AD-761 962

FLEX-A FLEXIBLE EXTENDABLE LANGUAGE

Alan C. Kay

Utah University

Prepared for:

Advanced Research Projects Agency

June 1968

9

AD 761962

Technical Report 4-7

Alan C. Kay

FLEX - A FLEXIBLE EXTENDABLE LANGUAGE

June, 1968

COMPUTER SCIENCE

Information Processing Systems

University of Utah

Salt Lake City, Utah

Advanced Research Projects Agency • Department of Defense • ARPA Order 829

Program code number 6D30

104

Acknowledgments

This work could not have prospered without the challenging and pertinent criticisms of many people. I would particularly like to thank Professor David C. Evans and Robert S. Barton of the University of Utah, for providing an inspirational environment in which to work. I also wish to thank C. Stephen Carr for many hours of fascinating discussion (both here and there) that certainly has had a lot to do with the present form of the work.

Table of Contents

Abstract

The FLEX system consists of merged "hardware" and "software" that is optimized towards handling algorithmic operations in an interactive, man-machine dialog.

The basic form is that of a hardware implementation of a parametric compiler embedded in an environment that is well-suited for semantically describing and pragmatically executing a large class of languages. The semantic language is called FLEX, includes the compiler-compiler as a string operator and is used as the basic medium for carrying out processes. It is of a higher-level nature and may itself be used for describing many algorithmic processes.

The machine itself is designed to be of the desk-top variety and sell at a low price. Because of these design parameters, many compromises in time and space had to be made to save money. The software system is implemented in read-only memory. To allow any possibility at all of debugging such a scheme, the algorithms involved were distilled down to their essence so that the entire system for the machine can be displayed (in flow diagram form) on a small wall chart.

In many senses the described system is a "syntax-directed" computer.

I. Preface

This is a working document submitted as work-in-progress for the degree of Master of Science. It proposes an integrated hardware-software system for performing algorithmic operations.

The following is intended to be a complete and concise description of the system rather than a mere report of results in the hope that readers will not have to spend valuable time trying to figure out how it is all accomplished. Apologies for any and all gaps, chasms, and crevices.

II. Introduction

The FLEX language is intended to be a simple yet powerful and comprehensive notation to express computer-oriented algorithms. It follows the traditions set by ALGOL 60 and several generations of EULER. [1,2]

a. Calculation

At the lowest level of use FLEX is easier to learn than either FORTRAN or ALGOL. The use of it as a desk calculation language may be mastered in a few minutes. For example: we may wish to evaluate a qalculation involving only numbers. The expression is simply entered through the keyboard as shown. Assigning the answer to the reserved word "display" indicates that the answer is to be returned to the CRT.

```
'display←1.6*2.9522/(19.7-9.2);



4.4985905
```

At this level of use the entered FLEX code is executed statement by statement so that it acts as an interactive language. The " ' " is supplied by the processor and indicates to the user that FLEX is ready for input.

At any time the entered text may be modified by using the powerful text editor associated with the language.

b. <u>Variables</u>

The next step for the initiate would be the evaluation of simple algorithms using variables as well as constants and perhaps a more interesting display of results. The following routine should be studied.

```
'begin
 new a,b,c,d;
   b←1;c←10.3;
   display←"a="#(a+b*c/1.2+c)
     #"d=" # (d+a-b+c*1.2);

   a=18.883333  b=30.243333
```

Notice that no format statements or separate write commands are required. The handling of strings of textual characters is a primitive operation within FLEX and the catenation operator "#" is used to connect together literal strings enclosed by quotes: "a=", to numbers generated by executing arithmetic assignment statements.

The whole is realized as one string of characters at the display end and is output on the lower half of the screen as shown.

The creation of variables is indicated to FLEX by the use of the reserved word new followed by a list of variable names. Type need not be specified. FLEX is entirely free form in nature. There are no card column numbers to worry about as in FORTRAN.

c. Decisions

Decision-making and branching are handled by one comprehensive statement. It is of the form if $\boxed{\quad}$ then $\boxed{\quad}$ else $\boxed{\quad}$. The boxes may be any construct in the language including blocks and entire programs.

In almost all cases this eliminates the need and use of one of the most common pitfalls in programming: the label and associated go to statement. These are provided in FLEX but they will rarely be used. Former FORTRAN programmers who convert to ALGOL find that they almost never need to use labels or go tos and time spent in debugging gets reduced by a sizable factor.

d. The Use of Blocks as in ALGOL 60

The scope of variable identifiers may be delimited by further use of the parentheses begin end. For example:

```
begin
  new  a,b,c,d,e,f;
       .
       .
       .
       a←b+c+d+e;
```

```
          .
          .
          .
          begin
            new  a,b,c;
                 .
                 .
                 .
                 a←b+c+d+e;
                 .
                 .
                 .
          end;
          f←a+b;
    end;
```

Within a block delineated by a begin end pair, all identifiers declared by a new list are considered to be local to that block. An identifier used in the block but not declared there will be the one declared in the nearest containing block.

In the example above 'a' in the outermost block is given a value in the assignment

a←b+c+d+e;

following this a new block is entered and a new declaration is executed: new a,b,c. Effectively this overrides the previous declaration so that, in this inner block, the variables a,b,c are considered to be totally new and local. In the assignment of identical form:

$$a \leftarrow b \leftarrow c \leftarrow d \leftarrow e;$$

the a,b,c are from the inner block and the d,e are from the outer block. The a,b,c of the outer block are not touched.

When the end of the inner block is reached, the inner block ceases to exist; we are again in the scope of the outer block. The assignment f←a+b sets f to the value a+b where a,b are the outer block variables.

The use of Block Structure in this way allows sections of programs written as blocks to be arbitrarily inserted without fear of destruction when variable names happen to match as can easily happen in unstructured languages like FORTRAN.

The use of the word new means just that. The variables following are created fresh each time a block is entered.

e. Extendability

New binary and unary operators may be declared giving the programmer powerful control over the language itself. For example, the functions max and min may be useful as operators, i.e.:

```
begin  new max, min;
  bop max ←'new a,b. if a>b then a else b';
  bop min ←'new a,b. if a<b then a else b';
```

```
        a+b+c*d max b-c*d; '''if d>b, then a+b'''
```

In this manner the programmer may tailor the operator structure of FLEX to suit his needs. This feature both eases the programming burden and causes the program to be easier to read and be understood by others.

FLEX may also be extended by either modifying itself via the compiler-compiler contained in the language or a wholly new language may be created using the same tools. The use of the operators com and scribe will be discussed in a later chapter.

f. Comments

Comments are handled very simply. Any text inserted between the symbols '''-''' will be ignored by FLEX. This allows comments to be inserted anywhere--even in the middle of an arithmetic expression.

Examples:

```
        a+b+c*d'''this expression is simple''';
        a+b'''this expression is simple'''+c*d;
```

This introduction has barely scratched the surface of the FLEX language. It was not intended as an exposition of FLEX, but only to give the average user (a FORTRANer) a feel for the more comprehensive discussions that follow.

III. The Language Environment

A. Explanation of the Formalisms Used

Syntax

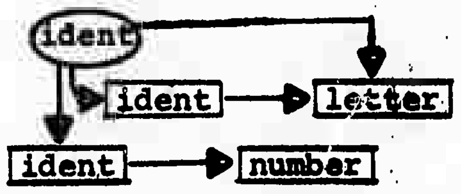
Two formalisms are used to describe the syntax of FLEX: A variant of BNF (Backus Normal Form) (with factoring) and the syntax-chart method developed by Burroughs Corporation. [10]

For an example, let us describe a FLEX identifier.

In English: An identifier is a text string of arbitrary length starting with a letter and thereafter composed of either letters or numbers.

In BNF: <ident>::=<letter> |<ident> <letter> |<ident><number>

In Chart:



A box says that the construct is defined elsewhere on the chart; a lozenge indicates that this is the definition.

Semantics and Pragmatics

The semantics and pragmatics of FLEX will be largely described in English (drawing heavily from accepted notions in mathematics and computer science).

Whenever possible, FLEX, itself, will be used for description and, indeed, this is done in the SCRIBE chapter where FLEX is presented written in itself.

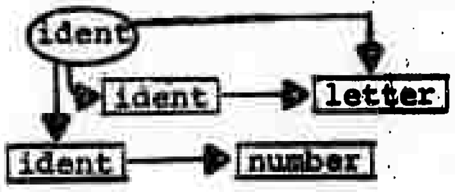
III. The Language Environment

A. Explanation of the Formalisms Used

Syntax

Two formalisms are used to describe the syntax of FLEX: A variant of BNF (Backus Normal Form) (with factoring) and the syntax-chart method developed by Burroughs Corporation. [10]

For an example, let us describe a FLEX identifier.

In English:	An identifier is a text string of arbitrary length starting with a letter and thereafter composed of either letters or numbers.
In BNF:	<ident>::=<letter> |<ident><letter> |<ident><number>
In Chart:	

A box says that the construct is defined elsewhere on the chart; a lozenge indicates that this is the definition.

Semantics and Pragmatics

The semantics and pragmatics of FLEX will be largely described in English (drawing heavily from accepted notions in mathematics and computer science).

Whenever possible, FLEX, itself, will be used for description and, indeed, this is done in the SCRIBE chapter where FLEX is presented written in itself.

Because of the recursive nature of FLEX (and the FLEX description) it is impossible to describe it in a linear order. Therefore, some reliance has been placed on the user's intuition for some of the examples presented.

Examples

The examples will be largely presented in FLEX although occasionally they will be drawn from ALGOL 60 and FORTRAN to present some interesting contrasts.

B. The FLEX Language

1. Syntactic Atoms

Syntax

```
<letter>:: = A|B|...Y|Z|a|b|...y|z|^|

<digit>:: = 0|1|2|3|...|9

<delimiter>:: = , | ; | ( | ) | ' | ' | [ | ] | : | ← | # | ⌈ | ⌋ | ⌉ | ↑ |
                + | - | * | / | ÷ | = | ≠ | < | > | ≤ | ≥ | ∧ | ∨ | ⊻ |
                ≡ | ⇒ | ¢ | ∈ | ∞ | Ω | ∩ | ∪ | ⊂ | ? | " | . |   |

<reserved words>:: = begin | end | new | if | then | else |
                     goto | type | ceil | floor | sin |
                     cos | atan | abs | rand | prand | hash |
                     exp | ln | sqrt | is | isn | any | of |
                     while | as | to | by | do | array | field
                     bop | uop | name | val | map | act |
                     terminate | leave | xin | yin | plst |
                     plpt | plln | control |
```

Semantics

The text characters are simply numbers of small precision. The numbering starts with (0,...,9) for ("0",...,"9") and continues with (10,12...60) for

("A",...,"Z") and (11,13...61) for ("a",...,"z"). A special space symbol is numbered 62. The delimiters are numbered from 63 on. (true, false) are identical with (1.0).

Pragmatics

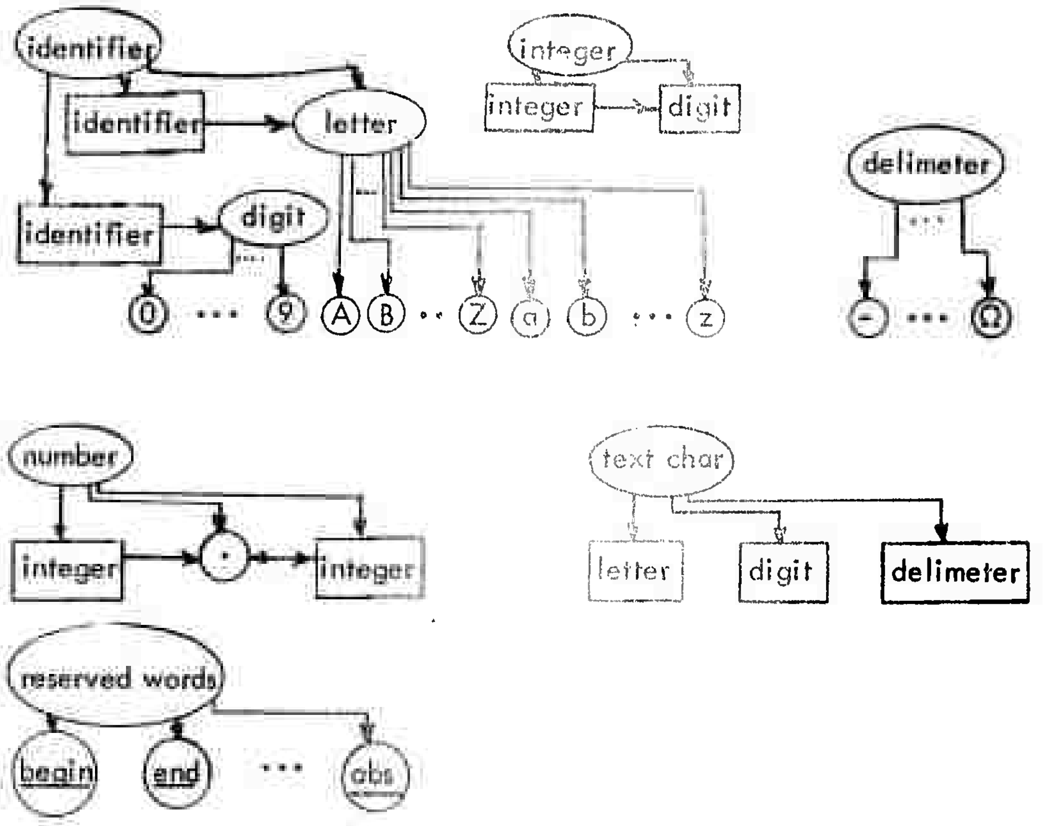
Text characters are integers.

Justification

Many internal character sets have been used by the industry. The principle reasons for this one are:

1. It is sortable.
2. It is easily extendable for number systems of higher radix than 10.
3. It eliminates a table lookup for every character that is input to the compiler.

```
<identifier>:: = <letter> | <identifier> <letter>
[illegible]      |<identifier> <digit>

<integer>    :: = <digit> | <integer> <digit>

<number>     :: = <integer>{.{<integer>}}
                  |.<integer>

<text char> :: = <letter> | <digit> | <delim>
```



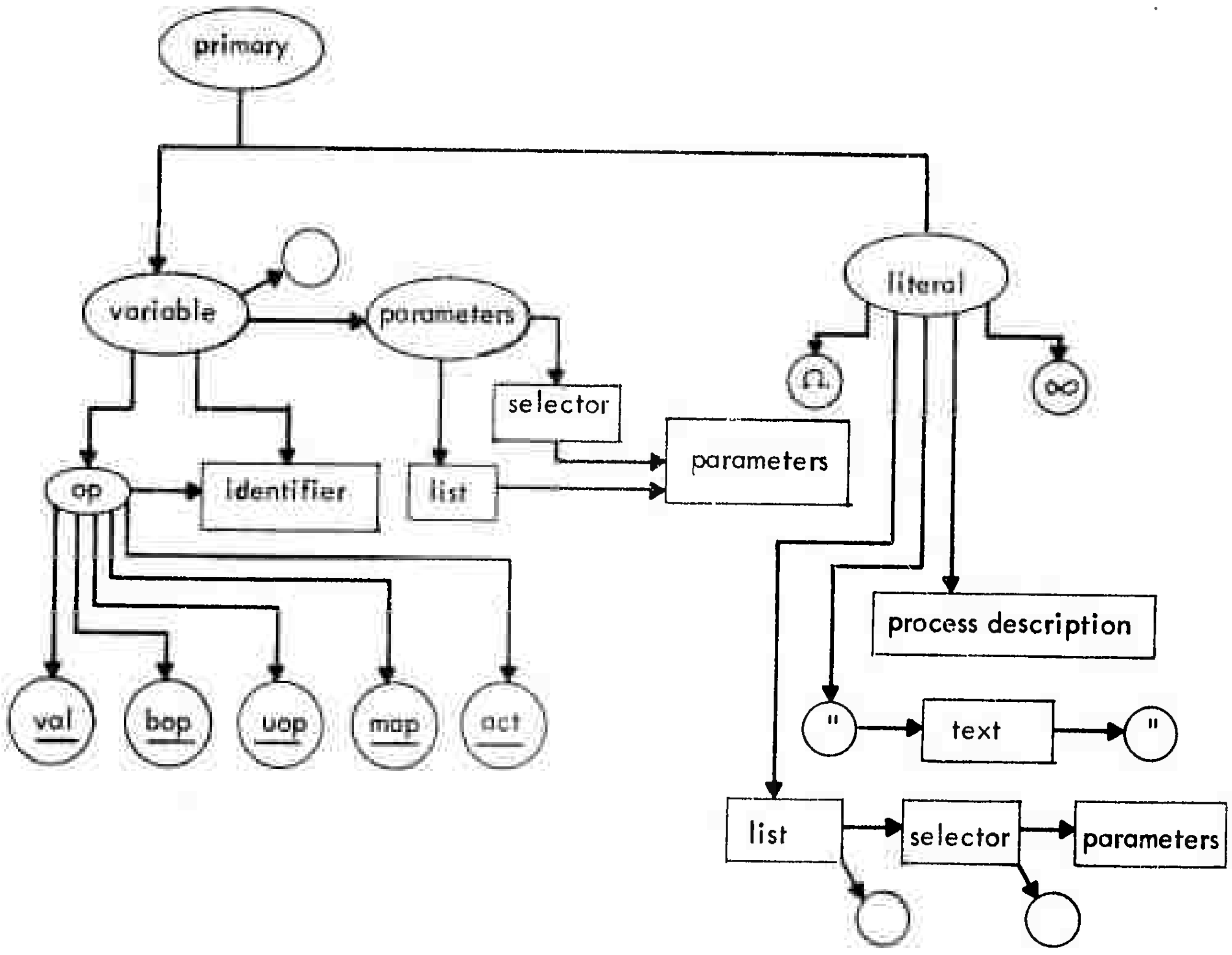
<u>Semantics</u>

<Reserved words> have the form of identifiers but are considered to be semantically identical to <delimiters>. Indeed, many <reserved words> have their exact counterparts among the <delimiters>. For example, <u>begin</u> and (,<u>end</u> and) are exactly identical -- so are $\wedge$ and <u>and</u>, $\vee$ and <u>or</u>.

<Identifiers> are considered to be names for constructs in the system and the basic flow of FLEX consists of assigning these names dynamically to the various objects which may be created.

<Numbers> are either integers or fractions. The precision will be unspecified for this chapter.

Primaries



<primary> :: = <literal>|<variable>{<parameters>}

<variable> :: = {<op>}<identifier>

<parameter> :: = <list>{<parameter>}|<selector>{<parameter>}

<literal> :: = Ω|∞|<list>{<selector>{<parameter>}}|
<process description>|"<text char>"

<op> :: = val | mop | bop | ucp | act |

a. Variables

1. <simple variable>	Semantics. Although this is not the smallest syntactic unit for a non-literal, in some cases it acts as the smallest semantic unit.
1a. <ident>	Semantics. This has attributes 1. name 2. type 3. topology 4. value all of which may be assigned dynamically.
1b. val<ident>	Semantics. The val overrides any value that may have been assigned. On the left side of the assignment arrow it will destroy any previous value. Justification. This allows the programmer to override name considerations to reassign a procedure quotation and to access a name.
1c. map<ident>	Semantics. This allows a user-derived process description (procedure) to be assigned to the access path of the variable. This allows complicated user structures to be indexed in the same manner as FLEX defined data structures. The map is described more completely in the section on <selectors>.
1d. bop<ident>	Semantics. This moves the <ident> into the parsing table as a binary operator. If the ident has had a '<Body>' assigned to it, then it will act as a binary operator. Pragmatics. A simple name inclusion using the generality of the quotation to full advantage. Justification. The language may be extended in a simple manner.
1e. uop<ident>	Note: Same as bop except <ident> is parsed as a unary operator.

Examples:

```
p←'a'

p←x+y  '''same as a←x+y'''

val p←'b'

p←x+y  '''same as b←x+y'''

bop max←'new a,b; if a<b then b else a'
```

2. `<simple variable> <selector>`

Semantics. The `<simple variable>` is assumed to contain data. Selection is performed as in ALGOL 60 and Euler. It acts like a simple variable after selection.

3. `<simple variable> <list>`

Semantics. This is just a procedure activation with actual parameters.

4. `<simple variable> <selector> <list>`

Semantics. Selection is performed first. It then acts like a simple variable.

5. Note: All further generalizations of this type are evaluated from left to right applying procedure activation and selection where needed.

Examples:

```
a [x,y,x]  (b,c,d); ''' "a" is an array of procedures'''

a (b,c,d)  [x,y,z]; ''' "a" is a procedure delivering an array'''
```

b. Literals

1. Ω **Semantics.** Means undefined. It is the result of illegal operations. All identifiers are set to this at declaration time.

   Pragmatics. The logical word is flagged.

   Justification. Allows a much more free syntax while still permitting a check of illegality.

2. ∞ **Semantics.** Is the result of division by zero. It is also used to map extendable arrays.

   Pragmatics. The logical word is flagged.

   Justification. Permits checking for overflow and declaring unspecified bounds without giving rise to a fatal error.

3. <number> **Semantics.** A fraction of unspecified precision.

   Pragmatics. Space is created to contain it.

   Justification. Useful for arithmetic.

4. "<text>" **Semantics.** A text literal is identical to the string quote of ALGOL 60. It has as wide a use as the <number>. Also, it will be seen later that com "a+b*D" is equivalent to 'a+b*D'.

   Pragmatics. A text literal is mapped and stored as a one-dimensional array.

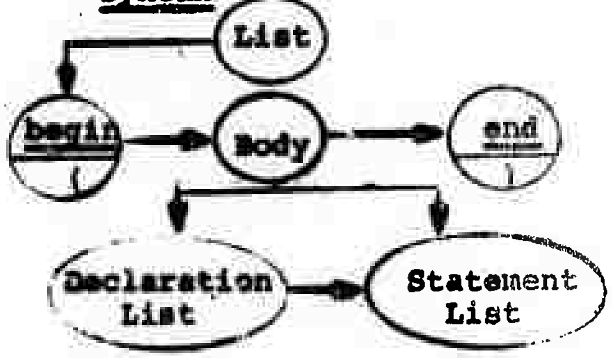
   Justification. Needed to generate text.

Examples:

```
if a ≠ Ωthen b←a+1.34;

if a ÷b ≠ ∞then display←a

          else display←"error in a";

display←if a÷b ≠ ∞then a else "error in a";
```

5. The List

Syntax



<List>::=begin<Body>
end|(<Body>)

<Body>::=<Declaration List>
<Statement List>
|<Statement List>

Semantics. The meaning of a list depends greatly on its form. All lists are thought of as executable elements that are delimited by the parenthetic pairs begin end or ( ). Execution of the <Body> takes place first. What remains (if anything) is then handled as an operand.

Taken as a unit, the list may have value or it may consume itself during execution. If a declaration is present then the list acts as an ALGOL Block in that identifiers declared in it are considered local to the list.

Pragmatics. The extent of a list is delineated both by parentheses and by commas. A list during execution is considered to be a vector on the runtime stack whose topology is determined by keeping track of the list of delimiters.

Justification. Here we have one concept and one construction replacing many that have been considered useful in ALGOLic languages. Also, by adopting this form, mnay useful new constructs are possible.

Examples

(a+bxc)	''' as a simple arithmetic primary '''
((2,1), (3,2))	''' as an array literal '''
begin+a b+c; x+a-b **end**	''' as an ALGOLic compound statement'''
prod (a,b,'c+d')	''' an an actual parameter list '''
begin	''' as a valued block '''

```
begin
  new a,b,c.
  a+a+(b+a*2+c);
end
```

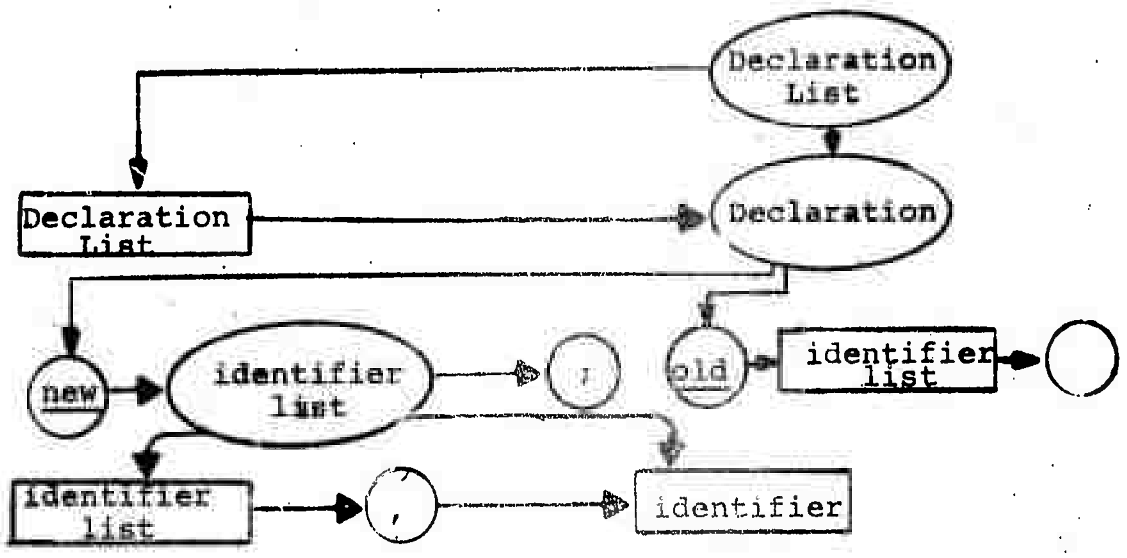
The Declaration List

Syntax

<Declaration List>::={<Declaration List>}<Declaration>

<Declaration>::=**new**<identifier list>;

<Identifier List>::={<identifier list>,}<identifier>



Semantics. The purpose of a declaration is to create and determine the scope of a name which will serve as a token or representative of some language elements. A name has as its scope the <body> in which it was declared. Since both type and topology may be assigned dynamically nothing more need be done than to list the new names for each <body>.

New declarations may be considered to be executed in the sense that a vector consisting of undefined values (Ω) is created in the runtime stack for the duration of the block. Positions in the vector correspond to each identifier declared.

Justification. Block structure has proved to be an extremely useful and important concept in ALGOLic languages. Besides aiding the programmer greatly in his own debugging, block structure is also the ideal way to delimit the scope of users in a multiprogramming and/or time-sharing environment.

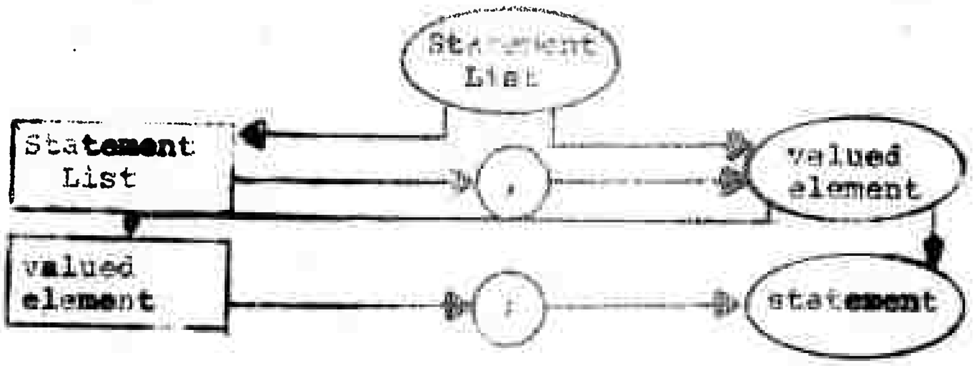
Examples:

```
A:begin              '''a,b,c are local to block A.'''
    new a,b,c.
    a←a+(b+c-2);
 B:begin
    new a.     '''this a is local to block B and supersedes'''
    a←b+c+2;   '''the previous declaration. b,c are'''
   end         '''from block A.'''
    a←a+2;     '''this a is the one declared in Block A'''
 end;
```

The Statement List

Syntax

```
<Statement List>::={<Statement List>,}<valued element>
<valued element>::={<valued element>;}<statement>
```



Semantics. Both commas and semicolons have their usual ALGOL meanings although they are allowed a much more flexible usage. A comma delimits elements in a vector so it may be considered to preserve the value of the previous expression or statement. A semicolon, then, may be considered to destroy the value of the previous expression or statement. As seen above, both delimiters may be freely mixed in a <body>.

Pragmatics. A semicolon flushes the top element in the run-time stack. A comma increases the vector count by 1 and leaves the top of the stack intact.

Justification. As seen in the examples, this construct allows great freedom and flexibility in creating lists of values and is pragmatically quite simple.

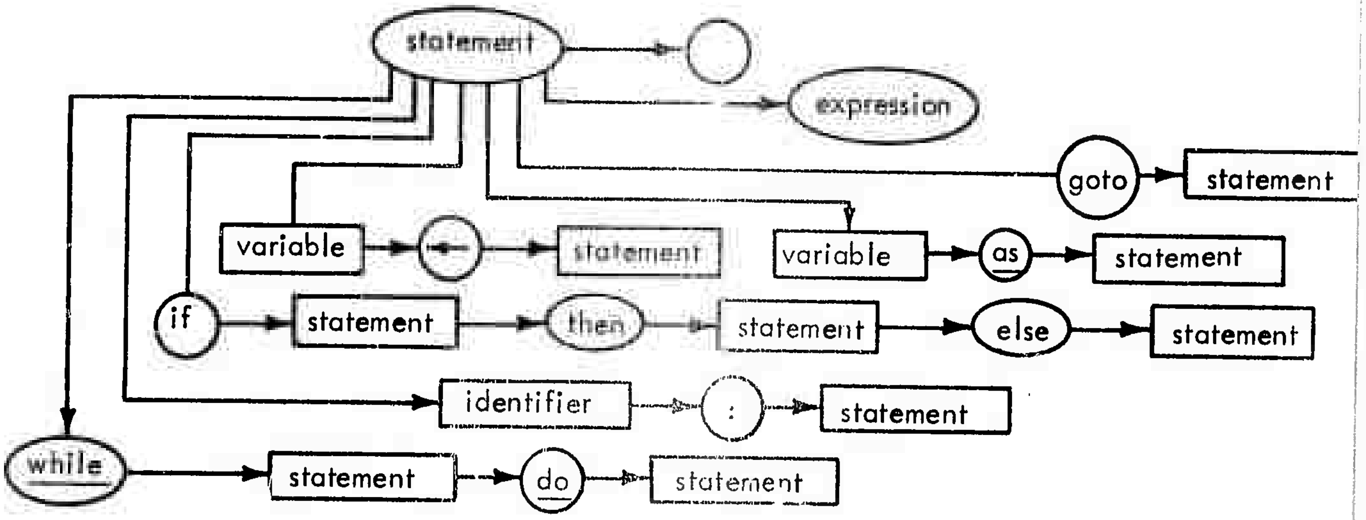
Examples:

```
(a,b,(c,1),b+c)          '''simple list'''

(1,if a then b else c,  '''b or c is left depending on whether
                         a is true or false'''

B:X←.5*(X+A/X);  '''Statements followed by ";" are not
                 retained'''

if X²-A>e then

goto B else b←X+.45,  '''value of b is left when sqrt
                       algorithm terminates'''

x+y)  '''value of x+y is left'''
```

The Statement

Syntax

```
    statement ::=<empty>|<expression>| go to <statement>|
                    <variable>←<statement>|<variable>as<statement>
                    |if<statement>then<statement>else<statement>
                   |<identifier>:<statement>
                    |while<statement>do<statement>
```



Semantics. All statements are considered to have value except for the go to statement (which is not considered to be a fundamental language concept, but is included for practical reasons). The go to initiates an unconditional branch in program control to the label specified by the (label-valued) <statement>. For the same reasons the labeled statement is not considered to be a language primitive. The other statement types will be considered separately.

Pragmatics. Since the syntax of a <statement> is so free, and more meaningless constructions are permitted, the limiting factor of produced nonsense is execution-time semantic checking.

The values of the statements are fetched into the runtime stack and are operated on in turn by the many operators of the languages.

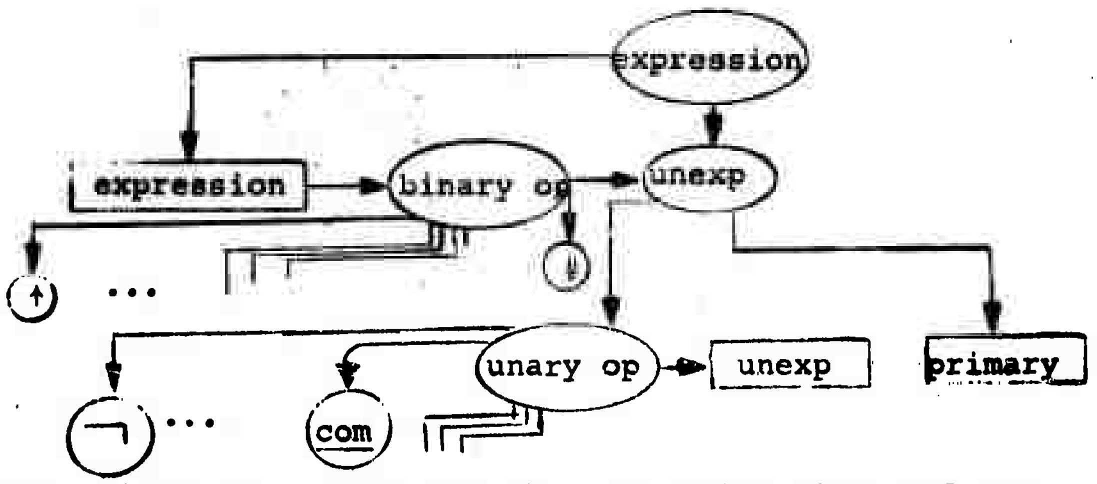
Examples:

```
abc:  a*b+z,
go to if a then abc else xyz;
xyz:  go to f [a←a+1]←abc;
```

<u>The Expression:</u> A. Presentation of Operators

<u>Syntax</u>

```
<expression>::={<expression><binary>}<unexp>
<unexp>     ::=<unary op><unexp>|<primary>
<unary op>  ::= Head | Tail | type |   |   | sin | cos | ...
               |com | scribe |...
<binary op> ::= ↑ | * | / | // | mod | + | - | = | ≠ | < |
               > | ≤ | ≥ | ∧ | ∨ | ⊻ | ≡ | ⇒ | of | is |...|
               <user binary ops>| ... | #...
```



Note: Binary operators are given in order of precedence.
Unary operators associate from the right.

<u>Semantics</u>. All operators generalize whenever possible to arrays and lists.

a. <u>Unary Operators</u>

<u>Operator</u>	<u>Meaning</u>
	logical negation
	unary minus
	<u>floor</u> x: integer part of x
	<u>ceil</u> x: <u>if</u> <u>floor</u> x=x <u>then</u> x <u>else</u> <u>floor</u> x+1;

Operator	Meaning
type	yields current disposition of a primary
head	first element of a list
tail	list with the head deleted
sin, cos	the usual trig functions
abs	absolute value
scribe	described in Section C. The operand is a string in "scribe" format which defines a language. The result is a set of tables for the compiler-compiler (com) in effect a compiler.
com	described in Section C. As a unary operator com accepts a string for one operand, and assumes the FLEX language tables as the other. The result is a compilation of the text resulting in an executable process description.

b. Binary Operators

Operator	Meaning
Arithmetic	
↑	x↑y means x^y in the usual mathematical sense for fractions
*	x*y means x.y in the usual mathematical sense for fractions
/	x/y means x/y in the usual mathematical sense for fractions
÷	x÷y means floor (x/y) in the usual mathematical sense for fractions
mod	x mod y means floor (x-(x y*y)) in the usual mathematical sense for fractions
+	x+y means x+y in the usual mathematical sense for fractions
-	x-y means x-y in the usual mathematical sense for fractions

Operator	Meaning
relational	
=, ≠, <, ≤, ≥, >, yield true or false	
logical	
, , ,/,≡, , the usual logical operators yield true or false	
compilative	
com	described in Section C. As a binary operator com accepts tables created by scribe for its left operand and, for its right operand takes a string in the new object language. The result is a compilation of the text resulting in an executable process description.
range	
to	the to operator describes a range of integer values either ascending or descending. Useful in any kind of interaction. a [3 to 6] means (a[3], a[4], a[5], a[6],). "abcdfg"[2 to 5] means "bcdf".
by	the by operator modified the interval within the range of a to. 1 by 3 to 10 means (1, 4, 7, 10)
associative	
of, is, isn,	These operators permit the formation and storage of relations between names.

We may say: John is son of Bill
and: Eric is son of Bill

We may then ask questions:

x←? is son of Bill,

-x will contain: ('John', 'Eric')

x←John is son of ?

-x will contain ('Bill')

x←? is ? of Bill

-x will contain (('son', 'John'), ('son', 'Eric'))

The ε operation yields a logical result.
The possible associative operators follow.

form	meaning
x is y of z	creates and stores the relationship
x isn y of z	destroys the relationship if it exists
xεy of z	asks if relationship is true
x∉y of z	asks if relationship is false
	--in general we ask for
x R y of ?	all z having relation y with x
x R ? of z	all relationship between x and z
x R ? of ?	all relations that x is involved in
? R y of z	all values with relation y with z
? R y of ?	all pairs having relation y
? R ? of z	all relations and values that z is involved in

Concatenation

# is the concatenation operator. The result is the concatenation of the two operands. The topology of the result is the most general topology of the two operands.

A	B	A‡B
array	literal	extendable array
array	array	extendable array
array	list	extendable list
list	list	extendable list

Pragmatics

a. Unary operators replace the top element of the process stock with the result.

b. Binary operators replace the top two elements of the process stock with the result.

c. The associative operators are really trinary in nature and therefore three names are actually collected in the stack before any action is taken. The resultant name replaces the three operands.

Justification

The use of binary and unary operators is justified both by tradition and the fact that fewer parentheses are needed than with functional notation.

The Expression: B. **Generalization of Operators**

Scope

If an operation is legal between two operands then it is also legal between other structures that have these operands as elements.

Arrays and Lists

If the dimensions are different between operands, then a logical "adjustment" is moade which logically creates enough copies of the operand of smaller dimension until the dimensions are matched. Then the operation is performed as a vector operation.

```
(a, b, c) * (x, y, z) means (a * x,b * y,c *z)
a * (x, y, z) means (a * x,a * y,a * z)
(a,b) * (x, y, z) means (a * x,b * y,z)
```

Examples of Expressions:

__display__ "a="(b↑2-4gc) #"f=" #__if__ a<b __then__ c __else__ g;

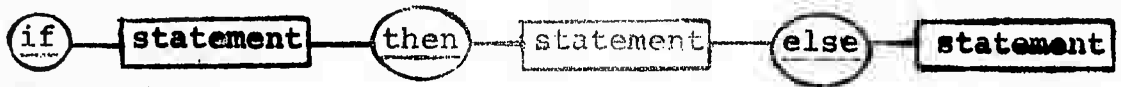
__Note:__ __if__ b=2, g=3, c=4 __then__ this would output on the CRT:

```
a= -44  f=4
```

The Conditional Statement

Syntax

if <statement> **then** <statement> **else** <statement>



Semantics

The value of the statement following the **if** must be reducible to **true** or **false** (one or zero). If a one, the statement following the **then** is executed, then the statement following the entire conditional statement is executed. If a zero, the sequence is similar to the above except the statement following the **else** is executed instead of the **then**.

The entire conditional statement has a value equal to that of the executed branch.

Note: This, the so-called "Long-form: of the **if** statement, is the only type presently available. It includes the "shortform: semantically since the empty statement is allowed. If it proves awkward to use, then the short form will also be added.

Pragmatics

There are only two jumps needed in the underlying environment: Jump-unconditional and Jump-if-top-stack-zero. These are invisible to the user and are inserted during the parse.

Examples:

```
if a←b+c-d+e↑.5=G
  then b←'a+b'
  else b←'a-b';....

if a<b<c
  then (a←b; b←c)
  else; '''use of empty statement'''
```

The Assignment Statement

Syntax

$$\begin{tikzcd} \boxed{\text{variable}} \arrow[r] & \bigcirc\!\!\!\!\leftarrow \arrow[r] & \boxed{\text{statement}} \end{tikzcd}$$

<variable>←<statement>

Semantics

The value of the assignment statement is considered to be the value of the <statement> and is assigned to the variable in a number of ways depending on which attribute of a variable is to be assigned.

As will be seen later, besides attaching a numeric value to a name, we may also dynamically specify the particular topology of that value. This includes gross structure such as whether the data is an array of such and such size, is a single item, or possibly a directed graph or tree. Fine structure may also be indicated. An item can be considered to be a number, a character, a byte of any width, a quotation of a program, a record, etc.

Pragmatics

The ←operator has a value for its right operand and a name for its left operand. The value is left in the stack and the name is destroyed.

Examples:

```
a←b+c                          '''value of b+c is named a'''

a←'b+c'                        '''a becomes the name for the
                                  procedure b+c''',

a←array(x,(5*b,c),(1,2))       '''a is mapped as a two-
                                  dimensioned array whose
                                  elements are x bits wide''',

(if a<b then a else b)←b+c     '''either a or b is assigned
                                  b+c depending on previous
                                  values of a,b''',

a←b←c←b+c                      '''multiple assignment''',

a←b*c+(d←b+c)                  '''nested assignment''',

a←"b+c"                        '''assignment of a text
                                  literal''',
```

The Assume Statement

Syntax

$$\begin{tikzcd}\boxed{\text{variable}} \arrow[r] & \text{as} \arrow[r] & \boxed{\text{statement}}\end{tikzcd}$$

<variable> as <statement>

Semantics

This is a generalization of the assignment statement in that the <variable> "takes on" or "assumes" values indicated by the <statement> one at a time if a loop is indicated by an interative while statement. Outside of a while only the first possible value is assumed and execution continues. The value of the assume statement is boolean: --being true if the variable has just assumed a value and false when there are no more possible values which may be assumed.

Pragmatics

The as operator has a value for its right operand and a name for its left operand. After all possible assignments are done, the value in the stack is replaced by a boolean value.

Justification

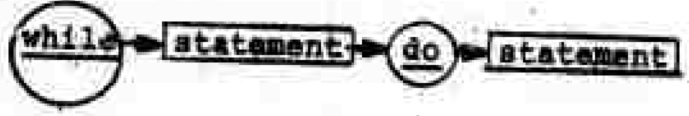
This particularly general form is most useful in iteration and applies itself well to all kinds of operands.

Examples

Will be given in the section on the while statement.

The Iterative Statement

Syntax



while <statement> do <statement>

Semantics

The statement following the do will be executed as long as the statement following the while is true. If it becomes false, then control transfers to the next sequential statement.

Pragmatics

Similar to the if statement except that a jump back is inserted after the statement following the do.

Justification

Besides covering a great many iterative situations in a simple manner, the while statement allows for the complete cessation of use of the go to and <label>.

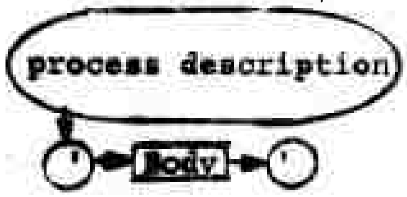
Examples:

```
while I as 1 by 2 to 13  '''as an ALGOLike for statement'''
  do <state>;
while I as 1   X<5  '''as ALGOLike for while state'''
  do <state>;
while I as (1,5,3,1 to 10,3 by -2 to -1) '''whichever list'''
  J as (5,10,A by B to C)              '''runs out first will'''
 do <state>;                           '''terminate execution'''
```

```
while X as ((Jones is parent of?) ∩ (male is sex of ?))
    do <state>; '''X will assume all sons of Jones'''
```

9. The Process Description

Syntax



<process description>::=
'<Body>'

Semantics

The process description is the backbone of the FLEX language. The user at a console is considered to be inside a process and he is handled by the system as just another active process.

The quotation may be named in the same manner as other literals in the language. A process may be created from the process description in one of two ways: as a serial procedure which is executed before the calling program is resumed, or as a parallel execution entity which runs concurrently and independently of the parent process.

In either case, if a new follows the ', the variables named are taken to be the formal parameters of the process description.

Pragmatics

The <Body> enclosed by the quotes is compiled separately and set aside in the same manner as other literals. If it is named, a reference is placed in the variable name area where it may be easily retrieved. A new stack is created for each process and an event notice is entered into the process que. User processes are executed on a "round robin" basis with a time quantum of about 10-16 ms. Processes that are alive may be active or passive; these states may be changed by themselves or by an interrupt by the real time processes.

Justification

Procedures and data handling are the keys to a successful language. In FLEX both these concepts have been generalized in a powerful manner.

Examples of simple procesures:

```
a       ← 'b';   '''a "run-time" equivalence statement''
x       ← 'b+c'; '''simple quotation without parameters'''
y       ← if b<c then 'b+c' else 'b-c'; '''conditional
                 assignment'''
z       ← a;     '''the name z is assigned to the value of b'''
a       ← z;     '''the name b is assigned to the value of z'''
val a ← 'c';     '''a is requivalenced to c'''
z       ← x'     '''the name z is assigned to the value of
                 b+c'''
x       ← z;     '''a pragmatic error is generated since x is
                 a value'''

for ← 'new a, b, c, d, e; '''an ALGOLike "for" procedure'''
        a ← b;
        loop: if a ≤ d
              then (e; a ← a+c; goto loop)
              else ';
```

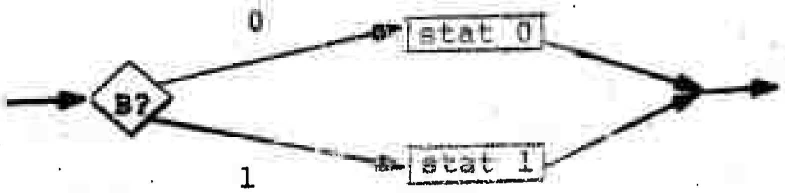
```
ALGOL                                    FLEX
for I: = 1 step 1 until 50 do   .  for  ('I', 1, 1, 50,
begin a [I] : = I + 5;                 'a [I] ← I+5; b[I+2] ← I')
      b [I + 2] : = I
end;
```

<u>Note</u>: a,b,c,d,e are the formal parameters. Enclosing an actual parameter in quotes is equivalent to the ALGOL "call by name". Unquoted actual parameters are equivalent to value calls. Nesting is obviously easy.

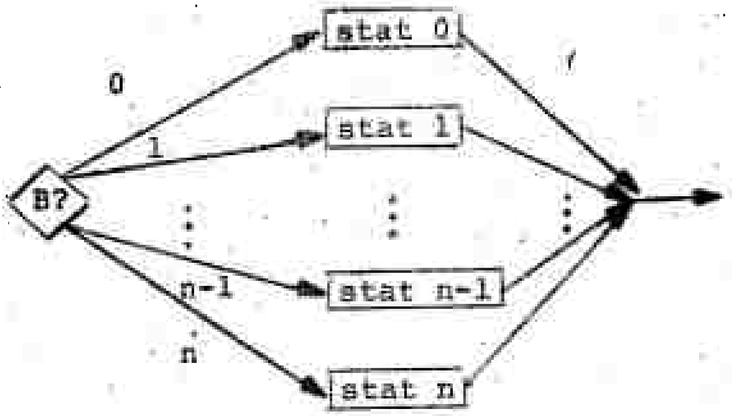
The "Case" Statement

One of the most useful concepts in programming is the protected branch. This is illustrated by the if statement:

if B then <stat>1 else <stat>0



This concept can easily be generalized to n branches:



It is quickly accomplished in FLEX by the following method:

```
    a ← ('<stat 0>', '<stat 1>', ···, '<stat n-1>', '<stat n>');
```

and used:

```
    a[B];
```

Parallel Processing

If, in the previous example, the entire vector was indicated instead of just one element:

```
    a;
```

then FLEX would execute the n statements in "parallel".

This is also a protected scheme. The global process is passive until all of the n statements are done. To create and release a process which will execute concurrently with the global process, the following is done:

```
    act for ('I', 1, 1, 50, 'b [I] ← I');
```

The for-loop will be executed in parallel with the statements following this call:

We may also do:

```
    a ← '('<stat 0>', '<stat 1>', ···,'<stat n-1>', '<stat n>')';
     act a;
```

The n statements will be executed in parallel with the global process and with themselves.

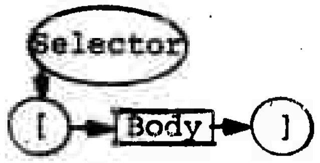
Coroutines

Another useful concept in programming is the coroutine which is simply a process description which allows a return from the middle of the code. The exit point is saved and, when the code is again called, control is transferred to the previous exit point rather than to the beginning.

The leave reserved identifier facilitates this feature-- it indicates to the event scheduler that the current process is to be passivated and the current program step saved.

7. The Selector

Syntax



<Selector>::= [<Body>]

Semantics

The construction [<Body>] is used to pass parameters to the access mechanisms of FLEX. It is used both with the FLEX mapping operators array and field, and with user-defined maps to select from some previously defined data structure.

Pragmatics

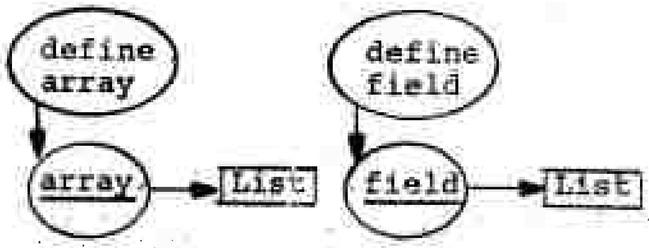
A data descriptor may be marked with the information that a segment is mapped. The map is executed to finally produce a data descriptor of the selected element.

Justification

The separation of structure and data is the prime consideration in any useful file system and allows great flexibility as well as the use of "stupid" channels.

System Mapping

Syntax



<define array>::=array List
<define field>::=field List

Semantics

a. Array

The first parameter is the byte size of the elements in bits. The following parameters describe the lower and upper bounds of each dimension of the array. A logical "procedure" is assigned to the map of the variable. The actual parameters are reconciled with the bounds when an access is requested to produce a descriptor or a value of the element selected.

b. Field

The operator produces a logical procedure which may be assigned to a variable just like any other quotation. The procedure operates on a descriptor describing a field of bits to produce a description of a new field. The first parameter is the offset in bits of the new field in relation to the old. The second parameter is the end bit of the new field.

Pragmatics

Both routines are called and executed like any other simple procedure.

Justification

These routines allow the user to extract an arbitrary sequence of bits from some other sequence of bits.

Examples:

```
a ← array (7, 1 to 10, -255 to 0) '''a is mapped as a
    two-dimensional array whose elements are 7 bits wide'''

a [5, -3] '''selection of a byte'''

a [5,] '''selection of a row'''

a [2 to 5, -10 to -2] '''selection of a new square array'''

id ← field (0 to 15); wages ← field (16 to 31);

son ← field (32 to 32 + 16); son 2 ← field (32 + 15 to 63);

'''this is a definition of the fields of a 64 bit wide

record.  a use follows'''

employee ← array (64, 1 to 1000); employee ← tape 2;

display ← wages (employee [3 to 15]),

display ← employee [son (employee [5])];
```

C. The SCRIBE Language

Introduction

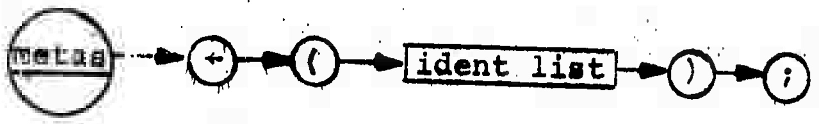
Although SCRIBE is a super-set of the core language FLEX, it is presented last with the feeling that some intuitive grasp of the language environment will have been achieved by now.

SCRIBE has its roots in the "Floyd-Evans production scheme" [4,5] and FSL [6,7]. Basically it is a bottom-up, bounded-context recognizer that uses FLEX as a sublanguage to express semantic relationships. Because of its bounded-context properties, it will deliver the canonical parse of any language (which may be expressed in this form) without backing up. This ability allows a one-pass compiler to be created simply and compactly; ideal attributes for inclusion in the hardware of a machine.

The Basic Elements of SCRIBE

There are four levels of description necessary for creating a language translator: meta declaration, terminal declaration, the syntax algorithm, and semantic relationships.

1. Meta Declaration



metas ← (<ident>);

Semantics

A meta symbol in SCRIBE is an <identifier> which has the same use as the symbols enclosed in <> in BNF; it is used for taxonomic purposes as a generic or class name for a certain construction. The meta symbols ident, delim, text are automatically included in any meta symbol list.

Pragmatics

The meta symbols are transliterated to unique integers which may be used in generating the canonical parse.

Justification

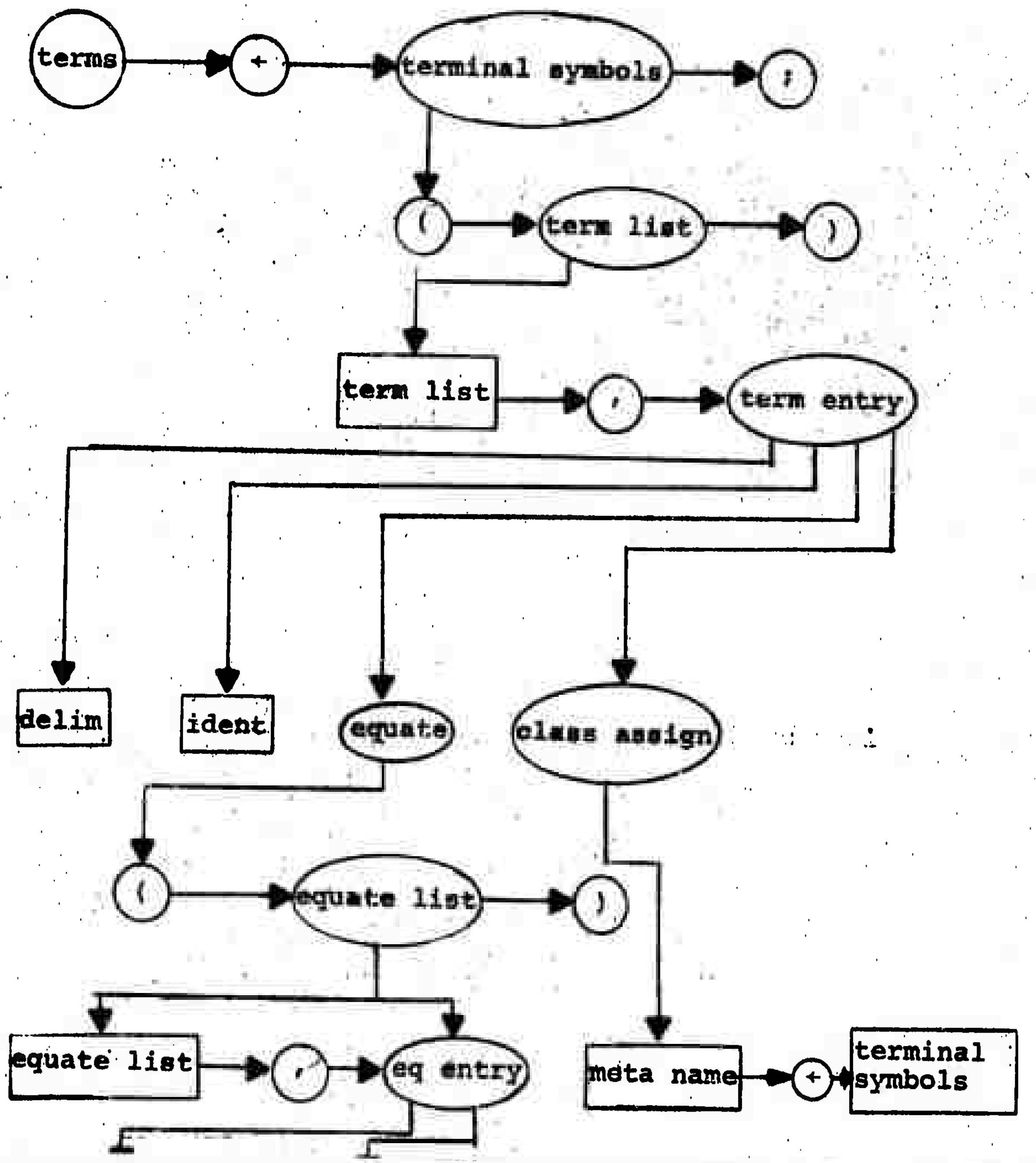
The use of meta symbols as class names is well-justified in phrase structure language theory.

Examples:

metas ← (aexp, term, factor, prim);

2. Terminal Declaration

Syntax



```
terms    (<term list>);
<term list>::= [<term list>,] <term entry>
<term entry>::= <delim> | <ident> | <equate> | <class
                                                assignment>
<equate>::= (<equate list>)
<equate list>::=  [<equate list>,] <eqentry>
<eqentry>::= <delim> | <ident>
<class assignment>::= <meta name> ←(<term list>)
```

__Semantics__

Terminal symbols are the syntactic atoms of a language. In theory they are treated as single characters but, because of limitations of character sets, aggregates of characters may also denote a terminal symbol. For example: "+" and "-" are terminal in FLEX and so also are __begin__ and __new__: identifiers whose meaning is reserved. SCRIBE allows terminal symbols to be declared either in the form of single character delimiters or as identifiers.

It may be that more than one representative for a terminal is desired for purposes of serving more than one character set or for clarification. __begin__ and "(" are an example from FLEX. Syntactically the two representations are equal and may be declared in SCRIBE as an equate: (begin, "(").

Many terminal symbols may belong to the same syntactic class and an ability to assign them to a meta symbol can save a great deal of effort in writing the syntactic algorithm. The multiply operators provide an example; they are usually assigned the same level of precedence and this fact can be indicated in SCRIBE by the class assignment: mop ← ("*","/","÷"). This provides an abbreviation or "parse name" for the three delimeters.

In fact, every terminal symbol can be considered to have both an "external name", which is the character itself, and a "parse name", which is either the same as the external name or is a meta symbol indicating membership in a class. All comparisons in the syntax section are done on the parse name.

Pragmatics

A table is built from the indicated relationships so that the textual scanner may separate the terminal symbols and discover their external and parse names.

Justification

The terminal declaration supplies a finite state algorithm in the machine with enough information to completely strip down the text into primary syntactic atoms which is typically the dirtiest job in compiling.

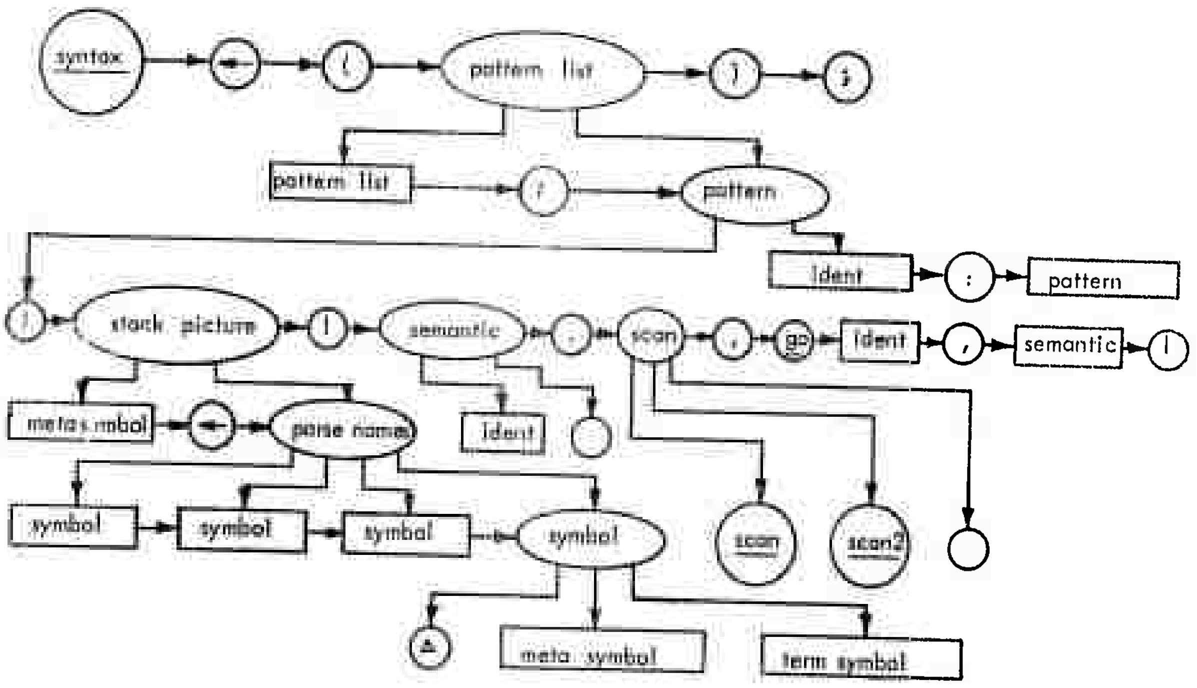
Examples:

'''from both sections--to declare all symbols necessary for handling arithmetic expressions'''

```
metas ← (aexp, term, fact, prim, aop, mop);

terms ← ( "(", ")", ("↑", "exp"), mop ← ("*", "/", "÷"),
  aop ← ("+", "-"));
```

3. The Syntax Algarithm

Syntax

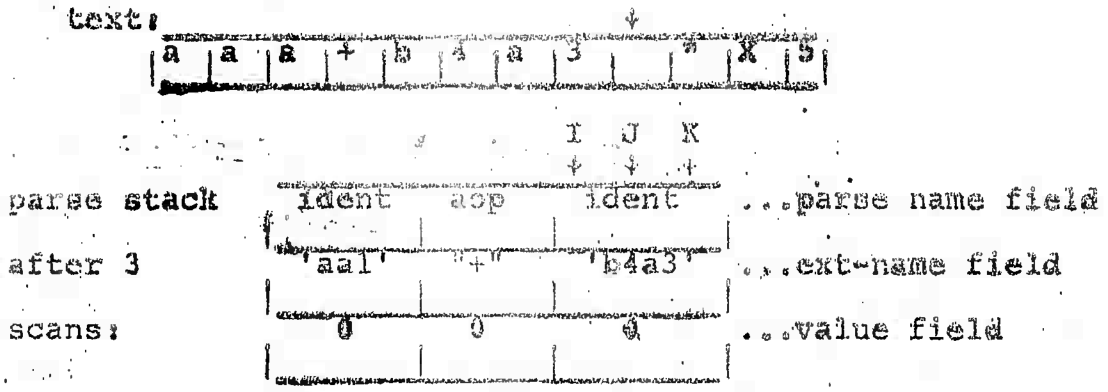


```
syntax ← (⟨pattern list⟩ );

⟨pattern list⟩ :: = {⟨pattern list⟩ ;} ⟨pattern⟩
⟨pattern⟩   :: =  "|" ⟨stack pictures⟩ "|"  ⟨semantic⟩ ,⟨scan⟩  ,  go ⟨ident⟩, ⟨semantic⟩ "| "
                  "| " ⟨ident⟩  :  ⟨pattern⟩

⟨stack picture⟩  :: = {⟨meta symbol⟩ ←} ⟨parse names⟩
⟨parse names⟩   :: = {{{⟨symbol⟩} ⟨symbol⟩} ⟨symbol⟩} ⟨symbol⟩
⟨symbol⟩        :: =   △ |⟨meta symbol⟩  | ⟨term symbol⟩
⟨semantic⟩      :: =  ⟨ident⟩   | ⟨empty⟩
⟨scan⟩         :: =  scan  |  scan 2  |  ⟨empty⟩
```

Semantics

When a scan command is issued, the text at the current point is scrutinized and a terminal symbol is isolated. This is looked up in the table that was created by the terms declaration and the parse name and external of the symbol is pushed into the parse stack.

Example:



The parse-name field in the stack contains the class names for the two identifiers and the "+". The value field which is used by the semantic processor is blank.

A pattern to recognize this configuration would be:

```
|aexp ← ident aop ident | sum, scan, go axp, err 1 |;
```

The section "ident aop ident" asks if that configuration is present in the stack. It is, so the semantic routine "sum" is executed. This will be a routine written in FLEX and defined in the next section. It will be able to use the pointers I and J which after a successful pattern match are set to the lower and upper bounds of the pattern in the stack.

(K always points

	I ↓		J ↓ K ↓	
parse stack	ident	aop	ident	...
after a	'aal'	"+"	'b4a3'	...
match:	0	0	0	...

to the top of the stack).

After the return from the semantic routine, the first section is examined to see whether or not a reduction is requested. "aexp+" is present so the region of the stack between I and J will be replaced with "aexp" in location I.

	I ↓ J ↓ K ↓	
parse stack	aexp	...
after the	'aal'	...
reduction:	0	...

Now the <scan> field is examined. A single scan is requested and executed.

text:	a	a	a	1	+	b	4	a	3		*		X	5

parse stack	aexp	mop
after scan:	'aal'	"*"
	0	0

Lastly, the go is executed and control is passed to another pattern - in this case, the pattern labeled "axp".

If the pattern match had not been successful, then the last field would have been examined. If <empty> control would be transferred to the next sequential pattern. If an <ident> is present, then the semantic routine named by the ident will be called--for this case it would be "errl".

A "Δ" will always be accepted in the match.

<u>Pragmatics</u>

The handling of text, parse stack and patterns is accomplished by a compact "wired-in" algorithm. The patterns themselves require only 64 bits apiece.

<u>Justification</u>

The algorithmic form of the syntax handler though somewhat removed from the phrase structure descriptive method allows the user much more knowledge of what is going on at each point and thus makes for a very compact description of a language. It is this feature which allows the tables for FLEX and SCRIBE to be implemented in hardware.

<u>example</u> ''' combining the above three sections to form a recognizer for arithmatic expressions'''

<u>metas</u> ← (aexp, term, factor, primary, aop, mop);

<u>terms</u> ← ("(",")",("↑","exp"), mop←("*","/"," "), aop←("+"," "));

<u>syntax</u>← (

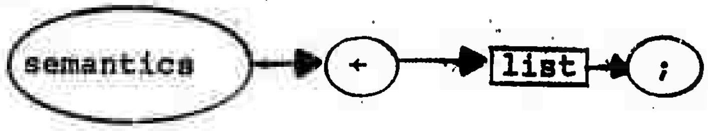
```
start:  |                                  Δ     |          ,scan   ,go atoms   ,          | ;
atoms:  |                                 "("    |          ,scan   ,go atoms   ,          | ;
        |                                 aop    |          ,scan   ,go atom 1  ,          | ;
atom 1: | primary ←                      ident   |  idt     ,scan   ,go prim    ,          | ;
        | primary ←                       num    |  nm      ,scan   ,go prim    , err[1]   | ;
prim:   | factor←factor "↑"   primary      Δ     |  opr     ,       ,go fact    ,          | ;
        | factor ←            primary      Δ     |          ,       ,go fact    ,          | ;
fact:   |                     factor      "↑"    |          ,scan   ,go atoms   ,          | ;
        | term ← term mop     factor       Δ     |  opr     ,       ,go term    ,          | ;
        | term ←              factor       Δ     |          ,       ,go term    ,          | ;
term:   |                     term        mop    |          ,scan   ,go atoms,             | ;
        | aexp ← aexp aop     term         Δ     |  opr     ,scan   ,go aexp    ,          | ;
        | aexp ←      aop     term         Δ     |  unsum   ,       ,go aexp    ,          | ;
        | aexp ←              term         Δ     |          ,       ,go aexp    ,          | ;
aexp:   |                     aexp        aop    |          ,scan   ,go atoms   ,          | ;
        | primary ←  "("      aexp        ")"    |          ,scan   ,go prim    ,          | ;
        |                     aexp         Δ     |          ,       ,go halt    , err[2]   |);
```

'''Notice the similarity between the above and the phase structure definition for arithmetic expressions:'''

```
<aexp>::=<aexp> <aop> <term>
        | <aop> <term>
        | <term>
<term>::=<term> <mop> <factor>
        | <factor>
<factor>::=<factor>↑<primary>
        | <factor>
<primary>::= ident | num | (<aexp>)
<aop>::= + | -
<mop>::= * | / | ÷
```

4. Semantic Relationships

Syntax



semantics ← <list>;

Semantics

All identifiers used in the semantic fields are considered to be global to the <list>, as are the various system routines to aid the compiler writer. The identifiers must be defined by a quotation assignment in the <list>. Additional content in the <list> is left to the programmer's discretion. The system aids will be described separately.

Pragmatics

Essentially, the semantics are in the form of a <case> statement with each <case> being one of the identifiers found in the semantic fields of the patterns.

Justification

The use of FLEX as a powerful descriptive language for providing semantic referrents to the maching allows translation building to be relatively easy.

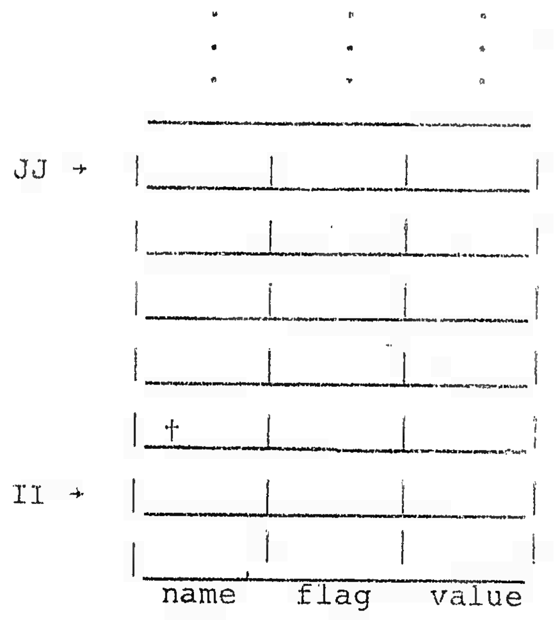
Examples

(Will be given after exposition of system aids)

Global Data Structures and Algorithms

A. The Symbol Table

The symbol table is a stack in which information may be retained about <identifiers> in the system. Automatic controls for handling block structure are provided.



Symbol Table Routines

New Block

II and JJ delimit a block of symbols. New block causes a push to occur and II and JJ are reset to handle a new group of symbols.

Old Block

A pop of the symbols delimited by II and JJ is performed and II, JJ are reset to their lower level values.

find (name, from, to, found)

A search is performed in the range specified. "found" is set to true and a global variable LL contains the desired location if a match occurs. Otherwise "found" is set to false.

enter (name, flag, value, enor)

A search is performed in the current block. If a match is not made, then the "name, flag, value" indicated are pushed into the current block. If a match was made denoting that a symbol with the same name already exists in the current block, "error" is set to true and no entry is made.

B. Code Generation

System aids in this area are currently somewhat primitive. A canonical parse will deliver operators and operands to the semantic routines in a polish post fix order--all that need be done is to generate the two kinds of operators that the FLEX polish requires. The following routines will eventually be replaced with machine independent aids.

sop (op)

In the FLEX machine simple operators are identical with their delimiter representation. sop ("+") will generate an "add" command.

cop (op, value)

Compound operators are necessary for specifying declarations, variables, procedure calls, etc. Their description and use is supplied in Section V on pragmatics of the machine. The user's intuition will be relied on.

<u>strval</u> (loc, value)

Stores just into the value field of a compound operator at the specified location.

<u>strop</u> (loc, op)

Stores just into the value field of a compound operator at the specified location.

<u>Global names</u>

"cpd" is the current process description segment into which code is being generated. "P" is the current code location.

```
Examples:    ''' a simple compiler for arithmetic assignment statements delimited
                 by " # " '''

metas ← (body, stat, aexp, term, factor, primary, aop, mop);
terms ← (" ; ", "( , )", "  ", "   ", mop    (" * ", "/", "   "),
            aop    (" + ",  " - " ) );

          Syntax  ← ( |                        Δ  |              , scan     go start,           | ;
Start: |  Body  ←                           " # " |  Set up      , scan 2   go on      err 1    | ;
On:    |                            ident  "← "   |              , scan     go atoms   err 2    | ;
Atoms: |                                   "("    |              , scan     , go atoms ,        | ;
       |                                   aop    |              , scan     , go atom 1 ,       | ;
Atom 1:| primary ←                         ident  |  idt         , scan     , go prim  ,        | ;
       | primary ←                         num    |  nm          , scan     , go prim  , err1   | ;
prim:  | factor  factor "↑"   primary      Δ      |  opr         ,          , go fact  ,        | ;
       | factor ←             primary      Δ      |              ,          , go fact  ,        | ;
fact:  |                      factor       "↑"    |              , scan     , go atoms ,        | ;
       | term ← term mop      factor       Δ      |  opr         ,          , go term  ,        | ;
       | term ←               factor       Δ      |              ,          , go term  ,        | ;
term:  |                      term         mop    |              , scan     , go atoms,         | ;
       | aexp ← aexp aop      term         Δ      |  opr         , scan     , go aexp  ,        | ;
       | aexp ←       aop     term         Δ      |  unsum       ,          , go aexp  ,        | ;
       | aexp ←               term         Δ      |              ,          , go aexp  ,        | ;
aexp:  |                      aexp         aop    |              , scan     , go atoms ,        | ;
       | primary ←   "("      aexp         ")"    |              , scan     , go prim  ,        | ;
       | stat ← ident  "← " aexp           Δ      |    assign    ,          , go fold,   err 3  | ;
Fold:  | body ←         body  stat  "; "          |              , scan 2   , go arr,           | ;
       | body ←         body  stat  "# "          |              ,          , go half,   err 4  |);

     Semantics ← (
     set up ← 'new biock; cop ("new", Ω) '''set up for handling variables''' ';
     idt  ← 'enter (ext (I), JJ+l, err )''' if name not there, put in ''';
             'cop ("value call", value (LL)) ''' generate a fetch request ''' ';
     nm  ← 'numb (ext ( I )) '''generate a literal for a number''' ' ;
     op  ← 'sop (ext (I+l) '''pick up and output operator from external name ''' ';
     unsum ← 'if ext (I) = "-" then sop (un min) else' '''unary minus'''
     assign ← 'enter (ext (i), JJ+l, error) '''if name not there put in''';
             cop ("name call", value (LL)) '''generate an address request''';
             sop ("← ") '''generate a store command''' ');
```

'''The compiler for FLEX itself is a good example'''

```
                    flex←scribe  "      metas ← (body, list, stat, selr, prim, labl, svar, var,
                                          expr, factor, term, arit, rexp, rterm, aterm,
                                          and1, oterm, or1, bool, assert, utm, uex, ifcl
                                          trupart, aop, mop, rel, lop, asop, bup, unop,
                                          iter, sop, set);

                                       terms ← (",",";",(begin, "("), (end, ")"), "‘", "’",
                                          "[", "]", ":", new, if, then, else, "#","''",".",
                                          com, unop←(scribe, type, ( "⌈", ceil), ("⌋", floor),
                                          sin, cos, atan, "¬", abs, rand, prand, hash, exp,
                                          ln, sart, length), "↑", aop←("+", "-"), mop←
                                          ("*", "/", "÷", mod), rel←("=","≠", "<", "≤", ">",
                                          "≥"), "∧", "∨",  lop←("⊁", "≡", "⊅"), sop←
                                          ("∩", "∪", "⊂"), asop←(is, isn, "∈", "∉"), of,
                                          while, to, by, do, "∞", "Ω", tops←(bop, uop,
                                          val, map), ident←(array, field, act, leave, term,
                                          xin, yin, plst, plpt, plln, control), (goto, go), ("?",
                                          "any"), ar←("←", as ));


             syntax← ('' patterns...          |  semantic field, scan field, jump field ,  error field |'''
             |                      △         |                   , scan      , go head  ,              |;
      head:  | body ←               "‘"       |  quot             , scan      , go bod   ,    err1      |;
      bod:   | body ←           body new      |  new              , scan2     , go decl  ,              |;
      empt:  | body ←           body ";"      |                   , scan      , go empt  ,              |;
             | body ←           body ","      |  com1             , scan      , go empt  ,              |;
             | body ←           body end      |  end1             , scan      , go lis   ,              |;
             | stat ←           body "’"      |  equot1           , scan      , go sta   ,              |;
             | selr ←           body "]"      |  esel1            , scan      , go sel   ,              |;
```

```
atoms: | body ←                     "("    | quot   ,scon    ,go bod,          |;
       | body ←                     begin  |        ,scan    ,go bod,          |;
       | prim ←                     lit    | push1  ,scan    ,go pri,          |;
       | prim ←                     "Ω"    | push1  ,scan    ,go pri,          |;
       | prim ←                     "∞"    | push1  ,scan    ,go pri,          |;
       | prim ←                     "?"    | push1  ,scan    ,go pri,          |;
       |                            if     |        ,scan    ,go atams,        |;
       |                            while  |        ,scan    ,go atoms,        |;
       |                            com    |        ,scan    ,go atoms,        |;
       |                            unop   |        ,scan    ,go atoms,        |;
       |                            αop    |        ,scan    ,ga atams,        |;
       |                            goto   |        ,scan    ,go atoms,        |;
       |                            △      |        ,scan    ,go next,         |;
next:  | lobl ←             ident   ":"    | label  ,scon    ,go otoms,        |;
       | svar ←             ident   △      | nops   ,        ,go sur,          |;
       | svar ←             tops    ident  | tps    ,scon    ,go sur,    err2  |;
decl:  | body ←      body   ident   ","    | decl1  ,scon2   ,go decl,         |;
       | body ←      bady   ident   ";"    | decl2  ,scan    ,go empt,   err3  |;
svr:   | body ←                     begin  |        ,scan    ,go bod,          |;
mbr:   | body ←                     "["    |        ,scan    ,go bod,          |;
       | vor  ←             svar    △      |        ,        ,go vr,           |;
       |                    var     ar     |        ,scan    ,go atoms,        |;
       | prim ←             var     △      |        ,        ,go pri,          |;
pri:   | prim ←      unop   prim    △      | unop   ,        ,go pri,          |;
       | expr ← prim com    prim    △      | pop1   ,        ,go exp,          |;
       | expr ←      com    prim    △      | uncom  ,        ,go exp,          |;
       |                    prim    com    |        ,scan    ,go otoms,        |;
```

```
      |                       prim   "↑"    |         , scan   , go  atoms  ,l;
      | prim ← prim    "↑"   prim    △      |  pop1   ,        , go  pr1    ,l;
      | factor ←             prim    △      |         ,        , go  fact   ,l;
fact: | term ← term    mop   factor  △      |  pop1   ,        , go  trm    ,l;
      | term ←               factor  △      |         ,        , go  trm    ,l;
trm:  |                      term    mop    |         , scan   , go  atoms  ,l;
      | arit ← arit    aop   term    △      |  pop 1  ,        , go  art    ,l;
      | arit ←         aop   term    △      |  unmin  ,        , go  art    ,l;
      | arit ←               term    △      |         ,        , go  art    ,l;
art   |                      arit    aop    |         , scan   , go  atoms  ,l;
      | rexp ← rterm   rel   arit    △      |  pop1   ,        , go  rxp    ,l;
      | rterm ←              arit    △      |         ,        , go  rtm    ,l;
rtm:  |                      rterm   rel    |         , scan   , go  atoms  ,l;
      | rexp ←               rterm   △      |         ,        , go  rxp    ,l;
rxp:  | aterm ←              rxp     "∧"    |  mark   , scan   , go  atoms  ,l;
      | and1 ←               rxp     △      |  fill   ,        , go  an1    ,l;
an1:  | and1 ←         aterm and 1   △      |         ,        , go  an1    ,l;
      | aterm ←              and 1   "∨"    |  mark   , scan   , go  atoms  ,l;
      | or1 ←                and 1   △      |  fill   ,        , go  O1     ,l;
O1:   | or1 ←          aterm or  1   △      |         ,        , go  O1     ,l;
      | bool ←         bool  lop or 1 △     |  pop1   ,        , go  bl     ,l;
      | bool ←                 or 1  △      |         ,        , go  bl     ,l;
bl:   |                      bool    lop    |         , scan   , go  atoms  ,l;
      | set ← set      sop   bool    △      |  pop1   ,        , go  st     ,l;
      | set ←                bool    △      |         ,        , go  st     ,l;
st:   |                      set     sop    |         , scan   , go  atoms  ,l;
      |                      set     asop   |         , scan   , go  atoms  ,l;
      | assert ← set   asop  set     △      |  asop   ,        , go  ast    ,l;
      | utm←assert of  set    Δ      |         ,       ,go   ut     ,       |;
      | utm←           set    Δ      |         ,       ,go   ut     ,       |;
ast:  |           assert      of     |         , scan  ,go   atoms  , err4  |;
ut:   | uex←uex bup    utm    Δ      | pop 1   ,       ,go   ve     ,       |;
      | uex←           utm    Δ      |         ,       ,go   ve     ,       |;
ve:   |                vex    bop    |         , scan  ,go   atoms  ,       |;
      | expr←expr "#"  vex    Δ      | pop 1   ,       ,go   exp    ,       |;
      | expr←          vex    Δ      |         ,       ,go   exp    ,       |;
exp:  |                expr   "#"    |         , scan  ,go   atoms  ,       |;
      | stat←          expr   Δ      |         ,       ,go   sta    , err5  |;
sta:  | stat←var   ar  stat   Δ      | pop 1   ,       ,go   sta    ,       |;
      | stat←     iabl stat   Δ      |         ,       ,go   sta    ,       |;
      | stat←  trupart stat   Δ      | fill    ,       ,go   sta    ,       |;
      | stat←     gots stat   Δ      | unop    ,       ,go   sta    ,       |;
      | stat←     iter stat   Δ      | itr     ,       ,go   sta    ,       |;
      | if cl ←   if   stat   then   | mark    , scan  ,go   atoms  ,       |;
      | trupart ← if cl stat  else   | els     , scan  ,go   atoms  ,       |;
      | iter ←   while stat   do     | whl     , scan  ,go   atoms  ,       |;
      | body ←    body stat   ","    | com 2   , scan  ,go   empt   ,       |;
      | body ←    body stat   ";"    | cln 2   , scan  ,go   empt   ,       |;
      | list ←    body stat   end    | end     , scan  ,go   lis    ,       |;
      | stat ←    body stat   "⁆"    | equot   , scan  ,go   sta    ,       |;
      | sclr ←    body stat   "]"    | esel    , scan  ,go   sel    , err6  |;
lis:  | svar ←    svar list   Δ      |         ,       ,go   sur    ,       |;
      | svar ←         list   Δ      |         ,       ,go   mbr    ,       |;
sel:  | svar ←    svar scln   Δ      |         ,       ,go   svr    , err7  |);
```

IV. The User's Environment

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 the powerful editor which controls a free-running compiler that is translating everything that is entered through the keyboard to FLEX code. Since his process is also declared active, the pragmatic system will attempt to execute all produced code. This will appear to the user as though his commands at this lowest level are being executed statement by statement.

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 <u>leave</u>. This is the coroutine exit command and, since the routine which called him is the process scheduler itself, his process is <u>passivated</u> 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.

<u>Scope of the User</u>

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.

V. The Pragmatic Environment

1. Introduction

In this chapter 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.

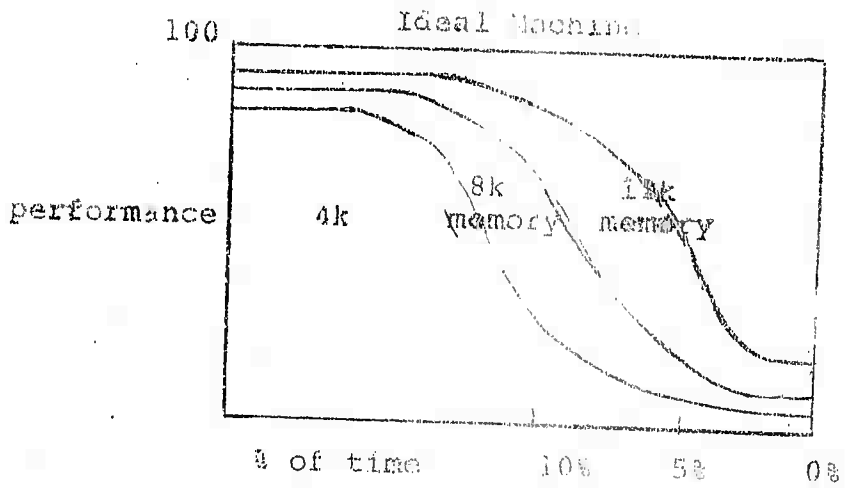
Second, a statistical viewpoint is adopted. For almost all computer problems in general on any machine (and in particular those problems for which the object machine is suited) neither 100% efficiency 100% of the time nor blinding speed is necessary. Fortunately from the software point of view, the first 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 clog the system.

2. Segment and Process Control

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 16-bit words in core memory and secondary storage whose length may be changed with varying degrees of effort.

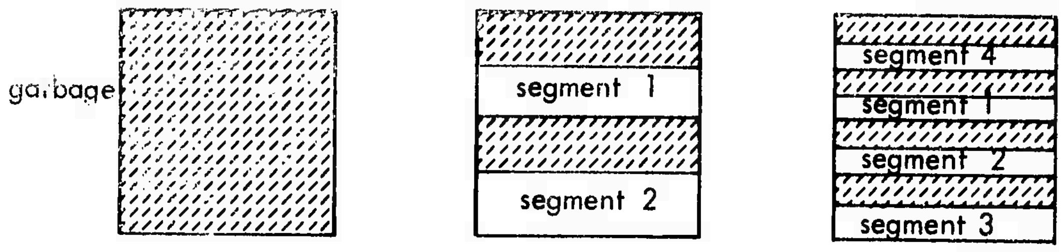
Addressing in the system is relative to the segment not to any particular memory location so that a particular segment may be moved anywhere without disturbing access to it.

segment	displacement

Typical Segment Address

Core Memory

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.



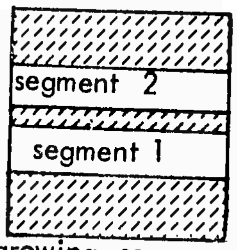
Segment Allocation in Core Storage

At this point paging should be considered. With paging, the logical entity (variable length segments) would 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, outweigh the advantages for the address path is more complicated requiring two table lookups rather than one.

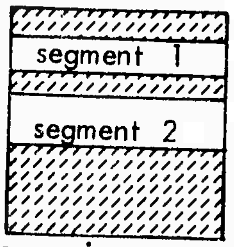
segment	page	displacement

Typical Paged Address

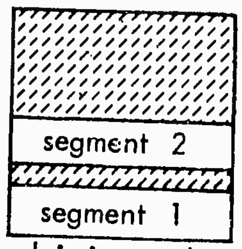
In keeping with the strategy of optimizing most used operations (and accessing memory is certainly at the top of the list), while allowing a certain amount of dirty work a small percentage of the time, paging is rejected as too expensive for every access. Accordingly, segments are mapped contiguously and memory must be reordered when one segment threatens to overrun another.



growing segmen



remapping segment



reclaiming garbage

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 usable 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.

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. Once a segment is swapped out it will remain in the secondary memory until an access is requested.

We now have the same problem that was presented to us in core: garbage must be found to accommodate the segment being swapped in. Again if no garbage can be found (or made), it must be created by first swapping out one or more active segments. Then the access-request may be swapped in.

The realization of these algorithms will be presented after process control is discussed since both operations are intertwined.

b. Process Control

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

Definition of Terms

A process description is a segment that contains executable code generated by the compiler. 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.

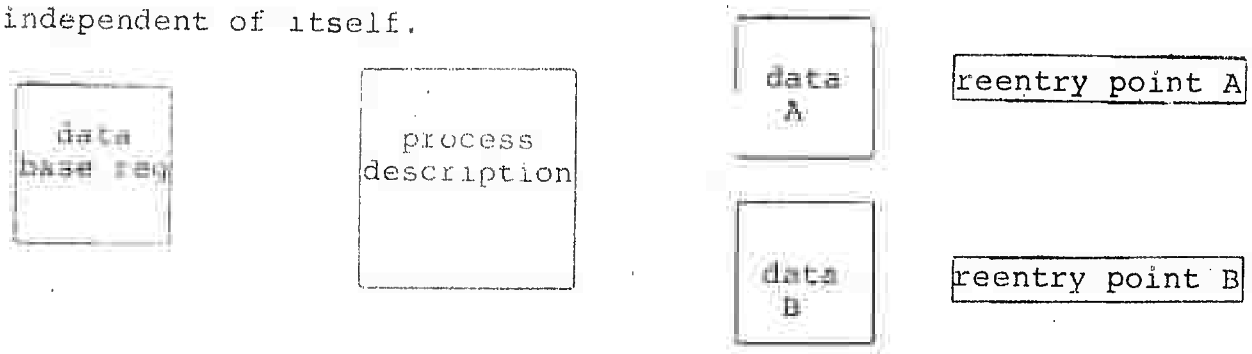
Parallel processes are required for operation of the system. The I/O, the display, the keyboard, the compiler, etc...must be able to run concurrently with the user's programs and with themselves. Moreover, the compiler which is interacting with the user at the keyboard may have to run in parallel with a logical "copy" of itself 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 MM-8000.

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.



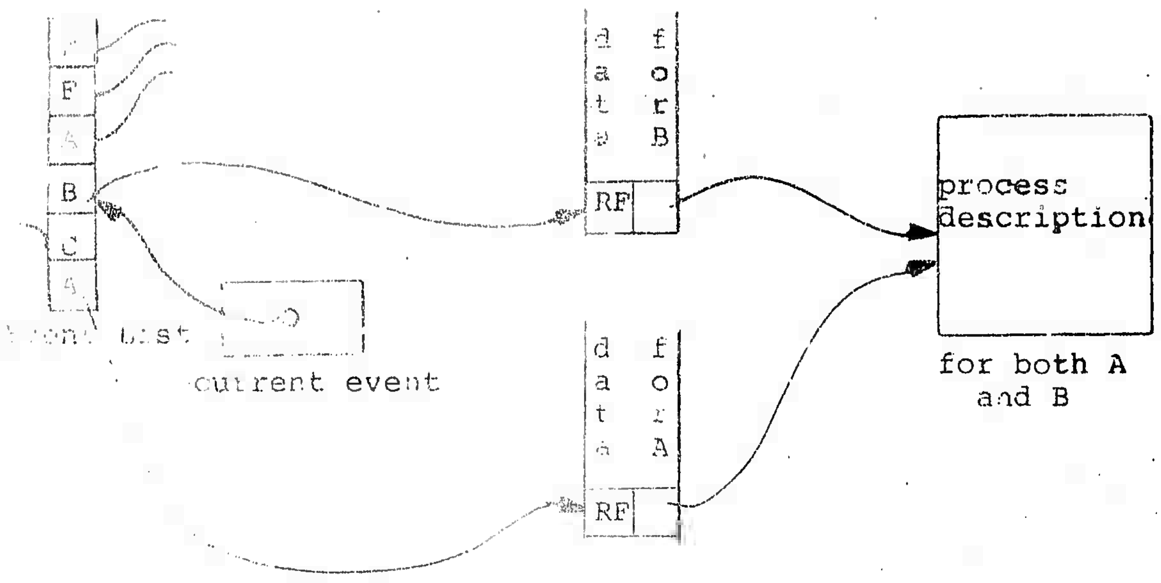
Process Description with Two Sets of Data

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 contain 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 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 becomesa 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.



FLEX Process Control

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 be swapped.

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.

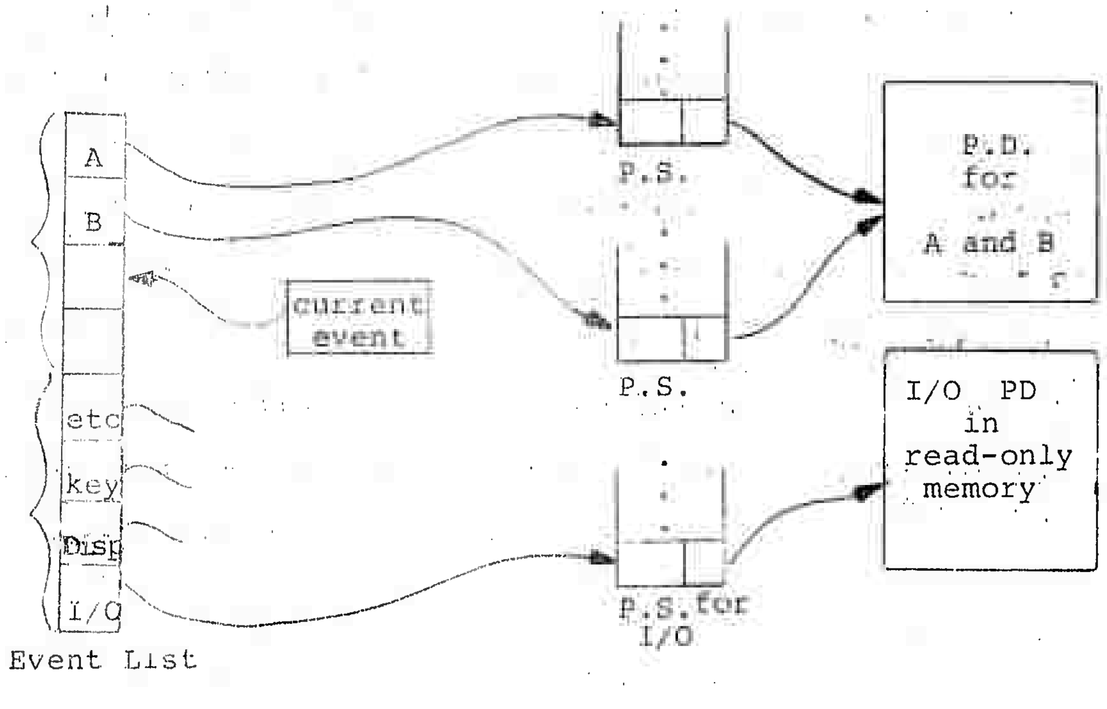
While B is having an event, the other processes are said to be passive. When the process stack is collapsed and the process name is removed from the event list, the process is said to be terminated.

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.

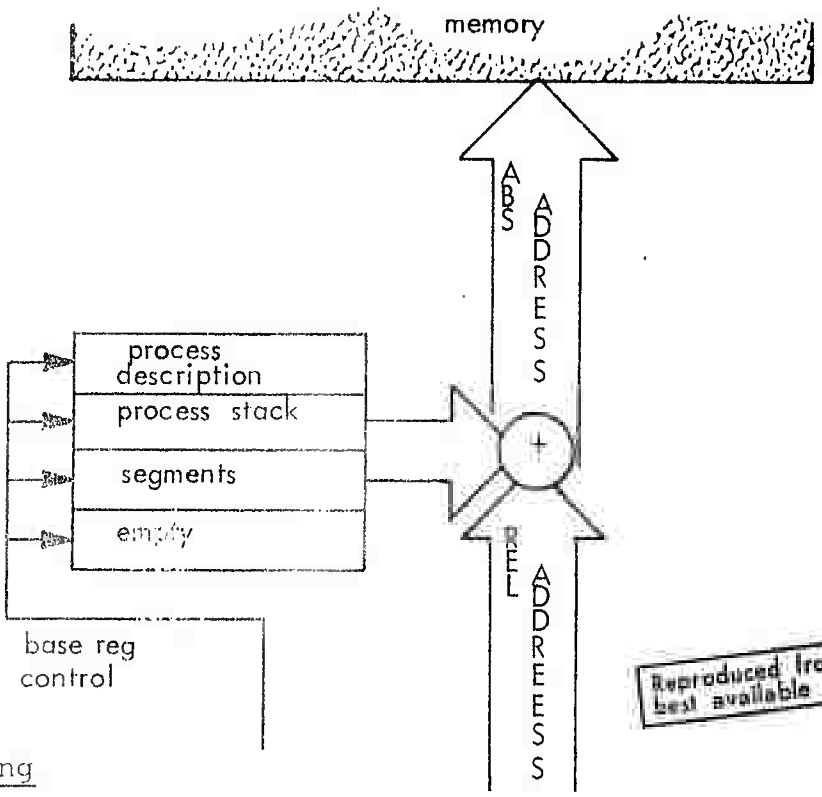


All the real-time event notices are logically clustered. Between each event they are scanned by the scheduler to see if activation is required. If not, the process pointed to by the current-event pointer is activated. At any time an outside interrupt may point into the real-time area. When this happens,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

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 short add whose time will be absorbed by the micro code hardware.

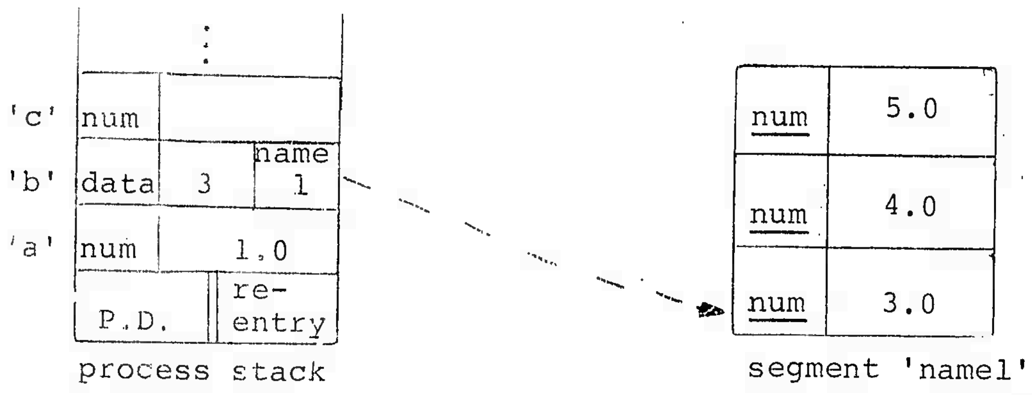
Data Segments Associated with a Process Stack

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 <u>descriptor</u>).

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 effe- tively a self-typed indirect address.

Example:

```
    'new 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'. Or course, the data descriptor in "I" 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.

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: loc is base address of seg-name;

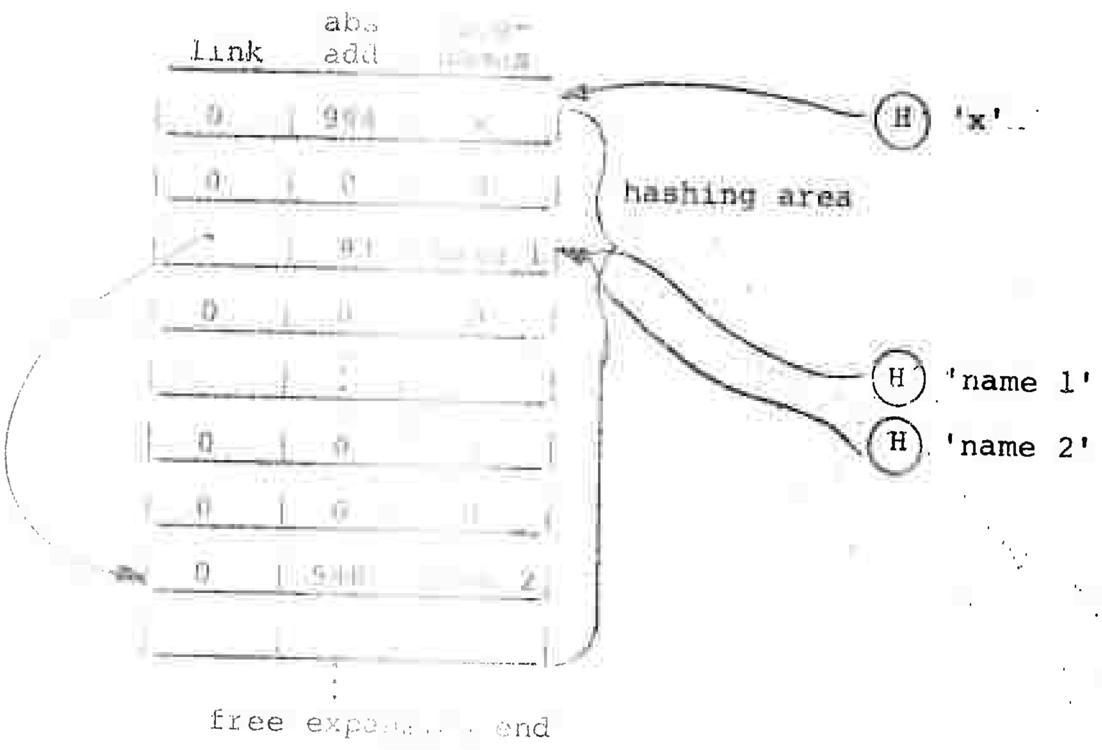
The inverse operation needs to be performed when the segment is accessed: ? is base-address of seg-name;

This will return the absolute location of the segment which will be placed in the fourth base register and then used. When a segment is swapped out or destroyed, the association needs to be removed: ? isn base-address of seg-name;

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 great 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 hardw[illegible] prohibitively expensive, so it cannot be used to [illegible] the information. Feldman [7] and others [8] have sh[illegible] hashing may be used to stimulate an associate [illegible] very effectively. The method used is similar to that [illegible]AP [9], although the hashing technique is derived [illegible] different source.

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 we have uniqueness. If we do (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 2 650 µs memory cycles. That for a fault is 1 or 2 650 µs 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.

The new name may then be entered into the segment table along with the absolute address of the displayed garbage and segment creation is accomplished.

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.

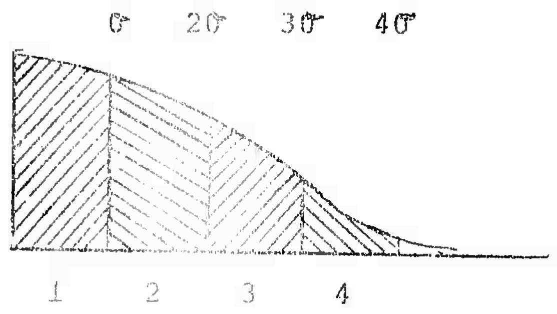
The influence of the compiler extends directly down to the lowest level of the machine and provides information that is not commonly available on other machines. Some of this information has to do with an insight into the use to which each segment will be put which may be partially derived from mapping 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 <u>yes</u> answer to the question: "Should this segment be swapped?" is:

$$\frac{\text{area of a number}}{\text{total area of a curve}}$$

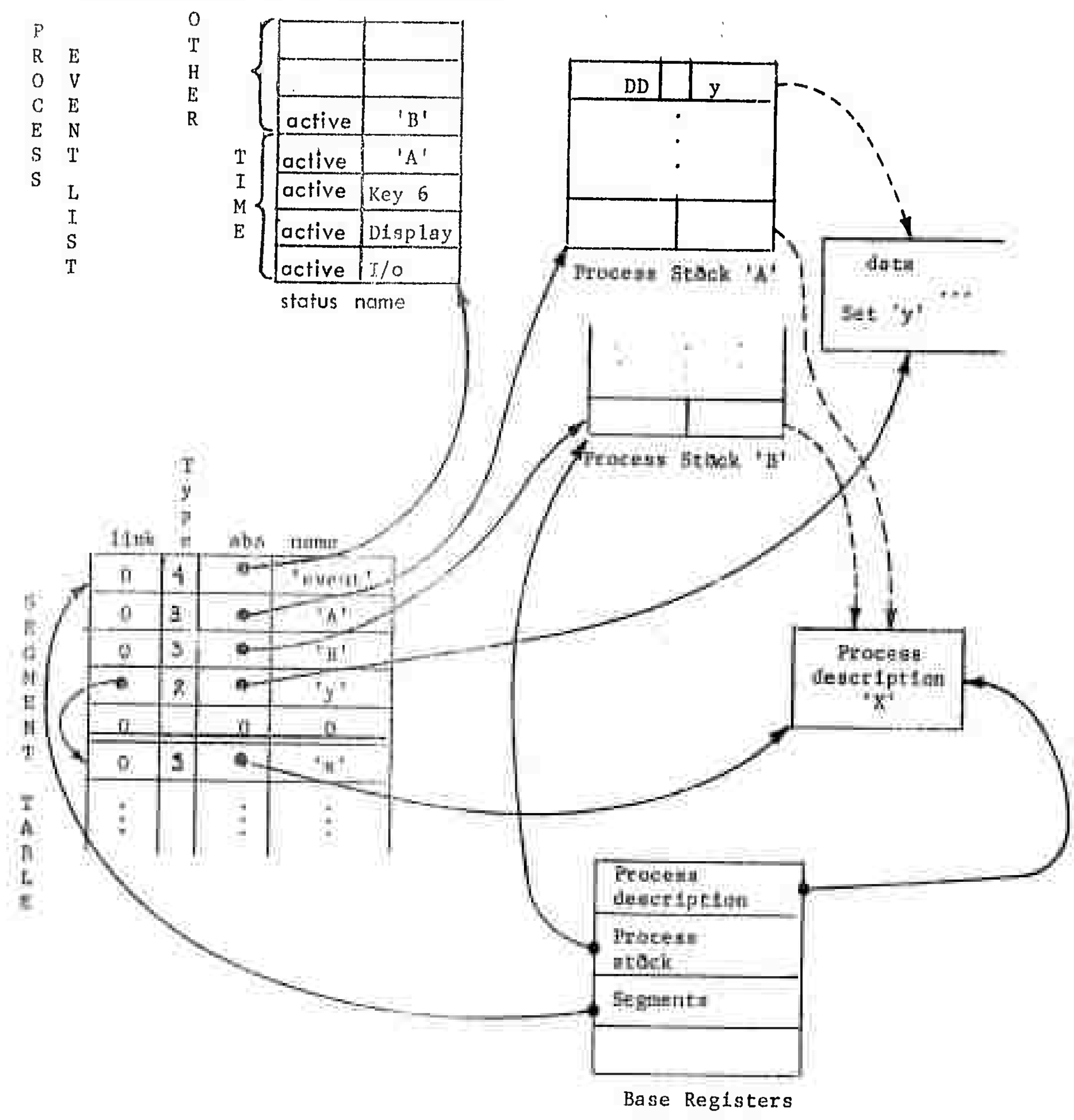
If the segment was text, then it would have a 67% probability of being swapped and a corresponding 33% chance of staying in. A system segment (having priority 4) will have a zero chance of being swapped.

Suppose now that a segment is in heavy use and is swapped. Then it will come right back in--but the chance for it 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.

The Strategy For Segment and Process Control

It now remains to put everything together.



The dotted lines are effective pointers (meaning that the reference really has to go through the segment table. An event for 'A' is taking place as shown by the base register configuration.

[illegible] Execution

1. 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.

```
 1     7                         1    9          11
 0 | op code                     1 | op code | value |
 A simple operator               A compound operator
```

The first bit in each phrase determines the length to be either 8 bit's or 16 bits. All the FLEX operators are compiled as simple operators. The names, requests for [illegible] jumps, etc. are compound operators.

The compound operators

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	displacement	Jump unconditionally
de	number of var	create space
par	number of parameters	to match actual with formal parameters
commas	vector size	create a new segment 'vector size' long
literal	integer	create a stack number from 'integer'
label	[illegible] address	used for operand for goto's

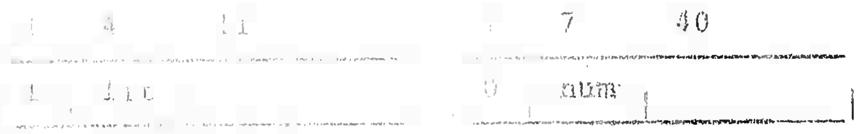
The coding is very compact. To add two numbers, it takes:

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.

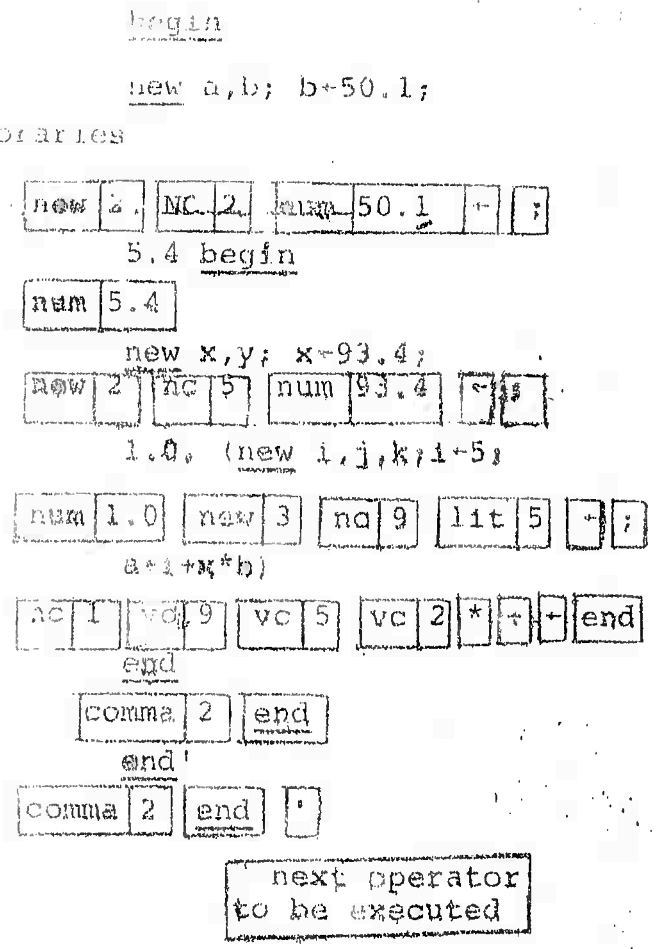


Numeric Literals

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 post-fix form. Execution is done serially except for jumps. Declarations and intermediate results are stored in the process stack which has the following logical form:



```
begin

new a,b; b←50.1;

|new|2.| |NC.|2.| |num|50.1| |←| |;|
        5.4 begin
|num|5.4|
        new x,y; x←93.4;
|new|2| |nc|5| |num|93.4| |←||;|
        1.0, (new i,j,k;i←5;
|num|1.0| |new|3| |na|9| |lit|5| |←||;|
        a+i+x*b)
|nc|1| |vc|9| |vc|5| |vc|2| |*| |+| |+| |end|
        end
    |comma|2| |end|
        end'
|comma|2| |end| |'|
```

28 16-bit words
of code generated

[illegible] form of FLEX is [illegible] [illegible], the compiler is able [illegible] where in the [illegible] the declarations will fall [illegible]. This means all that is needed [illegible] a variable name in the code is an index from [illegible].

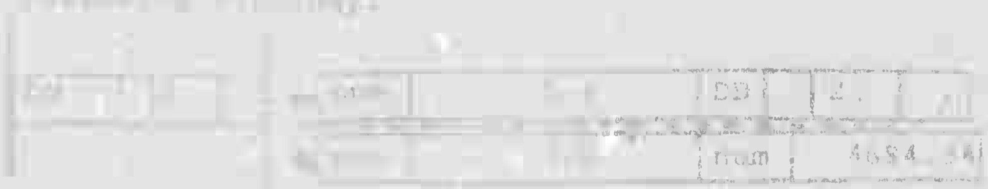
Reproduced from best available copy.

[illegible]

[illegible] declaration section consisting of [illegible] and is topped off by a pointer [illegible]. This mechanism is used by [illegible] a deallocation of the [illegible]. As the stack is collapsed, temporarily [illegible] by descriptors are also [illegible] many variables are [illegible] reaches zero, the segment [illegible].

In the statement [illegible], the name of x is referenced by [illegible] allocated in the 5th location of the [illegible] the bottom.

[illegible] list structure with the following [illegible]:



[illegible]

<u>Commas</u>

A <u>comma</u> 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.

<u>Processes</u>

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

```
                a(x+y, b, c, 1);
|mc | 'a'| |vc | 'x'| |vc | 'y'| |+| |vc | 'b'| |vc | 'c'|
|lit | 1| |call | 4|
```

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

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 <u>not passivated</u>. A <u>return</u> from the new process will terminate the new process while a <u>leave</u> will simply passivate it until the next time around.

Recursion

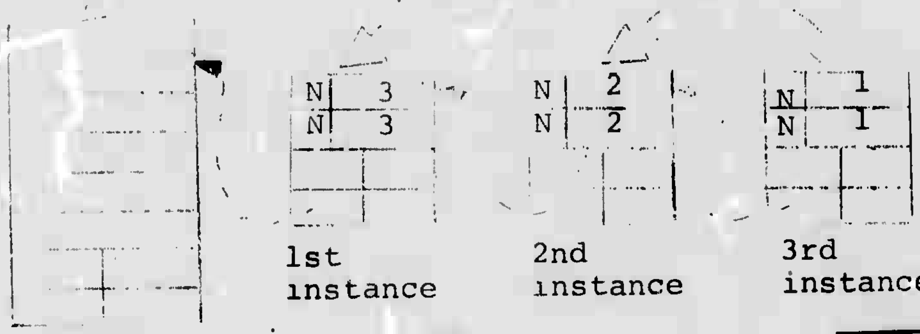
Recursion is easy in the FLEX environment. Each parent process simply creates a new child which is linked to the parent by the returns. The $\boxed{\text{call}}$ operator is used so that just one instance of the recursion is active at any one time.

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'''

```
display ← "fact (3) = "#fact (3);
          fact (3) = 6.0
```



parent   1st instance   2nd instance   3rd instance

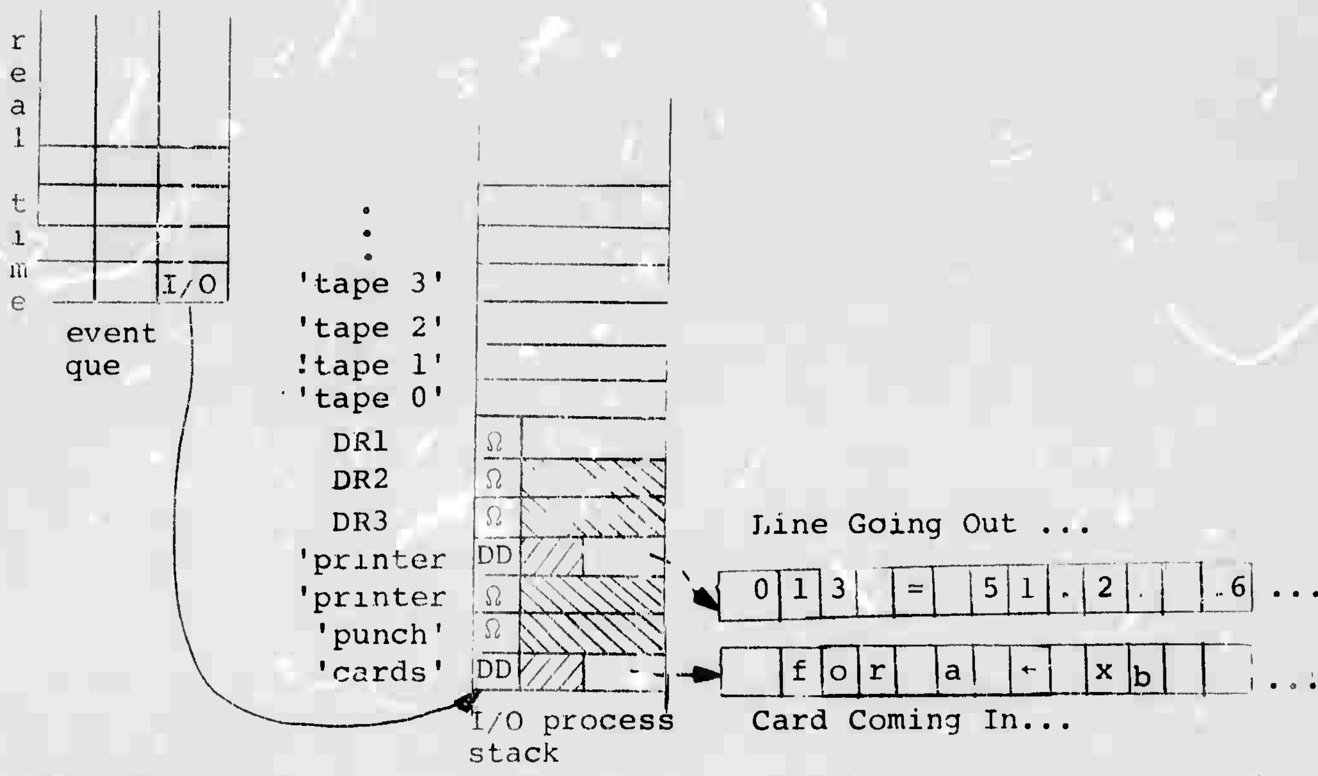
All returns from a call as well as reactivating the parent process also transfer the top element in the stack from the returned process to the top element in the stack of the parent process. This is the way results are passed back when a procedure is used as a function.

Input/Output Conventions

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 que.



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 Ω 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:

<u>printer 2</u> ← "a13 = " # a13# " 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 <u>temporary</u> 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 consequentively generated values of a13 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 <u>passivated</u> 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 reult. The card image (being a temporary) is renamed as the first parameter of format 1 and is thereupon operated on.

Pure input devices are immediately supplied with a new data descriptor input request. So the "cards" section is again set up to receive a card.

<u>Two-Way Devices</u>

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 <u>mapped</u> and then selected on <u>as</u> the <u>data enters or leaves the machine.</u> 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.

VI. <u>Progress to Date</u>

<u>Implementation</u>

Two FLEX compilers have been programmed in ALGOL on the UNIVAC 1108 and have been running since mid-February 1968. Several partially successful attempts were made to combine the compilers with a number of the operating text editors at the University of Utah. 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.

Implementation of the interpreter has been severely delayed for several reasons--the main one being that it took longer than expected to work out a rationale for a segmenting and swapping system that would work on such a small scale.

Current implementation is now taking place on an IBM 1130 partially because the machine can be dedicated most of the time to this task and partly also because it is small and does not tempt one into grandiose schemes. Implementation on a PDP-10 is also being contemplated.

<u>Future Expansion</u>

The process-oriented nature of FLEX should make it an ideal kernel for numerous discrete simulation schemes.

A search for primitives in this as well as in the semantic transformation area is currently going on, and it is expected that some fruit will be available for plucking in the next month on this field of discourse.

Application packages are also being studied with a view toward both allowing FLEX to do something useful and providing a test-bench on which to evaluate the system. To this end, the solid-state circuit design program developed by W.R. Sutherland on the TX-2 computer at Lincoln Laboratories is being eye-balled. FLEX and LEAP (the implementation language at Lincoln) share some properties--notably the ability to store and retrieve associations--and it will be interesting to notice the difference between the 256k words of fast memory on the TX-2 versus 4 to 8k smaller words on the FLEX machine.

REFERENCES

Naur, P. (ed.) "Report on the Algorithmic Language ALGOL-60," Communications of the ACM 3 (May, 1960). pp 299-314

Wirth, K. "Euler - A generalization of ALGOL, and its formal definition: Part I, Part II," Communications of the ACM 9 (January, February, 1966), pp. 13-25, 88-99.

3. Floyd, R.S., "A Descriptive Language for Symbol Manipulation," Journal of the ACM 8 (October, 1961), pp. 579-584

[illegible] A., "An ALGOL-60 Compiler," Annual Review in Automatic Programming, Vol. 4, 1964, pp. 87-124.

Feldman, J., "A Formal Semantics for Computer Languages and its Application in a Compiler-Compiler," Communications of the ACM 9 (January, 1966), pp. 3-9.

Mondshein, L., "VITAL Compiler-Compiler Reference Manual TN 1967-1," Lincoln Laboratories, MIT, January, 1967.

Feldman, J., "Aspects of Associated Processing," MIT, Lincoln Laboratory Technical Note 1965-13, April, 1965.

Newall, A., "A Note on the Use of Scrambled Addressing for Associative Memories," unpublished paper, December, 1962.

9. Rovner, P., and Feldman, J., "The LEAP Language and Data Structure," MIT, Lincoln Laboratory Technical Note DS-5436, October, 1967.

Taylor, W., Turner, L., and Waychoff, R., "A Syntactical Chart of ALGOL 60," Communications of the ACM 14 (September, 1961), p. 393.

NOTE: For a complete exposition on compiler-compilers and a formidable biography on the subject see:

Feldman, J., and Gries, D., "Translator Writing Systems," Communications of the ACM 11 (February, 1968), pp. 77-113.