CS 120

MUTANT 0.5

AN EXPERIMENTAL PROGRAMMING-LANGUAGE

BY

E. SATTERTHWAITE

TECHNICAL REPORT NO. CS 120
FEBRUARY 17, 1969

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

MUTANT 0.5, an Experimental Programming Language

E. Satterthwaite
Computer Science Department
Stanford University
February 1969

The research reported here was supported in part, by a National Science Foundation Graduate Fellowship and in part by the Atomic Energy Commission. Preparation of this report was supported by NSF Grant GP-7615.

Abstract

A programming language which continues the extension and simplification of ALGOL 60 in the direction suggested by EULER is defined and described. Techniques used in an experimental implementation of that language, called MUTANT 0.5, are briefly summarized. The final section of this report is an attempt to assess the potential value of the approach to procedural programming language design exemplified by MUTANT 0.5. Implementation and use of the experimental system have indicated a sufficient number of conceptual and practical problems to suggest that the general approach is of limited value; however, a number of specific features were found to be convenient, useful,, and adaptable to other philosophies of language design.

A. Introduction

In his thesis, McKeeman [McKee 66] describes MUTANT, a "kernel" language which he proposes as a nucleus for the design of procedural programming languages. Many features of that language appeared useful for expressing algorithms of both graph theory and graphical data processing. In addition, MUTANT suggests possible approaches to the problems of data structuring and the specification of parallel processing. To gain some experience with such facilities, an experimental language with semantics similar to those of MUTANT was designed. Certain semantic concepts were generalized, and others restricted; also the syntax was substantially altered. In addition, the current version of the language does not include all facilities of MUTANT (most notably, unordered sets and real number arithmetic); it is therefore called MUTANT 0.5.

A processing system for MUTANT 0.5, consisting of a compiler and an interpreter, was implemented on the IBM System/360, and some experience with that system has been obtained. The language has been found to allow a concise and natural expression of many algorithms, but a number of difficulties were experienced in designing both the language and the interpreter for the system. Although certain language improvements are obviously needed and efficiency could be significantly increased by use of more suitable hardware, I agree with Wirth's conclusion [Wirth 67a] that difficult logical problems underlie both the design and implementation of such a language. No further development or use of the current system is planned. Thus in terms of providing a useful language and processor, the project was a failure. It was, however, a valuable exercise in language design; this report is an attempt to analyze, for the benefit of future work, some of the strengths and weaknesses of the MUTANT 0.5 design revealed by both the implementation and the use of the language.

A.1 Organization of the Report

In section B, MUTANT 0.5 is defined in the style of the ALGOL 60 report [Naur 63]. Syntax is described by productions in Backus-Naur form (BNF); semantics, by English prose. Some examples of programs in MUTANT 0.5 are then presented and explained. Section C is a brief summary of the techniques used in implementing the experimental compiler and interpreter. Use of these programs is described in section D. Finally, section E is an attempt to charac-

terize MUTANT-like programming languages, to identify some inherent problems in their definition and implementation, and to assess their practical utility. That section is a minor revision of a draft written in September 1967; some of the positions stated there have since been substantially extended or reformulated as a result of more recent reading, discussion, and research.

Appendices III, IV, and V, although referenced in the text, are not reproduced in this report. They are compilation listings of the various programs used in the experimental implementation of MUTANT 0.5.

A.2 Comments on Notation

In this report, two different character and terminal symbol representations are used in the description of the syntax of MUTANT 0.5 as well as for the representation of programs written in the language. One may be considered the publication representation; the other, a hardware representation reflecting the available character set. The former is introduced in the belief that it is somewhat more agreeable and readable. Appendix I establishes the correspondence between these two representations. In addition, a slight variant of BNF has been adopted for compatability with the output of certain processing programs: alternate right parts of a production are placed on consecutive lines without repetition of the corresponding left part. In the remainder of this report, publication and hardware representations will be freely cross-referenced, usually without explicit comment.

A.3 Acknowledgements

The work reported below is based on a CS239 project directed by Professor W. F. Miller during the fall and spring quarters of academic 19664967. It includes additional modifications suggested by CS360 research done during the summer quarter of 1967. Support was provided in part by a National Science Foundation graduate fellowship and in part by the Atomic Energy Commission. Preparation of this report was supported by NSF Grant GP-7615.

The definition and implementation of MUTANT 0.5 have drawn heavily from ideas presented informally by various faculty members and fellow graduate students at Stanford; discussions with Professors W. F. Miller and W. M. McKeeman and with Mr. W. Hansen were especially helpful. In addition, all aspects of the project owe much to the teaching and research of Professor N. Wirth, and his syntax processing programs and PL360 system were essential tools in the language design and implementation work.

B. The MUTANT 0.5 Language

The syntax and semantics of MUTANT 0.5 are defined below. In general, McKeeman's statement of the principles of language design [McKee 66, pp. 71-73] has been accepted as valid. Conciseness of notation has been carried somewhat further by adapting the notation of set theory whenever possible. The concepts and notation of the language were primarily inspired by McKeeman's MUTANT [McKee 66]; they also draw directly and indirectly from ideas found in ALGOL 60 [Naur 63], EULER [Wirth 65a], APL [Iver 62], and PL/I [IBM 66].

B.1 The MUTANT 0.5 Grammar

The grammar of MUTANT 0.5 is listed, in the hardware character set, in Appendix II. In addition, -relevant productions of the grammar will be included at appropriate, points in the following discussion of the semantics of MUTANT 0.5. Such productions are related to Appendix II by the use of marginal production numbers. The somewhat artificial appearance of some productions reflects two constraints placed on the grammar, namely

(1) the production set must lead to a simple precedence grammar [Wirth 65a];

(2) the productions must be chosen to simplify the translation of the language during the process ofsyntactic analysis.

For the reasons given by McKeeman [McKee 66, p. 93], it is convenient to define the syntax of identifiers, strings, and integers external to the formal grammar of MUTANT 0.5; informally, it may be described by the following set of productions.

```
(identifier)  ::= (letter)
                  (identifier) (letter)
                  (identifier) (digit)
(integer)     ::= (digit)
                  (integer) (digit)
(string)      ::= (string head) "
(string head) ::= "
                  (string head) (non-quote character)
                  (string head) " "
```

B.2 Semantic Description of MUTANT 0.5

MUTANT 0.5 programs describe the creation and manipulation of values. In the language, values of three unstructured types (integer, process, and name) and two structured types (string and list) are available to the programmer. Values of type integer have the properties of mathematical integral values. process values are designations of computational processes (procedures); name values designate special computational processes which, upon activation, compute the name of a storage cell. string values are sequences of character values, which correspond to elements of a fixed set of symbols. In MUTANT 0.5, list values are ordered sequences of structured or unstructured values of arbitrary length, in which all elements are not required to be unique in value.

Comment

The above types were chosen as a minimal set adequate for experimentation. In any serious programming language, real number arithmetic would be essential. McKeeman's type set also appears to be a valuable addition to programming languages, omitted only because of limited time and goals.

Since MUTANT 0.5 is a highly involuted language, description of many constructs requires the use of terms before they are defined. The reader unfamiliar with MUTANT or a similar language is advised to consider some of the simpler examples of section B. 3 before continuing.

B.2.a Constants

Syntax

```
(constant) . ::= (integer)                    (64)
               | (string)                     (65)
               | ⟨begin⟩ }                    (66)
(begin)      ::= {                            (70)
               | (declare) |                  (71)
```

Semantics

A non-negative integer is denoted by a sequence of decimal digits, and the value of that integer is the value of the digit sequence interpreted as a decimal number. Negative integers are syntactically recognized as primaries.

A string is a sequence of characters and is denoted by a sequence,

delimited by string quotation marks ("), of the graphic symbols corresponding to the character values. In the denotation of a string, two contiguous string quotation marks signify a single string quotation character.

The construct " { } " (or " (declare) \| } ") denotes the null list, i.e., a value of type list with no elements.

A constant alwayshas a value.

Examples

0 3 100 32767

"This is a string." " " "Hamlet " " "

{ }

B.2.b Declarations

syntax

(declare) ::= { $ (72)

(declare) (identifier) (73)

(declare) (identifier) ((v-expression)) (74)

Semantics

MUTANT 0.5 provides values of several types as well as storage cells into which such values may be placed. Declarations serve to create cells and also to provide names for either cells (and their contents) or for values. At most one declaration appears at the head of a list (B.2.c), and the scope of the identifiers in that declaration is exactly the corresponding list.

If the identifier is immediately followed by an expression in parentheses, that identifier is considered to name the value of that expression. All such expressions are computed sequentially before computation of the values of any of the list elements, and these expressions are evaluated as if they were written in an immediately containing list. No explicit assignment to an identifier naming a value is permitted.

If the identifier is not so followed, it names a cell. Values of any type may be assigned to any cell, and such assignment dynamically determines the cell structure. Thus the structure of a cell may be undefined, or atomic, if the cell contains a value of unstructured type, or structured. If, in the last case, the cell contains a value of type list, that cell has a composite structure consisting of a sequence of (atomic or structured) subcells, one for each list element. Similarly, if the cell contains a value of type string, that cell is structurally a sequence of atomic character cells.

An identifier names the cell or value associated with it by a declaration.

Every non-reserved identifier not contained in a declaration either must designate a controlled value (B.2.1) or procedure formal parameter value (B.2.m) or must occur within the scope of an identifier of the same name. If an identifier is associated with more than one scope, a use of that identifier designates the cell or value associated with it in the smallest possible containing scope. Subcells or subvalues are designated by a uniform indexing scheme (B.2.f).

Examples

```
{ $ a b newidentifier
{ $ x y twototheI5th (32768)
{ $ sum (a+b) difference (a-b)-
```

Comment

Named values may alternatively be viewed as the contents of cells which may be initialized upon scope entry but are "read-only" within the scope of the naming identifier. The rules of scope and evaluation of the initializing expression do not admit initialization to recursive procedure values; a facility similar to Landin's rec [Land 64, 66] is absent.

B.2.c Lists

Syntax

```
(list)          ::= (list head) }                          (67)
(list head)     ::= (begin) (g-expression)                 (68)
                    (list head) , (g-expression)           (69)
(begin)         ::= {                                      (70)
                    (declare) |                            (71)
```

Semantics

A list is an ordered sequence of general-expressions. In the execution of a MUTANT 0.5 program, the expressions within the list are computed successively from left to right. A general-expression may conditionally fail to designate any value. Lists have structured values of type list; the value of a list is the (possibly empty) sequence of values of those contained general-expressions yielding values. A declaration does not have a value or constitute a list element. In general, the number of elements in the list value cannot be determined a priori. A list always has a value.

Examples

```
{ 1, 2, 3, "abc" }        { 1, { 2, { 3 } } }
```

```
{ 1 → x, y-2 → y, 10⊗a }
{ $ a b | x → a, y → b, a+b, a-b }
{ $ a | { $ a | 2 → a }, 2 → a }
{ a+b, a-b, a⊗b, b≠0 => a÷b }
```

B.2.d Simple Primaries

syntax

```
⟨s-primary⟩      ::= (s-primary *⟩                          (53)
(s-primary *⟩ ::= (constant)                                (54)
                  get                                       (55)
                  (list)                                    (56)
                  (primary *⟩ (list)                        (57)
                  ( (v-expression) )                        (58)
                  (case head) (v-expression) )              (59)
                  (for head) }                              (60)
                  (while head)' }                           (61)
                  (for/while head) }                        (62)
                  (s-primary *⟩ [ (v-expression) ]          (63)
```

Semantics

Productions 57 (B.2.m), 59 (B.2.e), and 60-62 (B.2.1) are listed for completeness but are not discussed in this section.

Every simple primary has a value, which may be of either structured or unstructured type.

The value of the primary get is of type string and consists of the next string (according to the MUTANT 0.5 conventions) found in the interpreting mechanism's sequential input stream when the primary is evaluated, and such evaluation causes that string to be deleted from the input stream.

Parentheses serve to control the association of operands, and hence the application of operators, in the conventional way.

Square brackets are used to designate the subscripting of simple primaries. In the application of the subscript operator, the simple primary and then the subscripting value-expression are evaluated, and the value of the result is determined by application of the following algorithm:

(1) If the value of the simple primary is of unstructured type, then the result is not defined.

(2) If the value of the simple primary is of type list, then

(a) if the value of the subscripting expression is of type integer

and that integer is positive and not greater than the number of elements in the list, then the resulting value is the list element with that integer as index, where element indices begin with one and increment by one;

(b) if the value of the subscripting **expression** is of type list, then the resulting value is the list of values obtained by **successively** applying each element of the subscripting list to the simple primary;

(c) otherwise, the result is not defined.

(3) If the value of the simple primary is of type string, then

(a) if the value of the subscripting expression is of type integer and that integer is positive and not greater than the number of characters in the string, then that integer is used as an index to select a string character, and the resulting value is an encoding of that character of type integer;

(b) if the value of the subscripting expression is a list of integers, all satisfying the bounds conditions of (a), then the value of the result is a string consisting of the sequence of characters obtained by successively using each integer in the list as a subscript;

(c) otherwise, the result is not defined.

Examples

```
3     "abc"     get     { 1, { 2, 3 }, 17 }     (a+b)
```

In the following examples, all those on the same line have identical values.

```
{ 1, 2 }[1]                    1                    { 2,1 }[2]
{ 1, { 2, 3 } }[2]             { 2, 3 }             { 1, 2, 3 }[{ 2, 3 }]
{ 1, 2, 3 }[{ 1, { 2, 3 } }]                  { 1, { 2, 3 } }
"ABC"[1]                       193                  "(A)"[{ 2 }][1]
"abc"[2  3]                    "bc"                 {  {"bc"}  }[1][1]
```

Comment

In MUTANT 0.5, lists are considered linear sequences. The operation of subscripting of lists has been extended from selection to the construction of general sublists. Subscripting has similarly been extended to provide a substring operation.

B.2.e Case Expressions

syntax

(s-primary) ::= (case head) (v-expression)) (59)

(case head) ::= (case index) ((v-expression) ; (75)

(case head) (v-expression) ; (76)

(case index) ::= [(v-expression)] (77)

Semantics

A case **expression** consists of a case index followed by a sequence of value-expressions. In the evaluation of a case expression, the value of the expression in the case index is determined. If that value is of type integer and is positive and not greater than the number of expressions in the sequence, then that integer is used as an index to select an expression for evaluation, and the value of that expression is the value of the case expression. Otherwise, the result is not defined. A case expression **always** has a value.

Examples

[n] (3; 2; 1)

[opcode] (0 → **acc**; **acc+a** → **acc**; ace-a → **acc**; a → pc)

[(x=0) + 1] ("x is non-zero"; "x is zero")

Comment

Case expressions are generalizations of **ALGOL 60's** conditional **expressions** and **conditional** statements, which have not been specially distinguished in MUTANT 0.5.

B.2.f Cell Designators

syntax

(cell id) ::= (cell id *⟩ (46)

(cell id *⟩ ::= (identifier) (47)

(cell id *⟩ [(v-expression)] (48)

(cell id *⟩. (49)

Semantics

MUTANT 0.5 provides named cells in which values may be stored and also named **values** which are not associated with storage cells. A cell designator is used in the formation of either a primary **(B.2.h)**, in which case it designates a value, or an assignment **(B.2.k)**, in which case it designates a cell or subcell. As explained in section **B.2.b**, the structure of a cell is determined

dynamically by the structure of its contained value; thus the interpretation of a (sub)ceU designator is dependent upon the concurrent contents of the cell.

If the cell designator is used in the formation of an assignment, then the named **(sub)cell** is determined as follows:

(1) If the designator is not subscripted or dotted, then by the rules governing the scope of identifiers **(B.2.b)**, that identifier must name a declared cell and it designates that cell; otherwise, the result is not defined.

(2) If the cell designator terminates with a subscript, then the contents of the cell named by that simpler cell designator obtained by deleting the rightmost subscript is determined, and the subscripting expression is evaluated. If the cell contents is not a value of type list or if the subscripting expression is not of type integer, the **result** is not defined. If the value of that integer is positive and not greater than the number of elements in the list, the designated cell is that subcell containing the list element selected by use of that integer as an index.

(3) If the cell designator terminates with a dot, then the value named by the cell designator obtained by deleting the rightmost dot is determined according to the algorithm of the next paragraph. If that value is of type name, the **computation** designated by that value is activated to determine the designated cell **(B.2.g)**. **Otherwise,** the result is not defined.

If the cell designator is used in the formation of a primary, then the named (sub)value is determined as follows:

(1) If the cell designator is not subscripted or dotted, then

(a) if, by the rules governing the scope of identifiers, the identifier names a value **(B.2.b)**, a controlled value **(B.2.l)**, or a procedure formal parameter value **(B.2.m)**, then the value of the cell designator is the named value;

(b) if, by the rules of scope, the identifier names a cell, then the value of the cell designator is the concurrent value of the contents of the named cell.

(2) If the cell designator terminates with a subscript, then the value named by the cell designator obtained by deleting the rightmost

subscript is determined. The subscripting expression is evaluated and used as an index operating on the previously obtained value to produce a resulting new value as described in section B.2.d.

(3) If the cell designator terminates with a dot, then the value named by the cell designator obtained by deleting the rightmost dot is determined. If that value is of type name, the computation designated by that value is activated to determine the designated cell (B.2.g), and the resulting value is the value contained in that cell. Otherwise, the result is not defined.

Thus in the cases in which a cell designator is valid in the formation of either a primary or an assignment, the value in the first case corresponds to the contents of the cell named in the second case, but more general indexing is allowed in the formation of primaries.

Examples

a b[1] a[2][x[3]]

pointer. x[i]• x[2]•[1]

B.2.g References

syntax

(reference) ::= (reference *⟩ (50)

(reference *⟩ ::= ℓ (identifier) (51)

(reference *⟩ [(v-expression)] (52)

Semantics

Values of type name, which are computational processes for determining cell names, are designated by references. Thus every reference has a value of type name. Upon activation of such a process (B.2.f), the resulting cell name is determined by analysis, as described in section B.2.f, of the cell designator obtained by deleting the "ℓ" in the formulation of the reference.

Examples

ℓa ℓb ℓa[x+y][1]

Comment

References provide an explicit method of processing cell names as values. They are intended primarily for use as "pointers" to complex data structures and as procedure actual parameters. References are defined as computational processes rather than actual cell names for technical reasons; EULER [Wirth 65a] demonstrates an alternative

approach using cell names. Unlike MUTANT, MUTANT 0.5 requires the **programmer** to distinguish between cell name and value.

B.2.h primaries

syntax

⟨primary⟩	::=	(primary *⟩	(33)
		(prefix) (primary)	(34)
		(infix) / (primary)	(35)
⟨primary *⟩	::=	(cell)	(43)
		(reference)	(44)
		⟨s-primary⟩	(45)
(prefix)	::=	#	(36)
		type	(42)
		abs	(39)
		neg	(40)
		¬	(37)
		list	(38)
		put	(41)

Semantics

A **primary** always has a value. The value of a primary without prefix or infix operators is the value of the corresponding cell designator, reference, or simple primary.

A prefix operator designates a partial function of one argument; the value of the corresponding primary is obtained by evaluation of the operand followed by the application of that function. If the following rules do not specify the resulting value, then the result is not defined.

(1) If the operator is "#" and the operand is of type list or string, the result is of type integer and is the number of top level elements in the list or string.

(2) If the operator is "type", the result is of type integer and is an encoding of the type of the operand.

(3) Otherwise the operator is applied recursively according to the following algorithm:

(a) If the type of the operand is not list, then the result is defined if the operator and **operand type** correspond to one of the following table entries:

Operator	Operand Type	Result Type
abs	integer	integer
neg	integer	integer
ι	integer	integer
list	integer	list
put	string	string

The first three are numeric operators which may be defined by the following λ-expressions [Land 64]:

abs = λx. if x ≥ 0 then x else -x

neg = λx. -x

ι = λx. if x = 0 then 1 else 0 .

The operator "list" produces a list of elements with unspecified value; the number of elements is the maximum of the operand value and 0. The operator "put" is an identity function with the side effect of writing a carriage return followed by the operand onto the sequential system output stream.

(b) If the type of the operand is list, then the result is a value of type list, the elements of which are the values obtained by applying the operator to each element (sequentially in the case of "put") of the list, if all such elements are defined.

(c) Otherwise, the result is not defined.

An infix operator designates a partial function of two arguments as described in section **B.2.i.** In the application of an infix operator to a primary, the primary is first evaluated. If the value obtained is not of type list, the result is not defined. Otherwise, the value of the primary is obtained as follows:

(1) An initial value is chosen according to the operator from the following table:

Operator	Initial Value
+, -, ∨	0
⊗, ÷, mod, ∧	1
=, ≠, <, ≤, >, ≥	not defined
base	not defined
|	""or{} *

* The empty string is chosen unless the first list element is a **list.**

(2) Beginning with that initial value as an intermediate result, the operator is applied to the intermediate result and a list element to produce a new intermediate result.

(3) The resulting value is the value of the intermediate result after each list element, chosen in sequence, has been used as indicated in (2).

Thus the value of the primary so obtained is an accumulation, relative to the operator, over the entire list.

Examples

```
a      { a, x }      { $ Y I x+z → y}      z[2]
abs x      "TITLE"      1st { 1, {2,2 } }
```

In the following examples, all those on the same line have identical values.

# { 1, 2, 3 }	3
# { 1, { 2, 3 } }	2
¬ { 1, 0, 1 }	{ 0, 1, 0 }
neg { 1, { 2, 3 } }	{ neg 1, { neg 2, neg 3 } }
type 0 = type "0"	0 .
+ / { 1, 2, 3 }	6
| / { "a", "b", "c" }	"abc"
| / { { 1, 2 }, { 3, 4 } }	{ 1, 2, 3, 4 }

Comment

The extension of the definition of prefix operators in MUTANT 0.5 is a slight generalization of Iverson's extension of such operators to vectors and matrices. Accumulation over a list with respect to an infix operator corresponds to Iverson's reduction [Iver 62].

B.2.i Simple Expressions .

syntax

```
(s-expression)   ::= (s-expression *⟩                              (15)
⟨s-expression *⟩::= (primary)                                      (16)
                    (s-expression *⟩ (infix) (primary)             (17)
(infix)          ::= |                                             (32)
                     +                                             (18)
                     -                                             (19)
                     ⊗                                             (20)
                     ÷                                             (21)
```

mod	(22)
=	(24)
≠	(25)
>	(26)
≥	(27)
<	(28)
6	(29)
A	(30)
V	(31)
base	(32)

Semantics

Every simple expression has a value. The value of a simple expression without an infix operator is the value of the corresponding primary.

An infix operator designates a partial function of two arguments; the value of the corresponding simple expression is obtained by evaluating the left operand, then evaluating the right operand, and then applying that function to the operand values. If the rules below do not specify the resulting value, then the result is not defined.

(1) If the operator is "|" and

(a) if both operand values are of type list, then the resulting value is of type list and is obtained by appending in sequence the list elements of the right operand to the list value of the left operand;

(b) if both operand values are of type string, then the resulting value is of type string and is obtained by concatenation of the two operand values, taken in order.

(2) Otherwise the operator is applied recursively according to the following algorithm.

(a) If the type of neither operand is list, then the result is defined if the operator and operand types correspond to one of the following table entries:

Operator	Type of Operands	Result Type
+, -, ⊗	integer	integer
+, mod	integer	integer
=, ≠, >, ≥, <, ≤	integer	integer
∧, ∨	integer	integer
base	integer	string

The operators "+", "-", and "⊗" designate the mathematical functions of integral addition, subtraction, and multiplication respectively. The operators "÷" and "mod" are not defined if the value of the right operand is 0. Otherwise, they are defined for integral operands by the following h-expressions, using real division and Iverson's floor and ceiling operators [Iver 62]:

÷ = λ(x,y). if x⊗y ≥ 0 then ⌊ x/y ⌋ else ⌈ x/y ⌉

mod = λ(x,y). x - y⊗(x÷y)

The relational operators have integral value 1 if the mathematical relation on the integers is satisfied and value 0 otherwise.

The operators "∧" and "∨" may be defined for integral operands by the following h-expressions:

∧ = λ(x,y). if x = 0 then 0 else if y = 0 then 0 else 1

∨ = λ(x,y). if x ≠ 0 then 1 else if y ≠ 0 then 1 else 0

The operator "base" produces a string, the characters of which are a sequence of digits constituting a representation of the left operand to the base specified by the right operand.

(b) If the type of exactly one operand is list, the result is a value of type list, the elements of which are the values obtained by sequential application of the operator to the non-list operand and each element of the list operand (taken in the original order), if all such elements are defined.-

(c) If the type of both operands is list, the result is a value of type list. The number of elements in that list is the minimum of the numbers of elements in the operand lists; the resulting list elements are obtained by sequential application of the operator to corresponding elements of the operand lists (taken in the original order), if all such elements are defined.

All infix operators have equal precedence, and the association of operands is from the left.

Examples

3 a + b c ⊗ abs x a + b - c a ∧ (b ∨ c)

In the following examples, all those on the same line have identical values:

3 + 4 - 1 3 ⊗ 2 13 ÷ 2 13 mod 7 6

3 > 4 1 - 2 + 1 (3 = 3) - 1 0

"abc" | "def" "abcdef"

```
{ 1, 2 }|{ "2", { 2} }          { 1, 2, "2", { 2 } }
2 ⊗ { 0, 1, 2 }      { 0, 2, 4 }      { 4, 6, 8 } - 4
{ 1, 2, 3 } ⊗ { 3, 2, 1 }             { 3, 4, 3 }
15{base 2{2, {8}, 4 }16 } + { 0, 1, 2, 3   }  { { 1, 2 }, { 4, 5 }, 7 }
                10,    }               { "1111", "17", "15", "F" }
```

Comment

The extension of infix operators follows Iverson [Iver 62]. It corresponds to both scalar and vector operation in ordinary vector arithmetic.

B.2.j Segments

syntax

```
(v-expression *) ::= (segment)                                          (6)
(segment)        ::= (s-expression)                                     (12)
                     (s-expression) _ (s-expression)                    (13)
                     (s-expression) _ (s-expression) _ ⟨s-expression⟩   (14)
```

Semantics

A segment always has a value. The value of a segment without the " " operator is the value of the corresponding simple expression.

'Otherwise, the value of the first (leftmost) simple expression is called the initial value; of the second, the limit value; and of the third (if present), the step value. The value of the segment is obtained by the following process:

(1) All of the simple expressions are evaluated in the order of appearance; if the step value is not explicitly provided, it is taken to be the integer 1.

(2) If the values so obtained are all of type integer, then the result is a value of type list. Otherwise, the result is not defined.

   (a) If the step value is positive, the elements of the list are all those integers, ordered in algebraically increasing value,' which are obtained by adding non-negative integral multiples of the step value to the initial value and which are not greater than the limit value.

   (b) If the step value is negative, the elements of the list are all those integers, ordered in algebraically decreasing value, which are obtained by adding non-negative integral multiples of the step value and which are not less than the limit value.

(c) Otherwise, the result is not defined.

Examples

```
1  n        2 '_ #S-1      { 1, 2, 3 }       0
```

In the following examples, all those on the same line have identical values.

```
1 _ 3                 1 _ 3 _ 1             { 1, 2, 3 }
2  10  4              2 _ 13 _ 4            { 2, 6, 10 }
3 _ 1 _ neg 1                               { 3, 2, 1 }
1  0                  2 _4 _ neg            { }
```

B.2.k Assignments

Syntax

⟨ v-expression *⟩ ::= (v-expression *⟩ → (cell) (8)

Semantics

Assignments serve to assign values to cells. In the evaluation of an assignment, the value of the expression to the left of the "→" is obtained, and then the (sub)cell named by the cell designator is determined as described in section B.2.f. If both these processes produce results which are defined, the computed value is assigned to the designated (sub)cell. Such assignment may dynamically change the structure of the cell.

Every assignment has a value, which is the value of the expression to the left of the arrow.

Examples

```
0 → a                      [index]( a+b; a-b ) → p[2][1]
d•[ 1_n 1 → c•[3]          0 → x → y → z
```

B.2.1 Iterative Statements

Syntax

```
(s-primary *⟩    ::= (for head) }                                  (60)
                     (while head) }                                (61)
                     (for/while head) }                            (62)
-(for head)      ::= (begin) (for set) (g-expression)              (78)
(for set)        ::= (for set *⟩ :                                 (81)
(for set *⟩      ::= (identifier) ϵ (v-expression)                 (82)
(while head)     ::= (begin) (while cdn) (g-expression)            (79)
(while cdn)      ::= (while '⟩ :                                   (85)
```

(while ') ::= (while) (q-clause) (84)

(while) ::= * (83)

(q-clause) ::= (v-expression) => (4)

(for/while head) ::= (begin) (for set) (while cdn) (g-expression) (80)

Semantics

Iterative expressions provide for 'controlled repetitive evaluation of a general **expression,** which in each of the various forms of the iterative **expression** is called the controlled expression. Such an expression may conditionally fail to have a value. The value of every iterative expression, if defined, is of type list; the elements of that list are, in order, the successive values obtained from those evaluations of the controlled expression producing values. A declaration may be included in the heading of any **iterative** expression; **the scope** of the identifiers in such a declaration is the iterative expression.

For-iterative **expressions** specify iteration over a list. The evaluation of such an expression proceeds as follows:

(1) Any declarations in the heading are processed as described in section B.2.b.

(2) The **value-expression** of the for-set is evaluated. If the resulting value is not of type list, the result is not defined.

(3) With the **identifier** of the for-set naming a value of an element of the list thus obtained, the controlled **expression** is evaluated. That identifier is said to designate a controlled value; it is implicitly declared by its **appearance** in the for-set, and its scope is the controlled expression. The list element values named are successively taken in order over the entire list value.

(4) The final value of the iterative **expression** is a list as specified in the preceding paragraph.

While-iterative expressions specify repeated evaluation of the controlled expression as long as a specified condition holds. The evaluation of such an expression proceeds as follows:

(1) Any declarations in the heading are processed as described in section B.2.b.

(2) The value-expression of the qualifying clause contained in the while-condition is evaluated. If the resulting value is not of type integer,

the result is not defined. If that value is 0, evaluation of the iterative expression is terminated, and its final value is a list as specified above. Otherwise, the controlled **expression** is evaluated, and the step is repeated.

For/while-iterative expressions specify iteration over a list as long as a specified condition holds. They are processed as described for for-iterative **expressions** with the following exceptions:

(1) The scope of the identifier designating the controlled value is extended to include the while-condition.

(2) Before each evaluation of the controlled **expression,** the value-expression of the qualifying clause contained in the while-condition is evaluated. If the resulting value is not of type integer, the result is not defined. If that value is 0, evaluation of the iterative expression is terminated, and its value is the list of values obtained to that point. Otherwise, processing continues with evaluation of the controlled expression.

Examples

```
{ i ∈ 1 n  : S + (i⊗i) → S }
{ * abs(x1 - x2) > delta => :  { x2 → x1, f{x1,x2} → x2 } }
{ x ∈ table : * looking => :  x[1] = arg => { 0 → looking, x[2] } }
```

AU the following examples have identical values.

```
{ x ∈ 2_10  :  (x mod 2) = 0 => x }
{ $ x | 0 → x, { * x<10 => :  x4-2 → x } }[2]
{ x ∈2 100 2  : * x≦10 => :  x }
{ 2, 4, 6, 8, 10 }
```

Comment

The for-iterative expression provides the effect of a generalized list mapping function. Since the controlled expression may conditionally fail to have a value, that function can include selection.

B.2.m Procedures

syntax

```
⟨v-expression *⟩ ::= (procedure head) (v-expression) '                (7)
(procedure head) ::= ⟨proc head +⟩ |                                 (11)
⟨proc head +⟩    ::= '                                               (9)
                     ⟨proc head +) (identifier)                      (10)
```

(s-primary) ::= (primary *⟩ (list) (56)

Semantics

A procedure definition is delimited by apostrophes (') and designates a value of type process.

The computational process designated by a value of type process is activated by the evaluation of a simple primary consisting of a primary followed by a list. If the value of such a primary is not of type process, the result is not defined. Otherwise, the expression in the definition of the procedure corresponding to the process value is evaluated, subject to the rules below, and the resulting value is also the value of the simple primary. The rules governing such evaluation are the following:

(1) The identifiers appearing in the procedure head are associated, in order, with the values of the elements of the argument list. If the number of identifiers exceeds the number of list elements, the values named by the extra identifiers are not defined.. If the number of list elements exceeds the number of identifiers, the extra list elements are disregarded. Such identifiers&e said to designate procedure formal parameter values.

(2) In the application of rules of scope in the evaluation of the expression, the applicable scopes are those at the place of procedure definition, not procedure activation*

Examples

The following examples define and assign process values:

' | a + 1 → a ' → increment1

' a | a• + 1 → a• ' → increment2

' x y | { x+y → sum, sum+ 2}[2]' → average

The following examples indicate the activation of the above process values.

increment11 }

increment21 ℓx }

average{ a+b-c, sum }

Comment

In MUTANT 0.5, a parameter list, which can be empty, must be associated with every procedure activation.

B.2.n Expressions with Value

Syntax

⟨v-expression⟩ ::= (v-expression *⟩ (5)

⟨v-expression *⟩ ::= (segment) (6)

(procedure head) (v-expression) ' (7)

(v-expression *⟩ → (cell) (8)

Semantics

A value-expression, if defined, always has a value. It is the least restricted type of **expression** with such a property provided in **MUTANT** 0.5.

B.2.o General **Expressions**

syntax

(g-expression) ::= (v-expression) (2)

(q-clause) (g-expression) (3)

(q-clause) ::= (v-expression) => (4)

Semantics

A general expression may conditionally fail to designate a value. If it does not contain a qualifying clause, **then it** has a value, and that value is identical to that of the corresponding value-expression. Otherwise, the value, if any, of the general expression is determined by first evaluating the expression in the leftmost qualifying clause. If that value is not of type integer, the result is not defined. If the value is 0, the general expression has no value. Otherwise, the value, if any, is that of the general expression obtained by deletion of the leftmost qualifying clause.

Examples

$y \neq 0$ => x÷y predicate{x} => **function{x}**

x < **max** => y > min => x÷y

Comment

In **MUTANT** 0.5, an unsatisfied qualifying clause gives rise to no value, not **an** undefined value. Thus general **expressions** can be used in contexts only in which such a property is meaningful, i.e., in the formation of list values.

B.2.p **Programs**

syntax

(program) ::= eof ⟨v-expression⟩ eof (1)

Semantics

A program is a **value-expression** delimited by end-of-file marks. The value of a program is that of the value-expression.

Examples

See section B.3.

Comment

In MUTANT 0.5, the end-of-file marks are assumed to be supplied by the interpreting mechanism and are not normally written.

B.3 Examples

Listings produced during the compilation and execution of some sample MUTANT 0.5 programs are included as Appendix III. In these examples, comments are delimited by question marks. Selected examples are repeated below, with commentary, in the (more readable) publication character set.

B.3.1 Factorial Calculation

Program

```
{ $ factorial |
   ' n | [2 - (n=0)] ( 1; n ⊗ factorial(n-1) ) ' → factorial,
   { n ∈ 1 6 :
      put ( (n base 10) | " factorial = " | (factorial(n) base 10) )
   }
}
```

Output

```
1 factorial = 1
2 factorial = 2
3 factorial = 6 ,
4 factorial = 24
5 factorial = 120
6  factorial  =  720
```

Comment

This example corresponds closely to McKeeman's Example 1 [McKee 66, p. 75]. The following is a similar ALGOL 60 program, which assumes a suitable write statement.

```
begin integer n;
   integer procedure factorial(n); value n; integer n;
      factorial := if n=0 then 1 else n ⊗ factorial(n-1);
   for n := 1 step 1 until 6 do
      write( n, " factorial = ", factorial(n) )
end
```

In the MUTANT 0.5 program, the process value (delimited by apostrophes) assigned to the cell "factorial" gives the usual recursive definition of the factorial function. The parameter "n" is used to compute a case index for selection of one of two expressions to be evaluated. Thus case expressions are generalizations of ALGOL 60's conditional expressions and statements. Note that in MUTANT 0.5, the expression "n=0" has integral value 1 if the value of n is zero and value 0 otherwise. The expression "1_6" is equivalent to the expression "{ 1, 2, 3, 4, 5, 6 }", and iteration over each element of that list is specified.

B.3.2 Extended Factorial Calculation

Program

```
{ $ factorial prod |
   {' n | [2-(n=0)]( 1; n ⊗ factorial[1]{n-1} ) ',
    ' n | ⊗ / (1_n) ',
    ' n | prod{ 1_n } '
   } → factorial,
   ' L |[2 - (#L=1)] ( L[1];
     prod{ L[1   #L+2] } ⊗ prod{  L[#L+2+1 _ #L] ) ' → prod,
   { i ∈ 1_3 :
      { put " ", put ("method "| (i base 10)),
         { n ∈ 1 8 :
            put ( (n base 10)|" factorial = "|(factorial[i]{n} base 10) )
         }
      }
   }
}
```

Output

```
method 1
1 factorial = 1
2 factorial = 2
   • • •
8 factorial = 40320

method 2
1 factorial = 1
   • • •
```

Comment

In this example, the value assigned to the cell "factorial" is a list of three process values giving possible definitions of the factorial function. The first is the recursive computation of the previous example. The second is an example of Iverson's reduction, in which the multiplication operator is used to reduce a vector (list) of the first n positive integers. The third process applies the auxiliary function "prod" to the same vector. "prod" designates a process intended to illustrate one possible hardware implementation of multiplicative reduction in which, recursively, the'vector is bisected and reduction applied to each part. Note the use of a list-valued subscript to select a sublist, which in turn is used as a procedure parameter.

B.3.3 Further Examples from MUTANT

Program

```
put( ( + / ({1,2,3} ⊗{3,2,1} ) ) base 10 )
```

Output

```
10
```

Program

```
{ $ perm |
   ' x | [2 - (#x=1)]
      ( {x}; .
       |/{ i ∈ 1 _ #x :
            { t ∈ perm{ x[1 _ i-1]|x[i+1 _ #x] } :  x[i_i] |t }
         ,
      ) ' → perm,
   { test ∈ { "a", "ab", "abc", "abcd" } :  put perm{test}
```

Output

```
3
a
ab
ba
abc
acb
bac
bca
cab
cba
 . . .
```

Comment

These programs for computation of inner product and permutations of string characters or list elements are MUTANT 0.5 versions of McKeeman's Examples 2 and 4 [McKee 66, pp. 77-78]; they are presented mainly for comparison.

B.3.4 A prime Sieve

Program

```
{ $ primesieve
   ( ' n | { $ L t | 2_n → L,
             { * #L≠0 => :
                { L[1] → t, { i ∈ L : imodt ≠ 0 => i } 3 L }[1]
             } }[2] '
   ) |
   {n∈{  25,  250}:
      { put ("primes in 2 to "|(n base 10)|":"),
        put (primesieve{n} base 10)
      }
   }
}
```

Output

```
primes in 2 to 25:
2
3
5
7
11
13
17
19
23

primes in 2 to 250:
2
  . . .
```

Comment

This program is an adaption of the sieve of **Eratosthenes** to the computation of all prime integers not exceeding a given integer. "primesieve" names a **process** value. In that process, a list of the integers from 2 to the given value is assigned to the cell named "L". While the length of that list is non-zero, the first element of the list is saved and then the list is replaced by a new list consisting of all the former list elements not multiples of the first element. Note that the saved value is selected as the value of the controlled expression in the while-iterative **expression** by the second subscript "[1]"; thus the value of the entire iterative **expression** is a list of the primes so saved. The subscript "[2]" selects that list as the value of, the procedure. Also note the use of the extended "put" and "base" operators.

B.3.5 Other Examples

Also included in Appendix III are programs illustrating the following:

(1) a slightly different permutation generator;

(2) a set of algorithms adapted from **Pohl's** graph package [Pohl 67] for computing the reachability matrix and maximal strongly connected **sub**-graphs of a graph from its connectivity matrix;

(3) an integer square root routine based on **Newton's** method.

C. Implementation Techniques

An experimental processing system for the language MUTANT 0.5 was developed. It consists of a compiler, which translates MUTANT 0.5 programs to a compact internal string code based upon Polish suffix notation, and an interpreter, which performs processing as directed by such strings. The processing system was implemented on the IBM System/360 hardware; it functions in the environment provided by the PL360 system [Wirth 67d]. In addition, an existing syntactic analysis program, written in Burroughs B5500 Extended ALGOL, was modified for use as an aid in developing the compiler.

C.1 The Syntax Processor (see Appendix IV for listing)

The syntax processor is an extension of a B5500 Extended ALGOL program originally developed-by Professor Niklaus Wirth at Stanford. Blocks B1 and B2 were taken from that program without significant modification. Block B1 establishes the precedence matrix as described by Wirth and Weber [Wirth 65a], using partial word operations for storage efficiency. Block B2 establishes the precedence functions using Wirth's algorithm [Wirth 65b].

Additional pre- and post-processing was added to produce punched tables in the PL360 syntax suitable for direct insertion into the compiler source deck. This processing includes:

(1) classification and sorting (according to the IBM EBCDIC collating sequence) of the terminal symbols of the syntax,

(2) assigning internal codes to the symbols,

(3) encoding and sorting the productions of the grammar,

(4) formatting the required tables.

A B5500 ALGOL program was chosen for modification because of the relatively powerful format capabilities provided.

Those cards at the beginning of the compiler (Appendix V) lacking "CMP" in the sequence field were produced by the syntax processing program. (Strictly, they are translations from such cards produced for a previous version of PL360, translated by a conversion program). The availability of this syntax processing program greatly facilitated modification of the syntax of MUTANT 0.5 as the system developed.

C.1.1 Symbol Recognition Tables

The following tables produced by the syntax processor are used by the

compiler procedure INSYMBOL in recognizing the terminal symbols of the language:

CCODES	a translation table which maps' characters occurring in the input stream into either their internal symbol codes or entries into other tables.
BREAK	a table of partitioning indices classifying characters, by their translation codes, as (1) single character terminal symbols, (2) characters possibly forming character pair terminal symbols, or (3) characters initiating identifiers, numbers, strings, or comments.
PAIRTAB	a sorted table of special character pairs forming terminal symbols.
RSVD	a table of entry indices into the reserved word table.
RSVWD	a table of reserved words, ordered by length and, within each length group, alphabetically.

C.1.2 Parsing Tables

The following tables produced by the syntax processor are used by the compiler% syntactic analysis routine in parsing input strings:

F, G	tables of precedence functions for the symbols of the vocabulary.
PLIM	a table of entry indices into the table RIGHTPART according to the leftmost symbol of the production right part.
RIGIDPART	a table of production right parts, exclusive of leftmost symbol, ordered by the (omitted) leftmost symbol.
LEFTPART	a table of corresponding production left parts.
RULE	a permutation vector giving the original interpretation rule number for each of the (reordered) productions.

C.2 The Compiler (see Appendix V for listing)

The compiler is a syntax directed, one-pass translator using the principles of semantic analysis controlled by a simple 'precedence syntactic analysis. The general organization of such translators described by Wirth [Wirth 65a, 67c, Shaw 66] has been adopted. In addition to the "value stack", information about previously scanned symbols is collected in an

identifier table and a separate (nested) table used in the processing of case expressions.

The compiler is written in PL360 [Wirth 67c]. Since analysis is table driven, the syntax processor was designed to produce tables which could be efficiently scanned (see section C.1). In particular, binary search is used for the table of special character pairs, while entries into the tables of reserved words and production right parts are controlled by key transformations on the identifier length and leftmost symbol of the right part respectively [Iver 62]. The table of declared identifiers is organized to reflect the block structure of the language [Shaw 66].

C.3 The Interpreter (see Appendix VI for listing)

The interpreter is a program simulating a machine for processing the Polish suffix string-code produced by the compiler from MUTANT 0.5 source programs. It is basically similar to well-described proposals for EULER machines [Wirth 65a, Weber 67]. In particular, it incorporates:

(1) the traditional ALGOL 60 stack organization and addressing structure [Rand 64],

(2) organization of composite data structures based on descriptor logic similar to that of the Burroughs B5500, and

(3) data-directed interpretation of operators.

Data storage for the interpreter is organized into a push-down stack and a free storage area. Composite data structures are implemented as collections of cells, defined by a descriptor scheme, in the free storage area. In the interpretation of a 'Polish suffix string, syllables of that string are sequentially scanned. Action specified by most such syllables falls into one of the following-classes:

(1) branching within the program string, possibly with analysis and modification of the top stack elements;

(2) fetching of values to the top of the stack, either from the program string or, under the direction of existing stack entries, from free storage;

(3) replacing a number of the top stack elements by a function of those elements, including constructing from them a composite data structure in free storage and placing a new descriptor in the stack;

(4) storing a stack value into a composite data structure as directed by other stack entries.

The interpreter is written in PL360 [Wirth 67c]. Its general organization resembles the EULER interpreter written for the Burroughs B5500 by Wirth and McKeeman [Wirth 65a]. A machine cell (System/360 double word) containing a list or string value actually contains a descriptor, which includes a type code and the base address and length of a contiguous block of machine cells that contain the values of the list elements or string characters. A compacting garbage collection scheme originally proposed by Weber [Wirth 65a] is used, so that available free storage always consists of a single contiguous area. Data-directed recursive application of certain operators is controlled by the interpreter procedures MAP, MAPLEFT, MAPRIGHT, and MAPBOTH.

D. Use of the MUTANT 0.5 Processor,

D.1 Language Restrictions

The following restrictions are imposed upon programs to be processed by the experimental system:

(1) The hardware character set (Appendix I) is used; thus the reserved words of that character set cannot be used as identifiers, and spaces are significant in delimiting adjacent reserved words or identifiers.

(2) No limit is imposed on the length of identifiers, but only the first eight characters are used in distinguishing them.

(3) No single string constant can consist of more than 256 characters.

(4) Arithmetic operations are defined by the IBM System/360 hardware. In particular, addition, subtraction, and multiplication are actually the corresponding operations in the ring of integers modulo 2^{32} (with appropriate interpretation of negative numbers).

In addition, certain valid MUTANT 0.5 programs can cause overflow of compiler tables or object code instruction fields (see D.3).

D.2 Operating Instructions

The MUTANT 0.5 compiler and interpreter must be compiled and the object programs placed in the PL360 system library by the use of SYSTUP [Wirth 67d]. In systems to date, these programs have been named MUTANT 1 and MUTANT 2 respectively. The compiled MUTANT 0.5 program is written by the compiler onto logical device 8, which must be appropriately defined, and is read by the interpreter initialization process. The following deck set-up (within a PL360 batch) is then required:

```
%MUTANT 1
      (MUTANT 0.5 source program)
%MUTANT 2
      (data, if any)
%EOF
```

D.3 Compilation Listing

The source program is listed as it is compiled. The hexadecimal numbers printed to the left of each line indicate the number of bytes of object program produced prior to analysis of that line. Under the RASP spooling system, the printed time is primarily a measure of the time required to load

the compiler or interpreter.

The following messages correspond to errors detected by the compiler. A vertical bar is printed beneath the character being scanned at the point of error recognition, and compilation is terminated. A possible error recovery technique has been described by Wirth [Wirth 67c].

SYNTAX	A syntax error (according to the grammar of Appendix II) was detected.
PROG OVFL	A program assembly area in the compiler overflowed.
BRANCHADDR OVFL	The relative address generated for an implicit branch overflowed the allocated instruction field.
CASE TABLE OVFL	An internal table used in processing case expressions overflowed.
UNDCL ID	An undeclared identifier was used.
IMPROPER ID	An identifier associated only with a value (named value, procedure parameter value, or controlled value) was used in a context (assignmerit or reference formation) in which an identifier associated with a cell is required.

E. Reflections on Language Design

The present MUTANT 0.5 system would benefit 'substantially from further development. There are a number of rough edges in the language definition and several known errors in the design of the interpreter. Some of the more unpleasant features of the language reflect oversights or poor decisions in the system design, and no conceptual problems arise in their elimination. Some examples are cited in section E.3.e. Other rough edges are related to fundamental questions about the design and use of MUTANT-like languages; some progress in resolving these questions should be made before further detailed implementation work is justifiable. The remainder of this section is an attempt to characterize such languages, to consider their potential as practical programming tools, and to discuss some specific issues raised by the definition and implementation of MUTANT 0.5.

E.1 MUTANT-like Programming Languages

In the past few years, several languages which attempt to extend and simplify ALGOL 60 [Naur 63] have been designed and experimentally implemented at Stanford. The two most directly of interest, in addition to MUTANT 0.5, are Wirth and Weber's EULER [Wirth 65a] and McKeeman's MUTANT [McKee 66]; the following remarks should also be applicable in part to similar languages, such as LISP 2 [Abra 66] and the AED family [Ross 66], being developed elsewhere. To a first approximation, theselanguages may be considered ALGOL 60 extended to allow various types of list (ordered set) manipulation. In particular, such languages include the following features:

(1) Programs consist of conditionally selected sequences of imperatives.
(2) Named variables are provided in the context of a block and declaration structure.
(3) An assignment operator is provided.
(4) Structured values may be created and manipulated dynamically, and the format of these structures need not be defined prior to program execution.
(5) Definitions of certain operators are extended to be dependent upon dynamic analysis of the operands.

E.2 Practical Applications

For purposes of this analysis, problems currently amenable to computer

attack fall into the following three broad catagories:

(1) Problems in which the natural data structures are simple, fixed, and reasonably well reflected in the storage'organization and operation set of existing machines. Many problems of classical numerical analysis fall in this catagory. In many cases, efficient use of the machine hardware is essential.

(2) Problems in which the natural data structures are complex but predetermined and well-defined. Processing requirements may or may not be easily satisfied by machine facilities. Much of systems programming and business data processing belongs in this catagory. Again a premium Is often placed on efficiency.

(3) Problems in which the natural data structures are complex and cannot be predefined. Examples are found in such areas as artificial intelligence, general symbol manipulation, and graphical data processing. In most cases, a moderate amount of avoidable system overhead is acceptable if it significantly increases flexibility and ease of programming in the system.

Experience with MUTANT 0.5 indicates that algorithms for solving problems in the first and third catagories can be naturally expressed in a MUTANT-like language. Since the structure of values is arbitrary in such a language, a uniform scheme (e.g., indexing) must be used to name substructures. Algorithms in the second class, however, can usually be expressed more clearly in the notation advocated by Wirth [Wirth 66a, 67a] in connection with record classes, a notation which demands static specification of possible data structures.

Experience also suggests that a simple translator-interpreter mechanism for a MUTANT-like language is unable to achieve the high efficiency required in applications in the first two areas. Translator recognition of, and optimization for, simple cases is, in fact, precluded by the lack of a descriptive declaration facility in EULER and MUTANT. Such a declaration structure, possibly including the record concept, could be used to advantage only by a considerably more sophisticated translator; even then, it is not clear that a great deal of efficiency can be gained without sacrifice of all dynamic features. Thus it appears that, with current machine designs, MUTANT-like languages are of potential practical value in the third problem area above and that they may be fairly evaluated in the context of such

problems.

E.3 Language Design

Presented below are some of the issues which were found to be critical in the design and use of the MUTANT 0.5 system. Some of these became clear only after much of the system had been implemented, and no claim is made that many optimal, or even good, solutions were found.

E.3.a Assignment of Structured Values

In MUTANT-like languages, declarations serve to name cells but not to define their structure. Instead, structured values may be created in an arbitrary way by computation, and such values may be assigned to any named (sub)cell; at the time of assignment, that cell assumes the structure of the assigned value. Thus the structures of cells must be dynamic. The principal objections to such a scheme have been discussed by Wirth [Wirth 67a, 67b]. Briefly, they are the following:

(1) Restructuring of cells is highly implicit, generally expensive in interpretation, and deceptively simple in appearance to the programmer. Known storage allocation and referencing methods for implementation are not efficient enough, especially in the first two of the problem areas above.

(2) Subcells (subvalues) must be referenced by a fixed and uniform naming scheme (such as indexing) with little mnemonic value.

(3) The compiler has very limited information for selecting code, type-checking, etc.

Wirth [Wirth 67a,67b] proposes to avoid these problems by assigning to each named cell a structure, possibly complex, fixed at the point of declaration. He claims that "for practical purposes this turns out to be hardly a restriction at all" [Wirth 67a, p. 3]. The claim is reasonable for programs arising in the first two problem areas above, but it is questionable as a general assertion. Among evidence to the contrary are the following points:

(1) Programs from the third problem area inherently deal with dynamic, complex, and interacting data structures. The information content of such structures can indeed be represented within a set of static data structures, but often this requires considerable bookkeeping effort on the part of the problem programmer and makes the resulting

program difficult to write, to document, and to modify or extend without drastic revision.

(2) Experience with MUTANT 0.5 indicates that some of the most useful and convenient features of the language generate or depend on dynamically structured values. Notable examples are the iterative expression and the Iverson interpretation of certain operators.

It is tempting to conclude that a desirable solution is to allow the programmer to specify that a cell must have structures from some subset of the set of structures of all values computable within the system. In particular, if the specified subset contains exactly one element, the translator is expected to check and optimize appropriately. There is some merit in such a scheme; however, experience suggests that the effort required to produce and adequately test such a translator using currently known techniques is usually very great, even for languages much "simpler" than MUTANT 0.5. In addition, the optimization gained has often been rather disappointing.

E.3.b The Name-Value Problem

A familiar problem in the design of programming languages is distinguishing the denotation of the name of a cell and the name of the contents of that cell (or more generally, the name of an expression and its value). EULER and MUTANT 0.5 (but not MUTANT) resolve this problem by allowing (and normally requiring) the programmer to make the distinction. Thus in MUTANT 0.5, "a" denotes the value contained in the cell a, while "ℓa" is the name of (address of, pointer to, etc.) the cell itself. A concession to tradition is made in assignment; Although this is an operation between a value and a cell, MUTANT 0.5 allows, e.g.,

b + 3 → a

in place of

assign (b+3, ℓa) or b + 3 → ℓa

Allowing the programmer to explicitly manipulate cell names creates some subtle but fundamental problems in MUTANT-like languages:

(1) The role of block structure and the interpretation of declarations is unclear, as illustrated by the following example:

{ $ a | { $ b | ℓb → a }, 0 → a• } .

If the second assignment is considered valid, then the cell b must remain accessible after it can no longer be directly named; in

particular, the machine storage assigned to b cannot be reallocated after exit from the block (list) to which b is local. On the other hand, if the second assignment is considered invalid, detection of such assignments within the block and procedure structure of MUTANT-like languages becomes a surprisingly subtle problem, and no satisfactory solution was discovered.'The difficulty of the problem is indicated by the following example:

```
{ $ x y p |
   ' a | {$b | ℓb → a•, x → y, 0 → x }' → p,
   p{ ℓx }, { $ c | 0 → y• }
}
```

A quite similar problem arises in the assignment of values of type process, as indicated in the following:

```
{ $ p |
   { $ a | 10 → a, ' x | x + a ' → p },
   PC 3}
}
```

(2) Names which are meaningful at the point of creation may become meaningless at the point of use due to the dynamic structuring of cells, as shown in the example below:

```
{ $ a b | { 1, 2,3 } → a, ℓa[1] → b, 0 → a, 1 → b• }
```

Such situations cannot be detected easily by an interpretation mechanism using machine addresses or equivalents as the representation of values of type MUTANT. MUTANT 0.5 effectively treats such values as parameterless procedures which return a machine address upon activation. This solution also defers evaluation of subscripts, sometimes with undesirable results. A better scheme is to construct a similar procedure after evaluation of all subscripts, but such a solution can be quite expensive.

E.3.c The Copy problem

In MUTANT 0.5, the traditional notion of assignment of values to cells has been retained. This decision has fundamental implications for the design of an interpreting mechanism implemented using a conventional digital computer. In such machines, cells have simple fixed structures,

and values are generally not accessible except as contents of such cells. As a result, the structured cells (values) of MUTANT-like languages must be implemented as collections of machine cells (values). Furthermore, since structure is dynamic in such languages, these collections must include descriptive information sufficient to identify the structure.

In interpreting the assignment of such structured values (possibly contained in named cells) to named cells, the question arises of how much of this collection must of logical necessity be copied upon assignment. For example, the interpretation mechanism must compute 3, not 0, as the value of the following expression:

```
{ $ a b I { 1, { 2, 3 } } → a, a[2] → b, 0 → b[2], a[2][2] }[4]
```

The answer is that, if by any name and process, the contents of a machine cell can be changed, there must be at most one name (which may, however, be the value of the contents of any number of cells) through which that machine cell or its contents can be referenced. Such names are created by explicit or implicit assignment to a structured cell. Implicit assignments in MUTANT-like languages include use of a value as a procedure actual parameter as well as the implicit assignments within an iterative expression.

In the implementation of interpreters of MUTANT-like languages, assuring such uniqueness proves to be very expensive in terms of efficiency. Such implementations to date have used an interpreter based upon a push-down stack, manipulated by program operators, and a free storage area of machine cells, from which structured cells are created. Uniqueness of reference to machine cells can be guaranteed by unconditionally copying completely every structured value as it (or a descriptor of it) is fetched to and stored from the stack. Copying is itself expensive; moreover, each copying reduces the (finite) number of machine cells in free storage available. Eventually, free storage must be restructured ("garbage collected"), and a second substantial expense is incurred. Implementations to date have, in fact, attempted to avoid some of this copying. In Wirth and McKeeman's B5500 implementation of EULER [Wirth 65a], for example, values are copied only upon fetch-into the stack; as a result, in that EULER implementation,

expressions such as

a ← b ← c

and p(a ← c)

are semantically disallowed if (and only if) c is found to contain a structured value at the time of interpretation. An alternate approach is to adopt a scheme of including marking information with (sub)structures and deferring copying until it is logically demanded. In this investigation, no such scheme was discovered which seemed sufficiently attractive (see below).

In view of the expense of copying, it is important to note that in mostcases such action is neither anticipated nor desired by the programmer. Furthermore, in many cases, difficult or impossible to detect during the translation process, omission of such copying will not change any of the final values produced (or, even more frequently, any of the output strings written). Given the high cost of copying and associated storage management in available machines, this observation is probably the basis of the most fundamental objection to the practical use of MUTANT-like languages. A number of partial solutions to the copy problem are considered below.

(1) In EULER and MUTANT 0.5 programs, it is possible to create a value of type name. This facility creates certain logical problems (see above), but it is valuable in allowing the programmer to create references to a named cell (and hence the contained value). In certain situations (not necessarily obvious to the programmer), it will be more efficient to access a value with complex structure indirectly via a reference than it will be to copy the value. Such indirect reference is particularly natural and appropriate in connection with procedure parameters. It has several drawbacks:

   (a) Each value to be indirectly referenced must first be assigned to to some named cell.

   (b) Efficiency is critically dependent upon the programmer% careful (implementation dependent) choice of reference or value in each situation.

   (c) The programmer must be exactly aware at all times of the level of indirectness being used.

(2) In SLIP [Weiz 63], a list-processing language of quite different design,

a superficially similar problem was encountered and solved by the use of a reference counting scheme. A count of the number of valid names referencing each relevant collection of machine cells is encoded in that collection and dynamically adjusted. A brief examination failed to discover a reasonably efficient adaptation suitable for MUTANT-like languages, but further investigation might be profitable., Briefly the difficulty seems to be that the encodings of such counts which can be efficiently maintained are not the encodings efficiently usable in avoiding copying.

(3) It is possible to interpret the notion of value in a manner consistent with any particular scheme of internal representation and strategy of copying that is convenient for implementation. In particular, if a cell contains a structured value in the MUTANT 0.5 sense, it is attractive to instead consider the value of the cell to be a description of that structured cell and its subcells. In certain situations (such as array procedure parameters called by name in ALGOL 60) such an interpretation is consistent with the spirit of the language and represents an efficient implementation trick. In general, however, there are several valid objections to such an interpretation:

(a) It is an ad hoc expedient and tends to make the semantics of a language dependent upon the implementation facilities which happen to be available.

(b) It further confuses the distinction between the name of a cell and of its value.

(c) As most naturally implemented, an embarassing lack of consistency arises in the meaning of the language. In particular, it is more convenient and efficient to reference unstructured values directly but structured values indirectly.

(4) Analysis of programs in various languages with an assignment operator suggests that a significant fraction of all cells are declared and used to preserve intermediate results and avoid repeated calculation of the same value. Such cells are created for the purpose of naming values; the fact that these cells (as opposed to the contained values) are structured is of no interest or use to the programmer, for he never assigns to a subcell. This suggests that the language should provide a facility for naming computed values without requiring assignment to a logically

distinguished cell. Such a facility exists for simple constant values in present languages.

MUTANT 0.5 recognizes that previously computed values may be used as such intermediate values and thus may effectively be 'constants throughout the scope of a declared name. A construct is provided to initialize at the point of declaration the value denoted by a **name** to a constant (which may be computed **from** the values denoted by names non-local to the corresponding block). Such values will be called locally **constant.** Since all procedure parameters in MUTANT 0.5 are effectively called by value, it is easy for the translator to check that such names are never used in the (implicit or **explicit)** formation of cell **names.**

The important fact is that such naming does not create a name by which the contents of a machine cell can be changed. Thus in the composition of the designated value, any **subvalue** which is a **constant,** either by denotation or by being locally constant in a containing block, need not be copied. The idea can be extended **somewhat** further than is done in MUTANT 0.5. **If** a value contained in a named cell is used in the **computation** of a locally constant value, then there are various sets of sufficient **conditions,** verifiable by the translator, that insure that the contained value cannot be changed within the scope of the name of the local constant. If these conditions are satisfied, it is again not necessary to copy the contained value in formation of the local constant.

The effectiveness of this solution is critically dependent upon the programmer's style. Programmers experienced with LISP **1.5** [McCar 62] find it relatively easy to make effective use of local constants; in fact, such use is **very similar to** one use of LISP h-expressions. There is also a trade-off of run-time efficiency versus **compiler** speed; in particular, **code generation** based on a very sophisticated set of sufficient conditions for local constancy is probably incompatible with one-pass translation. Experience with the MUTANT 0.5 interpreter suggests that the most **promising** approach to the copy problem is a finer distinction **among** the various uses of the traditional assignment operator and a syntactic structure which **distinguishes** among such uses. The provision of ":=", "=", and "≃" for assignment, "initialization by value", and "initialization by reference", **respectively,** in **CPL** [Buxt 66] reflects exactly such a distinction. **Landin's** let

and where constructs [Land 66] are also used in CPL and are attractive syntactic devices for designating local constants.

E.3.d Extended Operator Definitions

In MUTANT and MUTANT 0.5, definitions of operators have been extended in the sense of Iverson [Iver 62] whenever possible. Such extension leads to at least three difficulties:

(1) For the results of a given computation to be well defined, the exact order of the evaluation of operands as well as the application of operators must be specified. This is due to the involution of assignment as well as the possibility of procedures with side effects. Difficulties are not limited to pathological cases; using the extended assignment operator of MUTANT, McKeeman [McKee 66] illustrates a useful application of

$$\{ a, b \} \leftarrow \{ b, a \} \quad .$$

Specification of either complete evaluation of both operands in a specified order followed by operator application (as in MUTANT 0.5) or any of various levels of conceptual parallelism is likely to lead to gross inefficiencies in some implementations. McKeeman [McKee 67] has suggested a partial solution based upon the distinction between types set (unordered) and list (ordered).

(2) The meanings of operators intended to act upon structured values generally cannot be extended without the loss of such meaning. For example, the value of

$$\{ 1, 2, 4 \} = \{ 1, 3, 4 \}$$

will be 0 or { 1, 0, 1 } depending upon the interpretation of the extended equality operator: In the first case, the extended meanings of equality operators will be very different from those of the other relational operators; in the second case, comparisons of structured values must be explicitly programmed (as in MUTANT 0.5) or require another equality operator. Extension of the subscripting and assignment operators present special difficulties:

(a) There are two common interpretations of subscript notation. In one, such notation is considered simply a naming device. In the other, the subscript brackets are considered to denote an operator which maps a value (or cell name) and a numerical value into a subvalue

(or subcell name). From this viewpoint, there is a natural generalization of the subscript operator: a value subscripted by an ordered set (list) yields an ordered set of values obtained by applying each element of the set as a subscript, e.g.,

a[{2,{3,4}}] ≡ a sub {2,{3,4}} ≡ { a[2],{a[3],a[4]} } .

Such an interpretation allows a very powerful and elegant method of constructing new ordered sets from a collection of elements and has been adopted by MUTANT and MUTANT 0.5. Note, however, that the extension is not quite Iverson's; the subscripted value must be structured but must be treated formally as unstructured. Furthermore, if cell names are allowed to be subscripted by sets (as in MUTANT), the result must be a collection of (sub)cell names, and one is led to an extended interpretation of assignment. If such subscripting is not allowed (as in MUTANT 0.5), string manipulation is quite awkward and an asymmetry is introduced in the language.

(b) There is a fairly obvious similar extension of the assignment operator. It is again, however, not quite the Iverson extension used elsewhere in MUTANT 0.5, for one would prefer

{ 1, { 2, 3 } } → { a, b } ≡ { 1 → a, { 2, 3 } → b }

instead of

{ 1, { 2, 3 } } → { a, b } ≡ { 1 → a, { 2 → b, 3 → b } } .

In addition, sequencing is critically important in assignment; by one possible definition,

{ a, b } → { b, a } ≡ { a → b, b → a } ,

which is usually not the desired interpretation.

(3) Some data types are not either clearly structured or clearly unstructured. The primary examples in MUTANT 6.5 are strings. It is desirable, for example, to be able to access substrings by the subscript notation for structured values; on the other hand, when used as operands to, e.g., the put operator, it is convenient to consider them unstructured. A hierarchy of structure can be introduced, but probably at the cost of some loss of uniformity, and hence simplicity, in the interpretation mechanism.

E.3.e Miscellaneous Problems

A number of decisions made in the design and implementation of MUTANT 0.5

were later found to be mistakes, but these mistakes do not reflect fundamental problems in the design of MUTANT-like languages. Some of these are listed below:

(1) Choice of Character Set

In the design of MUTANT 0.5, it was decided to choose as concise a notation as possible and to reflect the usage of set theory as well as conventional algebra. When desired special symbols were not available in the IBM EBCDIC character set, they were usually represented by pairs of special characters rather than by word delimiters'or reserved words. The elegance of this approach is debatable; however, it is clear that readability suffers severely, especially in the hardware representation.

(2) Deletion Operator

The value of a MUTANT 0.5 program at any point is generally a very 'large list structure, the structure of which reflects the history of interpretation up to that point in considerable detail. Such lists consume a large amount of storage and often are of no practical use. A sequencing operator, similar to the comma but deleting the last value canputed for an element of the list being constructed, would be very useful, particularly when an expression is evaluated for its effect rather than its value.

(3) Extended Case Expressions

McKeeman [McKee 66] has demonstrated an elegant application of a list-valued case index in his MUTANT compiler. Such indices are prohibited in MUTANT 0.5 only because of an oversight in the design of the interpreter.

E.4 Methodology

Implementation of MUTANT 0.5 has followed the example of EULER and MUTANT; it is based upon a straight-forward compiler producing Polish postfix operator strings and a stack-oriented interpreter of such strings. For experimental purposes, such a system seems entirely adequate. Weber [Weber 67] has demonstrated the suitability of presently available hardware for implementing proven compilation and interpretation algorithms in microcode, and presumably results with specially designed hardware would be

even better than what he'reports,. In addition, it should be noted that with sufficiently powerful operators the additional overhead of interpretation is relatively small; for example, the MUTANT 0.5 interpreter makes quite efficient use of the System/360 general registers in vector manipulation when such manipulation is expressed in Iverson's notation, and this efficiency is possible without an optimizing compiler.

The grammar of MUTANT 0.5 was chosen to be a simple precedence grammar because of familiarity with the techniques involved and availability of suitable syntax processing programs. Other well understood formalisms, summarized by Feldman and Gries [Feld 67], could have been used equally well, with some trade-offs among speed, space, and generality. In general, it was found that, with the available machinery, modifications to the MUTANT 0.5 grammar or compiler were fairly trivial to make. On the other hand, many unfortunate features of the interpreter could not be changed without substantial rewriting; further investigation of the related problems of formal semantics and machine description seems more appropriate than continued work oriented entirely toward syntactic questions.

F. References

Abra 66 Abrahams, P., et al., The LISP 2 programming language and system, AFIPS Conf. Proc. 29 (Fall 1966), pp. 661-676.

Buxt 66 Buxton, J. W., Gray, J. C., and Park, D., CPL elementary programming manual, The University Mathematical Laboratory, Cambridge (January 1966).

Feld 67 Feldman, J. A., and Gries, D., Translator writing systems, Technical Report CS69, Computer Science Department, Stanford (June 1967).

IBM 66 IBM Systems Reference Library, PL/I: Language specifications, IBM Form C28-6571.

Iver 62 Iverson, K., A programming language, Wiley (1962).

Land 64 Landin, P. J., The mechanical evaluation of expressions, Comput. J. 6 (January 1964), pp. 308-320.

Land 66 Landin, P. J., The next 700 programming languages, Comm. ACM 9 (March 1966), PP. 157-166.

McCar 62 McCarthy, J., et al., LISP 1.5 programmer's manual, Computation Laboratory, MIT (1962).

McKee 66 McKeeman, W. M., An approach to computer language design, Technical Report CS48, Computer Science Department, Stanford (August 1966).

McKee 67 McKeeman, W. M., private discussion (Spring 1967).

Naur 63 Naur, P., et al., Revised report on the algorithmic language ALGOL 60, Comm. ACM, 6 (January 1963), pp. 1-17.

Pohl 67 Pohl, I., Graph package, GSG Memo 43, Graphics Study Group, SLAC, Stanford (June 1967).

Rand 64 Randell, B., and Russell, L. J., ALGOL 60 implementation, Academic Press, 1964.

Ross 66 Ross, D. T., AED bibliography, Mem. MAC-M-278-2, Project MAC, MIT (September 1966).

Shaw 66 Shaw, A. C., Lecture notes on a course in systems programming, Technical Report CS52, Computer Science Department; Stanford (December 1966).

Weber 67 Weber, H., A microprogrammed implementation of EULER on IBM System/360 model 30, Comm. ACM 10 (September 1967), pp. 549-558.

Weiz 63 Weizenbaum, J., Symmetric list processor, Comm. ACM 6 (September 1963), pp. 524-544.

Wirth 65a Wirth; N., and Weber, H., EULER: A generalization of ALGOL, and its formal definition, Technical Report CS20, Computer Science Department, Stanford (April 1965) (also, in part, Comm. ACM 9 (January and February 1966), pp. 13-25, 89-99).

Wirth 65b Wirth, N., Find precedence functions, Algorithm 265, Comm. ACM 8 (October 1965), pp. 604-605.

Wirth 66a Wirth, N., and Hoare, C. A. R., A contribution to the development of ALGOL, Comm. ACM 9 (June 1966), pp. 413-432.

Wirth 67a Wirth, N., On certain basic concepts of programming languages, Technical Report CS65, Computer Science Department, Stanford (May 1967).

Wirth 67b Wirth, N., ALGOL project memo 55 (internal memo), Computer Science Department, Stanford (1967).

Wirth 67c Wirth, N., A programming language for the 360 computers, Technical Report CS53 (revised), Computer Science Department, Stanford (June 1967).

Wirth 67d Wirth, N. (editor), The PL360 system, Technical Report CS68, Computer Science Department, Stanford (June 1967).

Appendix I

Publication / Machine Character Set Mapping

Publication Character Set	Machine Character Set (EBCDIC)	Publication Character Set	Machine Character Set (EBCDIC)
a	(no equivalent)	⊗	*
...	...	+	DIV
z	(no equivalent)	mod	MOD
A	A	base	BASE
...	...	=	=
Z	Z	≠	¬=
0	0	>	GT
...	...	≥	GTE
9	9	<	LT
"	"	≤	LTE
{	<	∧	AND
}	>	∨	OR
(	(	/	/
)	)	ℓ	∂
[	(_	.	4
]	_)	→	->
$	$		
|	|	'	'
,	,	=>	=>
get	GET	:	:
#	#	;	;
↑	↑	∈	&
list	LIST	*	.
abs	ABS	eof	!
neg	NEG	(comment bracket)	?
put	PUT		
type	TYPE		
+	+		
-			

```
*SYNPROC

      1   <PROG>          ::= !  <V-EXPR>  !
      2   <G-EXPR>        ::= <V-EXPR>
      3                       <Q-CLAUSE> <G-EXPR>
      4   <Q-CLAUSE>      ::=  <V-EXPR>  =>
      5   <V-EXPR>        ::= <V-EXPR*>
      6   <V-EXPR*>       ::= <SEGMENT>
      7                       <PROC  HD>  <V-EXPR>  '
      8                       <V-EXPR*>  ->  <CELL>
      9   <PROC HO+>      ::= '
     10                       <PROC HO+>  (IDENT)
     11   <PROC HO>       ::  =<PROC HO+>  |
     12   <SEGMENT>       ::  =<S-EXPR>
     13                       <S-EXPR>  _  <S-EXPR>
     14                       <S-EXPR>  _  <S-EXPR>  _  <S-EXPR>
     15   <S-E XPR>       ::= <S-EXPR*>
     16   <S-EXPR*>       ::= <PRIM>
     17                       <S-EXPR*>  <INFIX>  <PRIM>
     18   <INFIX>         ::= +
     19
     20                       *
     21                       DIV   .
     22                       MOO
     23                       BASE
     24                       =
     25                       ¬=
     26                       GT
     27                       GTE
     28                       LT
     29                       LTE
     30                       AND
     31                       OR
     32                       |
     33   <PRIM>          ::= <PRIM*>
     34                       <PREFIX> <PRIM>
     35                       <INFIX> / <PRIM>
     36   <PREFIX>        ::= #
     37                       ¬
     38                       LIST
     39                       ABS
     40                       NEG
     41                       PUT
     42                       TYPE
     43   <PRIM*>         ::= <CELL>
     44                       <REF>
     45                       <S-PRIM>
     46   <CELL>          ::= <CELL*>
     47   <CELL*>         ::= (IDENT)
     48                       <CELL*>(_  <V-EXPR>  _)
     49                       <CELL*>  %
     50   <REF>           ::= <REF*>
     51   <REF*>          ::= @  (IDENT)
     52                       <REF*>  (_  <V-EXPR>  _)
     53   <S-PRIM>        ::= <S-PRIM*>
     54   <S-PRIM*>       ::= <CONSTANT>
     55                       GET
     56                       <LIST>
     57                       <PRIM*>  <LIST>
     58                       (  <V-EXPR>  )
```

```
59                          <CASE  HD>  <V-EXPR>  )
60                          <FOR H O > >
61                          <WHILE H O > >
62                          < F / W HD> >
63                          < S - P R I M * > (_ <V-EXPR> _)
64      <CONSTANT>    ::=   (INTEGER)
65                          (STRING)
66                          <BEGIN> >
67      <LIST>        ::=   <LIST  HO>  >
68      <LIST HD>     ::=   <BEGIN> <G-EXPR>
69                           < L I S T HD> ,  <G-EXPR>
70      <BEGIN>       ::=   <
71      <BEGIN>       ::=   = <DECLARE> |
72      <DECLARE>     ::=   <  $
73                          <DECLARE>  (IDENT)
74                          <DECLARE> (IDENT)  (  <V-EXPR>  )
75      <CASE HO>     ::=   <CASE IDX>  (  <V-EXPR>  ;
76                          < C A S E HD> <V-EXPR> ;
77      <CASE I OX>   ::=   (_  <V-EXPR>  _)
78      <FOR HD>      ::=   <BEGIN> <FOR SET> <G-EXPR>
79      <WHILE HD>    ::=   <BEGIN>  <WHILE  CDN> <G-EXPR>
8C      <F/W HD>      ::=   <BEGIN> <FOR  SET> <WHILE CON> <G-EXPR>
81      <FOR SET>     ::=   <FOR SET*>  :
a2      <FOR SE T*>   ::=   (IDENT)  &  <V-EXPR>
83      <WHILE>       ::=   .
84      <WHILE'>      ::=   <WHILE> <Q-CLAUSE>
a5      <WHILE CDN>   ::=   <WHILE'> :
```

PRECEDENCE FUNCTIONS

1	<PROG>	1	1
2	<G-EXPR>	3	2
3	<Q-CLAUSE>	2	3
4	<V-EXPR>	5	4
5	<V-EXPR*>	4	5
6	<PROC HO+>	9	5
7	<PROC HD>	4	5
a		7	5
9	<S-EXPR>	7	5
10	<S-EXPR*>	a	6
11	<INF IX>	6	a
12	<PR I M>	10	6
13	<PREFIX>	6	7
14	<PRIM*>	10	7
15	<CELL>	12	7
16	<CELL*>	12	8
17	<REF>	12	7
18	<REF*>	12	7
19	<S-PRIM>	12	7
20	<S-PRIM*>	12	7
21	<CONSTANT>	13	7
22	<LIST>	13	10
23	<LIST HO>	2	11
24	<BEG IN>	2	11
25	<DECLARE>	9	11
26	<CASE HD>	4	7
27	<CASE IDX>	13	7
28	<FOR HD>	2	7
29	<WHILE HO>	2	7
30	<F/W HD>	2	7
31	<FOR SET>	2	2
32	<FOR SET*>	1	3
33	<WHILE>	3	3
34	<WHILE'>	1	3
35	<WHILE CDN>	2	2
36	!	4	5
37	=>	14	5
38	'	10	5
39	->	7	6
40	(IDENT)	13	9
41		14	9
42	_	5	7
43	+	14	9
44	-	14	9
45	8	14	9
46	DIV	14	9
47	MOD	14	9
48	EASE	14	9
49	=	14	9
50	¬=	14	9
51	GT	14	9
52	GTE	14	9
53	LT	14	9
54	LTE	14	9
55	AND	14	9
56	OR	A4	9
57		6	6
58	#	14	7
59	¬	14	7
60	LI ST	14	7
61	ABS	14	7
42	NEG	14	7
63	PUT	14	7
64	TYPE	14	7
65	(_	4	1'2
66	_)	14	5
67	%	13	12
68	@	9	7
69	GET	13	7
70	(	4	13
71	)	13	5
72	>	13	2
73	(INTEGER)	13	7
74	(STRING)	13	7
75	,	2	2
76	<	14	11
77	$	10	14
78	;	14	5
79	:	14	1
a0	&	4	13
81	.	14	3

ELAPSED J IME IS 00:06:00

```
%MUTANT 1
                ?
  0000          $   FACTORIAL CALCULATION - SECTION 8.3.1   ?  .
  0000        <   FACTORIAL |
  0001          ' N I (_ 2-(N=0)_)(1;N*FACTORIAL< N-1 > )' -> FACTORIAL
  0032          < N & 1_6 :
  003D             PUT((N  BASE  10)|" FACTORIAL = " | (FACTORIAL< N > BASE 10| )
  0060          >
  0061        >

END          OF COMPILATION .

ECAPSEC TIME IS 00:00:49
```

```
%MUTANT 2
1 FACTORIAL = 1
2 FACTORIAL = 2
3 FACTORIAL = 6
4 FACTORIAL = 24
5 FACTORIAL = 120
6 FACTORIAL = 720

ELAPSED TIME IS 00:00:36
```

```
%MUTANT 1

  0000          ?   EXTENDED  FACTORIAL CALCULATION - SECTION 8.3.2    ?
  0000        <$ FACTORIAL PROD|
  0002           < ' N |(_ 2-(N=0) _) ( 1 ; N*FACTORIAL(_1_)< N-1 > ) ' ,
  0033             'N| */( 1_N) ' ,
  0043             ' N | PROD< 1_N > '
  0053           >-> FACTORIAL,
  0059           ' L |  (_ 2-(#L=1) _)
  0067                 ( L(_1_) ; PROD< L(_ 1_#L DIV 2 _) >*PROD< L(_ #L D | V 2+1_#L _) > )
  00A4              -> PROD,.
  00A9
  00A9           < | & 1_3 :
  0004              C PUT"", PUT("METHOD "|(I BASE 10) ),
  00CA                 < N & 1_8 3
  00D5                    PUT((N BASE 10)|" FACTORIAL ="|(FACTORIAL(_I_)< N > BASE 10) )
  00FC                 >
  00FD              >
  0100           >
  0101        >

ENDCF COMPILATION

ELAPSED TIME IS 00:00:40




%MUTANT 2


METHOD 1
1 FACTORIAL = 1
2 FACTORIAL = 2
3 FACTORIAL = 6
4 FACTORIAL = 24
5 FACTORIAL = 120
6 FACTORIAL = 720
7 FACTORIAL = 5040
8 FACTORIAL = 40320

METHOD 2
1 FACTORIAL = 1
2 FACTORIAL = 2
3 FACTORIAL = 6
4 FACTORIAL = 24
5 FACTORIAL = 120
6 FACTORIAL = 720
7 FACTORIAL = 5040
a FACTORIAL = 40320

METHOD 3
1 FACTORIAL = 1
2 FACTORIAL = 2
3 FACTORIAL = 6
4 FACTORIAL = 24
5 FACTORIAL = 120
6 FACTORIAL = 720
7 FAC  JOR IAL = 5040
8 FACTORIAL = 40320

ELAPSED TIME IS 00:00:43
```

```
%MUTANT 1

0000          ?   INNER PROOUCT - SECTION B.3.3(A)   ?
0000           PUT ( ( +/( < 1,2,3 >*< 3,2,1 > ) ) BASE LO )

END Q F COMPILATION

ELAPSED TIME IS 00:00:35


%MUTANT 2

10

ELAPSED TIME IS 00:00:37
```

```
%MUTANT 1

  0000          ?   PERMUTATION GENERATOR - SECTION 8.3.3(B) ?
  oooc        < $ PERM |
  000 1           ' x  | (_ 2-(#X=1) _)
  0013              ( < X > ;
  001A                |/< I & 1_#X :
  0029                    < J &PERM<X(_ 1_I-1 _)|X(_ I+1_#X _) > : X(_ I_I _)|T >
  006 1                 >
  0064              ) ' -> PERM,
  0076          <TEST&<"A","AB", "ABC","ABCD">:PUT PERM<TEST>>
  009A        >

END OF COMPILATION

ELAPSEC TIME IS 00:00:36.


%MUTANT 2

A
A8
BA
ABC
ACB
BAC
BCA
CAB
CBA
ABCD
ABDC
ACBD
ACDB
ACBC
ADCB
BACD
BADC
BCAD
BCDA
BDAC
BDCA
CABD
CAD8
CBAC
CBDA
CDAB
CDBA
DABC
DACB
DBAC
DBCA
DCAB
DCBA

ELAPSED TIME IS 00:00:43
```

```
%MUTANT 1

0000          ?   PRIME SIEVE SAMPLE PROGRAM - SECTION 8.3.4   ?
0000        C $PRIMESIEVE
0000           ( N  I< $ L T | 2_N -> L,
0014                   < . #L ¬= □ => :
001E                    < L(_1_) ->T,<|&L:I MQD T ¬= 0 => I > -> L >(_1_)
004 8                  >
004C                 >(_2_) "
0053           ) |
0054           < N & < 25, 250 > :
0064             <  PUT ("PRIMES IN 2 TO "|(N BASE 10)|":"),
0082                PUT(PRIMESIEVE< N > BASE 10)
008E             >
008F           >
0090        >

END OF COMPILATION

ELAPSED TIME IS 00:00:40


%MUTANT 2

PRIMES IN 2 TO 25:
2
3
5
7
11
13
17
19
23
PRIMES IN 2 TO 250:
2
3
5
7
11
13
17
19
23
29 31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
101
103
107
109
113
127
131
137
139
149
151
157
163
167
173
179
181
191
193
197
199
211
223
227
229
233
239
241

ELAPSED TIME IS 00:01:09
```

```
%MUTANT 1

  0000        ?   PERMUTATION GENERATOR ?
  0000        < $ PERM |
  0001            ' x | (_ 2-(#X=1) _)
  0013              ( < X > ;
  001A                |/< I & 1_#X :
  0029                     < T & PERM< X(_ 1-I-1 _) | X(_ I+1_#X _) >:
  0052                      < X(_ I _) > | T
  005B                     >
  005F                  > )' -> PERM,
  0074            C TEST &< 1_1, 1_2, 1_3, 1_4 > :
  009C              < P & PERM< TEST > :PUT(|/(P BASE 10) ) >
  00AB             > >

END OF COMPILATION

ELAPSED J IME IS 00:00:39



%MUTANT 2

1
12
21
123
132
213
231
312
321
1234
1243
1324
1342
1423
1432
2134
2 143
2314
2341
2413
2431
3124
3142
3214
3241
3412
3421
4123
4132
4213
4231
4312
4321

ELAPSED TIME IS 00:00:44
```

```
%MUTANT 1

  0000          ? GRAPH MANIPULATION ROUTINES ?
  0000        C $ REACHVEC
  0000          (' c I I
  0004            < $  PR  R  BB  N (#C)|
  000B              ( 1_N)¬=I -> BB, C(_I_) -> R,  LIST N -> PR,
  0030              < . +/(PR=(R->PR))¬=N => :
  0047                < J &  1_N : BB(_J_) A  N  D PR(_J_) =>
  0062                  < 0  -> BB(_J_), R  OR  C(_J_) -> R > >
  007F              >, R >(_5_) ')
  0080            DISPLAYMATRIX
  0080          (' C TITLE | <PUT " ", P  U  T TITLE,
  009A            < I & 1_#C : P U T(|/(C(_I_) B A S E 2)) >> ')
  00BC          REACHMATR  I  X MAXSC SUBGRAPH T
  00BE          c ( < <1,1,0,0,0,0>, <0,1,0,1,0,0>, <0,1,1,0,0,0>,
  00EA              <0,0,0,1,1,0>, <0,1,0,0,1,0>, <1,0,0,0,0,1> > )  I
  0115          ' c I < I & 1_#C  : REACHVEC<'C, I >3 ' -> REACHMATRIX,
  013E          'R|<   1   &1_#R : < J &  1_#R :R(_I_)(_J_) A N D R(_J_)(_I_) > > '
  0179            -> MAXSC SUBGRAPH,
  017E          DISPLAYMATRIX< C, "C M A T R I X" >,
  0191          OISPLAYMATRIXC REACHMATRIX< C >, "R MATRIX" >,
  01AA          DISPLAYMATRIX<  MAXSC SUBGRAPH< REACHMATRIX< C > >, "MSC M A T R I X" >

END CF COMPILATION

ELAPSED T I M E  I S 00:00:42


XMUTANT 2


C MATRIX
1 10000
010100  .
011000
000110
010010
10000 1

R  MATRIX
1101.10
010110
011110
010110 -
010110
110111

MSC MATRIX
100000
010110
001000
010110
010110
00000 1

ELAPSED TIME IS  00:00:48
```

```
%MUTANT 1

  0000        < $ SQRT
  oooc           (' N |< $  X  ERR  |
  0006              ( N DIV 2 -> X )*N*2-> ERR,
  001F              C . ERR  GT(ABS( X*X -- N )->  ERR  ) =>:  ( X + (N DIV X)) DIV 2 ->
  004B              x   >(_3_) ' )
  0053            PRIMESIEVE
  0053           (' N | < $ L  T  | 2_N -> L,
  0067            <. #L¬=0 => :
  0071              < L(_1_) -> T, < I & L : I MOD T ¬= 0 => I > -> L >(_1_) >
  009F            >(_2_)' ) |
  00A7           PUT" PRIMES AND INTEGERIZED SQUARE ROOTS",
  0002           < I & PRIMESIEVE<   250  >:
  00E0              PUT (( I BASE 10 )|"    "|( SQRT< I > BASE 10 )) >  >

END GF COMPILATION

ELAPSE0 TIME IS 00:00:39



%MUTANT 2

 PRIMES ANG INTEGERIZEDSQUARE  ROOTS
2     1
3     1
5     2
7     2
11     3                          149     12
13     3                          151     12
17     4                          157     12
19     4                          163     12
23     4                          167     12
29     5                          173     13
31     5                          179     13
37     6                          181     13
41     6                          191     13
43     6                          193     13
47     6                          197     14
53     7                          199,    14
59     7                          211     14
61     7                          223     14
67     8                          227     15
71     8                          229     15
73     8                          233-    15
79     8                          239     15
83     9                          241     15
89     9
97     9                          ELAPSED TIME IS 00:01:17
101     10
103     10
107     10
109     10
113     10
127     11
131     11
137     11
139     11
```