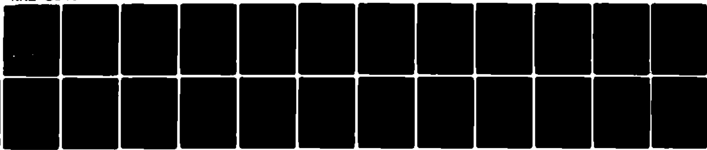
AD-A079 312 NAVAL RESEARCH LAB WASHINGTON DC F/G 9/2
FLEX: A FLEXIBLE, AUTOMATED PROCESS DESIGN SYSTEM.(U)
NOV 79 S A SUTTON, V R BASILI
UNCLASSIFIED NRL-8349 SBIE -AD-E000 345 NL

1 OF 1
AD
A079312



END
DATE
FILMED
2-80
DDC

AD-E000 345

NRL Report 8349

AD A079312

FLEX:
A Flexible, Automated Process Design System

STEPHEN A. SUTTON

Mechanics of Materials Branch
Ocean Technology Division

AND

VICTOR R. BASILI

Computer Science Department
University of Maryland
College Park, Md.

November 30, 1979

DDC FILE COPY

DDC
JAN 11 1980
A

NAVAL RESEARCH LABORATORY
Washington, D.C.

Approved for public release; distribution unlimited.

79 12 14 048

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NRL Report 8349	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) FLEX: A FLEXIBLE, AUTOMATED PROCESS DESIGN SYSTEM		5. TYPE OF REPORT & PERIOD COVERED Final report on the problem 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Stephen A. Sutton and Victor R. Basili *		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Research Laboratory Washington, DC 20375		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NRL Problem F01-04 Project RR02393
11. CONTROLLING OFFICE NAME AND ADDRESS Department of the Navy Office of Naval Research Arlington, VA 22217		12. REPORT DATE November 1979 13. NUMBER OF PAGES 29
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

*Computer Science Department, University of Maryland, College Park, Md.

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

Computer programming
Programming language
Procedure oriented languages
Computer systems programs

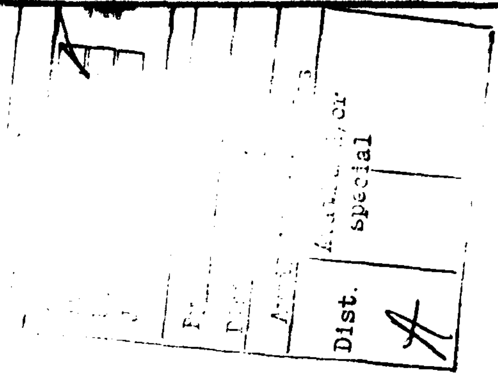
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

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

Among the features offered by the FLEX language are: a modular design structure, 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.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



CONTENTS

INTRODUCTION 1

The Process Design Language 1
PDLs and Programming Languages 1

THE FLEX SYSTEM 1

The FLEX Language 1
Outer and Inner Syntax 2
The FLEX Processor 3
Using the System 3
A "PDL Generator" 3

FLEX AND DESIGN METHODOLOGY 4

Top-down Design 4
Use in the Team Environment 4

THE FLEX PROCESSOR 4

Structure 4
Statistics 5

THE FLEX LANGUAGE 5

Escapes 5
Modular Structure 6

The Programming System 6
The Module 6
Elements 8
The Data Segment 8
The Routine 8
Routine Interface Control 9
Rules 9

Data Environments 10

Data Environment for Data Segments 10
FIX/ALT Protection 11
Data Environments for Routines 11
Routine Data Environment Restrictions 11

Use 11
Rules. 12

Data Objects 12

Storage Attributes. 12
Type Attributes. 12
Type Equivalence 14

Operators 14

Operator Definition. 14
Using Operators. 15

Generic Routines. 15

Functional Environment 15
Unbound Parameters. 15
Type Pseudo Functions. 16
Type Space Execution. 17

Statements 18

General Statements 18
Assignment 19
Iteration 19
Loop Clauses. 20
The FOREACH Clause 21
Rules. 22

Expressions. 22

General Expressions 22
Record Selection. 23
Selector Selection 23
Rules. 23

Summary. 24

FUTURE DEVELOPMENT 24

BIBLIOGRAPHY 25

FLEX: A FLEXIBLE, AUTOMATED PROCESS DESIGN SYSTEM

INTRODUCTION

The Process Design Language

Process Design Languages (PDLs) are tools that aid and control the software design process. Their use has varied from pencil-and-paper languages to automated systems. PDLs are open-ended specializations of natural language, permitting the designer to specify the early stages of a software design in a natural, English-like form with few syntactic constraints. They are implementation independent, and have application to other areas, e.g., the writing of user manuals.

PDLs have an outer and inner syntax [2,11,20]. The outer syntax is a specific set of forms used to show the general structure of a software design (data bases, routines, access paths, etc.) and flow of control within routines. The inner syntax has little syntactic or semantic constraint and is used to describe the specific data structures and actions of an algorithm. This description may be general or detailed at the designer's discretion.

PDLs and Programming Languages

Programming languages and pencil-and-paper PDLs are the extremes of a spectrum. A programming language is required to produce an executable code, and must be defined with rigorous, detailed syntactic and semantic rules. It is not a design tool, because a software design cannot be processed until the lowest levels of the code are completed. The pencil-and-paper PDL provides a freedom of expression unmatched by programming languages. However, it cannot be fully automated without a detailed syntax, and consistency must be checked by code reading.

THE FLEX SYSTEM

The FLEX Language

The FLEX system was designed to bridge the gap between PDLs and programming languages, and has features found in both. The system has a two-fold flexibility: first, the processor is designed so that the language can be configured or altered to fulfill the requirements of a particular programming environment; and, second, the language has self-extending features similar to those of modern programming languages.

Manuscript submitted June 22, 1979.

Outer and Inner Syntax

The inner syntax ("Escapes") allows an informal, natural English description of a software design. These can be used in a top-down development, where general Escapes are successively replaced by more specific ones until a given level of detail is reached.

The outer syntax has a detail similar to that of modern programming languages [4-8,13]. Its upper level is used to describe the general components of a software design:

- Programming System: The Programming System is the largest unit of a software design, and is the complete specification of a software design solution.

- Module: The Programming System is composed of a set of uniquely name Modules. The Module is the principal scoping concept in the language, and provides a protected environment for the components of a design. The Module can be used to encapsulate an abstract type definition or protected data base, or to organize logical subsystems of the Programming System.

- Segments: Each Module is a collection of named Routines and data bases (collectively called Segments). Segments have declarable, controlled access to other Segments in the Programming System, and can be made implicitly available ("global") to all other Segments in the Programming System.

The lower level outer syntax is used to specify data types, expressions, statements, etc., and contains several significant features:

- Generic Routines: Functions and Procedures may be generic, recursive, and restricted in the way they interact with their environment.

- Definable Operators: Infix and prefix operators invoke either a Function or Procedure.

- Shared Data: Data Segments may be accessed by other Segments, but protected from alteration. Many of the problems associated with shared data can be controlled by specifying the manner in which Routines can access shared data.

- Definable Types: Abstract types are definable with controlled access to their underlying form.

- Extendable Language Features: Iteration and submember selection can be extended to defined types.

- Flexibility: The user can define fundamental language features such as equivalence, assignment, and arithmetic operations.

The FLEX Processor

The FLEX processor has an omniscient view of the Programming System, and assures the consistency of the outer syntax as soon as it appears in a top-down development. The processor has an external syntax definition system so that language features can be modified, and internal features for configuring the language and enabling certain consistency checks. The processor also has open-ended capabilities for measuring statistics of the Programming System. Some of the significant capabilities of the processor are:

- Strong Type Checking using name equivalence.
- Interface Checking for each Routine call.
- Statistics generation on the Modules in the Programming System.

Using the System

The users of the FLEX system can be grouped into three hierarchical categories:

- Caretaker: The Caretaker makes fundamental changes to the language and processor. He may alter the syntax and semantics of the language by adding or restricting features to satisfy the requirements of a particular design environment.

- Administrator: The administrator uses the self-extending features of the language version provided by the caretaker to build a set of basic programming tools for the design environment. The administrator must provide fundamental operations (arithmetic, Boolean, etc.) and data types (stacks, arrays, lists, etc.). These will be global to the design environment and protected from redefinition.

- Designer: The designer is the end user of the system and sees a language that has been tailored to the solution of his particular design problem. The caretaker must decide which extensible features the designer may use; the designer can be given wide latitude to further create fundamental elements of the language, or can be restricted to a closed outer syntax.

A "PDL Generator"

A family of design languages can be created in this manner. At one end of the spectrum, the caretaker may decide to delete low-level portions of the outer syntax (statements, expressions, types, etc.). The designer would then be using a language similar to the PDL of Caine and Gordon [11] which relies heavily on Escapes. At the other end of the spectrum, the language may be configured by both the caretaker and administrator to conform to a particular programming language to simplify conversion of the final design to an executable form.

FLEX AND DESIGN METHODOLOGY

Top-Down Design

FLEX can be used as a vehicle to implement a software design methodology [2,9,19]. The caretaker and administrator are instrumental in configuring the processor to conform to any such philosophy. There may be different versions of the language for different design stages. For example, the internal structure of Routines may not be allowed until later stages of a top-down design.

The FLEX system does not impose a specific design technique on the designer, and the inner syntax of the language can be used to create special features that depend upon a common understanding among the design readers. For example, the designer grasping for a forbidden GOTO statement might surreptitiously use the inner syntax " branch to the fourth statement down from here ."

Use in the Team Environment

The FLEX system can aid the programming team environment. The processor can determine the consistency of subsystem interfaces created by different design groups, and its statistics can be used for management decisions. The system can aid in code reading by assuring the mechanical consistency of the design (types, Segment access, statement sequencing, etc.).

A librarian is important in using the FLEX system, especially if there are different versions of the processor for different stages of the design. The units managed by the librarian would be the Modules provided by the administrators and designers. The librarian would have primary responsibility for running the FLEX Processor, maintaining the various computer files, and collecting the statistics produced by the processor.

THE FLEX PROCESSOR

Structure

The FLEX processor is similar to a programming language compiler except that it does not produce executable code, and its syntax and semantics are generally more flexible. The processor contains a table-driven LL 1 parser [1] whose syntax definition tables are generated by an external system, developed for FLEX, from a user-defined syntax definition. The caretaker maintains this syntax definition, and makes changes or additions as required by the programming environment.

Like many modern language compilers, the processor does not allow separate compilations. However, the processor has three passes which may be run independently, and there is a facility for including completed Modules into "image" files so that the program text of these Modules need not be included with the rest of the Programming System. There are processor options available for disabling the consistency checking mechanisms when it is

known that there are no such errors, and many of the options that the caretaker may exercise in configuring the language are provided by a set of internal processor switches.

Statistics

The FLEX processor keeps statistics on each Module in the Programming System. Certain kinds of raw data are gathered and made available for external routines that may be created to compute specific software metrics. A crossreference listing shows the structure of the Programming System with many of the following statistics:

- Simple Size Measures reflect various size properties of the design, including the number of lines, characters, comments, escapes, etc.

- Functional Structure is given by indicating, for each Routine: the Routines it calls, the Routines that call it, and the lexicographic frequency of these calls.

- Semantic Error history is the frequency of each semantic error detected by the processor, and can be used to detect language features that are troublesome to the users.

- Syntactic Measures are the frequency with which each production in the language syntax is reduced, and the alternative that was selected. This information can be provided to external programs that, with a knowledge of the syntax definition, can compute a variety of measures of the design. These might include complexity measures or the frequency with which various statements and clauses are used [10,12].

- Environment Structure shows the Internal, External, and Parametric Environments for all Segments in the Programming System.

THE FLEX LANGUAGE

This section presents an overview of the syntax and semantics of the FLEX language with an indication of how the processor enforces these rules. Since this is a cursory description, the User's Manual [1] may be consulted for more detail.

The language terminology generally conforms to that in the literature of software engineering and programming language design. Since the processor is not a compiler, the terms "inspect" and "design text" will be used instead of "compile" and "source code" In the examples below, reserved words of the language and user names are in lower and upper case. Terms that are reserved words in the language have the reserved portion capitalized (e.g., "STATic", "INCLude").

Escapes

Escapes suspend the rigorous syntax of the language and are similar to the inner syntax of earlier PDLs. They are indicated by text enclosed in brackets ("{this is Escape text}")

and are used throughout the examples below. Escapes are an informal description of an action that will later be replaced by a more explicit description. Escapes may be used as statements, expressions, loop clauses, and type specifications.

Generic Escapes are used to indicate Escapes that are used in different parts of a Programming System with the same logical intent. The Escape text is preceded by an identifier that describes that purpose (e.g., "{ID: some text}").

Comments ("{{this is a comment}}") are a form of Escape that are allowed anywhere in the design text but are not considered "action" items; they merely provide commentary on the surrounding text.

Modular Structure

The Programming System

The largest unit of organization in the FLEX language is the Programming System. It is the description of a software design solution, and is complete in that every element referenced from within the system is also within the system. The processor maintains an omniscient view of the processing system and can assure global consistency.

A Programming System is composed of a set of uniquely named Modules, which are composed of a set of uniquely named Segments. A Segment may be either a Data Segment or a Routine, and there are two classes of Routines: Functions and Procedures.

The Module

The Module is the principal scoping concept in FLEX, and provides a protected environment for its Segments. These Segments are made visible to other Segments by an EXPORT definition in the Module header (Fig. 1). The Module header can contain the USE and GENER declarations that are implicitly added to each Segment within the Module, and there can be one SYStem Module that is used to make certain concepts (types, operators, etc.) global to the Programming System.

The Module has three typical uses:

- Data Base Encapsulation: Modules can be used to control access to a Data Segment ("information hiding"), where Routines within the Module are the only ones allowed direct access to its data. External users must rely on these Routines for information in the data base, and are not affected by changes in its internal representation. The Module "SYMBOL" (Fig. 1) encapsulates the symbol table for an example Programming System (a compiler).

- Type Encapsulation: Modules can be used for type encapsulation, which is similar to data base encapsulation except that a type definition is kept in an EXPORTed Data Segment. The EXPORTed Routines in the Module are used to directly access and manipulate data objects created in other Modules from this type definition. The external users can

```
mod SCAN                                 {{Module for scanner and its data}}
    {{Routines and data for scanner}}
dom
--------------------------------------------------------------------------
mod SYMBOL                               {{Symbol table and Routines}}
    export FIND, CLEAR                   {{Make only Routines accessible
                                           from other Modules, not the TABLE}}
    data TABLE
        {{Main symbol table}}
    atad
    func FIND (SYM)
        {Find the symbol name "SYM" in
         symbol table for the caller}
    cnuf
    proc CLEAR
        {{Clear (initialize) the symbol table}}
    corp
dom
--------------------------------------------------------------------------
mod LIST                                 {{Define "LIST" type and access
                                           Routines for objects of this type}}
    export DEF                           {{Make definition available}}
    export TOP, LEN, CLEAR               {{Make Routines available}}
    data DEF
        {An abstract type definition of "LIST"}
    atad
    closed *func LEN (X)                 {{"closed"
                                           "*func"
        {Evaluate to the length of (i.e., current
         number of members in) the list "X"}
    cnuf
    closed access *func TOP (X)
        {Return reference to last member of some list "X"}
    cnuf
    closed *proc CLEAR (X)
        {Set list "X" to empty}
    copr
dom
```

Fig. 1 — Example Programming System (a Compiler)

freely create data of this type, but may be prohibited from accessing the underlying representation. The Module "LIST" (Fig. 1) encapsulates the defined type "LIST".

- Subsystem Organization: The Module can also be used to organize the logical subsystems of a design. The "SCAN" Module (Fig. 1) contains all the scanning Routines for the example compiler design, and an "IO" Module could have been defined to contain all system-dependent I/O functions.

Elements

There are three named Elements in FLEX: data objects, type definitions and operator definitions.

The Data Segment

The Data Segment is the sole repository for shared data objects (data objects that are accessible from more than one Segment) type definitions, and operator definitions. The ability to reference the named elements of a Data Segment is termed "access" to that Segment. Data Segments and Routines may gain access to a data Segment with the USE or INCLude declaration.

The Routine

Routines are the actors of the FLEX language. They are fully recursive and can be generic in nature. They cannot contain subroutines, as in many Algol-based languages, but have a modular structure similar to the SIMPL family [8] and CLU [5,16]. The Routines are as follows:

- Procedures are not value-returning, can alter their formal parameters subject to the FIX/ALT constraint, and are invoked from a procedure call statement.

- Functions are value-returning, do not produce side effects, and are invoked from within expressions.

- Access Functions return a reference to an existing data object in the caller's Data Environment. For example, if "X" is an object of some defined type that has a meaningful "TOP" component (stacks, queues, etc.), then an Access Function "TOP" could be created to address this member and be referenced as in the assignment:

TOP (X) : = TOP (X) + 2

- Iteration Functions are used to iterated the members of a defined type, and are described in connection with the FOREACH loop clause.

Each Routine may consist of one or more CASEs, each having its own declarations and text body, where the particular CASE activated by a call to a Routine is determined by the types of the actual parameters. This allows generic Routines when there are an enumerable number of interface conditions. The "ADDITION" Function (Fig. 2) has two cases, one for integer, and one for real addition.

```
closed func ADDITION (A, B)
    case
      form A, B int                    {{declare formal parameter types}}
      returns int                      {{declare type of value returned}}

        {evaluate sum of 2 integers, "A" and "B"}

    esac
    case
      form A, B real
      returns real

        {evaluate sum of 2 reals, "A" and "B"}

    esac
cnuf
```

Fig. 2 — ADDITION Function with two CASEs

Routine Interface Control

Routine interfacing is tightly controlled in FLEX. For Procedures, each formal parameter is declared as FIX or ALT. Each actual parameter expression in a Procedure call may be prefixed by a FIX/ALT attribute (FIX is the default), which must not violate the inherent alterability of the expression. For each Procedure call, the FIX/ALT attributes of the actual and formal parameters are checked for correspondence. All formal parameters are FIXed for Functions.

A formal parameter that is FIXed cannot be altered during the course of the execution of its Routine, even by other Routines that may be called.

For each Routine call, a CASE must exist where the corresponding actual and formal types are equivalent. The RETURNS declaration declares the type that a Function returns, and each expression returned within its body must be of this type.

Rules

Segments are generally addressed by their Module and Segment names ("LIST:DEF", and "LIST:TOP" in Fig. 1) and elements by their Module, Segment, and declared names ("MODB:SEGB1:XX" in Fig. 3). However, there are instances when the Module, or Module and Segment names can be omitted for brevity [1].

The CASEs are scanned in order when determining which CASE is to be activated, and the interface type (the collection of formal parameter types) of a CASE is allowed to be a subset of later CASEs. This allows some CASEs to process certain interfaced types, leaving later ones to do "all the rest," and is analogous to the IF-ELSEIF-ELSE-FI statement where conditional cases can subsume earlier ones.

```
mod MODA                              {{Module named "MODA"}}
    func FUNCA                        {{a Function}}
        use alt MODB:SEGB1            {{gain access to Data Segment}}
    cnuf
dom

mod MODB
    export alt SEGB1                  {{make available}}
    data SEGB1                        {{a Data Segment}}
        use fix SEGB2
        incl fix SEGB3
        stat XX int                   {{a "STATic" integer variable}}
    atad
    data SEGB2
        stat YY int                   {{an integer variable}}
    atad
    data SEGB3
        stat ZZ real                  {{a real variable}}
    atad
dom
```

Fig. 3 — Example environment access

The Internal Environment of "SEGB1" (Fig. 3) consists of its local element declaration ("X") and the Internal Environment of the INCLuded "SEGB3" ("Z"). Its External Environment is the USEd Internal Environment of "SEGB2" ("YY").

Data Environments

Data Environment for Data Segments

A Data Environment is the set of elements to which a Segment has access. The Data Environment of a Data Segment is partitioned into an Internal and External Environment:

- The INTERNAL Environment contains the elements declared within the Segment, and the elements of the Internal Environments of other Data Segments accessed with INCLude declarations.

- The EXTERNAL Environment contains elements of other Data Segments accessed with the USE declaration.

The USE and INCLude declarations simply gain access to existing elements in other Data Segments; new elements are not created. The INCLude is stronger than the USE; it not only gains access, but makes the INCLuded Environment appear as if it were contained in the INCLuding Data Segment.

FIX/ALT Protection

The FIX or ALT protection attribute is attached to each element in an environment, and is relative to the environment and not absolute to the Element. For example, and element existing in the environments of different Segments may be FIXed in some, and ALTerable in others. The FIX/ALT attribute can appear in the USE, INCLude, and EXPORT declarations.

Data Environments for Routines

The environment of a Routine consists of an External, Internal, and Parametric Environment. The Internal Environment consists solely of declared local data objects (i.e., the INCLude declaration, and the type and operator definitions are not allowed). The External Environment is defined by USE declarations, as it was for Data Segments. For example, the External Environment of "MODA:FUNCA" (Fig. 3) consists of the Elements of "MODB:SEGB1" (ALTerable) and "MOBD:SEGB2 (FIXed). The Parametric Environment contains the parameters of the Routine.

Routine Data Environment Restrictions

The Parametric Environment of a Function cannot be altered, and is subject to the FIX/ALT formal parameter protection for Procedures. In addition, Routines are declared to be ALT, FIX, or CLOSED to define how they may interact with their Internal and External Environments:

- ALT PROCEDURES can alter their External Environment. (ALT Functions are not allowed since this leads to side effects.)

- FIX ROUTINES cannot alter their External Environment.

- CLOSED ROUTINES are ones whose actions depend only on their Parametric Environment (no history dependence). They cannot share non-CONSTant data with any other Segment.

Use

Shared data can lead to strong coupling between Routines, and history dependence can obscure their function. The caretaker may limit these features by prohibiting ALT USE and INClude declarations that cross Module boundaries, the sharing of all except CONSTant data, certain kinds of FIX/ALT/CLOSED Routines, etc.

Rules

One environment cannot be added to another if it has been previously added, i.e., access paths may not be recursive, and multiple access paths to an element may not exist.

Neither the External Environment of a Data Segment, nor the Environment of a Routine may be accessed with the USE of INCLude declarations. If the USEd or INCLude declaration is FIXed (Fig. 3) then the added elements all become FIXed in the new environment, but if declared as ALT, then the elements retain their previous attributes. Therefore, USE and INCLude can never gain ALT access to a FIXed element.

A CLOSED Routine can only reference CONSTant data objects in its External Environment, and STATic declarations may not appear in its Internal Environment. Also, CLOSED Routines may call only CLOSED Routines, and FIXed Routines may call only FIXed or CLOSED Routines.

Data Objects

Storage Attributes

FLEX has both static and dynamic data. Dynamic data are the normal "stack-based" data that may be declared only in Routines, and considered to be created at Routine entry and released at exit. STATic data can be shared, are never re-initialized or released, and can be declared in Routines (with possible FIX/ALT/CLOSED restrictions) and Data Segments. If static data are declared in a Routine, then all instantiations of that Routine share the data. CONSTant data is a class of static data that is FIXed in all environments.

Type Attributes

There are four types of generators in FLEX: scalar, record, sequence, and parameterized type macro. There are no dynamic types such as variant records in language, however.

- Scalars are definable types that have no definining form, and are accessable only by Routines that have some inherent "knowledge" of their internal structure. A list of unordered scalar constants may be included in a scalar definition:

```
type COLOR = scalar (RED, BLUE, YELLOW)   {{definition}}
stat X COLOR                              {{data object of type "COLOR"}}
```

This example defines the scalar type "COLOR" with three objects of that type. ("RED" bears the same relationship to "COLOR" as "TRUE" does to "BOOLEAN" for common Boolean types.) The user would presumably create Routines for manipulating data of type "COLOR". Common scalars provided by the language are: INTeger, REAL, CHARacter, and BOOLean.

- Records are Cartesian products of data objects. A data object declared to be a Record is a collection of subjects of potentially different types, and the subjects can be addressed by a named, noncomputable identifier called a selector. Selector names need only be unique within each record, and are not a part of the "type" of the record.

- Sequences are ordered sets of data objects of the same type. There is an implied mapping from the set of integers to the members of a sequence, although any restrictions on the range of the integral mapping must be built into types defined in terms of sequences (stacks, arrays of any dimension, queues, sets, etc.).

- Parameterized Type Macros are the means for type abstraction. They are type templates in which named subtypes ("formal types") are left unspecified until a particular data object of this type is created. At this time these subtypes ("actual types") are given and replace the corresponding formals. The actual type may be restricted to one of a set of specified types. Figure 4 shows a type definition ("YTYPE") and a subsequent a data object ("X") of that type.

```
type YTYPE (F1: int;  real, F2)   {{formal names}}
  record
    SEL1:  <F1>
    SEL2:  <F2>
    SEL3:  seq ( <F1> )
  drocer

decl X ytype (int, char)     {{record
                                 SEL1:  int
                                 SEL2:  char
                                 SEL3:  seq (int)
                               drocer}}
```

Fig. 4 — A type definition and data declaration

There are two formal types ("F1" and "F2") in the definition of "YTYPE", where "F1" must be either an INTeger or REAL type, and "F2" can be any type.

The internal structure of a defined type is visible from only certain environments. A Routine has access to this underlying structure if and only if it has ALT access to the Data Segment in which the type is defined. For example, the following reference to "X" (Fig. 4) is legal only if the Routine has ALT access to the Segment in which "YTYPE" is defined:

X.SEL3 : =   a SEQuence of integers

Type definitions are not allowed to be directly or indirectly recursive; the template cannot contain references to itself. However, instantiations of a type can include that type, for example, the declaration ("decl ABC ytype (int, ytype (int, real))" is legal.

FLEX allows, but does not require encapsulation of the Macro definition with the Routines that have access to the underlying structure.

Type Equivalence

FLEX is strongly type checked, where type equivalence is a "name" equivalence (as opposed to a structural one) defined by the rule:

> Two data objects are type equivalent if and only if they were created from the same type definition, and the corresponding actual types used were type equivalent.

Morris [18] presents five criteria for the implementation of abstract data types. Name equivalence fulfills the fourth requirement by making defined types truly "new," and not simply templates.

Operators

Operator Definition

PREFIX and INFIX operators are a convenient way to call Routines that have one or two parameters, respectively, and can be either symbol strings or identifiers. In FLEX, all operators and the Routines they represent must be defined by the user (often the administrator); the language provides none.

INFIX operators may invoke Procedures (as a statement) as well as Functions, and are given a priority for expression evaluation. Figure 5 illustrates some typical operator definitions.

```
infix 1  '+'  = func  ARITH:ADD {{lowest priority, will call "ARITH"
                                  in Module "ARITH }}

prefix  NOT  = func  BOOL:NOT

infix   ':='  = proc  ASSIGN    {{used in statements to call
                                  Procedure "ASSIGN"}}
```

Fig. 5 — Operator definitions

Using Operators

The administrator is responsible for defining fundamental operators and their Routines, and these will have a significant effect on the programmer's environment. Equivalence ("=", "<>"), assignment (":="), arithmetic operations ("+", "-", "/", "*", "**"), Boolean operations ("AND", "OR", "NOT"), and relationship ("<", ">", "=>", "<=") will be globally defined for most Programming Systems.

Generic Routines

Functional Environment

The set of Routines that a Routine may call is its functional environment, which is divided into specific and generic parts that are not necessarily mutually exclusive. The specific environment of a routine consists of all Routines in its own Module and the EXPORTed Routines of all other Modules. Routines in this environment are uniquely addressed by their Module and Segment name (e.g., "SYMBOL:FIND" in Fig. 1).

The generic environment of a Routine is defined by the GENER declarations in the Routine that name a set of Modules. Each EXPORTed Routine that has been declared to be generic (i.e., "*func" or "*proc") in these Modules is added to the generic environment.

Routines in the generic environment are addressed by their Segment name only. There may be several Routines of this name, but the one selected is the one that has a CASE whose formal parameters match the actual parameters in type and number. If more than one such case exists then the reference is ambiguous.

Generic Routine references (also called "overloading") can serve as simply a shortened notation for invoking common Routines. More importantly, generic Routines are ones whose functions are not specific to a particular data type. Concepts such as equivalence, assignment, and the addressing of a "member" of a defined data type are within this category. For example, a "CLEAR" procedure (Fig. 1) that sets a data object to some "initial" state is meaningful for may data types. A user group may decide to define "CLEAR" for all data types with the understanding that there is an implicit call to "CLEAR" for each DECLaration at entry to a Routine.

Unbound Parameters

The ability to create Routines whose parameters contain arbitrary types is important to the principle of type abstraction. For example, the "TOP" Function that selects the last element of a "LIST" (Fig. 1) should apply to lists of any type.

In FLEX, the type specification for formal parameters may contain unspecified ("UNBOUND") subtypes that cannot be determined until the Routine is called from a particular source. The unbound types are then replaced by the corresponding types of the actual parameters which are completely bound.

Unbound types enable Routines to be generic, but this differs from CASE alternation in that an unbounded number of interface conditions can be specified. Two examples of UNBOUND types are given in Fig. 6.

```
closed access *func TOP (LISTX)

   form LISTX list (unbound)           {{list of any type}}

   returns typeof (LISTX [int])        {{return value whose type is that of a member
                                         of the list "X"}}

      return (LISTX [LEN (LISTX)] )    {{"return" statement}}
cnuf
```

```
closed *func EQUAL (A, B)

   form A unbound                      {{allow any type here}}

   form B typeof (A)                   {{bind to actual type of "A"}}

   returns bool

      {Return TRUE if A is equal to B,
       else return FALSE}
cnuf
```

Fig. 6 — Examples of type pseudo functions

Type Pseudo Functions

Formal parameter types may depend on the types of other formal parameters. These type pseudo functions take the form:

typeof (<variable>)

The "<variable>" has the same format as a variable reference in the Routine body, except that the only identifiers allowed are those of previously defined formal parameters, and type specifications are used in place of any other arguments.

Figure 6 shows the "TOP" Access Function for lists (Fig. 1), where the expression "LISTX [<int exp>]" can be used to address a member of the list, and "<int exp>" is any integer expression. The RETURNS type is a type pseudo function denoting that the returned type wil be that of the members of "LISTX", which may vary from one caller to the next.

Figure 6 also defines the equivalence Function "EQUAL" for a typical Programming System. The type of formal "A" is bound to the type of the first actual parameter, which can be any type. The type of the formal "B" is then bound to this newly bound type of "A" and subsequently checked against the actual type of the second actual parameter. This will assure that both arguments are of the same type.

Type Space Execution

When variables containing UNBOUND types are used within the body of a Routine, there is no way to check their type when the Routine is inspected "in a vacuum" (i.e., not called from a particular source) because the UNBOUND types are not known. The Function "SEARCH" (Fig. 7) determines whether the list "LISTX" contains the member "MBR".

```
closed func SEARCH (LISTX, MBR)

   form LISTX list (unbound)           {{ list of any type }}
   form MBR typeof (LISTX [int])       {{ must be same type as objects in LISTX }}

   returns bool                        {{ return TRUE if MBR is in LISTX }}
      do foreach Y in LISTX            {{ see "STATEMENTS" below }}
         if EQUAL (Y, MBR) then
            return (TRUE)
         fi
      od
      return (FALSE)
cnuf
```

Fig. 7 — Example of type space execution

When inspected "in a vacuum" there is no way to determine whether "Y", a member of "LISTX", and "MBR" will be correctly accepted by the Function "EQUAL" because their types are not bound.

The following scheme (type space execution) provides a solution to this problem. When inspecting some Routine "X", a call is made to "SEARCH", the inspection of "X" is suspended, the actual parameter types are bound to the formal parameter types of "SEARCH" (resolving any unbound types), the interface is inspected for type equivalence,

and "SEARCH" is completely inspected with all data object types bound. This will involve the suspension of "SEARCH" and the inspection of "EQUAL" with the appropriately bound types.

In data space execution (i.e., the normal "execution" of an executable program) the values of data objects are bound at Routine entry; the Routine is executed, and a data object returned. The type space execution scheme is analogous, except that types are bound at entry, the Routine is inspected, and a type is "returned."

Although a complete discussion of how this could be implemented is beyond the scope of this paper, a Routine need only be inspected once for any given set of actual types presented. Recursion is handled by observing that if a given Routine is to be added to a string of recursive references, and it is already in that string with the same actual parameter types, it need not be added and the recursion can be "unwound." To perform a type space execution that is independent from data space execution, data types must be uncoupled from the value of any data. Essentially, the type space execution scheme is independent of the data state of the system.

Although type space execution is assumed by the FLEX language, it is not yet implemented by the processor. Interface checking to a Routine that has UNBOUND parameters or type pseudo functions is complete, but when these types are used within that Routine, their types are not known and cannot be checked (the processor notes these occurrences).

A type space execution scheme can be applied to other problems as well. For example, formal parameters can represent a renaming of data objects in the External Environment (aliasing) and this cannot be detected by inspection in a "vacuum," but can be detected during type space execution. When a data object is being iterated in a FOREACH loop, it should not be alterable by Routines within the iteration loop. This cannot be done for shared data without a dynamic inspection scheme such as type space execution.

Statements

General Statements

FLEX contains a small set of general statements:

- IF-ELSEIF-ELSE-FI statements provide conditional flow of control and are used in place of CASE statements.

- RETURN statements cause an immediate return from the Routine, and, for Functions, return a data object indicated by a general expression. An Access Function must return a reference to an object that is contained in the first actual parameter.

- PROCEDURE CALL statements invoke Procedures and may be indicated by an infix operator.

Figure 8 shows these statements where "{en}" and "{bn}" are general expressions and Boolean expressions, respectively.

```
if bi then

   MODX:PROCX (alt {e2}, fix {e3})  {{ call to Procedure }}
                                    {{ "PROCX" in Module "MODX" }}

elseif {b2} then

   return ({e5})

else

   PROCY ()                         {{ call to generic Procedure with no }}
                                    {{ parameters }}

fi
```

Fig. 8 — Example statements

Assignment

The assignment statement is created by the user (generally the administrator), and is invoked with an infix operator (e.g., ":="). Assignment between nonequivalent types and assignmentlike operation can be defined (such as the "EXCHANGE" operation in which two data objects exchange values).

Nordstrom [15] suggests that the assignment statement not be allowed to alter any data except the destination of the assignment. In FLEX this is accomplished by defining a CLOSED assignment procedure whose first formal parameter (the destination of the assignment) is ALTerable and the second (the source) is FIXed. A typical assignment Module is shown in Fig. 9.

Iteration

The iteration loop syntax offers many opportunities for the caretaker to redefine the language; the processor provides several internal switches for this purpose. The caretaker may wish to delete certain loop features (e.g., the EXIT statement), to limit the number and combinations of clauses that appear in the loop header, or to completely redefine the loop format.

The philosophy behind the FLEX iteration structure is: (a) to reduce the number of nested loops by allowing more than one iteration conditions for each loop; (b) to make the

```
mod ASSIGN
   export OP, ASSIGN                  {{ available to others }}

   data OP                            {{ define operator }}
      infix ':=' = ASSIGN:ASSIGN
   atad

   closed *proc ASSIGN (alt DEST, fix SRC)

      form DEST unbound               {{ allow any type }}
      form SRC   typeof (DEST)        {{ require same type as DEST }}

        { This Procedure copies the SRC into the DESTination so that
          the two are equivalent upon completion. }
   cnuf
dom
```

Fig. 9 — Typical assignment module

iteration conditions apparent to the code reader and unaffected by actions within the loop body, and (c) to allow iteration over data objects of defined data types.

Loop Clauses

The DO-OD loops of FLEX may be nested to any depth, be named, and contain EXIT statements that allow for immediate termination of any enclosing loop (although this violates condition c. The loop header may contain various combinations of clauses ((Fig. 10) where "{bool}", "{int}", and "{exp}" are Boolean, integer, and general expressions, respectively). The "ID" 's in Fig. 10 must be previously undefined, and are only valid for the duration of the loop. Examples of typical uses of these clauses will be shown in Fig. 12.

```
while   {bool}                         {{ test at loop begin, exit if false }}
until   {bool}                         {{ test at loop end, exit if true }}
unless  {bool}                         {{ skip current interation if true }}
times   {int}                          {{ iterate a computed number of times }}
count   ID                             {{ new FIXed ID for interation number }}
using   ID = {exp}                     {{ bind new ID to existing data object }}
foreach ID in {exp}                    {{ iterate over data bbject }}
for     ID = {exp} by {exp} to {exp}   {{ "normal" FOR clause }}
```

Fig. 10 — Loop iteration clauses

The FOREACH Clause

Iteration over the members of a data object without knowledge of its internal structure is important in type abstraction languages [5,14,17]. The FOREACH clause and iteration function perform this iteration in FLEX. The FOREACH clause binds a new (currently undefined) identifier to a submember of a data object at the beginning of each iteration, where the particular submember is determined by a call to a generic Iteration Function. The Iteration Function "INORDER" (Fig. 11) is used to select the members of a "LIST" of integers in the order in which they exist in the list, and can be referenced as in the Procedure "PROCX".

```
iter *func INORDER (LISTX)
   form LISTX LIST (int)              {{ list of integers }}
   returns int

   do for I = 1 to LEN (LISTX)
      return (LISTX [I])              {{ return each time, suspended here }}
   od
cnuf                                  {{ at end, terminate loop }}
```

```
proc PROCX
   decl X LIST (int)

   do foreach MBR in INORDER (X)

     {{ "MBR" is successively bound to the elements of the
        list "X" as determined by the Iteration Function
        "INORDER" }}

   od
corp
```

Fig. 11 — FOREACH iteration

Iteration Functions are CLOSED ACCESS Functions that can only be activated by the FOREACH clause. After RETURNing the appropriate member for the first iteration, they are suspended (i.e., their environment is preserved) until the beginning of the next iteration when they are continued to select the submember for that iteration. When they reach their lexicographic end, the caller's iteration is terminated. Two constraints are placed on these Functions: 1) they must not interact with any other Function and are therefore always CLOSED, and 2) the object being iterated is not allowed to vary during the course of the iteration.

Several Iteration Functions can be defined for one data type. For example, "REVERSE", "SMALLESTFIRST", and "ANYORDER" could transverse a list in reverse order, ascending order of its values, or random order, respectively.

The loops in Fig. 12 perform the same action CLEARing each member of a list "LISTX". The FOREACH clause is the most powerful of these and is preferable wherever it can be used. The UNLESS clause suggested by Grampp [3] often reduces the need for an IF-FI structure enclosing the loop body.

```
do for each MRL in ANYORDER (LISTX)
   unless { MBR is already CLEARed }

         CLEAR (MBR)
od

do for Y = 1 to LEN (LISTX)
   using MBR = LISTX [Y]

         CLEAR (MBR)
od

do times LEN (LISTX)
   count Y                          {{ Y = 1,2, . . . }}

         CLEAR (LISTX [Y])
od
```

Fig. 12 — Three equivalent iteration loops

Rules

There are rules regarding the interactions of the various loop clauses that are "run time" restrictions and cannot be checked by the FLEX Processor. For example, the USING clause (Fig. 10) can evaluate its expression and bind its identifier at the beginning of each iteration, or before only the first iteration. User groups may formulate such rules themselves, although the Language Manual [1] provides a set that may serve in the absence of such agreement.

Expressions

General Expressions

The language contains generalized expressions that are simply nested Function invocations, where definable operators are available for user convenience. Operators need not be used by those who enjoy LISP-like Function nesting syntax.

Infix operators have an assignable priority, are left associative, and are unspecified as to the order of evaluation of the two actual arguments. Prefix operators are of a higher priority than all infix operators.

Record Selection

The following is a reference to a variable "X", where "{e1}, {e2}, . . ." are general expressions:

X.SELX [{e1}].SELY [{e2},{e3},{e4}] [] . SELZ.

Subjects of "X" are being selected progressing from left to right across the reference. If the subobject at any given point is a record, then the "dot" convention (e.g., "SELX") can be applied to further select a component of the Record.

Selector Selection

Selectors allow the user to define Functions to select members of data objects created from defined data types. The selector convention satisfies Morris's second criteria [18] for the implementation of data type abstraction (i.e., language features be extendable to defined data), and is similar to a feature found in the programming language EL1 [21]. After the selection "X.SELX" has been made and the Selector "[{e1}]" is encountered, the processor issues a call to the generic Function named "SELECT1" ("1" refers to the number of expressions within the brackets). The first argument in this call is the object, and the remainder are the expressions within the brackets, which can be illustrated by:

SELECT (X.SELX, {e1})

The user must have defined the "SELECTn" (n = 0,1,2, . . .) Functions, and they will often be Access Functions. They may implement any algorithm for determining the returned subobject, and may have selector expressions of any type. The object returned by this Function can be further selected.

As an example, the Access Function "SELECT0" could be defined to return the top element of a the familiar "STACK" type, so that the reference "XSTACK [ ]" would return that element. A three-dimensional array type could be defined whose members are selected by the user-defined "SELECT3" Access Function using the familiar notation "XARRAY [{e1}, {e2}, {e3}]." The Function "SELECT2" could be defined to return a contiguous subset of a "LIST", as determined by two integer expressions.

Rules

"SELECT1" and "SELECT2" are provided for SEQuence types and return a particular member and a subsequence, respectively.

A Routine that can be invoked by an operator must be invoked by that operator. Hence "ASSIGN (alt X, Y)" cannot be used in place of "X : = Y" (Fig. 9).

SUMMARY

The caretaker must make several decisions when configuring the language for a particular software design environment, including:

- Specific loop structures and their rules of usage.

- Pure syntactic changes (new statements, reserved words, etc.).

- Restrictions on the manner in which Routines interact with their Data Environment (FIX/ALT/CLOSED).

- Limitations on the modes of Data Segment access through the USE and INCLude declaration.

Within this framework, the administrator must provide fundamental design tools for the designer, including:

- Standard operations and operators, such as equivalence, assignment, and arithmetic and Boolean operations.

- fundamental data types (lists, stacks, arrays, sets, etc.) and their manipulating Routines, many of which may be generic ("TOP", "CLEAR", etc.).

- Selectors for various data types so that familiar addressing modes can be used (e.g., two-dimensional arrays addressed as "XARRAY [33, 44].

- Iteration Functions for defined types, many of which will be Generic (e.g., "INORDER", "REVERSE", "ANYORDER", etc.).

- Whether these features are to be made global or must be specifically requested.

The designer uses these tools in building his applications system using some design methodology. Escapes are available for topdown design, and the Modules provide a means for subdividing and organizing the system. The processor will maintain the consistency of the design as the lower levels of the outer syntax appear, and its statistics can be used to keep management informed of the progress of the design.

FUTURE DEVELOPMENT

FLEX is an open-ended system whose development will continue beyond the capabilities described above. The long range goal is to develop a design tool that is an integral part of the software development process, providing diverse and timely feedback to aid the

software development process, providing diverse and timely feedback to aid the software creation process. There are several projects in various stages of evaluation or planning. In order of their immediacy, they are:

- Flex Utility: A utility program to off-load the statistical functions of the main Processor, and provide an extensive "pretty print" output. Complexity and other software metrics will be included in this system.

- Type Space Execution: This system will generate "code" to be subsequently "executed" by a small simulation processor. This will allow complete interface type checking in the presence of unbound formal parameters, and the detection of aliasing between the Parameteric and External Routine Data Environments.

- Prototype Design System: This system will execute programs written in a subset of the FLEX language. This may take the form of a translator from FLEX to some other programming language (e.g., Pascal or SIMPL) or produce code for a simulation machine.

BIBLIOGRAPHY

1. S.A. Sutton, "The FLEX System: User and Caretaker's Manual," Technical Report TR-765, Department of Computer Science, University of Maryland, 1979.
2. R.C. Linger, H.D. Mills, and B. Witt, *Structured Programming Theory and Practice*, Addison-Wesley, 1977.
3. P.T. Grampp, correspondence to ACM Sigplan Notices 13 (No. 7), 6-7 (July 1978).
4. W.A. Wulf, R.L. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs," IEEE Trans. Software Eng. SE-2 (No. 4), 253-265 (Dec. 1976).
5. B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," Comm. ACM 20, No. 8 564-576 (Aug. 1977).
6. B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, and G.J. Popek, "Report on the Programming Language EUCLID," SIGPLAN Notices 12, 2 1-79 (Feb 1977).
7. K. Jenson, and N. Wirth, *PASCAL User Manual and Report*, Springer-Verlag, 1974.
8. V.R. Basili, and A.J. Turner, *SIMPL-T: A Structured Programming System*, Paladin House Publishers, 1976.
9. G.J. Myers, *Composite/Structured Design*, Van Nostrand Reinhold Company 1978.
10. M. Halstead, *The Elements of Software Science*, Elsevier Publ. 1977.
11. S.H. Caine, and E.K. Gordon, "PDL-A Tool for Software Design," AFIPS Conf. Proceedings 4, 271-276 (1975).
12. T.J. McCabe, "A Complexity Measure," IEEE Trans. Soft. Engr. SE-2, 4 308-320 (Dec. 1976).

13. D.A. Fisher, "Department of Defense Requirements for High Order Computer Programming Languages (Revised 'IRONMAN,' July 1977)," SIGPLAN Notices 12, 12 39-54 (Dec. 1977).

14. R.R. Atkinson, B.H. Liskov, and R.W. Scheiffer, "Aspects of Implimenting CLU," Proceedings ACM Annual Conf., Washington D.C. 123-129 (Dec 1978).

15. B. Nordstrom, "Assignments and High Level Data Types," Proceedings ACM Annual Conf., Washington, D.C. 630-640 (Dec 1978).

16. B.H. Liskov, S.N. Zilles, "Specification Techniques for Data Abstraction," IEEE Trans. on Soft. Engr. SE-1, 7-19 1 (March 1975).

17. M.S. Shaw, W.A. Wulf, and R.L. London, "Abstraction and Verification in ALPHARD: Defining and Specifying Interation and Generators," Comm. ACM 20, 8 553-564 (Aug. 1977).

18. J.B. Morris, "A Synopsis of Data Type Abstraction in Programming Languages," Los Alamos Scientific Laboratory Report LA-UR—76-1750 1976.

19. M. Jackson, Principles of Program Design, Academic Press, London 1975.

20. B.P. Buckles, "Formal Module Level Specifications," Proceedings ACM Annual Conf., Seattle 130-144 (Oct. 1977).

21. B. Wegbreit, "The Treatment of Data Types in EL1," Comm. ACM 17, 5 251-264 (May 1974).