Technical Report TR-223 January 1973
N00014-67-A-0239-0021
(NR-044-431)
NGL-21-002-008

SIMPL-X
A Language for Writing Structured Programs

by

Victor R. Basili

This research was supported in part by the Office of Naval Research and the National Aeronautics and Space Administration under Grant N00014-67-A-0239-0021(NR-044-431) and Grant NGL-21-002-008, respectively.

Abstract

This report contains a description of the programming language, SIMPL-X, which is the base language for a family of programming languages that will be extensions to SIMPL-X and whose compilers will be written in SIMPL-X and its extensions. It is a transportable compiler-writing, systems language which was developed to provide a basis for the redefinition of the graph algorithmic language GRAAL.

SIMPL-X is a procedure-oriented, non-block structured language with an extensive set of operators, including arithmetic, relational, logical, bit manipulation, shift, indirect reference, address reference, and partword operators. It is designed for writing GRAAL programs that conform to the standards of structured programming and modular design and for efficiently expressing and implementing algorithms written in it. In addition, it appears to be a good language for modeling and certifying the correctness and equivalence of programs.

Preface

1. Introduction

2. Informal Definition of SIMPL-X

3. Syntax and Semantics of SIMPL-X

4. Some Examples

Preface

The primary motivation for the development of SIMPL-X derived from experience with the graph algorithmic language, GRAAL. A principal objective of GRAAL was to allow for the computational solution of applied graph problems involving a wide variety of graphs of different types and complexity, with as little degradation in the implementation as possible. In order to provide for this range of requirements, a modular structure was introduced into the design of GRAAL. However, only a limited aspect of this modularity was realizable in the first implementation of GRAAL. This was due mainly to its definition as an extension of ALGOL, and later FORTRAN, with the inherent restrictions in these algebraic languages which allow neither for the desired range of modularity nor for the flexibility in the data structures for representing different types of graphs. Extensions of these standard languages are cumbersome and this limits the possibilities for providing features such as the direct definition of flexible data structures, or the implementation of complex structures, such as hierarchical graphs, etc. For these reasons, SIMPL-X was developed to provide a new basis for the redefinition of GRAAL and to allow the further evolution of the GRAAL system.

The author would like to acknowledge, with thanks, many invaluable suggestions and recommendations by Dr. R. E. Noonan, Mr. A. J. Turner, and Dr. M. V. Zelkowitz, as well as partial support for this research received from the Office of Naval Research under Grant N00014-67-A-0239-0021(NR-044-431).

1. Introduction

SIMPL-X is the base language for a family of programming languages which will be extensions to SIMPL-X and whose compilers will be written in SIMPL-X and its extensions. The language is designed for writing programs that conform to the standards of structured programming and modular design and for efficiently expressing and implementing algorithms written in it.

Further design criteria for SIMPL-X were in part motivated by its intended use for the further development of the GRAAL system. Some of these criteria included:

1) Support for the construction of a variety of data structures. In this way a variety of graph types and their associated data structures could be easily implemented.
2) Modularity in design to provide flexibility of choice to the user. In this way GRAAL could provide the user with a flexible selection of data structures for representing a graph along with the corresponding graph operators.
3) Ease of extension. In this way the full graph language could grow out of the basic design.
4) Reasonable transportability. This would allow the GRAAL system to be implemented on a variety of computers.
5) Semantic modeling of the language. This would provide a formal definition for the language that would be useful as a tool for certifying the correctness of graph algorithms or the equivalence of such algorithms.

All of the above are actually reasonable criteria for the design of a transportable compiler-writing or systems language. Consideration of all these criteria led to the basic design of SIMPL-X. Several of the most salient design features are

> The main statement constructions are the assignment, while, if-then-else, case, and call statements. There is no go-to statement.
>
> Every program consists of a sequence of procedures which can access a set of global variables, parameters, or local variables.
>
> There are compound statement constructions but there are no block constructions other than procedures.
>
> Facilities for declaring external references and entry points are available.
>
> Procedures may be recursive if they are so declared. Functions do not have side effects.
>
> An extensive set of operations are permitted in an expression. There are arithmetic, relational, logical, bit manipulation, shift, indirect reference, address reference, and partword reference operators.

This last feature supports the first criteria which was the construction and manipulation of data structures. In particular, there is an operator that yields the address of a variable, an indirect address operator, several shift operators, several bit manipulation operators, and a partword operator along with the standard arithmetic, logical and relational operators.

In order to meet the objective of modularity, the language and its compiler were designed concurrently. The design of the compiler is horizontally and vertically modular. It is modular horizontally in that the interfaces are well-defined and the scanner, parser, and code genrator are written as separate units. The vertical design is also modular in that the compiler is written in SIMPL-X which forces all segments to be written as separate procedures. The parser is modular in that it is hierarchical. The grammar

is partitioned into subgrammars in which the start symbol of one subgrammar is a terminal symbol in one or more of the others. In this way a variety of separately compiled syntactic and semantic routines for representing various sublanguages could all exist in a library of routines and be called by the main parser at different points in the same program. Since these routines could represent different versions of similar constructs, this design would support the user's ability to choose at compile time the data structure for representing a graph and the corresponding graph operators.

Because of the modular design of the compiler and the strict structural design of the language, SIMPL-X is easily extendable.

In order to make the language as transportable as possible, the compiler for SIMPL-X is written in itself using a relatively machine-independent subset of the operators. A SNOBOL program exists that translates SIMPL-X into standard FORTRAN. In order to transport SIMPL-X onto another machine, a code generation module for the new machine must be written in SIMPL-X. Some minor modifications to the scanner, parser, and symbol table might also be necessary depending upon the word size of the new machine. The SNOBOL program can then be used to translate these SIMPL-X programs into FORTRAN. This FORTRAN version of the language can then be used as a bootstrap to get SIMPL-X running on the new machine. (Obviously, the appropriate system interface routines must also be written.) Since the graph language will be written in SIMPL-X, a compiler for the graph language would be available on the new machine. In fact, any programs, including any extensions to SIMPL-X, written in SIMPL-X, could be transported onto a new machine in the same way.

SIMPL-X has been modeled using several semantic models including the Vienna Definition Language and hierarchical graphs. These models were used to help in the design of the language so that the program structures would be amenable to program certification techniques. They were also used to help design a language that could be efficiently implemented and were used to test the final design of the actual implementation.

With regard to efficiency, the language is simple in design and structure. There are no complications such as block structure with procedures declarable at any level and no cumbersome features such as the ability to pass procedures as parameters so that keeping track of the environment for variables is a relatively straightforward task. Parameter passing is as efficient as possible; simple variables are treated as called by value and arrays are called by reference. The use of global variables is encouraged to minimize the need for parameter passing in procedures. Nonrecursive procedures are not burdened with the overhead of recursive procedures.

Work with SIMPL-X supports its recommendation as a transportable compiler-writing and systems language. The fact that it lends itself easily to compiler-writing is supported by the fact that the compiler for SIMPL-X was written in SIMPL-X for the Univac 1108 and was completed in two man months. A partial test of its transportability is that it is being used to cross compile code for the PDP 11. In fact, there are plans for using SIMPL-X in part as a systems language here at the University of Maryland for the development of an operating system for the PDP 11. Work is also presently underway to develop a translator-writing-system written in SIMPL-X.

Omitting the extended operator set, SIMPL-X is a useful educational tool for teaching programming. It encourages the student to write well-structured and modular programs. In addition, it appears to be a good language for modeling and certifying the correctness and equivalence of programs.

2. Informal Definition of SIMPL-X

This section presents an informal introduction to the language. It is meant to give the reader an overview of the basic components of the language. A full syntactic and semantic definition of the language is contained in the next section.

A. The Core of SIMPL-X

It is the core of the language which is recommended as an educational tool for teaching programming.

A1. Basic Data

The basic data type of the language is integer. An integer may be written as a constant or it may be represented by a variable identifier.

A constant may be represented in integer, binary, octal hexadecimal, or character string form.

A variable identifier is defined as any sequence of letters or digits beginning with a letter. Every integer variable must be declared. A global integer variable may be initialized to a constant value at compile time.

A2. Basic Operators

The basic operators act only on integer values. They are divided into five classes:

a) the arithmetic operators of addition (+), subtraction (-), multiplication (*), division (/), and unary minus (-). The arithmetic operators return the integer value which results from the operation.

b) the logical operators and (and), or (or), and not (not). The logical operators return a 1 if the result of the operation is true and a zero if the result of the operation is false. An individual operand is considered true if it is nonzero and false if it is zero.

c) the relational operators greater than (>), greater than or equal to (≥), equal to (=), less than or equal to (≤), less than (<), and not equal to (≠). The relational operators return a 1 if the relation is true and a 0 if the relation is false.

d) The bit operators bit and (A), bit or (V), bit exclusive or (X), and complement (C). The bit operators treat the (integer) operands as bit strings and return the bit string result (bit by bit) of the operation.

e) the shift operators right algebraic shift (ras), right logical shift (rls), left logical shift (lls), and left circular shift (lcs). The shift operators shift the bit string representation of the left operand the number of bits specified by the right operand. The algebraic shift extends the sign bit, the logical shifts are end-off with zero fill and the circular shift is end-around.

A3. Data Structures

The only data structure is a one-dimensional array. Arrays must be declared. The bounds for an array are static. The lower bound is always zero, and the upper bound is an integer constant specified in the declaration. Arrays may be initialized to any constant value. For referencing an array variable, the subscript may be any legal expression.

A4. Statement Structures

The choice of statement structures was motivated by the desire to promote structured programming. The basic statement structures of the language are

a) the assignment statement

<variable> := <expression>

b) the if-then-else statement

if <expression>
then <statement list>$_1$
{else <statement list>$_2$}
end

which executes <statement list>$_1$ if <expression> evaluates to a nonzero value, or <statement list>$_2$ if <expression> evaluates to zero. The else part of the statement is optional.

c) the while statement

while <expression> do
<statement list>
end

which executes <statement list> followed by the while statement if <expression> evaluates to a nonzero value. If <expression> evaluates to zero, control passes to the statement following the while statement.

d) the case statement

```
case <expression> of
    \n1\ <statement list>1
    \n2\ <statement list>2
          .
          .
          .
    \nm\ <statement list>m
    {else <statement list>m+1}
    end
```

which executes only <statement list>$_i$ if <expression> evaluates to n_i for some i = 1,2,...,m, or only <statement list>$_{m+1}$ if <expression> does not evaluate to n_i for any i = 1,2,...,m. The else part of the statement is optional.

e) the call statement

call <proc name>{(<argument list>)}

which invokes the execution of the procedure <proc name>. The actual parameters (if there are any), may be expressions or array names. Expressions are passed by value and arrays are passed by reference.

A5. Program Structure

The choice of program structure was motivated by the desire to promote the modular design of programs. A complete SIMPL-X program consists of a set of global declarations, followed by a nonempty sequence of segment definitions, followed by the symbol start, followed by a segment name. The program begins execution by calling the named segment.

A6. Segment Definitions

A segment is either a procedure or a function. A procedure definition takes the form:

{rec} proc <procname> {(<parameter list>)}
{<local declaration list>}
<statement list>
return

where <procname> is the name of a procedure (an identifier), <parameter-list> is an optional parameter list which consists of typed formal parameters, and <local declaration list> is a possibly empty list of local integer variable or array declarations that may not be initialized. A procedure is not recursive unless the symbol rec is used before the symbol proc. The actual parameters corresponding to the simple variables are passed by value; those corresponding to the array variables are passed by reference. Global variables may appear anywhere in the statement sequence. In fact, procedures usually have an effect on the program by altering global variables.

A function definition takes the form:

int func <funcname> {(<parameterlist>)}
{<local declaration list>}
<statement list>
return (<expression>)

where <funcname> is the name of the function (an identifier), and <parameterlist> and <local declaration list> are as defined above. <Expression> represents the value returned by the function. Functions may not be recursive and may have

no side effects, i.e., they may reference global variables but they may not alter them. Note that this means arrays passed as arguments may not be altered.

A7. External Interfaces

Externals. To facilitate modular structure in the entire system and permit efficient interface between separately compiled program elements, external references and entry points may be defined.

A program may make any of its segments or data accessible to another program (compiled separately) by declaring those segments or data as entry points. A program may access a segment or data entry point of another program by declaring that segment or data item as an external reference. A program may be designated as nonexecutable, i.e., may only be used by another program if the segment name following the symbol start is omitted.

I/O I/O commands available in the language include the basic forms of stream and record I/O, plus specialized commands for systems use.

Comments

A comment is any string of characters between /* and */ and may appear in the program wherever a blank may occur.

B. Extensions

Several of the following features have already been incorporated into the language; others are in the process of being tested out and added. A full description of these extensions is given in part two of the next section.

B1. Additional Statements

In order to permit an abnormal exit from a while loop, the while statement may be labeled and an exit statement may be used within the while loop. The exit statement takes the form:

exit {(<label>)}

which causes control to pass to the statement following the while statement with the specified label. The exit statement may be nested any number of levels inside the loop. If the label part is not specified, control passes to the statement following the innermost while loop containing the exit statement.

The while statement is labeled as follows:

\<exit designator>\ while <expression> do <statement list> end

In order to perform an abnormal exit from anywhere in a segment, a return statement may be used. In a procedure, it takes the form

return

and in a function it takes the form

return (<expression>)

where <expression> represents the expression to be returned by the function. More than one return statement may appear in the body of a segment.

B2. Additional Operators

There are three additional operators in the extension of the language. They are

a) the address operator (↑) which is a unary operator that returns as its value the location of the operand to its right.

b) The indirect address operator (↓) which is a unary operator that returns as its value the contents of the location specified by the low order bits of the operand to its right.

Note: The operator ↑ is highly machine-dependent. In order to make this type of operation machine-independent, a syntactic modification is planned.

c) the partword operator ([<partdesignation>]) which is a unary operator that returns as its value the bit string (as specified by the <partdesignation>) contained in the operand to its left.

3. Syntax and Semantics of SIMPL-X

This section contains the syntactic and semantic definition of the language. It is divided into three parts. Section 1 contains the description of the core of the language. The second section contains the extensions to the basic language. The definitions of these extensions may be altered by further experience with the language. Section 3 contains a description of the input/output commands.

1. Basic Language

The structure {...} means that the enclosed structure is optional. {X} Y is equivalent to the BNF notation X Y|Y.

1.1 Program

1.1.1 Syntax

<program> ::=

{<declaration list>} <segment list> start {<identifier>}

1.1.2 Semantics

A program consists of a set of declarations which are global to the whole program, followed by a series of segment definitions which are either procedure or function declarations followed by the symbol start and the name of the segment which will act as the main program.

If no identifier appears after start, then the program cannot be executed on a stand-alone basis.

1.1.3 Example

```
int X,Y
proc A
     X := 1
     return
proc B
     call A
     Y := X
     return
start B
```

1.2 Declarations

1.2.1 Syntax

<declaration list> ::= {<declaration list>} <declaration>

<declaration> ::= <integer declaration> | <array declaration> |
<external declaration>

<integer declaration> ::= {entry} int<integer declaration list>

<array declaration> ::= {entry} {int} array <array declaration list>

<external declaration> ::= ext int <identifier list> |
ext {int} array <identifier list> |
ext proc <identifier> (<type list>) |
ext int func <identifier> (<type list>)

<integer declaration list> ::= {<integer declaration list>,}
<integer declaration item>

<integer declaration item> ::= <identifier> | <identifier> = <signed constant>

<array declaration list> ::= {<array declaration list>,}
<array declaration item>

<array declaration item> ::= <identifier> (<constant>)

<identifier> (<constant>) = (<array initialization list>)

<array initialization list> ::= {<array initialization list>,}

<array initialization item>

<array initialization item> ::= <signed constant> |

<signed constant> (<constant>)

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

<signed constant> ::= <constant> | - <constant>

<type list> ::= {<type list>,} <type>

<type> ::= int | {int} array

1.2.2 Semantics

The declaration list consists of all the integer variables and arrays that are global to the program along with all of the external procedures, functions, and variables used in the program.

All arrays begin with subscript 0.

Integer variables and arrays may be initialized with their values assigned at compile time. In order to facilitate the initializing of several elements of an array with the same value, any initialization value may be followed by a constant in parentheses, implying that initial value should be assigned to the next consecutive sequence of elements whose length is defined by the constant in parentheses.

External declarations (those preceded by ext) refer to procedures, functions, and variables that are defined or declared in another program compiled separately from the program in which the external declaration resides. External procedure and function declarations must be followed by

a type list which lists the types of the formal parameters in the order they appear in the parameter list of their segment definition. The entry integer and array declarations (entry int and entry int array) refer to integers and arrays that may be referenced by separately compiled (external) programs.

1.2.3 Example

```
int X, Y, Z = 0
int A = 1, B = 2, C = 3
array V(2) = (1,2,3), U(99) = (-1, 2, -3, 0(97))
ext proc scan (int,array)
ext int func lookup
entry int dog, cat = 17
```

1.3 Program Segments

1.3.1 Syntax

<segment list> ::= {<segment list>} <segment definition>

<segment definition> ::= <proc definition> | <func definition>

<proc definition> ::= <proc heading> <segment body> return

<func definition> ::= <func heading> <segment body> return (<expr>)

<proc heading> ::= {entry} {rec} proc <identifier> {(<parameter list>)}

<segment body> ::= {<local declaration list>} <statement list>

<func heading> ::= {entry} int func <identifier> {(<parameter list>)}

<parameter list> ::= {<parameter list>,} <parameter>

<parameter> ::= int <identifier> | {int} array <identifier>

<local declaration list> ::= {<local declaration list>,} <local declaration>

<local declaration> ::= <local int declaration> |

<local array declaration> |

<external declaration>

<local int declaration> ::= int <local int declaration list>

<local array declaration> ::= {int} array <local array declaration list>

<local int declaration list> ::= {<local int declaration list>,}
    <identifier>

<local array declaration list> ::= {<local array declaration list>,}
    <identifier> (<constant>)

1.3.2 Semantics

A SIMPL-X program consists of a sequence of procedures and functions, called program segments.

1.3.2.1 Procedure

A procedure is a sub-program composed of a procedure heading followed by a segment body. The procedure heading consists of the symbol proc followed by the name of the procedure optionally followed by a parameter list. The parameter list is simply a sequence of typed variables (integers or integer arrays). The set of formal parameters in the formal parameter list must agree in type and number with the actual parameters in the actual parameter list of the call statement. All expressions are called by value, and all arrays are called by reference.

The segment body consists of an optional set of locally declared variables followed by a statement list. The local declarations take on the same form as the global declarations, except that they may not be initialized at compile time. Global variables may appear anywhere in the statement list. (In fact, procedures usually have an effect on a program by altering the contents of global variables.) Each segment body for a procedure terminates with the symbol return, which returns control to the calling program at the statement following the program call.

Procedures may be recursive only if the symbol rec is included in the <proc heading>.

1.3.2.2 Functions

A function is composed of a function heading followed by a segment body. The function heading consists of the symbol func which is preceded by the type symbol int, and followed by the name of the function, optionally followed by a parameter list.

The segment body for the function must be terminated by the symbol return followed by the expression, enclosed in parentheses, whose value is to be returned as the result of the function.

Functions may not be recursive, and may have no side effects (i.e., they may reference global variables, but may not alter them.) Thus, arrays passed as arguments may not be altered.

External identifiers declared locally by a procedure or function are treated as global variables during execution of the procedure or function. Identifiers may be declared locally to override a global identifier with the same name.

1.3.3 Example

```
proc add (int X, int Y)
     Z := X+Y+1      /*Z is a global*/
     return
int func sum(int X, int Y)
     int Z
     Z := X+Y
     return (Z+1)
```

1.4 Statements

1.4.1 Syntax

<statement list> ::= {<statement list>,} <statement>

<statement> ::= <assign stmt> | <if stmt> | <while stmt> |
<case stmt> | <call stmt>

<assign stmt> ::= <variable> := <expr>

<if stmt> ::= if <expr> then <then clause> {else <else clause>} end

<then clause> ::= <statement list>

<else clause> ::= <statement list>

<while stmt> ::= while <expr> do <while clause> end

<while clause> ::= <statement list>

<case stmt> ::= case <expr> of <case list> {else <else clause>} end

<case list> ::= {<case list>,} <case form>

<case form> ::= <case designator> <statement list>

<case designator> ::= {<case designator>} \<integer>\

<call stmt> ::= call <identifier> {(<actual parameter list>)}

<actual parameter list> ::= {<actual parameter list>,} <actual parameter>

<actual parameter> ::= <expr> | <array identifier>

1.4.2 Semantics

A statement list is a sequence of statements.

1.4.2.1 Assignment Statement

Assignment statements serve for assigning the value of an expression to a variable. The order of evaluation of the elements of an assignment statement is unspecified.

1.4.2.2 If Statement

The if statement causes certain statements to be executed or skipped depending upon the values of the specified expression. If the specified expression evaluates to a nonzero value at runtime, then the then clause is executed, and the else clause, if any, is skipped. If the specified expression evaluates to zero at run time then the then clause is skipped, and the else clause, if any, is executed.

1.4.2.3 While Statement

The while statement permits the repeated execution of a set of statements depending upon the value of the specified expression. When control passes to the while statement, the expression is evaluated. If the expression evaluates to a nonzero value, then the while clause is executed. After the while clause, control passes to the re-evaluation of the expression, and the process is repeated. If the expression evaluates to zero, then the while clause is skipped and control passes to the statement following the symbol end of the while statement.

1.4.2.4 Case Statement

The purpose of the case statement is to allow the execution of only one particular set of statements from among many sets of statements depending upon the evaluation of an expression which yields an index into the sets of statements. Each set of statements has one or more integer values associated with it. If the specified expression evaluates at runtime to an integer which is associated with one of the statements, then that statement set is executed and all other statement sets are skipped. Control then

passes to the statement following the symbol end associated with the case statement. If the specified expression does not evaluate to any integer associated with one of the statement sets, all of the statement sets are skipped, and the else clause, if any, is executed. Control then passes to the statement following the case statement.

1.4.2.5 Call Statement

The call statement is used to invoke the execution of a procedure body. There may or may not be an actual parameter list. If there is an actual parameter list, it must have the same number of entries as the formal parameter list of the procedure declaration heading of the procedure being called. The correspondence is obtained by taking the entries in those two lists in the same order. The entries in the actual parameter list may be expressions or array identifiers. Expressions are passed by value; arrays are passed by reference. The corresponding actual and formal parameters must agree in type, i.e., if the formal parameter is int or array, the corresponding actual parameter must be int or array, respectively.

1.4.3 Examples

```
if X
     then
          Y := X
     else
          Y := Z
     end
```

```
while X>Y do
     A(X) := X
     X := X-1
     if X=Y
          then
               call proc1(X)
          end
     end

case X+Y-7 of
   \3\
          X := 2
   \5\\9\
          X := 7
   \7\
          X := 19
    else
          X := 22
    end
```

1.5 Expressions

1.5.1 Syntax

<expr> ::= {<expr> or} <logical product>

<logical product> ::= {<logical product> and} <relation>

<relation> ::= {<relation> <relational op>} <simple expr>

<simple expr> ::= {<simple expr> <add op>} <term>

<term> ::= {<term> <mult op>} <bit sum>

<bit sum> ::= {<bit sum> <bit or op>} <bit product>

<bit product> ::= {<bit product> A} <shift>

<shift> ::= {<shift> <shift op>} <factor>

<factor> ::= {<unary op>} <primary>

<primary> ::= <constant> | <variable> | <function designator> | (<expr>)

<relational op> ::= = | ≠ | > | < | ≥ | ≤

<add op> ::= + | -

<mult op> ::= * | /

<bit or op> ::= V | X

<shift op> ::= ras | rls | lls | lcs

<unary op> ::= - | not | C

<function designator> ::= <identifier>{(<actual parameter list>)}

<variable> ::= <identifier> {(<expr>)}

<constant> ::= <integer> | <binary> | <octal> | <hexadecimal> |
<character string>

<integer> ::= {<integer>} <digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<binary> ::= B' <binary form> {<trailing zeros>}'

<binary form> ::= {<binary form>} <binary character>

<binary character> ::= 0 | 1

<octal> ::= O'<octal form>{<trailing zeros>}'

<octal form> ::= {<octal form>} <octal character>

<octal character> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<hexadecimal> ::= H'<hexadecimal form> {<trailing zeros>}'

<hexadecimal form> ::= {<hexadecimal form>} <hex character>

<hex character> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
A | B | C | D | E | F

<character string> ::= '<character form>'

<character form> ::= {<character form>} <character>

<trailing zeros> ::= Z<integer>

<identifier> ::= {<identifier>} <letter> | <identifier> <digit>

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
N | O | P | Q | R | S | T | U | V | W | X | Y | Z

1.5.2 Semantics

An expression is a rule for computing a numerical value. A primary is either a constant, variable, function, or an expression enclosed in parentheses. The operators have the following meanings associated with them. Note that several of the operations are machine dependent.

1.5.2.1 Unary Minus (-)

Returns the negative of the primary it precedes.

1.5.2.2 Logical Not (not)

Returns a 1 if the value of the primary it precedes is zero, and returns a 0 if the value of the primary is nonzero.

1.5.2.3 Complement (C)

Returns the complement of the bits in the primary it precedes.

1.5.2.4 Right Algebraic Shift (ras)

The <shift> is shifted right by the number of bits specified by the <factor>, with sign bit extension.

1.5.2.5 Other Shifts

Right logical shift (rls), left logical shift (lls), and left circular shift (lcs) function similarly to ras. A logical shift is a zero-fill shift.

1.5.2.6 Bit And (A)

Returns the result of bit by bit anding the bit product with the shift.

1.5.2.7 Bit Or (V)

Returns the bit by bit oring of the bit sum and bit product.

1.5.2.8 Bit Exclusive Or (X)

Returns the bit by bit exclusive or of the bit sum and bit product.

1.5.2.9 Multiplication (*)

Returns the product of the term and bit sum.

1.5.2.10 Division (/)

Returns the quotient of the term and the bit sum.

1.5.2.11 Addition (+)

Returns the sum of the simple expression and the term.

1.5.2.12 Subtraction (-)

Returns the result of subtracting the term from the simple expression.

1.5.2.13 Equal (=)

Returns a 1 if the relation and the simple expression are equal, and 0 otherwise.

1.5.2.14 Other Relations

Not equal ($\neq$), greater than (>), less than (<), greater than or equal to ($\geq$), and less than or equal to ($\leq$) are similar to equal (=).

1.5.2.15 Logical And (and)

Returns a 1 if the logical product and the relation are non-zero and a 0 otherwise.

1.5.2.16 Logical Or (or)

Returns a 1 if either or both of the expression and logical product are nonzero, and a 0 otherwise.

1.5.2.17 Precedence

The precedence of the operators is as follows:

-, not, C

ras, rls, lls, lcs

A

V, X

*, /

+, -

=, ≠, <, >, ≥, ≤

and

or

1.5.2.18 Character String

A character string constant normally occupies one machine word. The constant is padded on the right with blanks or truncated on the right as needed to fit it into one word. Note that this is machine dependent to the extent of both word size and internal character representation.

The exception to the one word per character string rule occurs in the initialization of an array. In this case as many consecutive array elements

as needed are used to contain the character string (provided enough elements exist in the array), padded on the right with blanks as needed to fill an integral number of words.

1.5.3 Examples

```
X+Y*Z/U-2
A lcs 6 A mask1 V mask2 X mask3
A and B<C
-.not X rls 1
int array cat (9)=('char string 1','str2') /* initializes cat(0) to
     cat([13/X] + [4/X]) where X = number of characters per word and [Y]
     denotes the least integer greater than or equal to Y. */
```

1.6 Blanks

One or more blanks may appear anywhere except within a symbol, identifier, or operand.

1.7 Comments

Comments are any string of characters between /* and */. A comment has no effect on the program and may appear anywhere that blanks may appear.

2. Extensions

2.1 Additional Statements

2.1.1 Syntax

add to <statement> :

<statement> ::= ... |\<exit designator>\ <while statement> |
<exit statement> | <return statement>

<exit statement> ::= exit {(<exit designator>)}

<exit designator> ::= <identifier>

<return statement> ::= return {(<expr>)}

2.1.2 Semantics

2.1.2.1 Exit Statement

The exit statement permits an abnormal exit from a while loop. In the form exit, without an exit designator, control passes to the statement following the symbol end associated with the innermost while loop containing the exit statement.

If the exit statement appears in the form exit (<exit designator>), control passes to the statement following the symbol end associated with the while loop which is labeled by that exit designator. The exit statement must be a statement in the while clause of the while loop that it is exiting. If it is not, or if no such designator exists, an error occurs.

2.1.2.2 Return Statement

A return statement must appear somewhere in the segment body of a procedure definition and function definition. In a procedure it appears in the form return, while in a function it appears in the form return (<expr>).

Note that no return is required at the end of a segment but is assumed if needed. More than one return statement may appear in the statement list of a segment.

2.2 Additional Operators

2.2.1 Syntax

<assign stmt> ::= <left variable> := <expr>

<factor> ::= <part primary> | <unary op> <part primary>

<part primary> ::= <primary> | <primary> <part designator>

<primary> ::= <constant> | <extended variable>|

<function designator> | (<expr>) |

↓<constant> | ↓(<expr>) |

↓<function designator>

<extended variable> ::= <variable> | ↓<variable> | ↑<variable>

<part designator> ::= [<first bit>, <bit count>] |

[<partword identifier>]

<first bit> ::= <integer>

<bit count> ::= <integer>

<part word identifier> ::= <identifier>

<left variable> ::= <variable> | <part var> | ↓<variable> | ↓<part var>

<part var> ::= <variable> <part designator>

2.2.2 Semantics

Three new operators have been introduced into an expression. The definitions of these are as follows:

2.2.2.1 ↑

This operator returns the location of the variable that it precedes. It is effectively an immediate operator.

2.2.2.2 ↓

This operator returns as its value the contents of the location specified by the low order bits of the variable, constant, or expression that it precedes. It is in effect an indirect addressing operator.

2.2.2.3 Part Variable

This operator returns as its value the bit string contained in the partword specified by the fields F1, F2, or F.

If it is of the form [F1,F2], then the part of the word is designed as F2 bits wide and starting with bit F1. Bits are numbered from left to right starting with 0.

If it is of the form [F], then F has been specified in the implementation as some specific designator for some specific partword.

2.2.2.4 Precedence

The precedence of the new operators is indicated by

part variable operators
↑, ↓
.
. previous operators
.

That is, the new operators have higher precedence than any of the initially defined operators, and the partword operators have higher precedence than the indirect and immediate operators.

2.2.3 Example

A := ↑B+I assigns to A the address of the location I locations after B.

X := B[5,3] assigns bits 5, 6, and 7 to the rightmost positions of X with zero fill.

Y[0,2] := C assigns the two low-order bits of C to the two high-order bits of Y with the remaining bits of Y unchanged.

3. I/O

3.1 Read Card Input File

3.1.1 read (<inlist>)

The syntax is explained by

<inlist> ::= {<inlist>,} <inlist item>

<inlist item> ::= <variable> | <skip>

<skip> ::= skip | skip0 | skip1 | ... | skip9

This is essentially PL/I stream input. The variables read are separated by commas or blanks. Additional blanks between items are not significant. The skipN subcommand causes N records to be skipped. skip is the same as skip1. skip0 causes realignment at the beginning (column 1) of the current card image. Cards are assumed to be end-to-end so that a variable may cross a card boundary. read(skip,X) will read the first variable on each card.

3.1.2 readc (<inlistc>)

<inlistc> ::= {<skiplist>,} <variable> | <skiplist>

<skiplist> ::= {<skiplist>,} <skip>

This causes the character image of the card to be placed in successive words beginning at <variable>. The image is padded with blanks to the number of words needed to contain one card image.

The <skip> subcommands cause alignment at the beginning of the appropriate card image. skip0 has no effect if alignment is already at the beginning of a card. Successive readc(skip,<variable>...) commands will read every other card.

3.1.3 eoi

eoi is a function call that returns a 1 if no more data can be obtained by read, and a 0 otherwise.

3.1.4 eoic

eoic is similar to eoi, but applied to readc, rather than to read.

3.2 Printed Output

3.2.1 write (<outlist>)

<outlist> ::= {<outlist>,} <outlist item>

<outlist item> ::= <expression> | <skip> | eject

This functions similarly to PL/I stream output. It is essentially a free-form output with the values of expressions printed at installation defined tab settings.

The subcommands <skip> and eject apply to line and page alignment, respectively.

3.2.2 writeℓ (<outlistℓ>)

<outlistℓ> ::= {<outcontrol list>,} <variable> {,<variable>}

<outcontrol list> ::= {<outcontrol list>,} <skip> |

{<outcontrol list>,} eject

Here, the first <variable> specifies the beginning of a character line to be printed, and the second gives the length, in words, of the line. If the second <variable> is omitted, the installation defined line size is assumed.

3.3 Other Files

3.3.1 readf (<id>,<variable list>)

This reads one word from file <id> into each successive <variable> in <variable list>. The file name <id> may be used to access an external file. A file is viewed as one continuous stream of words.

3.3.2 writef (<id>,<variable list>)

This operation is the inverse of readf.

3.3.3 eoif (<id>)

This functions in a manner similar to that of eoi and eoic.

3.3.4 endfile (<id>), rewind (<id>), backspace (<id>)

These function similarly to the analogous Fortran commands.

3.3.5 File Declaration

A file <id> must be declared as a global or external identifier. The syntax is specified by

<file decl> ::= {ext} file <identifier list> | {entry} file <identifier list>

3.4 Machine Code (Relocatable) Output

These commands are highly installation dependent. However, any local implementation should be reasonably close to the examples given below for the Univac 1108.

3.4.1 wrtxt

The syntax is given by

wrtxt (<textword>,<ℓc1>,<offset>{,<ℓc2>{,<seqno>}})

This causes the word at 'textword' to be written into the relocatable output file. The location of this word in the relocatable output is at location counter 'ℓc1', offset 'offset'. The address bits are relocated with respect to location counter 'ℓc2', if there are four parameters. If there are five parameters, then relocation is with respect to external reference number 'seqno'. Relocation is positive or negative depending on whether ℓc2 is positive or negative (+0 or-0 for external reference).

3.4.2 wrend (<ℓc>,<offset>)

This specifies the end of the relocatable output, and the start address of the generated program. wrend (0) denotes a nonexecutable relocatable element.

3.4.3 wrtbl (<preamble>,<length>)

This writes out the information necessary for systems interface, such as the entry point and external reference tables. This information must be formatted by the user.

3.4.4 Runtime Options

Again, this command is installation dependent and may be either a function or procedure. The purpose of the command is to make available to the program any options specified at runtime via normal system interface. The syntax on the Univac 1108 is given by

OPTIONS

which is a function returning the Exec 8 options word.

4. Some Examples

This section contains three examples to demonstrate the use of SIMPL-X as a programming language.

Example 1

```
/* This is a procedure which may be called by a separately compiled program.
   The procedure sorts the  n  element array  a  using a bubble sort tech-
   nique. */
entry proc bsort (int array a, int n)
     int i, /* array index */
          temp, /* storage temporary */
          switch  /* switch for re-execution */
     switch := 1
     while switch do
          switch := 0
          n := n-1
          i := 0
          while i < n do
               if a(i) > a(i+1)
                    then switch := 1
                         temp := a(i)
                         a(i) := a(i+1)
                         a(i+1) := temp
                         i := i+1
                    end
               end
          end
     return
start
```

Example 2

```
/* This is a recursive descent algorithm for parsing strings generated by
   the grammar
                <assign> ::= var := <expr>;
                <expr> ::= <term> | <expr> + <term>
                <term> ::= <factor> | <term> * <factor>
                <factor> ::= var | const | (<expr>)
   The terminal symbols are tokenized in the form (class, address) by the
   external procedure scan according to the following table:
```

type	class	name
var	'i'	actual identifier
const	'n'	actual constant
any other symbol	the symbol	0

```
   The external procedure output can be defined to output the string in some
   semantic form. */
ext proc scan /* reads the next token placing the class into the variable
                 'symbol' and the name into the variable 'value' */
ext proc output (int) /* outputs the operator or operand as a postfix form
                         stack */
entry int symbol, /* contains the class of the last token created by the
                     scanner */
      value, /* contains the actual identifier or constant if the token is
                a variable or constant, otherwise it is set to zero */
      error, /* set to 1 if improper statement terminator,
                      2 if improper assign operator,
                      3 if no right parenthesis after an expression,
                      4 if improper factor */
```

```
proc assign
     call scan
     call scan
     if symbol = ':='
          then call scan
               call expr
               call output (':=')
               if symbol ≠ ';'
                    then error := 1
                    end
          else error := 2
          end
     return

rec proc expr
     call term
     while symbol = '+' do
          call scan
          call term
          call output ('+')
          end
     return

rec proc term
     call factor
     while symbol = '*' do
          call scan
          call factor
          call output ('*')
          end
```

```
rec proc factor
     case symbol of
          \'i'\\'n'\ call output (symbol)
                     call scan
          \'('\      call scan
                     call expr
                     if symbol ≠ ')'
                          then error := 3
                          end
                     call scan
          else       error := 4
          end
     return


start assign
```

Example 3

```
/* This is a program which symbolically differentiates an expression  f
   involving a "symbol" (i.e., variable) x, real constants, and the operators
   + and * and produces an expression which is the derivative of  f  with
   respect to the variable x.  The expression to be differentiated is stored
   in tree form with each node three words long.  The contents of the first
   word is the type of operation and the contents of words 2 and 3 are point-
   ers to the left and right operands, if they exist.  The set of possible
   nodes are
     word 1                    word 2                    word 3
        +              ptr to left operand       ptr to right operand
        .              ptr to left operand       ptr to right operand
        c              ptr to the constant                  0
        v              ptr to the symbol x                  0            */
int free, /* contains a pointer to a list of free storage nodes */
    value, /* contains a pointer to the present node */
    error = 0, /* is a switch which results in a 1 if there is no more free
                  storage, and a 2 if a derivative of an unknown operation
                  is attempted. */
    zero = 0, one = 1, two = 2
      .
      .
      .
    (plus any other global variables in the program)
      .
      .
      .
proc next /* sets the next free node into value */
    if free = 0
        then write ('no more free storage')
             value := 0
        else value := free
             free := ↓free
        end
    return
```

```
proc setnode (int op, int op1, int op2, int node)
     /* sets up a new node with the trip (op, op1, op2) and returns the
        address of the node */
     call next
     if value ≠ 0
          then ↓value := op
               ↓(value +1) := op1
               ↓(value +2) := op2
               ↓node := value
          else error := 1
               ↓node := ↑one
          end
     return

proc deriv (int f, int result)
     /* f  is the address of the root node of the expression to be differen-
        tiated, result receives the address of the resulting differentiated
        tree */
     int r1, r2, n1, n2
     case ↓f of
          \'c'\↓result :=↑ zero
          \'v'\↓result :=↑ one
          \'+'\ call deriv (↓(f+1),↑r1)
                call deriv (↓f+2),↑r2)
                call setnode ('+', r1, r2, result)
          \'.'\ call deriv (↓(f+1),↑r1)
                call deriv (↓(f+2),↑r2)
                call setnode ('.',↓(f+1),r2,↑n1)
                call setnode ('.',r1,↓(f+2),↑n2)
                call setnode ('+',n1,n2, result)
```

```
     else write ('no such operator')
          call setnode (0, 0, 0, result)
          error := 2
     end
return
  .
  .
  .
```

(plus any other segments such as those that read and write the expressions)

```
  .
  .
  .
```

Bibliography

1. Basili, V. R., Mesztenyi, C. K., and Rheinboldt, W. C., "FGRAAL, Fortran-Extended Graph Algorithmic Language", University of Maryland, Computer Science Center, Technical Report TR-179, 1972.

2. Dijkstra, E. W., Letter to the Editor, CACM, March-August, 1968.

3. Dijkstra, E. W., "Notes on Structured Programming", Technische Hogeschool Eindhoven (THE), 1969.

4. Good, D. I., and Ragland, L. C., "NUCLEUS, A Language for Provable Programs", The University of Texas at Austin, 1972.

5. Lucas, P., Lauer, P., and Stigleitner, H., "Method and Notation for the Formal Definition of Programming Languages", IBM Laboratory Vienna, Technical Report TR 25.087, 1968.

6. Mesztenyi, C. K., Breitenlohner, and Yeh, J. C., "FGRAAL, Technical Documentation", University of Maryland, Computer Science Center, Technical Report TR-200, 1972.

7. Mills, H. D., "Mathematical Foundations for Structured Programming", IBM Federal Systems Division, Gaithersburg, Md., 1972.

8. Pratt, T. W., "A Hierarchical Graph Model of the Semantics of Programs", SJCC 34, 1969, 813-825.

9. Rheinboldt, W. C., Basili, V. R., and Mesztenyi, C. K., "GRAAL, A Graph Algorithmic Language", in Sparse Matrices and Their Applications, D. J. Rose and R. A. Willoughby, Editors, Plenum Publishing Corp., New York, 1972, 167-176.

10. Rheinboldt, W. C., Basili, V. R., and Mesztenyi, C. K., "On a Programming Language for Graph Algorithms", BIT 12, 1972, 220-241.

11. Wulf, W. A., Russell, D. B., and Habermann, A. W., "BLISS, A Language for Systems Programming", CACM 14, 1971, 780-790.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

Computer Science Center
University of Maryland

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE

SIMPL-X, A Language for Writing Structured Programs

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Technical Report

5. AUTHOR(S) *(First name, middle initial, last name)*

Victor R. Basili

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
January 1973	44	11

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
NGL-21-002-008 and N00014-67-A-0239-0021	Technical Report TR-223
b. PROJECT NO. NR-044-431	
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Mathematics Program Office of Naval Research Arlington, Virginia 22217

13. ABSTRACT

This report contains a description of the programming language, SIMPL-X, which is the base language for a family of programming languages that will be extensions to SIMPL-X and whose compilers will be written in SIMPL-X and its extensions. It is a transportable compiler-writing, systems language which was developed to provide a basis for the redefinition of the graph algorithmic language GRAAL.

SIMPL-X is a procedure-oriented, non-block structured language with an extensive set of operators, including arithmetic, relational, logical, bit manipulation, shift, indirect reference, address reference, and part-word operators. It is designed for writing GRAAL programs that conform to the standards of structured programming and modular design and for efficiently expressing and implementing algorithms written in it. In addition, it appears to be a good language for modeling and certifying the correctness and equivalence of programs.

DD FORM 1 NOV 65 1473 UNCLASSIFIED

S/N 0101-807-6801

UNCLASSIFIED
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
programming language systems language compiler-writing structured programming modular design						