosen for the alphabet. Suppose an abstraction is formed by creating some worthwhile bindings.

```
member • letter = "a"
member • letter = "b"
  .   .   .     .
  .   .   .     .
  .   .   .     .
member • letter = "z"
```

This may further be abbreviated by just:

```
letter ← "a"
letter ← "b"
  .   .   .
  .   .   .
  .   .   .
```

letter ← "z"

and even more condensed by

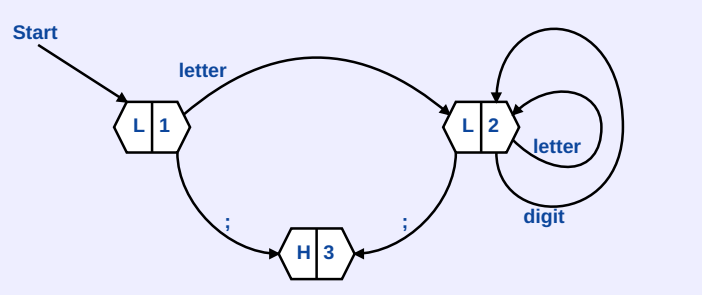
letter ← "a" | "b" | "c" | "d" | ... | "z"

where the | may be pronounced *or*.

And, of course:

digit ← "0" | "1" | "2" | "3" | ... | "9"

So



is an equivalent diagram. And

name ← letter | letter | letter digit | letter letter ...

is also equivalent, except that it is not very concise, because the description is potentially of infinite length. There are two ways to form a new abstraction for this concept.

Iteration

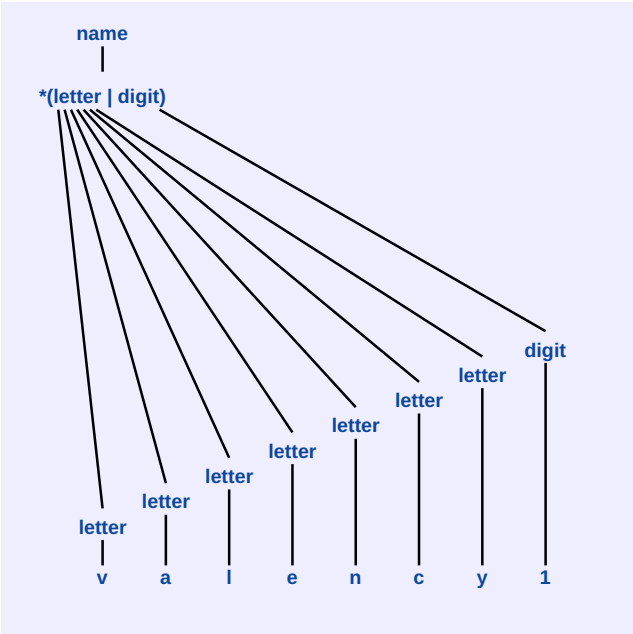
name ← letter | letter *(letter | digit) (1)

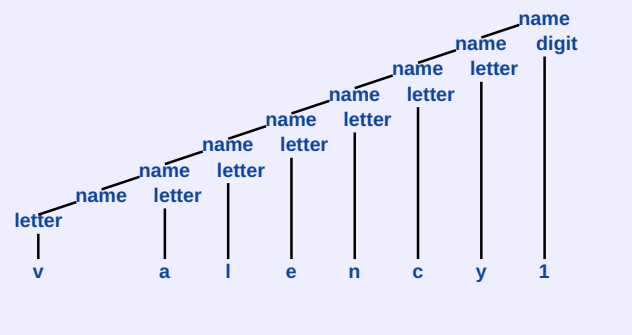
where the * signifies zero to *many* repeats of letters or digits. Note the correspondence between the two alternatives and the state diagram. The other way to represent this situation seems trickier, but is even more fundamental to these questions.

Recursion

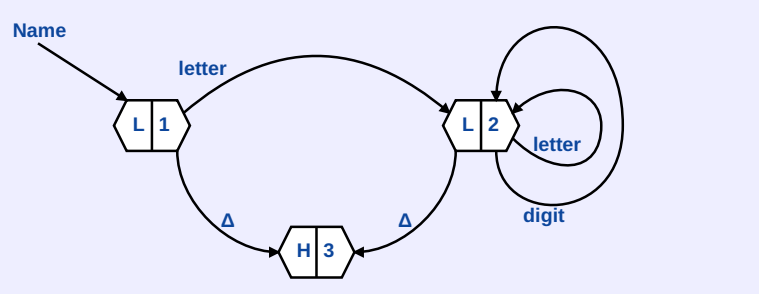
name ← letter | name letter | name digit (2)

This would be a *circular* definition, except for the first alternative. A name *valency1* would be deciphered by the two descriptions in a slightly different manner.



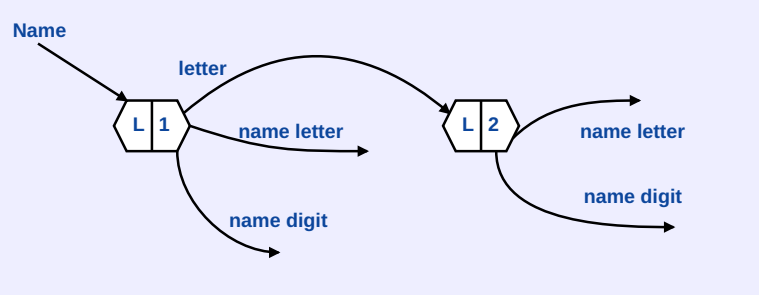


Although recursion isn't necessary to specify the form of a name, there will be cases further on that cannot be described by iteration alone. The above *derivation* or *recognition* trees are called parses of the string *valency1*, according to the grammar (1) or (2). Now it is seen that the *entrance* point to the *f.s.* machine could be called *name*.



Here, the *Δ* stands for all other characters. So to find if the tape contains a name, entry is made to state 1.

The recursive grammar does not have a simple mapping into a finite state machine because it gets into an infinite loop by reentering itself without moving the tape. However, there does exist a very simple mapping on the grammar itself which will cause the right thing to happen.



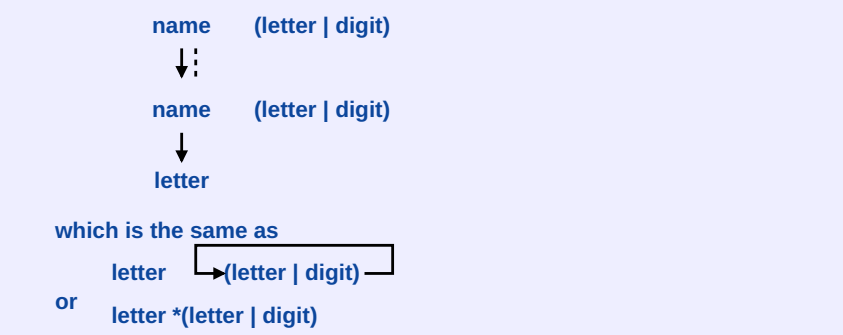
In effect, it is a kind of factoring. Look at (2) again.

name ← letter | name letter | name digit

Combining the last two alternatives:

name (letter | digit)

This gives rise to:



the same as (1).

Turing Machine Description

So far, the issue of how recursion is effected has been sidestepped by playing tricks. Now it is time to consider a grammar which recursively describes strings that cannot be recognized by a finite state machine.

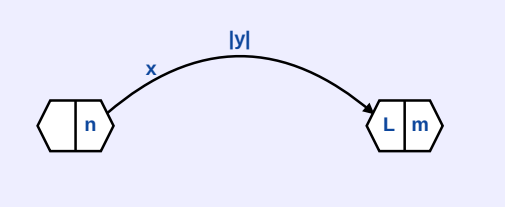
Language ← "x" List_of_parens "x"
List_of_parens ← List_of_parens Parens | Parens
Parens ← "(" List_of_parens ")"

This is not the simplest grammar for these strings, but it does describe the different elements which contribute to their form.

This describes strings like *xx*, *x()x*, *x()()x*, *x(()())x*. The reader will quickly discover that there is an exception to every finite state machine

he uses for recognizing these strings. Something more is needed. To get it, only one more ability has to be added to the logic box and that, of course , is control of the tape.

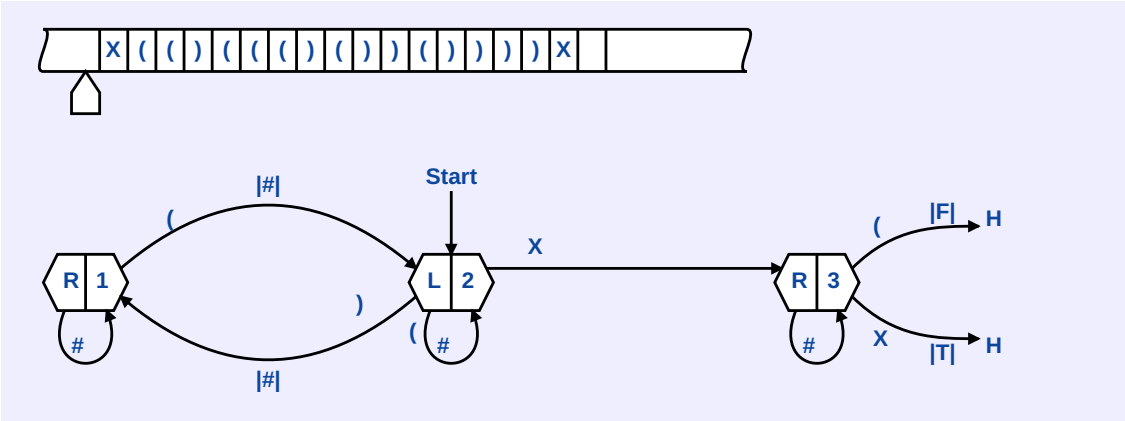
A transition may now be described by stating five things.



name of source state	character to be recognised	character to be written	Movement of tape (L or R)	Destination state
n	"x"	"y"	L	m

Now the new ability may be applied to the previous problem.

A Parenthesis Checker



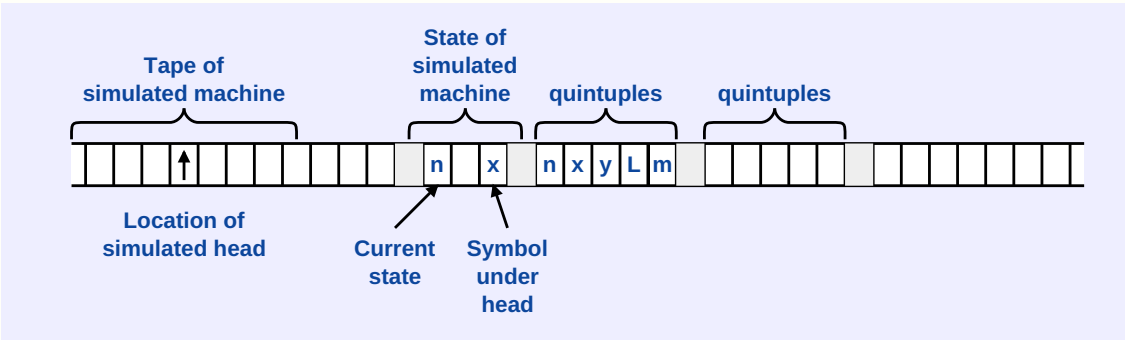
When started, it moves to the right, scanning over *(*'s. When a *)* is encountered, it is replaced by a# and a transition is made to state 1 which scans left copying # until a *(* is found, which is overwritten with# . An *F* (for False) is emitted if the left-hand *X* is found, which means there is at least one extra *)*. When the right *X* is found on the right scan (state 2), state 3 scans back through the #'s to see if there are any left over *(*'s. If not, a *T* (for True) is written and the machine stops.

This machine is clearly equivalent to the grammar.

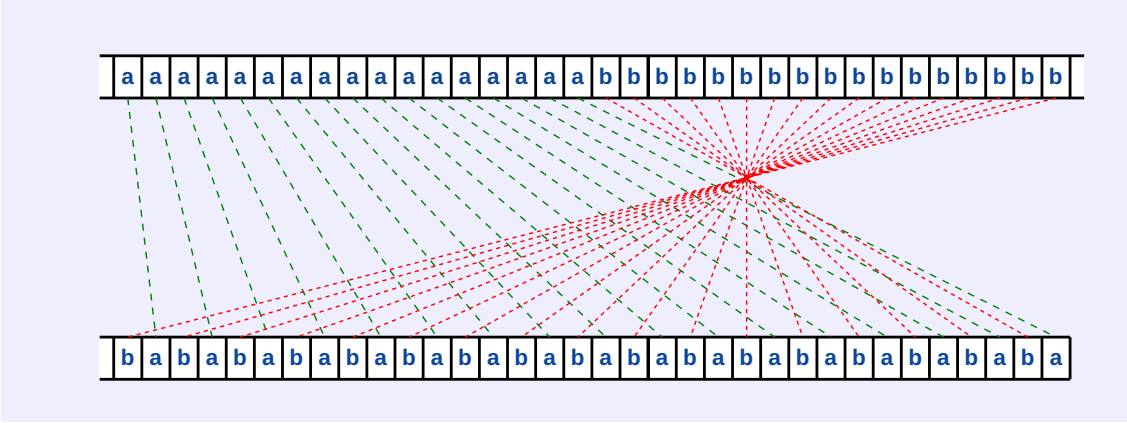
Note that although the number of states is just three, the size of the nested *(*'s is unbounded.

Any Turing machine may be described by specifying the collection of quintuples which determines its *state diagram* and by a finite sequence which represents the initial condition of the tape. Since this is all that is necessary, it is natural to ask about the possibility of *simulating* one Turing machine on another.

A Universal Turing Machine



The entire tape for the simulated machine is the *left* portion of tape of the simulator. Of course, a tape that is indefinite in both directions may be mapped onto one that is indefinite in just one direction in the same spirit that there are countably as many even integers as there are integers. It is done by folding and then mapping onto every other location.



The quintuples representing the automata part of the simulated archive are held on the right-hand tape. All that is left to perform the simulation is for the name of the current quintuple and the character under the simulated *head* to be maintained in the middle and a special flag to be kept in the square under the simulated head.

Now the action of the simulation is obvious. It searches the quintuples for a match on the current state and character. When this is found: *n* | *x*, the remaining three fields dictate the next action. On the way back *M* will be copied into current state. The flag is then searched for and *y* is copied into it.

The real head then moves *L* to pick up the character there and writes in↑ to denote the new position of the simulated head. The new input character is then copied into the current state section of the tape and one *simulated transition* has been completed.

It is not known how *large* the minimum size Universal Turing Machine must be. Minsky cites a four state, seven symbol universal machine as being the most compact representative now known (2).

Conclusion

It is easy to see why finite automata are not used as a model for very complicated processes. Therein lies their weakness, for they have no real powers of abstraction themselves. The reader is directed to Computation: Finite and Infinite Machines by Marvin Minsky for a delightful introduction. However, it is amazing how close compactly thought-out languages are to Turing Machine structure, recursive function theory and production systems. This seems somewhat natural, since all of the schemes mentioned in this chapter were developed for abstract symbolic manipulation. Some of them are more realizable than others on current hardware, just as some formulations are more inherently useful to human beings, no matter what their philosophical equivalence may be.

2. A Simple Interactive Language Interpreter (SILI)

To iterate is human, to recurse: divine! .. L. P. Deutsch

This chapter is a sort of *one-act play* presenting some of the needs, motivations and problems that will be constantly encountered in the rest of the book. A string-oriented macro-replacement language is discussed, accompanied by a number of sketches of some reasonable interpreters for it. The notion of communication with the outside world is also seen to be a natural consequence of the *macro* concept.

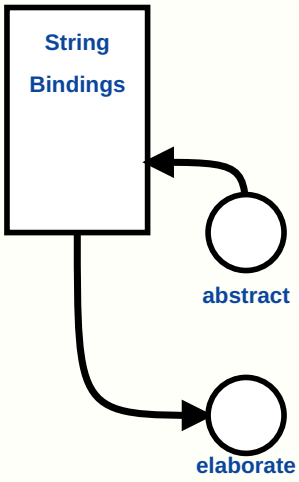
The language has roots in a number of good macro systems (including (7, 8, 9, 10, 11, 12)), but it is somewhat unusual, in that it is *self-extensible*; i.e., both the syntactic forms and the semantic intensions may be dynamically extended by the user.

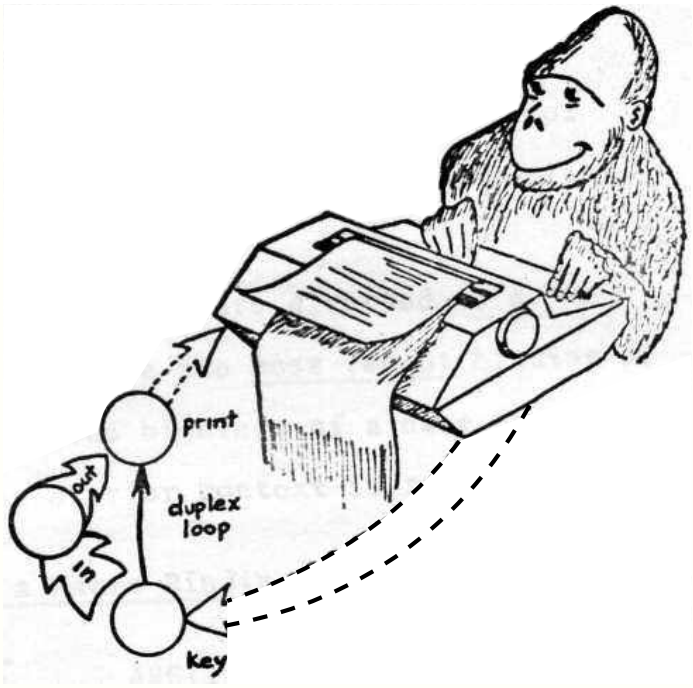
Macros

A macro is a string of text which may be held in reserve and later inserted into another string (or even itself). When bound to a name, it has a very *functional* kind of elaboration, and parameters may be supplied at the time of *macro-expansion*.

Contexts in SILI

Any string contained within *" "* or < > and any character prefixed by *'* is considered to be in a *literal context*. As such, it is treated strictly as a sequence of *raw* text. Numbers composed of digits also stand for, themselves. All other strings are in an *active context* and are subject to elaboration.





in out

key print literal s will echo back to print

key abstract macros are defined and bound

elaborate abstract inner-nested macros are defined as bound

elaborate print elaboration has produced a literal

Directing Traffic in Interactive Macro System

Gratefully adapted from (8).

The drawing shows that a literal string submitted at the keyboard will not only be duplexed back immediately (so the user can see what he is typing), but will also appear again on the keyboard, since there is no other elaboration possible (except to *null* it entirely). Other elaborations will be determined by the initial rules of the interpreter, possibly augmented by new rules supplied by the user.

Bindings of literals to names are diverted to a dictionary which makes available the *most recent* binding of a name when queried. Previous bindings of a name are also saved, then restored when a prior context is re-entered.

Informal Description of a Macro Binding

syntax of parameters

name ← < List:Arg [... ARG(1) ... ARG[3]] > ;

active text

The entire literal string between < > is bound to the name. The body of the macro consists of a syntactic description of its parameters (if any) and possible actions to be taken when recognition is made. The example shown conforms to other macro systems in its form, although greater generality is possible.

A typical call might be:

↓name(X, plus(3,2), Y)

Unlike other systems, *name* is elaborated *as soon as it is determined that it has a binding*, which is when it is recognized as an identifier. The parenthesized list is not touched, since the macros in this system can control the gathering of their own parameters. At this point, then,

name↓(X, plus(3,2), Y)

the macro is elaborated. The first name that is found is *List* (which is a previous binding) and is a syntactic description of how *name* wishes to receive its parameters.

List ← < "("value*("," value)")">

List says that it expects a *(*, followed by at least one *value* and possibly many, terminated by a *)* and separated by commas. *value*, of course, is a further syntactic description that will force elaboration on the list elements so that if X←55, Y←92, then the effective call is:

name(55, 5, 92)↓

Having reached the final *)*, *List* returns to the elaboration of *name*, where the interpreter finds *:Arg*. This means, bind the name *Arg* to the previous context, which was (55, 5, 92). Then *[* is found, which signifies active elaboration of the contents. At some point the subscripted name *Arg(1)* is discovered and elaborated to the first substring of the list, which is *55*. Later on, *Arg[3]* is encountered and expanded to the 3rd character of the list: *,*.

So the contents of strings may be accessed in two ways: by subfields separated by commas and by individual characters. Following the usual conventions:

```
Arg[4,7]   yields      5,92 and
L(ARG)     yields      3   (subfields)
L[ARG]     yields      7   (characters)
```

It may also be assumed that the construction *if < ... > then < ... > else < ... >* exists as a primitive. The string following the *then* will be elaborated only when the conditional clause evaluates to other than *0*. Otherwise, the string following the *else* will be expanded. The exact meaning of these and the other primitive notions will be covered more formally in a succeeding section.

Some Other Elementary Conventions

```
name ← <list: Arg: new_params [ ... ]>;
```

binds both *Arg* and *new_params* to the same thing.

```
name ← <list: Arg, new, thirst [ ... ]>;
```

binds *Arg* to the *first* substring, *new* to the 2nd substring, etc. So, of course

```
name ← <list: Arg (x, y, z) [ ... ]>;
```

also does the right thing by binding *Arg* to the entire list, then x to the first substring, etc.

Extensions

Suppose addition only exists in functional form; e.g., plus(3, 2) yields *5*. This gets tiresome very quickly,so it would be nice to be able to describe the corresponding *infix* form. This can be done by setting up a small set of macros for arithmetic expressions. Define

```
A_Exp ←  <Term:A *( "+" Term:B [plus(A,B)] )>
Term  ← <value:A *( "*" value:B [mul(A,B)] )>
```

This says an arithmetic expression is composed of at least one *Term* (and possibly many, separated by +'s). Each *Term* is at least one *value* (and possibly many, separated by *). This looks like:

```
A_Exp
  ↓
Term      ( + Term  ( + Term  ( + ... )))
  ↓
value     ( * value ( * value ( * ... )))
```

Now include nested parentheses by augmenting the definition of *Term* to

```
Term  ← <value:A *( "*" value:B [mul(A,B)] ) | "(" A_Exp ")" >
```

The | stands for *or*, as before.

Now functions can he defined which expect infix expressions as arguments. However, the general form of the interior of *[]* has *not* been changed in the language. Yet, *A_Exp* is a name local to the user's abstractions, so the interpreter *can* reach it through user-defined functions.

Suppose the syntax and semantics of *[]* were defined as an abstraction in the language itself.

```
block   ← < "[" complete_Exp "]">;
```

Now, if | *A_Exp* could be inserted before the last >, the system interpreter would have to look for *A_Exp* as one of the alternatives for the interior of a *block*. Now the extension is global in that all active contexts will be affected.

Examples

Suppose *plus* didn't exist; then one could pull a similar trick as in Chapter 1 to get *0*, *suc* and +, this time in infix. Assuming the above has been done, define:

```
0    ← ""
suc  ← <value:X [if eq(X,"") then "1" else ε(1,X)]>;
AExp ← <Term:A *("+" Term:B [if eq(A,"") then B else tl(A) + suc(B)]>;
```

Note that + can be used recursively in its own definition as an infix operator! ...

In any case, that is a terrible definition for addition in this system. It is relatively easy to do the *schoolboy* algorithm (which is left to the reader).

Interaction

The user is considered to be supplying text for the elaboration of the *body* of a *block*; i.e., a sequence of *C_exp* expressions. As shown, his previous transactions are duplexed back, then dumped into the *bit-bin*. To him, he is communicating with a statement-at-a-time language like JOSS and CAL. Since he is continuously in a block, all local variables he defines remain in the value bindings dictionary. If his current context is frozen when he logs off the system, these variables will act like permanent files.

It is very apparent that placing the user at the correct place in a recursively-defined system will allow close interaction for free. No additional mechanisms need be devised.

Editing

Since the macro language is a string processor and name-string bindings are global (hence, available to the user), most editing can be done through SILI itself. However, certain simple but useful *keyboard* edits are provided for easily correcting current (and retrieved) context.

The stream of text last typed in is in the range of the edit and is saved by SILI for reappraisal. It can be considered to reside *under the new page* as the user types. If he types in something new, it is really an edit of the old lines and replaces them.

Lines from the dictionary may be retrieved by two operators:

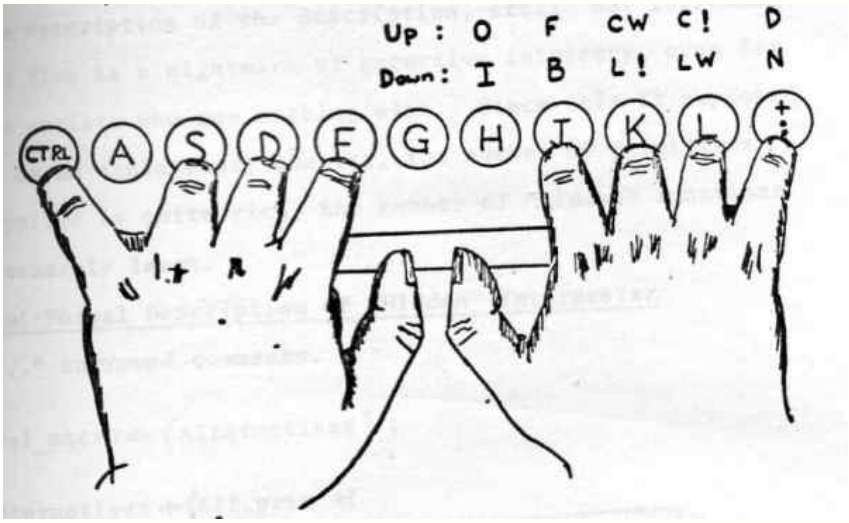
edit name
modify name

Both replace the *old lines* by the literal binding of the name. *edit* will also type that binding for perusal, while *modify* assumes the user is more sure of what he is doing.

Four independent sets of options are provided for modifying the buffered text. They may be

SET	MOVE	TO	DELETE
O(verwrite)	F(orward)	C(har)	D(elete)
I(nsert)	B(ackward)	W(ord)	N(o)
		L(ine)	
		!CHARACTER PATTERN !	

set by a rippling motion of the right hand while the left holds down the *control* case shift.



(K^C, L^C combine to give the four options of the *TO*).

Some typical uses follow.

Command	Mnemonic	Coding
Backspace	B	J^C
Reedit	B	L
Find "ABC"	!"ABC"!	K^CABC K^C
Insert "xy" after "BZ"	!"BZ"!I"XY"I	K^CABCH^{C}XYH^C

Interpretation

As will be seen, almost all of the description of SILI could be made global to the user, with the only expense being time consumed by elaboration. (Actually, something as spare as the LISP-like language in Chapter 1 could interpret *all* the rest of the syntax description, including the description of the description, etc.) But following the flow is a nightmare of recursive intricacy, even for the purists who use nothing else. Since this is supposed to be an explanatory chapter, the number of primitives supplied is quite rich, the number of *hidden* functions reasonably large.

Semi-Formal Description of "Hidden" Interpreter

Comments in green.

```
Eval_macro ← <Alternatives>;
Alternatives ← <Alt_prim *( ' | [ if flag then  else ] Alt_prim)  > ;
                    if an alternate is true, skip the rest
Alt_prim ← < (star | choice | Literal | name: X [ Eval_macro X]
                | *Ω [flag ← true ]
                | Ω) * ( ' : (name | list_names) [do binding])
                    this performs bindings
                * (Block) > ;
                    the execution of "[" "]"
Star ← < "*("  Alternatives [ if flag then  else flag ← true  ] )  > ;
                    this allows 0 to many and leaves flag true
Choice ← < *( Alternatives *) > ;
                    this allows factoring
Literal ← < "char | "" *(char)"" | num | "<" *( *(char) | Literal) ">" > ;
Name ← < letter *(letter|digit) | delim > ;
Num ← < digit *(digit) > ;
Char ← < letter | digit | delim > ;
letter ← < "A" | "a" | "B" | "b" | ... | "Z" | "z" > ;
digit ← < "0" | "1" | "2" | ... | "9" > ;
delim ← < "ε" | "$" | "+" | "-" | ... | "*" > ;
```

Semi-Formal Description of "Exposed" Interpreter

```
Block ← < ' [Cexp # ( ( ',|' ; ) Cexp )'] > ;
Cexp ← < "if" Cexp "then" Cexp ( "else" ) Cexp > ) | Logic * ( "ε" Logic ) > ;
Logic ← < Aexp * ( ( "AND" | "OR" ) Aexp ) > ;
Aexp ← < Term * ( ( "+" | "-" ) Term ) > ;
Term ← < Prim * ( ( "*" | "/" ) Prim ) > ;
Prim ← < name | Num | Literal > ;
```

Functions

ε(x,y) is primitive; ε("ABC,"XYZ) is "ABCXYZ"
Eq(x,y) is a primitive; it returns 0 if the strings x and y are *not* equal.

```
hd      ← < List: Arg [ Arg(1) ] > ;
first   ← < List: Arg [ Arg[1] ] > ;
tl      ← < List: Arg [ Arg(2, L(Arg) ) ] > ;
last    ← < List: Arg [ Arg[2, L[Arg] ] ] > ;
end     ← < List: Arg [ Arg (L(Arg)) > ;
endchar ← < List: Arg [ Arg L[Arg]] > ;
word    ← < '(value: X', value : Y') [ε (X, ε(",",Y) ) ] > ;
... and so on ...
```

word could have been defined using *List*, but it was desired to restrict the number of incoming paramters to two.

Examples

The language LOGO (13) was developed as an experiment for teaching mathematics to grade-school students. It is quite pretty and easy to use. It has no parentheses, labels or jumps and is sentence-oriented. Although (13) does not explicitly give the syntax of LOGO, it is possible to guess its form from the many examples. Because they have a number of things in common, LOGO can easily be described in SILI, as follows:

```
LOGO ← < "LOGO" *(statement "$CR" ) >;... "CR" stands for carriage return
Statement ← < verb_phrase | Expression | Assignation | Procedure | Test  > ;
Verb   ← < "PRINT" Expression:A [type ""A"" ] |'DO" Expression
           | 'IF YES" [ if set  then flag=true]   Expression
           | 'IF NO"  [ if ¬set  then  flag=true] Expression
           | 'RETURN"'  Expression:   A[A]
           | Statement > ;
Assignation ← < "NAME $CR" [ "THING: "] Literal "$CR" : A
                 ["NAME OF THINGS"] Literal "$CR" : B [B←A] > ;
Procedure ← < "TO" Name:A (Thing *("AND" Thing) | Ω) : B *(char): C "END"
            [A← "<"εB ε"." εC ε">" ] > ;
Test ← < "IS" Expression:A Expression:B [ if A = B then set ← true else set ← false ] > ;
Expression ← <  (Thing | Literal):A[A] Name:B "OF" Expression:C * ("AND" Expression:D)
                [C ← C ε ","  ε D] ) [B (C) ] > ;
```

```
Thing ← < "/" CHARS "/">
Literal ← < """ CHARS """>
```

LOGO functions

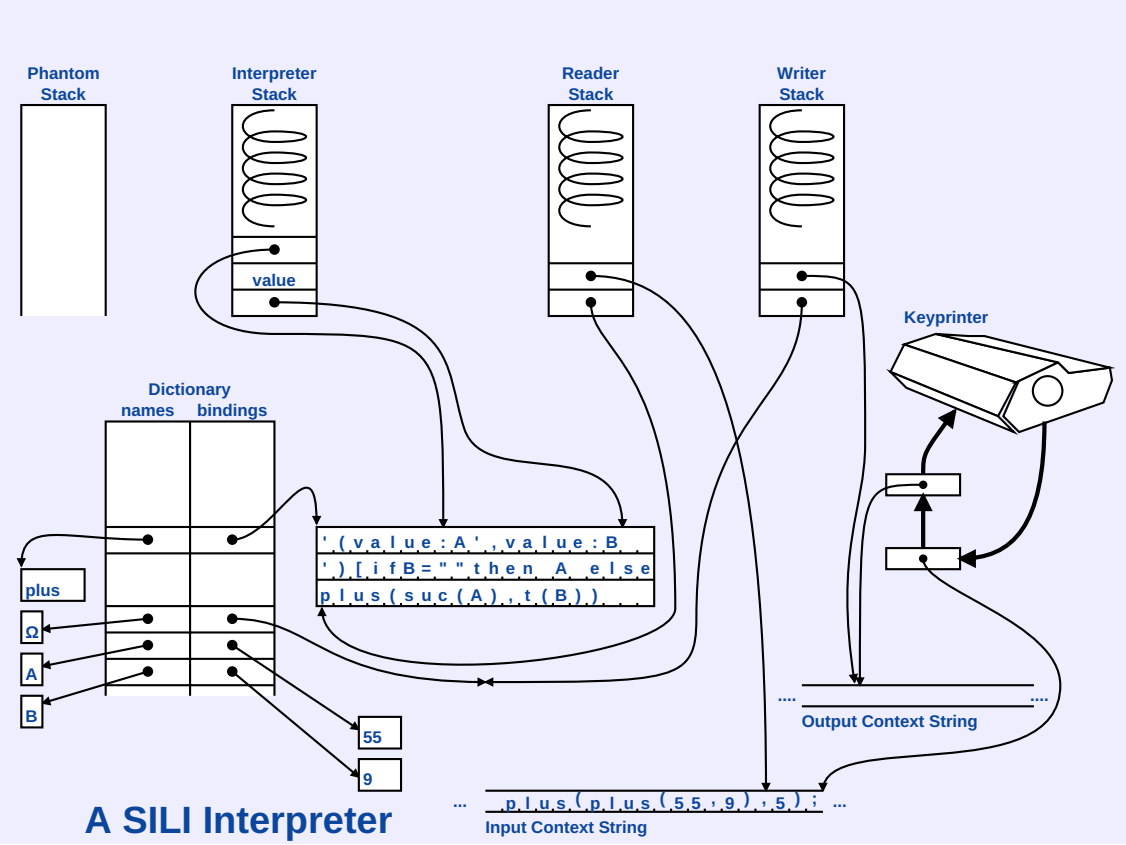
```
WORD  ← < string:A [if A = ""then"" else hd(A) ε WORD (tl(A)) ] > ;
SENTENCE  ← < string:A [if A = ""then"" else hd(A) ε " " ε SENTENCE (tl(A)) ] > ;
```

Now that logo has been defined it may be used

```
LOGO
PRINT FIRST OF "CAT"
     "C"
PRINT SENTENCE OF FIRST OF "HERE WE" AND "ARE US"
     "HERE ARE US"
... now for some Pig Latin samples from (13) ...
TO PIG1 /X/
  DO WORD OF BUT_FIRST OF /X/AND FIRST OF /X/
  AND "AY"
END
TO PIG2 /X/
  DO WORD OF /X/ AND "WAY"
END
... the above takes care of the endings
TO CONTAIN /B//A/
 IS /A/ ""
 IF YES RETURN "FALSE"
 IS /B/ FIRST OF /A/
 IF YES RETURN "TRUE"
 DO CONTAIN OF /A/ AND BUT FIRST OF /B/
END
TO ENDING /A/
 IS CONTAIN OF LAST OF /A/  AND "AEIOU"
 IF NO RETURN PIG1 of /A/
 RETURN PIG2 of /A/
END
... that was for words, now for sentences ...
TO LATINIZE /A/
    IS /A/ ""
    IF NO PRINT.ENDING OF FIRST OF /A/
    LATINIZE OF BUT FIRST OF /A/
END
... so ...
DO LATINIZE OF "THE CAT IN HIS HAT"
    HETAY
    ATCAY
    INWAY
    ISHAY
    ATHAY
```

Informal View of Interpretation

The drawing shows a SILI processor. It is definitely not the most compact description possible, but it does exhibit the structures necessary for macro elaboration in this system.



The *reader stack* maintains all input contexts. Input text is always obtained through the top member. The bottom-most pointer is the reader of the keyboard input text queue.

The *writer stack* is just the inverse, in that all output contexts are exhibited. The bottom-most pointer feeds the output queue, all others write into the binding table. Since every macro has a value upon elaboration (if only Ω = "" = empty), the writer stack follows the top of the binding table so that an outlet for the value of a block is provided.

Notice that *55* gets picked up during the demand for *value* in the call of *plus*. The writer for *value* parks the *55* in the top of the stack. No name binding is given at this time. But the very next thing that is done upon return to the *plus* expansion is to find *:A*. The *:* binds *A* into the blank name position. When *value* is entered again, the *9* is tucked into the new top of binding table and *B* is bound into it in a similar fashion on return.

The *phantom stack* is for the hidden part of the interpreter. It contains return addresses, etc. for the re-entrant hardware *mill*.

Efficiency

Readers may be wondering whether the SILI machine will just grind to a halt when exercised. Even ordinary macro expanders are not famous for blinding speed. This one has an additional layer of recursion and interpretation because routines compile their own parameters.

The situation is not as bad as it looks, providing the implementer gets to solve *all* of the problem.

About 70% of the time of any string interpretation is spent in reconciling the differences between inadequate hardware and over-adequate needs. A string-oriented memory (not just byte-oriented) would go a long way towards bringing SILI up to speed.

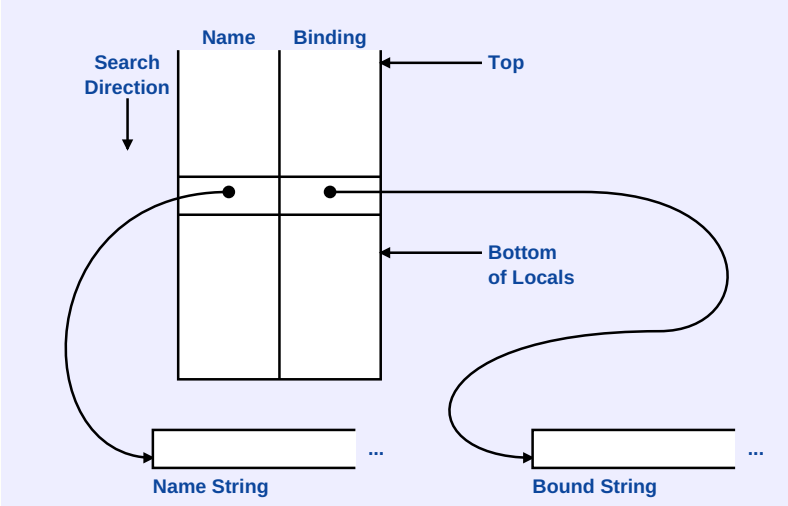
A micro-coded machine (such as Interdata IV) could do reasonably well with the algorithmic part of the interpreter. Note that all bindings really form programmed operators (*POP*'s). Some trap to the micro-code hardware, the rest undergo interpretation. In either case, however, the structure is uniform, thus requiring only one algorithm for all cases.

A third tack is to find and improve bottleneck situations; e.g., searching the dictionary for bindings. This can be done much faster using hashing techniques, as will be explained later. Another grinding point is the execution of branches. If the text itself were structured so that there would be actual branches reflecting the *then* and *else* forks, a great deal of lost time would be made up. This will be covered in more detail in section 3.b.

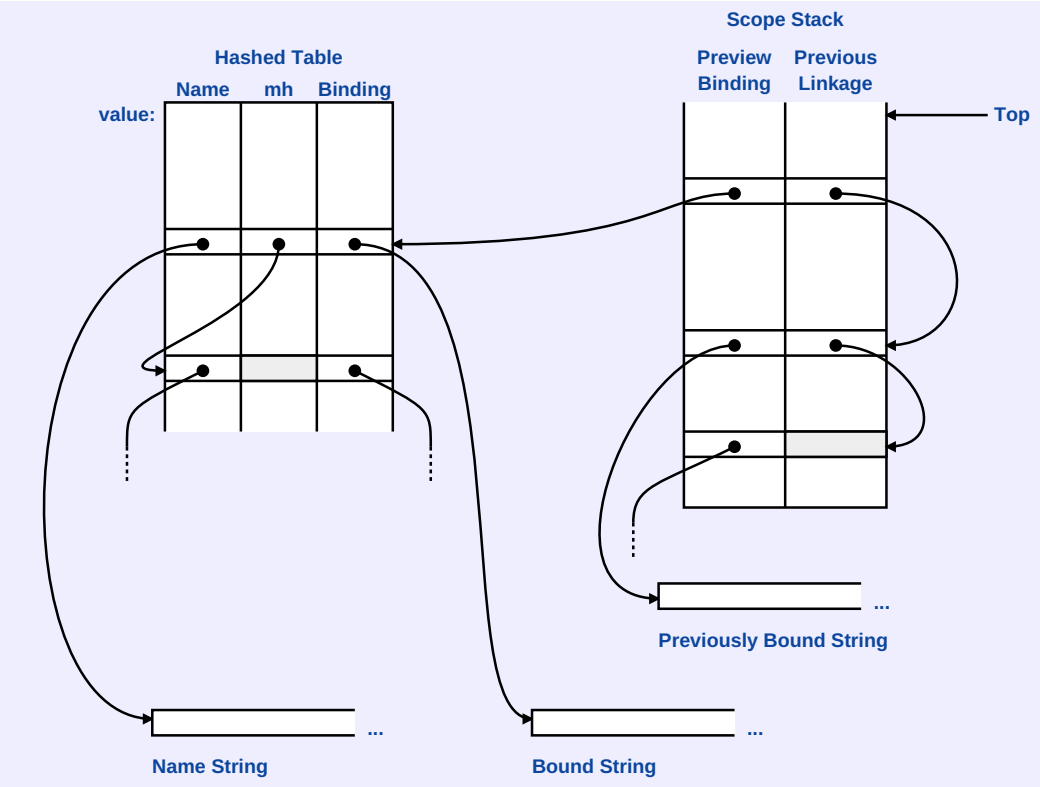
Still another area for improving throughput is to find parallelisms inherent in the interpreter, then to build separate machines into a pipeline processor with very little lag in each box.

Binding Lookup

Although hardware associative memories are expensive, there is no doubt that something must be done to improve the backwards linear search that was being done on the binding tables.



The problem is that scope checking must still be done last-in-most-recent-binding-basis.

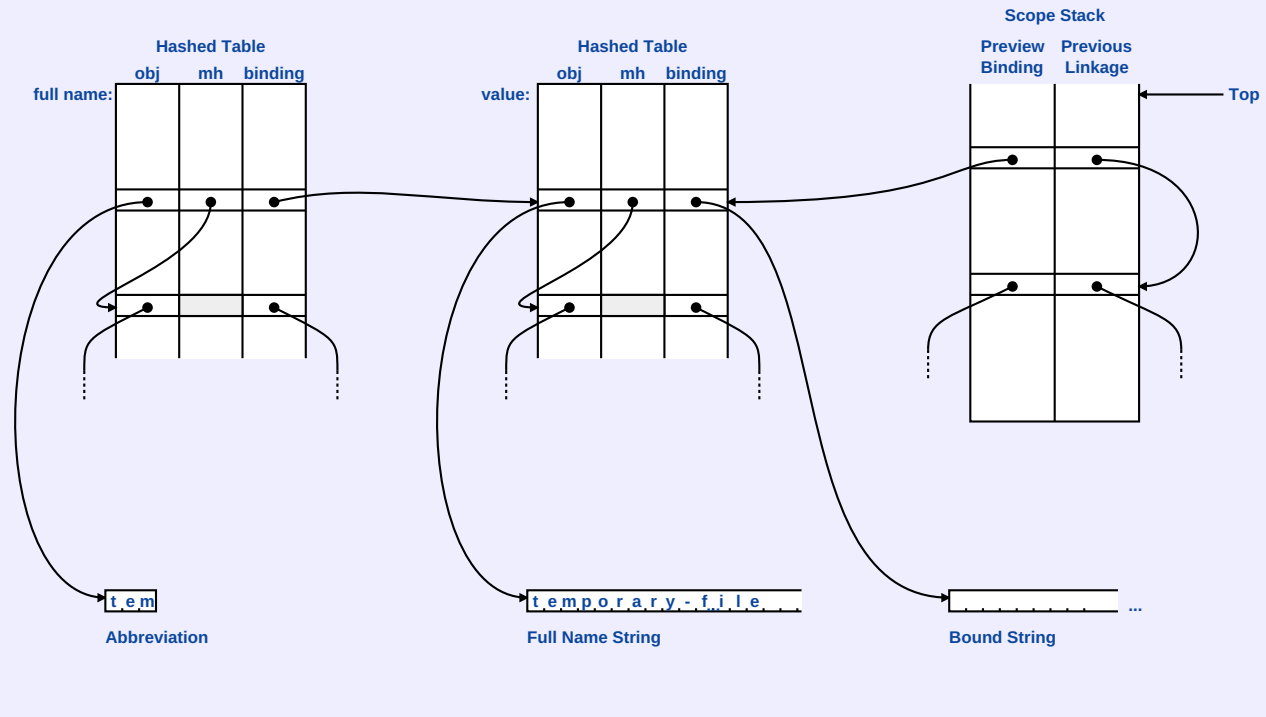


Hashing will be covered thoroughly in the chapter on memory systems. For now, it can be considered as a way to evenly distribute large names over a small area. Suppose the hash table has 256 possible slots. Now the name is *scrambled* somehow and taken *mod* 256. This provides an index into one of the slots. If the hash worked perfectly, then 256 arbitrary names could fit into the table without conflict. However, there is a good chance that one or two names will find slots already full. This is what the *mh* (multiple hit) column is, for it points to other names which have the same hash.

Each name entry in the hash table is unique and the entries form the set of all global names. *name_34* is an example. Its current value is found in the *binding* column. It also had a previous binding, as indicated by the pointers in the scope stack. When interpretation moves out of the current local context, all entries between the *top* and *bottom* pointers will be removed.

If the left column contains a pointer it means there was a previous binding. This is substituted for the current binding. If the column is blank, it means the name has no prior bindings and may be removed from the hash table.

A good hashing function can practically guarantee that there will be less than 8 double hits for 200+ entries in a 256-slot table. That means only two memory cycles will be needed to retrieve a binding 94% of the tine: almost as good as hardware!



Symbol Anticipation and Scope Checking

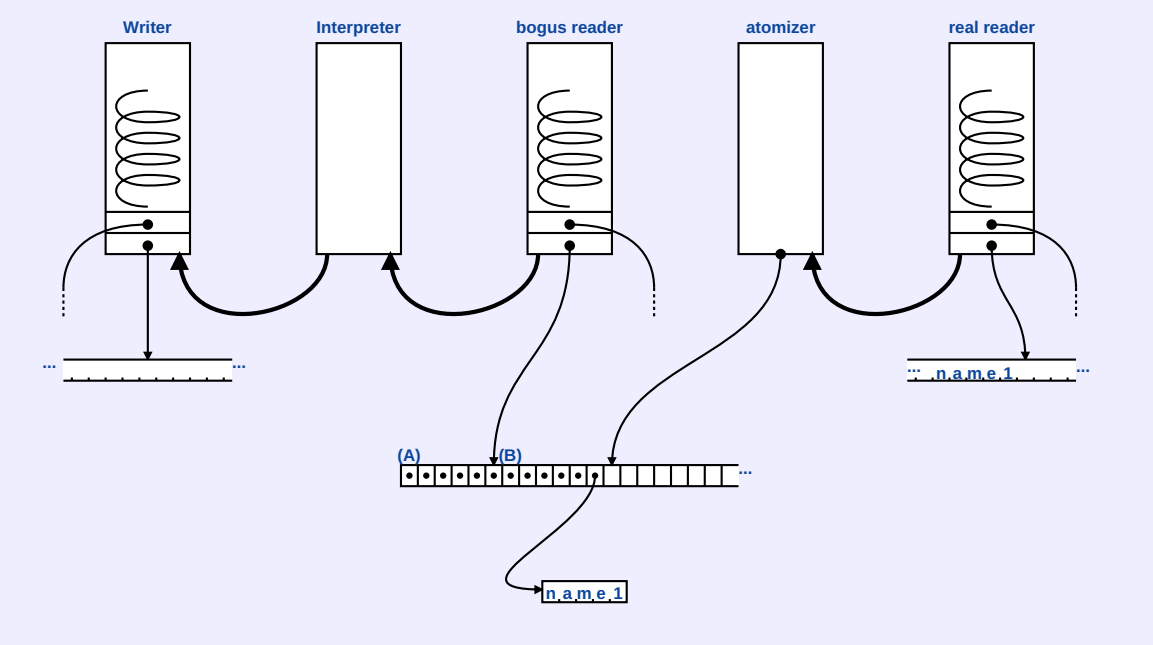
Another trick that can be done to encourage the interacter to use large, readable symbols is to provide an *abbreviation service*. Hashing again provides the vehicle. For any collection of names, there is a unique abbreviation for each one. If there are names *tepid* and *temporary-file*, then the abbreviations are *tep* and *tem*.

As each name is bound into the dictionary, it is hashed to see if there exists an abbreviation. If there does, new abbreviations have to be made up and entered. If not, the first character of the name will serve.

Now, as the user types in, the system will *hash away* at the incoming text, looking for a recognizable abbreviation. When it finds one, it *takes off* and supplies the rest of the name from the value hash table. New locals can be started by a special delimiter, say *' '*. This will cue the interpreter to enter new context into the dictionary.

Parallelism

If the grammatical forms are restricted enough, then the business of *atomizing* the text into names, numbers, delimiters, etc., can be done currently with interpretation. A buffer of atomized strings is provided between the reader pointer and the input to the parser.



Even better, two queues can be maintained when branch points are reached. The interpreter simply decides which queue to pick. Because the basic symbols are not nested strings, the recognizer can be a finite state algorithm, and thus eliminate recursive procedure linkages in a very critical area. Also, the amount of back-up permitted can be the number of symbols between (A) and (B).

Other Speedups

Most of the predefined operations in the interpreter can be buffered with the parser.

Strings need not be copied if they are only going to be read. A pointer to the string will do as well. Intermediate results (since they are not named) must be read-only, and the same dictum applies. The differences between *copies* and *instances* will be covered in great detail during the discussions of the FLEX language.

Tops of all stacks may be hardware registers, since three of the four stacks may only be accessed at the top.

The *hidden* parser could be bottom up to greatly increase speed of the predefined language.

These and like considerations are covered extensively for the entire rest of the book. Those who might suspect that SILI and FLEX are more than first cousins are entirely right, for FLEX is a further generalization oriented toward parallel processing, arbitrary modeling and mapping.

3. The FLEX Language

You would touch with your fingers the naked body of your dreams - K. Gibran

FLEX is an acronym for *FLexible EXtendable*. It was first used to denote a very Euler-like (15) language that was devised in the summer of 1967 to answer a need for a *higher level* system which could easily be interpreted by hardware. The lines of development were strongly suggested by the evolution of the B5000, B5500 (16) and its close ties with ALGOL-58 (17) and (more loosely) with ALGOL-60 (18).

There were several important differences, however. FLEX was always intended to be interactive. From the first, the hardware interpreter was to have a *scope* for graphical (as well as textual) output. As a generalized vehicle for modeling situations, there was a strong need for the ability to create logical *instances* as well as *copies* and to simulate running them concurrently. Neither ALGOL-60 nor Euler and its further generalizations (19) cared to tackle these problems. An ALGOL-60 deviant, SIMULA (20) made a notable attempt to describe concurrent processes, but was tremendously hampered by the statically-nested structure of ALGOL, which it adopted.

Structure of some kind (and control of it) was definitely needed. The initial delights of the strongly interactive languages JOSS (21) and CAL (22) have hidden edges to them. Any problem not involving reasonably simple arithmetic calculations quickly developed unlimited amounts of *hair*. While interactive conversation was pleasant, since it was transacted in the same language in which one programmed, there was a great lack of essential touches for a supposedly *kind* system. The editing features of JOSS are almost nonexistent. While CAL is better (it uses QED line editing), the very apparent string-handling capabilities cannot be used in the language itself. Neither language allows local variables (except for recursions on function calls). All the user's names are in one *pot* so that large problems are just unthinkable. Although BASIC (23) has a structure somewhat like FORTRAN, it is not really interactive, since the user transacts business in a simple *operating-system-like* manner, and the response is very much like a dedicated batch system.

Most other higher level, so-called *interactive* systems were like TINT (24) (a variant of ALGOL-68) in which the reactions with the system are very much like BASIC. The user is stuck out in a corner, viewing his world through a very narrow porthole.

A notable group of exceptions to all the previous systems are Interactive LISP (found on the Q32 (25), SDS-940 ((26), PDP-10 (27), etc.) and TRAC (28). Both are functionally oriented (one list, the other string), both talk to the user with one language, and both are *homo-iconic* (28) in that their internal and external representations are essentially the same. They both have the ability to dynamically create new functions which may then be elaborated at the user's pleasure.

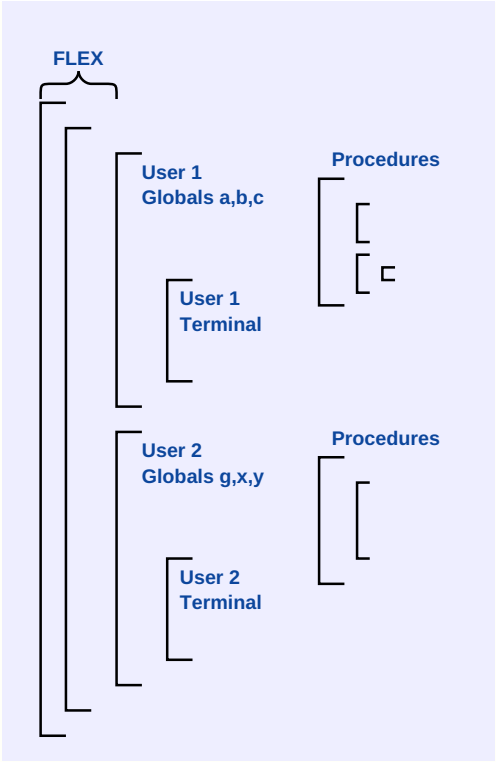
Their one great drawback is that programs written in then look like King Burniburiach's letter to the Sumerians done in Babylonian cuneiform! LISP has several other difficulties. Although it is a functional expansion language, a *feature* called PROG was added as a *pseudo* function to allow sequential execution and conditional branching. Unfortunately, change of control is effected by FORTRAN-like GOTO and labels, which is quite foreign to the spirit of LISP.

Because of the internal structure of LISP and the basic urge of humans to stir up any broth, other *features* were gradually *tacked* on until somewhat of a kludge resulted from one of the most elegant languages ever devised.

FLEX in 1968

The summer of 1968 saw a second FLEX language (29) (still resembling EULER) and a machine design for carrying out interactive dialogues with the user. It had some virtues and many more defects.

The language had static nesting (like ALGOL), and the interactive user was considered to be a local block, with the FLEX interpreter as the most global block and his program, etc. to be parallel blocks.



Various users could interact with their contexts protected by the parallel block structure. *Own* variables did not exist and *co-routines* were used, instead of procedures. Files were subsumed by the ability to use the co-routine exit *leave* to *log off* the machine, thus preserving state until next *log on*. Co-routines could be *activated* as well as *called* so that many parallel instances could be in the interpretive *mix*.

The terms introduced in this section will be explained in succeeding sections, so don't worry.

The interpretive structure was quite *conversative* and used syntax tables in *read-only* memory (ROM) to effect a *bottom-up* translation into polish *postfix* code, which was separately evaluated by a stack interpreter programmed in micro-code.

The rigid table structure required (if other compilers were to be written) a separate language called SCRIBE which described the bottom-up translation in a manner, very much influenced by FSL (30). It used a simple pattern natch for syntax recognition and used FLEX for expressing semantic intent. Later (unpublished) the pattern match was incorporated into the FLEX language as a conditional (like *if*). This expanded the size of a table entry from 8 (8-bit) bytes to 13 bytes, but eliminated more than 5,000 bits of the special table interpreter, which effected a long-run saving.

This machine is sketched in Part B, since it was workable and contained a number of interesting features.

The most traumatic discovery made by the author during the development of this machine was the incredible dependence of performance (and more important, of just being able to stuff the design into hardware) on the structure of the memory system. More than 75% of all the designing effort went into the memory system (that is a conservative estimate). It turned out to be the least satisfying and least elegant part of the design. This came about partly because there is very little known (in any formal way) about memory systems, compared to the wealth of information on grammars. The other great bottleneck has to do with the inability of memory manufacturers to realize that a randomly accessed but consecutively addressed core memory has very little to offer except to hold large arrays. Since almost everything that is done on a computer does *not* involve static arrays, there is an obvious incompatibility. Section 6 on memory systems presents a number of different ways to get around these problems (none of them wholly successful).

The language had other serious bugs, many of them revolving about the problems arising from static nesting in a non-nested (concurrent) environment. Some, but not all, have been fixed in the current FLEX.

FLEX in 1969

Why keep on calling it FLEX? Why not FLEX I, FLEX 2.2, or *son of FLEX*? First, because there is no wish to create a dynasty or *Black Museum*, as has happened with other languages. Second, because of its very nature, FLEX should be able to change its form without trading in its name. And third, it is really the author's own private vehicle for investigation into these areas and is (and will be) a project not considered closed.

The summer of 1969 sees another FLEX (and another machine). The goals have *not* changed. The desire is still to design an interactive tool which can aid in the visualization and realization of provocative notions. It must be simple enough so that one does not have to become a systems programmer (one who understands the arcane rites) to use it. It must be cheap enough to be owned (like a grand piano). It must do more than just be able to realize computable functions; it has to be able to form the *abstractions in which the user deals*.

These goals have not been reached.

However, something anticipatory of them has arisen which can serve as a model for future interactive designs. It needs to be pointed out that FLEX is not a general purpose language, any more than a computer is a general purpose machine. The goals of generality, simplicity and low cost somewhat preclude any great abilities in solving sets of partial differential equations. However, the program to do the solution can be *debugged* with less tears than on other systems.

FLEX is an *idea* debugger and, as such, it is hoped that is is also an idea *media*.

3.a. Translation

THINK RECURSIVELY. Recursion can be avoided with PROG, but people who use PROG too much will never get to Heaven.-- L. P. Deutsch & B. W. Lampson SDS-930 Lisp Manual

In this and succeeding sections, the reader may find it helpful to refer to the detached Appendix A: The FLEX Handbook. It contains both an informal and formal description of the language and may be examined concurrently with these descriptions.

By now, the reader should have an intuitive feel for the ins and outs of string handling and pattern recognition. Since this is an area that has been strongly wooed (practically raped) by Computer Science, not much of a formal nature will be mentioned, as there already is an abundance of such literature. However, these documents should not be ignored and this chapter will use them heavily.

There are two sides to the translation coin in this study. First, there is a wish to translate and interpret FLEX, the kernel language. Second, these abilities should be made available in a parametric (and recursive) fashion so that they can be used dynamically by the interactor and his programs. In fact, this is a paramount consideration, since FLEX programs would quickly become incomprehensible if there were no way to easily form new syntactic abstractions. The user would be reduced to *gearing* them up from scratch (as is done now). So tools should be provided.

RECOGNIZERS

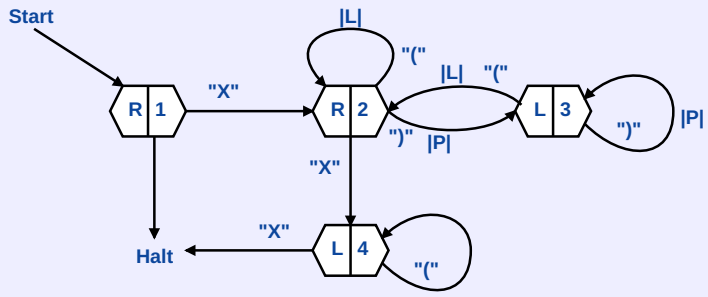
(A)
```
Language ← "X" List_of_Parens "X"
List_of_Parens ← Parens | List_of_Parens Parens | Ω
Parens ← "(" ")" | "(" List_of_Parens ")"
```

(B)
```
                                                          scan;
Start:  |                          "X" ‖                    scan;
Nest:   |                          "(" ‖                    scan;
[*]:    |                      "(" ")" ‖ ⇒ Parens           ;
        |    "(" List_of_Parens ")" ‖ ⇒ Parens             ;
        |  "X" List_of_Parens "X" ‖
        |                  "X" "X" ‖
List:   |  List_of_Parens Parens ‖ ⇒ List_of_Parens;       scan;
        |              Parens ‖ ⇒ List_of_Parens;          scan;
```

(C)
```
Language ← "X" List_of_Parens "X"
List_of_Parens ←(Parens *(Parens) )
Parens ← "(" (List_of_Parens | Ω ) ")"
```

(D)
```
Language ← "X" List_of_Parens "X"
List_of_Parens ← ( "(" (List_of_Parens | Ω) ")" *( "(" (List_of_Parens | Ω) ")" )
```

(E)



There are two basic questions which need to be examined before a direction of approach is chosen:

To compile, or not to compile ...? and *top down, or bottom up?* Discussion on the first point will be deferred until later. As for the second, there is not a great deal of difference. To take a case in point, consider the, parenthesis grammar of Chapter 1.

The fold-out.illustrates top down and bottom up strategies. (A) is the grammar from Chapter 1. Since it has a left recursion in line 2, it is of little use as a recognizer directly. (B) is a Floyd-Evans-like (31, 32, 33) recognizer. The * matches anything and is simply, a device to cause the initial scan into the stack of a character which (line *Start:* says) must be an "X" or error_1 is caused. The ‖ signifies the top of the stack. When a *stack picture* (between |, ‖) matches the stack, the actions to the right of‖ are performed; otherwise the next statement is executed, which may be an *error* or the next line. The ⇒ means replace the *matched* portion of the stack with the symbol that follows. *scan* enters the next character into the top of the stack. *to* is the jump operator. At the point where the action has been frozen, the recognizer is at line *[*]*. It has just made a successful match and is about to replace the *"(" ")"* with *parens*. It will then be able to match the line *List:* and so on.

(C) is the "factored" version of the grammar (A) . There is no possibility of an infinite loop, since the only recursion is *protected* by requiring parentheses to enter it (line 3). The equivalence between the two schemes should be obvious when its stack is examined. The *reader* shows where the return point is in each entered routine.

(D) is simply (C) with the body of line 3 substituted for *Paren* in line 2. It saves return notices at the expense of being less readable.

(E) is a. slightly more exotic parenthesis checker than the one in the first chapter, since it tries to mirror the other methods. *P* stands for *Paren*, *L* stands for *List of Parens*. As with the others, it must start with a *X*, then it *eats (*'s until a *)*. This it replaces with a *P*, then scans backward looking for a match. The action of turning all *L*'s to *P*'s is the analog of line 3. When the matching *(* is found, it does the analog of line 2 by replacing *P* with an *L* and scanning forward again. State 4 does Line_1 by making sure there are only *List_of_Parens* (*L*'s) or *Ω* between the *X*'s.

So (B) and (E) are *bottom up*, (C) and (D) are *top down*. Note that no back-up is required for any combination of *Language*. This is because there was always enough information to indicate the bracketing. (Which is natural enough, considering the form!!) This is a type (0,0) language, so-called because 0 look-ahead or look-back was necessary to make a decision.

```
                                    RECOGNIZERS
(A)   AExp ← AExp "+" Term | "+" Term | Term
      Term ← Term "*" Primary | Primary
      Primary ← name | number | "(" AExp ")"

(B)                 |          name ‖ out(name)          ; ⇒ Primary
      Start:        |        number ‖ out(number)        ; ⇒ Primary
      Nest:         |           "(" ‖
                    |           "+" ‖
      P:            |  Term "*" Primary Δ ‖ out("*")     ; ⇒ Term
                    |       Primary Δ ‖                  ⇒ Term
      P1:           |        Term "*" ‖
      [*]:          |  AExp "+" Term Δ ‖ out("+")        ; ⇒ AExp
                    |       "+" Term Δ ‖                 ⇒ AExp
                    |          Term Δ ‖                  ⇒ AExp
      A:            |        AExp "+" ‖
                    |       "(" AExp ")" ‖               ⇒ Primary
                    |       "#" AExp "#" ‖

(C)   AExp ← ("+" | Ω) Term *("+" Term [out("+") ] )
      Term ← Primary *("*" Primary [out("*") ] )
      Primary ← name | number | "(" AExp ")"
```

The illustration above shows an example of an (0, 1) language, indicating that a look ahead of one character *must* be made in order to decide what to do. The look aheads are at lines P1: and A: in B . Note that *A + B * C* is ambiguous without it; that is, it could mean

(A + B) * C (1)
or
A + (B * C) (2)

The usual precedence conventions wish for (2) to happen. This means that the decision to transform

```
 A   +    B
AExp +   Term
```

depends entirely on whether the next character is * or +. This major branch point is at *P1:*.

Back-up is more explicit in (C).The *call-like* nature of the program tests the * in the **(* before the *+)*. If either is unsuccessful, the preceding entry is still *true* and the recognizer must back up one character. It is important to see that *both* versions actually back up. (B) is just easier to see. It shoves everything down one notch, then ignores the top of the stack by using *Δ*, which will always match. Then it peeps, when necessary. Despite the fact that back-up occurs, these are both called deterministic recognizers, since no change of flow actually happens. It is interesting that a (1, 1) recognizer can recognize (m, n) languages (34). (C) is even more powerful, since it can make use of the entire history of the parse, so it is called an LR(1) recognizer - for Left Right:(1) look ahead. An LR(1) can recognize LR(K) languages, providing enough factoring is done (34).

Now notice the outputs. They are exactly the same polish *postfix* (operators occur after operands) string. The bracketing is

```
A  B  C  *  X  Y  +  *  +
   -------  -------
   -------------------
   ----------------------
```

If the *out* functions were placed to operate only when a complete right side was recognized - i.e., at lines *P:* and [*] :

P: | Term "*" Primary "Δ" ‖ out (Term, Primary, "*");...

The polish would have the following bracketing:

```
 B   C  *  X  Y  +  *  A  +
 -   ----  -  ----
 --------  -------
 --------------------
 ---------------------------
```

This is optimal for a single address machine, viz:

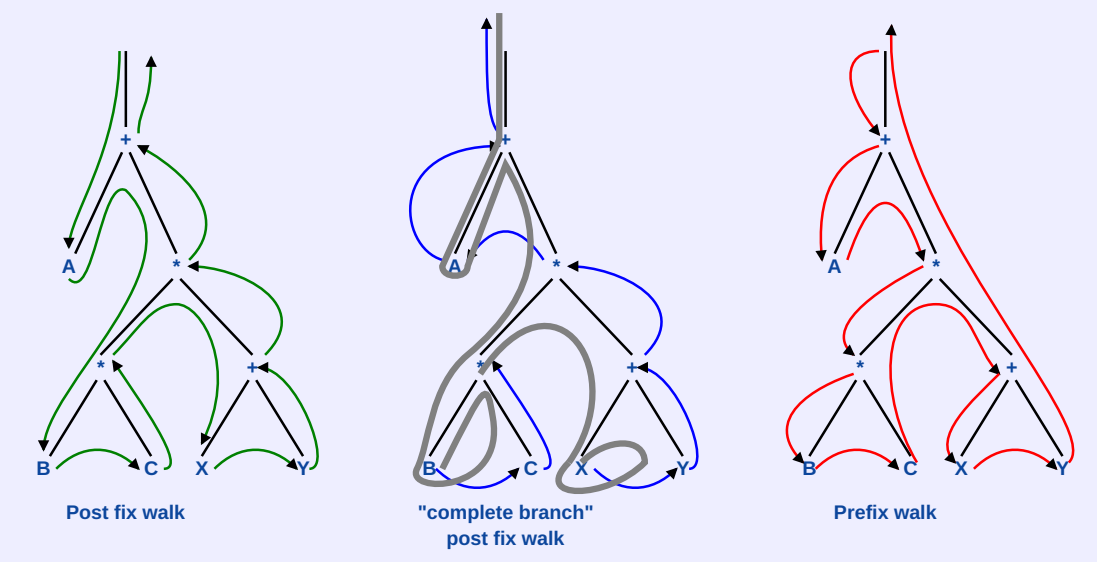
```
LOAD   A1, B
MUL    A1, C
LOAD   A2, X
ADD    A2, Y
MUL    A2, A1
ADD    A1, A
```

where A1 and A2 are registers which may do arithmetic.

Suppose the *out* functions were located at the look ahead spots to*out* the operators; the *out* functions at *P:* and **:* just writing the variable names. Then:

```
+ A * * B C + X Y
      ------- -------
  - ------------------
-------------------------
```

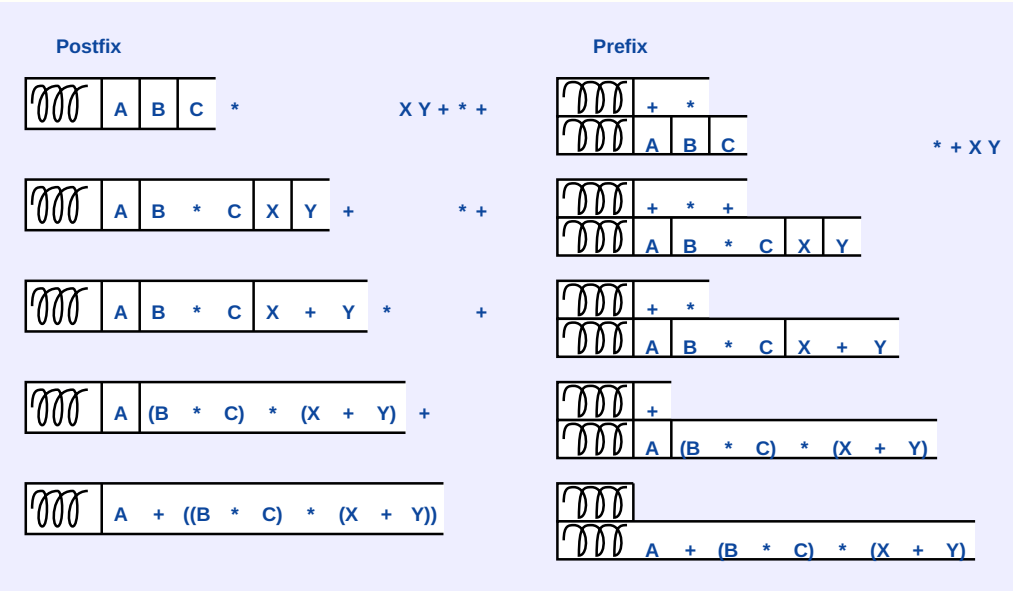
in *prefix* form. Suppose now that the expression is examined as a *dependency tree* showing the bracketing.



Outputting the contents of something which has been reduced is assumed to be the empty string.

The algorithms for doing the left and the right*walks* should be obvious. The middle walk is a bit more devious, and there is a corresponding *complete branch* prefix walk that is very similar.

The recognizing algorithm builds a more complicated tree (all the intermediate symbols are nodes also), but since examination is made only by the *out* function at *interesting places*, the parse is equivalent to the *binary* branching trees shown. Changing the position of the *out* is equivalent to doing a different walk. A parse that proceeds as shown, *folding up* the leftmost things first, is called the canonical parse, and recognition algorithms that can guarantee this are valuable. The trees can be permuted by rotations about the nodes without changing the semantic context, because of the commutativity of the operators. This would not be true if, e.g., a + were changed into a -. Then the order is quite important.



Elaboration of the Two Schemes

It is interesting to note the different kinds of information provided by each scheme. In the figure, the *values* are fetched into the stack when an entry is made. Because the postfix walk visited the *leaves* first, only one stack is needed; the operands are already there when the operator arrives. The prefix form clearly needs two stacks, since the operators precede the operands and need to wait until the correct values arrive.

On the face of it, the postfix form seems to be more economical of time and space. This is the scheme used by most stack machines and interpreters (including Euler). Notice, though, that operands must elaborate *without knowledge of their operators*. Suppose *A* is itself a stack. How does it know whether to *push* or *pop* a value at that point? It doesn't. An answer to nasty situations like this is to let *A* be a *reference* to its value and wait for elaboration until the operator comes along. But this is the *same as prefix*, since no values are fetched until the operator arrives. There is one catch. One of the references is buried under the top operand (*X* in line 2 of the example). It might be a function which needs to use the top of the stack for its own elaboration. The other operand is in the way, so it must be saved. It could be in another stack, or the top two elements could be swapped (which needs a flag set for non-commutative operators).

The second line on the prefix side has no difficulty. The operator that will be applied is already on the top of the operator stack. It can be

determined that *X* should yield a value *before* Y is admitted. If it is a function, there is no problem for space. Prefix also allows control of elaboration by the complete hierarchy of operators in the expression, since they must (by definition) already reside in the operator stack.

The language SILI expands its macros in prefix form, since all operators (and procedures) elaborate their own operands. Much more information is available with prefix execution at the cost of extra time and space (as might be expected).

Compilation or Not?

Since either parse generates output in linear order, it makes no. semantic difference to execute the output string directly or to cache it in storage somewhere for later elaboration.

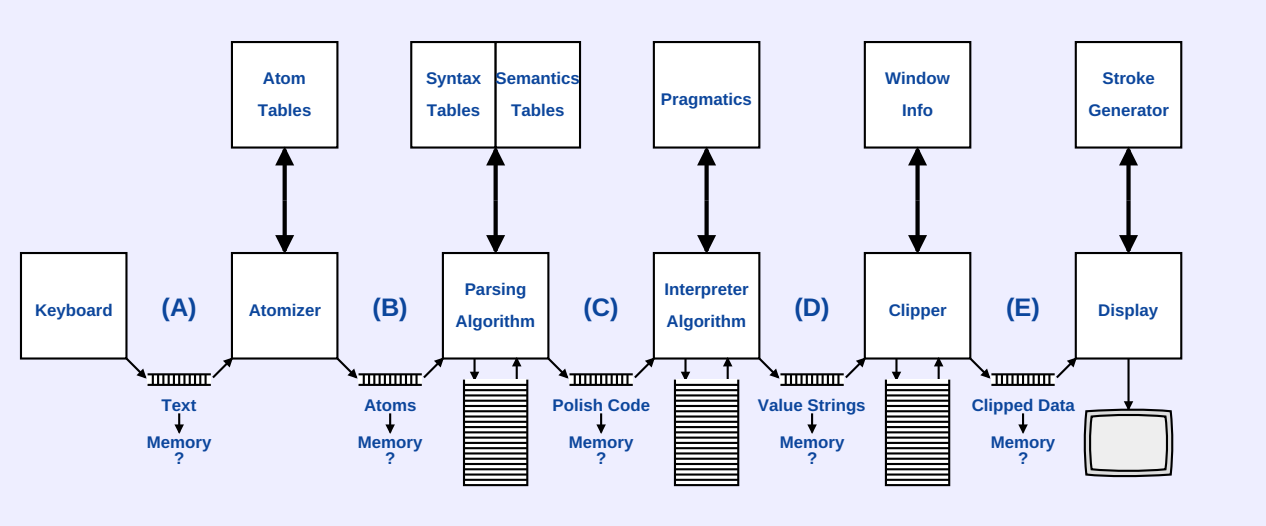
The pros and cons seem quite apparent: executing compiled code is faster and more efficient ... so say most people. Actually, this whole question depends greatly on how much knowledge can be gathered at *compile-time*, as well as the degree of concurrency and buffering available in the hardware interpreter (as was partially seen in Chapter 2).

3.b. Deferment and Coercions

> *When you begin to compose, do it mentally. Do not try the piece at the instrument until it is finished. If your music comes out of your inner self, if you feel it, it will he sure to affect others similarly. - Robert Schumann On Music and Musicians*

Previous sections have shown how it is possible to translate and interpret with one basic set of algorithms, which is necessary to establish formal notions of completeness. Yet, the sequential dependence of the routines for scope, bracketing and structure and interpretive elaboration, have pretty much precluded the idea of directly running from text. Instead, redundancy is removed from the source string to varying degrees in order that eventual elaboration may be done as swiftly as possible on sequential machines.

The illustration below shows the sequence of operations that is typically found in a translation system.



Text is entered and fed through a routine which finds *atoms* i.e., names, numbers, reserved names, etc. These aggregates are looked up in a table and assigned an *internal name*: a small integer of some kind. This can be done independently if the *atomization* does not depend on the gross syntactic structure of the program. For example, *begin* and *[b|e|g|i|n]* are two completely different symbols in publication ALGOL. Constructions such as

begin begin := A + 1 end

are legal. When typed in on a standard keyboard entry device, the string appears as

```
b e g i n  b e g i n  : =  A + 1 e n d ;
```

The question as to whether the *begins* start compound statements or are identifiers *cannot* be decided until the parse. This is very annoying. Some ALGOLS use special delimiters as case shifts for reserved symbols.

'Begin' Begin := A + 1 'end'.

Others require that a reserved identifier may not be used as a programmer symbol, - i.e. the statement is syntactically incorrect. The latter is the general viewpoint adopted in this discussion.

The section on translation described the dependencies of interpretation on parsing and, with certain obvious restrictions, the parsing algorithm can be made independent of the rest. Similar comments cover the interpreter, which accepts a bracket-less code of some kind and proceeds to execute it. It finally produces displayable output which may be trimmed by a *clipper* so that it will fit in the *window* of the display and finally output back to the user.

Several things should be noticed:

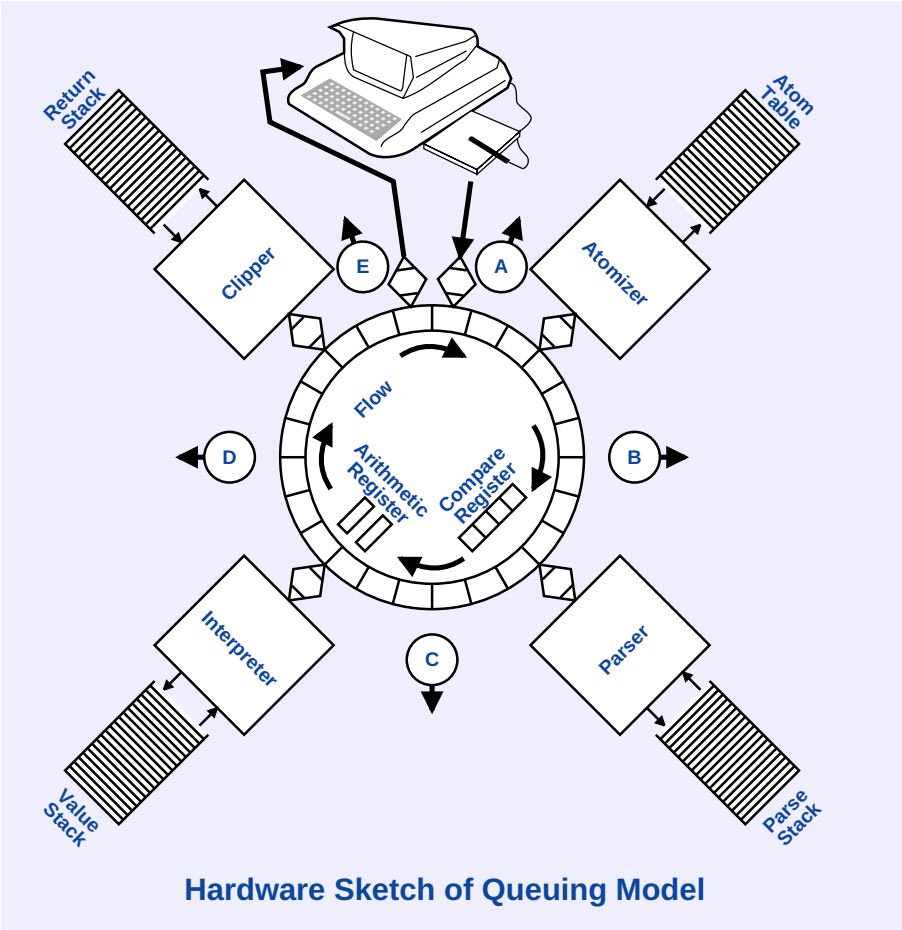
First, the number of operations is quite large to accomplish elaboration on a sequential machine.

Second, that any of the queues between the action boxes may reside in storage for arbitrarily long periods of time. The streams are commonly diverted at (A): called *text files*, (C): *compiled code* and (D): *Display lists*. In LISP, (A) is thrown away and (B) is retained, since

(A) can be recovered from (B) and the atom table which holds *print names* of all the symbols.

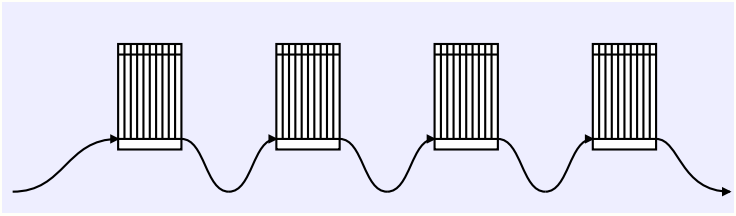
Third, *all* of the algorithms may run *concurrently*, constrained only by the necessity for communicating through queues, which act as buffers for the asynchronous operations. The length of the queues may be *tuned* so that any one routine does not get too far ahead of the others.

A hardware realization of such a scheme could look something like the diagram below.



Hardware Sketch of Queuing Model

A *circular* buffer acts as a lazy susan to distribute each stage of transformed text to the next processing box. The diamond-shaped pointers are the input and output pipes for a processor. The circularity of the registers is easily realized by making the total number of them be a power of 2: say $2^5 = 32$. Then the diamond-shaped pointers can be 5-bit pointer registers which *wrap around* mod 32, i.e., 31 + 1 = 0. Additional hardware can be used to make sure the pointers do not overlap each other. When two pointers are equal, the downstream processor must *passivate* itself until it has some input. A Turing Machine analogy would be:

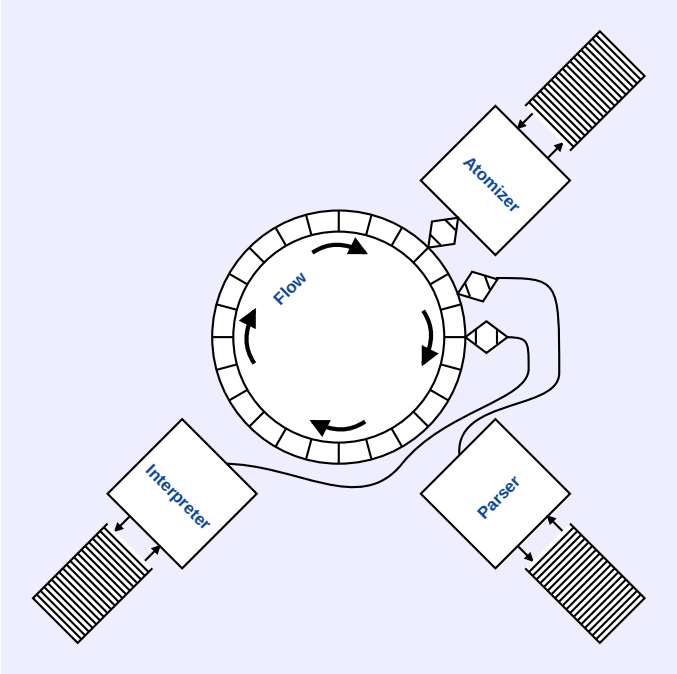


It is quite clear that a machine like this one can process raw text at a speed not appreciably slower than a more conventional sequential processor can execute computed code.

In order to be completely fair about this scheme, traumatic situations should be considered. The basic reasons that a high degree of concurrency can be obtained with this method are:

First, the memory is considered to be faster than the arithmetic unit. This is true of most machines, especially small ones. The delivery of text to the buffer and the manipulation of the stacks can then be gotten for *free*.

Second, no branching was assumed, so that processing was strictly in a one-dimensional order. Frequent branches could seriously impair the maintenance of concurrency, since all pointers bunch up to the interpreter when a branch occurs.

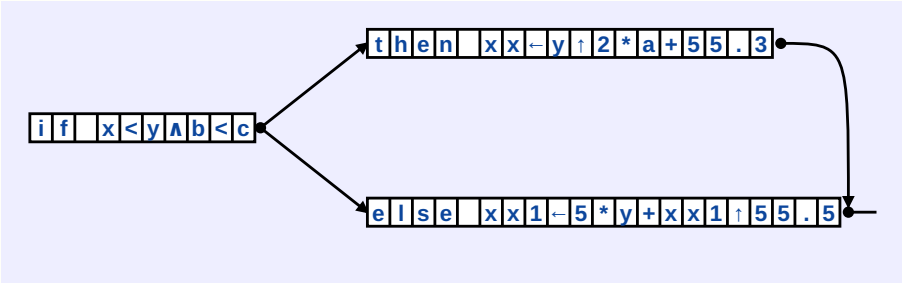


This means that both the parser and the interpreter must *passivate* themselves until the atomizer can turn the new text of the branch into atoms for the parser, etc. Processing now proceeds at the rate of a sequential machine until the atomizer and the parser can get far enough ahead of the interpreter. Since programs are known to branch quite frequently, this is an intolerable situation.

Suppose now that the textual representation is *not* linear, but is structured to mirror the control flow - i.e.,

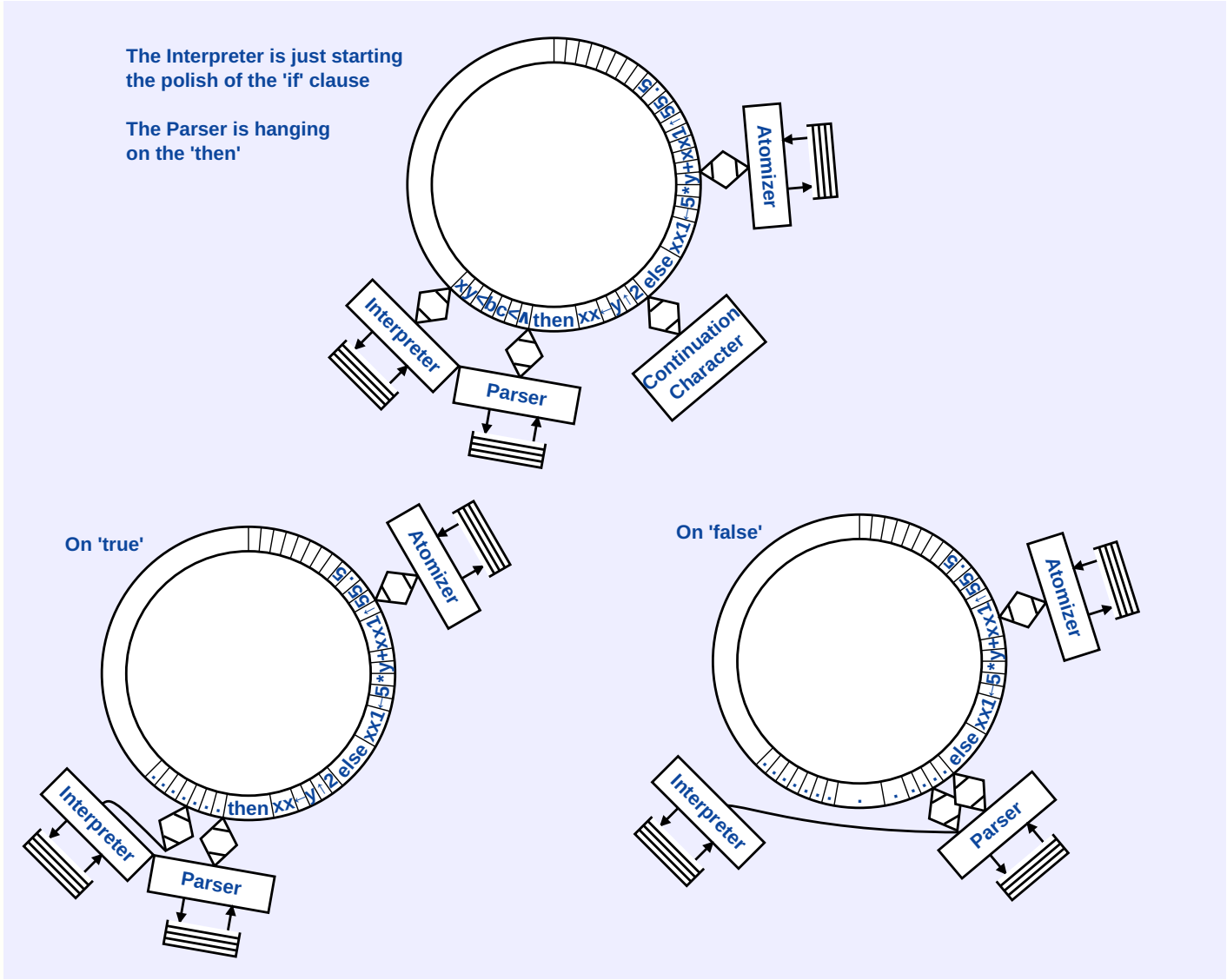
if x<y∧b<c then xx←y↑2*a+55.3 else xx1←5*y+xx1↑55.5;

would be



Now the atomizer can do the following thing: it process some fixed distance down *both* possible streams, as shown in the drawing. The simplest method would be to have the parser hang up on the *then* until the interpreter makes the decision. If the conditional clause is *true*, the atomizer switches to the pointer it left for the continuation character to take up where it left off on the *then* stream. On *false*, the atomizer stays where it is and both the parser and the interpreter switch to the false pointer.

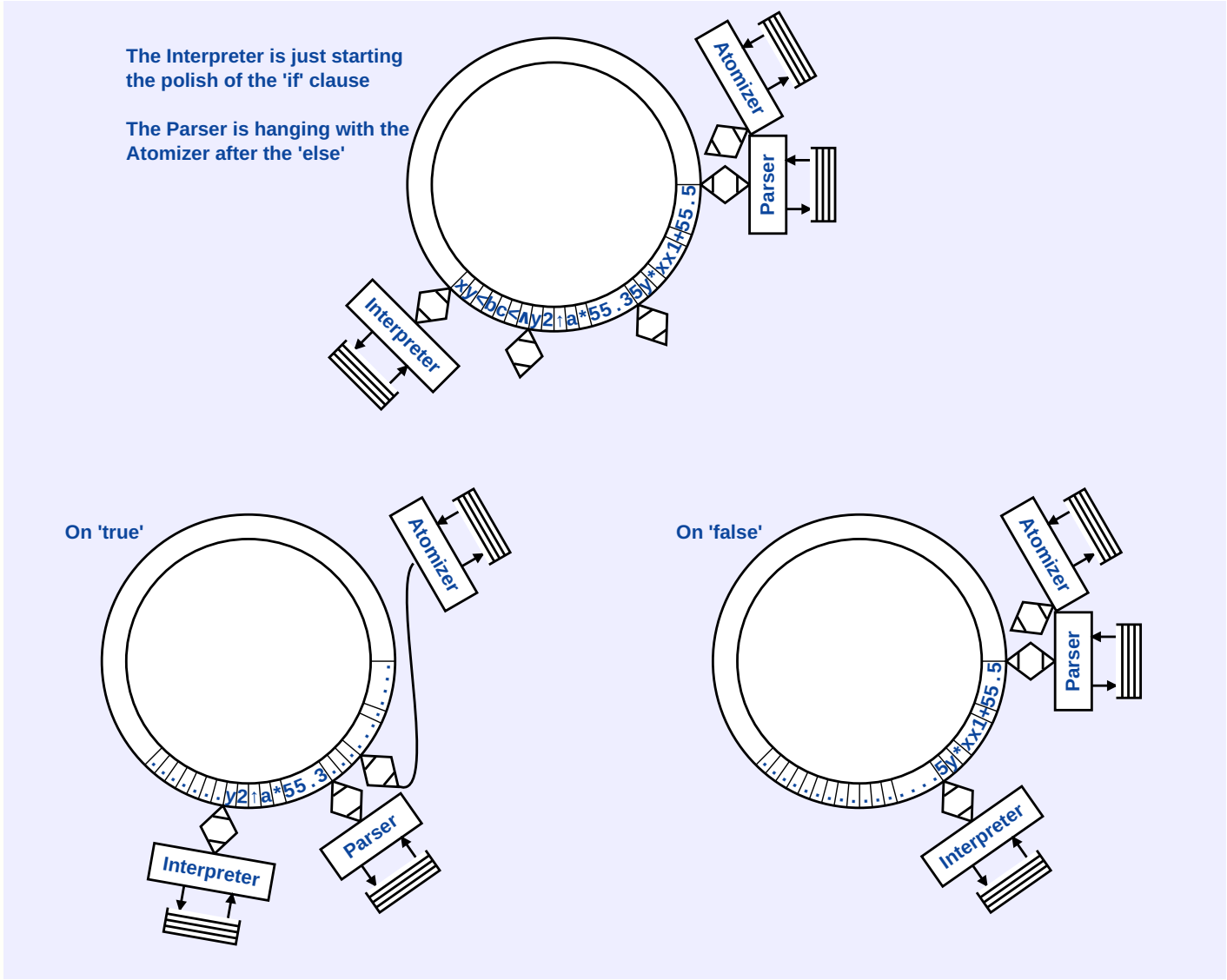
As a compromise solution, this scheme will work rather well. The shock of transfer is buffered by the 8-10 atomized bytes in either stream. By the time they are exhausted, the atomizer will have had time to get more ready. One noticeable drawback is that the interpreter must be idle until parsing procedures more polish code. This problem can easily be remedied by checking to see if both forks can be carried between the *parser* and the *interpreter*.



This situation is better. Because of the greater amount of removed redundancy, more information is packed into the parallel streams. The atomizer must now work to feed the idle parser, but the chances are extremely high that it will be able to stuff its queue before the interpreter meets the parser. When this can be done, the branch takes place at register speeds and no trauma is suffered.

Subroutine calls are even easier, since it is known for sure that the call will be made. While the interpreter works on the actual arguments, the parser may start processing the body of the procedure. In a sense, the interpreter will not even know that a transfer will have been made. Returns are handled in a similar manner.

Any well-nested language like ALGOL, EULER or FLEX (1968) may be processed with great efficiency using concurrent translators. This is one case where a little look ahead goes a long way. Problems of state saving do not arise, since context which must be retained is pushed into the respective stacks. The hardware does not get into the problems of STRETCH (35) , which also could process down both branches. But, not being a stack machine (and for other reasons), STRETCH had a great amount of logic just to undo things if a branch didn't turn out as expected.



3.c. Abstraction and Attributes

What's in a name? - W. Shakespeare, Romeo and Juliet

Chapter 1 introduced the idea that the semantics of an abstraction can be described by forming bindings on *property lists* that are associated with a name. The motivations for the current FLEX usage of these ideas came from a number of sources which are listed at the end of this section.

LEAP (36) and APL (37) can form associatively accessed triples of the form

A•0 ≡ V e.g.
son • Bob ≡ Bill

The triples may be interrogated in all possible ways by effectively leaving a field blank. The sets which satisfy these conditions are delivered.

(a) son • Bob ≡ ? all sons
(b) son • ? ≡ Bill all fathers
(c) ? • Bob ≡ Bill all attributes
(d) son • ? ≡ ? all <fathers • sons>
(e) ? • Bob ≡ ? all <attributes • values>
(f) ? • ? ≡ Bill all <attributes • objects>

Simula (20) and "Wirth Algol" (38) have the concept of *records* which may be used in a very similar fashion. A prototype record is defined (in (38)) as follows:

record person (set: sons, Integer: age, Integer: height, ...).

Then *instances* of this form may be dynamically created and named.

Bob := person ("Bill", 52, 70, ...);

and used:

```
son of Bob := Tony;
type son of Bob;
        Bill
        Tony
```

The disadvantages of having to choose the attributes at compile time and not being able to ask the inverse questions (e.g. (b-f) in the previous example) far outweigh the speed of access and ease of initially setting up the associations.

Simula can create many instances of the same ALGOL-like procedure, each of which may be named.

The naming ritual only evaluates and binds the actual parameters. Evaluations are caused by other means. Wirth records in this scheme are just procedures without execution bodies.

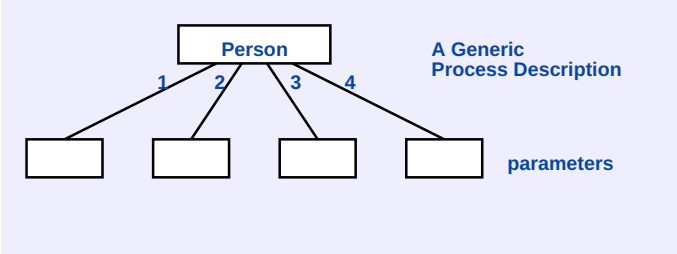
```
activity person (son, age, weight, ...) set son; integer age, weight, ...
```

It would be used just as before

```
Bob  := person   (Bill,   52,   175,   ...);
```

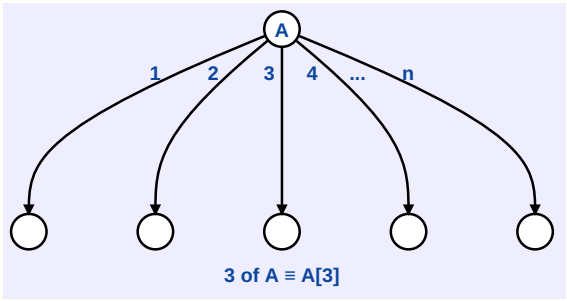
To access the parameters (or attributes, as they are called) is quite a chore, requiring extensive copying of information. An *activity* with an execution body is thought to have *behavioral* attributes as well as *factual* attributes.

Evans and LeClerc in a paper on memory mapping (39) used a scheme similar to records to state binding of address spaces to a process (described in more detail in Part B). What is of interest here is the development of the notion of control over the *access paths* to a bound parameter, which involved an intuitive visualization of *pipes* connecting parent routines and its parameters.

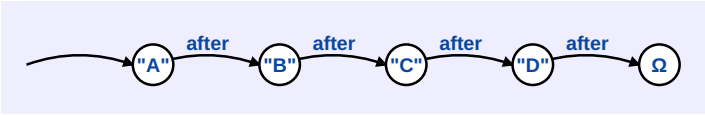


The access paths are indicated by parametric numbers on the pipes leading to the bound values. In this scheme, using numbers was quite useful, since it did not relate to any language and was only concerned with memory addresses. A language environment, however, will find it convenient to identify the *names* of parameters with the numbers. When this is done, it is seen to be reasonably equivalent to the other ideas in an intuitive fashion.

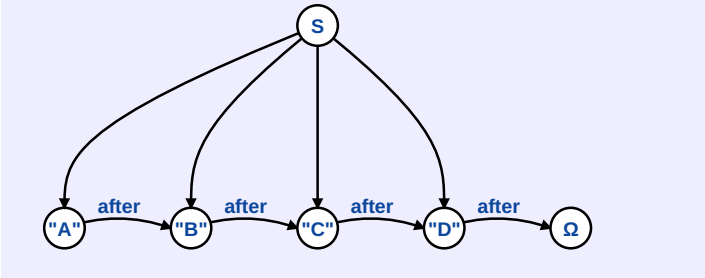
A. W. Holt and others in the work on theory (40, 41) also developed graph structures, whose arcs were named (as well as the edges), to indicate access (and connective) functions. An array might be



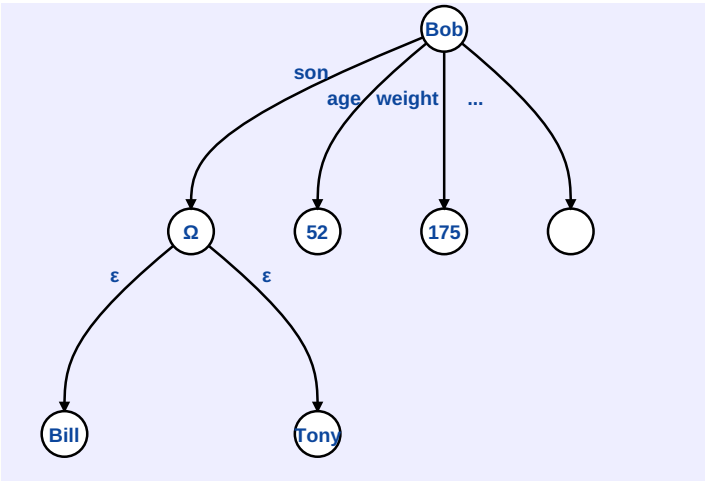
Strings (strictly speaking) only have the relationship *after*. So *ABCD* might be



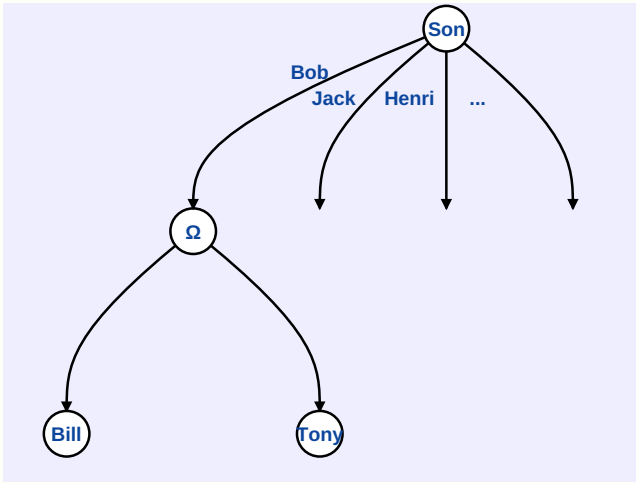
Strings with a more usual access arrangement might be represented:



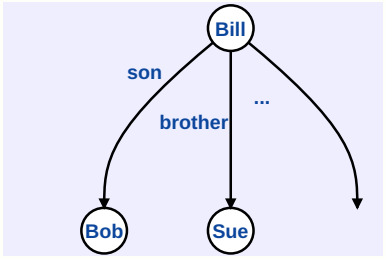
A Wirth record would be:



The Rovner-Feldraan scheme is more complicated in that all relationships are modeled, with care being taken to denote the aspect in which a name is used - i.e., whether it was an Attribute, Object, or Value. Son as an attribute would have:



Bob as an Object would be as before, and Bill as a Value would look like:



The actual implementation of this *inverted file* scheme for associations is very clever, and the reader is referred to the collection of papers on this subject.

The desire to model and manipulate abstract structures in FLEX, as well as the interactive nature of its environment and control over coercive processes, makes an attempt to draw all of the above ideas into one compact routine almost mandatory.

As will be seen in the section on modeling, a static ALGOL-like block structure is too confusing for constraining scope in a parallel event environment. The idea of bindings belonging to (and being controlled by) independently controlled entities is very appealing.

FLEX Abstractions

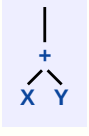
Any name may have an unlimited number of bindings which may be set up in different ways. Bindings may be established by *fiat*, by using the association dot •. They may also occur during entry to a procedure. They may be set up as a result of default cases or forced by coercions. A default case might be

A := 5 where it is
val•A := 5 that is meant

A coercion may be effected by an operator.

A := X + Y where it is
A := val•X + val•Y that is meant

Since complete control over the elaboration of operators is *sine quo non* in FLEX, many desirable side effects may be generated. In Formula ALGOL (42), if *A* were of type *formula*, the structure:



would be delivered to A, not the sum. This can be easily done in FLEX by modifying the assignment production, from:

```
assignment ← 'named_thing "←" value';
  to
assignment ← 'named_thing:A "←" [if type•A =  "forms" then true] value [ ] | formula [ ] ';
formula    ← 'Term:A *("+"Term:B [left•A←plus (left•A,  B)]';
plus       ← ':left, :right';
```

The assignment arrow now coerces the right side to be either a formula or a sum, depending on the type of the left side, which is just an attribute into this structure.

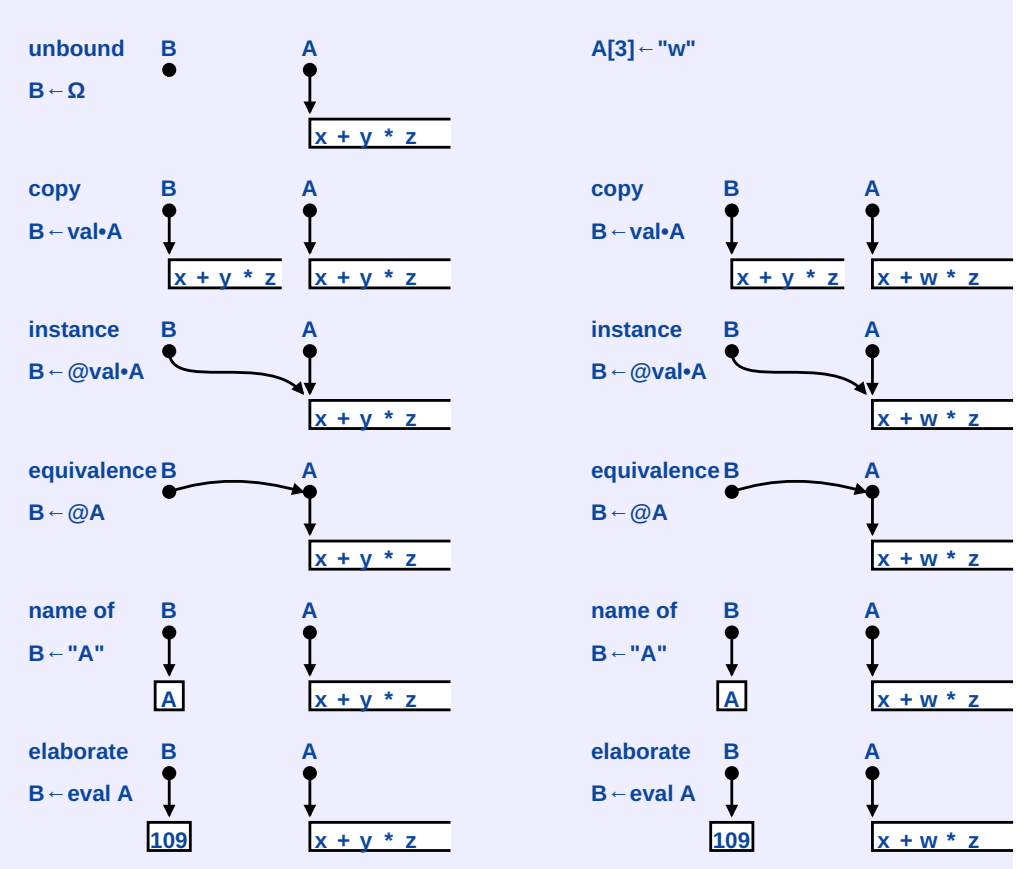
Many similar tricks can be played for creation of arrays, stacks, queues, etc., which will be covered in the section on mapping-functions.

Two default abstractions are *val* (already explained) and *att*, which stands for the set of all other attributes of the name which can act as local variables to a possible procedure bound into val. The procedure need not use its own name to get the value of its attributes - they may just be mentioned.

```
Bob ← ':age, :weight, :height•[if age = 40 then sleep]' ;
Bob (40, 175, 70);
and
type age•Bob;
  40
```

Bindings

Since abstractions are conveniently modeled by directed graphs, it is interesting to contrast various possible bindings that may be formed between two names.



The left-hand side of the illustration starts with *A* bound to a string and *B* initially unbound.

Copying transfers a literal image of *A*'s binding to *B*. *Instancing* lets *B* and *A share* the same string. If the system were asked to *type A, B;* the answer would be the same in both cases. The difference arises if *A* is changed and a request for *type A, B;"* is made.

Equivalencing has *B* containing a reference to *A* rather than its binding, *type A, B* will yield the same result, even if *A* is rebound. The instance will have *B* bound to the old value of *A* rather than the new.

Name of is even less direct than the previous bindings, *B* is not bound to a specific *A* but rather the name *A* itself. Any use of *B* will substitute whatever *A is current* in the binding dictionary.

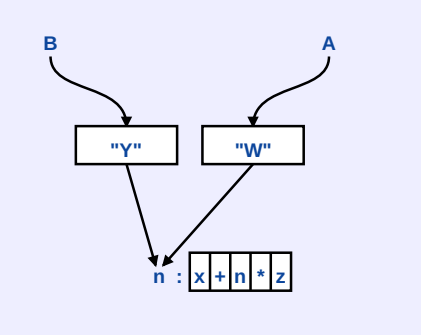
Elaboration is the final example. The value of *A* is first fed to the evaluator before being bound. The usual case of procedures falls under this category where the *eval* is considered to be part of the binding, rather than an *outside* operator.

There may be *degrees* of elaboration also. Here, evaluation proceeds until a literal is produced which is then bound. As seen in Chapter 3.b, however, any intermediate form could also be passed as a *partial evaluation*.

The reader may amuse himself by trying various changes to *A* while noting the effect (if any) on *B*. After the bindings have taken place, try:

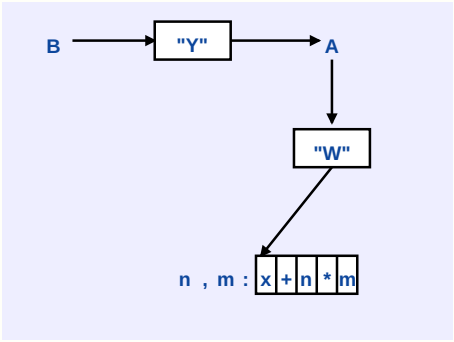
a. A[3] ← "w": ... obvious ...

b. Suppose the effect of the *copy* was wished to be retained in the *instance*.



This can be done by parameterizing the change with a dummy parameter (*z* in this case). Now when *B* is read, *Y* is bound to *n* and substituted *on the way down*. There is semantic difference between this and the copy. Pragmatically, however, there may be a great difference, since there may be thousands of logical copies needed with only a small amount of space to contain them. The latter method will work, whereas the *pure* copy will fail. This notion has profound ramifications in all areas where modeling of *somewhat similar* objects is done.

c. Suppose parameters are *interposed* in the *equivalence* binding.



What might this mean? *A* could be *x + w * m* with the first parameter filled. *B* could be either *x + y * w* or *x + w * y*, depending on the order that actual parameters are gathered. The second binding is probably more useful (*nearer* names get bound first) and will be used.

d. Languages that allow the *name of* binding (such as LISP 1.5, 2) allow the control of both static (as in ALGOL) and dynamic scope.
e. If *A* is a string and is only partially evaluated, then *B* can contain a procedure which will request further evaluation when examined.

These basic bindings and combinations thereof allow the language user to more precisely state his semantic intent. Although certain *default* bindings and elaborations may take place if nothing special is mentioned, it is always possible to completely control the binding process.

3.d. Elaboration

> *And he who is versed in the science of numbers can tell of the regions of weight and measure, but he cannot conduct you thither. - K. Gibran*

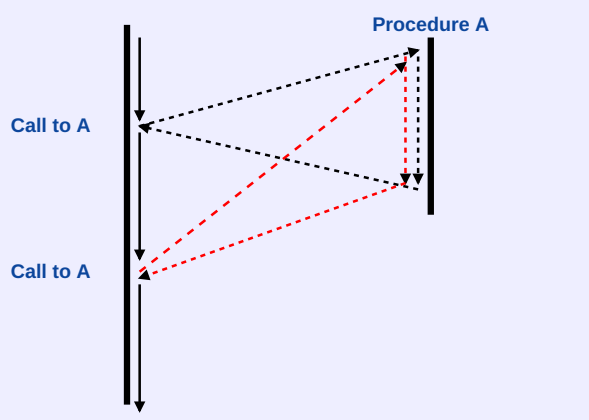
Now that all the preceding constraints have been mentioned, it may be said that elaboration in FLEX is very *straightforward*. It does what you think it will, since all control over these processes has been left to the user.

Only a few points need to be made here, since much of elaboration is in the province of the hardware realization schemes presented in Part B.

Routines

The notions of *procedure*, *coroutine*, *process*, and *parallelism* are intimately intertwined in the FLEX language so that distinctions between them cannot easily be drawn.

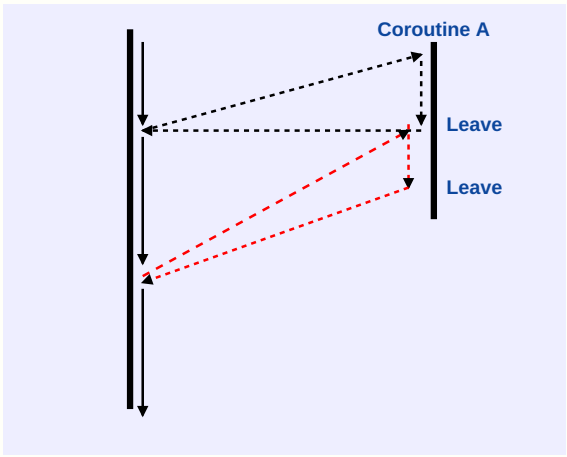
The *procedure* is possibly the primitive idea from which the terms are derived. It is considered to be a body of text which may be interpreted in a *serial* fashion. It may consist of a single function (which has no inherent order of execution except that the arguments are elaborated first) or it may be an explicitly sequential list of statements in the ALGOL sense.



A *strict* procedure will be entered at the start of the ordered statements *each* time it is called; the parameters passed to it are supplied fresh for each call. This means that the only way a procedure can retain state information of previous calls is to change variables global to it as a kind of *side effect*. This is somewhat undesirable, since these variables are also visible (and may be tampered with) by other routines.

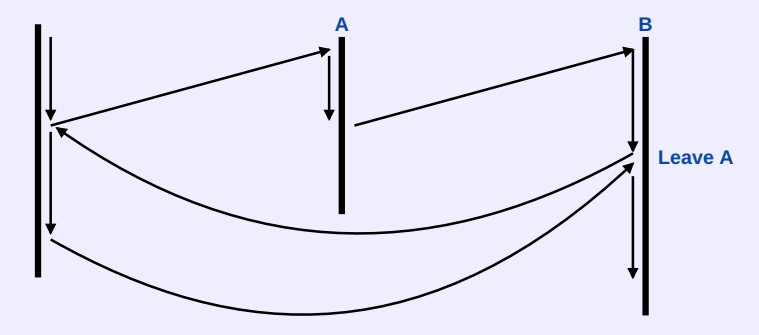
Coroutines

Coroutines are created by deferring the return (and consequent loss of state) of a procedure. This is done in FLEX by the *leave* directive, which will cause an exit back to the calling routine, but will preserve the state of the procedure.



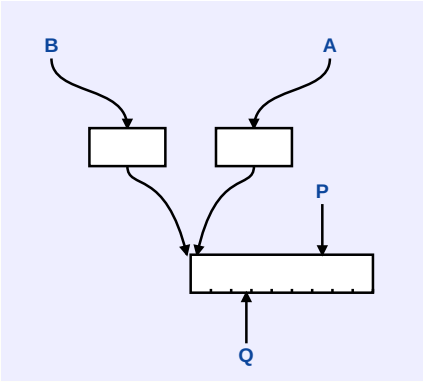
When the routine is again called, elaboration will resume from the previous *leave*, with all variables intact. Semantically, it is as though the *leave* never occurred. The *leave* is considered to be a *soft* interrupt.

The *leave* may also have arguments indicating how deep the *action* should be frozen. In routine *B* a *leave A* will save the combined states of *B* and *A* and will reenter *B* when action is resumed by another call to A.



Processes

It is just a short step from the notion of a coroutine (its elaboration controlled from within) to the more general idea of a process which is considered to be an *instance* of a coroutine and whose elaboration may be controlled from without as well as from within. The instance examples from the previous section provide illustration. *B* and *A* both share a *behavior pattern* which is represented by the code.



They may or may not have local parameters which distinguish them from one another. In any case, either may be elaborated as though the other does not exist, since all that needs to be known separately is the position of evaluation in the code. (It might be *P* for *A* and *Q* for *B*.) It is clear that the evaluator is free to switch from one to the other as it pleases without disturbing the semantic intent of the two routines. Thus, elaboration of *A* and *B* may be *time shared*.

Parallelism

When given a *vector* to elaborate, the FLEX interpreter will attempt to timeshare the evaluation. As will be seen in Chapter 10 more than one machine may be hooked together and jobs can be thus distributed.

1. A process being elaborated is *active*.
2. A process under elaboration but not currently being worked on is *dismissed*.
3. A process after a *leave* is *passive*.
4. A process receiving a Ω is *terminated*.
5. A process forced to wait for some other action to take place is *suspended*.

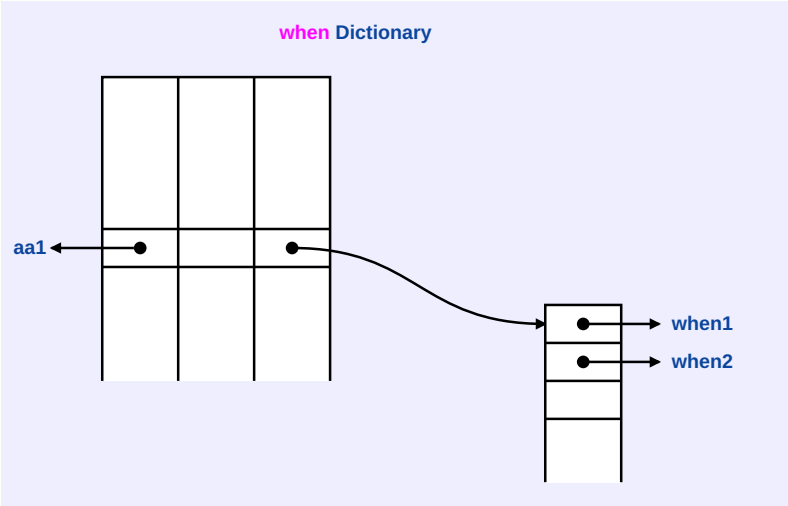
There are various means for controlling elaboration of processes. One of them is:

The *When*

This conditional verb provides a rather complete interrupt capability, on both hardware and software conditions. The logical action appears as though the *when* continuously and globally checks the conditional clause to find out if it has become true. When it does, the consequent clause is executed and the *when* is reset.

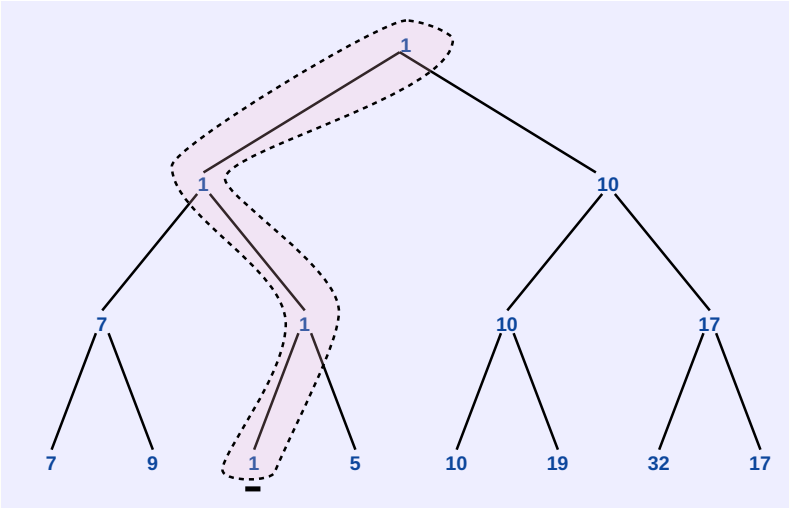
In order to make this happen, some tricks have to be played, and the user should be aware of what is entailed. The hardware form of a name depends on a number of conditions, which will not be elucidated here. It will suffice to say that the *value* attribute of a name is a fixed number of bits wide so that it is suitable for residing in a much-massaged stack. There are two extra bits residing with the entry. One says whether the rest of the bits are indeed the *value* or whether the interpreter must look further. The other says whether or not this variable is found in the conditional clause of a *when*.

If it is, the particular *when* (of *whens*) may be found by hashing the variable name into the *when* dictionary. And indeed, this is what is done every time a *write* passes through the name or a function call elicits a new value.

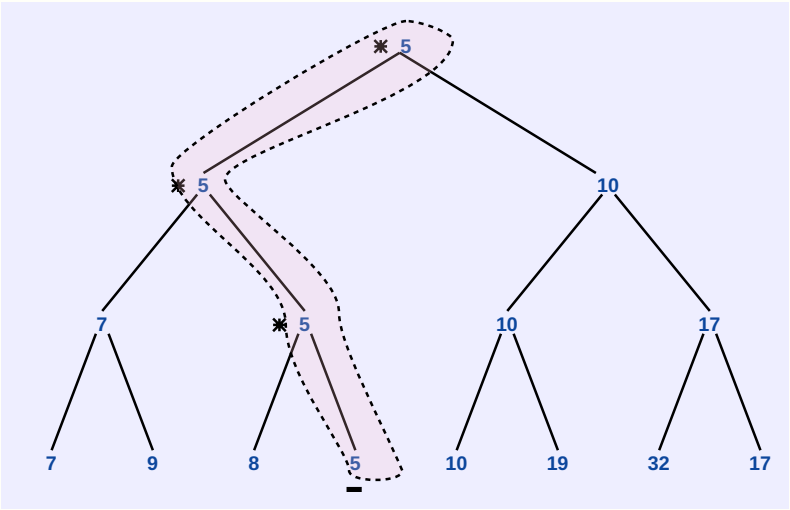


Actually, a *when* dictionary may be carried for each level of process scope, depending on how crowded the global dictionary is. This method is a *medium* hardware solution to a very sticky problem. The overhead is much less than if the *whens woke up* periodically to check their clauses. On the other hand, the effective time to store a *value* in a name is now many times slower.

Another way to handle this vexing problem is to look at the theory of *tournament* sorting, in which a tree of partial compares is built up by an initial n-1 compares on the entire table.



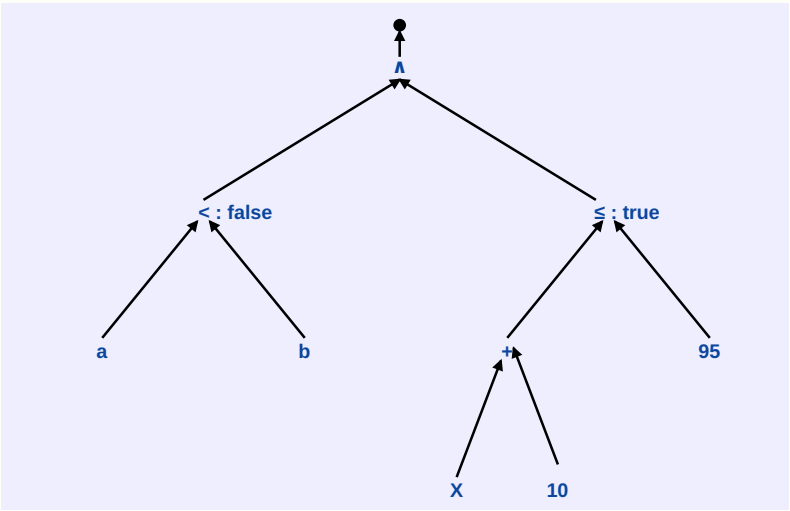
So it takes 7 compares to find the smallest of eight numbers. But now, notice that if the path is remembered and a new number is entered (say, 8) only $log_2 8 = 3$ compares need to be made.



This principle can also be applied to the *when*.

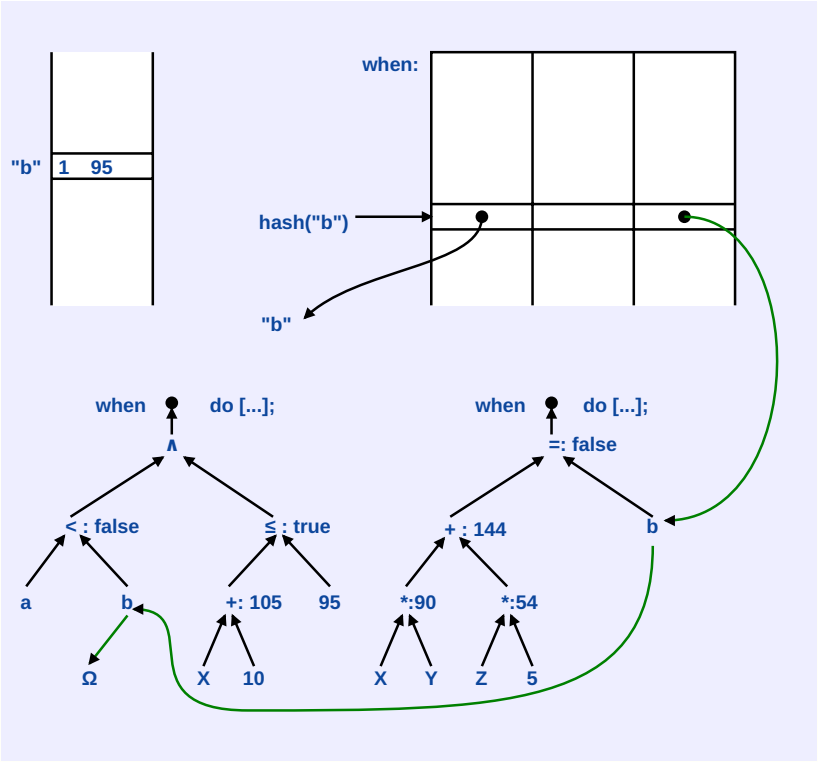
The conditional clause can be carried as a binary tree, rather than linear code. The evaluation of the lower branches can be carried at the nodes. For:

when a<b ∧ ((X + 10) ≤ 95) do



Suppose *b* changes so that it is larger than *a*. Now the < is true and so is the ≤, since it merely needs to check its other node for its value which has been *true*.

The dictionary set-up that now suggests itself would be:



The algorithm is completely recursive, *blind*, and probably an order of magnitude faster (on the average) than the previous method.

3.e. Mapping

The map is not the territory, the label is not the thing itself. - A. Korzybski

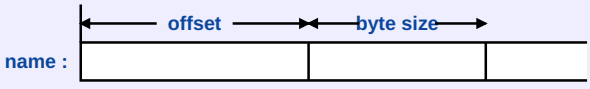
Although the term *mapping* in mathematics has a rather broad meaning, in Computer Science it has come to represent a very specific class of transformations which have to do with applying *structure* to *raw* representations for information.

Usually, the computer manufacturer already has very specific ideas about the mapping of information, and programmers learn to coexist with *word* and *byte*-oriented memories. The fact that these predetermined *chunks* are almost always too large or too small for easy use has not yet depressed the market in computer memories. This subject is treated from a hardware standpoint in Part B. For now, it will suffice to abandon the word and adopt the *segment*, which may be pictured as some arbitrarily extendable stream of bits.

Only a few primitives are needed:

```
create name;
destroy name;
describe [name, offset, byte size]
```

The first two are obvious (provided the system is willing and able); the third function allows an arbitrary *byte* of *byte size* to be described which is situated *offset* number of bits from the beginning of the segment (counting from zero).



If the procedure and coroutine structure works as advertised, then all other mappings in the FLEX language may be easily *devised*.

Simulation of Computer Memory

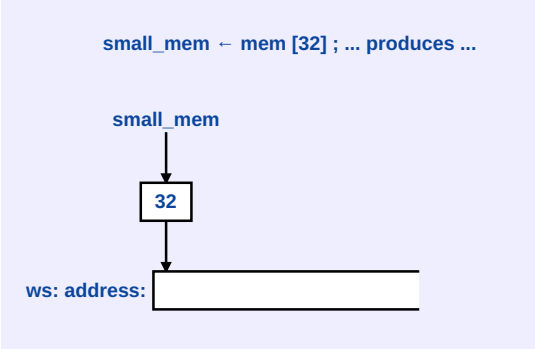
```
mem ←'value: address, :name.
        [create name; ... get a segment ...
         while true  ... loop forever ...
         do [describe [name, address * 32, 32]
         leave  ...deposit description ...
                  ]      ... and reenter here ...
        ]';
```

Suppose now that a *generic* memory routine is needed which can be partially set with the word size.

```
mem ←'value:  word size.
       ['value: ws, value: address, :name.
         [create name; ... get a segment ...
          while true   ... loop forever
          do [describe [name, address * ws, ws]
         leave  ... etc ...
```

```
            ]
          ]  ' [word size] ... application of first param ...
        ]
    '
```

The main body of this routine is a prototype quoted function that has the parameter*word size* applied to it immediately. The result is a new function with the substitution of the *values* of *word size* for *"w s "*. Looking at it from a coroutine standpoint, execution has halted at the comma after *value: ws,*. The routine is effectively waiting for further parameters. Another way to visualize this situation is to refer back to the section on abstractions.



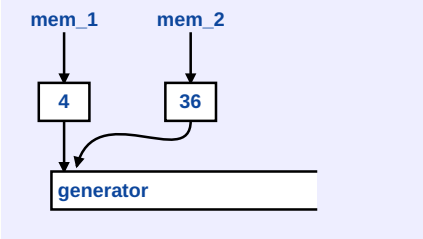
If *small_mem* is now called with an address:

small_mem [9200] ← 55

the correct thing happens. The 9200 becomes the second parameter, the routine is called, the segment is created, the loop is entered and *small mem* reacts as it would if it had been explicitly coded rather than generated.

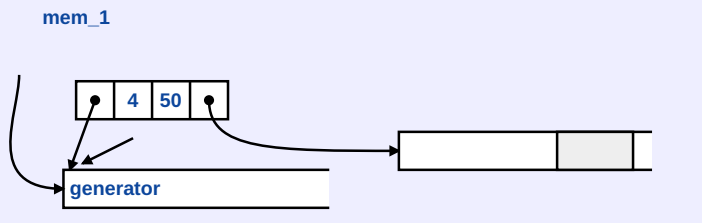
The generative routine *mem* is rather wasteful, since a *copy* of the mapping routine is created each time. A more aesthetic version might be

```
mem  'value: word size, "generator,
     [generator ←
      ' ... body of routine
        ';
      while true
      do[@ generator [word size]
         leave ]
     ]' ... now ...
mem_1 ← mem[4J; mem_2 ← mem [36]; ... will give ...
```



... which is what is desired ...

After mem_1 has been used, *mem_l [50]*, it will look like:



The *zeroth* parameter is the routine, *one* is *"ws" = 4*, *two* is *"address" = 50*, *three* is the local *"name" = segment*, *address* is the only parameter that may be rebound each time, the others are protected.

Arrays

The simulated memory is a one-dimensional array. Multi-dimensional structures may be created in several ways.

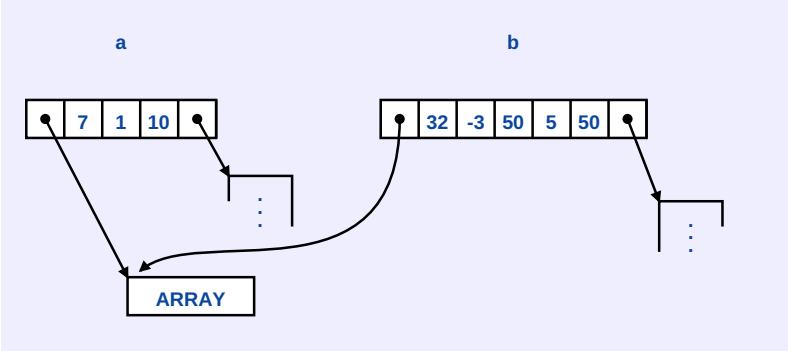
1. The entire array may be mapped into one segment, in which case the well-known recursive algorithm may be applied.

2. Each dimension may be mapped into a segment. Either case is easy to do. For convenience, the ALGOL-60 convention of specifying upper and lower bounds will also be used. The basic idea is to collect the unspecified number of bounds as a *list* whose length may be determined. The rest is straightforward.

```
[@'value: byte size list: bounds list; indexes, :name.
  [create;  .. same thing ..
   while true do
    [while: ← length •bounds by -2 to 0 do
     [
     ]
```

```
    leave;
   ]
  ]
]' ... done ...
```

The structure will look like:



Systems programmers will be interested to note the exact similarity between the data structures of this scheme and more usual ALGOL array descriptors. It is provocative that the totally general approach takes no more storage than necessary and provides byte handling besides.

Stacks and Queues

The mapping of these structures is exactly similar to the array (with a twist). The twist has to do with the different actions of stacks and queues, depending on whether they are being stored into or accessed. This requires the structure to somehow be sensitive to the operators that are using them. Elaboration in prefix form is sufficient knowledge, since the store arrow (←), if present, will have been visited before entrance to the mapping routine. It is, therefore, only necessary to provide a test if in the range of a ←. For this example, it will be called *next-op*.

```
stack ← 'value: width
          [@'value: wide, :(name, point).
            [create name; point 0;
              while true do
               [if next op = "←"
      ... pop ... then [point ← point-1;
                              describe [name, wide * point, wide ]]
      ...push ... else [point ← point+1;
                              describe [name, wide * point, wide ]]
               leave]
             ]' [width]
             ]';
```

Queues are done in the same way, except that two pointers must be maintained.

"Dataless" Mapping

The comprehensive abilities of the mapping concept allow generalizations and extensions to be made to the work of Balzer (49), where it was suggested that for debugging (and other purposes) it is useful to replace so-called data structures by functions, thus allowing more control over the access paths. This is essentially what has been done here. Output and sentinel routines may be inserted in the mapping routines themselves to alert the programmer to dangerous conditions.

The abstractions thus provided allow data-independent programs to be written.

3.f. User Interaction

> *UPRIGHT INSTRUCTION wherein the lovers of the clavier and especially those desirous of learning, are shown a clear way, not alone (1) to learn to play clearly in two voices, but also, after further progress, (2) to deal correctly and well with three obbligato parts; furthermore, at the same time not alone to have good inventiones, but to develop the same well, and above all to arrive at a singing style in playing and at the same time to acquire a strong fortaste of composition.*
> *Provided by Joh. Seb. Bach, Cappellmeister to his Serene Highness the Prince of Anhalt-Cothen, -- Title Page of the Inventions, 1723*

Introduction

Most interactive systems use a special command language for handling files, initiating jobs and communicating with the compilers. In the FLEX system this language is FLEX - no other languages need be learned. There are also no special entities called *files* in the system, as will be seen.

Admitting the User to the Machine

When it is desired to allow a new user access to the machine, a process is created and named with his password. This process will not terminate during the period that he is allowed to use the machine. Most of the time it will lie passive on the secondary storage waiting to be reactivated which is simply done by the user typing in his password on the console.

The user's process is activated, and he is now able to communicate with the machine through FLEX and its built-in editor which controls a free-running interpreter that is translating everything that is entered at the keyboard and attempting to execute it.

By these means the user may entertain himself by performing calculations, editing text, generating new compilers, and generally going where his thoughts lead him. When he desires to cease running, he simply types in a *leave*. This is the coroutine exit command and, since the routine which called him is the process scheduler itself, his process is *passivated* and the reentry point retained.

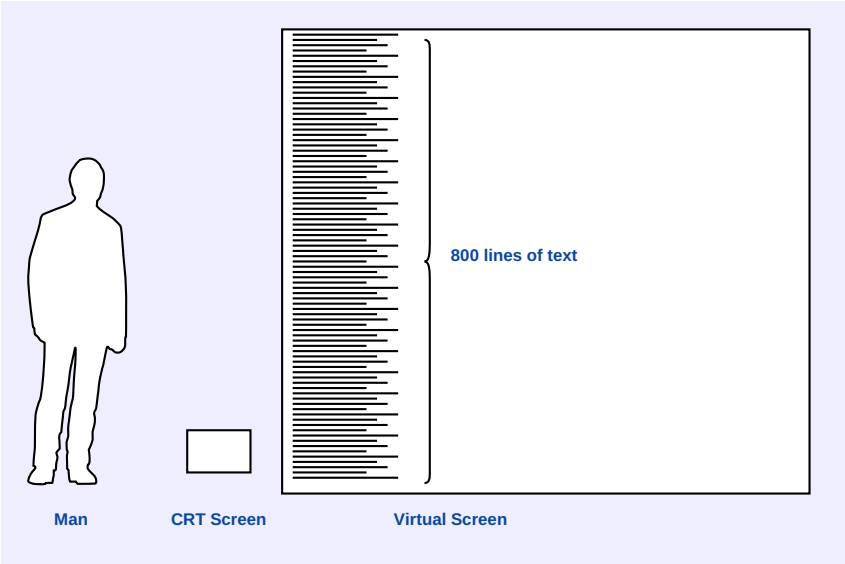
On the next day (or next week) when he again types in his password, his process is reactivated and control is passed to the reentry point; he is where he was the last time on the machine. This is why files (and file handling systems) are unnecessary on the FLEX machine. Any declarations he may have made (and possibly stored data in), have been saved to be used again.

Scope of the User

The user at the console is considered to be inside a process description which in turn is interior to the FLEX system and environment. This concept of system globality fits well the FLEX philosophy and provides a convenient means of allowing the user access to entities such as the FLEX language tables themselves, reserved identifiers whose meaning he may wish to redefine, etc. The program is much more readable, since drawings may be used where drawings are needed. Of course, ...□... will draw a comment box, etc. More exact display commands and the control of windowing are mentioned in the Handbook.

Virtual Screens

One other interesting ability of the display systems is the use of virtual screens. They may be thought of as rather large *bulletin boards* on which things may be tacked up.



The screens are 16384 × 16384 points, compared to the output CRT's 1024 × 1024. However, the windowing hardware allows any rectangular area of the virtual screen to be mapped to any rectangular area on the actual viewplate, so a *v. screen* is simply a high resolution place in which to draw. It should be no surprise to find out that text strings are already implicitly mapped to *v. screens*. About 800 lines can be tacked onto *one side*.

The terminology follows (48) where a more exotic but similar scheme was accomplished.

Windowing

The portion of the display inside the windowed area is only what is transmitted to the CRT. A *clipper* is used to find the intersection of lines with the window boundaries. Any number of v. screens, windows and viewports may be superimposed.

The IO Facility

The FLEX Handbook shows the standard set of interactive devices supplied with a FLEX console: keyboards, tablet and stylus, and vector-drawing scope.

The default base has the stylus and both keyboards connected to the same input stream so that drawings can be directly entered in place of text. This is handy when no great precision of coordinates is needed. Suppose he just wishes to draw a box with a number inside of it. Then he types:

box ← ' i ' ; i ← 5;

This is accepted as a perfectly straightforward piece of input text and later:

Show ← box; ... will give ...

5

The reason this works without much fuss is that all input motions of the stylus are one-dimensional, all outputs by the scope are one-dimensional also (one line at a time). The sapping into text is obvious. It also allows

Pointing

The coordinates of the stylus may be identified with a window for the purpose of pointing to constructions. *Hits* are automatically detected by the clipping hardware when *visible* lines are found. This will cause *user whens* to function, most likely.

3.g. Modeling

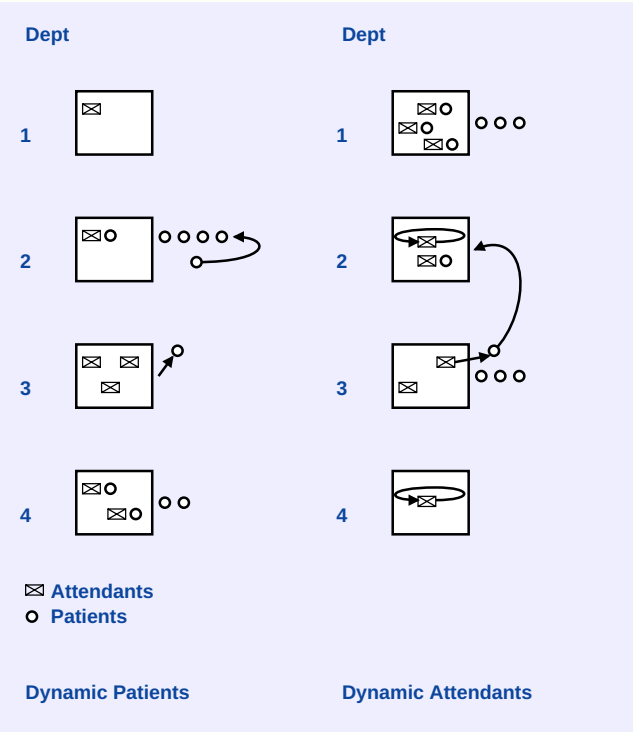
Of time you would make a stream upon whose bank you would sit and watch its flowing. - K. Gibran

Modeling is best demonstrated by rather complete examples. The next subchapter will give a detailed proposal and realization of a somewhat involved interactive language system. This section will content itself with a gradually expanding discussion of dynamic simulation and how it may be effected.

Hospital Simulation

To start with, a rather coarsely-grained model will be developed. The hospital will have some *n* number of departments, each of which has *m* number of attendants. The initial assumption will be that one attendant can serve one patient. If one of the departments is a patient ward, the *attendant* may well be just a bed. In any case, if no attendants are available, the patient seeking aid mast wait in a line by the department until an attendant becomes available. The patient's route will be fixed and is handed to him when he enters the door of the hospital. (This is a rather tight restriction, which will later be lifted.)

A conceptual model of these constraints can be devised with different points of view:



In the second case, the attendant would be represented by a program which would check the door periodically for patients; when one appeared, it would process for some period of time, then look at the patient's schedule and escort the patient to his new destination.

Or both patient and attendant could be active. This is a more natural state of affairs and would allow more *personality* to be ascribed to both of the activators; i.e., smoke breaks for the attendants, etc.

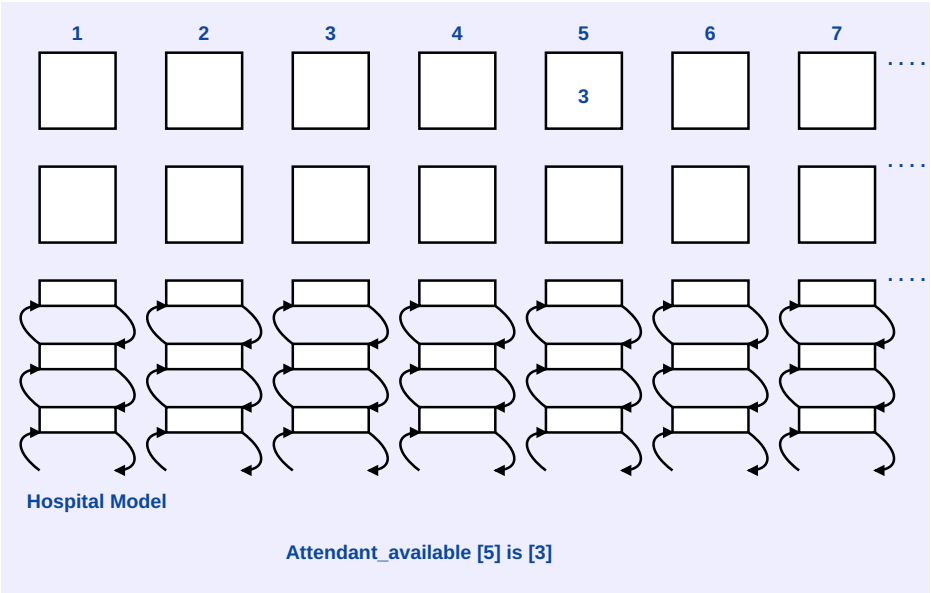
However, for the purposes of starting off this discussion on simulation, the third case, that of the patient being the only dynamic entity in the system, will be assumed. Now what does a typical patient do?

A. He picks up his schedule.
B. He visits a department listed on schedule.
    a. If an attendant is available, then he is commandeered for the time needed for service.
    b. If all attendants are busy, then the patient will have to wait in a line for some unspecified amount of time until *a* can be done.
C. He keeps on doing these things until all of his departments have been visited.
D. He then leaves the hospital (feet first, if his last department was the morgue).

The *F*irst *I*n *F*irst *O*ut Queues developed in the previous section on mapping provide a perfect model for the line (possibly empty) in front of each department.

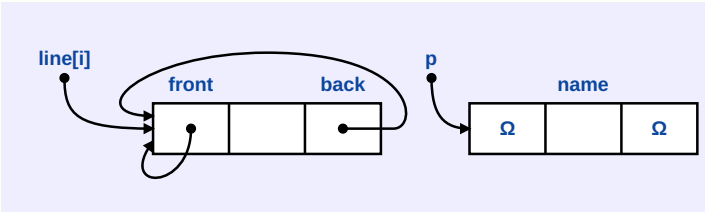
Since the patient algorithm is so simple, and behavior is governed only by the schedule, all of the patients may be represented by *instances* of one routine with separate parameters to distinguish one patient from another. Random distributions can be used to control entry of patients to the hospital, and the schedule will be considered to have been supplied globally by the simulator. Only one thing is left, and that is some way to control scheduling of these myriad of events. This can be effected by setting up a master sequencing queue (a la Simula) to constrain the order of events.

First, the waiting queues, called lines: There will be a need for a vector of them, one for each department.



Line Queues

Let a typical queue element be [:front, :name, :back]. The empty queue will have back and front pointing to themselves.



So to *join* a new element p, do the following:

```
front  •(back • line[i]) ← P; ... these two fix up the front pointers of the last element
front  • P ←  line[i];  ;
back • P ← back•line[i]   ; ... this fixes up the back pointers
```

...to remove an element from the front of the queue do...

```
temp ← front •line[i]  ;    ; ... get the name ...
back • front • temp ← line[i]; ; ... detach back pointer
front • line[i] ← front • temp; ; ... detach front pointer
```

Sequencing Queue

As long as queues are being discussed, it might be well to examine the sequencing structure that will be used. Only one additional field of information is needed, and that is time_of_next_event for this coroutine. That yields a conceptual structure:

The previous functions will take care of structure, but now entries will need to be made into the middle of the structure, depending on the time of next wake-up. There are various ways to do this. The easiest way of just sorting by time will be done.

So the *enter* function will be for Q:

```
p ← Line[i] ;
while time • P ≤ time • Q  .. find a time ≥ Q
do P ← front • P ; ;
join Q into P ;   ... insert ...
```

Now the scheduler need only activate and remove the event notice at the front of the SQ to allow all events to proceed when requested.

Event Scheduling

The coroutine exit, *leave*, will not be enough to schedule with facility. So a few other functions (reminiscent of SIMULA) will be composed.

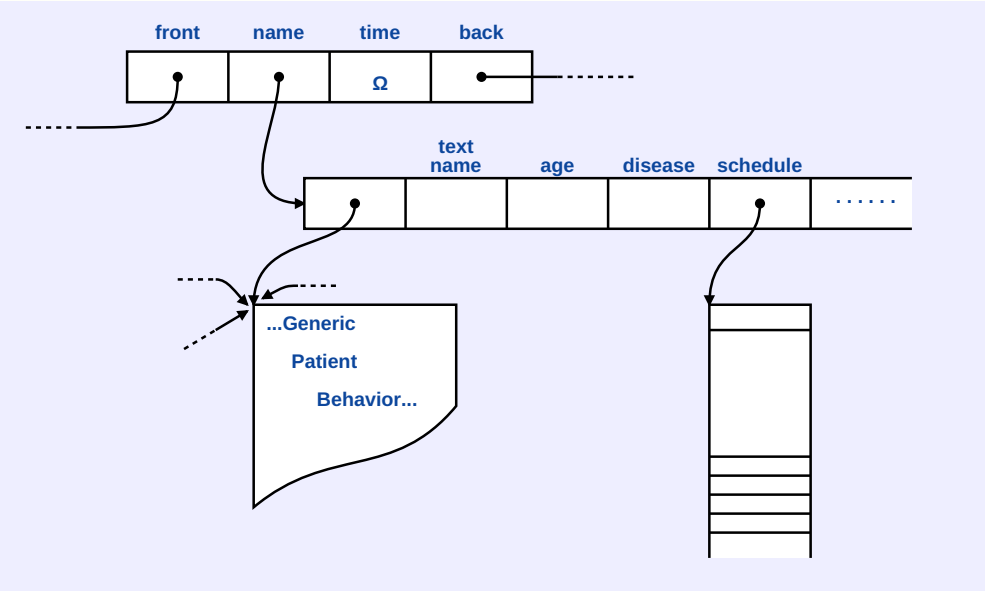
```
hold ← 'name: x"for" value: time   ... will set wake-up and passivate ...
```

```
[Enter X >for time into SQ;
 leave X ]';
```

Notice that the *pointers could be conceptual* only. The entire queue modeling is done in terms of associations, if no explicit mappings are specified. Since this is not the issue here, no further mention will be made of the exact data structure.

The Patient

A typical queue entry will be:



...Define *Patient* to be a routine that will link up an instance of general behavior to some specific attributes...

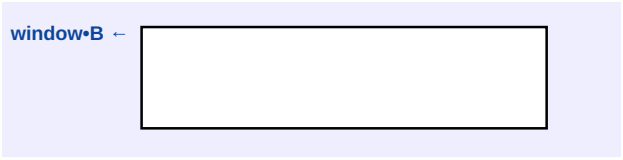
Gathering Information

At this point, the simulation will be rolling along beautifully until there are hundreds of patient*instances* roaming around the halls. Except that so far there is no way to measure the system. An hourly report might be something like:

```
hourly report ← 'Show ← length•?•Line;   ...this will dump out
   hold current for 60 ];  ... all lengths
...More interesting would be to use when on danger conditions...
when length...Line < 10 do Show  ← "!!!!!!!" or
when number•patients > 500 do Show  ← "too many"
```

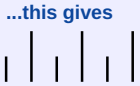
So far, the. display hasn't been used except as a text output device. Here is a chance to use it. A dynamic display of the queue lengths as moving bar graphs might be very interesting.

To do it requires setting up a common data structure between the display and the program. This scheme will have the display working on one frame of data while the next frame is being prepared.



Being able to draw in examples of graphics is tremendously useful as it precludes having to use functions like plot line, etc., except where one has to be precise. Now get line increment by:

```
inc ← "|"; ...about 1/8" ...
space ←  _____ ;
while i ≤ 1 to number•dept
do bottom • bottom # (i) size • space ;
...this gives
   (1) (2) (3) (4) (5)
get_top 'while; 1 to number • dept
do top ← top # length • line • i * size•inc size space ;
...this gives
```



```
...now...
when clock > 60
do [ get-top:
A ← top # bottom
B ← A
Show ← B ] ;
```

will wake up about 16 times a second, look around and generate a new display list.

But what if the simulation is proceeding too fast? Then a*when* is placed on the SQ operator to wait for some real time interval before allowing the simulator to continue. A *pot* on the screen could control subjective speed timing the tablet for adjustment:

```
when y1•sens ≤ taby_yh•sens
       x1•sens ≤ tabx ≤ x r•sens
do SQ hold ← (x1•sens - tab x) / (xr • sens - xl•sens)
 ... simple linear "pot"...When the stylus is hit see if
 tabx, taby fall into the sensitive area sens
 if so, SQ hold will be adjusted accordingly.
```

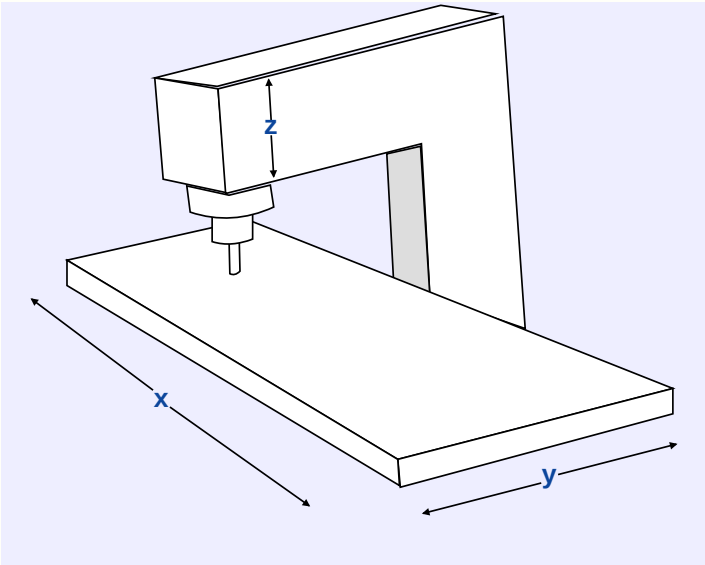
3.h. An NC Language Proposal and Realization

> *...the Analytical Engine weaves Algebraical patterns just as the Jacquard-loom weaves flowers and leaves... - Ada Augusta, Countess of Lovelace*

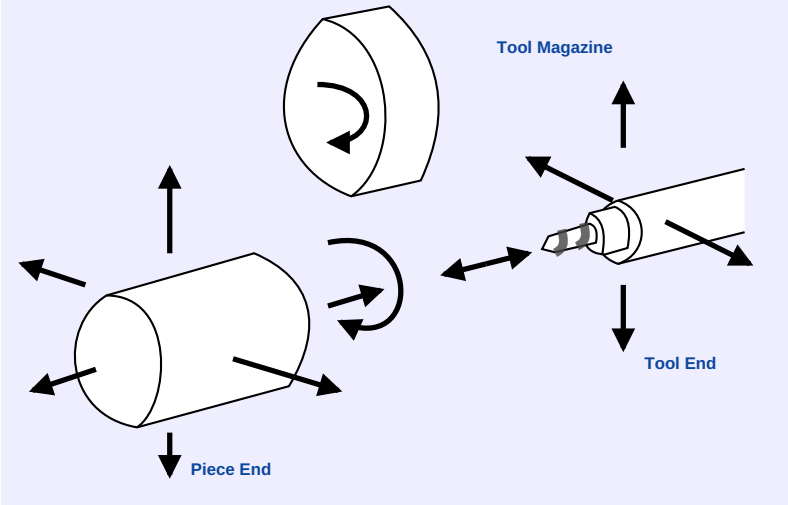
To cap off the previous discussions on modeling events, whether physical or temporal, a typical proposal for a new language system is presented, followed by an outline of its realization in the FLEX environment.

The area of discourse involves numerically-controlled tools (NC) of all shapes and sizes, which are guided by paper tapes prepared either by hand or generated by one of a number of special purpose language processors (including APT (43).

Among the simplest NC tools are so-called 3-axis mills. Operations typically involve positioning a table in x and y as a cutter is applied in the z direction.



Such a device may have a controlled turret with a number of different tools that can be program selected, or an operator may have to intervene to change the tool set-up. Some operations cannot be performed on such a rudimentary device, such as scooping out overhangs (the tool itself is in the way) or contouring the underside (the table is interposed), etc. In addition, there are questions of *latheing* as well as milling both inside and outside contours. A recent machine to deal with these problems is the Kearney & Trecher *Turn-12*. (51)



Degrees of Freedom of the Turn-12

This is a vastly more complicated environment in which to work, necessitating an entirely new approach to strategy and tactics than that used by APT, which was essentially abstracted from the early 3 and 5-axis mill available in the 1950's.

Postprocessors

APT is somewhat device-independent in the sense that surfaces are described that are then *revealed* by a generic tool. The driving tapes are prepared by a *postprocessor* that performs a transformation from the APT geometries to the particular (sometimes peculiar!) needs of the object tool.

Postprocessors are tremendously expensive to produce - they are usually done by the tool manufacturer - and there are about 70 in existence at the present time. Clearly, this is a prime example of a need to express, transmit and reconcile the semantics of two quite different objects: the abstract needs of the geometries involved, and the very pragmatic existence of the object tool.

An analogous problem remains to this day in computer science. The early sixties saw a tremendous amount of activity directed towards formalizing syntax and semantics which, as has been seen, can be rather easily done if a direct enough correspondence between form and content can be made. So, in a typical language-directed translator, one sees a rather clean description of the designer's intent expressed in higher level form. But all mention of the object machine is either omitted or in bald machine-language terms. The first situation assumes a *postprocessor* for the semantics to produce binary code, the second forces the designer to be the postprocessor. Neither of these systems are very interesting when a new machine comes along.

One of the chief difficulties that has so far prevented reasonable solutions to this most vexing of problems has been in describing and correlating the great number of states (and possible state changes) that are possible in digital devices. Jumps, procedure calls, etc. may all produce unpredictable side effects that are difficult to talk about and are even more difficult to anticipate.

The NC tool, however, provides a more docile object on which to practice. Most of them have no memory, nor do they have any possibility of internal iteration, since this has to be done by repeating commands sequentially on the input tape. The only states which have to be known are the x, y, z positions, tools and rotational speeds - all of which are easily controlled.

The problem of cutting out a surface, and describing the machine which is intended to do it, should not be much harder to solve than the *monkey and bananas* problem. At the very least, it is a promising area for approaching the problem.

The Proposal

I. Introduction

In the early .1950's, numerically controlled machine tools were programmed manually. A Flexowriter was used to punch paper tape with codes in a *word-address* format. Working at this low level limited the complexity that could reasonably be attempted and the speed with which new tapes could be produced.

Almost a decade ago, the APT language was developed to allow parts programming to be done at a higher level. Advances in computer science in the last five years now make possible another long step forward. The system proposed here can supersede APT in all ways.

II. The Problem

A good NC language should at least include the following characteristics:

1. All transactions will be carried out as an immediate user-system dialogue.
2. Because of the multi-dimensional nature of the objects to be manipulated, the basic means of communication will be a graphics display device with provisions for rendering pictures in both two and three dimensions.
3. a. It should allow tooling to take place on a large number of machines simply by specifying the machine name; and
   b. An unknown NC machine can be described to the system via a simple interactive dialogue.
4. In a similar manner, since no system can anticipate the future, an easy way to augment and extend the vocabulary and range of the original language *by the user*, without requiring the user to become a computer expert, *must* be provided.
5. The basic process of debugging will be a rather complete simulation of the tools' progress through the piece, totally controllable by the user, with views available at any magnification and orientation as orthographic, perspective or three-dimensional renderings. In this way, debugging may be carried on exhaustively before a single paper tape is punched. All debugging will be transacted in the higher level language of the system. Indeed, parts programs may be conceived, written and debugged at this level in a manner that is much easier and faster to use than APT.
6. The system should contain a great deal of machinist's knowledge, including feed rates, metal hardness and conductivity, etc., in such a way that it is easily available to both the system and the user. Not only will the user have an on-line handbook many times easier to use than a paper one, but also much of the knowledge may be applied towards automating feed, cutter and cool-out rates, etc.
7. The system should produce a workable parts program from the specifications of the geometry of the piece and the object NC tool. Both local and global optimization should be carried out to minimize a cost function which looks into accounts materials, cutters, coolants, set-up times, etc.
8. *Table driven* compilers (syntax-directed) have been in use for many years now, allowing a large class of languages to be completely described by a small number of parameters. This is the technique that will be used to implement the language of this system. It also can be used to describe a large class of machine tools simply by filling in a table. The most convenient way for this to be done is for the system to interrogate the user, asking him those questions which are necessary to fill in the table. Actually, the solution to 3a), if done correctly, (no *ad hoc* postprocessor for each machine) allows 3b), since that is the way the initial set of machines will be described.
9. Building subroutine on subroutine is a clumsy way to extend the capabilities of a language. Instead, the user communicates with a dynamic *dictionary* which initially contains a vocabulary local to the system. Whenever he defines for himself a useful construct, he may add it to the dictionary. He also may discover better ways of expressing a thought than the language presently allows. He may then define a new statement type to the system and use it thereafter.

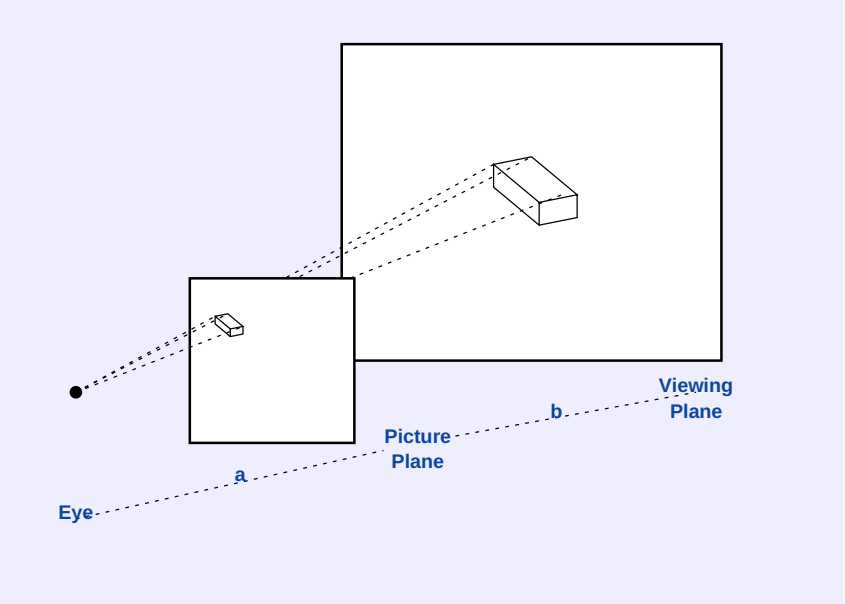
   In this way, each user will gradually extend the scope of the language, creating tools which most closely express to him the intent of a statement. Since he supplies his name every time he enters the system, the additional context that he has built during the previous months is automatically supplied.

   The language is now seen to *adapt* itself to the user's needs.
10. A parts program in this system is a *cartoon strip* consisting of a number of sequential frames. Each frame contains a picture of the piece, an indicated path for the tool, and textual notes giving constraints (tolerances, etc.) and explanations. It is in this form that a parts programmer plans his program and, because of the interactive and graphical nature of the FLEX terminal, it is possible to allow these plans to actually *be* the program - thus eliminating paper, rough sketches, etc., completely.

    A frame may be executed immediately and independently of all other frames, which allows the user to debug *as he conceives the program*. He does not have to wait until the entire tooling is described - indeed, he doesn't want to, for then the details become fuzzy and mistakes start to be made. Instead, the program is brought up in debugged sections, starting at the *frame* level, then proceeding to a *strip* of frames, then hooking a number of strips together.

    Any frame, strip or combination of these that can be made in the language can be parameterized and included in the dictionary as a vocabulary word or phrase. Thus, any successful program or part may be used to augment the basic language.

The picture of the piece in the frame is not static. The user is thought to look through a *window* at the representation of the piece and tool.



The object is fixed. The eye point and the two planes associated with it can be arbitrarily placed in the space. The view will be a projection of the object defined by the viewing plane onto the picture plane. All distances and angle parameters are under the user's control, so that the following views of the object are possible:

1. If the viewing plane is between the object and the picture plane, a perspective rendering will be shown. The *hidden lines* will not be seen.
2. If the distance *a* is increased to infinity and the angle decreased, then the view will remain the same size, but the projection will become orthographic.
3. If just *a* is changed, the viewer will appear to approach and retreat from the object.
4. If *b* is changed so that the viewing plane cuts the object, then a section will be displayed.

   So it is seen that, by allowing the user to vary only three parameters, plus the position of the *eye*, all possible views of the object may be seen. A stereo picture pair may be easily obtained if the system performs two projections with the eye point shifted 5°. This will be presented as two 512 × 512-point pictures on the face of the scope and may be perceived by using attached *goggles* to focus on the images.

   The user can change his point of view dynamically by varying parameters with his tablet. This allows the simulated piece to be presented at any orientation and magnification and allows a complete debugging of the program without the necessity to punch a tape and *try it* on the actual machine.

11. To add knowledge of the machinist's craft to the system really just requires the expansion of the system dictionary to that of an *encyclopedia*. The user will be able to apply the same techniques to find out things and, of course, add to them.

    An important part of the design of this portion is that the information be stored in such a way that it can be used by *the system* as well as by the programmer. This can be done.

    The data base will then form the basic knowledge that constrains part 7, which will completely automate the building of a program.

12. This section will be able to produce a complete parts program from just a knowledge of the geometry, materials and object machine tool. The program (as seen by the user) will be in the form of the cartoon strips used in part 5.

    This will allow the designer (or parts programmer) to review (and append) the automatically generated program.

    The method used to implement this section is basically that of mechanical theorem proving. The *facts* in section 6 are axioms to section 7. A parts program is generated from geometry and facts by proving theorems (that are implied by these facts) that are necessary to make the correct decisions. (48)

    This is the basic method used by human beings themselves in solving a problem that has not been done before.

Using the System

The designer (or parts programmer) sits down at the desk and joins the FLEX machine. He is automatically connected and enters the NC system. The first thing that is displayed is the top level of his own personal dictionary. He may choose to peruse it (explained in the next section), or he may just recover his context from the previous session and continue where he left off.

Since all portions of the language system consist of pictures and text, his normal mode of operation is as follows:

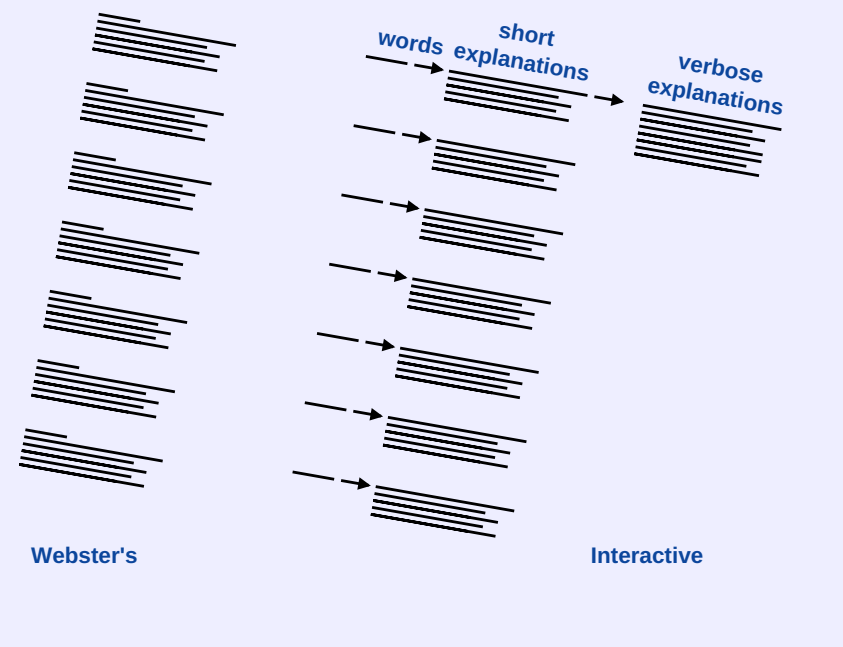
When typing in extended text, he will have both hands on the keyboard, as on a normal typewriter. Whatever he types will be echoed on the screen and he will have complete ability to edit and amend his text. When typing in statements in the language, the system will be interpreting as he goes, so that the system can do a number of desirable things.

1. The system will immediately inform him of a statement that has been typed incorrectly. He will not have time to forget what he is doing before being informed of an error.
2. The system can anticipate what he is doing. If there is a word *VECTOR* in the vocabulary, and it is the only one beginning with *V*, the system will know after the user types a *V* that there is only one possibility for the remaining letters, so that it can echo back *ECTOR*, thus speeding up the user's typing many times. This is primarily a feature for the expert user, and so can be turned off and on at will. Users of the SDS-940 time sharing system (where this is a standard feature) swear by it and greviously miss it on other systems. (45)

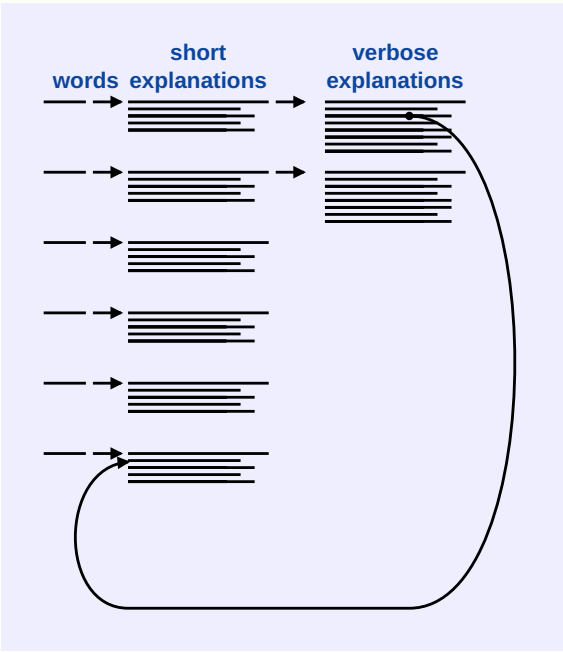
For pictures, the user will have one hand on the stylus, the other at the five-finger keyboard for function codes. (Both right and left-handed people may use their strong hand for pointing.) Words cannot do justice to the speed of interaction that is possible after only five to ten hours of use on a system like this. It really has to be demonstrated. (46)

The Dictionary

On an interactive system, this can be an exceedingly powerful tool. Webster's Collegiate is very clumsy in comparison, since all of its information is basically at one level. To find a word, the user knows how it is spelled, yet he is forced to wade through all of the information (including explanations) to find his entry:



Webster's International is rarely used, because almost never does the interrogator want as much information as there is in a typical entry. On an interactive system, the computer is on hand to search, so that *just* the words can be presented. The desired word can now easily be found and pointed to by the stylus. The display can then show a condensed explanation of *just* the word pointed to. If more information is needed, the stylus can again be used, to cause an expanded *encyclopedia* entry to be displayed. More tricks can be played. Suppose a word in the explanation is not understood. Does the user have to retreat to the top level to select the entry for the new word? Not if vocabulary words are linked together. Then the stylus can be used to point at the word in the explanation itself and its entry will then be displayed.



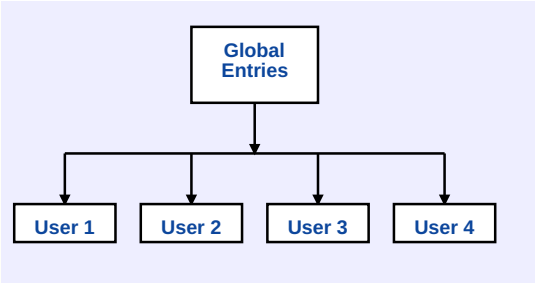
The user will return from each explanation in the reverse order in which he entered so that a perfectly natural way of using the dictionary develops. Whenever further explanation of something is needed, the user *descends another level*; after his curiosity is satisfied, he returns to exactly where he was.

In a sense, the vocabulary is not structured alphabetically only, but is set up so that a minimum of perusal is necessary to find out something. (51)

Documentation

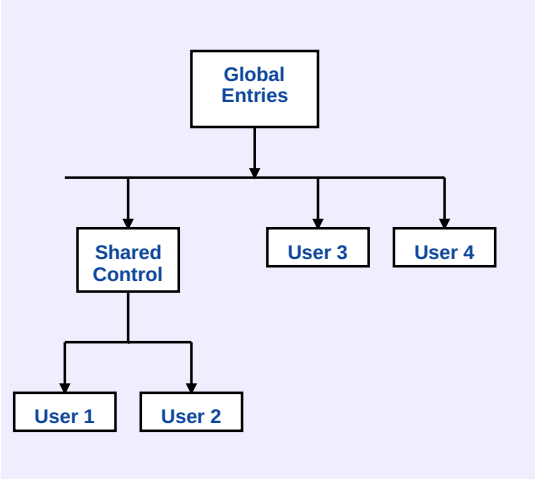
All knowledge there is of how to use the system will also be carried in the dictionary. Updates made by a programmer will thus be reflected automatically to all users. There will be a *mail* entry to leave messages, etc.

The user also adds and subtracts items from his dictionary. All programs and current working documents are saved and correlated as vocabulary entries. Although the programmer can define and delete items, the initial dictionary, global to all users, is made read-only so that it cannot be destroyed. Of course, the user can make any of his entries read-only also, changeable only by supplying a special key word.



Dictionary Structure

The dictionary is seen to be hierarchical in nature itself, with each user on a different branch. This prevents any user from destroying another user's context. Of course, it is easy to arrange that two or more users share some context.



Pictures and Objects

The screen is a matrix of dots, 1024 × 1024. This means that a rather high definition stereo pair may be presented as two 512 × 512 pictures on the upper or lower part of the screen (A TV picture has 525 horizontal lines).

Stylus and Tablet

Another absolutely necessary feature of the FLEX display is its facility for the user to point at picture elements. When the stylus is moved around on the surface, an x voltage and y voltage are applied alternately across the diode-isolated conductive rubber and measured by a voltage sample and hold. This information (of relative x and y) is used to position a non-writing dot of light on the screen, which moves as the stylus is positioned. The user never looks at his hand, but just follows the dot of light on the tube.

To point to a line already on the display, the stylus is moved until the light dot (cursor) is near the line. The stylus is depressed to tell the display to send the current x, y positions of the stylus to the computer. By various means the system can then determine what previously sent lines pass near that point.

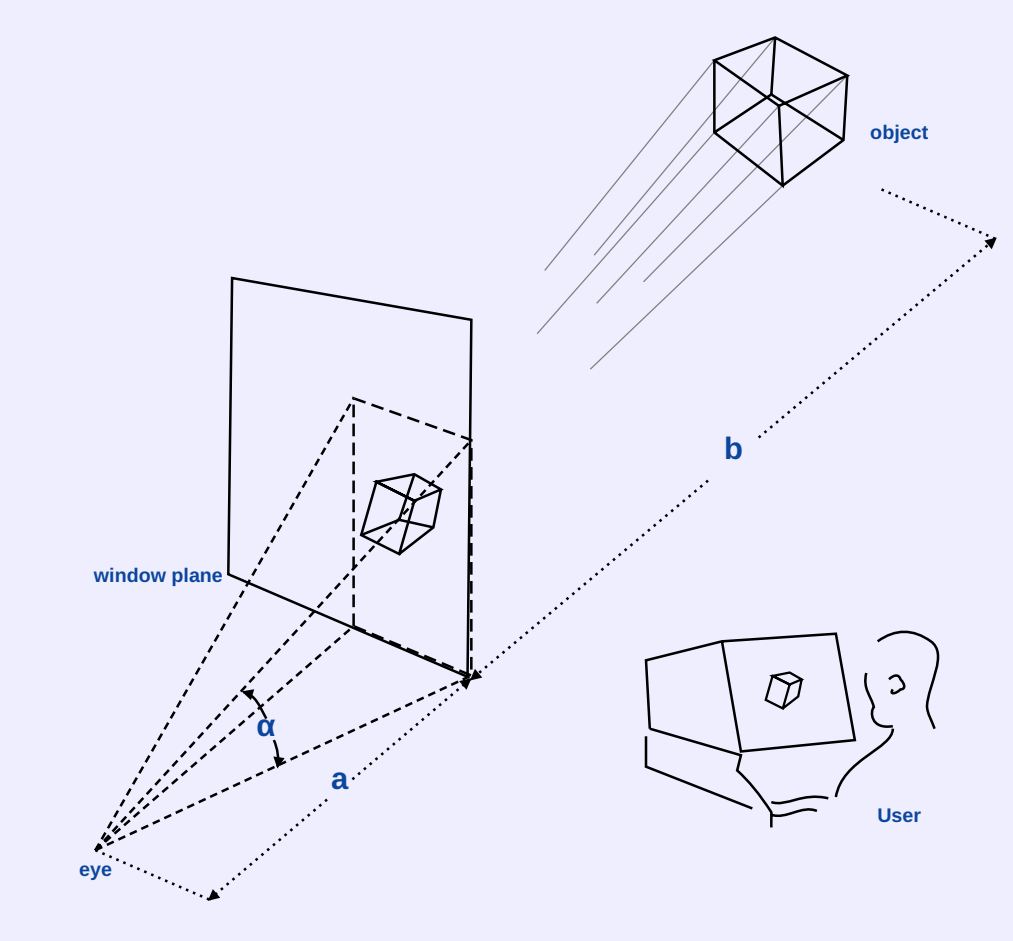
Graphical entities are stored as models of two or three-dimensional objects, *not* as representations. This is important, since much information as to the solid geometry of an object is lost in a picture of it.

However, the FLEX display (like many other displays) is only capable of a two-dimensional (or flat stereo pair) representation, so a reconciliation must be made when the user demands to see something.

The scheme used is to have the object remain fixed and allow the user to adjust the position of a mathematical *eye*. It turns out that all possible perspectives, orthographic and stereo views of an object can be obtained by simply projecting the object on a plane perpendicular to the eye and placed in front of the object. If the plane is bounded and identified with the viewer's CRT, then all magnifications can be also obtained by adjusting the angle (X) of vision. (44)

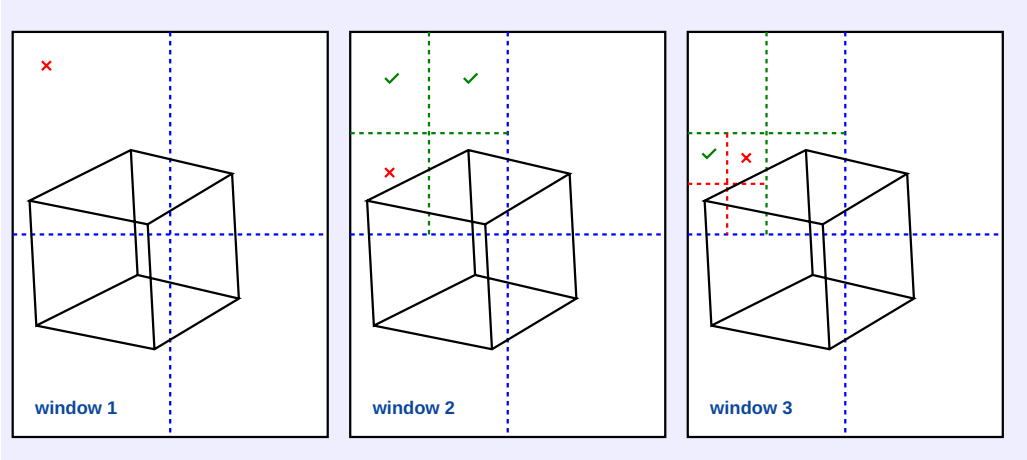
All of this can be done by manipulating one 4 × 4 matrix.

Note that all the edges of the cube are visible. This gets to be quite annoying very quickly. The solution to the problem of discarding *hidden* edges in a small amount of time for any object was not known until recently, when it was discovered by John Warnock at the University of Utah. (47)



The solution follows:

Hidden Edge Removal



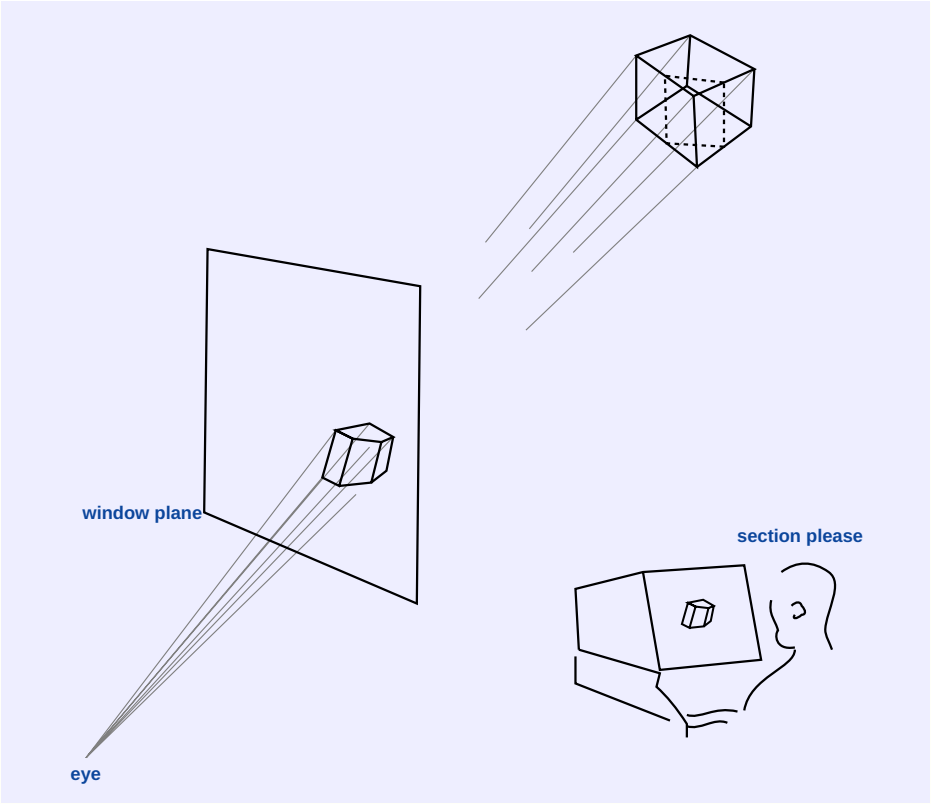
The projection onto the window plane is done as before. The rectangle of the window is examined. If it is *complicated*, the picture is subdivided into four quadrants. Each one is examined in turn (X). If it is *complicated*, then another subdivision is made. If it is blank or only has one edge, then it is not *complicated*, and may be checked off. Since the display can only show a finite number of dots, it is not necessary to subdivide more than nine times ever in a quadrant.

The first *uncomplicated* quadrant with just an edge is found in picture 3. This whole process will take a time somewhat proportional to the complexity of the object.

The FLEX version of the algorithm works in a similar manner, except that all quadrants*of the same size* are checked before subdivision takes place. This finds the largest simple lines first and sends them over, then shorter segments and, finally, crosshatching. The user quickly sees a *graining* but complete picture with fine detail constantly being filled in. Because communication is two-way simultaneous (full duplex) the user can act before the complete picture is sent. He can make a change and start the process over before it has completed. If he has to think about something, the picture will be completed while he waits.

Section Views

All possible section views can be created by simply using another plane (the section plane) whose orientation is related to the object.



The portion of the object behind the section plane will be displayed. Finding the section surface out by the section plane is an application of the hidden edge algorithm. The section plane is related to the object and may be manipulated independently of the eye. Any number of section planes may, of course.be used.

Shading

Objects are assumed to be solid, and the hidden edge algorithm acts as though the light source is at the eye (so that no shadows need to be computed). Warnock of Utah generates a synthetic T.V. raster with the apparent brightness of a surface proportional to both the angle of that surface distance and a *smoothness* characteristic. It would be nice to be able to do that for this system (and the future holds promise), but for the present, a display processor cannot support that much information, so a different tactic must be used.

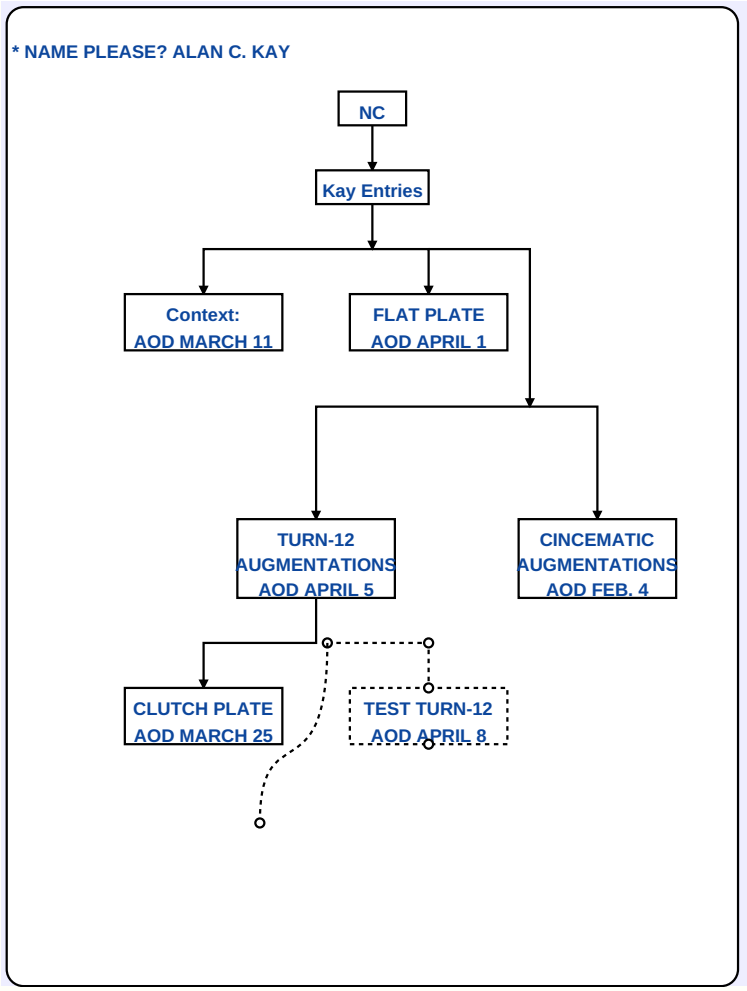
This system will employ a crosshatched mesh for shading; the mesh size determining the *darkness* of the face. A face created by a sectioning plane will have a characteristic surface hatching to uniquely identify it.

Augmenting the Language

This will be covered in more detail after the basic exposition of the language. It suffices to say now that anything the user creates with the system may be given an arbitrary name and included in the dictionary. This includes machines, tools, objects, subroutines, notes, etc. The system does *not* distinguish between its own entries and the user's. Therefore, a verb (standing for a subroutine, perhaps) created by the user has as much power, scope and useability as the system-defined verbs.

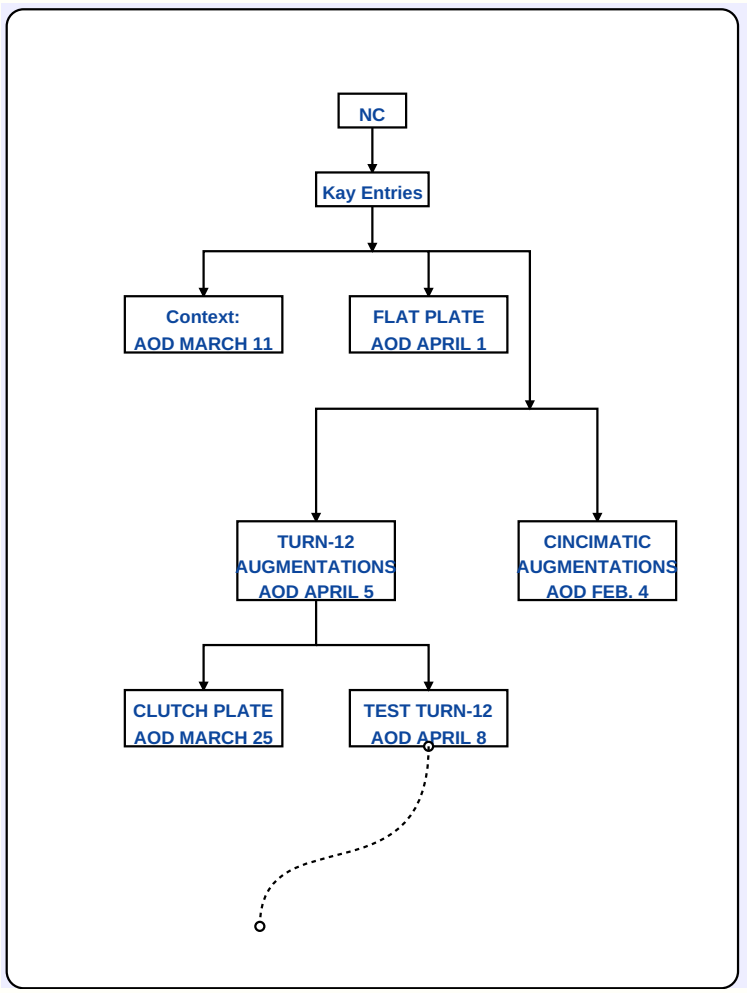
The grammar of the language may be extended by the user in a similar manner - i.e., by*example*.

A Sample Program

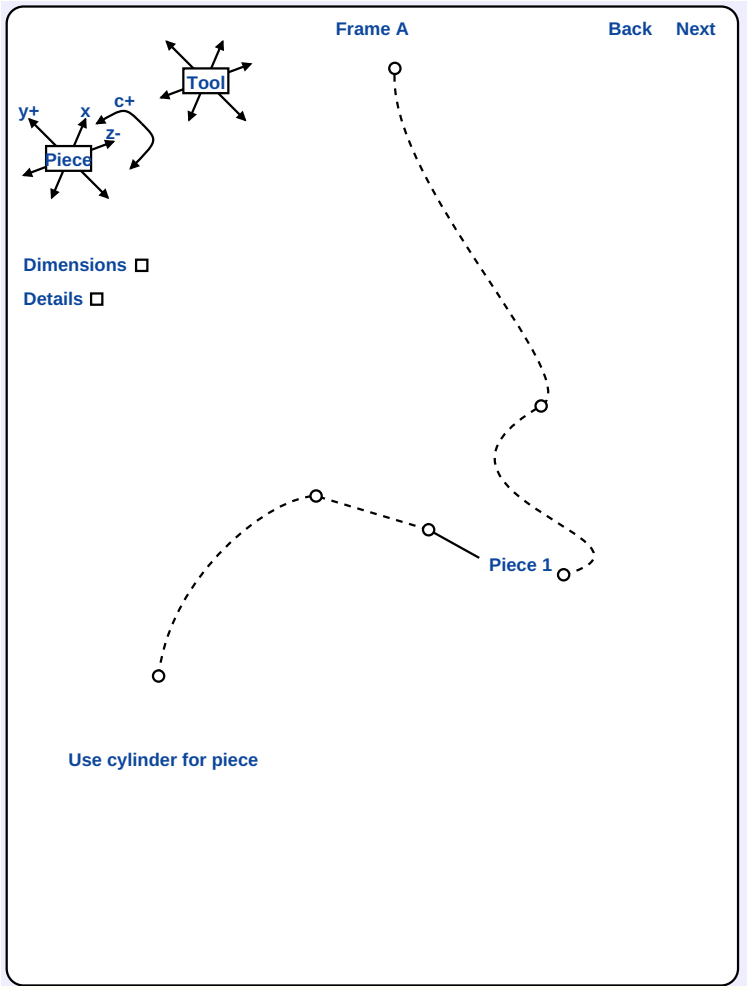


The user logs into the system and gives his name. His previous context is shown as a structured dictionary. He is doing a new piece on the TURN-12, so he moves the stylus to create a new entry.

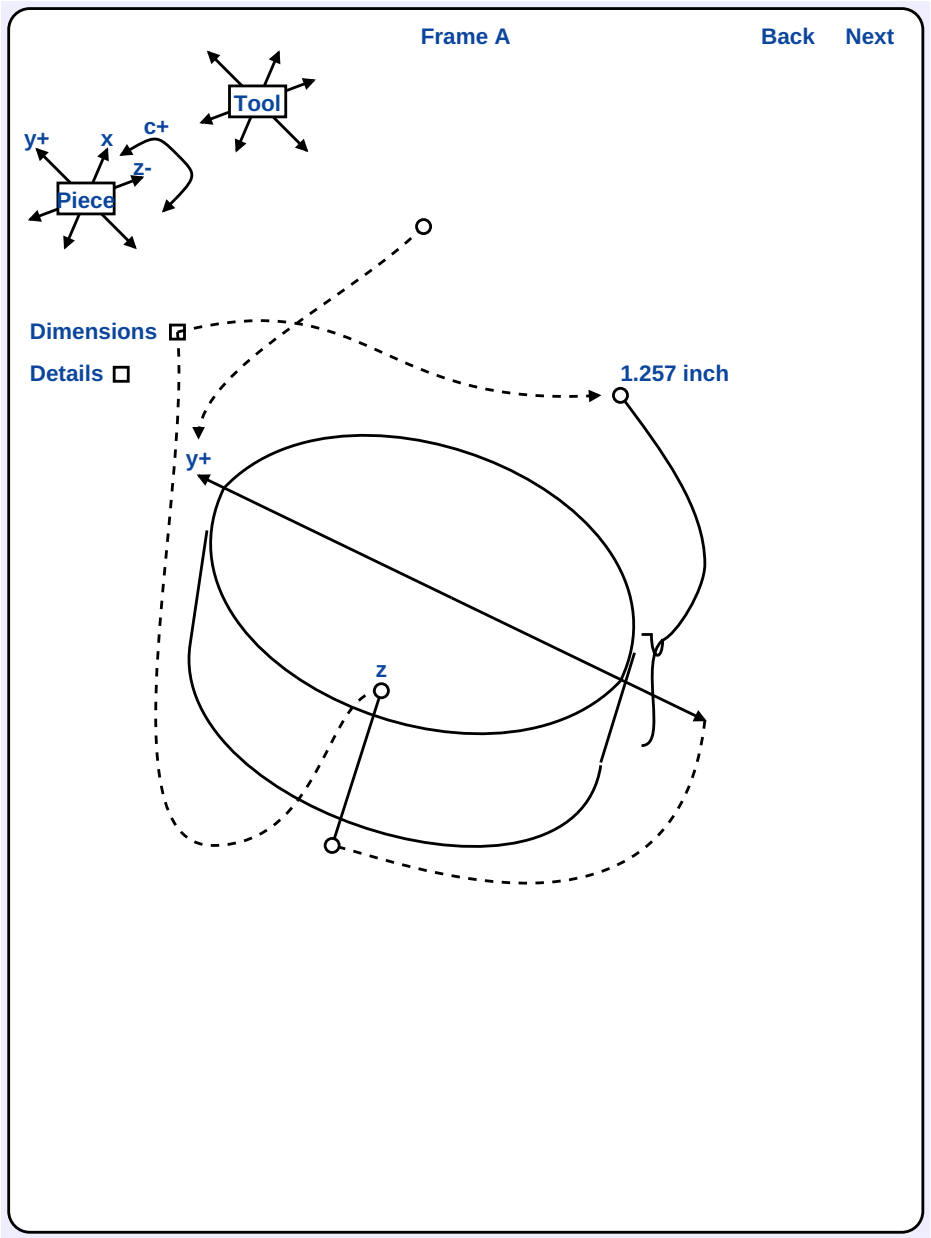
This yields:



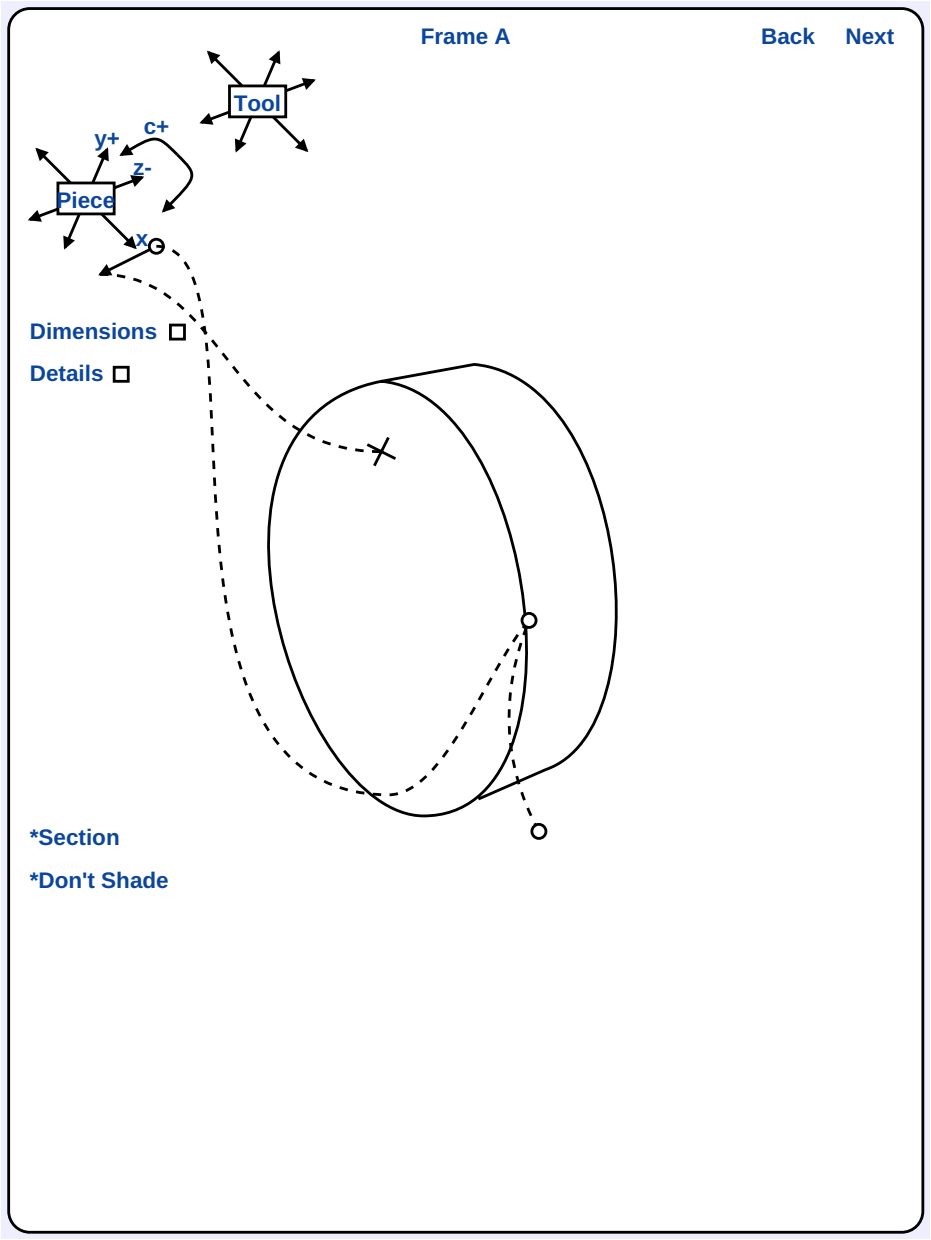
To start building the program, the user points into the box he just created. All upper nodes in the tree are global to him. All other nodes are invisible. Everything he creates will be logically *inside* the box called *TEST.TURN-12*.



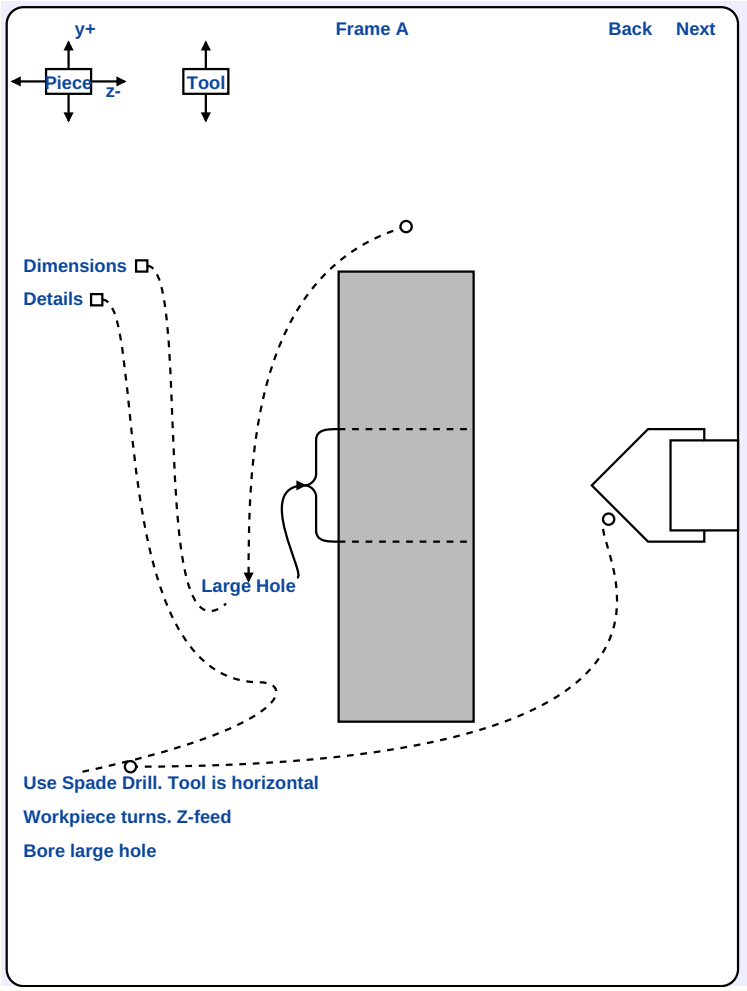
The user now sees a symbolic representation of the degrees of freedom available to him. Note there is no x freedom on the piece. This telltale will orient itself as the user's view angle is changed. He now begins to describe the blank.



He decides to reorient the piece and then proceeds to set up a section for the first bore.



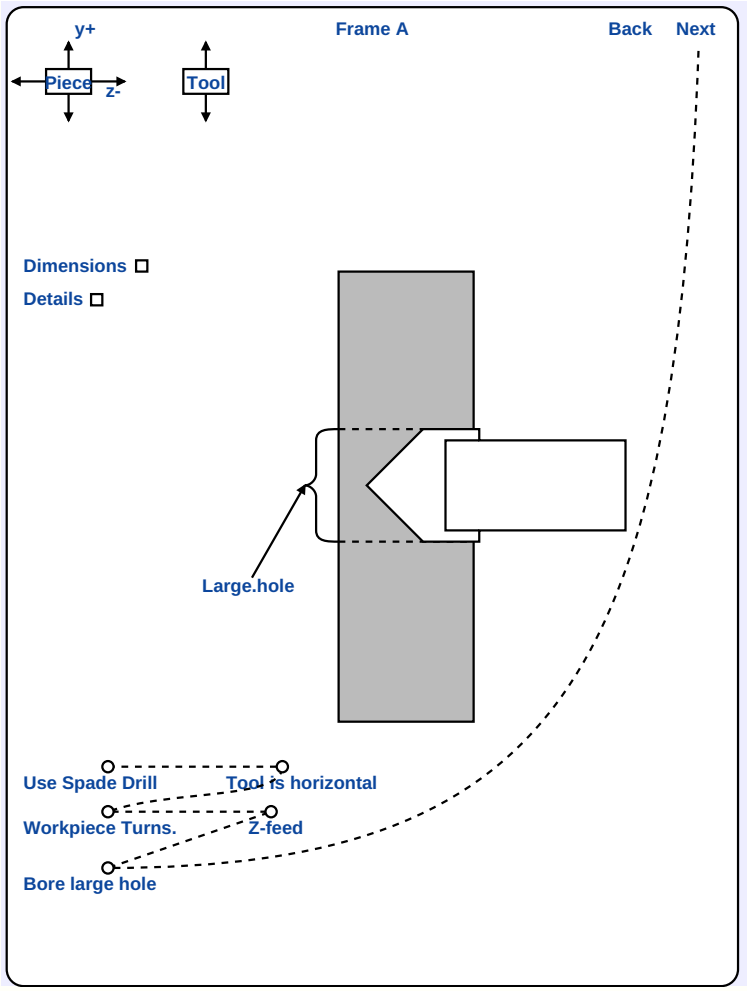
He has decided to submerge the dimensions of the piece one level for clarity, so there is a *light button* that he can point to with the stylus for that purpose.



All the foregoing has been executed immediately by the system. He is still on his first frame and is now ready to state his intentions to the language. *Large.hole* is defined. Dimensions are entered at sublevel. The words *Spade.Drill*, *horizontal*, *turns*, *Z-feed*, etc. are dictionary entries. They may have a previous meaning, or they may be defined at will by simply pointing at them.

Nothing will happen on the frame until the sentence *Bore Large.hole* is encountered. The user announces his intentions of removing the material that is described as *Large.hole*. If the tool and the volume to be removed are incompatible - wrong diameter, etc., an alarm is given.

The frame may be tested by typing *test* as a direct command.



As the command is executed, the cursor dot follows the program path, leaving a visual trail. The system feeds back OK if it found no obvious errors. The user then does the next frames in a similar manner.

III. Part B: Physical Considerations

> *One can conceive of Heaven having a Telephone Directory, but it would have to be gigantic, for it would include the Proper Name and address of every electron in the universe. But Hell could not have one, for in Hell, as in prison and the army, its inhabitants are identified not by name but by number. They do not have numbers, they are numbers. - W. H. Auden, Infernal Science*

4. The Pragmatic Environment

Introduction

In this part, we first consider the problems of physically realizing the philosophies presented in the previous sections.

There have been numerous approaches to solving this problem; some successful, many more unsuccessful. Computer programs in general also seem to work according to the same ratio.

One bottleneck is the attempt to *do all things for all people*; another is to try to make the program work at 100% efficiency 100% of the time. The first method usually entails huge, unmanageable programs; the second means that much fast hardware will have to be used.

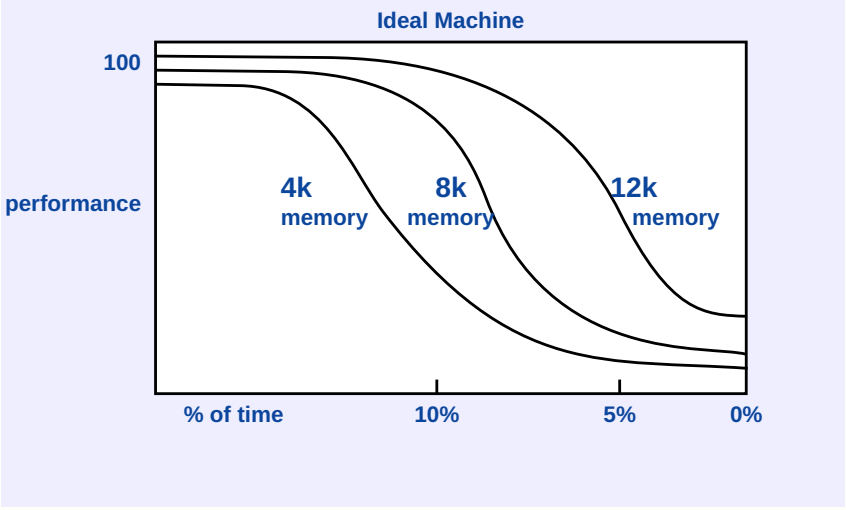
The FLEX environment on the object machine takes a different tack. First the machine structure may be designed so that it is compatible both with the language that will be executed and with the problems that will be solved.

Second, a statistical viewpoint is adopted. For almost all computer problems in general on any machine (and in particular those problems for which the object machine is suited) neither 100% efficiency 100% of the time nor blinding speed is necessary. Fortunately from the software point of view, the first is not needed and thankfully, for the price tag, neither is the second.

The environment seeks to keep the overheads to a minimum for things that are done 90-98% of the time. This means that most of the time the machine will act as though it were far larger and faster than it actually is. Witness some statistics from Stanford where a Burroughs B-5000, a machine suited for algorithmic languages, actually ran most problems faster than an IBM 7090 - a machine whose hardware was significantly faster than the B-5000.

Of course, occasionally, the piper must be paid. The FLEX system seeks a graceful degradation in performance as the load goes up. The machine simply appears to slow down. When there are too many active segments or numerous quite large segments in core memory, an increasing burden is put on the secondary storage. Where, most of the time, the cheap secondary storage allows the machine to look as though it had a large core memory, now saturation will force operating speeds to approach the speed of the secondary storage rather than that of the primary.

Another interesting consequence of this point of view is that the environment works quite independently of the particular storage limitations and conversely the efficiency of the machine depends very much on these same limitations. What does this mean:

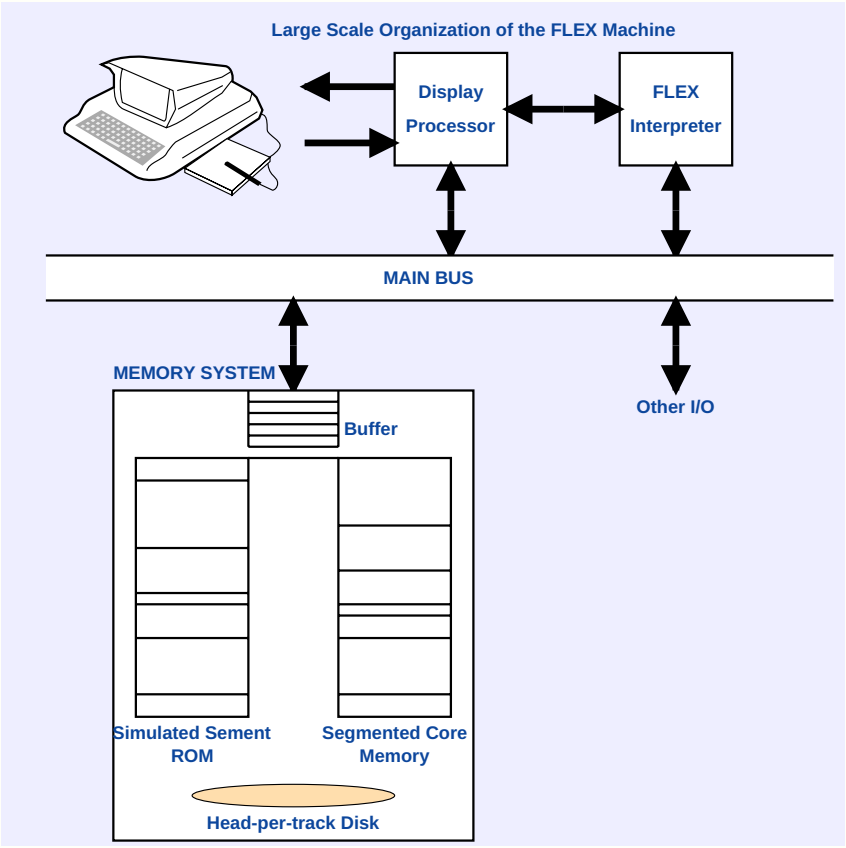


It means that for most problems, an increase in memory size will not drastically improve performance, but it will dramatically reduce the percent of time spent in overhead when the system becomes clogged.

It also means that the physical system may be expanded or reduced without requiring adjustment of programs - a handy feature. Most programs will not run any differently with an increase in memory; a few will run significantly better; still fewer will continue to flog the system.

Although there is a specific device called "*the* FLEX Machine", any computer programmed with a FLEX interpreter is, in some sense, *a* "FLEX Machine". Accordingly, the following sections are presented with a number of prospective environments in mind. Most particularly, the main interest lies in considering the complete hardware representation for FLEX, the large time-sharing system implementation, and FLEX in the micro code of a commercially available mini-computer such as the Interdata Model-4.

Gross Organization of the FLEX Machine



The layout is straightforward. The segmented memory is completely distinct from the other processors which are just hung on the main bus. The memory system has its own buffers, and byte extraction is performed there.

Reentrant FLEX system code is contained in cheap Read Only Memory in the form of simulated segments which are accessed in exactly the same way as volatile memory. As will be seen in Chapter 6, there are many ways to build this particular box, none of them terribly appealing.

The FLEX and display interpreters are driven by micro-coded routines also stored in ROM and still in the addressing space of the machine. This is so new micro codes may be tested out simply by directing the processors to the new segments.

5. Segments and Processes

a. Segments

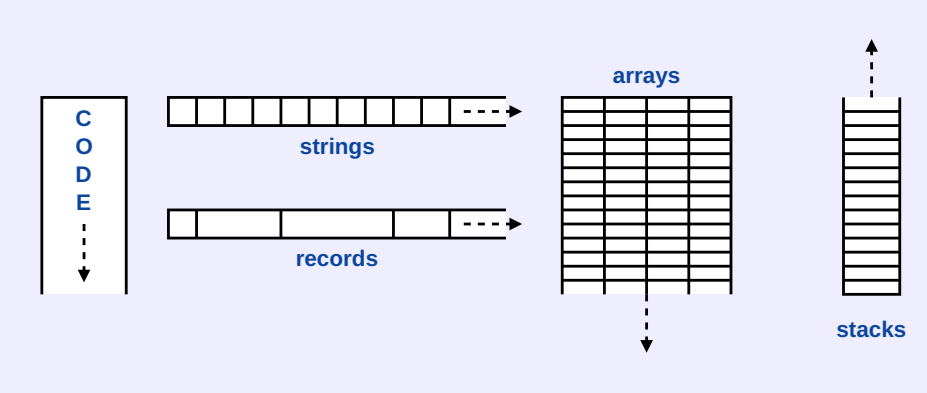
In the FLEX operating environment the basic logical structure and the basic physical structure are one and the same: The segment. Logically, the segment is a contiguous string of bits in core memory and secondary storage whose length may be changed with varying degrees of effort.

Addressing in the system is relative to the segment, not to any particular memory location, so that a particular segment may be moved anywhere without disturbing access to it.

```
----------------------
| name | displacement |
----------------------
```

Typical Segment Address

The idea of a logical segment seems to have something in common with other logical entities such as a sequential program, *records*, arrays addressed by position, stacks growing at one end, strings accessed by the relation *next*, etc. It does not adapt itself well to other useful structures such as double-ended queues (and queues of all kinds) LISP pointers, arrays which dynamically change dimensions, etc., so that compromises need to be made when mapping these structures.



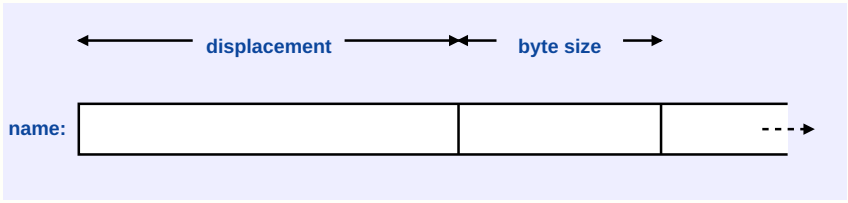
Segment Structures

The prime reason for the use of segments is that they cannot intrude on each other because of the way the address is formed. The *name* portion of the address may not become arithmetically involved with the calculation of the displacement. This means that no segment besides the one named can be accessed.

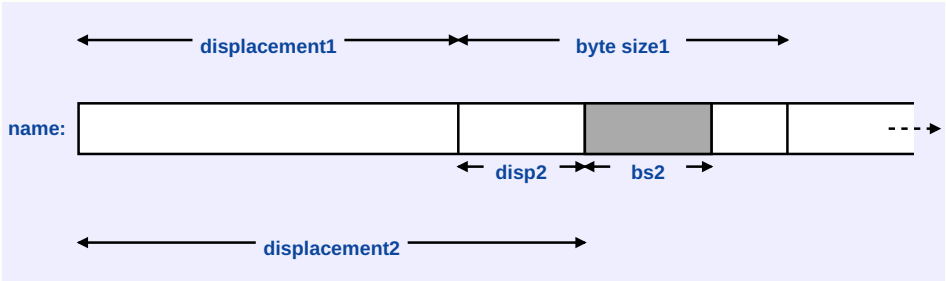
In order for the accessing of information from the segment to be perfectly general, the following commands to the logical memory have been developed.

```
create name;
destroy name;
fetch name | displacement, byte size
store name | displacement, byte size
```

If the displacement is given in bits from the*front* of the segment, then any field may be accessed.

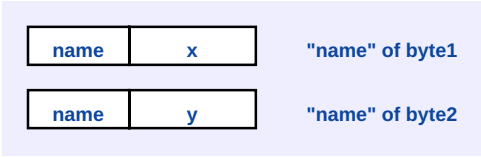


Since the displacement and byte size are truly independent of the segment name, a function *fld* may be created which acts on a described byte to produce a new byte description.



The shaded byte may be described in two ways: as a subfield of*byte size1* via *disp2*, etc., or as a subfield of the segment *name* via *displacement1*. The scheme becomes generalized if sub-bytes may be named like segments; then the situation is one of fields within fields.

However, there is one important difference.



The *names* of the two bytes are distinguished by the displacement field in the address, which may be calculated arithmetically. Since this is true, *name x* may be derived from *name y* by a simple calculation and, hence, the information it refers to may be too easily destroyed.

Now it is seen that *each* segment acts like an independent randomly-accessed core memory, with all the rights, privileges and drawbacks associated with the same. The extra level of protection needed to maintain the integrity of bytes within segments is provided by the descriptors in the *single language* system FLEX.

b. Process Control

The basic data structure has been discussed - now the basic execution entity will be covered: the *proces*.

Definition of Terms

A *process description* is a segment that contains interpretable code generated by some means. By its very nature, this code is reentrant, which means that it does not modify itself and therefore may be in several stages of execution at a given time.

A *process* is just an *instance* of execution of a process description; there may be more than one process in existence at one time for a given process description. (52, 53,54)

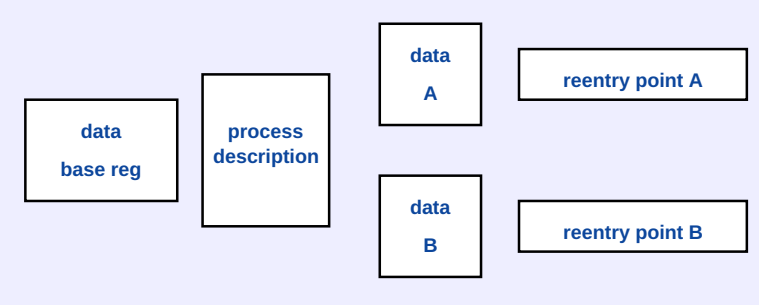
Parallel processes are required for operation of the system. The I/O, the display, the keyboard, the compiler, etc., must be able to run concurrently with the user's programs and with thenselves. Moreover, the compiler which is interacting with the user at the keyboard may have to run in parallel with a logical *copy* of itself; e.g., executing the *com* operator in a user program.

Since this mechanism is needed, it is no trick to allow the user in FLEX to create concurrent processes, of his own - all handled by the same algorithms.

This ability is literally invaluable for all kinds of programming, recursion, and event-oriented simulation -- a prime use for the FLEX Machine.

Process Creation

The basic idea is simple. Since the process description (the code) does not modify itself, it can contain no data. Therefore, it must have some way of accessing data which is independent of itself.



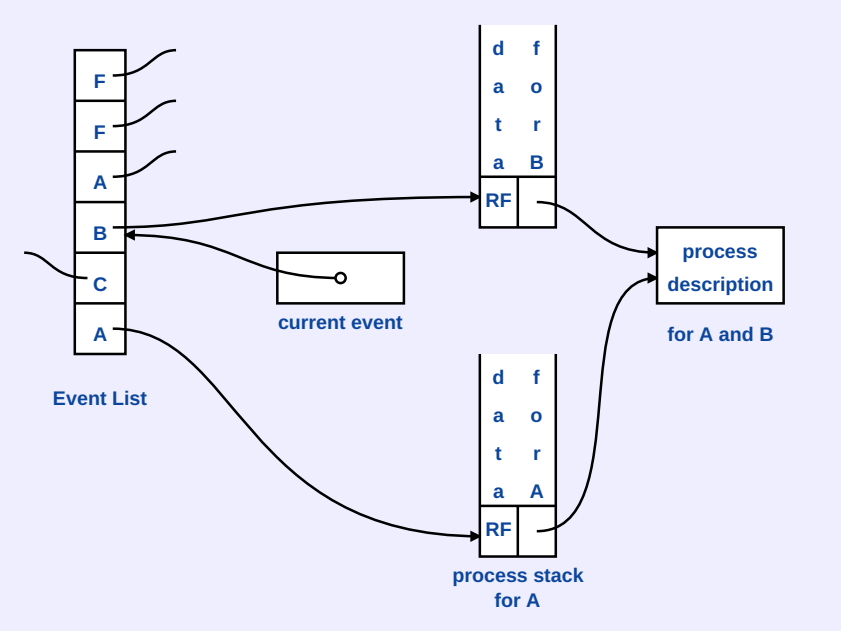
Process Description with Two Sets of Data

One way this has been done in conventional machines is to create a separate data area for each process and to force the process description to access all its data through a base register which contains the low order address of the desired data. Now the process description may be switched from one process (the handling of data A) to another (the handling of data B) with ease, provided the reentry point of each process is retained while the other is being executed.

To effectively run the two processes in parallel, a fixed time of execution may be assigned to one process before the other one must be started up. This is the *time quantum* and typically is about 10-16 ms.

It is not difficult to generalize this idea to the FLEX environment and the segment system.

The *process description* is a segment. Each data area becomes a segment and the base register refers to the data segment name. The reentry points may become part of their associated data. All that remains is to formulate a scheme for scheduling the execution of each process. A simple list containing the process names which is visited *round-robin* fashion every 16 ms will do.



The figure shows the system about to execute B. This is called *activation*. During execution, process B is said to be *active* and the period during the duration of B's *time quantum* is said to be an *event* of B. All entities in the figure are segments, and thus may reside on secondary storage, if necessary.

Process Stack

Because of the well-nested properties of algorithmic languages in general, and FLEX in particular, the data for a process is an extendable segment called a *process stack*. State information which is necessary for each event is retained in the base of the stack, such as the process description name and the reentry point associated with it.

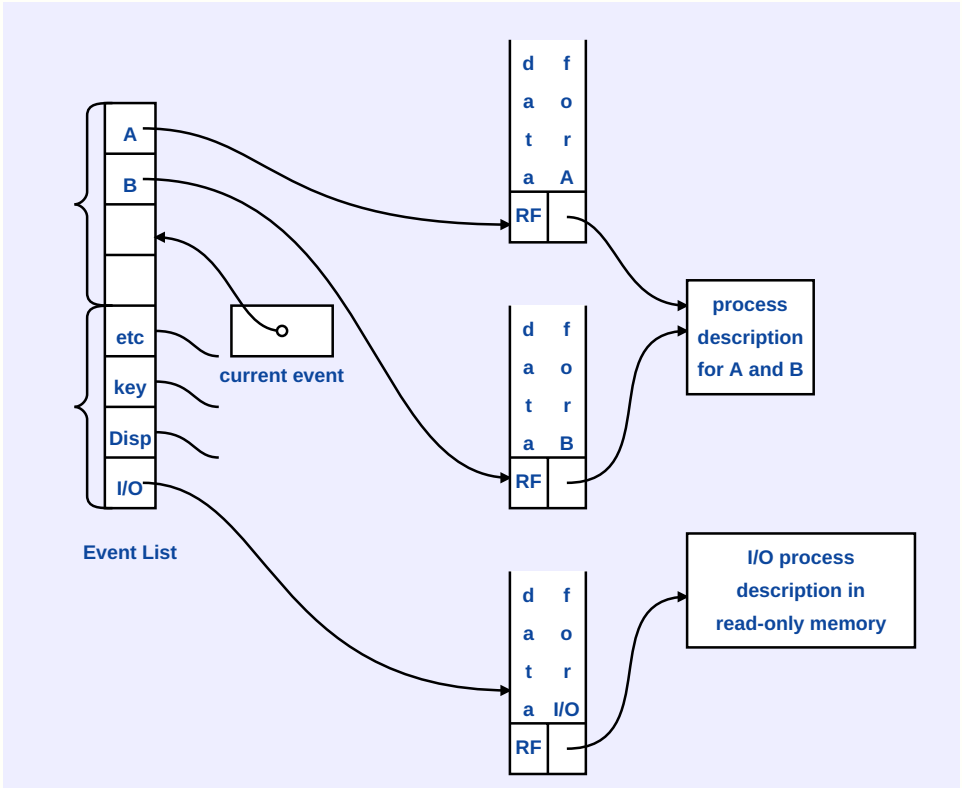
While B is having an event, the other processes are said to be *passive*. When the *process stack* is collapsed and the process name is removed from the event list, the process is said to be *terminated*

Passivation

A process may be made *passive* by more than just the ending of an event. In general, when a process initiates an I/O operation, it will be *passivated* while the I/O is running.

Indeed, all the real-time processes such as the I/O, the display, the keyboard handler, etc., because they cannot wait when something that involves them is happening, have the ability to *passivate* (or interrupt) any other process and to *activate* themselves.

The round-robin algorithm must be modified slightly to accommodate the real-time processes.



All the real-time event notices are logically clustered. Between each event they are scanned by the scheduler to see if activation is required. If not, the process pointed to by the current-event pointer is *activated*. At any time an outside interrupt may point into the real-time area. When this happens, the current event is *passivated* and the, real-time event *activated*.

Data Segments Associated with a Process Stack

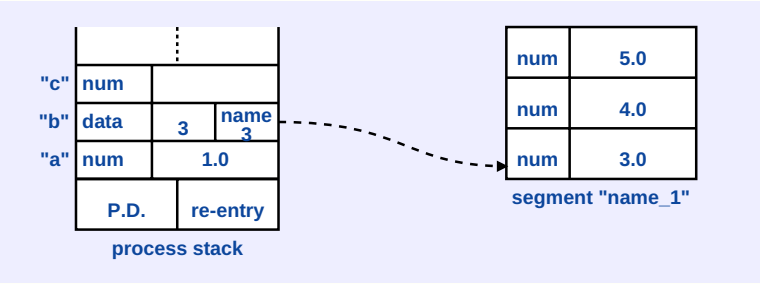
The exact format of a process-stack will be discussed in the next section on execution. Now it will suffice to say that each slot in the

process stack is associated with a different variable name in FLEX. During compilation a variable name is transformed to a relative index in the stack. The slot itself may hold a number or a pointer (called a *descriptor*).

If the variable remains unmapped and contains only numbers then this data may be accessed directly with no overhead. If, however, a list, any other entity, or a map is assigned to the variable, then the data is put in a fresh segment and a descriptor is created containing the new segment name and a description of the data. The descriptor is stored in the slot and is effectively a self-typed indirect address.

Example:

```
'list:(a,b,c).
a   1.;
b   (3,4,5)';
```



Program and Realization (Schematic)

The slots are created in order and are initially set to Ω. *a* contains a 1.0 while *b* contains a descriptor which effectively points to the freshly created segment *name_1*. Of course, the data descriptor in *b* cannot contain the absolute address for *name_1* because *name_1* may have migrated to secondary storage during some other processes event. So it must contain a name for the segment and the absolute address at any given time must be looked up.

Unique names for freshly-created segments are doled out by a system routine and consist simply of a 12-16-bit integer.

6. Memory Systems

> *Myself when young did eagerly frequent Doctor and Saint, and heard great Argument About it and about: but ever more came out by the same Door as in I went. --0. Khayyam*

6.a. Mapping

A number of methods for creating and maintaining named logical segments will be discussed. All involve time, space and money trade-offs of one kind or another.

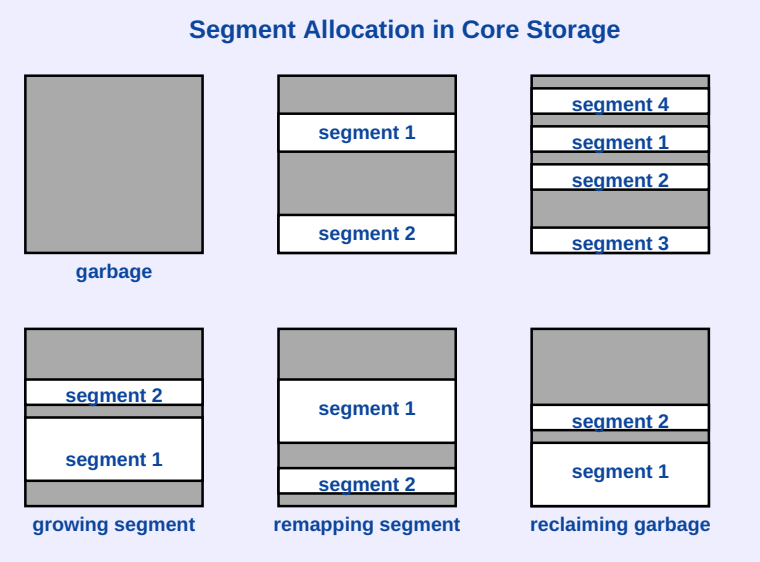
The first may be used when a minimum of money and extra hardware is available, since it requires *firm* or *soft* algorithms to maintain the fiction of orthogonality, as well as a *soft* name-address transformation table.

A second scheme also uses core and secondary storage but does the transformation dynamically, using special-purpose hardware designed for augmenting the abilities of core addressing.

Multi-level schemes are also discussed in regard to both the advantages gained by adding structure in this traditionally unstructured area, and to the costs involved by undertaking complex mappings for each access.

6.a.i. Contiguous Mapping

In this method, a high-speed memory initially consists of one segment called *garbage*. All other segments in the system are created by portioning the garbage. An attempt is made by the system to intersperse garbage segments between active segments. This allows some expansion without rearrangement of other segments. This strategy will work well with relatively static entities like process descriptions (code) and arrays. Process stacks are another matter, and some shuffling is required.



There are several ways to remap segments. The cheapest is to find a large enough area of garbage and reallocate the segment. If this cannot be done, then garbage must be collected and arranged to form a large contiguous free space which may then be used for allocation and new segment creation.

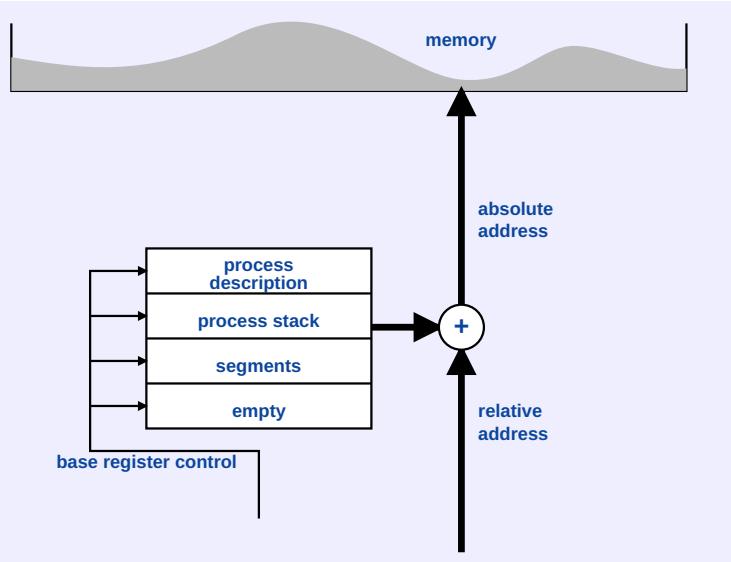
If no garbage is available, then some must be created by transferring one or more active segments to secondary storage. This operation is usually called *swapping*. If there are many segments in the system, but only a few are used at any one time, then the swapping overhead will be low and the machine will act as though its useable core memory is much longer. If there are many segments, and many are accessed, then a system clog is created, and the apparent access time becomes longer.

In all cases when a segment is expanded, it is not lengthened by just one word but by a number of words equal to some fraction of its current length. This allows some room for further expansion without disturbing the system.

A FORTRAN or ALGOL program consists of just one process and so do many programs in FLEX. Therefore, process handling and access should be optimized. This is accomplished by filtering every request to memory through one of four base registers. Since the rearrangement of memory requires an *activation* of the garbage process, during an event, core memory is still and the base registers may contain absolute addresses. These addresses are calculated during an activation and are the only contact with absolute address that the entire system has. (Excluding the *garbage* collector, of course.)

Base Registers

During an event one base register is free for system use. Another holds the base address of the process description; the next contains the base address for the process stack. The last, as will be seen later, will aid in accessing other segments.



Addressing

It can be seen that although the FLEX environment has effectively done away with direct addressing and introduced relative (and moveable) data entities in the segmenting scheme, things actually hold still for the major part of the time and the basic overhead during an event is a fast add whose time will be absorbed by the micro code hardware.

The Segment Association Table

The operation that needs to be performed is the same as the associative operations in FLEX and the same mechanisms and formats are used. When a segment is created or brought into core storage, we wish to form an association thus:

base-address • seg-name ← loc;

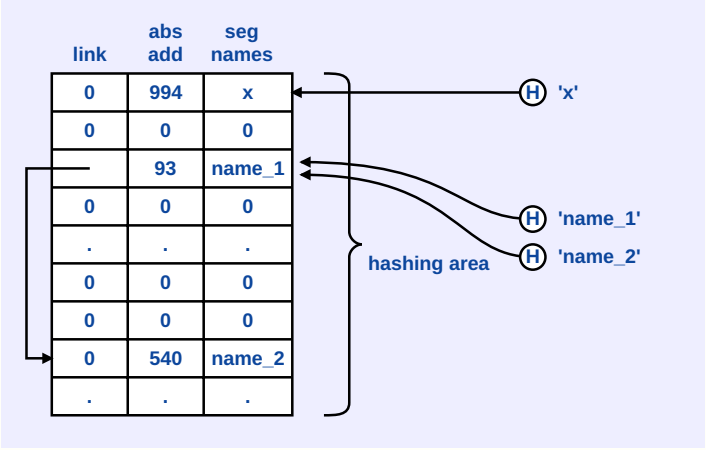
The inverse operation needs to be performed when the segment is accessed:

base-address • seg-name ← Ω;

If the association fails, then the segment is residing on the secondary storage and a somewhat more leisurely search may be made to find it and bring it into the core memory.

The Association Structure

This will be covered in greater detail in the next section on execution, but the operation may be demonstrated schematically. Of course, the table itself is a segment and has a logical format as shown:



The Segment Table

Associative hardware is prohibitively expensive, so it cannot be used to retrieve the information. Feldman (37) and others (55) have shown that hashing may be used to simulate an associative memory very effectively. The method used is similar to that in LEAP (36), although the hashing technique is derived from a different source (56).

The name, a 12-16-bit number, is reduced by the hash to a 4, 5 or 6-bit number which is used as an index to search the table. Since not even the best hashing algorithm can totally eliminate the possibility of two names hashing to the same place (as *name_1* and *name_2* have done) provisions must be made for this eventuality.

After the index selects a row, a comparison must be made to see if uniqueness obtains. If it does, (percentage dependent on the hash size) then the absolute address may be delivered without further ado. If the name column contains a zero, then there is nothing in core memory that hashes to this slot, and the segment must be out on secondary storage. The same applies to the case where the comparison fails and the link field is zero, indicating a chain is present. The overhead for a good hit is 2 μs memory cycles. That for a fault is 1 or 2 memory cycles.

Now the high overhead case is considered. If the absolute address of *name_2* is required, a chain of multiple hits must be followed. Fortunately, this does not happen very often.

In all the associative structures the percentage of multiple hits is calculated. 2-4% is the maximum allowable level; when this is exceeded, the associations are recalculated for a larger hashing area which pulls the multiple hits down to a safe level.

This scheme follows the philosophy of the FLEX environment. Most of the time it looks like something much better than it is: an associative memory. For 2-4% of the time it looks like a list-processing table.

Segment Creation

A segment is created by converting an area of garbage into active storage. A name for this segment must be found and entered into the segment table.

Since creation and destruction are dynamic, a way must be found to maximally utilize the small number of segment names available. This could be done by maintaining a pool of unused names in order to provide a new, unique label for a segment - but this is costly in terms of storage, so a different path is taken.

When it is desired to create a segment, garbage is found (or made) in the usual way. The machine contains a random number generator which is used to select a name. An access request for that name is then made to see whether or not that number is already in use as a segment name. If it is, a new random draw is done and a new test is made. Very rarely will a selected name be in use, so the algorithm will almost always work the first time.

Secondary Memory

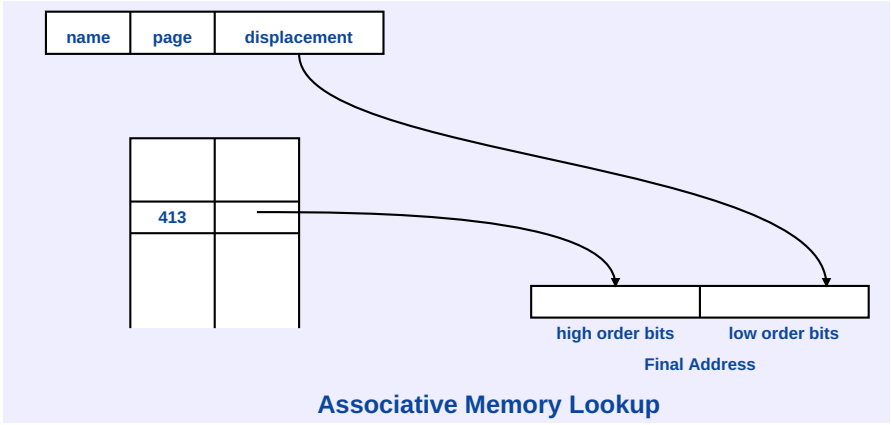
As it is pseudo-random in nature, secondary memory is handled in a similar manner. The scale is larger, the time slower, the intervals between garbage collection longer.

6.a.ii. Paged Segments

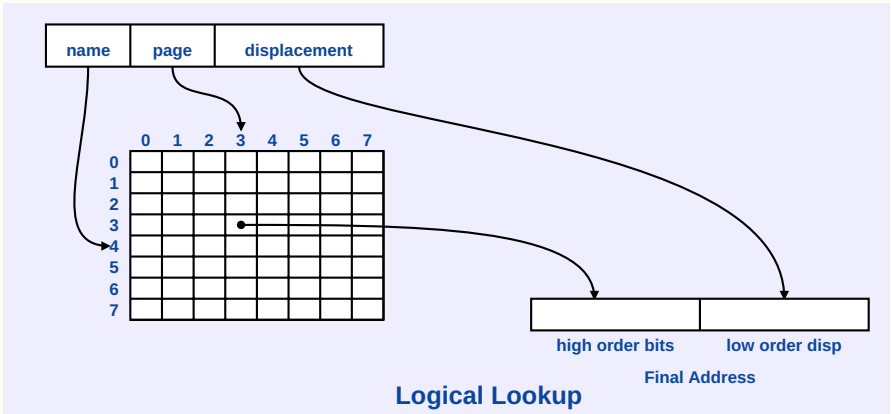
At this point paging should be considered. With paging, the logical entity (variable length segments) will be made up of one or more pages of some fixed size. Paging has some advantages in that reclamation of garbage and transfer to secondary storage is made easier. The disadvantages, however, may outweigh the advantages, for the address path is more complicated, requiring a table lookup on the page name, which consists of the segment name and the page portion of the displacement.

Paging and the Name/Address Transformation

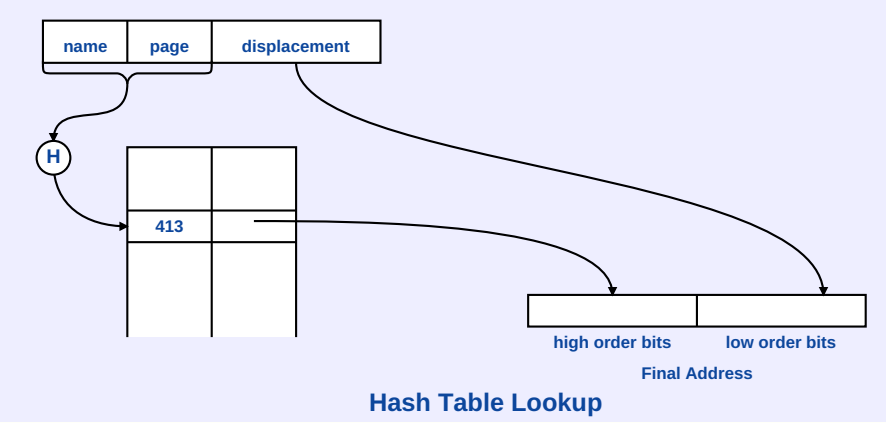
The conversion from the rather large compound page name to the location where it resides in core can be handled by a *soft* hash table, as in the previous section.



Associative Memory Lookup

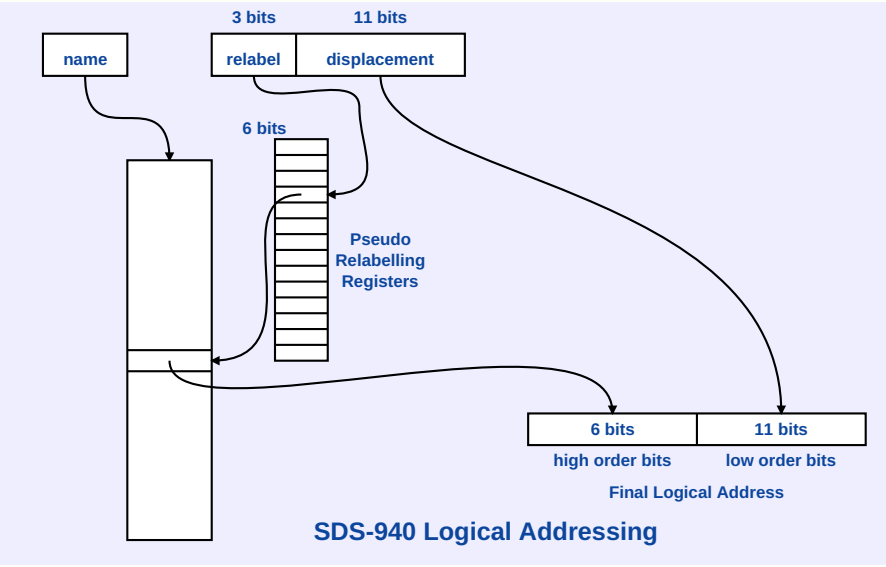


Logical Lookup



Hash Table Lookup

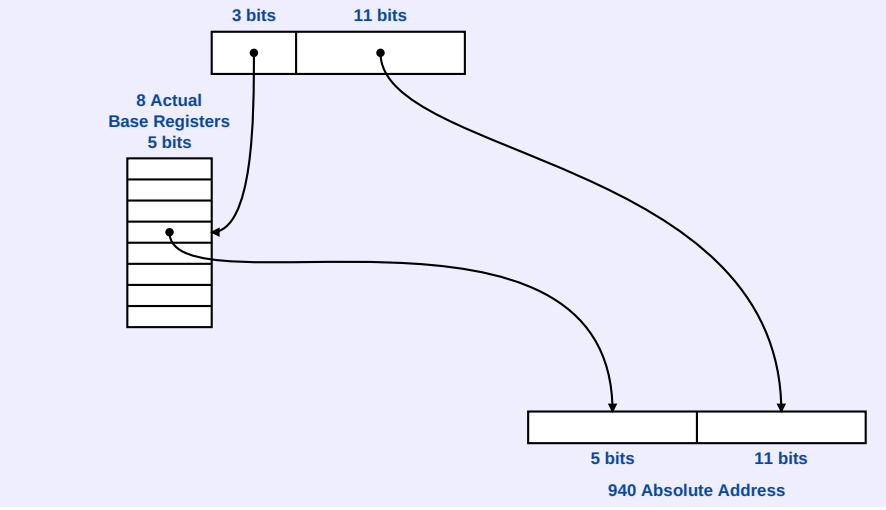
A more expensive solution is to use a hardware table which performs all possible lookups simultaneously. This is called an associative or content-addressed memory. It acts like a set of base registers which are addressed by name, rather than an index number. The transformation is effected in essentially zero time, so is at least twice as fast as the hash table method. The expense involved for this type of hardware depends on the number of entries in the table and, hence, on the number of pages in core, which in turn depends on the page size. These issues will be covered in more detail later.



SDS-940 Logical Addressing

An ingenious variant of segmented address was used by the Project Genie group at Berkeley during the creation of the machine that was later to be known as the SDS-940 (45). Since the machine address was 14 bits, it was possible to access only 16 K words directly. Yet the desire was to allow each user to think that he was working with a 128 K initial machine. This apparent bottleneck provided by inadequate hardware was brilliantly reversed into one of the better schemes for managing segment storage.

The user was allowed to have 128 K of virtual storage in his segment. The name field of the address is provided by the system and is thus global to these considerations. The page field is *not* provided by the small address and instead is selected by indirect indexing through an eight-entry pseudo-relabeling table. The six-bit number found there indexes the 64-entry program memory table, which contains the logical high order bits of the page address. Notice that only 16 K may be referenced at any one time, yet all the pages in the PMT can be available. The pseudo-relabeling table could more aptly be called the *pseudo-base register* of the user segment.

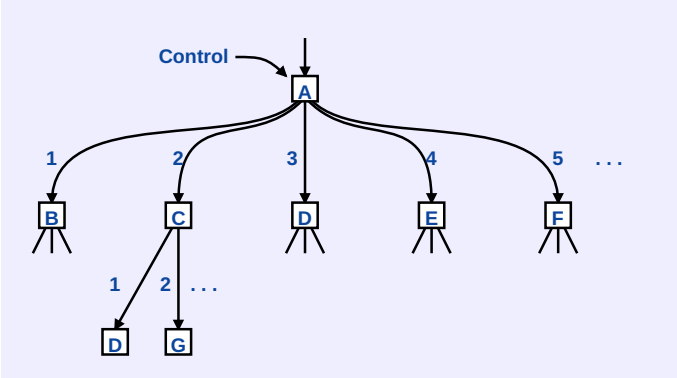


When a user process is *activated* on the 940, those pages referenced in the pseudo-relabeling table that are currently in core storage are entered in the corresponding base registers. Pages in the PRT which are not currently in core are marked by entering zeros into the base registers. Now the user's 3-bit indirect index may be used to directly select the high-order bits of the 16-bit absolute address. Since the *physical* core available on the 940 is 64 K and each user may reference a maximum of 16 K at any one time, it can be seen that at least 4 users may reside in storage at once. As will be elaborated in succeeding sections, these constraints lead to maximizing interaction and minimizing thrashing.

6.a.iii. Multi-level Schemes

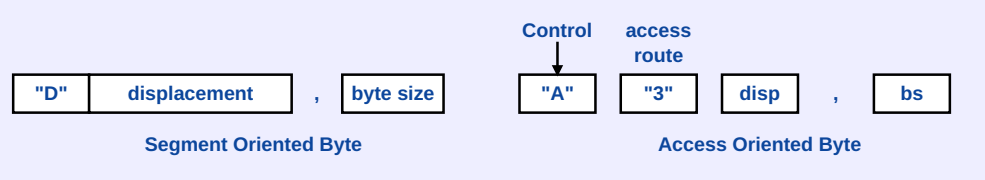
The 940 system requires each user to have one segment, the name of which is supplied by the monitor and is beyond his reach, thus allowing small addresses at the expense of affording protection from overwrite only between each user. The programmer himself is free to clobber his own 128 K. Some measure of safety is provided by the fact that he can reach only one-eighth of his total pages at any given time.

The method of Evans and LeClerc (39) offers the use of multiple names for segments, but still aspires to the small address. In this case, the page field is complete and the name field is the one that is abbreviated through an indexed lookup. This is possible when a hierarchical structure is used as context for supplying name fields.

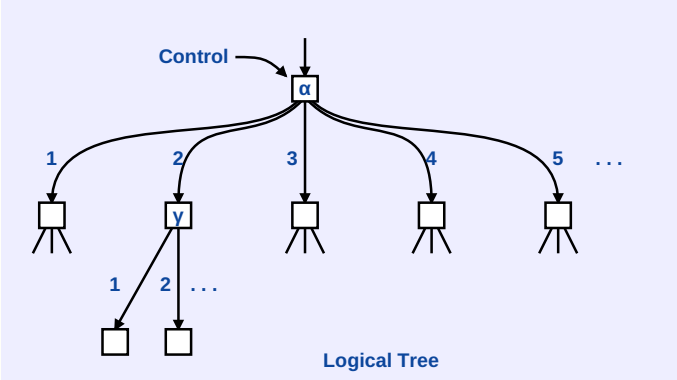


Hierarchy of Segments

A tree of segments is created by binding them together with access paths represented on the drawing as numbered arrows. The segment named *A* may access *B* only through path *1*. *D* may be accessed through path *3* of *A* or through path *1* of *C*, etc. If control can only reside at one node of the tree at one time, then it is clear that any field of segment *D* may be described in:



The advantages of this method are obvious. There will, in general, be many less access routes from any position of the tree than there are total segments in the system. Therefore, the access-oriented byte address will be much shorter than the segment-oriented address.

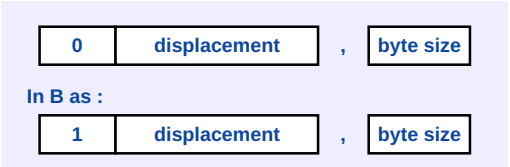


Since the determination of a path depends solely on the position of *control* and the path number, the actual segment names in the boxes may be submerged in favor of making the structure a *pure* tree. It wasn't quite clear in the previous drawing, since some of the nodes had the same name. Each set of binding may now be expressly a vector which contains the dominant node as the *zeroth* entry.

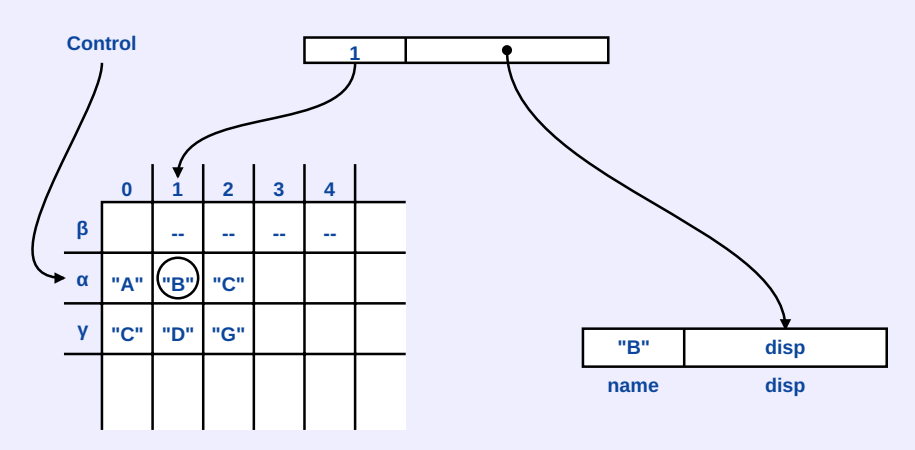
It may be expressed as

α: ("A", "B", "C", "D", "E", "F")

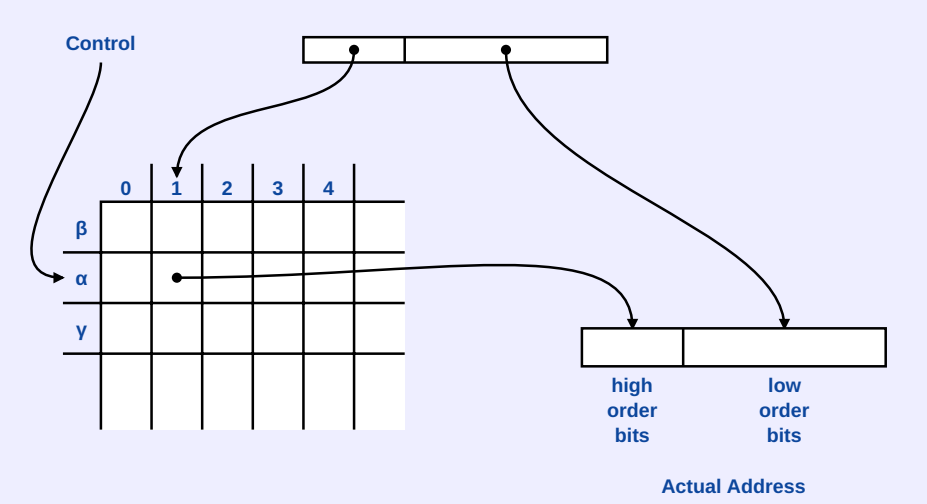
If *control* points to α, then the bytes in *A* may be expressed as:



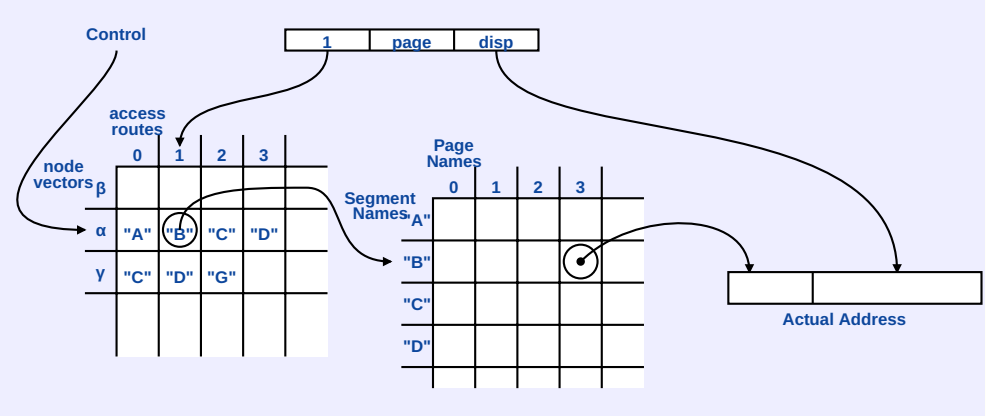
In order to restore the full names of the segment, it is only necessary to maintain a table of the node vectors; e.g.,



This table lookup brings the discussion back to the original question of segments. It can naturally be combined with the segment name/absolute address lookup to generate the absolute address of the base of the segment in core:



The previous two examples also apply if the displacement field allows paging.

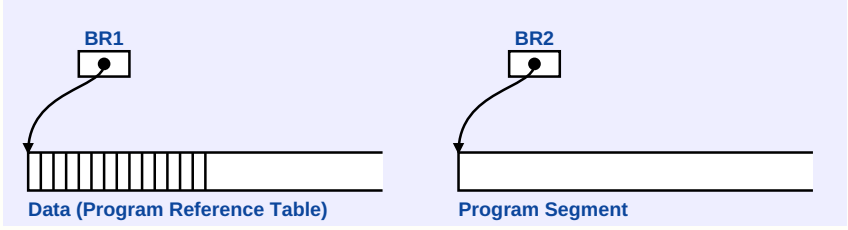


It is easy to see that the previously mentioned table lookup methods will work with this scheme if the *name* fields are sufficiently widened.

Descriptor-Protected Segmentation

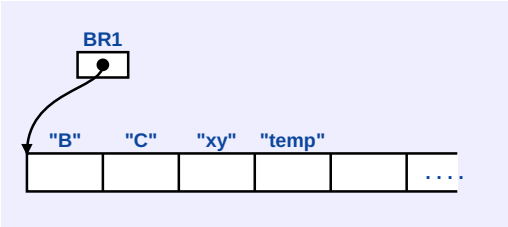
Another segmentation scheme, probably the first ever devised, was introduced years earlier on the B-5000 and B-5500. It is stated in different terms, yet can be made to look strikingly similar to that of Evans and LeClerc.

Program segments and data segments are distinct in the B-5000. All addressing is relative to the beginning of the segments and, since those locations are retained in separate base registers, it is impossible for a program segnment to overwrite itself.

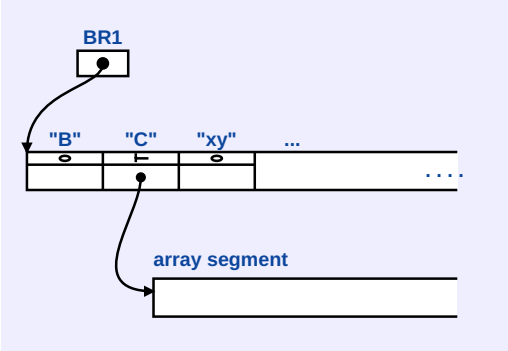


B-5000 Segments

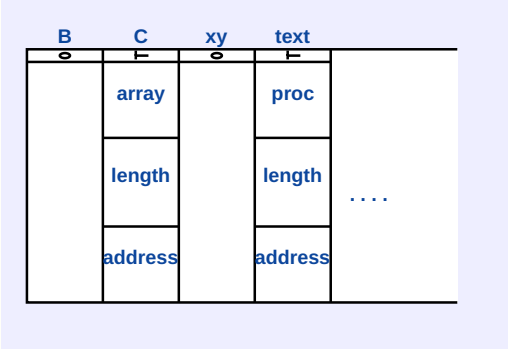
During compilation, a one-to-one correspondence is set up between the named variables (and constants) in the symbolic program and the linear arrangement of slots in the data segment.



Thus, *A* would be identified with an offset of 1, *B* would correspond with an offset of 2, etc. In order for indexing of the base register BR1 to be done reasonably, the slot size in the data segment should be uniform - in this case, it was chosen to be 48 bits. This is large enough for an integer or floating point number, etc.; but, suppose *C* was intended to be an array? The answer chosen by the Burroughs designers marked a significant advance in the philosophy of design. One of the 48 bits (called the flag bits) of the slot is reserved for marking the other 47 as a *number* or a *pointer*. Thus an array named *C* would have a pointer (called an array *descriptor*) in the slot corresponding to *C*.

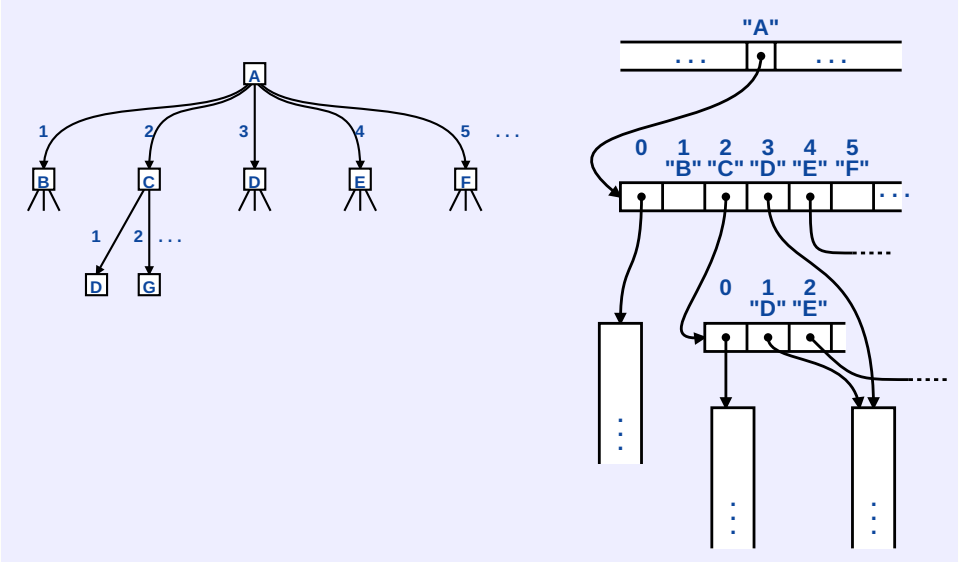


Now when an access is made to *C*, the contents of slot *2* are examined. The flag bit is on, which means the rest of the bits contain information about *C*, including the fact that it is an array, its location and its length. The length information may be compared against the prospective index to the array to prevent any possibility of an access being made outside of the array bounds.



Program segments (procedures) may be pointed to also. An access to temp would involve a *call* or *activation* of the segment referred to.

Now the correspondence between the B-5000 addressing and the Evans-LeClerc scheme is seen very clearly. Indeed, it provides a reasonable implementation of the more abstract discussion in the previous section.



A Correspondence

The segment pointed to by *A* is now revealed as the process stack (and, hence, stateword) of the program segment pointed to by BR2. The *data* segment pointed to by BR1 may be thought of as the parameter of program segment *CC* for this event. Note that the zeroth parameter points to CC. There may be other data segments pointing to CC, also representing concurrent instances of activation. It would be belaboring the point to push this analogy any further: it is too much of an over-simplification, as it now stands.

The important point to note is that the B-5000 system can be imposed over a one-level segmenting system to produce an effect like that

of the Evans-LeClerc structured scheme without adding any additional mechanism. It is this concept which affords the FLEX Machine great protection at low cost.

6.b. Secondary Memory

6.b.i. Swapping

Because core memory is quite expensive, it does not require very much to account for more than half of the entire cost of a machine, particularly if the machine is small. Also, there is traditionally never enough of it, no matter how much money is expended.

However, although a program may use vast amounts of temporary and permanent storage over its run-time life, it typically does not require an instantaneous amount that is larger than available core when examined at the 10's of milliseconds range. This phenomenon (when true) allows the computing capability of a system to be greatly extended by adding a vastly cheaper, vastly slower secondary memory without much degradation in performance. If segments that will not be used for a while can be *swapped out* while segments to be used in the near future are *swapped in* (this done as the computation proceeds on currently used segments), then performance of the system will not be degraded and the effective memory cycle time of the combined primary and secondary storage systems will be very near that of the primary memory.

The catch, of course, lies in determining just which segment *will* be used next and which ones *won't* be needed. The *solutions* that are presented are based on two different ways of looking at secondary storage. The first assumes that core storage is being augmented by buffing on the disk. The second assumes that storage is actually on the disk and being brought into core for execution.

Criteria for Swapping

How is a segment picked for transfer to secondary storage for the purpose of creating garbage? No swapping algorithm has been shown to be really satisfactory. The one presented here will work quite well, and following the FLEX philosophy, requires no bookkeeping on each memory access.

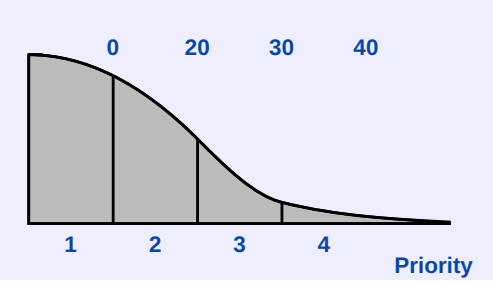
The influence of the compiler extends directly down to the lowest level of the machine and provides information, that is not commonly available on other machines. Some of this information has to do with an insight into the use to which each segment will be put which may be partially derived from mapping conventions and process use.

The volume of segments mapped as 8-bit bytes (text) will tend to be high - yet use is limited by storage and display restrictions. One might hope that these segments will migrate to secondary storage in a fairly rapid manner.

General data segments have a somewhat higher priority - yet they are clearly the next level to be thrown out.

Process descriptions and process stacks certainly have a higher need to remain in primary storage in order to sustain a rich amount of process activity.

The real-time processes, the event list, the segment table, and other system entities need to maintain residency in primary storage all of the time. Therefore, they should never be swapped.



These priorities may be expressed as a weighted conditional probability or as the number of standard deviations on a normal curve.

The swapping scheme now works as follows: A random number is selected, just as in segment creation. This is hashed to locate a slot in the segment table and thus, eventually, a segment. The type field is examined for priority and a question is asked whose answer is weighted towards that priority. For the normal curve weighting scheme, the probability of a yes answer to the question: *Should this segment be swapped?* is:

(area of a number)/(total area of a curve)

If the segment was text, then it would have a 67% probability of being swapped and a corresponding 33% chance of staying in. A system segment (having priority 4) will have a zero chance of being swapped.

Suppose now that a segment is in heavy use and is swapped. Then it will come right back in, but the chance of its being swapped again is now quite small, since a random selection is used. Conversely, a segment in little use will simply remain swapped.

The end result is that all segments in primary storage are scrutinized uniformly and those that are active tend to remain, while those that are not will tend toward secondary storage.

Another Method

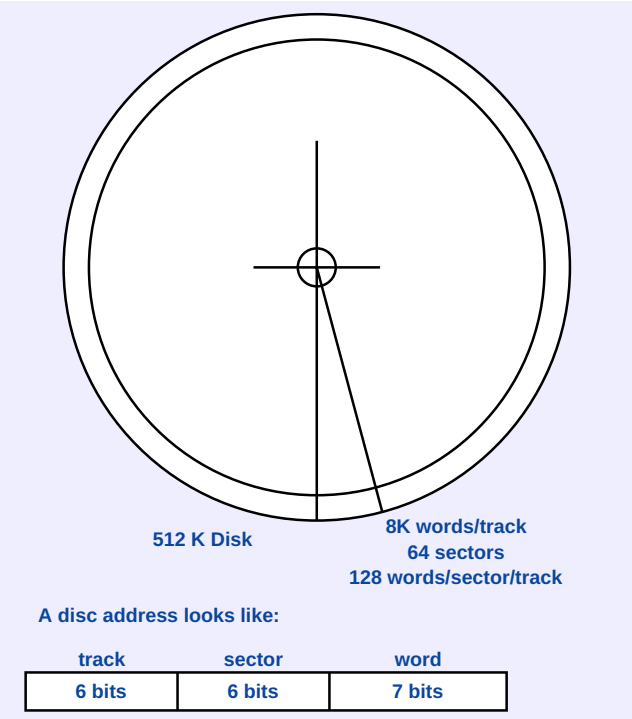
The philosophy here follows the very successful algorithm of the Berkeley System. A bit table is used to mark pages which haven't been written into. This means their copy on the disk is current and that they may be freely written into when a scratch page is needed. A bit that is set indicates that the page in the disk is not current. An *ubiquitous garbage collector* process continually runs around the system, throwing those pages out and turning the bits *off*.

Statistics from the 940 show that only 5-10% of the time is it necessary to forceably eject a page in order to make room for one coming in.

6.b.ii. The Disk as a Serial "Associative" Memory

Latency

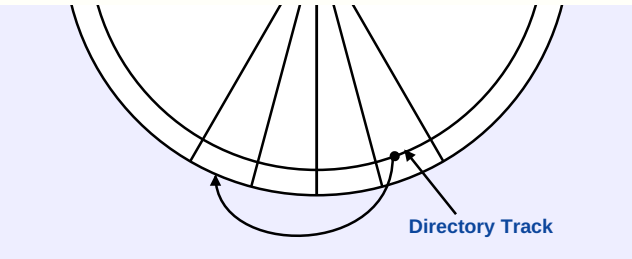
Periodic storage (disk or drum) has a word transfer rate that often closely approaches core memory (4-8 ms). The access time may be measured in milliseconds because the data may be 180° away from the read/write heads at the time an access is desired. The average access time to a datum is one-half the period or typically 17 ms. For this discussion, it will be assumed that the disk is organized as shown in the drawing.



Usually addressing is not possible to the word; the sector may be as close as one can get before all of it must be drawn into core for scrutiny. These considerations are not germane to the present discussion and so will be ignored for the moment.

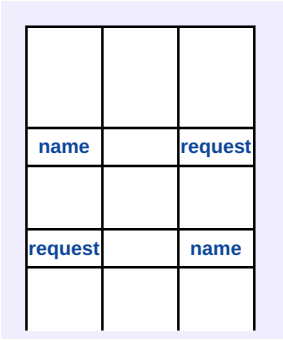
The disk may hold many segments (or pages of segments) and then the directory for its contents is very large and is a bother to keep in core. This organization scheme uses special hardware in an attempt to minimize the inherent latency of the disk, as well as to insulate the processor as much as possible from the memory system.

One of the 64 tracks will be *dedicated* to the disk dictionary and will have a special organization which depends on the physical location of the needed data.



The dictionary entries will be phased several sectors ahead of the actual repository for the segments so that advance warning is given of approaching sectors.

Instead of *first in first out* queues for handling I/O requests (which depends heavily on the time sequence of the request which may totally be out of phase with the disk) a hashed association table is maintained for all disk transactions.



It is set-like in nature and content-addressed. Each directory item is read by the disk hardware, hashed, and loaded up in the table to see if there are any input or output demands. If there are, they may take place immediately, since the disk has now rotated into position. The items are then removed from the hash table. *Output* transactions are also hashed by *request*, so that when blank areas are signaled in the dictionary, segments that need to be written out are transferred without waiting for their actual slot (it may not exist) to come around.

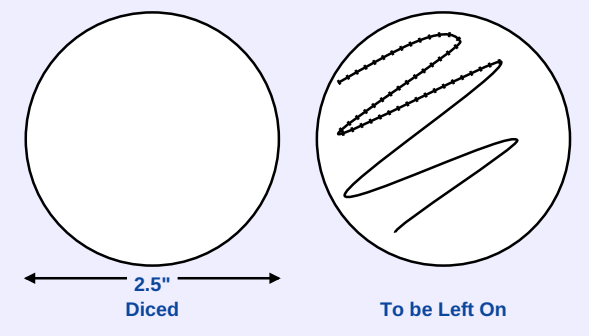
Now the dictionary needs to be updated. This request *is* sorted by dictionary position so that updates are carried out automatically the next time around.

It is interesting that this scheme performs best when there is a reasonable *load* on the disk. If so, there will be (on the average) several requests for each succeeding sector and traffic will be rather constant to and from the machine at an average access to a segment at 1/32 latency or about 1/2 ms.

6.c. An Associatively Mapped LSI Memory

In the beginning was the Word all right (general laughter) ... but it wasn't a fixed number of bits. -- R.S. Barton Software Engineering

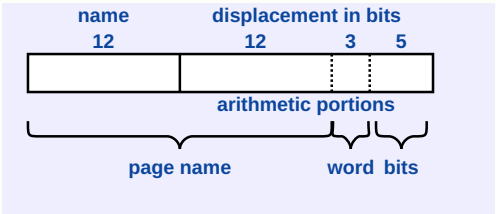
This is an attempt to solve many of the previous problems by *fiat*: to simply ignore them. Large Scale Integration provides a very provocative vehicle for trying to improve memory design. The cost lies not in the fabrication (as with core) but with yield, which is the percentage of chips which survive the traumas of evaporation, testing, dicing, and mounting. The basic material is a silicon wafer about 2-1/2" in diameter. On it can be placed about 300 chips using the conservative geometries of today.



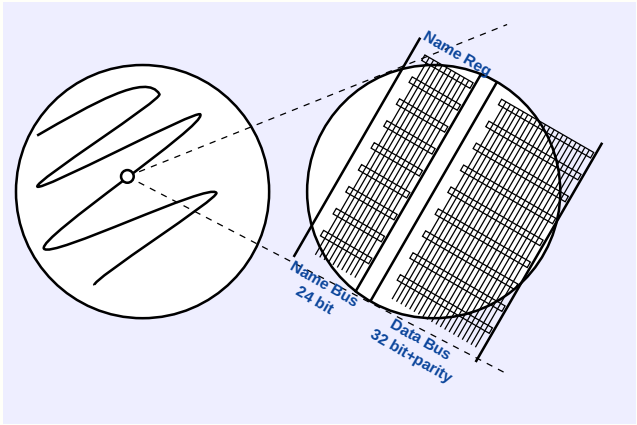
By means of stencil-like *masks* and various evaporative techniques, a pattern of *doped* metals can be formed at the site. Now the fun begins. First each chip is tested, while on the wafer. About 50% survive. Those that do are *diced* out of the wafer and mounted in a ceramic substrate eventually to be placed in a *can1*. Only about 10% out of the 50% (or 5%) survive. Those good chips have suddenly become expensive.

This mapping scheme partially depends on not dicing the wafer but instead leaves the chips on the wafer, thus hoping that there will be 150 rather than 15 good elements. Consider again the general segmented address:

Define fields as follows:



This allows 4096 segments of 32,768 32-bit words or of 2^{20} bits. The pages are rather small, having only 8 32-bit words or 256 bits. The key to this design is that the LSI chip cannot only possibly store data, but it is just as easy to create an active element, as seen on the drawing.



A Typical Chip Enlarged

All of the chips are strung together by a second layer of metalization bus. The name register in every chip is set to zero when the memory is blank. Several lines down the side transmit commands to all of the chips. They are:

001 Who is zero?
010 Name yourself *name*
011 Who is *name*?
100 Load word. 0...7
101 Store word 0...7
110 Make yourself zero

When a segment is created, nothing happens until a store is attempted. The memory, seeking to find the page to store into asks *who is (24-bit page name)?* If there is no response, it means a page must be created. Now the call goes out: *Who is zero?* The first chip to react closes off the command lines, preventing other chips who are also called zero from answering. Now the new name is passed down and then the data. The sequence is:

```
Who is       [54  |  000] ?
Who is       [ 0  |  0] ?
Name yourself [54  |  000]
Store word    [3]
```

Now when the data is to be retrieved, the commands are:

```
Who is       [54  |  000]  ?
Load word     [3]
```

Suppose the segment is destroyed. Then a masked search would be helpful, only looking at the first 12 bits, so:

```
Who is       [54  |  ###]  ?
Name yourself [0   |  0]
```

would deallocate the segment and return the chip to*free storage*.

Practical Considerations

There are about 1200 words on a wafer if 1/2 survive. (It doesn't natter which ones do, since they are found by name, not by location). This would allow 128 K to be packed into a 2-1/2" cube. Or one million words in a 5" cube. Unfortunately, there are some heating problems that would tend to require a more distributed packaging.

Now as to the cost, current estimates have it at about 3-5 cents per bit, which is competitive with core. But the 2-1/2" cube would cost about $200,000.00 which is quite a bit for a personal machine.

However, this kind of approach is the wave of the future, and there is a good chance that prices will come down into the .1 cent per bit level, where they should be.

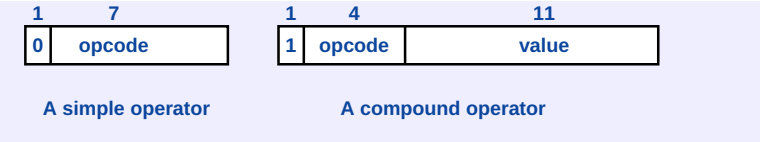
7. Algorithmic Processing

The Bird of Time has but a little way to Fly, and Lo! The Bird is on the wing. -- 0. Khayyam

a. Stack-Oriented Schemes

a. The Process Description

A process description is a segment that may be executed by the hardware of the FLEX machine. There are basically three kinds of entities contained in this segment: simple operators, compound operators, and numeric and string literals.



The first bit in each case determines the length to be either 8 bits or 16 bits. All the FLEX operators are compiled as simple operators. The names, requests for values, jumps, etc. are compound operators.

The compound operator

op code	value	Meaning
value call	stack index	find a value, put it on top of stack
name call	stack index	create a descriptor, put it on top of stack
Jump true	displacement	Jump if top of stack true
Jump false	displacement	Jump if top of stack false
Jump now	displacement	Jump unconditionally
New	number of var	create space
par	number of parameters	to match actual with formal parameters
commas	vector size	create a new segment
literal	integer	create a stack number from*integer*
label	rel address	used for operand for *goto*'s

The coding is very compact. To add two numbers, it takes:

```
40 bits if no operands in stack:     vc | 'a' | vc | 'b' | +
24 bits if 1 operand in stack:       vc | 'a' | +
 8 bits if both operands in stack:   +
```

This is more respectable than most single address machines by a fair amount.

```
1 2345   67890123456          1 2345678   9....40
1 lit                    0 num
```

Numeric Literals

Numbers come in various shapes and sizes. Integers from 0-2047 can be handled by the short form. Larger numbers are handled by longer forms.

Forms

The compiler arranges the operators in polish postfix form. Execution is done serially except for jumps. Declarations and intermediate results are stored in the process stack which has the following logical form:

Because the static textual form of FLEX is both properly nested and deterministic, the compiler is able to predict where in the stack the declarations will fall with absolute certainly. This means all that is needed to represent a variable name in the code is an index from the bottom of the stack.

Declarations

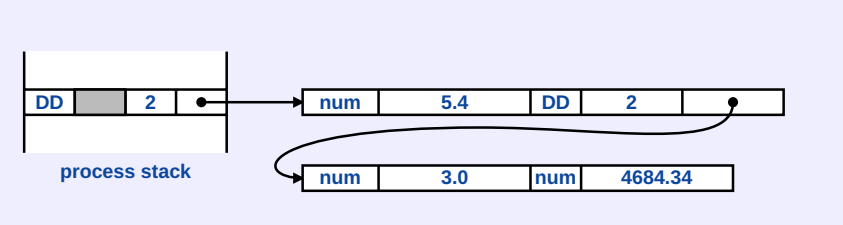
Each block has its own declaration section consisting of a slot for each variable and is topped off by a pointer to the previous block pointer. This mechanism is used by the execution of an end which carries a deallocation of the declared variables. As the stack is collapsed, temporarily data segments that are pointed to by descriptors are also deallocated by checking to see how many variables are pointing to it. When the count reaches zero, the segment is declared to be garbage.

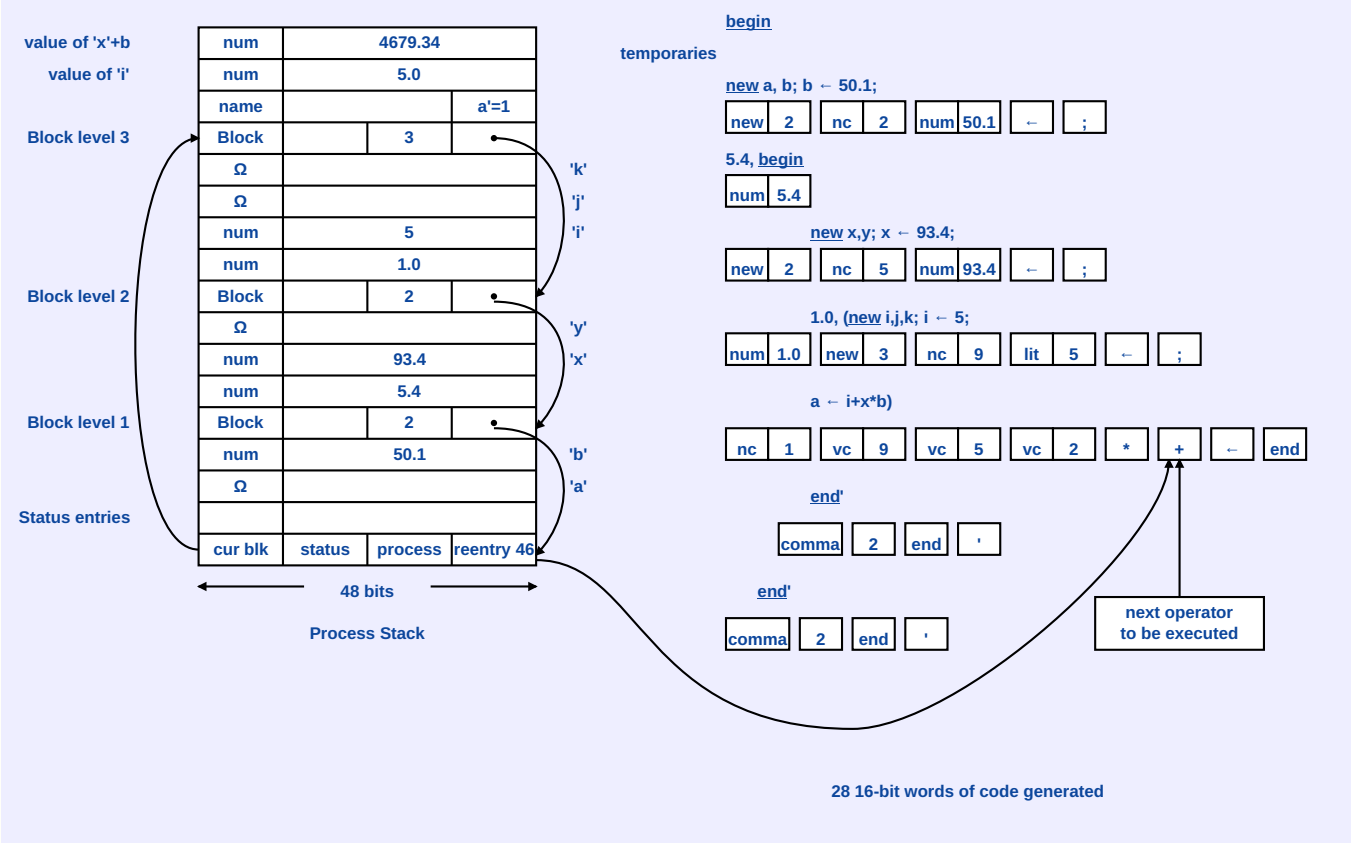
In the statement *x ← 93.4;* the name of x is referenced by a

| name call | 5 |

because x was allocated in the 5th location of the stack from the bottom.

The sample program creates a list structure with the following topology:



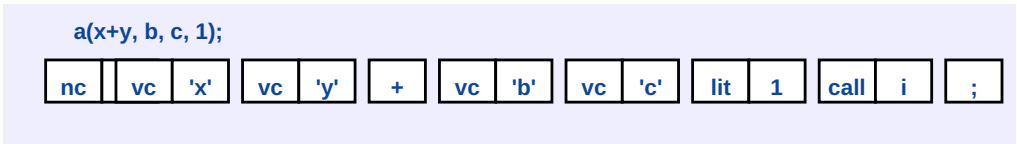


Commas

A *comma* operator creates a new segment from the vector in the stack and replaces it with a data descriptor. This has been done several times in the drawing.

Processes

A procedure call (or process activation) is realized in a similar manner.



The actual parameter list is built up in the stack like any other vector. *[nc | 'a']* creates the status information. *[call]* cuts loose the vector from the parent process stack (creating a new segment), enters an event notice for this process, and *passivates* the parent process. When a *[call]* is executed, the parent process will be *passive* until the called process *reactivates* it by executing a *return* or *leave*.

If the *[call]* were to be replaced by an *[act]* indicating that a parallel process is to be created, then the sequence of actions is the same except that the parent process is *not passivated*. A *return* from the new process will terminate the new process while a *leave* will simply

passivate it until the next time around.

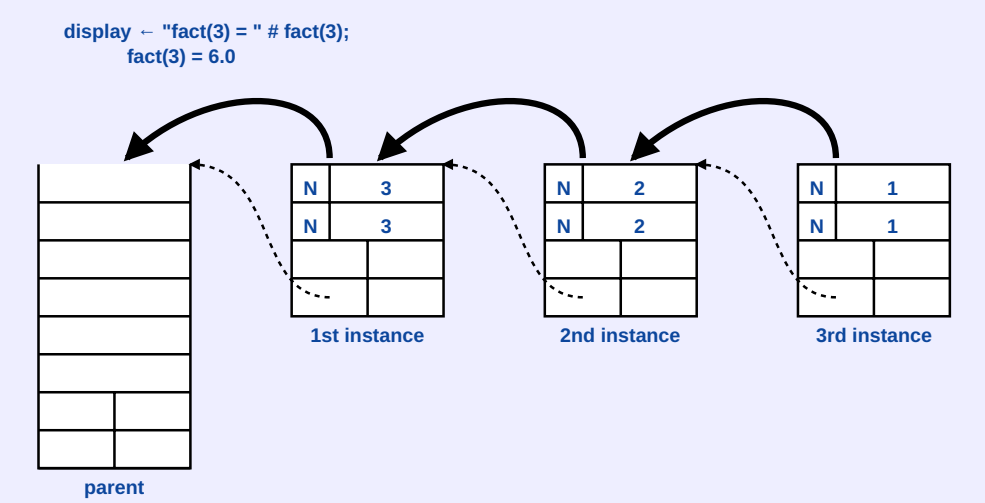
Recursion

Recursion is easy in the FLEX environment. Each parent process simply creates a new child which is linked to the parent by the returns. The *[call]* operator is used so that just once instance of the recursion is active at any one time.

Following is a subroutine which calls itself for determining the factorial of a number.

```
fact ← 'new a.  if a = 1 then 1 else a * fact (a -1)';
```

This creates the following structures when activated by the next statement



All returns from a call as well as reactivating the parent process also transfer the top element in the stack from the returned process to the top element in the stack of the parent process. This is the way results are passed back when a procedure is used as a function.

8. The Display Processor

The Moving Finger writes; and having writ
Moves on: nor all thy Piety nor Wit
Shall lure it back to cancel half a line
Nor all thy Tears wash out a word of it.
-- 0. Khayyam

This processor is a concurrently operating computer which shares memory with the rest of the system. Because of the lack of persistence of a spot on a quickly eraseable Cathode Ray Tube, the image that is drawn by the beam of electrodes needs to be constantly refreshed. This is done by re-executing the display program 30-60 times a second. There are storage tubes which can retain an image for up to 15 minutes without refreshing, but unfortunately the erase time is about one-half seconds to completely wipe off the picture. This disallows dynamic displays of any complexity or interest.

The rapidity of the refresh rate implies that the display program should be as compact as possible. It is also very loosely coupled to the semantics of the user program which only executes one assignment to the display, rather than many. The user will most likely use the associations to structure his display geometries:

```
endl•line_1 ≡ X
end2•line_1 ≡ Y
   ....
   Show ← line1
```

The display processor will not wish to run through the association in order to display *line1* for each refresh. Rather, a display program will be *compiled* from the associative structure *once* and then used many times until a change is needed.

It is unfortunate that this must be done, since it brings to rise all of the inherent problems of carrying information in several forms connected by an (essentially) one-way transformation. Much of the current wave of design research on display processors has been related to keeping the amount of necessary compilation to a minimum. The *matrix multiplier/clipping divide* display of Sproul and Sutherland (48) is a case in point. The display program for this mechanism is essentially a subroutined, high-precision (18-bit), three-coordinate representation of the edges of objects - i.e., something which may also be used by the programmer as his geometric model of the space. By a combination of brute force (three million multiplication/second) and cleverness (windowing in homogeneous coordinates), the processor is able to dynamically transform the display program into a two-dimensional perspective projection from any *eyepoint* in the 18-bit space at the rate of 10 µs per line, which is comfortably faster than most displays can draw lines.

The fact that the cost of this box is many times more than the entire FLEX machine somewhat precludes its use as the FLEX display processor. However, that is no reason to entirely rule out the use of special-purpose hardware to perform some of the necessary transformations on the programmer's logical structure. The tablet will also be discussed in this section, since most of the problems which arise from display-list compilation have to do with the difficulty of deciding where in the logical structure *pointing* refers when the only referent is a *space* display list.

The Virtual Screen

Instead of providing a large virtual three-dimensional space, as with the Sutherland display, the FLEX machine compromises by offering

a large two-dimensional (14-bit) *screen* on which displays may be *tacked* like notices on a bulletin board.

There are a number of ways to intuitively appreciate the resolution that is possible with the 16384 × 16384-point virtual screen. Assuming 100 points to the linear inch, the relative size of the screen would be about 13.5 × 13.5 *feet*, or room for a very large, very detailed drawing. Displaying text: 640,000 characters, or 12,000 80-character lines, or 42 *booksize* pages of text, may be mapped onto *one* virtual screen.

Any number of virtual screens may be created and used. Because of the size, scope and number of screens possible, it should not be surprising that the internal representation of strings of characters is *exactly* that needed to form a display list. In other words, character strings need not be converted in order to be displayed; they *are* their own display list.

Program segments can now be entirely identified with their displayed representation. The business of finding and showing an area of text to edit becomes merely searching for a pattern and adjusting the window into the text correspondingly. For debugging, the interpreter may be slowed down and its progress through the structure marked with a visible cursor to aid understanding of a procedure's elaboration.

Windows and Viewports

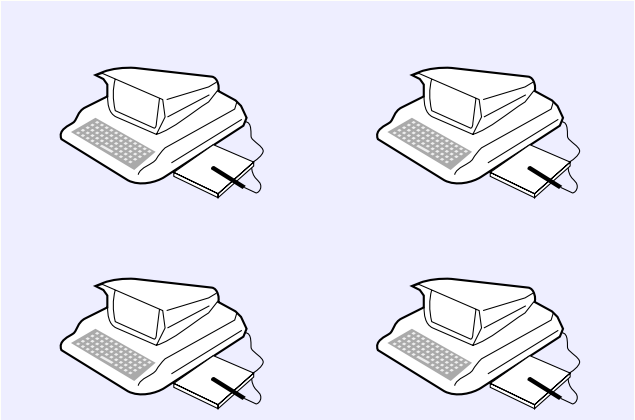
In order to effectively use virtual screens on an actual viewing surface of 1024 × 1024 points, windows and viewports must be used.

The window is described by its position and extent on the virtual screen and thus requires three 14-bit numbers: P, X, Y.

Windowing

The viewport is similar, except that only 10-bit numbers need be used for Q, x1, y1. The transformation is very straightforward, except that lines which are not entirely in the window must be clipped or eliminated so that the viewport on the actual screen will contain only the desired lines. There is no restriction on the number *or position* of the window and viewports.

Four windows into the same area:



Display instances are thus easily generated, as shown in the figure above.

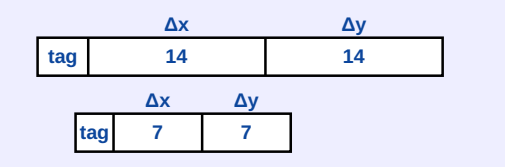
Zoom

Zoom is effected by holding the viewport coordinates constant while varying the window size. As the window gets bigger, relative to the

virtual screen, the displayed characters and lines appear to shrink in size. Too small characters and short lines first become points and then disappear entirely as the zoom continues. This is also effected by the clipping hardware, which will reject all lines which either are not visible or cannot be made visible by the actual display.

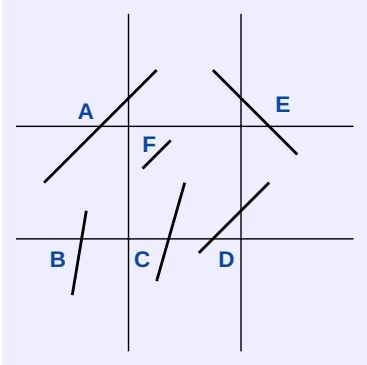
Lines

Visible lines are drawn by a digitally integrating stroke generator from a line descriptor which may be thought of as a call to the display. The generic line requires 14 bits of x and y in order to span the screen. A descriptor is provided for this. However, since many lines are considerably shorter, a short line descriptor is also provided, which allows a great compaction of data for the display list.



The Clipper

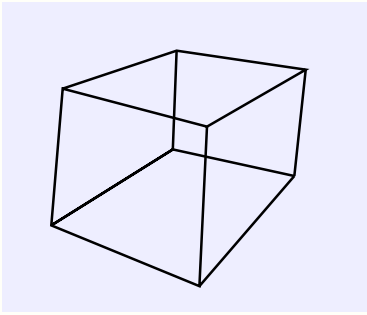
The clipping hardware uses the *mid-point algorithm* first suggested by D. Cohen (48) and independently discovered by Cheadle.



A *tic tac toe* board is made out of the window and coded, as shown. Lines such as B may be trivially rejected. Other lines (such as A and E) require more work. Since C and F have an end point in the window, at least part of them is visible. The fact that the other end of C is outside the window causes the action of the midpoint algorithm, which is simply a binary search conducted by shifting the coordinates of the line to effect a division. Eventually the edge of the window is found, and the coordinates of the visible portion of the line have been found.

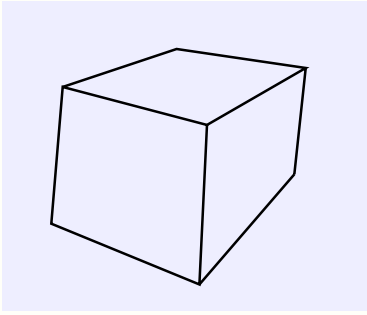
Perspective Transformations

Since the FLEX display can only handle two-dimensional information, a*soft* transformation must be applied to three-dimensional data so that it may be displayed. The methods of Sutherland are used (48); the program (a four × four matrix multiplier) is kept in read-only subroutine storage for easy reference.



Hidden Edges

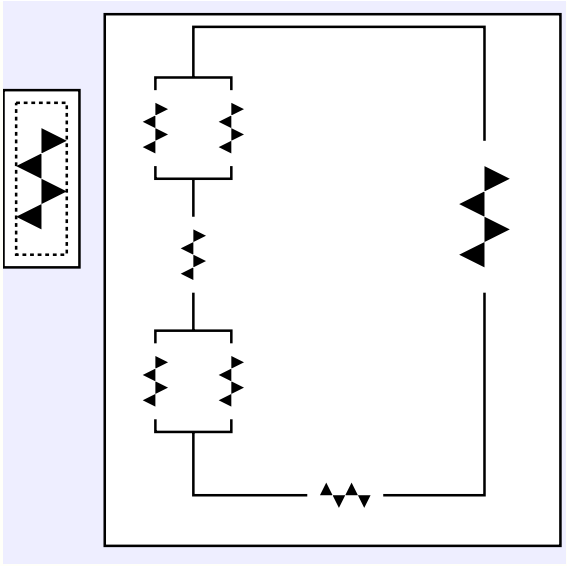
The basic algorithm for removing hidden edges, as discovered by Warnock (49) is discussed in Section 3.h. The basic test, that of a polygon with a rectangle, is just that which is done by the clipper hardware. It is not surprising that the clipper may be called as a coroutine in a way that is totally independent of the display in order to do segment-to-segment edge involvements.



Prototypes

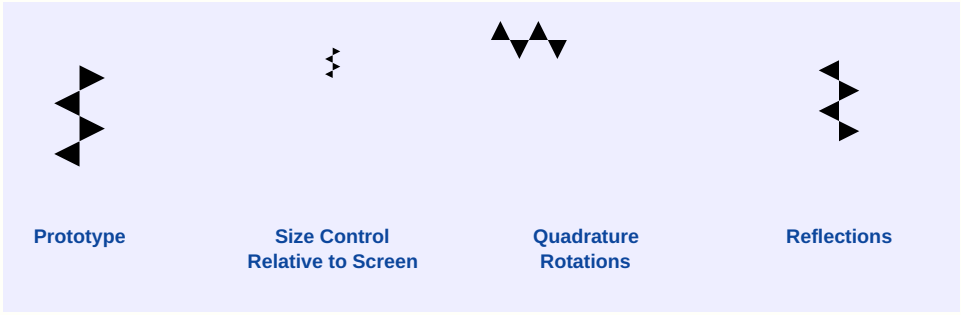
Subroutining allows many instances of an object to be displayed from one prototype routine, its position on the screen determined by global parameters applied to the *particular call* of the routine. For example, this allows one prototype resistor to be created and then used in many spots on the screen.

Seven Instances of Same Prototype Resistor:



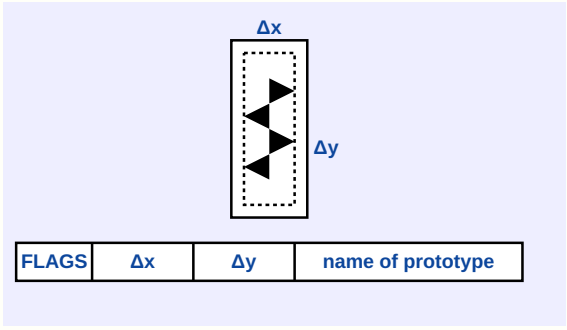
Other parameters may be applied globally. Rotation in ninety-degree increments (quadrature) and reflection about an axis are compromises for the more general cases of complete translation, rotation and reflection in x, y and z.

The size of the placed prototype may also be selected with relative magnifications in the range of up to 256 times the absolute size of the prototype.



Parameters Applied to Prototype Routines

The subroutining is controlled in a way that is very similar to the execution of deferred FLEX code. A *call* in the display list is represented as a description which points to the desired display procedure (which will itself be a display list, etc.). Return notices and applied parameters are handled exactly as in the FLEX language via stacks.

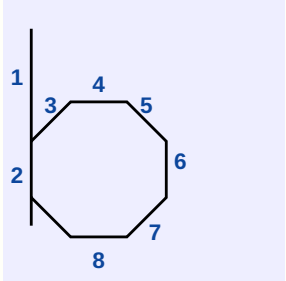


Intuitive Descriptor

Since the relative maximum displacement of a prototype routine can be calculated when it is formed, it is also possible to include these size parameters in the description itself. After the global parameters have been applied, an absolute x, and y may be computed. This information may be clipped before the prototype is called. If it is not involved with the current window, the call will not take place. This is particularly advantageous with large displays where most of the picture is not visible and also not able to fit into primary memory. The ability to clip prototypes before they are called will not force unnecessary secondary storage accesses to take place.

Characters

Textual representations are also prototypes, but are considered to be *intrinsic* to the display. They are drawn by a special set of micro-coded routines which specify a series of *hooks* and *strokes* to be drawn. For example, the letter *b* can be drawn as:



Or nine strokes or two hooks and one stroke. Coding hooks as a primitive vastly reduces the number of bits necessary to represent characters. 112 characters (the 96-character ASCII set plus special characters) has been coded in 512 12-bit words for implementation in ROM.

The picture below shows an example of text produced by the experimental FLEX display system (described later in this chapter). The image is a negative for clarity.

```
active 1519:12 WHOOPS!!!!!  Alan C. Kay 1
volume ←
  object:,pi,r,h;
  pi ← 3.14159265;

  Show "I can compute the volume of a sphere,
cylinder, cone or ""none""
  If you will indicate which please.";
check: Demand object as "object = ?";
       if object = "sphere"
        then [Demand r as "radius = ?";
           Show "volume ="  4/5 * pi r↑2]
        else if object = "cylinder"
              then
```

The character width is variable (like an IBM Executive typewriter) from 2 half-spaces to 5 half-spaces, which gives the illusion of type-set text. Digits all have the same width in order that columns of figures will line up.

Numerous *case* shifts are available, besides *upper* and *lower*.

Italic simply skews the hook generator a linear amount so that *vertical* lines are tilted to the right 15° to produce an italic effect.

Bold intensifies the characters so that they stand out from the rest.

Size chooses the height and width *relative to the virtual screen*. One of 16 sizes may be selected.

It should be pointed out that these case changes are entirely independent of one another and, hence, may be used simultaneously.

An *escape* character directs the hook generator to take its program from the volatile memory, which allows the FLEX programmer to devise new character sets which will then be drawn efficiently. Of course, the usual line drawing facilities of the display may also be used at the cost of more time and space. There is an additional advantage in using the programmer-defined characters. A font which is useful may be debugged in this fashion and then easily incorporated into ROM for permanent use.

Reserved Identifiers

In Algol, the word *begin* is considered to be a single character rather than *"begin"*. In FLEX, it is possible to actually identify *begin* with one of the 256 possible characters in the set.

The character generator simply needs to trap on (say) all characters whose *rank* is greater than 127. These may be found via a transfer vector in separate ROM in their string equivalent, possibly with an *italic* or *bold* case shift to draw attention to their different configuration.

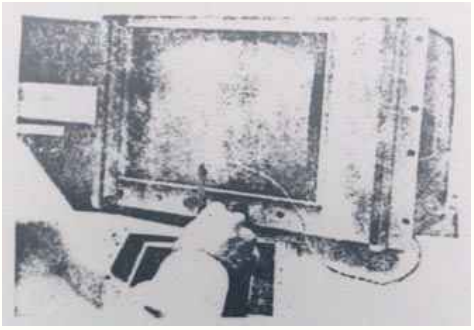
Pointing

The physical operation of the tablet will be sketched in the next section on the experimental display system. For this discussion, it is only necessary to say that at regular intervals a 10-bit *X* and *Y* and a two-bit *Z* are delivered by the tablet hardware into global registers accessible both by the display and by the FLEX processor.

It should be mentioned that the 10-bit *point-position* numbers have about 3% error due to the non-linearities and uncertainties in the tablet-measuring hardware. The tablet is not particularly useful for tracing because of this. However, the tablet is used for pointing, not tracing, so no problem arises.

The *Z* indicator is simply flip-flops which register whether the stylus is touching the tablet surface and, if so, how hard it is pressed down. This is a standard set-up.

A novel feature of the tablet is the ability to identify a *window* with the censor position. It may be any size, and will follow the stylus around on the virtual screen. The clipping hardware may be again called into play, this time to register tablet hits! The variability in size allows a coarseness of *grain* to be set, making it easy to point to gross things, yet allowing pinpoint accuracy when necessary. This is not easily done in less integrated systems.



The Experimental Display

An IBM 1130 (with disk) was used as the host computer for the experimental display system primarily because of its availability. The lab is shown. A home-made interface to the SAC channel of the 1130 connects the devices. The main display device is a Hewlett-Packard 1300-A electrostatic scope, particularly suited for this kind of work because of its low deflection voltage (100/inch) which may be driven directly from silicon output transistors. A high resolution Hewlett-Packard scope (seen to the right) was also connected for taking transparencies under controlled conditions.

The 1130 keyboard (particularly unsuitable in almost all ways) was used for input.

Sense switches were used both for additional case shifts and for editing.

The following sequence shows the creation of the cover picture on the HP 1300A and its subsequent rendering (in a reversed image) by

the high precision scope.

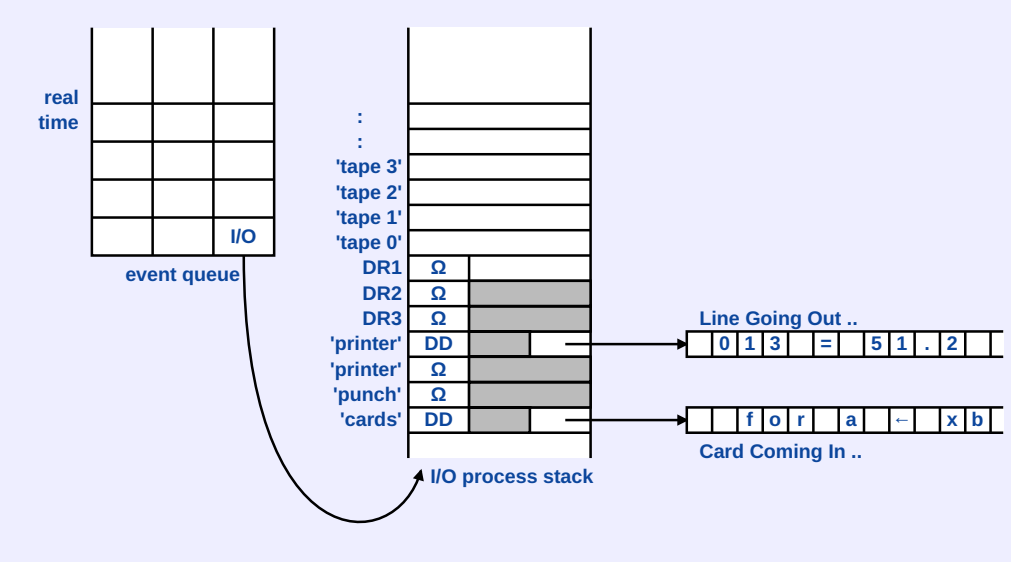
The various photos of the experimental display are almost completely unreadable in the reduced A5 version of the thesis.

9. Input and Output

I/O Devices

I/O in Flex does not require any special statements; it is handled as a generalization of the assignment statement. How is this realized in actuality?

Each device has a reserved variable name associated with it and, hence, there also exists a slot in a process stack somewhere in the system that is also associated with this name. This process stack is the I/O process stack and is pointed to by the *I/O* event request in the real-time section of the process control queue.



I/O Control

An I/O interrupt uses the number of the device that caused it as an index into the I/O process stack. In the slot associated with the device there is either an or a data descriptor pointing to a segment which contains data to go out or data coming in. The figure shows an execution of the statement:

printer 2 ← "al3 = " # al3# " b22=#b22;

Since no format of any kind has been specified, a FLEX free-format is assumed. As the concatenations are executed, a scratch segment is created in the usual way to contain the generated string. When the " " is executed it first looks at the description for the storand. It is marked as a *temporary* and therefore only a name transfer is needed rather than a copy. This is done into the slot in the I/O process stack which is now marked active.

Some time in the near future the I/O system will deliver an interrupt saying that printer 2 is free. The *printer_2* device number (in this case: 4) finds the data description in the stack indicating that something has to go out. This is set up and that data is squirted out on the channel coax. The *printer 2* slot is now marked empty and life goes on as before.

If the above FLEX statement were in a loop for printing out consecutively generated values of al3 and b22, it might very well be possible that another *printer 2* assignment might be made before the previous line was transferred out. The answer is simple. If the *printer 2* slot does not contain an Ω, then the current process is *passivated* until the next time around the round-robin. By then the line may or may not have gone out and the algorithm is continued. Eventually the line will be printed and the current *printer 2* statement will be executed. Naturally, more than one line may be output in one statement - a vector of lines may be assigned. The above just says that an I/O statement to a unit must be physically realized before another to the same unit may be made.

Input is similar. While assignments to the printer have been going on, the card reader had been active. An interrupt occurred saying that it had something to deliver. A data descriptor was found showing a read request (one is always there for pure input devices) and a card image was delivered to a newly-created scratch segment. Sometime later a FLEX statement might be executed:

new card ← format_1 (cards);

Formats in FLEX are simply functions or user-declared unary operators which take a string as an argument and deliver a string as a result. The card image (being a temporary) is renamed as the first parameter of format_1 and is thereupon operated on.

Two-Way Devices

These are handled in a similar manner to the printer and punch except that both read-request and write-request data descriptors are used. Also, it is important to note that, since all I/O devices are just variables in the system, they may be mapped and then selected on as the data enters or leaves the machine. Suppose only the first five words are needed from a tape record, then the following statement might be appropriate:

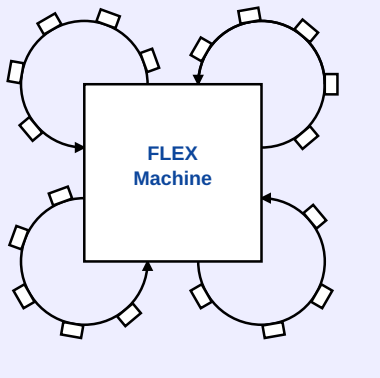
buffer ← tape 3 [0 to 4];

Only the first five words will be read in and transferred.

10. Multi-machine Configurations

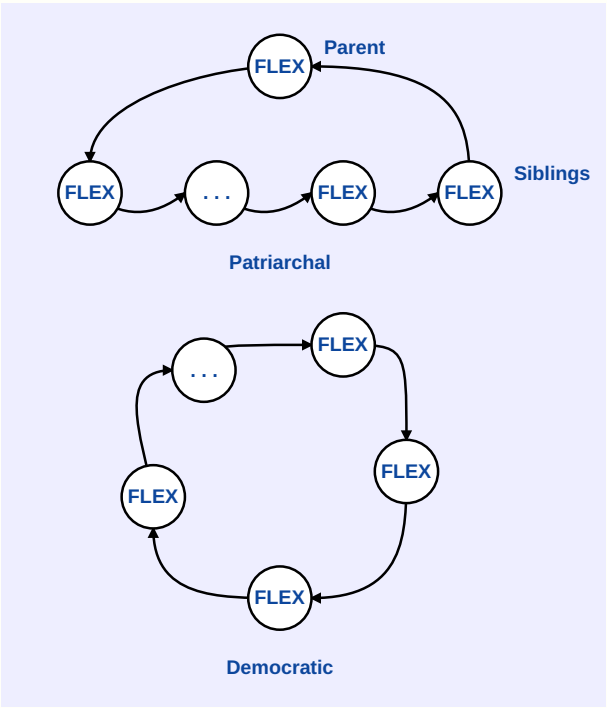
Because of the process-oriented nature of FLEX, it is natural to wonder how well [- aggregates of FLEX machines, will work together. The key to success lies in the absorption of the higher level information into a global representation which may easily be distributed over a group of machines operating concurrently.

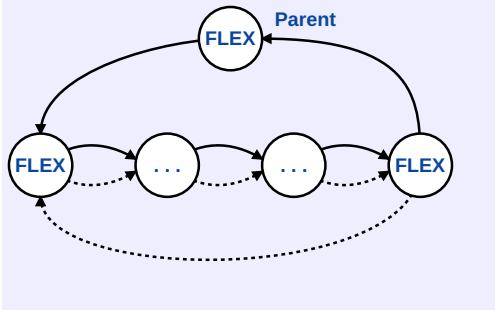
The basic I/O loops on the machine consist of pipeline priority serial paths. For now, the I/O configuration can be modeled by several directional loops belonging to each machine.



Any number of devices may be *hung* on a loop. The interface to the loop is a box which contains the priority algorithm for that device. Requests are sent serially through the *pipe*. Each device decides whether its need is greater than the preceding signal. If it is, it transmits its name and squelches previous devices. This is a rather convenient method for dealing with priority interrupts, since they are handled externally to (and concurrently with) the host computer.

FLEX machines themselves may be members of a loop. Usually a determination is made as to the relative subservience of a member machine. That is, does it belong to the loop set as a relatively passive entity, or is it an initiator and handler of priority requests?





The drawings show a number of possible configurations with like machines: the third figure shows a combination of *Patriarchal* and *democratic* relationships. The siblings have an equal footing as to communications with each other, but are subservient to the parent on another I/O loop.

Process Allocation

There are as many allocation possibilities as there are I/O hookups. (Some more useful than others). One would be to consider *siblings* as a *resource pool* of processors for realizing a *single* event queue.

One sibling loop would now look like a more powerful single machine. The scheduler would be the parent machine. When a process is included on its scheduling queue, a *when* becomes active and causes a request for elaboration to be propogated to the siblings. While these actions are taking place, the siblings are also following their generic behavior. As a sibling machine becomes passive, its name is distributed to each of its neighbors by following the loop. A job is handled in the same way as I/O requests by having its event notice traverse the sibling loop looking for an *empty*. When one is found, the process is *activated* and run.

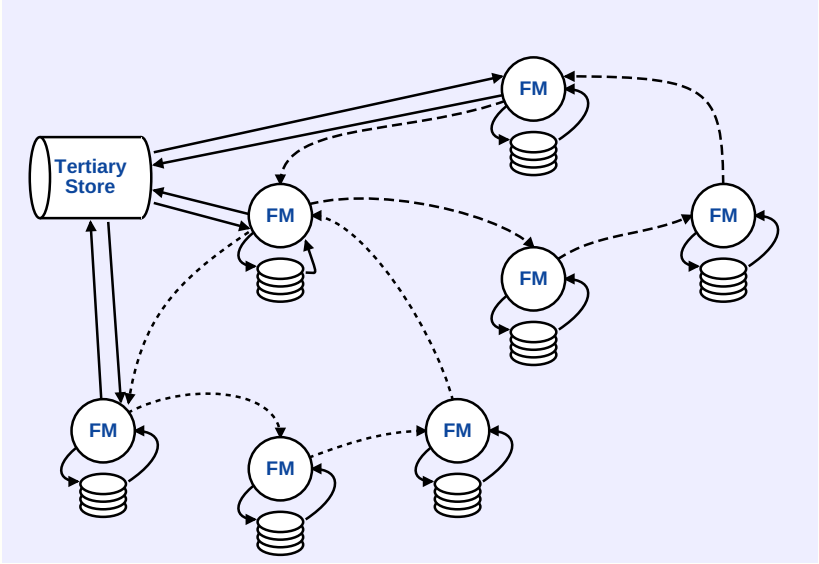
An important point to notice is that there is a one-for-one correspondence between the user's semantics and the machine-aggregate's pragmatics. This is due to the nature of FLEX as a process control language, which allows these modelings to take place without a *compiler-like* transformation being applied.

Configurations

a. Hierarchal

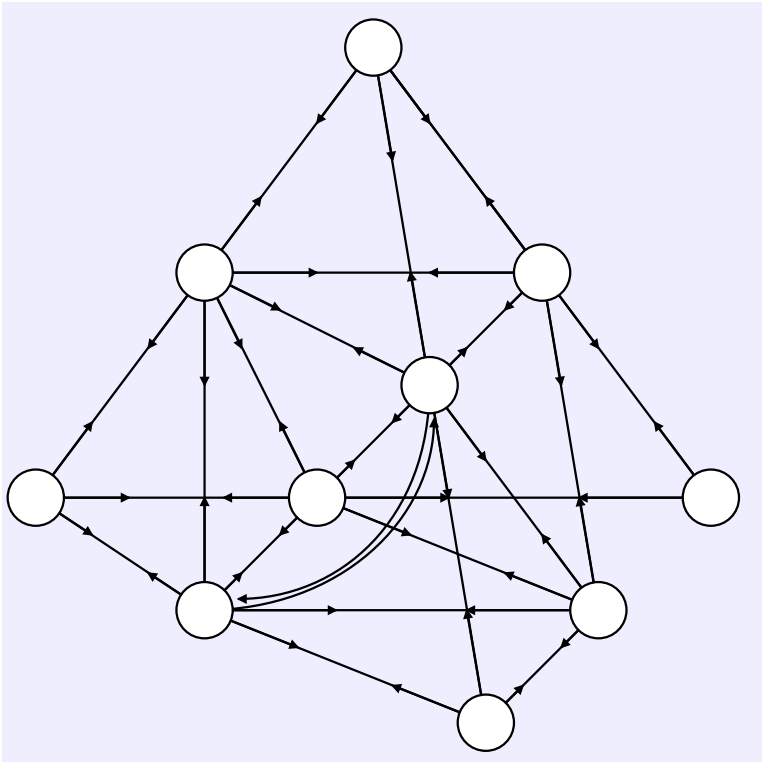
A reasonable configuration for an aggregate of machines is to have the following disposition of loops:

1. Secondary Storage
2. Tertiary Storage and Low Traffic.
3. *Son* FLEX machines
4. *Sibling* FLEX machines.



This configuration provides a hierarchical arrangement which is well suited to many forms of organization. Besides having the ability to communicate through each other's I/O loops, the current contexts may also be accessed through the tertiary store.

Redundant



The figure shows a tetrahedral arrangement of FLEX processors. Each machine is subservient to one or more parents, as in the previous examples. The sibling relationships are even more incestuous. There are clearly two other siblings for each parent-sibling pair, and control of communication becomes an interesting problem.

If the allocation of tasks proceeds from the top of the structure down through the levels, it is easy to see that the queuing problems encountered in the one-level-pool example are considerably lessened. This is because there are many more alternate (or redundant) paths through which requests may travel when searching for an elaborator.

Another convenience is that gaps in the structure (caused by machines not working, etc.) do not destroy the operative ability of the system.

Other mappings may be made on the structure. The *outside* machines clearly have more communicative *power* than the rest. Just looking at those hookups reveals the simple hierarchical structure of the previous example.

This example was inspired by (61).

Further Groupings

The crystalline structures found in nature, as well as human designs (such as Buckminster Fuller's and Ronald Resch's efforts), suggest

new configurations with (so far) unknown properties. This is an area that has hardly been touched in Computer Science because of the (up until now) prohibitive hardware costs.

IV. Implementations to Date

Two FLEX compilers (one with segments; the other without) were programmed in ALGOL on the UNIVAC 1108 system in February, 1968. Attempts were made to interface these programs with an interactive editor (also in ALGOL) and the graphics system (61). The failures were partially due to the inadequacies of ALGOL as a real-time and process language in general, and in particular, to the very real defects of the UNIVAC version of ALGOL-60.

The graphics display routines, character generator and editor ran for a year on an IBM 1130 computer with a *home-brew* interface. Unfortunately, the 1130 was straining to just act as a glorified display buffer, and none of the algorithmic routines were implemented.

In January, 1969, Dr. William Newman decided to use FLEX as the control language for the new PDP10 - PDP9 - UNIVAC display system. Henri Gouraud did the FLEX compiler in TREE-META (12) on the SRI (Englebart) SDS-940. Dr. Newman and Scott Bennion programmed a mini-FLEX interpreter for the PDP9, and it is able to serve four time-shared graphics users (58).

This system is now being brought up on the PDP10 to act in tandem with the programs on the PDP9. A complete implementation of FLEX on the PDP10 is being contemplated, although the monitor is not quite right as to process creation and memory mapping.

That, unfortunately, is the crux upon which all implementations revolve. Hardware implementations need to be performed in order to get a reasonable initial memory system.

Several *mini computers* which allow user-defined micro code have been carefully studied as a possible object for hardware implementation. The Interdata Model IV seems to be much more suitable than the others that were examined. However (like the rest), it suffers because its micro organization is strongly prejudiced towards the IBM 360-like organization of the Model IV. Its main feature is that it has many hardware registers, which cover a multitude of other sins.

FLEX will only find its real home in the machine environment for which it was designed. This machine is currently being designed, and it is hoped that it will eventually be built.

V. Appendix A. The FLEX Handbook

The following Appendix contains *excerpts* from the latest version of *THE FLEX HANDBOOK*. The scope of the Appendix is sufficient for understanding the contents of the thesis as applied to FLEX.

THE FLEX HANDBOOK

Written by Alan C. Kay

Department of Computer science
University of Utah
Salt Lake City,
Utah
August, 1969 Technical Report

1. Introduction

FLEX is an acronym for *FL*exible *EX*tendable Language, and was developed primarily to be a base language which could be implemented directly in the hardware of a small machine. Because of its extreme simplicity and generality, it has also been adopted as a means for expressing algorithms on the Digital PDP-9 and PDP-10 computers.

FLEX has many distinguished ancestors which influenced its final form. The non-conversational but highly powerful algorithmic languages ALGOL-60 (18) and its generalizations (EULER (15) and MUTANT (19)) lend structure and manipulative ability to the design. LISP (27) has set a standard for flexibility and simplicity. The conversational language JOSS (21) and its off-spring CAL (22) have greatly contributed to the philosophy of the user-FLEX dialogue. Other influences have been LEAP (36), APL (37), the Lamda-Calculus models of Landin (59), as well as the Project Genie Documents (45, 7,26, 52) and the Syntax Macros of Leavenworth (60).

- It is important to make the point that FLEX is *not* a *feature* language (e.g., PL/I, etc.), nor was it *tacked together* from bits and pieces. Rather, it is a simple, concise and powerful means of expressing algorithms. As such, it forms a core from which needed abilities may be built at the discretion (and control) of the user. Because the hardware has been optimized towards dynamic extensibility, the cost of generality is low, while the gains in flexibility are enormous.
- This handbook is designed to contain a complete summary of those things the FLUX user will want to know when using the system. The exposition is totally oriented towards using the FLEX machine. *Grubby details* involving the use of FLEX in other environments are covered (to date) in an Appendix. The format of the handbook gratefully follows the high standard (62) set by the *JOSSmeisters* of RAND, who not only paved the rocky road of interactive computing, but also created delightful explanations of their work. The following is a (hopefully) complete guide to FLEX as used by the expert, giving examples of conversations with the system and reasons for the actions and reactions of both FLEX and the adept user.
- The system comprising FLEX, which is a combination of a *hard* machine and *firm* algorithms, may be thought of as a literate active agent which will carry out the user's commands.
- Although FLEX cannot be fragmented into a *basic FLEX* and an *advanced FLEX* (as with PL/1), the core of the language may be learned by a novice in several minutes, allowing him to do basic computations and simple programs in his first session at the machine.
- After several sessions, the user will become fluent enough to need and use the combinatorial abilities that comprise the complete language, which allow rather comprehensive manipulation of symbols and constructs. Because FLEX is not a feature language, there are no real divisions between the arithmetic powers (with which most initiates are familiar) and the non-arithmetic elements (for which most novices soon discover a need) in the FLEX environment.
- This handbook, therefore, does not exhibit a division of FLEX, although the order of presentation is that of *first* elements, later giving way to more sophisticated ideas, capped off by a section advising those who would be experts, rather than casual users.
- The formal definition of the language and its interpretation is presented (in itself) in the Appendix. This is necessary for a complete understanding of FLEX, but is delayed until a rather complete intuitive grasp is acquired from reading the longer, informal description which comprises the greater part of the handbook.
- For those who are taking their maiden voyage in interactive computing, we wish you as much fun and satisfaction as we have had - and remember: *when in doubt, try it out!*

2. Joining FLEX

2.1 The FLEX Console

- FLEX may be reached through something as prosaic as a teletype (Sec Appendix I) or by one or another of graphically-oriented consoles especially made for conducting the FLEX dialogue.
- We will assume for the rest of this handbook that the object device is the *portable* FLEX machine so constructed as to fit the dimensions of an attache case. It is entirely self-contained, although it may be externally connected to a variety of devices in numerous ways (See Section 11).

2.2 The Keyboard

- The layout is exactly compatible with the TTY-37 keyboard, with the addition of a second control shift for the extra characters which are available on the FLEX character generator.
- Appendix 1 contains a discussion of the various character equivalences for all common input devices.

2.3 The Five-Finger Keyboards

- The two keyboards are symmetric (the thumbs control the same signal), so that either (or both) hands may be used.
- One or the other is commonly used in conjunction with the stylus for highly interactive tasks.
- The use of the auxiliary input devices is discussed in greater detail in the section on interaction.

2.4 The Stylus and Tablet

- The tablet slides out from under the keyboard. It may be separated completely from the FLEX machine, if necessary.
- When the stylus is applied to the surface of the tablet, the relative x,y position is measured and input to the FLEX machine, where it may be analyzed.
- A tracking dot on the screen will mimic the position of the stylus.
- A micro-switch on the stylus measures two levels of*z*. Usually the first level is just used to track, the second level to *point*.

2.5 Initiating the Dialogue

- Turn the machine to *on* with your *user key*.
- If you have never been admitted to the FLEX console, you will also need a *protection key*, which may be obtained frora the person responsible for the machine. Turn this to *admit*.
- The machine should respond with the herald: *FLEX Active* and the current time.
- Type in a name which identifies the context you wish to set up. A symbol should appear to indicate everything is O.K.

2.6 Terminating the Dialogue

- *leave FLEX;* from wherever you are will log you off and save all of your context under your log-in name. The next time you log in under that name, you will be placed exactly where you left.
- Try it a few times to get a feel for it.

2.7 If You Have Troubles

What to do when trouble hits depends greatly on where you are and what machine you are using. A space is provided for helpful telephone numbers.

INSTALLATION	PERSON	TELEPHONE NUMBER

3. General Points

3.1 The FLEX Dialogue

- The beginner may approach FLEX from the viewpoint of JOSS and CAL. As in those languages, he is directly connected to a translator-interpreter which will immediately act on his commands, thus forming a dialogue between FLEX and the user.
- To aid typing, the FLEX machine will attempt to anticipate the interactor's use of symbols (as in the SDS-940 (A.12)). When enough of a symbol has been typed for it to be uniquely identified, FLEX will supply the rest of the characters. This allows longer and more readable symbols to be used without putting a strain on the user. This ability, along with the extendable nature of FLEX, allows a natural lanpuage-like form of expression to be used, so that both the *conversation* and the programs are readable - a great help for novices and experts alike.
- Any text enclosed in ... will be ignored by FLEX, except for editing. This allows comments to be freely interspersed in the dialogue.

```
Show ← 1.2   ..."Show" is  the display   ...
  1.2
Show ← 1.6 * 2.9522/ (19.7 - 9.2) ... as a desk calculator... ;
  4.4985905
:xx1 ← 52 ...: defines a new local symbol ...;
Show ← (xx1 # 536.1, "This is text");
  52
  536.1
  This is text
Show ← (xx1 # 536.1 # "This is text);
  52536.1 This is text
Show ← (2 + 2 * x ; ... incorrect syntax ...
          ↑ What?
```

3.2 Initial Editing

- For now, anything that is visible may he edited.
- One of the conveniences of FLEX is that, like CAL, the user's previous statement is retained and may he thought to be *under the cursor*. This allows extensive editing to be done without requiring a great deal of retyping.
- The *previous* statement may be found in a number of ways. It may indeed be the previous line, or it may be selected by a variety of means, including the *Edit* function (See Section 8).
- Because the editing commands are part of FLEX, all of FLEX may be used to define edits on arbitrary streams of the user's text. However, it is convenient to also be able to edit directly at the keyboard *by eye*, so there are equivalent *invisible* commands to do this.
- This section will cover enough to get the user going. For a detailed description of editing, see Section 8.
- These options are conveniently selected with one movement of a hand.

```
  MOVE      DELETE      TO                SET

F(orward)   D(elete)      CHAR              WRITE OVER
B(ackward)  N(o)          WORD               INSERT
                    LINE
                    |  Character pattern |
```

3.3 Direct and Deferred Commands

- So far the user has experimented with direct commands and, indeed, these are truly the only kind that can be issued in FLEX.
- Although a great deal may be accomplished in the way of calculation by simply executing expressions, the user will most often have a problem that will require a series of commands in order to reach a solution. Also, he would like to save these steps in a form that is entirely controllable by him. To handle this, the notion of deferment is introduced.
- To *defer* execution of some interesting stream of text, the user merely *quotes* it by enclosing it in *" "*. This quoted form will act as an abbreviation of the contents. It may be named by assigning it to a newly created variable. Henceforth, it will replace the name whenever it is invoked.

```
: a ← 53;
: b ← 10;
: cc1 ←a + b;
: cc2 ← a + b ;
: cc2 ← "a + b";
: cc4 ← ... a + b ...;
                ↑ What?
a ← b ← 5;
Show ← (cc1, cc2, cc3, cc4);
  63
  10
  a + b
  Ω
```

- Although the quote will not he executed until later, FLEX will still look for obvious errors in form and will so inform the user.
- The text of the quote can be recovered for editing by applying the Edit function to the name. A copy of the text will be displayed as the *lost statement* for editing, etc.

3.4 Order of Program Execution

- FLEX programs are executed in serial order, except at decision points, which are constructed in a number of ways.
- The comprehensive decision statement is of the form: *if ### then ### else###*. The ###s may be *any* construct in the language. Other conditionals are *while* which repeats the *then* clause until a *false* is reached. Another is *wher* which does the *then whenever* the conditional clause *becomes* true. This allows an *interrupt* facility.
- Elaboration of quotes may also change the order of program execution but, except for possible side effects produced by the action of the quotes, it appears as though serial execution is taking place.
- These concepts will be handled in a more comprehensive manner in Sections 7 and 8.

```
:x ← :y ← 50;
Show ← if x<y then "x<y" else "x is not< y";
  x is not< y
while x ≤ y then [Show ← (x, y) ; x←x + 1] else  Show ← "done";
  50
  50
  done
when x>y then show ← "interrupt";
while x ≤ 48 to 52 then show ← x else show ← done";
  48
  49
  50
  51
  interrupt
  52
  done
```

3.5 The FLEX Environment

3.6 Scope of Names

- The novice may find it convenient to visualize his environment as a body of text, rather like that of a conventional program, with himself at the console as a *window* into this world.
- His direct commands may be thought of as the evaluation of a quote, so that anything he types in will be done.
- He may cause his own quotes to be executed, and he can *see* all of the names that are currently in use. He may also bring quotes back to his window, to be modified by editing.
- This is a useful model in view of what is possible with the full generality of the language.

3.7 Extendability

- The approaches to extending the form and effect of FLEX are many and varied, with different levels of sophistication required for complete appreciation of the concepts. For this reason, the more common devices are just mentioned in passing, with a complete exposition in Section 8.
- Adding new operators: Any quote may specify how it is to receive its parameters. The operator *max* may be defined in a number of ways, depending on the desired use.
- Changing the Syntax: The complete specification of the current language context is always global and accessible to the program. New contexts may be added - or the old environment may be completely replaced at the whim of the user.

```
...as a function ...
: max ← 'list: (a, b).  If a<b then b else a';
...as an infix operator ...
user op ← 'value:a "max" value:b.  If a<b then b else a';
...or,  more  simply   ...
user op ← ':a "max" :b.  If a<b then b else a';
...on an arbitrary list   ...
```

```
:max ← 'list:a.   While  :x ← 1  to lg[a]-1
do if a(x)>a(x +  1)
then a(x +  1) ← a(x) ';
```

3.8 Debugging

- No matter how careful a user may be, there will come times when his program will not work. Any system which does not provide tools for debugging is a poor one indeed, particularly in a conversational environment.
- By the very nature of its direct (deferred discourse) FLEX (as with the JOSS-like languages) provides very powerful debugging aids. The user may examine his variables at any time by assigning their values to *Show*. He may also test his programs step by step, by controlling the scope of execution. He may set sentinels that *lie in wait* for an error by using *when* statements.
- As will be seen in Section 8, the control paths to individual variables can be monitored in a very natural way. An even more powerful tool is to use the display as a dynamic window into the state of the program in order to get a gross picture of activity. Of course, the program can be run at any subjective speed to aid visualization.
- FLEX also allows the user to ride *piggyback* on the execution of his program with the ability to stop the flow and examine things when any number of present conditions are reached.
- Small programs tend to be easier to comprehend and debug. The extendable nature of FLEX allows simple abstractions to be made of complicated situations which may be then incorporated into the language (and the user's context) itself. Programs may always be short and readable (if the user desires to be concise).

3.9 Error Messages

- Because of the interactive nature of the FLEX machine and the loose and forgiving form of the syntax, the number and extent of error and diagnostic messages is spare. Most serious mistakes in logic are discovered during execution of a quote, rather than when it is bound to a name.
- Most syntactic errors are simply flagged with *" What?"*. The cursor is automatically positioned as near the error as can be determined and the edit case is entered for easy restructuring of the text.
- Essentially the same thing happens for a run-time error. The text of the quote is displayed with the cursor positioned near the error. What information there is about the error is displayed and the user is in *debug position*. That is, he is in a position so that the entire state of the program at that point is global to him. He may examine variables, etc. just as though he had been put into debug position and had a break in the flow of execution
- After the error has been found and corrected, he may choose to resume execution at that point, or to start over from scratch.

3.10 Saving Context

- Any routine may be exited so that its state is preserved on reentry by the *leave* operator.
- *leave*; will exit one level. Reentry is after the *;*. If the current level is *direct*, the user will be logged off with his context saved.
- *leave* name; will freeze all levels down to the quote called *name*. Hence,
- *leave* FLEX; will log the user off with all levels of execution frozen until his return to the machine.
- This operator completely obviates the necessity for *files* of any kind. *Files* are simply global variables that have been frozen by a *leave*.
- Note: Since FLEX is conversational, the novice is encouraged to *try things out*. It is much cheaper in the long run, all things considered. Only by experimentation will the beginner quickly learn to think in FLEX and thus will be well on the way to becoming an expert.

3.11 Syntactic Conventions

- The syntax and semantics of FLEX (in this informal exposition) will he described partly in FLEX itself and mostly in English.
- Allowable syntactic forms are described grammatically in a way that is reminiscent of BNF (and phrase structure grammars in general).
- Literals are described as text quotes - i.e., *"if"*, *"then"*. Forms that are built out of other forms are denoted by names (which are presumed to have been bound previously).

```
...syntactic description of a name
name ← 'letter *(letter digit)'
letter ← '"a" | "A" | "b" | "B" | ...| "Z" | "z"' ;
digit ← '"0" | "1" | "2" | ... | "9"'   ;
... the "*" indicates 0 to many repetitions of letter  ("or") digit ...
... using descriptor ...
gather ← 'name:a.  Show-← a
| Ω .Show  "not a name"';
gather fine
fine
gather 11.3
 not a name
```

4. Expressions

OPERANDS

4.1 Numbers

- ```
  Number ← '"." integer | integer ("." (integer | Ω)';
  integer ← 'digit *(digit)';
  digit ← '"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"';
  ```
- Without special provision, the following is true. Numbers are *exact* if their precision does not exceed *9 decimal digits*. Otherwise, they are inexact. The range of magnitude is 10^{-99} to 10^{99}. However, it is easy to redefine the number system to any desired precision.
- ∞ or *inf* is a number > any inexact number and is the result of dividing by zero.
  ```

- *true* and *false* are exactly equivalent with 1 and 0.
- Ω or *omg* stands for both an undefined value and the empty value. It is the result of an illegal operation and is the value of an unbound name.

```
:x ← 1.95678;
Show   (x, x↑x, x*x, x+x, x<x, x∧x);
    1.95678
      .
      .
    false
    true
```

4.2 Text Literals

- text literal ← """chars""*: | "<" chars ">"";
  chars ← ' *(char)';   ... 0 to many ...
  char   ← '"0" | "1" | "2" | ... | "A" | "a" | "B" | "b"   ... | "+" | "-" |...';

- The letter "O" is identical to the number "0". "A" is identical to the number "10". So a more comprehensive way of describing *digit* would be

  ```
  digit ←  '0 ≤char≤9';
  ```

- Characters are any visible display character, except $, which signifies the control shift. So $char is the control shift of that char, or $ integer, which is the binary coding for that char or $$, which is "$".

4.3 Names

- name ← 'letter *(letter digit)';
  letter ← ' "A" | "a" | ...| "Z" ';

- The content of a name is a set consisting of ordered pairs of the form attribute, value. Certain attributes such as "val" and "sym" are permanently attached to each name by FLEX, and the user is free to add his own attributes and values indefinitely.
- x ← 5 is the same as val.x ← 5 where <attribute.value> is <val. 5>. Mentioning the antecedent will return the value.

  ```
  Show ← val.x ;
     val.x = 5   ...or...
  Show ← x ;
     x = 5
  ```

- sym.x stands for the set of all symbols currently associated with x. Thus

  ```
  son.John ← Bill;
  father.John ← Henry;
  Show  att.John;
    John = {son,father}
  ```

- type.x will be "ident".

4.4 Lists

- List ← '("begin" | "[" |"(") Body ("end" | "]" | ")") """ Body """ ' ;
  Body ← 'Alternate *("|" Alternate) ';
  Alternate ← '(Star | Choice | Expression | "Ω")
      *(":"(name List_of_names))
      *("." write_aspect)';
  Star ← ' " " (range | Ω)"(" Body ")" ';
  Choice ← '"(" Body ")"';
  Literal ← '(text_literal | number)';
  Write_aspect ← '(value | Ω) *("," value)';

- This is the all-encompassing structured entity in FLEX. It has both a *read-aspect* and a *write-aspect*. It is a generalization of the *state.transition* of a Turing machine.
- The alternatives are tested against the current input stream. If a match is made, a *true* is propagated back through the structure and further testing is done. Any matched section of the input stream may be bound to a name which is considered to be local to the list and, hence, an attribute of it. This binding is denoted by a ":".
- The write aspect is entered via a ".". The list of expressions is evaluated and effectively replaces the list in the output stream.
- This somewhat confusing construction allows one concept to replace many that have been considered useful in ALGOL-like languages. Also, by adopting this form, many useful new constructs are made available with a minimum of punctuation.

- ...an arithmetic expression recognizer...
  Exp ← 'Term: a *("+"Term: b.a ←   a+b) ";".   Show ← a | .Show ← "error"   ';
  Term ← 'Factor: a *("*" Factor:b. a * b)';
  Factor ← '"(" Exp ")" | name | number';
  :x ← 15;
  Exp   1 + x * 5.5;
     83.5
  Exp 1 + + x * 5.5;
      error
  ... as "Wirth & Hoare" Records   ...
  ... define a generic hospital patient
  patient ← List: (name, age, weight, height, disease)

```
... produce instances of the patient record ...
Bob ← @patient["Bob", 18, 175, 62, "cancer"];
Charley ← @patient["Charley", 22, 178, 59, "embolism"];
Show ← (age. Bob, disease. Charley) ;
    18
    "embolism"
... as static parameters for a procedure.
max ← "List: a. while :x <= 1 to kg.a_1
                do if a (x) < a(x + 1)
                then a (x + 1) ← a (x)'
Show ← max [a, b, 5 + x↑ 54, -9];
     54094
```

4.5 Declarations

- Algol-like declarations are subsumed by the *read aspect"* of the list.
- In effect, the bindings allowed by the alternatives completely control the *parsing* and *evaluation* of the input parameter stream.The concept of *type* is also subsumed by this concept and is merely an attribute of the evaluated entity.

- ...normal procedure parameters, not worrying about "type" ...
  test ← 'List: (x, y, z). _____';
  ...binding to the argument list itself...
  test ← 'List: a. _____';
  ...constraining the input parameters...
  test ← '"(" value: x, string: y, List: z ")" . _____';
  ...getting the parameter text ...
  test test ← '"(" string: x ")". _____';

4.6 Value Elements

- value ← 'Expression *(";" Expression)';
- The ":" may be considered to destroy the value of the previous expression. It essentially *scans back* to the nearest comma or start of valued list.
- The value of the concatenated expressions is clearly that of the last expression.

4.7 Procedure Lists

- A procedure list is just a Body surrounded by """, """. Execution is prevented and a *reference to* the interior is formed.
- 'Body' is *exactly equivalent* to @[Body] where the "@" stands for *reference to.*The quotes are only used to improve readability.
- The *local* bindings that can occur in the alternates are considered to be the formal parameters of the procedure.

4.8 Parameters

- Parameters in the general FLEX sense are what is *left* in the input stream that is in the *range* of an elaborating list.

4.9 Reserved Names

- Reserved words ← '"begin" | "end" | "if" | "then" | "else" | "ceil" | "flr" | "sin" | "cos" | "atan" | "rand" | "prand" | "hash" | "exp" | "ln" |"syrt" | "any" | "while" | "when" | "to" | "by" | "do" | "array" | "field" | "name" | "val" | "leave" | ...etc ';

- These have the form of names but are considered to be semantically identical to *delimiters.* Many reserved words have an exact counterpart among the *delimiters.* For example, "begin" and "[", "end" and ")" are exactly identical - so are "∧" and "and", "∨ and "or".
- The FLEX user may consider them to be just global symbols which he can supersede at his pleasure.

4.10 Precedence of Operators

- *Unary* operators take only one operand: the right-hand one.
- A cascade of unary operators, viz: *un1 un2 un3 OPND* means: *(un1 (un2 (un3 OPND)))* the operations being performed from the inside out.
- All unary operators (including those defined by the user) have the same precedence: the highest.
- Binary operators take two operands: both right and left-hand.
- The situation *OPND BOP1 OPND BOP2 OPND* means: *OPND BOP1 (OPND BOP2 OPND)* if BOP2 has a higher precedence than BOP1.
- All operators generalize to their operand's data structure in a vector-like (point for point) manner whenever possible.
- The binary operators known to FLEX, listed from high precedence (just under the unops) to low, are:

```
↑              exponentiation
* / // mod       multiplication, division, integer division, residue
+ -             addition, subtraction
< > ≤ ≥ = ≠      relationals
∧               logical product
∨               logical sum
IMP, XOR, EQV    other logical operators
BY TO  ( )      range operators
User bop         user-defined (easy) binary operators
.               Associative
#               concatenation
←               replace value of
|              Alternate
```

- Grouping may be organized by ( ) in a well-balanced manner.

UNARY OPERATORS

4.11 Basic Arithmetic Operators

Square root: sqrt x x_0
Natural log: ln x x 0
Exponential: exp x e^x 10^{100} (use exp^1 for *e*)
Sine: sin x |x| < 100 radians
Cosine: cos x |x| < 100 radians
Tangent: tan x |x| < 100 radians
Arctangent: atan x |x| < 100 radians

- FLEX computes the true value rounded to about 11 significant digits in most cases. As in JOSS, certain *magic values* are *hit on the nose* - e.g., cos x ↑ 2 + sin x ↑ 2 = 1 *exactly*, etc.

4.12 Number Dissection Operators

- Absolute value: abs x
- Signum: sgn ← 'value: x. If x < 0 then -0 else if x=0 then 0 else 1 '
- Integer part: L, ip x ...carries sign of x ...
- Fractional part: fp x
- Next integer (ceiling): cl x
- Digit part: dp
- Exponent part: xp x ...carries sign of x ...
- Note that x ip x + fp x, and that x= dp x * 10 ↑xp x

4.13 Other Unary Operations

- not x or ¬ x is if x then false else true.
- + x means 0 + x
- - x means 0 - x
- User-defined unary operations are procedures with one argument that generates a value and are treated exactly as FLEX unary operators.
- hash x forms a uniformly-distributed *scrambled* number from x
- prand yields a pseudo-random number
- rand yields a real random number (generated from noise)
- any x randomly chooses from the list x.

4.14 List and String

- A string of text "ABCD" may be thought to be identical with "A", "B", "C", "D" for the following operators.
- head: hd "ABCD" = "A"
- but head: bh "ABCD" = "BCD"
- tail: tl "ABCD" = "D"
- but tail: bt "ABCD" = "ABC"

4.15 Binding

- The operators are described in detail in sections 7 and 8.
- Reference to: A ← @B will cause A to be synonymous with B
- Reference to: A ← @B [I] will be synonymous with B[val.I]
- Reference to: A ← @[B[I]] will be synonymous with B I
- Reference to: A ← 'B[I] '] will be synonymous with B I
- Bind fresh symbol: :A creates a local symbol A and (since is unary) binds it to Ω.

4.16 Symbolic

- These facilities allow easy symbol definition for the highly interactive user.
- *Define local symbol: :name* The symbol *name* will be declared local to the list that the user is currently *in*. If it is bound to Ω, FLEX will respond (OLD) if a local already exists with that name.
- *Define global symbol: :name* The symbol *name* will be declared in the most global list - the one thought of as the *direct* or *file* level. This construct obviates the need for files, etc. *(OLD)* will be displayed if the name is already in use.

BINARY OPERATORS

4.17 Numeric

- ↑ : x ↑ y means x^y. x ↑ y ↑ z means x ↑ (y ↑ z)
- * : x * y means x.y for fractions
- / : x / y means x/y for fractions
- // : means ip (x / y)
- mod : residue. x mod y means ip (x - ((x/y)*y)
- + : the usual for fractions
- - : the usual for fractions
- Except for ↑, all operations with exact operands give *exact* results unless *exact* range is exceeded.

4.18 Relational

- The operators yield *1* (true) or *U* (false), depending as to whether the relations hold.
- < , >: less than, greater than
- ≤ , ≥: less than or equal , greater than or equal
- = , ≠: equal, not equal
- A<B<C means A<B ∧ B<C
- "ABC" < "ABD" = true
- "AB" < "ABD" = true

4.19 Logical

- The logical operators are ∧, ∨, xor, imp, eqv

4.20 Range

- *A by B to C* and *A(B)C* means a string of numbers starting with A, A + B, A + 2B, ..., C
- B (C) D (E) F means: B, B + C, B + 2C, ..., D, D + E, D + 2E, ... F
- A range is a useful abbreviation for a list of values, since it is not explicitly evaluated until acted on by an operator.

4.21 Associational

- A • B ← C relates the three names so that

  Show ← A • B
    C

- *A • B is C* appends C to the set of values denoted by A • B
- *A • B isn C* removes C from association (if there)
- *A • B ⊂ C* is *true* if C is associative *else* false
- *? • B ⊂ C* will retrieve *all* A's which have been related.

4.22 Set

- ⊂ subset of
- ∈ inclusion
- ∉ not included

4.23 Other Binary Operators

- *User bops* are user-defined binary operators of fixed precedence. This is simply an easy way to define a new operator. Operators of any precedence can be defined using the input-aspect of the *List*.
- # is a concatenation operator which forces the result to be a string.
- ← binds the value of the right-hand operand to the name of the left-hand operand.
- ⇐ binds the nth member of the list, which is the right-hand operand for an nth time through. If a new member could be bound, then the value of the assignment is true; otherwise, the old binding is retained and the value of the expression is false. See *while* in Section 5.

CONDITIONAL EXPRESSIONS

- All conditionals are of the form *conditional_clause true_part* (*false_part* | Ω)
- The conditional_clause is heralded by *if*, *while*, *when*
- The true part is denoted by *then*, *do*
- The false part begins with *else*.
- If the conditional clause evaluates to *true* (1), the *true_part* is executed.

```
Show ← if a = b + c - d + e > .5 = G
        then  b ← a + b
        else  b ← a - b;

Show ← if a < b < e
        then [a, b, c]
        else "This is an error";
```

4.24 while

- The body of the conditional clause usually consists of boolean valued expressions which are elahoratcd for each iteration. If the value is *true*, the true_part is elaborated, followed by a fresh elaboration of the conditional.
- The boolean assignment operator may be used to assign new bindings to variables in a serial fashion.

```
...as an ALGOL-like for statement...
while  I ⇐ 1 by 2 to 3 do   ...;
...an ALGOL-like for while statement...
while I ⇐ 1 ∧ X < 5 do ...;
...whichever list runs out first will terminate..
while I ⇐ (1, 5, 3, 1 to 10, 3 by -2- to -I)
∧ J ⇐ (5, 10, A by B to C) do...;
...X will assume all sons of Jones...
while X ⇐ (offspring.Jones sex.? is male)
```

4.25 when

- This conditional provides a *software* interrupt. It is activated by having the elaborator *pass over* it in the current context. It stays active until the current context is deallocated.
- It is *not* passivated when its context is dismissed by a *leave*. It thus provides a convenient way to *wake up* a passivated process by some external condition.
- For an intuitive discussion of how the *when* is implemented, see Chapter 3.d in "The Reactive Engine".

References

This bibliography is presented both to acknowledge the great debt of the author to the work of others, and to share with the reader a compendium of provocative reading.

1. Whorf, W. L. Language, Thought and Reality. Edited by J. B. Carroll. Cambridge: Massachusetts Institute of Technology, 1954.

2. Minsky, Marvin. Computation: Finite and Infinite Machines. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1967.

3. McCarthy, John. "Basis for a Mathematical Theory of Computation." Computer Programming and Formal Systems. Edited by P. Braffort and D. Hirschberg. Amsterdam: North-Holland, 1963.

4. Church, Alonzo. The Calculi of Lambda-Conversion. Annals of Mathematics Studies, No. 6. Princeton: Princeton University Press, 1941.

5. Wijngaarden, A. van; Mailloux, B. J.; Peck, J. E.; and Koster, C. H. A. Report on the Algorithmic Language ALGOL 68. Edited by A. van Wijngaarden. Amsterdam: The Mathematical Center, 1969.

6. Barton, R. S. "A New Approach to the Functional Design of a Digital Computer." Proceedings of the Western Joint Computer Conference, XIX (May 9-11, 1961) , 393-396.

7. Lampson, Butler W. Interactive Machine-Language Programming. Document No. 30.50.11, December 6, 1965. Berkeley: University of California.

8. House, Roger. Reference Manual for NARP, an Assembler forr the SDS-940. Document No. R-32, January, T968. Washington, D. C.: A. R. P. A.

9. Lichtenberger, W. W. ARPAS Reference Manual for Time Sharing Assembler for SDS-930. 2nd Ed. Document No. R-26, February 24, 1967. Washington, D. C.: A. R. P. A.

10. Strachey, C. "A General Purpose Macrogenerator." Computer Journal, VIII (October, 1965), 225-241.

11. Mooers, Calvin N. "TRAC, A Procedure Describing Language for the Reactive Typewriter." Communications of the ACM, IX, 3 (March, 1966), 215-219.

12. Carr, C. Stephen; Luther, David A.; and Erdmann, Sherian. The Tree-Meta Compiler-Compiler System; A Meta Compiler System for the Univac 1108 and the General Electric 645. Technical Report RALC-TR-69-83, March, 1969. Salt Lake City: University of Utah Department of Computer Science.

13. Feurzeig, Wallace, and Papert, Seymour A. Programming-Languages as a Conceptual Framework for Teaching Mathematics. (Typewritten.) Unpublished results of a cooperative study made by representatives of Bolt Beranek and Newman, Inc., and Massachusetts Institute of Technology, in Cambridge, Massachusetts.

14. McCarthy, John. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part 1" Communications of the ACM, III, 4 (April, 1960), 184-195.

15. Wirth, K., and Weber, H. "Euler - A Generalization of ALGOL, and its Formal Definition: Part I, Part II." Communications of the ACM, IX, 1 and 2 (January and February, 1966), 13-25 and 88-99.

16. Burroughs B5500 Information Processing System Reference Manual Detroit: Burroughs Corporation (1964).

17. Burroughs Algebraic Compiler. Bulletin 220- 21011-P. Detroit:Burroughs Corporation (1961)

18. Naur, P., ed. "Report on the Algorithmic Language ALGOL-60." Communications of the ACM, III, 5 (May, 1960), 299-314.

19. McKeeman, W. M. An Approach to Computer Language Design. Technical Report No. CS 48, August 31, 1966. Stanford: Stanford University Department of Computer Science.

20. Dahl, Ole-Johan, and Nygaard, Kristen. "SIMULA - an ALGOL-Based Simulation Language." Communications of the ACM, IX, 9 (September, 1966), 671-678.

21. Shaw, C. "JOSS: A Designer's View of an Experimental On-Line Computing System." AFIPS Conference Proceedings, XXVI, 1 (Fall, 1964), 455-464.

22. Lampson, Butler W. CAL Reference Manual. Document No. 30.50.60, September 27, 1965. Berkeley, California: University of California.

23. BASIC. Hanover, N. H.: Dartmouth College (1965).

24. Kennedy, Phyllis R. The TINT Teacher Guidebook. Technical Memorandum TM-1933/000/01, August 10, 1964. Santa Monica, Calif.: System Development Corporation.

25. Weissmann, Clark. LISP Primer. Technical Memorandum SDC-PM2337-010-00, June 14, 1965. Santa Monica, Calif.: System Development Corporation.

26. Deutsch, L. Peter, and Lampson, Butler W. Reference Manual 930 LISP. Document No. 30.50.40. June 5, 1965. Berkeley, Calif.: University of California.

27. Quam, Lynn H. Stanford LISP 1.6 Manual. SAI, 1968.

28. Moors, Calvin N. "TRAC, A Text-Handling Language." Proceedings ACM, August, 1965, 229-246.

29. Kay, Alan C. FLEX - A FLexible Extendable Language. Technical Report 4-7, June, 1968. University of Utah Department of Computer Science.

30. Feldman, J. A. "A Formal Semantics for Computer Languages and its Application in a Compiler-Compiler." Communications of the ACM, IX, 1 (January, 1966), 3-9.

31. Evans, A. Syntax Analysis by a Production Language. PhD Thesis, 1965. Pittsburgh: Carnegie Institute of Technology.

32. Floyd, R. S. "A Descriptive Language for Symbol Manipulation." Journal of the ACM, VIII (October, 1961), 579-584.

33. Evans, A. "An ALGOL-60 Compiler." Annual Review In Automatic Programming, IV (1964), 87-124.

34. Knuth, Donald. "On the Translation of Languages From Left to Right" Information and Control, VIII, 6 (December, 1965), 607-659.

35. Buchholz, W. , ed. Planning a Computer System - Project Stretch. New York: McGraw-Hill, 1962.

36. Feldman, J. , and Rovner, P. The LEAP Language and Data Structure. Technical Note DS-5436, October, 1967. Lexington, Mass.: M. I. T., Lincoln Laboratory.

37. Feldman, Jerome. Aspects of Associative Processing. Technical Note 1965-13, April, 1965. Lexington, Mass.: M. I. T., Lincoln Laboratory.

38. Wirth, N., and Hoare, C. A. R. "A Contribution to the Development of ALGOL." Communications of the ACM. IX (June, 1966), 413-432.

39. Evans, David C. , and Leclerc, Jean Ives. "Address Mapping and the Control of Access in an Interactive Computer." AFIPS Conference Proceedings, XXX (April 18-20, 1967), 23-32.

40. Holt, A. W.; Shapiro, R. M.; and Warshall, S. A Mathematical Method for the Description and Analysis of Discrete, Finite Information Systems". Wakefield, Mass.: Computer Associates, Inc., 1965.

41. Shapiro, Robert M. A Method For Learning the Legitimate States For a Class of Systems. Wakefield, Mass.: Inc., Computer Associates, 1965.

42. Iturriaga, R.; Standish, T. A.; Krutar, R. A.; and Earley, J. C. "Techniques and Advantages of Using the Formal Compiler Writing System FSL to Implement a Formula ALGOL Compiler." AFIPS Conference Proceedings, XXVIII (Spring, 1966), 241-232.

43. APT Encyclopedia. St. Paul: UNIVAC Division of Sperry Rand Corporation (1963).

44. Green, C. Cordell, and Raphael, Bertram. The Use of Theorem Proving Techniques in Question-Answering Systems. Paper presented at the 1968 ACM Conference, Las Vegas, Nevada, August 27-29. Menlo Park, Calif.: Stanford Research Institute.

45. Lichtenberger, W. W., and Pirtle, M. W. A Facility For Experimentation in Man-Machine Interaction. Document No. 40.20.20, January 12, 1966. Berkeley, California: University of California.

46. Engelbart, D. C. Augmenting Human Intellect: Experiments, Concepts, and Possibilities. SRI Project 3578, March, 1965. Menlo Park, Calif.: Stanford Research Institute.

47. Warnock, John E. A Hidden Surface Algorithm for Computer Generation of Half tone Pictures . Technical Report 4-15, June, 1969. Salt Lake City: University of Utah Department of Computer Science.

48. Sutherland, I. E., and Sproull, R. F. "A Clipping Divider." AFIPS Conference Proceedings, XXXIII, 1 (December 9-11, 1968), 765-776.

49. Balzer, R. M. Dataless Programming. Memorandum RM-S290-ARPA, July 1967. Santa Monica, Calif.: The Rand Corporation.

50. Engelbart, D. C., and English, W. K. "A Research Center for Augmenting Human Intellect." AFIPS Conference Proceedings, XXXIII, 1 (December 9-11, 1968), 395-410.

51. Turn-12 Machining Center. Milwaukee: Kearney and Trecker Corporation (1969).

52. Lampson, Butler W. "A Scheduling Philosophy for Multiprocessing Systems." Communications of the ACM. XI, 5 (May, 1968), 347-360.

53. Conway, Melvin E. "A Multiprocessor System Design." AFIPS Conference Proceedings, XXIV (Fall, 1963), 139-146.

54. Dennis, Jack B., and Van Horn, Earl C. "Programming Semantics for Multiprogrammed Computations." Communications of the ACM, IX, 3 (March, 1966), 143-155.

55. Newall, A. "A Note on the Use of Scrambled Addressing for Associative Memories." Unpublished paper, December, 1962.

56. Klinkhamer, J. F. "On Key-to-Address Transformation for Mass Storage." Unpublished paper, 1964.

57. Copeland, Lee, and Carr, C. Stephen. Graphics System. Technical Report 4-1, November 15, 1967. Salt Lake City: University of Utah Department of Computer Science.

58. Newman, W. An Implementation of FLEX on the PDP-9. Unpublished paper, February, 1969.

59. Landin, P. J. "A Correspondence Between ALGOL 60 and Church's Notation." Communications of the ACM, VIII, 2 and 3 (February and March", 1965), 89-101 and 158-65.

60. Leavenworth, B. M. "Syntax Macros and Extended Translation." Communications of the ACM, IX, 11 (November, 1966). 790-793.

61. Heinlein, R. The Moon is a Harsh Mistress. N. Y.: G. P. Putnam and Sons, 1966.

62. Bryan, G. E., and Paxson, E. W. The JOSS Notebook. Memorandum RM-5367-PR, August, 1967.Santa Monica, Calif.: the Rand Corporation.

Vita

Name
Alan Curtis Kay
Birthdate
May 17, 1940
Birth Place
Springfield, Massachusetts
High School
P.D. Schrieber U.S.
College
Bethany College Bethany, West Virginia (1959-1961; biology, math)
Service
U.S.A.F. 1961-1963: Active, 1963-1967: Inactive: HD
University
University of Colorado Boulder, Colorado (1963-1966; math and anthropology)
Degrees
BA (math) University of Colorado, Boulder, Colorado, 1966.

MS (Computer Science) University of Utah, Salt Late City, Utah (with distinction) 1968

Computer Science

1962-63; Randolph AFB, Texas, Data Systems Programmer
1963-66; NCAR, Boulder, Colorado, Data Systems Programmer
1966-present; University of Utah, Salt Lake City, Utah, ARPA Research Assist.
1967-present; Creative X Corporation, Salt Lake City Utah, Consultant

Show Business

1958-1966; Eastern United States, Professional Jazz and Pit Guitarist
1965-66; Boulder Colorado Theatres, Composer, Music Director, etc.
1967-68; KUED-TV, Salt take City, Utah, Composer, Music Director, etc.

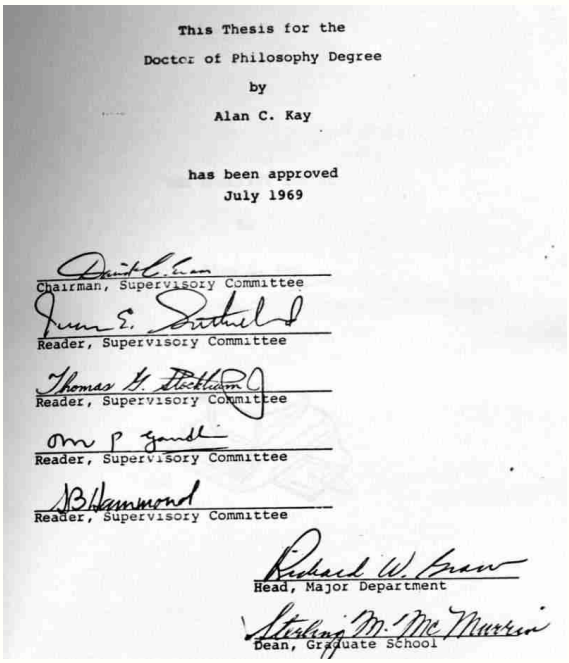
Hobbies

Musical Instrument design and fabrication, reading.


THE REACTIVE ENGINE.

KAY, A.C. 1970

Published on demand by
UNIVERSITY MICROFILMS INTERNATIONAL
Ann Arbor, Michigan, U.S.A. London, England


This Thesis for the
Doctor of Philosophy Degree
by
Alan C. Kay

has been approved
July 1969

Chairman, Supervisory Committee

Reader, Supervisory Committee

Reader, Supervisory Committee

Reader, Supervisory Committee

Reader, Supervisory Committee

Head, Major Department

Dean, Graduate School