AD-A073 997 MARYLAND UNIV COLLEGE PARK COMPUTER SCIENCE CENTER F/G 9/2

THE FLEX SYSTEM: USER AND CARETAKER'S MANUAL.(U)

JUN 79 S A SUTTON AFOSR-77-3181A

UNCLASSIFIED CSC-TR-765 NL

1 OF 1

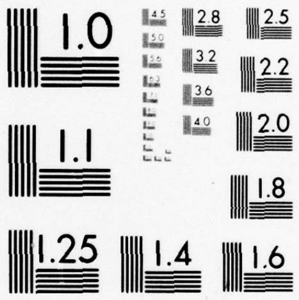
AD A073997

END

DATE FILMED

10-79

DDC


1.0
4.5
5.0
5.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

UNIVERSITY OF MARYLAND

COMPUTER SCIENCE CENTER

COLLEGE PARK, MARYLAND
20742

E10

(12) 79 p.

(9) Technical Report, TR-765 (11) Jun[illegible]979

(15) AFOSR-77-3181A

(6)

The FLEX System: User and Caretaker's Manual*

(10) Stephen A. Sutton

Naval Research Laboratory
Washington, D. C.

(14) CSC-TR-765

*Research supported in part by the Air Force Office of Scientific Research grant AFOSR-77-3181A to the University of Maryland.

(15)

403 018 mt

ABSTRACT

The FLEX Design System is a design language and its Processor that form a tool for use in computer software design activities. This report presents a detailed definition of the FLEX language, directions for using the Processor, and guidelines for installing, maintaining, and modifying the Processor software.

PREFACE

This is one of two current documents that describe the FLEX Design System developed at the University of Maryland and Naval Research Laboratory. It contains the detailed rules of the FLEX language, instructions for using the Processor software, and for maintaining the Processor software.

Reference [1] is a general presentation of the FLEX system and the philosophy behind it, and should be used with the current report.

Flexibility is a key feature of the FLEX Design System, and the user should be aware that rules for the language and Processor may often change. He should always look out for "revision" sheets documenting any changes to the installation on which he is working.

The current version is FLEX 1.5, and the versions 2.x will contain Type Space Execution when it becomes available.

The guidance and assistance of Dr. Victor Basili, University of Maryland Department of Computer Science, in the design of the FLEX system are greatefully acknowledged by the author.

CONTENTS

THE FLEX SOFTWARE DESIGN SYSTEM

The FLEX Design System [1] is a design language and processor that form a tool for use in computer software development activities. The system combines features originating in earlier Process Design Languages (PDL's) with many features found in modern programming languages. The system is quite flexible, and can be adapted to different programming environments; the language can in effect be configured to produce a family of less flexible Process Design Languages.

Among the features offered by the FLEX language are: a modular design structure, a form of type abstraction, definable operators, generic routines, strong type checking, consistency checking of all functional interfaces, and protection of selected data from alteration in certain environments.

This Manual contains the rules for the basic FLEX language. In general, the philosophy and description given in reference [1] are not repeated, and the user should be familiar with that reference.

Appendix 1 describes the use of the Processor, and its various features and options. Appendix 2 describes the Processor software, and contains information necessary for the installation, maintainence, and modifiction of the FLEX Processor. Appendix 3 describes the status of the Processor on various computer systems.

1. BNF Conventions

The syntax of the FLEX language is defined in a standard BNF form, where reduction operators ("::=") seperate the alternates of a production. A single object in single brackets ("{...}") is optional, and a single object in double brackets ("{{...}}") may be repeated any number of times, including 0.

Objects delineated by vertical bars within single brackets ("{ a | b | c } ") show that one of the objects must be chosen for the reduction. Within double brackets, the delineated objects may be taken in any combination, any number of times, including 0.

The empty alternate ("e-production") is given by the lower case "e", and metalinguistic comments are included in Algol comment delimitors, "/* <comment text> */". Reserved words are shown in capital letters in the BNF definition rules, but as lower case in the various examples. When a reserved word is used in the text below, its reserved portion is capitalized, e.g., "INCLude", or "GENERic".

2. Primitive Concepts

```
<id list>          ::= <id> {{ , <id> }}

<att id list>      ::= { <attr> } <id> {{ , { <attr> } <id> }}

<attr>             ::= FIX
                   ::= ALT

<id>               ::= <id char> {{ <id char> | <int> | _ }}

<id char>          ::= /* alphabetic A-Z or a-z */

<int>              ::= {{ 0|1|2|3|4|5|6|7|8|9 }}

<new id>           ::= <id>  /* not defined in current scope */

<mod id>           ::= <id>  /* of some Module */

<seg id>           ::= <id>  /* of some Segment */

<Escape text>      ::= <labs> { <id> : } <text> <rabs>

<labs>             ::= {

<rabs>             ::= }

<text>             ::= /* any text not containing delimitors */

<comment>          ::= <lcom> <text> <rcom>

<lcom>             ::= {{

<rcom>             ::= }}
```

2.1 Identifiers and Blanks

Identifiers in FLEX consist of up to 12 alphanumeric or the pseudo-space character "_". Lower case alphabetics are not equivalent to upper case; "A_BIG_BLOCK" and "A_Big_Block" are different identifiers. Reserved words are always lower case in FLEX text, but are shown as upper case in the BNF definitions, below.

Blanks have the same significance as in many Algol-based languages. They cannot appear within identifiers, reserved words, or integer and real constants, and must must be used to separate tokens that would cause confusion if not separated. The end of a line is syntactically equivalent to a blank.

User defined symbolic operators must be seperated by at least one blank when appearing in conjunction, (e.g., "A+ -B").

Statement seperators are not required, but several statements on a single line may be seperated by empty comments, if desired.

2.2 Escapes and Comments

Escapes may be used in place of expressions, statements, loop clauses, and types. If the Escape text begins with an identifier followed by a colon, then it is a Generic Escape, and must have been declared as such.

Comments may appear between any two tokens.

3. Module Structure

```
<prog sys>          ::= {{ <Module> }}


<Module>            ::= { SYS } MOD <id>
                                  {{ <mod decl> }}
                                  {{ <Segment> }}
                              DOM

<mod decl>          ::= <gener mod decl>
                    ::= EXPORT <att id list>
                    ::= <use decl>

<Segment>           ::= <Data Seg>
                    ::= <Routine>
```

3.1 Programming System and Module

A Programming System is a set of uniquely named Modules, and the Module is a set of Segments which are uniquely named within the Module.

3.2 Name Scopes

Module names are known globally to the Programming System, but Segment names are not global unless declared as EXPORTable in the Module header. Segment names are always visable from within the same Module, and intra-Module access is not affected by the EXPORT declarations. EXPORTed Data Segments have the FIX/ALT attribute, which defaults to FIX if not specified.

In general, Segments are referenced by their Module name followed by the Segment name ("MODX : SEGX"). Since this combination is unique in the Programming System, two Modules may EXPORT two Segments of the same name with no ambiguity. There are cases where the Module name is optional (5.2, 6.5, 10.4).

3.3 Module Header Declarations

The declarations in the Module header are used to specify implicit access for the Segments within the Module (see 6.8). The INCLude is not allowed here since this easily leads to cyclic or multiple access paths which are not allowed (x>>c/x.x).

3.4 SYStem Module

There may be one SYStem Module in the Programming System, and declarations in its header specify implicit access to the named Segments from all Segments in the Programming System (see 6.8).

4. Data Segments

```
<Data Seg>          ::= DATA <id>
                        {{ <static decl>   |
                           <access decl>   |
                           <operator decl> |
                           <Escape def>    |
                           <type def>      }}
                        ATAD
```

4.1 General

A Data Segment is a collection of Elements (data declarations, type or operator definitions) that can be accessed from other Data or Routine Segments with the USE or INCLude access declaration. Each Element name must be unique within the Data Segment.

5. Routines

```
<Routine>             ::= { <scope> } <Function>
                      ::= { <scope> } <Access Function>
                      ::= { <scope> } <Procedure>
                      ::= ITER <Function>

<scope>               ::= FIX | ALT | CLOSED

<Access Function>     ::= ACCESS <Function>

<Function>            ::= {*} FUNC <id> { ( <id list> ) }
                            <body>
                          CNUF

<Procedure>           ::= {*} PROC <id> { ( <attr id list> ) }
                            <body>
                          CORP

<body>                ::= <case body>
                      ::= {{ CASE
                                <case body>
                             ESAC }}

<case body>           ::= {{ <Routine decl> }}
                      ::=    <stmt list>

<Routine decl>        ::= <form decl>
                      ::= <returns decl>   /* not for procs */
                      ::= <dynamic decl>
                      ::= <static decl>
                      ::= <use decl>
```

5.1 Functions and Procedures

The syntactic definition of the Function and Procedure is shown to be the same, but there are differences:

1) Functions must contain a single RETURNS type specification; Procedures cannot contain any.

2) Procedure formal parameter names may each be prefixed by the FIX/ALT attribute, where FIX is assumed if none is given. Function formal parameters are always FIXed, and the attribute need not be specified.

3) All RETURN statements within a Function must give an

expression; the RETURN statements within a Procedure must not.

4) There is an implied RETURN before the CORP ending in Procedures (only), and an explicit RETURN here is optional.

5.2 Generic Routines

Routines that are to be referenced genericly (Segment name alone) must be declared by preceding their FUNC/PROC header with an asterisk (*).

5.3 Routine Scope

The FIX/ALT/CLOSED Routine scope restricts the manner in which a Routine may access its Environment, and defaults to FIX if not specified. ALT Procedures are allowed, but ALT regular, ACCESS, and ITER Functions are not.

A FIXed Routine cannot alter data objects, or access underlying type definitions in its External Environment.

CLOSED Routines may not have "own" (STATic) variables, and may not attempt to reference data objects in the data bases to which they have access. They can reference type definitions, operator definitions, or CONSTants.

These restrictions are checked when actual references are made within the Routine's text body, and not when the access is declared. These restrictions do not affect the type of access declarations that a Routine may contain, only his actions in accessing those Segments.

FIXed Routines may call only FIXed or CLOSED Routines, and CLOSED Routines may call only CLOSED Routines.

5.4 Regular, ACCESS, and ITERation Functions

A regular Function returns a data object that is uncoupled from the caller's Environment; a new data object is created that will be released when it can is no longer needed. An Access Function returns an object that is already a member of the caller's environment.

ITER Functions are used for iteration over abstract data types in conjunction with the FOREACH loop clause. ITER Functions are implicitly CLOSED ACCESS Functions, and may be generic.

5.5 Routine CASEs

Each CASE in a Routine has its own separate declarations and <case body>. The names, number, and FIX/ALT attribute of the formal parameters is the same for all cases, but the type of a formal parameter may vary between CASEs, as can the Function RETURNS type.

The CASE that is invoked by a Routine invocation is the one that matches the number and type of the actual parameters. The FIX/ALT attribute of actual and formal parameters are not considered in selecting the CASE. If more than one CASE satisfies these conditions, then it is an error.

6. Declarations

```
<dynamic decl>      ::= DECL    <id list> <type>

<static decl>       ::= STAT    <id list> <type>
                    ::= CONST   <id list> <type>

<form decl>         ::= FORM    <id list> <type>

<returns decl>      ::= RETURNS <type>

<access decl>       ::= <use decl>
                    ::= <include decl>

<use decl>          ::= USE <attr seg> {{ , <attr seg> }}

<include decl>      ::= INCL <attr seg> {{ , <attr seg> }}

<attr seg>          ::= { <attr> }    <seg id>
                    ::= { <attr> } "<seg id>"
                    ::= { <attr> }   <mod id> :  <seg id>
                    ::= { <attr> }   <mod id> : "<seg id>"
                    ::= { <attr> } "<mod id> :  <seg id>"

<Escape def>        ::= ESCAPE <id list>

<gener mod decl>    ::= GENER <mod id> {{ , <mod id> }}
```

6.1 Data Declarations

The dynamic (DECL) data object may appear only in Routines. It is "stack based" data that is "created" at the entry to a Routine, and "deleted" at exit.

A STATic data object (Algol "own") may appear in Data Segments and non-CLOSED Routines. It is neither created nor released at run time, and is shared by all instantiations of a Routine.

A CONSTant is a data object whose value can never be changed, and can be declared in Data Segments or any Routine. Values are assigned to CONSTants at inspection time only, and must currently be indicated with comments.

The FORMal parameter declaration gives the types of the formal parameters. This type may be an UNBOUND type or a Type Pseudo Function.

6.2 RETURNS Type

The RETURNS declaration is used in Functions to specify the type of the data object returned by the Function. Record selector identifiers in this type are superfluous (but must be present) because the selectors are not considered a part of the pure "type" of a Record. This type may be a Type Pseudo Function.

6.3 The INCLude Access Declaration

The INCLude and USE access declarations are used to gain access to the Elements of a Data Segment. Access may be nested as deeply as desired, for example, A may INCLude B who may INCLude C, and so forth:

```
<----- upstream -----
A => B => C => D => E
----- downstream ---->
```

Note that A is the only one who may be a Routine.

This structure so formed is a tree -- not a graph: cycles and multiple access paths to the same Segment are not allowed. Hence, if E INCLuded B, then a cycle would be formed, or if B had another access path to E, then a multiple access path would be formed.

Referring to this example, B adds the entire Environment of C (which includes D, and E) to its own External Environment. If B INCLudes C as ALTerable, then B has the same access privileges to that added Environment as did C. If FIXed, then the added Environment is FIXed with respect to B and also A, no matter which attribute is used by A in adding B.

If the INCLuded Segment is not in the same Module, it must have been declared as EXPORTable by its own Module, and if it has been exported as FIXed cannot be accessed as ALTerable.

6.4 The USE Access Declaration

The USE declaration is exactly the same as the INCLude declaration with respect to the accessing Segment (e.g., as far as B is concerned, "USE C" is equivalent to "INCL C"). Segments who USE/INCLude the declaring Segment, however, do not gain subsequent access to the USEd Segments, only to the INCLuded ones. (If B INCLuded C, then when A accessed B, it would also gain access to C, but not if B had USEd C.)

Effectively, INCLude signifies that another Segment is to be considered a permanent part of the current one for all accessors of the current one to see. The USE simply gains access to some other Segment where there is no intent to make it a permanent part of the current one.

6.5 The Quote Convention

The quotation marks in the USE or INCLude declaration determine how Elements in the referenced Data Segment are to be accessed. In general, Elements may be accessed by 1) their name only, 2) their owning Segment followed by the name, or 3) their owning Module followed by owning Segment and name.

The following table summarizes the rules that apply to this quote convention. Note that a different rule may apply for each accessor of a given Data Segment, depending upon how he applies this quote convention in the USE or INCLude declaration.

ACCESS	REQ'D REFERENCE SEQUENCE
<same Segment>	<name>
USE <seg id>	<name>
USE "<seg id>"	<seg id>:<name>
USE <mod id>: <seg id>	<name>
USE <mod id>:"<seg id>"	<seg id>:<name>
USE "<mod id>: <seg id>"	<mod id>:<seg id>:<name>

The example shows USE declarations, but the same rules hold for INCLude.

The requirement to use the Segment or Module/Segment name prefixes is an overriding restriction to all "upstream" Segments (6.5). When A accesses B (above example), A must use the same access convention that B uses for the variables downstream of B, unless A has applied a stronger quoting convention.

This mod/seg/name convention does not apply to identifier operator definitions and Generic Escapes, which must always be referenced by their identifier name only.

6.6 Generic Escape Definitions

Escape definitions define Generic Escape identifiers that may appear in the Escape text, as in "{ <id>: <text>}". Note that Escape identifiers must be used alone and cannot be preceded by the Module and/or Segment name.

6.7 Generic Module Declarations

The GENER declaration is used to add a Module to the Generic Environment of a Routine CASE. Whenever a generic Routine call is performed, all accessable generic Routines in the Modules of the Generic Environment are searched for one with a CASE that satisfies the actual parameter types.

It is legal to include the same Module more than one time in the Generic Environment.

6.8 Implict USE Access

Each Data Segment and Routine CASE in the Programming System may have implicit (i.e., not explicitly declared within the Segment) USE and/or GENER declarations. When USE or GENER declarations appear in a Module header, they become implicit to every Segment in that Module. However, if the Segment contains an explicit USE or GENER reference to the same Segment or Module, respectively, then the reference in the Module header is ignored.

If the Module is the SYStem Module, then the header declarations become implicit to every Segment in the Programming System. However, this implicit reference is overwridden by an explicit reference in a Module header or Segment.

This allows, for example, certain Segments to have explicit ALT access to some Segment, while the rest of the Programming System or Module is implicitly given FIX access.

6.9 Examples

```
      decl   XX, YY bool
      stat   ZZ, QQ int
      const  BB, CC seq ( int )

      returns record
                DUMSEL1:  int
                DUMSEL2:  seq ( seq ( real ))
              drocer

      use SEGA, alt SEGB, fix SEGC       {{Segs in current Module}}

      use MODA:SEGX                      {{access by var name only}}
      use alt MODA:"SEGX"                {{access by seg:element}}
```

```
use "MODA:SEGX"            {{access by mod:seg:element}}
```

7. Operators

```
<operator decl>      ::= INFIX {<prio>} <op spec> = <opr seg>
                     ::= PREFIX <op spec> = <opr seg>

<opr seg>            ::= {FUNC|PROC} { <modid> : } <segid>

<prio>               ::= <int>

<op spec>            ::= <id>
                     ::= ' <op char> { <op char> } '

<op char>            ::= /* special op character (7.2) */
```

7.1 Operator Definitions

The action taken by the FLEX processor on encountering a defined operator in a statement or clause as if a named Routine invocation had been encountered, where the Module and/or Segment names are those given in the operator declaration (i.e., <opr seg>). Hence the two expressions are equivalent given the following operator definition:

```
        infix 2 "+" = MODX:ADDITION

        <exp a> + <exp b>
        MODX:ADDITION ( <exp a> , <exp b>)
```

Procedures as well as Functions can be invoked by operators, as indicated by the FUNC or PROC keyword in the operator definition (FUNC is the default if none is given). Procedures can only be so invoked from Procedure call statements, and not from within expressions. PREFIX PROC operators are not allowed.

The operator is considered generic if only the Segment name is present in <opr seg>.

INFIX operators are given a priority for expression evaluation, where priority 6 is the highest, and the default is 1. The priority is ignored for Procedure invocations.

PREFIX operators have no priority among themselves but are of higher priority than all infix operators.

Identifier operators, once defined in an environment, are considered reserved words in that environment.

7.2 Allowable Operator Characters

The allowed characters in symbolic operators are:

```
+-*/&<>=!$%~\@`:
```

However, the colon (":") may only be used in double-character operators.

Syntactic conflicts may arise between infix and prefix operators, and the rule to be observed is:

> If a single character infix symbolic operator exists
>
> and
>
> a double character infix operator exists whose first character is the same as this single character operator
>
> and
>
> a prefix operator (single or double character) exists whose first character is the same as the last character of this double character infix operator
>
> then
>
> an error condition exists

7.3 Examples

```
infix 1 '+'  = SYSTEM:PLUS      {{arithmetic addition}}
infix 2 '**' = SYSTEM:EXPONENT {{arithmetic exponentiation}}
infix   ':=' = proc ASSIGN

prefix  NOT  = func SYSTEM:BOOLEAN    {{Bool 'not'}}
prefix  '-'  = func SYSTEM:MINUS      {{arithmetic negation}}
```

8. Types

```
<type>                 ::= <record>
                       ::= <seq>
                       ::= <scalar>

                       ::= <Macro ref>
                       ::= <Macro formal>

                       ::= UNBOUND
                       ::= <type pseudo func>

                       ::= <Escape type>

<record>               ::= RECORD
                                <selector id> : <type>
                             {{ <selector id> : <type> }}
                           DROCER

<seq>                  ::= SEQ ( <type> )

<scalar>               ::= INT
                       ::= <int>
                       ::= REAL
                       ::= CHAR
                       ::= BOOL
                       ::= <id>

<selector id>          ::= <id>

<Macro ref>            ::= <id> { ( <type list> ) }

<Macro formal>         ::= <lbr> <id> <rbr>

<lbr>                  ::= <
<rbr>                  ::= >

<type pseudo func> ::= TYPEOF ( <id> { <psf part> } )

<psf part>             ::= [ <psf sel> ] { <psf part> }
                       ::= . <selector id> { <psf part> }

<psf sel>              ::= INT
                       ::= e

<Escape type>          ::= <Escape text>
```

8.1 General

All types is FLEX are static; once a data object is created of some type, that type cannot subsequently change during the lifetime of the data.

The syntactic definitions of the various types must be tempered by semantic rules governing where certain type specifications may appear, and these rules are noted in the descriptions below.

8.2 The RECORD Type

The RECORD is a collection of data objects of potentially different types, where each object can be addressed by a named, non-computable Selector. Selector identifiers need only be unique within each RECORD, and the objects of the Record are selected using a DOT convention (11.2).

The selector identifiers are not considered as part of the type for type checking purposes, hence the following two types are equivalent:

```
record
  SELA: int
  SELB: real
drocer

record
  SELC: int
  SELD: real
drocer
```

8.3 The SEQuence Type

A SEQuence is an ordered set of data objects of the same type. There is an implied mapping to the set of integers, but there are no inherent length restrictions on SEQuences.

SELECT1 and SELECT2 are built into FLEX for SEQuences (11.3, 11.4).

8.4 The SCALAR Type

SCALARs are types that have no underlying form; they are defined in terms of no other types. The following SCALARs are built into FLEX: INTeger, REAL (floating point), CHAracter, and BOOLean.

Other SCALARs must be defined by the user (9.1).

8.5 Defined Types

The <Macro ref> is used to create an instance of a type defined elsewhere in the current Environment. The types in parenthesis are the ACTUAL PARAMETERS. They must match the number of Macro formal parameters, and their types must conform to any constraints in the type definition (see <tfrm elem>, 9.).

An integer constant (<int>) can be used as a Macro formal parameter. It is treated by the Processor as if it were the type specification "int" (a reserved word), but can be used in locations where the user wishes to indicate that a "length" is being used in the instantiation of of some defined type, for example:

```
    type ARRAY (LEN: int, ARG)
      record
        LEN:  <LEN>
        BODY: <ARG>
      drocer

    decl X ARRAY (32, real)  {{array of 32 reals}}
```

8.6 UNBOUND Types

The UNBOUND type specification may only appear in the types of FORMal parameters, for example:

```
      form X unbound
      form Y seq ( unbound )
      form Z record
                Z1: int
                Z2: seq ( unbound )
             drocer
```

8.7 Rules of Type Pseudo Functions

Type Pseudo Functions may only appear in Routines, and can appear in all declarations except STATic and CONSTant declarations. The <id> argument must be of one of the formal parameters, and the formal declaration must lexocographicly precede the TYPEOF reference. The type of that formal parameter need not contain

UNBOUND types, but logically it is desirable, and may contain Type Pseudo Functions.

Type Pseudo Functions may be contained within other type specifications except in the RETURNS specification, where if must be outermost, if present:

```
form A unbound
form B seq (typeof (A))
returns typeof (A)
returns seq (typeof (A))   {{ illegal }}
```

The <psf sel> used to specify a SEQuence selection may only be an integer expression or empty (i.e., a call to SELECT1 with a single integer expression, or to SELECTO, 11.3, 11.4). This is an interim restriction that will vanish when Type Space Execution is implemented.

8.8 Use of Type Pseudo Functions

This type specification is used to specify a type that, like unbound types, i is not known until a specific invocation of the Routine and is bound at that time. That type is actually some Function of the actual formal parameter types after they are bound.

A Type Pseudo Function has a format similar to a <variable> (11.3) specification involving one of the formal parameters, where any Selector expressions ("[<exp>]") are replaced by type specifications. It is as if the series of Record selections and calls to the SELECT1 Functions were made, except that the only purpose of these calls is return types, and not values.

The SELECT1 selection can only be applied to SEQuences, however. Any Routine using the pseudo-type must have sufficient access to any defined type Macros so that the SELECT1 convention can be applied. In essence, the Routine must be able to "see" enough of the underlying structure of the type of the FORMal parameter to correctly specify the SEQuence SELECT1 and Record selections it is requesting.

The formal parameter must have enough of its type not-UNBOUND to determine the correctness of this addressing.

Consider the following example:

3) All RETURN statements within a Function must give an

```
func FOO (W,X,Y)
   form W seq of int

   form X record
             X1: seq of unbound
             X2: unbound
          drocer

   form Y typeof (X)

   decl Z typeof (X.X1 [int])

   decl A typeof (W [int])    {{error}}

   returns typeof (X.X2)

   {body}

cnuf
```

This specifies that the formal "X" is must have the form indicated in its type specification, except that the "unbound" type is not known until this Function is invoked.

The formal "Y" is bound at that time to the actual type of "X" and then checked against the type of the second actual parameter for type equivalence. Essentially this requires the second actual parameter to be the same as the first, regardless of the type of the first.

The RETURNS type is also bound at invocation time and is bound to the type of the second selector of the formal "X". Effectively, the type returned by this Function depends upon the actual formal parameter types presented by the caller.

The local declaration "Z" is also bound when invoked and is bound according to the Type Pseudo Function "X.X1 [int]", which binds it to the type of the SEQuence member of the first selector of the formal "X".

8.9 Escape and Wildcard Types

The "wildcard" type is considered type equivalent to any other type. Escapes are given the wildcard type when used as type specifications. The wildcard type can always have a Record or Selector (11.3) selection done upon it that will yield the wildcard type again. The Processor often assigns this type when an error occurs that precludes the correct type from being identified.

CLOSED ACCESS Functions, and may be generic.

9. Type Definitions

```
<type def>          ::= TYPE <id> = SCALAR { ( <id list> ) }
                    ::= TYPE <id> { ( <tfrm list> ) } = <type>

<tfrm list>         ::= <tfrm elem> {{ , <tfrm elem> }}

<tfrm elem>         ::= <id> { : <id list> }
```

9.1 SCALAR Definitions

The optional identifiers at the end of the scalar definition are constants for the scalar being defined.

9.2 Parameterized Type Macros

Parameterized Type Macros are forms that define new types in terms of existing types and "formal parameter types" that are left unspecified until a data object is declared of this type. The formal parameters are replaced by "actual parameters" when a data object is declared of this Macro type (8.5).

The formal parameter identifiers need only be unique among themselves; since they are surrounded by brackets ("<...>"), they will not conflict with other identifiers in the type specification. An formal parameter identifier may be used any number of times within the body of the type definition, and each identifier must be used at least once.

The actual parameters may be allowed to be of any type (e.g., "(... A, ...)"), or to be one of a set of allowable types (e.g., "(... A: int, real ...)").

The underlying structure of a defined type can be "seen" only by those Routines that have ALT access to the Data Segment in which the Macro was defined.

Consider the following example of a stack:

```
        type STACK (MBR) =
                       record
                          LEN: int              {{its current length}}
                          BODY: seq ( <MBR> )   {{the body}}
                       drocer

        stat X STACK (real)                     {{a stack of reals}}
```

The "stack" is defined as a Record whose first member is the integral current length of the stack, and whose second member is a SEQuence of members whose types are left unspecified until a data object is declared.

As a second example, consider the definition of a stack whose elements are required to be integers, reals, or characters:

```
        type ASTAK (MBR: int, real, char) =
                     record
                       LEN:  int
                       BODY: seq ( <MBR> )
                     drocer

        stat X ASTAK (real)
        stat Y ASTAK (string)    {{ error !! }}
```

As a possible point of confusion, data objects can have no alternate types. In declaring a data object from a Macro definition, one of the alternate types is instantiated, and this type cannot change for the life of the data object.

9.3 Type Equivalence

Type equivalence in FLEX is a name equivalence, where defined types are truly new types, and not simply templates. The following rule defines type equivalence:

> Two Scalar types are type equivalent if and only if they are instances of the same defined scalar type.
>
> Two SEQuence types are type equivalent if and only if their objects are type equivalent.
>
> Two Record types are type equivalent if and only if their corresponding selector types are equivalent (but the corresponding Selector identifiers need not be the same).
>
> Two defined types are type equivalent if and only if they were created from the same type definition, and the corresponding actual types used were type equivalent.

9.4 Examples

These are examples of common type definitions. The LIST type referred to is defined in detail below (14.2). The definition of STRING is discussed below (11.6), and is a common way of defining strings in FLEX.

Function.

ASSOC_MEM is an associative memory where elements are stored according to some string. The user would presumably create Routines for accessing and maintaining objects of this type.

```
    type COLOR = scalar (RED, ORANGE, YELLOW, GREEN, BLUE)

    type STRING = LIST (char)

    type ARRAY3 (X) = seq (seq (seq ( <X> ))) {{a 3-dim array}}

    type ASSOC_MEM (X) =
        LIST (record
                TAG: string
                VALUE: <X>
             drocer)
```

10. Expressions

```
<exp>               ::= <attr> <exp1> { <op1> <exp> }

<exp1>              ::= <exp2> { <op2> <exp> }

<exp2>              ::= <exp3> { <op3> <exp> }

<exp3>              ::= <exp4> { <op4> <exp> }

<exp4>              ::= <exp5> { <op5> <exp> }

<exp5>              ::= <term> { <op6> <exp> }

<term>              ::= <prefix op> <term>
                    ::= ( <exp> )
                    ::= <func invo>
                    ::= <primitive>
                    ::= <Escape exp>

<op1>               ::= <infix op>        /* op2, op3,..., op6 */

<infix op>          ::= <op spec>    /* above in OPERATORS */

<prefix op>         ::= <op spec>    /* above in OPERATORS */
                    ::= -

<func invo>         ::= <Routine spec> ( { <parm list> } )

<Routine spec>      ::= { <mod id> : } <seg id>

<parm list>         ::= <exp> {{ , <exp> }}

<Escape exp>        ::= <Escape text>
```

10.1 Operators

Infix operators are prioritized, left associative, and must be defined in the current Environment. Prefix operators have a higher priority than any infix operator, and must also be defined in the current Environment.

If an operator is available for a particular Routine, then it must be used to invoke the Routine; a <func invo> cannot be used if an operator exists that will do the same job.

Module and/or Segment name.

10.2 Arithmetic Negation "-"

The arithmetic negation operator ("-") is unique in FLEX: it is the only operator that can be used as both infix and prefix. The user should define the "-" INFIX operator corresponding to some negation Function, and the Processor will take special action when used as a PREFIX operator. The expression "-<e>" will be treated for type checking purposes as of it were the expression "<d>-<e>", where "<d>" is a dummy wildcard type. Effectively, the INFIX negation Function is also used for type checking of PREFIX negation.

10.3 FIX/ALT Attribute

Each expression may be prefixed by the FIX or ALT protection attribute (FIX is the default), but the ALT attribute must not violate any inherent FIX that may be attached to the expression. These are matched against the FIX/ALT attributes of the formal parameters of each called Routine.

Note the form of the following expression:

```
      PROCX (fix (fix A + fix B))
```

The second and third FIX's apply to the arguments presented to the function represented by the "+" operator. The first FIX is then applied to the result of that Function for the call to the Procedure PROCX.

10.4 Function Invocation

If both Module and Segment names are given then the Function invocation is Specific; if the Segment name only is given, it is generic. If the Routine is in another Module then it must have been declared to be EXPORTed, and if generically called, must have been declared as generic (5.2).

If an error occurs in a generic Function invocation, for example when the called Routine is not EXPORTED or declared generic, then a "NOT FOUND" error will result.

The order of evaluation of the actual arguments is not specified.

The parenthesis in the explicit Function call (<func invo>) must be present even if there are no parameters so that the reader can easily distinguish between a Function call and a variable.

```
{{access by seg:element}}
```

10.5 Escape Expression

When Escapes are used as expressions, they assume the wildcard type. However, this can cause certain problems when used as actual parameters in a generic Routine call. Suppose their are two generic Routines "PLUS": one for integers and one for reals. If the actual parameters for a call to "PLUS" are wildcard types, then the system will detect an ambiguous generic call because there are two CASEs that satisfy the type interface conditions.

10.6 Examples

```
          LAST (STACKA) := FOO (XX, A+B) - MODA:FUM ()
          SOMEVAR := (A+B)-((C*E*F)/56.6)       {{all vars real}}
```

11. Primitives

```
<primitive>          ::= <variable>
                     ::= <constant>
                     ::= <denoted record>

<variable>           ::= { { <mod id> : } <seg id> : }
                           <id> <varpart>

<varpart>            ::= <rec sel> { <varpart> }
                     ::= <bracket> { <varpart> }

<rec sel>            ::= . <selector id>
<bracket>            ::= [ { <exp> {{ , <exp> }} } ]

<constant>           ::= <int>
                     ::= <real const>
                     ::= <bool const>
                     ::= <char const>
                     ::= <scalar const>
                     ::= <string const>

<real const>         ::= <decimal> { E {+|-} <int> }

<decimal>            ::= <int> .
                     ::= <int> . <int>
                     ::= . <int>

<bool const>         ::= TRUE
                     ::= FALSE

<char const>         ::= CHAR ('{ <esc char> } <char> ')

<scalar const>       ::= <id>            /* def'd in scalar def */

<string const>       ::= ' {{ <char>|<esc char> }} ' {{ <string const
> }}

<char>               ::= /* any keyboard character */

<esc char>           ::= @  /* the "escape" character */

<denoted record>     ::= RECORD ( <exp> {{ , <exp> }} )
```

11.1 Data Objects

Data objects are indicated by their identifier names, optionally prefixed by the Module and/or Segment name in which they were defined (e.g., "MODID:SEGID:VARID" or "SEGID:VARID" or "VARID"), depending on how the Quote Convention (6.5) was applied.

A subobject of a data object can be addressed by an Access Function, a Record selection using the "dot" convention, or the "Selector" convention.

11.2 Record Member Selection

If the variable was of type RECORD then one of its objects may be selected by the decimal point (".") followed by the selector id to be selected.

11.3 The Selector "[...]" Convention

Upon encountering the bracketed expressions in the parse of a <variable>:

```
    <var> [ <e1>, ... ,<em> ]
```

(where "<var>" is the variable subobject selected so far) the Processor issues a generic call to a Function named SELECTm (SELECTO, SELECT1, ..., where "m" is the number of expressions between the brackets), as illustrated by:

```
    SELECTm ( <var>, <e1>, ... , <em> )
```

The expressions within the brackets may be of any type, so long as a Selector Function exists that will accept the types, and each expression may be preceded with the usual FIX/ALT attribute.

Selector Functions must be generic, and cannot be called as a regular Function, as in "SELECT1 (<e1>, <e2>)". The user may create Selector Functions as either regular or Access Functions, where Access Functions will probably be the more frequent case.

SELECT1 for SEQuences is built into FLEX, and returns the N'th member of the SEQuence, where N is the integral value of the expression. SELECT2 is also built in, and returns a contiguous subsequence of the first through the last integral expression, or an empty sequence if the first integral value is greater than the last. There are no limits on the integral values accepted by these Functions for SEQuences.

Note that there need be no expressions within the brackets, and SELECTO may be used, for example, to fetch the last member of some data object type.

11.4 SELECTn for SEQ Macros

One rather important point must be made about SEQuences, defined types, and the Selector convention. Consider the type definition and instantiation:

```
          type FOO (...) = sequence (...)

          stat XX FOO (...)
```

When a Routine who has access to the internal form of the FOO Macro type makes a bracket reference to "XX [...]", then the internal SELECT1 or SELECT2 for sequences will be invoked to yield a member of the sequence.

If the Routine does not have type access, however, a generic Function invocation to SELECTn will be issued whose first argument will be of type "FOO".

In effect, if the Routine has no knowledge of the internal form, it must always call upon the user-defined Selector Function to return some subobject.

If the Routine does have access, then it immediately "sees" into the definition, and a Selector reference is assumed to apply to the SEQuence that forms the body of the type definition.

This also applies to multi-times-removed SEQuence definitions, as in:

```
          type FUM = FOO
          type FOO = FUD
          type FUD = seq (...)
```

11.5 Constants

Constants are provided for the scalars built into the FLEX language, i.e., integer, real, boolean, and character. A <scalar const> must be one of the constant identifiers (9.1) in a scalar type definition in the current Environment.

11.6 String Constants

Whenever a string constant is encountered in the design text (e.g., "'abcd'"), the current environment is searched for a defined type named "STRING" (see for example 9.4). The type of the definition so found is then assigned to the constant string. If STRING is not defined an error will result, and its Module and/or Segment name cannot be required for access.

Although strings (" 'abcd' ") may not cross line boundaries, they can be placed in conjunction with an implied concatenation. Strings that need to span more than one line can be broken apart with no loss of meaning. For example, " 'abc' 'def' 'ghi' " is equivalent to " 'abcdefghi' ".

An escape character ("@") is used to indicate the string delimitor (" ' ") when part of the string, where the character following the escape character is always considered part of the string. Two successive escape characters can be used to put the escape character itself in the string.

11.7 CHARacter Constant

The character constant is indicated using the built-in function "CHAR", and a string of one character, e.g., "CHAR ('X')" is the character "X". The escape character may also be used as in string constants.

11.8 Denoted Records

The denoted Record is a FIXed Record whose objects are built from the expressions in the RECORD expression list. For example, if some Routine "FOO" required a parameter that was a Record whose first member was an integer and second was a real, then it could be called by:

```
...FOO (record (123, 123.4E-14))...
```

12. Statements

```
<stmt list>         ::= <stmt> {{ <stmt> }}
                        { <return stmt> | <exit stmt> }

<stmt>              ::= <if stmt>
                    ::= <loop stmt>
                    ::= <call stmt>
                    ::= <Escape stmt>

<if stmt>           ::=      IF <bool exp> THEN
                               <stmt list>
                          {{ ELSEIF <bool exp> THEN
                               <stmt list>  }}
                           { ELSE
                               <stmt list> }
                             FI

<call stmt>         ::= <Routine spec> ( { <parm list> } )
                    ::= { <attr> } <call arg 1> <proc inf op> <exp>

<call arg 1>        ::= <variable>
                    ::= <func invo>

<proc inf op>       ::= <op spec>      /* above in OPERATORS */

<return stmt>       ::= RETURN { ( <exp> ) }

<Escape stmt>       ::= <Escape text>
```

12.1 The IF Statement

The IF statement provides conditional flow control. It may have any number of ELSEIF clauses, and an optional ELSE clause. The expression in the IF and ELSEIF clauses must be of type Boolean.

12.2 The Procedure Call Statement

This statement is used to invoke a Procedure. The actual arguments may be altered by the called Procedure unless they are FIXed. The parenthesis are required in the Procedure call statement even if there are no actual parameters.

Procedures can be invoked by an infix operator, where the operator must be in the current Environment, and must have been defined as a

"PROC" operator. This first argument (i.e., the "left hand side", <call arg 1>) is not a generalized expression, but must be either a variable or Function invocation. The attribute of this first argument defaults to ALT if not specified. (The two latter rules do not apply to Procedure calls not invoked by an infix operator.)

12.3 The RETURN Statement

The RETURN statement causes an immediate return from the Routine. If the Routine is a Function, then an expression must be given and must be of the same type as declared in the RETURNS declaration. The parenthesis enclosing this expression is manditory.

If the Function is an Access Function, then the data object returned must be a subobject of the first formal parameter.

There is an implied RETURN immediately preceding the CORP ending of a Procedure case. No statement should follow the RETURN statement in a statement list since it would be unreachable.

12.4 Escape Statement

The Escape statement may be used anywhere a statement may appear.

13. Loops and Iterations

```
<loop stmt>          ::= { ## <loop id> }
                         DO { <times clause> }
                            {{ <using clause> ! <foreach clause>
                               ! <for clause> }}
                            { <while clause> }
                            { <unless clause> }
                            {{ <Escape clause> }}

                           { <stmt list> }
                           { <until clause> }
                         OD

<loop id>            ::= <id>

<exit stmt>          ::= EXIT { ( <loop id> ) }

<using clause>       ::= USING <new id> = <exp>

<times clause>       ::= TIMES <int exp>

<for clause>         ::= FOR <new id> = <for exp> {{ , <for exp> }}

<for exp>            ::= <int exp>
                     ::= <int exp> { BY <int exp> } TO <int exp>

<int exp>            ::= <exp>    /* expression of type integer */

<unless clause>      ::= UNLESS <bool exp>

<foreach clause>     ::= FOREACH <new id> IN <exp>

<while clause>       ::= WHILE <bool exp>

<until clause>       ::= UNTIL <bool exp>

<Escape clause>      ::= <Escape text>
```

13.1 WHILE Clause

If the boolean expression in the WHILE clause is FALSE, the the loop is exited immediately. There can only be one while expression in the loop header, and it is tested at the beginning of each iteration.

13.2 UNTIL Clause

If the boolean expression in the UNTIL clause is TRUE, then the loop is immediately exited. 1534e can only be one UNTIL clause at the end of the loop, and it is tested at the end of each iteration.

13.3 UNLESS Clause

If the boolean expression in the UNLESS clause is TRUE, then the current iteration is skipped, but the loop is not terminated. This is placed after the optional WHILE clause.

13.4 Loop Identifier

The LOOP ID attaches an identifier to the loop, and must not conflict with any of the loop names for any loop enclosing the new loop (there may be two loops of same name so long as one does not contain the other). The loop id in the EXIT statement may be used for the exiting of several loops at once. A loop id should not be thought of as a label since nothing ever "branches" to it; it is simply the name of the loop.

13.5 EXIT Statement

The EXIT statement provides an immediate exit from some enclosing loop. If a loop id is specified then the enclosing loop with that name is exited. If no loop id is given than the closest (most recently-entered) loop is exited. Note that the loop identifier must be enclosed in parenthesis, and that no statement should follow the EXIT statement in a statement list because it would be unreachable.

13.6 TIMES Clause

The TIMES clause simply defines the maximum number of iterations to be done in the loop. It is placed first in the header because it is evaluated only once before the first iteration. The FLEX semantics consider a negative or zero value to cause the loop to be skipped. Since this restriction is not checkable by by the FLEX processor, it may be altered by user agreement.

13.7 New Identifiers

This new identifier will be bound to some data object by a USING, FOREACH, or FOR clause for use within the loop. It is added to the current internal environment when it is lexocographically declared and must not conflict with any other identifier in the current environment. It is released from the environment at the lexocographic end of the loop.

For the USING and FOREACH clauses, this new identifier is actually bound to some data structure (in effect, a system-controlled pointer) and that data structure can be changed through this identifier unless the identifier is FIXed. The identifier is FIXed if the USING or FOREACH expressions are inherently FIXED, or are preceded by the FIX attribute.

The identifier is available for use by USING and FOREACH clauses that appear lexocographically afterward in the loop header.

13.8 FOR Clause

A standard FOR clause is provided where a new FIXed integer identifier is created to be iterated from one expression to another by the increment of a third, optional expression. The exact semantics of the FOR clause involve "run time" restrictions that cannot be checked or implemented by the processor. FLEX assumes certain conventions below but these can be changed by user agreement.

The expressions are evaluated only once at loop entry. The iteration proceeds according to the following rules:

A positive increment expression implies that the iteration is terminated as soon as, at the beginning of an iteration, the indexed variable is greater than the final expression. A negative increment expression implies that the iteration is terminated as soon as the indexed variable is less than the limit expression.

If the initial expression is less than the limit expression and the increment is negative, then no iteration will occur and the denoted list is empty. Likewise, if the initial expression is greater than the limit expression and the increment expression is positive, then no iterations occur.

13.9 USING Clause

The USING clause binds a new identifier to an expression (not necessarily a variable) during each iteration. The FIX/ALT attribute may precede the expression to indicate whether access through the newly-bound identifier is to allow alteration of the expression. (Note that in order for the expression to ALTerable, it must be a variable or the result of an Access Function reference whose first actual parameter was alterable.) For example:

```
do using X = A.SELX
     X [14] := <exp>
od
```

is functionally equivalent to:

```
A.SELX [14] := <exp>
```

13.10 FOREACH Clause -- Iterators

The FOREACH clause is used in conjunction with ITERation Functions to provide iteration over data objects of user-defined types. The Function referenced in the FOREACH clause must have been declared an ITER Function, and may be generic.

13.11 Loop Termination Priority

The FOR, FOREACH, and TIMES clauses may specify different maximum loop iterations, and this is considered an error in the language semantics. However, this is a "run time" restriction and can be altered by user agreement (e.g., terminate on shortest maximum iteration).

The WHILE and UNTIL clauses and the EXIT statement cause immediate termination regardless of the state of the other clauses.

13.12 Escape Clause

The Escape clause is used to simulate any loop constructs that are not included in the FLEX syntax, and must appear after all other clauses in the loop head.

The versatility of these clauses should not be overlooked in designing with the FLEX system, for they can be used to emulate many iterative schemes found in programming languages.

13.13 Examples

```
    do times 14
       while {A not equal to 0}

       {body, iterated while A is not = 0, but 14 times, max}
    od
```

```
    do foreach X in INORDER (LISTX)
       unless X = 0
       {body}
    od
```

```
    do for X = 1, 2, 3 by 2 to 9, X+Y, FUNCX (Q)
       {body}
    od
```

```
    ## LOOPA
    do
      ##LOOPB
      do {forever}
        if {...} then
          EXIT           {{exit LOOPB}}
        elseif {...} then
          EXIT (LOOPA)
        fi
        {more in LOOPB}
      od
      {more in LOOPA}
    od
```

14. Extended Examples

These examples will define part of a Programming System, and are similar to those that the Administrator would define.

14.1 Equivalence Module

We want to specify that equivalence and non-equivalence can be invoked by the familiar "=" and "<>" operators, and are valid operations on any two data objects that are type equivalent.

We are assuming that equivalence is a fundamental operation of the Programming System, and there is no need to provide an explicit algorithmic description. This illustrates a major difference between a Design Language and a Programming Language.

```
mod EQUIV
-----------------------------------------------------------------

data EQUIV
  infix '='  = EQUIV:EQUAL
  infix '<>' = EQUIV:NOTEQUAL
atad
-----------------------------------------------------------------

func EQUAL (X, Y)
  form X unbound               {{allow any type}}
  form Y typeof (X)            {{require to be same
                                 type as X}}
  returns bool
  use alt EQUIV

  if {X is equal to Y} then
    return (TRUE)
  else
    return (FALSE)
  fi
cnuf
-----------------------------------------------------------------

func NOTEQUAL (X, Y)
  {{similar to EQUAL}}
cnuf

dom
```

14.2 LIST Module

We wish to define a fundamental data type "LIST" to be an ordered sequence of data objects that has some "current valid (nonnegative) length". The generic function "SIZE" will evaluate to this length (but not change it), the generic Procedure "CLEAR" is used to set the length to 0, and the generic Procedure "PUSH" appends a new member to a LIST using the infix Procedure operator "^". "LAST" is an access Function used to access the last member of a LIST, and "SELECT1" provides access to the "N'th" member of a LIST, where "N" is greater than 0. "CONCAT" concatenates two lists.

"ASSIGN" follows the philosophy of encorporating a Procedure to explicitly define assignment for each type in the Programming System. Each type, defined or built in, should have a generic "ASSIGN" Procedure defined for it, and will be invoked by the infix operator ":=".

Most of these Routines are generic because the operation they describe has application to other defined data types (e.g., stacks, queues, sets, arrays, etc.). Likewise, the PUSH and CONCAT operators ("^" and "&") are defined in an implicitly accessable Data Segment (SYSTEM:COMMON) because they are generic to several data types.

```
  ------------------------------------------------------------------
 !
 !  mod LIST
 !    use alt LIST                    {{give all segs ALT access}}
 !
  ------------------------------------------------------------------
 !
 !  data LIST                         {{define the type LIST}}
 !    type LIST (A) =
 !      record
 !        LEN: int
 !        BODY: seq (<A>)
 !      drocer
 !  atad
 !
  ------------------------------------------------------------------
 !
 !  *func SIZE (X)                    {{get current length of list}}
 !    form X list (unbound)           {{accept LIST of any type}}
 !    returns int
 !
 !    return (X.LEN)
 !  cnuf
```

```
! *func CLEAR (X)                {{clear/initial a list}}
!   form X list (unbound)
!
!     X.LEN := 0                 {{set length to 0}}
! cnuf
!
! access *func SELECT1 (X, N) {{Selector Function for LISTS}}
!   form X list (unbound)
!   form N int
!
!     if {N is less then 0 then
!       {call a system error, & abort}
!     else
!       return (X.BODY [N])
!     fi
! cnuf
!
! access *func LAST (X)          {{return ref to last mbr of LIST X}}
!   form X list (unbound)
!   returns typeof (X.BODY [])
!
!     if X.LEN=0 then
!       {call a system error routine & abort}
!     else
!       return (X.BODY [X.LEN])
!     fi
! cnuf
!
! *proc PUSH (X, MBR)            {{append MBR onto LIST X}
!   form X list (unbound)
!   form MBR typeof (LIST.BODY [])
!
!     X.LEN := X.LEN + 1
!     LAST (X) := MBR
! corp
!
! iter *func INORDER (X)         {{in-order iteration for LISTs}}
!   form X list (unbound)
!
!     do for I = 1 by 1 to SIZE (X)
!       return (X [I])
!     od
! cnuf
```

```
 *proc ASSIGN (DST, SRC)       {{assignment for LISTs}}
   form DST list (unbound)
   form SRC typeof (DST)       {{require types to be same}}

     SRC.LEN := DST.LEN
     do for I = 1 by 1 to SRC.LEN
       DST [I] := SRC [I]
     od
 corp
```

```
 *func CONCAT (X, Y)
   form X list (unbound)
   form Y typeof (X)
   decl Z typeof (X)

     CLEAR (Z)
     do foreach X1 in INORDER (X)
       Z ^ X1                  {{same as PUSH (Z,X1)}}
     od
     do foreach Y1 in INORDER (X)
       Z ^ Y1
     od
     return (Z)
 cnuf

 dom
```

14.3 SYStem Module

The purpose of the SYStem Module is to make certain concepts globally available to the Programming System.

```
 sys mod SYSTEM
   use   LIST:LIST, EQUIV:EQUIV, COMMON
   gener LIST, EQUIV
 dom
```

```
data COMMON
   infix 1 '&' = func CONCAT
   infix 1 '^' = proc PUSH
   infix   ':=' = proc ASSIGN
atad

dom
```

15. Common Errors and Pitfalls

There are several points in the FLEX language which may easily be a source for programmer error, often because the FLEX processor has difficulty in detecting the error. Listed below are a variety of miscellaneous errors and pitfalls that may befall the unwary FLEX user.

1) The RETURN (<exp>) and EXIT (<loop id>) statements must have their arguments enclosed in parenthesis. This is a syntactic parsing requirement caused by the exclusion of statement separators. The Procedure RETURN with no expression requires no parenthesis.

2) Statements directly after a RETURN or EXIT statement in a statement list are unreachable and not allowed.

3) Routine invocations require the parenthesis present if no parameters are passed. This was done so that code readers and the FLEX processor could distinguish a Routine call from a simple variable.

4) The omission of the Functional case or Data Segment terminator words (i.e., "cnuf", "esac", "corp", "atad") can cause the omission of large parts of the source code (the parser skips over large bodies of code looking for these words as stopping flags).

5) The symbolic operator "-" is unique in the FLEX processor because it is considered both a prefix and an infix operator. The user should use this symbol for the common infix arithmetic operation of negation. The PREFIX, "+" operator is not built into FLEX and cannot be defined as both an INFIX and a PREFIX operator.

6) Id operators may not be referenced by the Module:Segment:Identifier sequence as can variables and type definitions. The simplicity and convenience of these shorthand Routine call operators is lost if the Module and Segment name precedes the identifier.

7) Loop Escapes must appear after all other loop introduction clauses and just before the loop body.

8) The default FIX/ALT attribute for the first actual parameter expression for Procedures invoked using an infix or prefix operator is ALT. This is done in deference to the assignment Procedure so that the first argument would not have to be

always prefixed by an ALT. This is the only instance in the language where the FIX/ALT attribute defaults to ALT.

9) The data type "STRING" must be defined if string constants ("'abcd'") appear in the text (e.g., 9.4). The string definition cannot require the MOD:SEQ names to precede it. A CHAR is not the same type as a STRING of one character, just as an INTeger type is not the same type as a SEQuence of INTegers.

10) A space should appear between an infix and a following prefix operator (e.g., "(A+ -B)").

16. Reserved Words

The following is a list of the reserved words of the FLEX language. User-defined identifier operators in the current Environment are considered reserved words.

ACCESS	Function
ALT erable	FIX/ALT attribute
ATAD	end of DATA
BOOL ean	
BY	
CASE	
CHAR acter	
CLOSED	Routine scope
CNUF	end of FUNC
CONST ant	data declaration
CORP	end of PROC
DATA	Data Segment
DECL are	dynamic data declaration
DO	loop statement
DOM	end of MODule
DROCER	end of RECORD
ELSE	in IF statement
ELSEIF	in IF statement
ESAC	end of CASE
ESCAPE	Generic Escape declaration
EXIT	a loop
EXPORT	some Segments
FALSE	boolean falsity
FI	end of IF statement
FIX	FIX/ALT attribute, scope limit
FOR	loop clause
FOREACH	loop clause
FORM al	FORMal parameter type spec
FUNC tion	
GENER ic	declaration
IF	statement
IN	FOREACH ... IN ...
INCL ude	add environment
INFIX	operator
INT eger	
MOD ule	
OD	end of DO loop
PREFIX	operator
PROC edure	
REAL	type
RECORD	type
RETURN	statement
RETURNS	RETURN type spec

SCALAR	type
SEQ uence	type
STAT ic	data object declaration
SYS tem	global Module
THEN	in IF statement
TIMES	loop clause
TO	
TRUE	boolean truth
TYPE	type definition
TYPEOF	type pseudo func
UNBOUND	unbound formal parameter, Form formal
UNLESS	loop clause
UNTIL	loop clause
USE	use environment
USING	loop clause
WHILE	loop clause

REFERENCES

1. Sutton, S. A., "FLEX: A Flexible, Automated Process Design System", University of Maryland M. S. Thesis, College Park, Maryland, May 1979. (Copy available from S. Sutton, CODE 8433, Naval Research Lab, Washington, D.C. 20375.)

Appendix 1

USE OF THE FLEX PROCESSOR

This appendix contains general instructions for using the FLEX processor. The other appendices should be consulted for current status of the Processor, or restrictions imposed by particular installations.

1. General

1.1 Terminal Interaction

The user creates his design text in named files, where the definition of a "file" may depend on the particular computer system. Although file boundaries can occur between any two tokens in the text, it is good practice to keep each Module on a separate file.

The FLEX processor runs as an interactive program. It first asks for a list of file names to be processed, and will process these in the order entered. File names are terminated by entering a blank line or an asterisk ("*").

After the low and high pass numbers are requested, options can then be entered one at a time, and terminated by a blank line or an asterisk ("*").

All information input from the terminal should have no preceding blanks, and no blanks should appear within identifiers.

There is a special HELP printout that prints a summary of acceptable input information any time the word "!HELP" is entered as a file name or option.

1.2 Passes

The FLEX processor has three passes that can be run independently, assuming that earlier passes have been run at least once. In general, one pass should be made error-free before continuing to the next, or phantom errors may result.

Pass 1 looks only at Module headers, Data Segments, and Routine declarations (except for local declarations DECL, STAT, and CONST). Pass 1 need only be run if a change is made to these structures, or a new Module or Segment is added.

Pass 2 looks at the same parts of the text as Pass 1, and must be run between passes 1 and 3.

Pass 3 looks at everything not looked at by the two previous passes, i.e., local Routine declarations, and the Routine text body.

Passes 1 and 2 will be run mainly in the early stages of a topdown design when the upper level structure is being defined, and pass 3 in later design phases when the algorithmic body of the Routines in being added.

1.3 LOCKed Modules

A subset of the total Module set can be submitted to the Processor, and the rules for forming this subset can be cast in a rigorous form.

Consider "ALL" to be the set of all Modules currently in the Programming System, "LOCKED" to be the subset of ALL that have the property of being "locked", and "RUN (i)" as the subset to be presented to the FLEX processor for pass "i" (i=1, 2, or 3).

The Processor stores information from one pass in an IMAGE for use by the next, and creates IMAGEn files for this purpose (1.5). Consider "IMAGE (n)" (n = 0, 1, 2, 3) to be a subset of ALL for pass "n". IMAGE (0) is the image present before pass 1 and is always equivalent to LOCKED.

Each pass "i" produces an IMAGE (i) consisting of the union of LOCKED and RUN (i), and Modules presented to subsequent passes must be present in this IMAGE (i). Hence, RUN (j) must be a subset of IMAGE (j-1), for j = 2,3. However, for passes 1 and 2, Modules cannot be present in RUN if they are locked, hence the intersection of RUN (i) and LOCKED must be null, for i = 1,2.

The "NEW" option has the effect of clearing IMAGE (0) and LOCKED before pass 1. The option "LOCK" has the effect of adding RUN (2) to LOCKED.

Note that pass 3 can always be run alone, so long as RUN (3) is a subset of IMAGE (2).

Any Modules or Segments referenced but not defined will be considered an error, unless the STUB option is in effect.

1.4 Rerunning A Programming System

When changes are made to the program text, some minimal earlier pass has to be run. The following table should be consulted for types of changes and the minimal pass to be rerun:

Change	Pass
anything in a CASE body, or Routine local decl (DECL, STAT, etc.)	3
FORM or RETURNS type spec, or type spec in DATA Seg	2
new FORM, or type or oper definition, or anything else	1

1.5 File Names

The FLEX Processor keeps a current image of the Programming System on binary files named "IMAGEn", where n is 1, 2, or 3 for the pass number. (The IMAGE (0) referred to above is kept on file IMAGE1.) Pass 2 uses the image file produced by pass 1, and pass 3 that of pass 2. The user should not have to access these files, except to save or restore them if the computer system so requires.

If the "MAP" option is enabled, the Processor will produce map file named "MAPn" (n = 1,2,3) after each pass. Most of the internal Processor tables are printed on this file, and this can be of great help to the Caretaker when debugging the Processor. The general user should have little use for the MAP file since much the same information is displayed using the STATistic option. These MAP files need not be saved or restored.

2. Options

There are several options in the FLEX Processor, where each is known as an identifier (e.g., "ABCD") that may be preceded by an "X" to turn it off (e.g., "XABCD"), and only the first 4 characters of the option are significant. The "!HELP" command (1.1) will list the current options and their defaults.

!HELP

Invokes the HELP printout to inform the user of the rest of the options and their current defaults.

INTCheck

Enables type and FIX/ALT attribute checking for Routine calls. If disabled, called Routines are assumed to exist, and Functions evaluate to the wildcard type.

TYPE

Enables the type checking mechanism in the FLEX processor which is manifest almost entirely in Routine interface type checking. "XTYPE" implies "XINTC" regardless of the state of the "INTC" option.

STUB

Routines that are referenced but not defined will be considered as stubs, and will not considered an error. Stubs are summarized on the cross reference listing.

SYNTax

This option can be used only in passes 1 or 2, and disables all semantic processing; the portion of the design text looked at during passes 1 and 2 is simply parsed.

MULFunc

Check for the existence of more than one CASE that will satisfy a generic Routine call.

MULDeclaration

Check for the existence of more than one Element that can be referenced by the same identifier sequence in a given Environment.

LOCK

Locking is intended so that Modules that are effectively completed and correct need no longer be processed with the other Modules. Modules can only be unlocked by running the FLEX processor with the "NEW" option.

Locking occurs at the end of pass 2, so that passes 1 and 2 must be run if the "LOCK" option is used (pass 3 is optional), and locking will not occur if any errors or undefined references occur during pass 1 or 2. Hence the group currently being locked cannot reference (USE or INCLude) any Modules outside themselves.

NEW

Initializes (clears) the locked subset (LOCK (i) in 1.3) before pass 1. "NEW" is usually used in conjunction with "LOCK" to clear the image before a completely new set of Modules are locked.

PAUSe

Causes the FLEX processor to pause after each error message.

QUIK

Causes the scanner to operate in "quick scan" mode which speeds up the FLEX processor, and cures other troublesome problems. If certain reserved words (ie. "corp", "cnuf", "atad", "esac", or "mod") are omitted or misplaced in the FLEX code, large chunks of code may be skipped in parsing. The QUIK option should always be used.

STATistics

The statistic option causes a variety of statistical information and a cross reference listing to be printed at the end of pass 3, and requires that all three passes are run.

BUG1

Of use to only the FLEX caretaker, and causes a variety of information to be printed during processing that aids in debugging the FLEX processor's source code.

MAP

Prints a system map of the image file after each pass (see "FILE USAGE" above). (1.5).

UPPEr case

Converts all lower case in the program text to upper case before processing, but does not alter the original program text files. In effect, the text is seen through "upper case glasses". This option may be restricted for certain installations of the Processor, and the other Appendices should be consulted for more information.

3. Using the Options

The user can always use the default options and all three passes. However, judicious use of the options can decrease processor time. An environment employing a librarian is an excellent one in which to use the FLEX processor since that librarian can become proficient in applying the options. Hopefully, the design environment, with its emphasis on code reading, walk-throughs, and other software validation techniques will result in fewer invocations of the FLEX processor than a compilation environment where frequent compilations during debugging and testing are required.

The largest time savings comes from running some subset of the three passes, (1.2).

Another significant savings comes from LOCKing completed Modules early in the design process (1.3). Although locked Modules can contain no external references outside the locked set, the Routines need have no code body, and can be simply stubs with their interface conditions defined.

The SYNTax option can be used for passes 1 and 2 and will detect syntax errors without the overhead of the semantic processing. To produce a syntax-only parse for pass 3, disable all options except QUIK, and possibly PAUSE.

The MULfunction and INTCheck options can usually be run only on occasion, and need not be run unless new Routine invocations are added to the system. Likewise the MULDeclaration option can be used occasionally, and need not be used unless a new Element or access declaration is added.

STATistics and LOCK should only be run on error-free Modules. Before a Module can be declared as error-free it should have the following options run in all passes: INTC, TYPE, MULD, MULF, and XSTUB.

4. Errors

4.1 Errors, Warnings, and Notes

There are three classes of messages in the FLEX processor: ERRORS inform of situations that are definitely in error, WARNINGS inform of situations that are not in error, but may readily lead to errors if not attended to, and NOTES refer to the quality of the programming practice or design.

These messages may contain the user identifiers associated with the error, and a code number that reveals the source of the message in the FLEX Processor. The offending line will usually be printed, and, below that, a flag line where a single pointer indicates the token last scanned. The error will either be at or before this location. Some error messages may give only the first 8 characters of an offending identifier.

4.2 Error Recovery

The FLEX processor implements error recovery (not error correction) techniques. Semantic error recovery usually involves assigning some dummy value (e.g., the wildcard type).

Syntax error recovery tries to jump over the remainder of the offending statement or clause before continuing. This will usually recover successfully without phantom error messages, and is enhanced if the user is using "good" (i.e., readable, nice-looking) formatting techniques in his programs.

4.3 Special Errors

There are a variety of special error detections in the FLEX processor.

The "BAD IMAGE" error usually results when the user changes a part of the text but does not rerun a low enough pass. If this error is encountered during an otherwise error-free run through all passes, contact the FLEX caretaker.

A "FLEX PROCESSOR ERROR" will be logged if the FLEX processor detects flaws in its internal tables. This is (hopefully) a rare error and requires the immediate attention of the FLEX caretaker.

OVERFLOW errors occur when the internal tables of the FLEX processor fill up. This particular error message will give the name of a source code "PARAMETER" that should be increased, after which the FLEX processor can simply recompiled and run. This is a relatively simple job for the FLEX caretaker.

There are a number of special error Routines that print out more extensive error messages to the user and most of these are self explanatory.

Appendix 2

THE CARETAKER'S MANUAL

The "Caretaker" is the person responsible for in charge of the FLEX Processor software, and this manual contains the information necessary for the Caretaker to install, maintain, and modify this software.

1. Processor Design

The design of FLEX processor is similar to that of a language compiler except that no code is generated. The internal tables of the Processor are kept between passes in the IMAGEn files, and contain information on all of the Modules currently in the Programming System.

1.1 Parser

The parsing algorithm is a flexible, table-driven, stack-based LL (1) parser whose tables are automatically generated from a Symbolic Syntax Definition (SSD, 5.1). Syntactic error recovery is an integral part of parsing, and is defined by special markers of the SSD.

1.2 Scanner

The scanner is ordinary except that some of its tables are maintained and manipulated by the semantic routines (e.g., for user defined operators, which behave as reserved words and symbol strings of the syntax).

1.3 Semantic Routines

Most processing is done by the semantic routines. These are parameterless subroutines whose calling order in the parsing sequence is defined by the SSD. Flags that indicate calls to the various semantic routines are deposited on the LL (1) parsing stack along with nonterminal, tokens, and other markers. A particular semantic routine is called from the parser when it appears on the top of the main LL (1) parsing stack.

Since the semantic routines are parameterless, they communicate through data bases that keep the current state of the semantic processing.

1.4 Service Routines

Service routines maintain the various data bases, and may be called by many semantic routines to access the information in the data bases. For example, the symbol table has various routines for making new entries and performing searched on its entries.

1.5 Cache Memories

A small "cache memory" is kept to speed up access to the symbol table. The previous "n" successful symbol table locations are kept in a small cache table, and always checked before a symbol table search.

2. System Map

The Processor Fortran source code is organized in the following modules, which appear as separate files on most computers. The code is self-documenting, and should be consulted for detailed information.

All files with the suffix ".DATA" (which have been shortened to 6 characters for use on certain computers) are data common blocks, and are referenced through the source insertion (inclusion) features of the host computer. When a subroutine in the source code is referred to, it is often cited as the module name followed by its subroutine name (e.g., "MODNAME:FUNCNAME").

Parsing & Control

SYSCOM.DATA (SYSCDT)

System-wide common, referenced by all routines in the system, containing pass number, options, etc.

PDL

The main program, responsible for operator interaction, querying of file names, and option processing. Calls on "PASSES" to process the user's files.

PASSES

Global control of the three passes of the FLEX processor, and the manipulation of image files and map files.

ERRORS

The error handling routines.

PARSER.DATA (PARSDT)

Data for the parser, i.e., LL1 tables and main parsing stack. Block data (BNF.DATA) for this data base is automatically generated by the BNFGEN system.

PARSER

Main LL1 parsing algorithm, and quick scan routines.

DIRECT

Called by the parser and in turn calls a specific semantic routine. The source code for DIRECT is automatically created by the BNF processor.

SCAN.DATA (SCANDT)

Scanner data for internal use by scanning routines.

TOKEN.DATA (TOKEDT)

Contains information on current token. Read by many routines, changed only by SCAN routines.

SCAN

The scanner and error recovery ("RECOVR") routines. These routines reference OPRTAB.DATA for distinguished identifiers and symbols.

IO.DATA (IODT)

I/O information (file names, logical units, etc.).

IO

General I/O routines for operations on files (opening, closing, etc.), and dependent upon the host computer.

FILER

Reads and writes the current FLEX image to and from the image files, relying on "IO" for host-dependent file operations.

STATS.DATA (STATDT)

Statistical data. Many routines in the FLEX system make entries into these tables.

STATS

General routines for the manipulation, processing, and disposition of the data in "STATS.DATA".

Semantic Routines

MODULE

Semantic routines for the upper level parsing of modules. Mainly set up the various module tables in "MODSEG.DATA".

NEWSEG.DATA (NEWSDT)

Data used mainly by "SEGMEN" during the upper level parsing of segments. This data is mainly temporary and not as permanent as that in MODSEG.DATA.

SEGMEN

Semantic routines for the upper level parsing of Segments. Mainly, set up the segment tables in "MODSEG.DATA".

NEWDCL.DATA (NEWDDT)

Data used mainly by "DECL" during the parse of declarations. This data is mainly temporary and not as permanent as the main symbol table ("SYMBTB").

DECL

Semantic routines for processing declarations, and setting up the symbol table ("SYMBTB").

TYPDEF

Semantic routines for processing type definitions. The routines for manipulating completed types are contained in "TYPTAB".

GENSTMT

Semantic routines for parsing and processing of general statements (IF-statement, assignment, RETURN-statement, etc.).

LOOP.DATA (LOOPDT)

Data used by "LOOP", mainly the loop stack where information describing the current loops is kept.

LOOP

Routines to process loops (iterations, "DO") including the loop clauses and EXIT statements.

EXPSTK.DATA (EXPSDT)

Data stacks for the keeping of expression, function call, and side effect data. Used by "EXTSEG" for functional interface and side effect checks.

EXP

Semantic routines for the parsing and processing of general expressions.

Table Access & Service Routines

MODSEG.DATA (MODSDT)

Tables for information concerning current modules and segments in the processing system.

MODSEG

Routines for accessing the "MODSEG.DATA", including search and insertion routines.

TYPETB.DATA (TYPEDT)

All type information is kept in this table.

TYPETB

Routines for manipulating the type table, including type equivalence checking, the resolution of forward type references, etc.

PSTYPE

Routines for manipulation of psuedo types.

SYMBTB.DATA (SYMBDT)

The main symbol table, including all data base declarations, routine interface information, and the access (USE/INCLude) structure of the Programming System.

SYMTAB

Routines for manipulation the symbol table.

OPRTAB.DATA (OPERDT)

Table for operator definitions, including the functions they represent, and the distinguished identifiers and symbols used by the scanner.

OPERS

Routines for the manipulation of the operator tables. (Parsing of operator definitions done in "DECL".)

TRECHK

Routines for checking conflicting Element names.

EXTSEG

Routines for Routine interface checking.

SCAFFOLD

Routines for debugging the FLEX processor that print the various internal Processor tables.

DATA

Compile time data for all data bases.

BNF.DATA (BNFDAT)

Compile-time data for the various scanner/parser tables (see PARSER.DATA). This source is mechanically produced by BNFGEN.

BNF

The user-supplied symbolic source from which the syntax generator "BNFGEN" creates the parser and scanner tables. This file contains the SSD and is not a part of the FLEX Fortran source code.

3. Fortran Programming Conventions

The FLEX processor was coded into ANSII-66 Fortran so that it could be easily implemented on a wide range of computer systems. Certain conventions have been followed in writing the Fortran code and these are documented below.

The Fortran code does not assume that variables are initialized by the compiler, nor that locally declared data objects in subprograms remain intact between calls to that subroutine. It does assume that common block data objects are static, and are never refreshed or changed except by the user's code. The Fortran code does not change the value of DO indices during the execution of a loop since this is restricted on many systems.

Array locations are always addressed by the "(<var> + <const>)" format, although actual parameters in subroutine calls may be generalized expressions.

Tables in the FLEX processor are usually arrays that behave as lists. They have some current length, where all entries from the first array location through the current length are "valid", and the rest of the array is "empty" (not used). The current length of the list is kept in a variable whose name is formed by an "X" preceding the name of the array (truncated to 6 characters). The maximum size of the array is kept in a PARAMETER variable whose name is the same as that of the array except prefixed with a "Q". The array is dimensioned using this parameter, and its size can be increased by changing the size of this "Q" parameter.

All table and array lengths are checked by the FLEX processor itself so the Fortran compiler need not generate array bound checking.

The locked portion of each table is that from the first location through a "V" variable (whose name is formed similarly to the "X" or "Q" variables).

Two-dimensional arrays used as tables use the convention that each column is an entry, and the different rows in a column are the various parts of the entry.

Identifiers are kept in packed character format, four characters to each integer array location, with rightmost characters blanked. Three array locations are needed for the 12-character identifiers used in the FLEX language. When identifiers are stored in two-dimensional arrays, they are stored column-by-column (ie. "ARRAY (i,j), i=1,3" is the j'th id in the array) so that subroutines may receive the identifier as a singly dimensioned array. Note that this requires the host Fortran compiler to store two-dimensional arrays column-by-column.

Flags are often packed character formats with 4 characters per integer array location (left justified, blanked right). This provides for more readable and faster code, although space is sacrificed over a bit-encoded scheme.

The "unpacked" character format referred to in the FLEX source simply means that characters are stored one per word (left justified, blanked right).

Fortran standard READ and WRITE statements are used in several locations in the FLEX source code where formats are always provided and logical units are integer variables or PARAMETER constants.

All variables are explicitly declared, and most are followed by a short end-of-line comment describing their use. The variables "I", "J", "K", etc., are used mainly as DO-loop indices or temporary variables.

All code that is likely to vary from one host computer to another is flagged by the end-of-line comment containing "-HOST-". If this tag follows the subroutine/function declaration statement then the whole subroutine is likely to be host-dependent.

4. I/O

I/O is likely to be the most host dependent portion of the FLEX processor. The routines for common file operations, such as opening for reading/writing, closing, or rewinding, are kept in the I/O module. The modules PASSES, SCAN, and FILER are the main modules that make I/O calls.

The FLEX processor assumes that there is a user terminal logical unit from which it receives high level information and prints various status messages. Error messages are logged on a separate logical unit (which may be the same as the terminal unit).

5. Debugging Tools

The module "SCAFFOLD" contains a variety of debugging subroutines for displaying the contents of the various tables within the FLEX processor. The "MAP" option produces a separate map file at the end of each pass that displays the various tables kept in the "IMAGE" files and can be of use to the caretaker. In addition, a call to "SCAFFOLD:SHOWXX" will dump this map image to a file name specified in the calling sequence.

The run option "BUG1" will cause a variety of debugging information to be printed to the user terminal, include the printing of calls to the semantic routines.

There are vestiges of useful debugging aids within the FLEX source code that have been turned off in some manner, usually by making them comments, and these locations are noted with the end-of-line comment containing the string "DEBUG".

6. BNFGEN & SSD

All internal parsing tables are generated by a separate system named "BNFGEN". BNFGEN accepts a SSD file, and outputs tables as Block Data Fortran subroutine source programs to be included in the compilation of FLEX processor. In addition, BNFGEN creates the source code for the "DIRECT" subroutine.

6.1 Symbolic Syntax Definition (SSD)

The Symbolic Syntax Definition is created as a text file (using perhaps the computer system editor), and processed by BNFGEN.

The format of the SSD can itself be specified in a simply BNF form. Blanks are significant characters in this definition, and are indicated below with the "_" character. Each line must have at least 12 characters, filled out with blanks, if necessary. Lines not beginning with the character string "C<>_" are ignored by BNFGEN and may contain comments. Comments can also be place in-line starting in column 25 or further. All identifier lengths mentioned below are maximum lengths.

```
<input file>     ::= {{ <prod> }}

<prod>           ::= PROD_<prod name>
                     {{ <alts> }}

<alt>            ::= ::=
                     {{ <alt part> }}

<alt part>       ::= NONT_<prod name>
                 ::= FUNC_<func name>
                 ::= MARK_<mark name>
                 ::= TOKE_<token>
                 ::= TKMK_<token marker>
                 ::= RSWD_<reswd name>
                 ::= E
```

```
<prod name>      ::= /* 12-character id */

<func name>      ::= /* 6-character semantic function */

<reswd name>     ::= /* 8-character reserved word */

<mark name>      ::= EROR
                 ::= P1ON
                 ::= P1OF
                 ::= P2ON
                 ::= P2OF
                 ::= P3ON
                 ::= P3OF

<token>          ::= /* 4-char token, see "SCAN.DATA" */

<token marker>   ::= 0
                 ::= 1
                 ::= 2
```

An existing SSD (e.g., "BNF") is probably the best example of this format, and should be studied carefully. An additional example of this format is given below, where there is no particular meaning attached to this production.

```
C<> PROD FOOPROD
C<> ::=
C<> NONT FUMPROD
C<> RSWD MODULE
C<> FUNC FOOP01
C<> TKMK 1
C<> TOKE ID
C<> TOKE (
C<> MARK EROR
C<> MARK P1ON          THIS IS A COMMENT
C<> ::=
C<> NONT FIEPROD
C<> ::=
C<> E
```

The PROD line defines the beginning of a new production, and two productions cannot have the same name.

The NONT line names a nonterminal that must be defined elsewhere as a PRODuction.

The FUNC line names a semantic function. A parameterless call to this subroutine name will be generated during the parse of the FLEX program when this symbol is encountered in the parse. Semantic routines are usually named for the first 4 characters in the production name in which they are contained, suffixed with a number (e.g., "01", "02", etc.).

The MARK line indicates a special parsing marker. The MARK EROR is a LL (1) stack error marker used during error recovery (see "SCAN:RECOVR" in FLEX source). The MARK P1ON, etc. lines are used to turn OFF or ON all semantic routines during certain passes (see "PARSER:PASSON").

The TKMK line must be followed by the TOKE line and assigns the "token marker value" to that token to be used in error recovery (see "SCAN:RECOVR" in source code). If no TKMK line precedes a TOKE line, then that token gets a token marker value of 0.

The TOKE line defines the 4-character token and the value of the token may be anything allowable in "TOKEN" in "TOKEN.DATA". A given token may be used in more than one place in the syntax. After the parse of a token, the scanner deposits the information concerning the token into the data base "TOKEN.DATA", and this information is not changed during contiguous, subsequent semantic routine calls following the token in the syntax definition. This means that these contiguous, subsequent semantic routines have full access to the preceding token.

The RSWD line defines a reserved word, and a particular reserved word may be used in more than one place in the BNF definition.

The E line denotes an empty alternation ("e-production"). It can appear only once per production, and must be the last alternate.

6.2 Error Recovery

Error recovery is implemented by the MARK EROR and TKMK 0/1/2 lines in the SSD (above). These constructs must be placed in the BNF definition with a knowledge of how error recovery is accomplished, which is explained in the FLEX source code in "SCAN:RECOVR".

6.3 Pass ON/OFF Flags

The PnON and PnOF ("Pass n ON/OFf", where n = 1, 2, or 3) are used to shut off all semantic routines during portions of a pass. PnON will increment the "SEMFLG" during pass "n", while PnOF will decrement it (where "n" is 1, 2, or 3 for the pass number). No semantic routines will be called if "SEMFLG" is less than or equal to 0.

6.4 Listing

BNFGEN produces a detailed (but not very "pretty") listing of the SSD, including the names of all tokens, productions, semantic functions, and all LL (1) token tables.

6.5 Pretty Printing

The independent program "BNFLST" will scan the SSD file, and create a nicely formatted version which is considerably more readable than the original SSD.

7. Statistics of the Source Code

The current version of FLEX has the following approximate statistics. These reflect only the FLEX processor and not the various support software, such as BNFGEN or the symbolic syntax definition file, BNF.

modules:	25
subprograms:	231
databases:	15
total source lines:	14,000
executable Fortran stmts:	3,000
comment lines:	7,700
of these, non-blank:	3,600
subprog local variable id's:	800
common var id's:	250
man hours:	600 (approx)
feet of printout:	200

Appendix 3

STATUS AND UNIMPLEMENTED FEATURES

1. Status

The current version of the FLEX Processor (FLEX 1.4) has been installed on the PRIME 400 system at the Naval Research Laboratory, and is being installed in the Univac 11XX system at the University of Maryland.

2. Unimplemented

The following features have not yet been implemented, although implementation is in progress unless otherwise noted.

1. Alternation within Parameterized Type Macro formal parameters is accepted correctly, but not checked when instantiated (9.1).

2. Type Space Execution [1] is not implemented, and the following interim rules apply.

Unbound types (including pseudo-type functions) in Routines work fine for interface checking to that Routine. However, when used in the code body of that Routine, any unbound types are considered as wildcard types, and a degree of type checking is lost. Segments in which such type checking is lost are specially flagged to notify the user that special caution is required.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The FLEX System: User and Caretaker's Manual		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER TR-765
7. AUTHOR(s) Stephen A. Sutton		8. CONTRACT OR GRANT NUMBER(s) AFOSR-77-3181A
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Maryland College Park, Maryland 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Math. & Info. Sciences, AFOSR Bolling AFB Washington, D. C. 20332		12. REPORT DATE June 1979
		13. NUMBER OF PAGES 71
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

18. SUPPLEMENTARY NOTES

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

Process Design Language, design notation, automated processor, user's manual, software design tool

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

The FLEX Design System is a design language and its Processor that form a tool for use in computer software design activities. This report presents a detailed definition of the FLEX language, directions for using the Processor, and guidelines for installing, maintaining, and modifying the Processor software.

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)