

SAMSO-TR-67-23

1

SPECIFICATION FOR
SPACE PROGRAMMING LANGUAGE (SPL)

Prepared By:

L. J. Carey
A. E. Kroger

System Development Corporation
Santa Monica, California 90406

August 1967

Prepared For:

SPACE AND MISSILE SYSTEMS ORGANIZATION
AIR FORCE SYSTEMS COMMAND
AIR FORCE UNIT POST OFFICE
LOS ANGELES, CALIFORNIA 90045

This document has been approved for public release and sale; its distribution is unlimited.

SAMSO-TR-67-23

SPECIFICATION FOR
SPACE PROGRAMMING LANGUAGE (SPL)

Prepared By:

L. J. Carey
A. E. Kroger

System Development Corporation
Santa Monica, California 90406

August 1967

Prepared For:

SPACE AND MISSILE SYSTEMS ORGANIZATION
AIR FORCE SYSTEMS COMMAND
AIR FORCE UNIT POST OFFICE
LOS ANGELES, CALIFORNIA 90045

This document has been approved for public release and sale; its distribution is unlimited.

FOREWORD

This is a technical report defining a common Space Programming Language (SPL). It was produced by the System Development Corporation during the contract period from February 1967 through August 1967. This work was performed under Contract Number FO 4695-67-C-0096. Also produced under this contract was a document entitled Compiler Requirements for Space Programming Language (SPL), SAMSO TR-67-3.

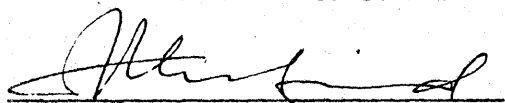
Based on a study of spaceborne software, SDC previously produced a four volume report, SSD TR-67-11. Volume III of that series contains an initial description of SPL in English prose form.

This report specifies the SPL language including two extensions for space computer programming applications.

The personnel involved were:

- a. Air Force Project Officers
Roger B. Engelbach
Lieutenant, USAF

Michael A. Ikezawa
Major, USAF
- b. Air Force Project Consultant
Dr. Walter A. Sturm
Aerospace Corporation
- c. SDC Project Manager
Gerard A. Hirschfield
- d. SDC Technical Staff
L. J. Carey, Project Head
A. E. Kroger
C. J. Shaw

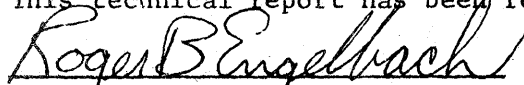


G. A. Hirschfield, Manager
Advanced Space and Range Projects



Levi J. Carey, Project Head
Space Programming Language

This technical report has been reviewed and is approved.



Roger B. Engelbach, Lieutenant
USAF Project Officer

ABSTRACT

This document contains a complete specification of the Space Programming Language (SPL) in Backus-Naur form. A description of Basic SPL and an extension is given. SPL is a space application language with a large array of capabilities. It is further an extendable language with punctuation rules and vocabulary designed for ease of learning and programming.

TABLE OF CONTENTS

	<u>Page</u>
1. INTRODUCTION	1
1.1 Intended Use of the Specification	1
1.2 Background	1
1.3 Language Requirements	2
1.4 SPL Recommendation	3
1.5 Significant Features of SPL	5
1.5.1 Basic SPL	5
1.5.2 SPL Extension	9
1.6 Notation and Metalanguage	10
1.6.1 Notation	10
1.6.2 Metalanguage	11
2. ALPHABET, VOCABULARY, AND PROGRAM STRUCTURE	13
2.1 Alphabet and Symbols	13
2.2 Names	14
2.3 Primitives	15
2.3.1 Delimiters	15
2.3.2 Operators	15
2.3.3 Descriptors	16
2.4 Constants	17
2.4.1 Numeric Constants	17
2.4.2 Textual Constants	18
2.4.3 Pointer Constants	19
2.4.4 Boolean Constants	19
2.5 Statement Structure	20
2.6 Comments	20
2.7 Program Structure	21

CONTENTS

	<u>Page</u>
3. DATA DEFINITION	22
3.1 Item Declarations	22
3.2 Array Declarations	25
3.3 Group Declarations	26
3.4 Mode Declarations	27
3.5 Storage Declarations	28
3.6 Variables	30
3.6.1 Subscripted Variables	32
3.6.2 Catenated Variables	36
3.6.3 Conditional Variables	37
3.6.4 Hardware Operands	37
3.7 Compool Declaration	38
4. STATEMENTS	39
4.1 Assignment Statements	39
4.1.1 Formulas	42
4.1.2 Direct Code	53
4.2 Control Statements	54
4.2.1 Transfer Statements	54
4.2.2 Repeated Statements	55
4.2.3 Conditional Statements	56
4.2.4 Parallel Statements	57
4.2.5 Stop Statements	58
4.2.6 Delayed Statements	59
4.2.7 Chronic Statements	60
4.3 Input-Output and Files	61
4.3.1 File Declarations	63
4.3.2 Assign Declaration	65

CONTENTS

	<u>Page</u>	
4.3.3	Opening and Closing Files	65
4.3.4	Testing and Positioning Files	67
4.3.5	Reading and Writing Records	68
5.	PROCEDURES AND FUNCTIONS	70
5.1	Procedure Declaration	70
5.1.1	Procedure Call Statements	72
5.1.2	Entry and Exit Statements	73
5.2	Function Subroutines	74
5.3	Recursive and Reentrant Subroutines	75
5.4	Built-in Functions	76
5.4.1	Trigonometric Function	76
5.4.2	Absolute Value	76
5.4.3	Base e Exponential	77
5.4.4	Base 2 and Base 10 Logarithm	77
6.	COMMANDS	78
6.1	Debug Command	78
6.2	Optimization Command	79
6.3	Count Command	79
6.4	Define Command	80
6.5	Execute Command	86
7.	EXTENDED SPL	87
7.1	Complex Values	87
7.2	Code Declarations	87
7.3	List Declarations	92
7.4	List Processing Statements	92

CONTENTS

	<u>Page</u>	
7.4.1	Reference Statements	93
7.4.2	Link Statements	95
7.4.3	Free Statements	98
7.5	Encoding and Decoding	100
7.6	Algebraic Formula Manipulation	106
7.6.1	The Evaluation Function	112
7.6.2	The Represent Function	112
7.6.3	The Defined Function	113
7.6.4	The Identity Function	113
7.6.5	The Approximate Function	114
7.6.6	The Reduce Function	114
7.6.7	The Expand Function	115
7.6.8	The Coefficient Function	116
7.6.9	The Differentiation Function	116
7.7	Interactive Programming	118
7.8	Commands	119
7.8.1	Edit Commands	119
7.8.2	Save Commands	123
7.8.3	Get Commands	125
7.9	Built-in Functions	127
7.9.1	Functions	127
8.	LISTING OF SYNTAX EQUATIONS	130
	INDEX	143

1. INTRODUCTION

1.1 INTENDED USE OF THE SPECIFICATION

This document contains a complete specification of the Space Programming Language (SPL). Included is a description of the alphabet, the syntactic forms and the meaning of each language element. The metalanguage used to describe SPL syntax is the Backus-Naur form with a few modifications.

The first and foremost consideration made in presenting this material is precision and completeness of information content. Thus, the intent is to present a language in a well defined form. This description is not a learner's text; it is intended to be the authoritative reference on SPL for the programming language designer, reviewer, developer, and implementer.

1.2 BACKGROUND

The development of the Space Programming Language is a result of a recommendation made to the Space Systems Division Directorate of Technology based on a study performed by the System Development Corporation as documented in SSD-TR-67-11, "Recommendations for a Common Space Programming Language - Volume III," January 1967. This study of programming languages for spaceborne software was performed between the time period August 1966 and January 1967. The primary purposes of this study were to determine:

- a. the language elements required for spaceborne programming in the time period 1968 - 1973.
- b. if a common higher-order language would be feasible and useful for spaceborne software.

A comprehensive analysis of spaceborne software projects, such as Minuteman, Centaur, Apollo, Gemini, and Titan III, indicated a trend toward increasing reliance on data processing for mission planning, simulation for vehicle development and on-board data processing functions.

1.3 LANGUAGE REQUIREMENTS

In the analysis of the spaceborne software application area, it became apparent there were three distinct areas for language requirements which we shall refer to as: Flight Programming, Development Programming and Support Programming.

The data processing applications in these three areas can be summarized as follows:

Table 1. Three Requirement Areas for SPL

Flight Programming	Development Programming	Support Programming
Keyboard & Display	Mission planning	Computer simulators
Event Sequencing	Equation formulation	Vehicle simulators
Navigation	Scientific simulations	Programming tools
Guidance		Data reduction
Flight Control		
Experiment monitoring		
System testing		
Digital communications		

Programming personnel utilized to perform the programming for each of these three areas can be summarized as follows: for flight programming, professional programmers oriented toward numeric data processing are utilized; for development programming engineers and personnel from other scientific disciplines are utilized; for support programming, programmers and, to a lesser extent, engineers, perform the programming tasks.

The computers to be utilized in spaceborne data processing during the projected time period will be off-the-shelf machines, of a capability comparable to present-day machines. Implementation of SPL, however, will be, in most cases, on a large, general-purpose, ground machine.

The computer program production technique will continue to be largely batch processing. Interactive or time-sharing program production technique should play an increasing part in the program production process.

The burdens assumed by spaceborne software will become greater with more ambitious space programs. Astronauts and scientists on manned missions will use on-board computers for analysis of experimental data, as well as for on-board navigation, guidance, system monitoring and control, and perhaps even crew training and assignment. In addition, ground-based users will rely on computers for reduction of space data, as well as for satisfying the computational requirements of general research. For greater effectiveness, an increasing amount of the software development work should be done interactively with the scientist or programmer in direct communication with the machine.

1.4 SPL RECOMMENDATION

A recommendation for the development of SPL was made after analysis of existing languages. Because of the diversity of the application, the study further recommended that a basic language be designed for spaceborne data processing and that to accommodate the other two applications in space

software, an SPL extension should be developed. This conclusion was reached after an analysis of the needs of potential language users, the mission functions and their required data processing support, the types of programming required, the hardware utilized, and the program production methods used. The resulting language, SPL, is based, in part, on JOVIAL which has been recently adopted as an Air Force Standard Programming Language.

The study further identified the present time as particularly opportune to develop a higher-order language for space applications. Several factors, such as ground and space hardware changes, expanded space programming requirements, and the lack of a higher-order language for use in spaceborne applications, combine to make this a useful period for implementation of a language for space software for use during the latter 1960's and early 1970's.

1.5 SIGNIFICANT FEATURES OF SPL

The varied requirements of computer programming for space applications, which encompass mathematical programming, system programming, and real-time programming, result in a widely expressive, yet easily extendable programming language. Other equally vital needs for the language are: easy for nonprogrammers--engineers and space scientists--to use, practical to implement, and, in the hands of professional programmers, economical of computing resources. And finally, the language must be highly machine-independent and yet capable of exploiting unique machine characteristics.

The requirements of SPL have been established with these needs in mind, and it has been possible to outline a language that meets the needs of each area of space programming without compromising the needs of others.

1.5.1 Basic SPL

Since the specification for SPL has been organized by first defining the basic language and then the SPL extension, the description of the capabilities of SPL will be organized in the same manner.

1.5.1.1 Operations

Basic SPL incorporates a limited but very powerful specific set of operations. They include:

- a. Logical and relational operations
- b. Built-in operations (functions)
- c. Arithmetic operations (formula evaluation)
- d. Real-Time Control operations
- e. Input/output operations
- f. Command operations (for the compiler)
- g. Notational extension operations

SPL incorporates all of the ordinary logical and relational operations and a capability for incorporating built-in operations is provided. The most significant of these operations is arithmetic operations, real-time control operations and command operations.

1.5.1.1.1 Arithmetic Operations. In addition to the ordinary arithmetic operations, special vector and matrix operations are incorporated as primitives in the language. Further, arithmetic operations can include pairs of operands which can differ in dimension (scalar vs. multi-dimensional values) in representation (fixed-point vs. floating-point values) and in other subsidiary attributes. The precision (or accuracy) of arithmetic operations may be rigorously controlled through scaling information attached to the processing statement. Parentheses may be used freely in constructing numeric formulas of arbitrary complexity, according to the notational conventions of ordinary algebra.

1.5.1.1.2 Control Operations. Extensive program control operations are available for the handling of interrupts, device monitoring, parallel processing and input/output processing. Execution of program statements (including compound statements) may be specified as conditional, repetitive, chronic (occurring whenever a specified condition occurs), delayed (until a specified condition occurs), or in parallel with the execution of other statements. These primitive control operations, in conjunction with a set of implementation-defined hardware operands, are the minimum needed to provide the professional programmer with complete control through the SPL programming language. Though complete, they do not entail unnecessary burden by implying operations that can more effectively be specified by custom-tailored sequences of other available operations. Statements which allow for the control and monitoring of time increments are also provided. This is very important where code sequences must be executed within a given time frame.

1.5.1.1.3 Command Operations. Command Operations include commands to the compiler to produce optimized code for time (object program run time) or space (object program computer storage requirements). Commands are also provided for, for debugging the object program and for time counts of object program code time requirements. An execute command exists to allow a set of code to be operated at compile time to initialize a parameter.

One of the most significant of SPL's command operations is the notational extension capability. This provides for defining notational extensions for new data types and structures and new operations in terms of existing language elements. The facility in building new operations and extending punctuation and vocabulary allows versions of SPL to be customized to satisfy special programming problems. For instance, a programmer might define an extended notation and vocabulary for his particular programming area, and build a highly problem-oriented vocabulary and language capability.

In addition, notational definitions may be used to make existing programming languages compatible with SPL. A notational definition package can be generated which would map languages such as FORTRAN, PL/I, or JOVIAL into SPL. This would allow an SPL compiler to process these languages, thus allowing a programmer to code in these languages while the SPL compiler produces an equivalent SPL code and listing. This would also circumvent the necessity to reprogram the existing inventory of problems which are operational and coded in some other language.

The notational definition capability can also be used to aid in the implementation of SPL. A core subset of SPL can be implemented using conventional means; notational definitions can then be used to "define" the balance of the language, thus reducing implementation time and cost.

1.5.1.2 Data Declarations

Basic SPL incorporates a wide variety of operand types and structures. Data structures include: item declarations, array declarations, and group declarations. One declaration is used to describe the storage of these collections of data;

storage declarations

Further provision is made in basic SPL for one type of input/output declaration;

file declaration

SPL incorporates numeric operands including fixed-point, arbitrary-precision floating point, vector, and matrix values; primitive (i.e., built-in) alphabets; and symbolic operands including Boolean and status values. Basic SPL also provides for almost any type of data structure, including combinations of arrays, groups, and files. To achieve the most efficient use of storage, the programmer has the option of specifying exactly how storage is allocated to his data elements.

1.5.1.3 Program Structures

Program structure in SPL is based on the powerful, generalized block-structure concept. Procedure subroutines and function subroutines, recursive subroutines, and re-entrant subroutines may be specified.

The language syntax has been designed to minimize grammatical rules and punctuation. This will serve to minimize the amount of training required and reduce scripting errors when programming. There has also been an attempt to minimize vocabulary without sacrificing clarity. Where a needed capability already exists in the JOVIAL language, JOVIAL notation has been used if it is consistent with the criteria previously described.

1.5.2 SPL Extension

The SPL extension contains the basic operations, plus additional features suited for the applications area. For developmental programming, algebraic formula and interactive programming operations are specified. For support programming, simple text and list processing operations are included. Additional data definitions in extended SPL include list declarations, code declarations and program-declared alphabets (see Section 7).

1.6 NOTATION AND METALANGUAGE

1.6.1 Notation

This report gives a complete specification of SPL and extension using as a syntax metalanguage, a modified Backus-Naur Form* (BNF). Some typographic conventions are introduced to distinguish among terms (which are constructed by the language designer to identify and categorize the various parts of the language), names (which are constructed by the programmer to identify the elements of his program), and primitives (which are the built-in "words and symbols" of the language).

Terms are printed using lower case letters. For example:

statement
algebraic-formula

Names are printed using upper case letters. For example:

ALPHA
GROSS
T23

And finally, word-like primitives are underlined. Implementation defined primitives, such as hardware names, are capitalized in addition to being underlined. For example:

for
while
and
CLOCK

It is important to remember, however, that these typographic conventions are part of the metalanguage notation, and not part of SPL.

* As used in the "Revised Report on ALGOL 60," Communications of the ACM, Peter Naur, May 1960.

The syntax of SPL is specified by defining terms. The main elements of these definitions are the signs, symbols, and other terms. Except for the non-printing graphic character for "space" then, SPL signs stand for themselves.

1.6.2 Metalinguage

The metalanguage used in this specification has three basic elements. They are:

- a. ::= This symbol signifies syntactic equivalence and should be read as the word "is".
- b. | This symbol signifies selection between alternate strings of elements and should be read as the word "or".
- c. < > These symbols signify a grouping and are used to enclose alternatives. They should be read as the word "either" and are used with the symbol "|".

There are two metalinguistic extensions used:

- a. Subscripting is utilized as a semantic cue to distinguish among otherwise identical terms.
- b. nothing This term signifies a null term or an empty string of symbols or signs.

There is one SPL term "space" introduced which is represented in SPL by the lack of characters and is represented in the metalanguage notation by the following symbol:

- △ This symbol signifies separation of syntactic strings and is inserted for clarity. The symbol "△" represents optional, not required separation.

To simplify the semantic explanation, alternative definitions of certain terms are given at different places in the report. This has been noted, but the index at the end is perhaps the most convenient guide. Finally, it is worth noting that no attempt has been made to specify in BNF the syntax of SPL with complete rigor. Certain syntactic aspects of any programming language can more clearly and simply be described in prose, where a BNF description would be lengthy, and complex.

For those already familiar with BNF, the extensions used in this report are essentially just two: the brackets, "<" and ">", are used for grouping rather than delimiting terms, and semantic-cue subscripts are used to distinguish otherwise identical terms. The purpose of these two extensions is to reduce the number of terms that have to be defined, with the ultimate goal being to define syntactically all and only those terms needed in the prose description of the semantics of the language. Without such extension, BNF ordinarily requires the syntactic definition of many otherwise unnecessary terms.

BNF, even as extended here, is actually quite easy to read. A pair of examples, defining a parade should make this clear.

```
parade ::= parade-unit <parade-unit|parade>
parade-unit ::= float | band | drill-team | bunch-of-guys-on-horses
```

The first definition says: a parade is a parade unit followed by either another parade unit, or a whole parade. And the second says: a parade unit is a float, or a band, or a drill team, or a bunch of guys on horses. The first definition specifies, precisely, that a parade must have at least two parade units, but the number of parade units it may have is not limited. When a thing is defined in terms of itself it is called a recursive definition, and is frequently used in BNF language description for reasons of clarity and conciseness.

2. ALPHABET, VOCABULARY AND PROGRAM STRUCTURE

2.1 ALPHABET AND SYMBOLS

SPL's symbols may be formed from a basic alphabet of 48 characters consisting of the 26 letters, the 10 decimal digits, and a dozen miscellaneous marks including the space and the dollar sign. This alphabet is almost universally available on mechanical printing, typing, and card punching equipment. However, SPL also permits the use of an extended character set. In practice, the extended characters will depend upon the characteristics of the equipment that is available.

SYNTAX

character ::= letter | digit | mark

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

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

mark ::= space | (|) | + | - | * | / | . | , | ' | = | \$

Where lower-case letters are available they may be used indistinguishably from the basic set of uppercase letters to improve the typographical appearance of the program.

The alphabet of SPL is used to form symbols which are the basic elements of the language. Symbols are syntactically defined as names, primitives, and constants.

SYNTAX

symbol ::= name | primitive | constant

2.2 NAMES

Names serve to identify the various program and data elements that may be referenced in an SPL program: statements; subroutines, items; groups; arrays; files; patterns; hardware operands; hardware operators. A name is a string of one or more letters and digits which may be punctuated for readability by the period. Notice that a name must begin with a letter, must not end with a period, may contain no embedded spaces and no embedded period strings of length greater than one.

SYNTAX

name ::= <letter | name> <nothing | letter | digit | .letter | .digit>

Examples:

ALPHA

Z

STEP.27.3

BRANCH

Names are defined at the point where they are terminated by a period. Thus, ALPHA. indicates that the name ALPHA is being defined.

To facilitate the independent composition of portions of a program, statement names defined in a program have a strictly determined scope of definition for statement reference, being bounded by the innermost pair of named begin and end brackets containing the definition of the name.

It should be noted that the strict determination of the scope of definition of names for statements and declarations does not apply to commands; any previously defined name may be referenced in a command.

Although it is poor programming practice, where a name is defined to be identical to a SPL primitive word, there is no ambiguity in contexts where the syntax rules out one or the other.

2.3 PRIMITIVES

Primitives are the built-in symbols of SPL--its punctuation, verbs, adjectives, etc. Primitives have fixed meanings, as described in later sections. Primitives for basic SPL fall into the categories given below:

SYNTAX

primitive ::= delimiter | operator | descriptor

2.3.1 Delimiters

Delimiters are those symbols of the language which serve exclusively to indicate the bounds of other syntactical elements.

SYNTAX

delimiter ::= . | , | / | \$ | (|) | ' | '' | begin | end | term | program

2.3.2 Operators

Operators are those symbols in a language which indicate some action is to be performed on an operand.

SYNTAX

operator ::= catenation-operator | repetition-operator | conditional-operator |
arithmetic-operator | define-operator | logical-operator |
relational-operator | assignment-operator | functional-operator |
discrimination-operator | sequential-operator | input-output-operator |
location-operator | editing-operator | compile-operator

arithmetic-operator ::= - | + | * | / | **

logical-operator ::= not | and | or

relational-operator ::= eq | nq | gr | ls | gq | lq | equiv

conditional-operator ::= if | then | else

assignment-operator ::= = | set

repetition-operator ::= times | by | while | until | for

catenation-operator ::= // | ///

discrimination-operator ::= sub | () | at | in

sequential-operator ::= goto | stop | when | on | call | entry | exit | for | do

input-output-operator ::= open | close | read | write | assign | status | position

location-operator ::= store | at

editing-operator ::= out | is | all | to | show | thru

compile-operator ::= optimize | count

define-operator ::= execute | where | then | names of-terms*

2.3.2.1 Functional Operator. Functions are a special class of operators allowable within SPL and which are governed by a set of rules outlined in Section 5. In general, functions act on a parameter list which follows the name and returns a value which may be part of a larger formula. A list of intrinsic (built-in) functions is given below:

SYNTAX

functional-operator ::= log.e | log.10 | sin | cos | tan | abs

The programmer may define his own set of functions that will be treated in a manner similar to those intrinsic to SPL.

2.3.3 Descriptors

The descriptors are the functional modifiers and descriptions of operands in SPL.

SYNTAX

descriptor ::= integer | real | pointer | boolean | array | mode | procedure |
function | pattern | file | dec | oct | hex | bit | text | fixed |
float | cell | true | false | ready | busy | error | addr |

* Listing of terms in Section 7.

item | minimum | digit | maximum | signed | unsigned | round |
truncate | group | compool | full | unready | value | result |
recursive | reentrant | time | space | none

2.4 CONSTANTS

SPL programs manipulate both numeric data (integer and real) and nonnumeric data (textual, pointer, and Boolean values). The symbols that denote these values are constants.

SYNTAX

constant ::= numeric-constant | textual-constant | pointer-constant |
boolean-constant

2.4.1 Numeric Constants

SPL includes constants for denoting integer or real values. Integer values may be denoted by binary, octal, decimal, and hexadecimal constants.

SYNTAX

numeric-constant ::= number | real-constant | binary-constant | octal-constant |
decimal-constant | hexadecimal-constant

numeral ::= digit <nothing | numeral>

signed ::= + | -

number ::= numeral <nothing | $e_{\text{xponent-base-10}}$ numeral>

real-constant ::= <numeral . | . numeral | numeral . numeral> <nothing |
 $e_{\text{xponent-base-10}}$ numeral | $e_{\text{xponent-base-10}}$ signed numeral>

Examples:

018

123e4

.5

6.789e-10

Integer and real constants denote numeric values in the conventional decimal sense. The numeral following e in these constants is a decimal scaling factor expressed as an integral power of 10. Binary, octal, decimal, and hexadecimal constants have the obvious meaning of unsigned binary, octal, decimal, or hexadecimal integers.

SYNTAX

binary-constant ::= <name. | nothing> Δ bit ' binary-string '
binary-string ::= <0 | 1> <nothing | binary-string>
octal-constant ::= <name. | nothing> Δ oct ' octal-string '
octal-string ::= <0 | 1 | 2 | 3 | 4 | 5 | 6 | 7> <nothing | octal-string>
decimal-constant ::= <name. | nothing> Δ <nothing | dec> ' numeral '
hexadecimal-constant ::= <name. | nothing> Δ hex ' hexadecimal-string '
hexadecimal-string ::= <numeral | A | B | C | D | E | F> <nothing |
hexadecimal-string>

Examples:

AL. bit'11011100'

oct'334'

ABC. dec'156'

hex'9C'

2.4.2 Textual Constants

A textual constant is a symbol used to denote a string of one or more alphanumeric characters or a status value. The omission of the word text indicates that a status value rather than an alphanumeric string is being defined.

SYNTAX

textual-constant ::= <name. | nothing> <nothing | text> ' character string '
character string ::= character <nothing | character-string>

Examples:

text 'THIS IS AN ALPHANUMERIC CONSTANT.'

text 'SO IS THIS.'

ALP. text '92768'

QUALITY 'GOOD'

STATE 'KANSAS'

2.4.3 Pointer Constants

There are two types of pointer constants. The first type addr gives the effective address value of a statement, procedure, subroutine, array, or group and the second cell gives an index value within arrays, groups, or tables.

SYNTAX

pointer-constant ::= cell Δ name | addr Δ name

A cell is not necessarily the same as a computer word: internal storage is not necessarily limited to hardware considerations but is logically oriented and depend upon item structure within tables or arrays.

Pointer constants serve to denote pointer values and are represented as integers.

An addr, on the other hand, refers to the actual value of the location to which a particular element in the language has been assigned.

2.4.4 Boolean Constants

The Boolean constants true and false have the obvious meanings; true is represented by 1 and false is represented by 0.

SYNTAX

boolean-constant ::= true | false

2.5 STATEMENT STRUCTURE

Statements are the operational units of SPL. They describe the data processing actions that the program is to perform: computational actions; input-output actions; and statement sequence-control actions. It is convenient, however, to recognize two types of statements: 1) simple statements, which express computational, input-output, or control actions whether simple or conditional, and which may incorporate other statements within them and control their execution; and, 2) compound statements, which group together whole strings of simple, or compound statements. Compound statements may also contain declarations and commands.

Statements are normally executed in the sequence in which they were written, although control statements affect this sequence and provide exceptions to this rule. Statements may be named--so they can be referenced and executed out of the normal sequence.

SYNTAX

```
statement ::= simple-statement | compound statement
compound-statement ::= <name. | nothing> Δ begin Δ statement-string Δ end Δ
                    <name | nothing>
statement-string ::= statement | declaration | command <statement-string |
                    nothing>
declaration ::= data-declaration | subroutine-declaration | file-declaration
simple statement ::= simple-control-statement | input-output-statement |
                    procedure-call-statement | assignment-statement
```

A comma (,) may be optionally used to terminate any simple statement.

The definition of command is deferred to Section 6.

2.6 COMMENTS

A comment allows a remark or clarifying prose or punctuation to be included among the symbols of an SPL program. Comments are ignored by the compiler and so have no operational effect whatever on the program.

SYNTAX

comment ::= "character-string"

The character-string in a comment may not contain the comment delimiter.

Example:

"THIS IS A COMMENT"

The omission of a comment bracket, or the inclusion of an extraneous comment bracket within the comment, is a major error, for subsequent commentary is interpreted by the compiler as part of the program.

2.7 PROGRAM STRUCTURE

In SPL, a program is merely a named statement-string, beginning with the program delimiter and followed by declarations, statements, and/or commands followed by the term delimiter. The program name defines it for external reference.

SYNTAX

program ::= program Δ name.Δstatement-string Δ term <name | nothing>

3. DATA DEFINITIONS

Basic SPL provides declarations for defining numeric, textual, pointer, and Boolean items as well as for defining arrays, and groups. In addition, the arrangement of elements in memory may be specified and various default descriptions (modes) may be specified.

SYNTAX

data-declaration ::= item-declaration | array-declaration | group-declaration |
storage-declaration | mode-declaration

Other declarations for defining functions, procedures, files, and textual patterns (see Sections 4,5,6) will be described in later sections.

3.1 ITEM DECLARATIONS

In SPL, the basic (scalar) units of data are called items. All necessary attributes of an item's value, such as its type and format, are supplied only once in an item description. In the absence of an item declaration, data is assumed to be mode-defined.

In SPL, values other than those denoted by constants or those used only as intermediate results must be declared. Several different but similarly described items may be declared at once.

SYNTAX

item-declaration ::= item | nothing > Δ name-string Δ item-description Δ <nothing |
initial-value-string>
name-string ::= name. Δ <nothing | name-string>
initial-value-string ::= /item-value Δ <nothing | initial-value-string>
item-value ::= numeric-constant | pointer-constant | textual-constant |
boolean-constant
item-description ::= numeric-item-description | textual-item-description |
pointer-item-description | boolean-item-description

numeric-item description ::= full-integer-item-description | full-real-item-description

full-integer-item-description ::= integer Δ number₁ Δ <bit | digit> Δ <minimum | nothing> Δ <nothing | number₂ Δ maximum> <nothing | signed | unsigned> Δ <nothing | round | truncate>

full-real-item-description ::= real Δ number₁ Δ <bit | digit> Δ <nothing | minimum> Δ <nothing | -> number₂ <bit | digit> Δ <nothing | float | fixed> Δ <nothing | signed | unsigned> Δ <nothing | round | truncate>

textual-item-description ::= text Δ <nothing | number Δ character | name of-integer-item Δ character>

pointer-item-description ::= pointer

boolean-item-description ::= boolean

- Note
- In real item declarations, a negative scale number (of fractional or exponent) bits is only relevant where the scale is fixed.
 - Number₁ indicates minimum number of bits or decimal digits.
 - Number₂ indicates either number of fractional bits or decimal digits or number of bits or decimal digits needed to represent the exponent.

Examples:

item ADAM. integer 6 bit minimum /74

item BE9. real 31 bit 7 bit float signed truncate

item ROD. text 4 character

item SA. pointer

item BOB. boolean

The numeric item descriptions have several common elements. In an integer, real, or item description, number_1 indicates the minimum number of binary bits or decimal digits*--including any fraction or exponent but excluding any sign--needed to represent the item; the unsigned descriptor indicates that the item's value is always positive or zero; the omission of the unsigned descriptor or indication of a signed item indicates that the item can also take on negative values; the truncate descriptor indicates that any value assigned to the items is to be truncated rather than rounded, as would be the case if round or nothing were used.

Abbreviated descriptions are possible for numeric items, according to the declared mode (see Section 3.4). In an integer item description, the maximum absolute value that the item will be assigned is indicated by number value. (Where this is omitted, the maximum absolute value is taken to be either $2^{\text{number}_1 - 1}$ or $10^{\text{number}_1 - 1}$, depending, of course, on whether bits or digits are indicated.) In an integer item if the minimum number of bits is omitted the initial value will be used as an indicator of the minimum number of bits.

In a real item description, fixed-point representation may be indicated by the fixed descriptor. Floating-point representation is assumed where nothing or float is indicated. Where fixed-point representation is indicated, number_2 indicates the number of fractional bits or digits. If the indicated number_2 of fractional bits or digits is negative (as indicated by the presence of the minus sign, -), the number_2 of low order integer bits or digits are not significant and therefore need not be carried. And if the indicated number_2 of fractional bits or digits is greater than number_1 , then the $(\text{number}_2 - \text{number}_1)$ high order fractional bits or digits are not significant and therefore need not be carried. On the other hand, where floating-point representation is indicated

*Only one base is used, although the programmer can specify numeric item size in terms of either.

(by the omission of the fixed descriptor), number₂ indicates the minimum number of bits or digits needed in the binary or decimal exponent to adequately represent the item's value.*

In a textual item description, the length of the item in characters may be indicated by a number, or it may be indicated by the name of an integer item, where the current length of the textual item is specified by the current value of the integer item. Where no length is indicated, a length of 1 character is assumed. The last symbol in a textual item description indicates the item's alphabetic code.

Pointer and Boolean items are described with the pointer and boolean descriptors. (A Boolean item is actually represented as a one-character, binary textual item, with the Boolean constants true and false equivalent to bin '1' and bin '0'.)

3.2 ARRAY DECLARATIONS

An array declaration describes the structure of a collection of similar data elements--either items or groups, Rectangular arrays of practically any number of dimensions may thus be declared and several different but similarly described arrays may be declared at once.

SYNTAX

array-declaration ::= array Δ name-string Δ array-description

array-description ::= <item-description | group-description> Δ dimension-string
Δ <initial-value-string | nothing>

dimension-string ::= <number | name_{of-integer-item}> Δ <nothing | by Δ dimension-string>

*The floating-point operations on most machines permit only limited variations (if any) on the values of number₁ and number₂. The SPL compiler must therefore translate these into the values indicating the appropriate single- or multiple-precision floating-point representation.

Examples:

array PRESSURE. integer 4 digit 10 by 20 by 5

array GRID1. boolean 32 by 32

array A. begin I. J. K. integer 25 bit end (A)

In designating an individual element from an n-dimensional array, the array name must be subscripted by an n-component index string of numeric formulas. And where the size of a dimension is K elements, the integral value (truncated, if necessary) of the corresponding component of the index string can only range from 1 to K.

Abbreviated descriptions are possible for arrays according to the declared mode (see Section 3.4).

3.3 GROUP DECLARATIONS

A group is a collection of (usually) dissimilar data elements--items, arrays, and even subgroups. A group declaration serves to describe the elements of a group and give it, optionally, a name. (A group name may be omitted when the group is never referenced as an entity, but only its elements.) In addition, functional relationships among the elements of a group may be declared within a group declaration, as functional data elements of the group. Several different but similarly described groups may be declared at once.

SYNTAX

group-declaration ::= group Δ <nothing | name-string> Δ group-description

group-description ::= begin Δ declaration-string Δ end Δ <nothing | (name)

declaration-string ::= <item-declaration | array-declaration | group-declaration | function-declaration | mode-declaration | storage-declaration> Δ <nothing | declaration-string>

Note: Where groups are nested, the inclusion, in parentheses after the group description, of the group or array name (if any) immediately preceding the group description, automatically "ends" any "open" subgroups (or compound statements) within the groups. Thus, "end (name)" in the expression "name. begin ... end (name)" may be syntactically equivalent to a string of several end brackets.

Examples:

```
group begin item I. integer end
```

```
group A. begin item W. integer group Q. begin item P. text 6  
character item V. boolean end (A)
```

```
group TRACK. begin group INITIAL. begin item X. Y. Z. real end item N. integer  
"NUMBER OF LEGS" 2 digit, 80 digit maximum unsigned truncated end (TRACK)
```

3.4 MODE DECLARATIONS

Mode declarations serve to declare normal modes of description for numeric items, arrays, lists, and files.

SYNTAX

```
mode-declaration ::= mode Δ <numeric-item-description | array-description |  
full-file-description>
```

Examples:

```
mode integer 15 bit unsigned truncate
```

```
mode real 31 bit 7 bit scale
```

3.5 STORAGE DECLARATIONS

Although the programmer is often unconcerned with the details of memory allocation, he may control it with storage declarations. A storage declaration serves to indicate to the compiler the desired arrangement within memory of the various program elements--items, arrays, groups, files, statements, functions, and procedures--named in the declaration.

SYNTAX

storage-declaration ::= store Δ block-description Δ at Δ pointer-formula

block-description ::= name_{of-element} Δ <nothing | block-description>

Note a. The name of any element may appear only once per storage declaration, but may appear in other storage declarations if logical inconsistencies are avoided, such as declaring once that A is stored after B and again that B is stored after A.

Note b. When a storage declaration appears within a group description, only those items, arrays, and subgroups declared within the group may be named in the storage declaration. This excludes functions declared within the group as well as external elements.

Examples:

store A at cell sub I

store A B at step

store A at bit (9 to 11)

store B at byte hex (1 to 6)

The elements--items, arrays, groups, files, statements, functions, and procedures--named in a block description are allocated, in the sequence given, a block of consecutive units of storage. Each different block described in a storage declaration is allocated storage beginning at a common origin cell. In

other words, each block has the same pointer value, and this value may be explicitly specified in the storage declaration by a pointer formula. Each block thus "overlays" the other blocks listed in the declaration, permitting the programmer to utilize the same block of memory for different purposes at different times during the computation.

3.6 VARIABLES

In SPL, variables (scalar) item values, arrays of values, and groups of values may be altered during the course of program execution. Variables may be specifically located in memory and they may be subscripted by numeric formulas to distinguish them from other elements of arrays. Variables may be both conditional and subscripted. Variables may also be enclosed in parentheses to alter or emphasize the sequence in which these operations are performed. In addition, certain compiler-dependent hardware operands and certain functions may also serve as variables.

SYNTAX

variable ::= name | subscripted-variable | conditional-variable |
catenated-variable | hardware-operand

Note 1. The name must be that of an item, an array, or a group.

Note 2. To be a variable, a hardware operand must be one that can be assigned a value by programmed action.

Examples:*

The exemplary variables in this section will often involve the following data elements:

array A. begin ... end (A) 6

$$A \equiv \begin{bmatrix} (A_1) \\ (A_2) \\ (A_3) \\ (A_4) \\ (A_5) \\ (A_6) \end{bmatrix}$$

* The \equiv sign stands for semantic equivalence.

array I. integer 3

$$I \equiv \begin{bmatrix} (I_1 \equiv 2) \\ (I_2 \equiv 4) \\ (I_3 \equiv 1) \end{bmatrix}$$

array B. begin ... array C. begin ... end 2 end 3 by 2

$$B \equiv \begin{bmatrix} B_{1,1} \equiv C_{1,1} \equiv \begin{bmatrix} (C_{1,1,1}) \\ (C_{1,1,2}) \end{bmatrix} & B_{1,2} \equiv C_{1,2} \equiv \begin{bmatrix} (C_{1,2,1}) \\ (C_{1,2,2}) \end{bmatrix} \\ B_{2,1} \equiv C_{2,1} \equiv \begin{bmatrix} (C_{2,1,1}) \\ (C_{2,1,2}) \end{bmatrix} & B_{2,2} \equiv C_{2,2} \equiv \begin{bmatrix} (C_{2,2,1}) \\ (C_{2,2,2}) \end{bmatrix} \\ B_{3,1} \equiv C_{3,1} \equiv \begin{bmatrix} (C_{3,1,1}) \\ (C_{3,1,2}) \end{bmatrix} & B_{3,2} \equiv C_{3,2} \equiv \begin{bmatrix} (C_{3,2,1}) \\ (C_{3,2,2}) \end{bmatrix} \end{bmatrix}$$

array J. integer 2 by 3

$$J \equiv \begin{bmatrix} (J_{1,1} \equiv 5) & (J_{1,2} \equiv 3) & (J_{1,3} \equiv 2) \\ (J_{2,1} \equiv 6) & (J_{2,2} \equiv 1) & (J_{2,3} \equiv 4) \end{bmatrix}$$

\$LIGHT \equiv a Boolean 36-array of console lights

Note: \$ signifies hardware operand (see Section 3.6.4).

3.6.1 Subscripted Variables

Elements of a nonscalar variable (e.g., an array) may be designated as a variable by subscripting with an index string, which is essentially a numeric formula that is interpreted according to the dimension of the variable being subscripted.

SYNTAX

subscripted-variable ::= variable <(index-string) | Δ sub Δ index-string>

index-string ::= index Δ <nothing | < // | to Δ index-string>

index ::= <numeric-formula | index-string> Δ <nothing | index> | (index)

Examples:

$$A \text{ sub } 3 \equiv A_3$$

$$A \text{ sub } (3//1) \equiv (A \text{ sub } 3, A \text{ sub } 1) \equiv \begin{bmatrix} A_3 \\ A_1 \end{bmatrix}$$

$$A \text{ sub } ((3//1)) \equiv [A_3 \ A_1]$$

$$A \text{ sub } (1 \text{ to } 3) \equiv (A \text{ sub } 1//A \text{ sub } 2//A \text{ sub } 3) \equiv \begin{bmatrix} A_1 \\ A_2 \\ A_3 \end{bmatrix}$$

$$A \text{ sub } I \equiv (A \text{ sub } 2, A \text{ sub } 4, A \text{ sub } 1) \equiv \begin{bmatrix} A_2 \\ A_4 \\ A_1 \end{bmatrix}$$

$$A \text{ sub } (I \text{ sub } 2 //2) \equiv (A \text{ sub } 4, A \text{ sub } 2) \equiv \begin{bmatrix} A_4 \\ A_2 \end{bmatrix}$$

The description of the catenation operator // is in section 3.6.2.

$$A \text{ sub } (J \text{ sub } ((1//2), 2)) \equiv A \text{ sub } (J \text{ sub } (1,2), J \text{ sub } (2,2)) \equiv A \text{ sub } (3,1) \\ \equiv A \text{ sub } (3//1)$$

$$B \text{ sub } (J \text{ sub } ((1//2), 2)) \equiv B \text{ sub } (3,1)$$

$$A \text{ sub } J \equiv A \text{ sub } (5,6//3,1//2,4) \equiv A \text{ sub } ((5//6)//(3,1)//(2//4)) \equiv \begin{bmatrix} A_5 & A_6 \\ A_3 & A_1 \\ A_2 & A_4 \end{bmatrix}$$

$$B \text{ sub } (2//3) \equiv B \text{ sub } ((2,1) \text{ to } (3,2)) \equiv B \text{ sub } ((2,1//2,2//3,1//3,2)) \equiv$$

$$\begin{bmatrix} (B_{2,1}) & (B_{2,2}) \\ (B_{3,1}) & (B_{3,2}) \end{bmatrix}$$

$$(A \text{ sub } J) \text{ sub } 2 \equiv (A \text{ sub } J) \text{ sub } ((2,1) \text{ to } (2,2)) \equiv (A \text{ sub } J) \text{ sub } \\ (((2,1)//(2,2))) \equiv [A_3 \ A_1]$$

$$J \text{ sub } 1 \equiv [5 \ 3 \ 2]$$

$$(J \text{ sub } 1) \text{ sub } 2 \equiv J \text{ sub } (1,2) \equiv J_{1,2}$$

An index string conforms to the syntax rules for numeric formulas. In this light, an index string is a row-vector of indexes, which are themselves column vectors whose elements are either positive (i.e., greater than zero) truncated integers as specified by numeric formulas, or else index strings. An index containing an index string is semantically equivalent to an index string, as explained in the following example:

$$A, (B // C // D), E \equiv A,B,E // A,C,E // A,D,E$$

By such transformations, any index string may be simplified by expansion to an equivalent index string whose component indexes do not themselves contain index strings.

The effect in an index string of the repetition operator to is as follows:

$(i_n, \dots, i_2, i_1) \text{ to } (t_n, \dots, t_2, t_1) \equiv (\dots (*i_n, \dots, i_2, i_1 \text{ to } i_n, \dots,$
 $i_2, i_1 \pm 1 \text{ to } \dots \text{ to } i_n, \dots i_2, t_1 \pm 1 \text{ to } i_n, \dots i_2, t_1) \text{ to } (i_n, \dots, i_2 \pm 1, i_1$
 $\dots, i_2 \pm 1, i_1 \pm 1 \text{ to } \dots \text{ to } i_n, \dots, i_{k+1}, t_k, t_{k-1}, \dots, t_2, t_1) \dots) ** \text{ to}$
 $(\dots (*i_n, \dots, i_{k+1} \pm 1, i_k, i_{k-1}, \dots, i_2, i_1 \text{ to } \dots \text{ to } t_n, \dots, t_2, t_1 \pm 1 \text{ to}$
 $t_n, \dots, t_2, t_1) \dots) *$. While this is the general rule, a simpler example may
 prove helpful. Thus, $(1,1,1) \text{ to } (2,2,2) \equiv ((1,1,1 \text{ to } 1,1,2) \text{ to } (1,2,1 \text{ to } 1,2,2))$
 $\text{to } ((2,1,1 \text{ to } 2,1,2) \text{ to } (2,2,1 \text{ to } 2,2,2))$.

Nominally, an index for a k-dimension variable contains k or less components. And where an index contains less components, a rule of the following type holds. Consider the 3-dimension array: array P. begin ... end X by Y by Z. Then, P (A) \equiv P ((A,1,1) to (A,Y,Z)). Actually, an index with more than the nominal k components has a defined meaning. For a k-dimension variable, then, an index string of n simple indexes (i.e., indexes containing no component index strings) behaves as if assigned to a data element, E, declared as follows:

```

begin
K. integer
array E. integer K
end N
  
```

where K and N are positive integer items with maximum values k and n, and E is a positive, truncated integer array. After this (hypothetical) assignment, E (1) contains the first index, E (2) the second, and so on.

* n parentheses

** k parentheses

The result of a subscription operation on a variable is a column vector of the elements specified by each index in the (unsimplified) index string. Of course, these elements may be scalar or nonscalar, and in general, a subscription operation may produce a nonrectangular array.* Consider, for example, array Q. begin ... end J by K, where Q (A,B to C to D, E to (F,G to H,I))

$$= \begin{bmatrix} Q_{a,b} \\ Q_{c,1} \quad Q_{c,2} \quad \dots \quad Q_{c,k} \\ Q_{d,e} \\ Q_{f,g} \quad Q_{h,i} \end{bmatrix}$$

It is important to note, then, that where index strings are nested, the elements designated by the indexes of the outermost index string are row catenated, the elements designated by the indexes of the next level index strings are column catenated, and so on.

For purposes of subscription, a complex, scalar numeric value (e.g., Q) may be considered as a 2-element, real vector, so that Q (1) designates the real part of Q and Q sub 2 designates, as a real value, the imaginary part of Q. Similarly, a scalar, n-character textual value (e.g., T) may be considered as an n-element character vector, so that T (i) designates the ith character of T and T (i to i+k) designates the k-character subtext beginning at the ith character.

* Variables with the structure of a nonrectangular array can, of course be subscripted; but it must be done very carefully.

3.6.2 Catenated Variables

In SPL, scalar (single dimension) variables may be catenated into nonscalar (multi-dimensional) variables, which may in turn be catenated into nonscalar variables of greater size or number of dimensions.

SYNTAX:

catenated-variable ::= variable Δ <nothing | catenation-operator> Δ variable

Note. Row catenation is assumed where a column, plane, etc., catenation operator is omitted.

Examples:

$$A \text{ sub } (1 \text{ to } 3) // I \equiv \begin{bmatrix} A_1 & I_1 \\ A_2 & I_2 \\ A_3 & I_3 \end{bmatrix}$$

$$A \text{ sub } (6 \text{ to } 4) // C \text{ sub } (1 \text{ to } 3, 1) /// A \text{ sub } (1 \text{ to } 3) // I \equiv \begin{bmatrix} A_1 & I_1 \\ A_2 & I_2 \\ A_6 & C_{1,1} & I_3 \\ A_5 & C_{2,1} \\ A_4 & C_{3,1} \end{bmatrix}$$

$$I \text{ J sub } 1 \equiv \begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ J_{1,1} & J_{1,2} & J_{1,3} \end{bmatrix}$$

$$J \text{ sub } 1 (I \text{ sub } 1 // I \text{ sub } 2 // I \text{ sub } 3) J$$

$$\text{sub } 2 \equiv \begin{bmatrix} J_{1,1} & J_{1,2} & J_{1,3} \\ I_1 & I_2 & I_3 \\ J_{2,1} & J_{2,2} & J_{2,3} \end{bmatrix}$$

3.6.3 Conditional Variables

SPL permits variables to be conditionally specified.

SYNTAX

conditional-variable ::= if Δ condition Δ then Δ variable₁ Δ <nothing | else Δ variable₂>

A conditional variable specifies one of two alternative variables--if the condition is true then variable₁, otherwise (else) variable₂. Thus, if true then A else B \equiv A, and if false then A else B \equiv B.

Examples:

if I ls J then J else I

if A eq 27 or T then ALPHA

if T then I

3.6.4 Hardware Operands

Hardware operands are compiler-dependent data elements that may in general, acquire their values independently of, and without, programmed action.* A hardware operand may be a numeric, pointer, or Boolean item.

Hardware operands are often, but not always, "read-only" in nature. Some typical hardware operands are: clocks, switches; elapsed time counters; device status indicators; device control signals; program interrupt signals. Clearly, some hardware operands could also be described as files, with the choice depending on program efficiency considerations peculiar to the particular system. One such consideration is, of course, the practicality of automatically monitoring the changes in value of a hardware operand that is used in specifying the condition for executing a chronic statement.

*This is not a prerequisite, though. A hardware operand may be completely under programmed control.

SYNTAX

hardware-operand ::= \$ name

Examples:

\$LIGHT

\$CLOCK

\$KEYBOARD

\$TAPE

\$OVERFLOW

\$ACCUMULATOR

In a hardware operand, the \$ identifies the name as that of a hardware operand. Where a complete set of hardware operands is available, the programmer may command the entire machine. Hardware operands generally differ from machine to machine, so that programs containing them are usually machine-dependent.

3.7 COMPOOL DECLARATION

A compool declaration defines the name of the compool to be utilized for a program. The compool contains definitions of items, array, groups, or programs that are commonly used by a number of programs or procedures. Where this common usage exists the data may be defined once in a compool and then called upon by the program desiring to utilize it through the compool declaration. Data declarations within a program take precedence over compool declarations.

SYNTAX

compool-declaration ::= compool Δ name_{of-compool}

Examples:

compool BA1

The compool resides in the binary system library and is not recompiled with each program which references it unless it is changed. Where there is a conflict in name definition between the compool and the program, that conflict is resolved in favor of the individual program. The compool declaration should immediately follow the program identification statement.

4. STATEMENTS

4.1 ASSIGNMENT STATEMENTS

An assignment statement assigns the value specified by a formula to be the value thereafter designated by a variable. The assignment of nonscalar values is on an element-by-element basis.

SYNTAX

assignment-statement ::= <nothing | set> Δ variable Δ = Δ formula

Note: The statement prefix, set, is an optional "noise" word, useful for improving readability in certain contexts.

Examples:

ALPHA (1 to N) = 0

\$SIGNAL (K) = true

WEATHER (AIRBASE) = 'CLOUDY'

set IDENTIFIER = text 'CALCIUM CYCLAMATE'

set SWITCH = 16*44

set IT = p**y - (x+a**2)

PI = 4* arctan 1

Assignment is done as if in two steps: first, the formula is evaluated; then, the resulting value is assigned to the variable. The formula may involve the variable, in which case the old value of the variable is used in the calculations needed to evaluate the formula.

Assignment is defined basically on scalar operands. Nonscalar assignments are done on an element-by-element basis by index.

Assignment by index is done according to the following rules:

1. Both the variable and the specified operand are considered to be dimensionally normalized. This is done by, in effect, rearranging the dimensions of the two operands as follows:
(a) wherever a dimension in the variable or specified operand has only one element, unless that dimension is one whose size varies in the variable operand, it is moved so that it is a higher dimension than any with more than one or a variable number of elements; (b) both operands are then reduced in dimension by disregarding all dimensions higher than the highest dimension with more than one or a variable number of elements.

By considering both the variable and the specified operand to be thus dimensionally normalized, both a 2 by 1 by 1 by 3 by 1 by 9 array, and a 1 by 2 by 1 by 3 by 9 array, for example, may be considered, for assignment purposes only, as 2 by 3 by 9 arrays. As another example, a 2 by 1 by 3 by 9 by 9 specified array assigned to a 2 by 1 by 3 by N by 9 variable array, may be considered as a 2 by 3 by 1 by 9 specified array assigned to a 2 by 3 by N by 9 variable array. To further illustrate, where A is an N-element row vector and B is an N-element column vector, then $A = B \equiv A = B(1, I) (I=1 \text{ to } N)$, and $B = A \equiv B = A(I) //(I = 1 \text{ to } N)$.

2. After any dimensional normalization, where the number of elements in the *i*th dimension of the variable is specified by an integer item, that item is automatically assigned the value $\min(x, y)$, where *x* is the size (*i*) of the specified operand and *y* is the maximum value declared or determined for the item.

3. The specified operand is then dimensionally adjusted, where necessary, to the dimensions of the variable. Where the specified operand has fewer dimensions than the variable, it is converted to the higher dimension by appropriate replication. And where it has more dimensions than the variable, its higher dimensions are truncated (disregarded).

Dimensional adjustment of the specified operand permits a scalar value, for example zero, to be appropriately replicated for assignment to each of the elements of an array.

4. After any dimensional adjustment, elements of the variable are assigned the values of identically indexed elements of the specified operand. Where an element of either the variable or the specified operand has an index that is not the same as the index of any element in the other, however, no assignment involving that element is made.

Assignments between scalar operands--including multicharacter texts--obey the following, additional rules:

5. Assignment is only done between pairs of numeric operands, pairs of textual operands, pairs of pointer operands, or pairs of Boolean operands--allowing, however, for the equivalences between Boolean values and binary textual values, and between binary, octal, decimal, and hexadecimal textual values and integer numeric values.
6. Where necessary, scalar numeric values are automatically converted during assignment to the mode and representation, and are rounded or truncated to the precision, of the variable to which they are being assigned. Truncation of "overflow digits" (i.e., most significant integer digits) is done as a last resort. Assigning a negative value to an unsigned variable is equivalent to assigning an absolute value.

7. Where necessary, scalar textual values are considered to be automatically converted during assignment to their highest common code. Where either or both scalar textual operands in an assignment are multi-character texts, assignment occurs exactly as if between vectors of one-character texts, except that considerations of right or left justification are applied first.
8. Assignments between scalar pointer values and scalar Boolean values are straightforward.

4.1.1 Formulas

A formula specifies a value and is, in effect, a computing rule for obtaining that value. A formula may contain variables and so the value it specifies, in general, is dynamically dependent on these variables, as will be described.

A formula results in a single value which is designated by a combination of variables, constants, arithmetic or logical operators, and grouping brackets. Hence, variables and constants are also formulas. The same characteristics apply to both formulas and variables.

SYNTAX:

```
formula ::= numeric-formula | textual-formula | pointer-formula |
          boolean-formula | (formula)
```

As a formula, the value specified by a variable is, of course, that which it designates. A function specifies the value computed by a subroutine, and a constant specifies the value it denotes. Formulas may also be enclosed in parentheses to alter or emphasize the sequence in which the operations on formulas are performed. In addition, any compiler-dependent hardware operand may serve as a formula to specify a value.

A formula may specify values containing a variety of data types. The rules for the evaluation of such formulas are given later. Formulas that specify values with all numeric components are classed as numeric formulas, however. Likewise for textual, pointer, and Boolean values.

4.1.1.1 Numeric Formulas

A numeric formula specifies a scalar or nonscalar numeric value computed from the values expressed by its individual operands, which are themselves numeric formulas (e.g., variables and functions of numeric type, numeric constants, etc.). The arithmetic operators +, -, *, /, and ** have the conventional meanings of addition, subtraction or negation, multiplication, division and exponentiation. Enclosing an arithmetic operator in parentheses converts it from a binary to an n-ary operator. Double arithmetic operators are useful in specifying matrix operations.

As in algebra, division by zero (and the equivalent raising of zero to a negative power) is undefined. Mixed and fractional exponents are allowed, as are exponentiations.

SYNTAX

numeric-formula ::= constant | function | variable | hardware-operand
 Δ <nothing | arithmetic-operator> Δ <nothing |
 numeric-formula> | n-ary-arithmetic-operator Δ numeric-
 formula | numeric-formula Δ matrix-operator Δ
 numeric-formula | boolean-formula

n-ary arithmetic-operator ::= (arithmetic-operator)

matrix-operator ::= arithmetic-operator₁ . arithmetic-operator₂

The meaning of double arithmetic or matrix operator is explained later in this section.

Examples:

27

(T-1)

A sub 0 = Q

(ALPHA sub (T-2)) /1.889E-6

(A, B, C)**(2,3,4) ≡ (A**2, B**3, C**4)

Parentheses may, of course, be used freely in constructing numeric formulas of arbitrary complexity, according to the notational conventions of ordinary algebra. Arithmetic operations in a numeric formula are generally performed in sequence, from left to right, except that parenthesized operations are performed first, and then operations are performed in the following order of precedence:

- a. n-ary arithmetic operations
- b. matrix operations
- c. exponentiation, unary addition and negation (these are performed in sequence from right to left, in exception to the general rule stated above)
- d. multiplication and division
- e. addition and subtraction

Arithmetic operations involving any pair of numeric operands are defined in SPL (except for division by zero, of course), even though they may differ in dimension (scalar vs. nonscalar values), in representation (fixed-point vs. floating-point values), and in other subsidiary attributes. To achieve compatibility in dimension and representation between operands, where this may be necessary, the following conversions are automatically applied:

- a. Binary (including Boolean 1-bit texts) octal, decimal, and hexadecimal texts are converted to unsigned integer values according to the obvious positional notation. For example: bit'0011011110' \equiv 222. The empty text (zero characters), as denoted by null, is converted to the value zero.
- b. Integer and fixed-point values are converted to floating-point representation.
- c. Scalar values are (in effect) converted to nonscalar values by assuming appropriate replications of the scalar. Similarly, nonscalar values of lower dimension are converted by replication to nonscalar values of higher dimension. For example: $(A,B,C//D,E,F//G,H,I)+1 \equiv (A+1,B+1,C+1//D+1,E+1,F+1//G+1,H+1,I+1)$ and $(A,B,C//D,E,F//G,H,I)*(1,2,3) \equiv (A*1,B*2,C*3//D*1,E*2,F*3//G*1,H*2,I*3)$.
- d. Arithmetic operations involving nonscalar operands of similar dimension but different size (number of elements) are done by truncating (i.e., disregarding) the excess elements (of either operand) in any of the dimensions. For example: $(A,B,C)+(1,2) \equiv (A+1,B+2)$, and $(A,B,C//D,E,F)/(1,2//3,4//5,6) \equiv (A/1,B/2//D/3,E/4)$. In other words, arithmetic operations are only done on operand pairs with identical indexes.

Arithmetic operations on real operands are done according to the following scaling rules, not all of which apply in any given case. The following notation is used. Note, this notation is in part SPL notation and standard mathematical notation.

$N_1 \equiv$ number of integer digits*

$N_2 \equiv$ number of fraction digits

$N_3 \equiv$ number of significant digits

$N_4 \equiv$ number of exponent digits

$W \equiv$ maximum number of digits in an integer or fixed-point
valued operand**

Where N_1 , N_2 , and W apply to integer and fixed-point valued operands, and
 N_3 and N_4 to floating-point valued operands.

1. Fixed-point addition and subtraction:

$$C = A + B, C = A - B.$$

$$N_1(C) = 1 + \max [N_1(A), N_1(B)]$$

$$N_2(C) = \text{if } N_2(A) \text{ eq } N_2(B) \text{ or } N_2(B) \text{ eq } 0 \text{ then } N_2(A) \text{ else}$$
$$\text{if } N_2(A) \text{ eq } 0 \text{ then } N_2(B)$$
$$\text{else } 1 + \min [N_2(A), N_2(B)]$$

2. Fixed-point multiplication: $C = A * B$.

$$N_1(C) = N_1(A) + N_1(B)$$

$$N_2(C) = N_2(A) + N_2(B)$$

*It is assumed in these equations that $N_i(A) \equiv$ number of digits in A of
type i.

**This is an implementation constraint, but should not be less than 10 decimal
or 32 binary digits.

3. Fixed-point division: $C = A/B$

$$N_1(C) = N_1(A) + N_2(B)$$

$$N_2(C) = \text{if } N_2(A) \text{ eq } 0 \text{ and } N_2(B) \\ \text{eq } 0 \text{ then } W-1-N_1(A) \text{ else} \\ \text{if } N_2(A) \text{ eq } 0 \text{ and } N_2(B) \\ \text{ng } 0 \text{ then } 2*N_1(B)+N_2(B)-1 \\ \text{else if } N_2(A) \text{ ng } 0 \text{ then} \\ N_1(B) + N_2(A)$$

4. Fixed-point exponentiation: $C = A**B$.

Exponentiation is done in floating-point, unless: The base A is an integer or fixed-point value; and the exponent B is an integer value; and the greatest possible magnitude of the exponent, times the number of (integer and fraction) digits in the base, is less than W. If this condition is satisfied, the scaling rules for fixed-point multiplication and division apply--as many as W-2 multiplications followed by, at most, one division.

$$N_1(C) = [N_1(A)+1]*B-1$$

$$N_2(C) = N_2(A)*B$$

5. Floating-point arithmetic operations.

$$N_3(C) = \max [N_3(A), N_3(B)]$$

$$N_4(C) = \max [N_4(A), N_4(B)]$$

Intermediate integer and fixed-point results are represented by W digits. Where the number of digits (determined above) exceeds W, excessive digits are truncated. Truncation is done first on least significant fraction digits then, if necessary, on most significant integer digits.

In defining the arithmetic operators as n-ary rather than binary operators, there are three cases to consider: where the operand (A, for example) specifies a scalar numeric value; where it specifies a vector (one-dimensional array); and where it specifies a multidimensional array. The result of an n-ary arithmetic operation (C, for example) is defined in the following table:

	<u>Scalar A</u>	<u>Vector A</u>	<u>Multidimensional A</u>
$C \equiv (+)A$	$C \equiv A$	$C \equiv (A_1 + A_2 + \dots + A_n)$	$C_1 \equiv (+)A_1, \dots, C_n \equiv (+)A_n$
$C \equiv (-)A$	$C \equiv -A$	$C \equiv (A_1 - A_2 - \dots - A_n)$	$C_1 \equiv (-)A_1, \dots, C_n \equiv (-)A_n$
$C \equiv (*)A$	$C \equiv A$	$C \equiv (A_1 * A_2 * \dots * A_n)$	$C_1 \equiv (*)A_1, \dots, C_n \equiv (*)A_n$
$C \equiv (/)A$	$C \equiv A$	$C \equiv (A_1 / A_2 / \dots / A_n)$	$C_1 \equiv (/)A_1, \dots, C_n \equiv (/)A_n$
$C \equiv (**)A$	$C \equiv A$	$C \equiv (A_1 ** A_2 ** \dots ** A_n)$	$C_1 \equiv (**)A_1, \dots, C_n \equiv (**)A_n$

The meaning of the matrix operators is defined, in general, as follows. Given two numeric arrays X and Y declared

array X. begin ... end m by p by ...

array Y. begin ... end q by n by ...

and any two arithmetic operators, op_1 and op_2 , then the result, C, of $X \cdot op_1 \cdot op_2 \cdot Y$ may be defined by defining the elements of C.

$$\begin{aligned}
 C_{\text{sub}}(1,1) &\equiv (op_1) (X_{\text{sub}} 1 \ op_2 \ Y_{\text{sub}} (1 \ \text{to} \ q, 1)) \\
 &\vdots \\
 C_{\text{sub}}(i,j) &\equiv (op_1) (X_{\text{sub}} i \ op_2 \ Y_{\text{sub}} (1 \ \text{to} \ q, j)) \\
 &\vdots \\
 C_{\text{sub}}(m,n) &\equiv (op_1) (X_{\text{sub}} m \ op_2 \ Y_{\text{sub}} (1 \ \text{to} \ q, n))
 \end{aligned}$$

In particular, where X and Y are declared, for example,

array X. real M by P

array Y. real P by N

then $X * Y$ is the familiar operation of matrix multiplication.

4.1.1.2 Textual Formulas

A textual formula specifies a scalar or nonscalar textual value computed from the values expressed by its individual operands, which are themselves textual formulas--textual variables and functions, textual constants, etc. No special operators are provided in SPL just for text processing; instead the generally applicable operations of subscription, catenation, etc., are used. Boolean formulas specify values that, textually, are 1-character binary texts.

SYNTAX

textual-formula ::= textual-constant Δ <catenation-operator | subscription-operator>

Examples:

'THOMAS ROMANOV' sub (9 to 11) \equiv 'OMA'
('T' 'H' 'OMA' 'S') sub 3 \equiv 'OMA'

4.1.1.3 Pointer Formulas

A pointer formula specifies a scalar or nonscalar pointer value computed from the values expressed by its individual operands, which are themselves pointer formulas--pointer variables and functions, pointer constants, etc.

Intermediate floating-point results are carried in $N_3(C)$ significant digits. Unless a floating-point result is for use in computing a value to be assigned to a truncated real item, it is, in effect, first calculated to $N_3(C)+1$ significant digits and then rounded (away from zero) by adding one to this least significant digit and then discarding it and renormalizing if necessary.

SYNTAX

pointer-formula ::= name Δ < sub Δ index-string | nothing > | cell Δ sub Δ
index-string | cell Δ variable | bit Δ sub Δ

Examples: index-string

STEP sub J

cell SORT

STEP5

cell sub 3

bit sub N

The i th cell in memory may be specified by cell sub i ; the origin cell for a variable V may be specified by cell V . (Where cell _{c_1} , cell _{c_2} , and

cell _{c_n} contain a value, the value's origin cell is cell _{c_i} , where $i \leq j$, for

$j = 1, 2, \dots, n$. The pointer formula cell V , for any variable V , therefore always specifies a scalar pointer value.

4.1.1.4 Boolean and Relational Formulas

A Boolean formula specifies a scalar or nonscalar Boolean value computed from the values expressed by its individual operands, which are themselves Boolean formulas--Boolean variables, functions, and constants, and relational formulas--and binary textual formulas. The conventional logical operators and, or, equiv (equivalence), and not are available, as are the relational operators eq (equals), neq (is not equal to), ls (is less than), gr (is greater than), lq (is less than or equal to), and gq (is greater than or equal to). N-ary logical and relational operators are also available. Logical and relational operations on nonscalar operands are done on an element-by element basis. A scalar Boolean formula is more conveniently called a condition, and a binary text may be considered a vector of Boolean values.

SYNTAX

condition ::= boolean-formula

boolean-formula ::= boolean-constant | not Δ boolean-formula | boolean-formula Δ <and | or | equiv> Δ boolean-formula | relational-formula | n-ary-logical-operator Δ boolean-formula

relational-formula ::= <nothing | n-ary-relational-operator> Δ formula
Δ <nothing | relational-operator Δ relational-formula>

n-ary-logical-operator ::= (<and | or | equiv>)

n-ary-relational-operator ::= (relational operator)

Examples:

not T

not T and (B equiv C or T)

A eq Q or not T

bit ' 1 ' ≡ true

false, not T, V sub I to V sub (I+1 to J)

A eq Q lq D nq I

Parentheses may, of course, be used freely in constructing Boolean formulas of arbitrary complexity. Logical operations in a Boolean formula are performed in sequence, from left to right, except that parenthesized operations are done first, and then operations are done in the following order of precedence:

1. n-ary-relational operations
2. relational operations
3. n-ary-logical operations
4. not
5. and and or
6. equiv

Relational operators are defined primarily on pairs of scalar values. If the indicated relation holds, the operation specifies the Boolean value true, otherwise it specifies the value false. A relational formula involving scalar operands and several relational operators specifies a Boolean vector, e.g., $A \text{ eq } B \text{ eq } C \text{ nq } D \equiv A \text{ eq } B, B \text{ eq } C, C \text{ nq } D$. Relational operations on nonscalar operands are done on an element-by-element basis, yielding a nonscalar Boolean value, and where it is necessary to achieve dimensional compatibility, the dimensional conversions described in Section 4.1 on numeric formulas are automatically applied.

In comparing numeric operands, where it is necessary to achieve compatibility in dimension, mode and representation, the conversions described in Section 4.1.1 on numeric formulas are automatically applied. However, only the relations equal and not equal are defined when either operand is complex. In addition, in comparing a pair of integer or fixed-point values A and B, the comparison is only carried out to [if $N_2(A) \text{ ls } 0$ and $N_2(B) \text{ ls } 0$ then min ($N_2(A), N_2(B)$) else max ($0, \text{min}(N_2(A), N_2(B))$)] fraction digits (where $N_2(X) \equiv$ the number of fraction digits of X.) And in comparing a pair of floating point values Y and Z, the comparison is only carried out to [min ($N_3(Y), N_3(Z)$)] significant digits (where $N(X) \equiv$ the number of significant digits of X). In comparing scalar textual³ operands of the same length in the same alphabet, the shorter text is left or right justified and filler characters are appended. Then, for textual operands of the same length in the same alphabet, comparison involves the pair-by-pair comparison of characters according to the alphabet's collating sequence.

Scalar pointer values are compared according to the following ascending order: null.* cell sub 1, cell sub 2, cell sub 3, and so on.

Boolean values may also be relationally compared, according to the following ascending order: false, true.

* See section 7.1 for a definition of null.

Mixed nonscalar operands may be compared, although for scalar operands the relational operators are only defined for pairs of numeric values, pairs of textual values, pairs of pointer values, and pairs of Boolean values-- exceptions are due to the equivalence between Boolean and 1-character binary values and between binary, octal, decimal, and hexadecimal texts and integer numeric values.

Logical operators are defined primarily on scalar Boolean values. Logical operations on nonscalar Boolean operands are done on an element-by-element basis, yielding a nonscalar Boolean value, and where it is necessary to achieve dimensional compatibility, the dimensional conversions described in Section 4.1 on numeric formulas are automatically applied. N-ary logical operators are defined in the same manner as n-ary arithmetic operators; that is, (and) (eq), $(A, B, C, D) \equiv (A \text{ eq } B, \text{ and } B \text{ eq } C, \text{ and } C \text{ eq } D)$. The n-ary and operator may be omitted where it is clear from context that a scalar rather than a nonscalar Boolean value is appropriate, for example, where a nonscalar Boolean value is assigned to a Boolean item, or is used as a condition.

4.1.2 Direct Code

SPL provides for direct code statements, however they must be preceded by a \$. A machine instruction or hardware operator followed by a parameter string may be used in any SPL form in which a simple statement or compound statement is allowed.

SYNTAX

direct-code-statement ::= \$name_{of-hardware-operator} Δ (actual-parameter-string)

Examples:

\$STO (\$P, cell (Q))

\$LDA (Q, \$IX.1)

\$BRU (STEP sub (ALPHA-I), 'I')

4.2 CONTROL STATEMENTS

Control statements are provided in SPL to: transfer control to a specified statement; execute one statement or another, or none, depending on a specified condition; execute a statement repeatedly, perhaps each time with different values for designated variables; initiate an asynchronous process; delay execution of a statement until a specified condition is evaluated to be true; stop a process; execute a statement whenever a specified condition is evaluated to be true; and call a procedure subroutine.

SYNTAX

control-statement ::= simple-control-statement | complex-control-statement

simple-control-statement ::= transfer-statement | stop-statement | procedure-call-statement

complex-control-statement ::= repeated-statement | conditional-statement |
parallel-statement | delayed-statement |
chronic-statement

Procedure call statements are discussed in Section 5.1.1.

4.2.1 Transfer Statements

Transfer statements break the normal, written sequence of statement executions by transferring control to the statement whose origin cell is specified by a pointer formula.

SYNTAX

transfer-statement ::= <go Δ to | goto> Δ pointer-formula

Examples:

go to COMPUTE

go to STEP (I)

go to cell (oct '150000')

goto cell ALPHA

In the case of a transfer-switch the variable at the end of the statement controls the switch direction.

It should be remembered that pointer formulas can specify the origin cells of data elements as well as statements, so it is the programmer's responsibility to see that execution control does not get transferred, for example, to an array of floating-point numbers. However, the ability to transfer execution control to what is nominally a data element is an occasionally desirable, therefore not prohibited, action--for example, when it is desired to execute a machine-language program text that has just been read.

4.2.2 Repeated Statements

SPL provides for the repeated execution of statements, either a specified number of times, or where the number of repetitions depends on some condition--perhaps each time with different values for a designated (control) variable.

SYNTAX:

repeated-statement ::= for Δ repetition-clause Δ statement

repetition-clause ::= variable Δ = Δ value-sequence

value-sequence ::= formula | numeric-formula₁ Δ by Δ numeric-formula₂ Δ
<while | until Δ condition

Note: A chronic statement may not be repeatedly executed.

Examples:

```
for I = 1 by I until I gr 100, PRINT I  
for I = 1 by 2 while I ls 99, for J = I by 3 until J gr  
I+99, A sub J = I
```

A sequence of one or more values to be assigned the control variable may be given by a formula specifying an array of dimension equal to or one greater than that of the control variable. Or, where the control variable is numeric in type, a sequence may be given by assigning it an initial value (for that sequence) as specified by numeric-formula₁, with subsequent values being determined by the addition of an increment value, as determined by numeric-formula₂. In this latter case, the sequence of assignments continues while or until the given condition--which usually involves the control variable--specifies the value true. Since the evaluation of the condition is done prior to each assignment in this sequence, zero or more assignments may thus be specified.

The statement in a repeated statement is repeatedly executed, zero or more times. The repetition clause determines the number of such executions.

In a repeated statement, the repeatedly executed statement may, of course, reference and even alter the value of the control variable.

Any transfer of control into a repeatedly executed statement from outside will generally produce undefined results. Furthermore, while it is possible to terminate the repeated execution of the statement by a transfer of control to outside the repeated statement, the value of the control variable remains defined as of its last setting.

4.2.3 Conditional Statements

A conditional statement expresses the action of deciding to execute one statement or another from a pair of statements, or of deciding to execute or skip a single statement.

SYNTAX:

conditional-statement ::= if Δ condition Δ then Δ statement₁ <nothing | Δ else
 Δ statement₂>

Note: A chronic statement may not be part of a conditional statement. Aside from that, statement₂ may be any statement while statement₁, may not be a conditional statement.

Examples:

if HOURS sub EMPLOYEE nq 0, then COMPUTE begin ... end

if T then A = 0 else B = 1

if A ls 0 or T then go to NEXT else if A gr 0 and not T then go to ALPHA

A conditional statement causes the execution of one of two alternative statement--if the condition is true, then statement₁ is executed, else, statement₂ is executed.

4.2.4 Parallel Statements

A parallel statement is a complex control statement that serves to cause the parallel or asynchronous execution of its component statement (where the implementing system will support this kind of operation). The body (incorporated statement) of a parallel statement may be executed in parallel or asynchronously with the subsequently written statements.

SYNTAX:

parallel-statement ::= do Δ statement

Note: A chronic statement may not be part of a parallel statement.

Examples:

do begin T = false, A = B, T = true end

do read RECORD = SENSOR

do go to START

A parallel statement ordinarily serves only to establish a new task--a temporary, parallel or asynchronous sequence of execution--which ends when the incorporated statement completes its execution. However, a parallel statement can also establish a new process--which is a more permanent, parallel or asynchronous sequence of execution--merely by executing a transfer of control out of the parallel statement, thus bypassing the implicitly built-in stop at the end of the parallel statement. The difference between a parallel task and a parallel process is mainly one of subjective convenience.

A transfer of control into a parallel statement from outside does not establish a parallel task or process, however. With reference to either of the previous pair of examples, go to THIS will cause the execution of statement₁ followed by the execution of the (implicit or explicit) stop statement.

When a statement that is normally executed in parallel--such as an input-output statement--is written without a do indicating that it is not to be executed in parallel, the next statement is automatically delayed until the execution of its predecessor is completed.

4.2.5 Stop Statements

A stop statement serves to halt a (main or parallel) process. It signifies the completion of the statement sequence in which it is executed. A stop statement may be conditionally invoked.

SYNTAX:

stop-statement ::= stop

Examples:

stop

if A eq Q then stop

A stop statement also has the incidental effect of closing any files left open by the process or task in which it is executed. (see Section 4.3 on opening and closing files).

4.2.6 Delayed Statements

Delayed statements cause conditional delays in a process or task. The execution of a statement may be thus delayed until any specified condition occurs.

SYNTAX:

delayed-statement ::= when Δ condition Δ statement

Note: A chronic statement may not be part of a delayed statement.

Examples:

when T gr 4, set A = I

when \$TCS "Teletype Channel Status" eq 'FREE' write TELETYPE = MESSAGE

The condition for execution of a delayed statement is specified, of course, by a (scalar) Boolean formula. The delayed statement is repeatedly executed until its enabling condition is evaluated as true. Delayed statements are similar in this respect to chronic statements, (although a delayed statement is not automatically re-executed upon reoccurrence of its enabling condition), and they are often used to synchronize parallel tasks and processes.

An example of this involving four parallel assignments, is shown below.

```

T1, T2, T3 = false
do begin A = B, T1 = true end
do begin C = D, T2 = true end
do begin E = F, T3 = true end
set G = H when T1 and T2 and T3 EQ true

```

4.2.7 Chronic Statements

Chronic statements cause the execution of any given statement whenever a specified condition occurs or re-occurs. Chronic statements are useful for interrupt processing, priority processing, and parallel processing. They are executed asynchronously--depending on the hardware resources--either in parallel with, or by interrupting, some current process or task without, however, affecting that process or task beyond perhaps delaying it or explicitly altering its data.

SYNTAX:

chronic-statement ::= on Δ condition Δ statement

Examples:

```

on $ETC "Elapsed Time Counter" gr 500 "milli-seconds" begin ...
end

on $FPO "Floating-Point Overflow" go to ABORT
PROCESS. on SIGNAL gr 0 and SIGNAL gr CURRENT begin ... end
on $DS "DEVICE STATUS" eq 'DONE' go to CONTINUATION sub I

```

The condition for executing a chronic statement is specified by a (scalar) Boolean formula, which is automatically evaluated whenever any of its operands is assigned, or acquires, a new value. The chronically-executed statement is executed as a parallel task if a processor is available or, if all are busy, as a primary task--by interrupting some current process or task. Thus, a chronically-executed statement when its "time has come," it has paramount claim to processing. The programmer, however, has complete control over specifying the condition under which a chronic statement is executed, and in particular, conditions may be specified so that a chronic statement is not executed until the facilities it needs are available, so that a low-priority chronic statement does not interrupt a high-priority task or process, so that a high-priority chronic statement does interrupt a low-priority task or process, and so that the automatic evaluation of the condition occurs no more frequently than, and indeed, exactly when, desired.

Chronic statements are executed only on the occurrence or re-occurrence of their enabling condition. They are not part of the "normal" sequence of statement execution, which is why they are inappropriate components of other control statements. Aside from this restriction (which is not a syntactic necessity but is meant primarily to prohibit confusing statement constructions), chronic statements may be written wherever convenient in the program; the "normal" sequence of statement executions will automatically bypass them. And while a chronic statement may not be transferred to, (the automatic bypass would frustrate this), its component statement can be transferred to.

4.3 INPUT-OUTPUT AND FILES

Many data storage devices impose accessing restrictions in that reading or writing an individual value may, for efficiency, ordinarily involve the transfer of an entire block of data. Such devices are called external storage devices,

as contrasted with the internal memory of the computer. To allow a description of reasonably efficient input-output operations, therefore, data entering or leaving the computer's internal memory are organized into files. A file is thus a body of data contained in some external storage device, such as punched cards or tape, or magnetic tape, discs, or drums.

To provide maximum flexibility for real-time computation, the input and output features of SPL place major emphasis on the activities of reading and writing and little emphasis on data manipulation and conversion, for which adequate facilities are otherwise provided. (In particular, the operations of encoding and decoding a record according to a given format, though described in the SPL extensions, have been removed from the operations of reading and writing so that they may be applied to operands other than records.) Data conversion and record buffering and blocking, where they are needed, must therefore be explicitly specified either in the program or in library subroutines.

In SPL, files are defined by file declarations and processed by the input-output operations of opening and closing, positioning and testing,* and reading and writing a file.

SYNTAX:

```
input-output-statement ::= open-statement | close-statement | read-statement |  
                           write-statement
```

*Positioning and testing are input-output operations that involve functions, rather than statements.

4.3.1 File Declarations

Files, which are collections of data that are externally stored or available, or input or output, are considered in SPL to be strings of records, each record distinguished by its position in the file. In turn, a record is considered to be a linear array of texts, called lines. A file declaration gives the dimensions of the file, names its alphabet and the device and module used to access and hold it, and provides several other file attributes, some of them implementation-defined. In certain cases, several different but similarly described files may be declared at once.

SYNTAX

file-declaration ::= file Δ name-string Δ file-description

file-description ::= device-name Δ <nothing | \$(character-string)> Δ <nothing | dimension-string> Δ <nothing | code-name>

device-name ::= name. | device-name. <nothing | number> Δ <nothing | module-name. number>

code-name ::= <bin | oct | dec | hex | text>

- Notes:
- a. The character string may not contain the ")" close parenthesis.
 - b. With regard to the dimension string, a file is a three-dimension entity: records per file, lines per record, and characters per line.

Examples:

file A. SITE

file D. TAPE.07 REEL.3661

file E. TTY.14 STATION.71 \$(213-3993411) I by 1 by J

file F. TAPE, K by L by 32 bin

In a file declaration, the device name indicates the type of peripheral device used to access the file. For systems with several units of the indicated type, the suffix numbers tell which units shall be used to access the file, and may be omitted if any unit of the indicated type is acceptable. Device names and the interpretation of suffix numbers are implementation defined. However, they should account for cases where a device is used to access several files, and where several devices are used to access a single file.

A given type of device may imply any or all of the other attributes of a file, in which case these attributes may be omitted from the declaration, or it may place limits on them--for example, a printer that cannot produce lines longer than 132 characters.

The module name in a file declaration indicates the particular storage module --tape reel, card deck, disc pack, type of preprinted form, display area, etc. --used to access the file. Like device names, module names usually include module type and suffixed serial number, and are implementation defined. And they should also account for cases where a (physical) module contains several files, and where several modules are needed to contain a single file. Module name may be omitted if the identification of a module is unimportant or irrelevant to the device, or handled outside the system, e.g., manually.

Any implementation-defined, machine- or system-dependent file attributes may be declared within the \$(and) brackets. Some examples might be: password; work order number, special labeling instructions; source or destination for the file; author; expected activity; security classification; purge date.

The three-component dimension string gives the dimensions of the file: number of records per file, number of lines per record, and number of characters per line. Where device and module permit, any of the dimensions may be given, by an integer-valued numeric item, as varying. However, number of characters per line may only be considered as varying between records, not within a record. And where number of records is given by a number, this is taken as a maximum value.

The code name indicates the alphabet in which the file is coded. The alphabet must, of course, be defined. And it may contain both nonprinting control characters, and graphic characters.

Where the file description is not so specific as to be pertinent to one and only one file, it may be used in declaring several files at once, and in a mode declaration.

4.3.2 Assign Declaration

The distinction can be made in SPL between files which represent logical units and the actual physical units to which a file is assigned. This logical unit/physical unit equation is done by means of an assign declaration. What physical units are available is, of course, implementation dependent.

SYNTAX

assign-declaration ::= assign Δ name to Δ device-name

Examples:

assign MASTER to TAPE 1

assign LOG to PRINTER

assign OUTPUT to PUNCH

4.3.3 Opening and Closing Files

A file may be opened with an open statement, which designates the file and completes or overrides the catalog of the file's declared attributes. A file may be closed with a close statement, which designates the file and indicates whether an end-of-file is to be written, or with a stop statement.

SYNTAX

open-statement ::= open Δ device-name Δ <nothing | \$(character-string) Δ <nothing | dimension-string> Δ <nothing | code-name> Δ file-designation

close-statement ::= close Δ <nothing | out | out Δ module-name> Δ file-designation

file-designation ::= name_{of-file} | file Δ at Δ pointer-formula

Notes:

- The character-string may not contain the ")" close parenthesis.
- The elements in an open statement between the primitive, open, and the file designation, may be written in any order.

Examples:

open A

close A

open TAPE.08

close out REEL D

open file at P sub I

close out file at P sub I

An open statement supplies missing or overriding file attributes, which are taken to hold until the file is closed. Opening a file may cause manual accessing and mounting of the indicated module. And where the module is of a type that cannot be accessed by the indicated device, it may cause loading of the file into the appropriate external storage medium. Opening a file does not alter the position of the file, should it have previously been opened, accessed, and closed. Otherwise, an open statement will automatically position the file to the first record. Other implementation-defined actions, such as label checking, may result from opening a file.

Closing a file releases the device used to access the file, but it does not alter the position of the file should it be subsequently opened. Closing out a file causes an end-of-file to be written and also releases the device, but it leaves the position of the file undefined. In addition, in closing out a

file, a new module name may be given. This new module name may merely be substituted for the old one, or where the module type cannot be accessed by the file accessing device, the file is unloaded from the external storage medium containing it, onto the module.

4.3.4 Testing and Positioning Files

The testing and positioning of files in SPL is done with non-input-output statements employing a pair of built-in functions, status and position. For addressable files, the position function serves as a functional variable.

SYNTAX:

function ::= file-designation Δ <status | position>
functional-variable ::= file-designation Δ position

Examples:

A status

(file at P sub I) position

The file-status function specifies a one-character textual value in an implementation defined code that may differ from device-type to device-type. Regardless of these differences, certain codes are established with standard meanings for all devices and implementations. These are:

1. 'READY' the device has transmitted a record or is ready to transmit a record
2. 'BUSY' the device is in the process of transmitting a record
3. 'EOF' an end-of-file has been encountered by the device
4. 'FULL' another write operation would cause the file to exceed the capacity of the module or modules allocated to it
5. 'ERROR' the device is unsuccessful in transmitting a record due to an error which cannot be corrected
6. 'UNREADY' the storage device is not ready or unavailable, or the module has not been mounted
7. 'CLOSED' the file has been closed, or not opened

With these codes, and any implementation-defined, nonstandard ones, the status of a file may be determined with such Boolean formulas as: A status eq 'BUSY'. Appropriate hardware operands may also be referenced for more specific status information in many cases.

So far as position is concerned, an SPL file is self-indexing, meaning that the record available for transfer to or from the file depends on the file's current position. The records of an n-record file have the positions 1 to n, and the position of the record currently available for transfer is specified by the position function. The transfer of a record to or from a file automatically increments (or decrements, for a reverse file) by one, the file's position. Furthermore, where the storage or input-output device allows, the position function designates a scalar, unsigned integer variable that may be altered by the assignment of an arbitrary numeric value, thus repositioning the file. Such a file is an addressable file, as opposed to a serial file, where such a general positioning operation is to be avoided as impossible or inefficient. Some serial files do, however, permit restricted forms of the positioning operation. For a tape file T, for example, it might be possible to specify rewind by "T position =1," backspace by "T position = T position -1," and skip N records by "T position = T position +N."

4.3.5 Reading and Writing Records

Reading and writing a 'READY' file is done in SPL by read and write statements, wherein the programmer designates the file and designates or specifies the data elements to receive or provide the record.

SYNTAX:

read-statement ::= read Δ variable Δ into Δ file-designation

write-statement ::= write Δ file-designation Δ from Δ textual-formula

Note: In a read statement, the variable must be textual in type.

Examples:

read BUFFER into A

write F from BUFFER

write E from 'THE QUICK BROWN FOX...TESTING'

do read bit (ALPHA sub (J to K)) into C

Read and write work precisely according to the rules of the assignment statement, where the record is one of the operands and is considered to be a row vector of texts (lines) of dimension and alphabet declared (or given in the open statement) for the file. Where the number of lines per record and/or the number of characters per line are declared, with an integer item, as variable, read and write have the effect of assigning the appropriate value to these items. Read and write also advance by one the position of the file-- either forward or, for a reverse file, backward. In general, a file may be both written and read.

5. PROCEDURES AND FUNCTIONS

5.1 PROCEDURE DECLARATION

The procedure is a type of closed subroutine that may be classified as program independent because it can operate upon data independent of item names and their definitions as defined in the main program or in the compool. This is accomplished by the use of formal data declarations (dummy data) defined in the procedure. The data to be operated upon (parameters) are transmitted from the calling routine to the procedure via the procedure call. During the operation of the procedure, the data transmitted from the calling routine are referenced by the formal data declarations defined in the procedure. General purpose routines may now be generated which enable the many programs in a system to centralize their common routines within procedures and call upon each one when needed.

Thus, a procedure declaration sets up a closed subroutine that may have input parameters, output parameters, or both. A procedure declaration is independent of outside loop statements; it may be invoked from within any loop statement in the main program or in other processing declarations without deactivating the loop variables. On the other hand, the outside loop variables are not defined in the procedure declaration.

SYNTAX

procedure-declaration ::= procedure-heading Δ <nothing | parameter-declaration-string> Δ statement

procedure-heading ::= procedure Δ name_{of-procedure} Δ <nothing | (formal-parameter-string)>

formal-parameter-string ::= formal-parameter Δ <nothing | formal-parameter-string>

formal-parameter ::= name

parameter-declaration-string ::= parameter-declaration Δ <nothing | parameter-declaration-string>

parameter-declaration ::= <item-declaration | array-declaration | group-declaration>
 Δ <nothing | value> Δ <nothing | result> <procedure-
 heading | function-heading> Δ <nothing | parameter-
 declaration-string> | file-declaration

Note: The statement in a procedure declaration may not be a chronic statement.

Examples:

procedure A. begin ... end (A)

procedure SORT. (N, VECTOR) integer N. value real VECTOR. array N result begin
 ... end (SORT)

procedure G. (Y, FCT, X) real Y. function FCT. (Z) Y = FCT (X)

Formal subroutine parameters, like actual parameters, fall into three categories:

(1) Formal value parameters correspond to actual parameters that are values (although designated or specified by variables and formulas). A formal value parameter must be declared in the parameter declaration string as an item, array, or group with the value descriptor appended. Storage is allocated within the subroutine for value parameters, and references to them within the subroutine refers to that storage. (2) Formal expression parameters correspond to actual parameters that are variables or expressions. A formal expression parameter must be declared in the parameter declaration string as an item, array, or group but, of course, without the value descriptor. No storage is allocated within the subroutine for expression parameters. Instead, the subroutine is executed as if the variable or formula constituting the actual parameter were substituted for the formal parameter name throughout the subroutine. (3) Formal name parameters correspond to actual parameters that are names. A formal name parameter may be declared in the parameter declaration string as a file or a subroutine (complete with everything except processing statements). Those not declared are name parameters whose attributes are determined by their use in the subroutine, and by the corresponding actual parameters appearing in the subroutine's various calls. For a formal name parameter, the subroutine is executed as if the name constituting the actual parameter were substituted for the formal parameter name throughout the subroutine.

Formal value and expression parameters are themselves divided into two categories: argument parameters and result parameters. Formal argument parameters correspond to actual parameters whose values are not affected by the execution of the subroutine, while formal result parameters correspond to actual parameters whose values are affected. An actual parameter corresponding to a formal result parameter must therefore be a variable. Formal result parameters are declared with the result descriptor; formal argument parameters are declared without it.

5.1.1 Procedure Call Statements

To execute the computation defined in a procedure declaration, it is necessary to invoke the procedure by executing a procedure call statement, which may be thought of as an abbreviated description of process it invokes.

SYNTAX

```
procedure-call-statement ::= call Δ nameof-procedure
                          Δ <nothing | actual-parameter | (actual-parameter-
                          string)>
```

```
actual-parameter-string ::= actual-parameter Δ <nothing | (actual-parameter-
                          string)>
```

```
actual-parameter ::= variable | formula | name
```

The actual parameters of a procedure call statement or a function must correspond to formal parameters of the subroutine declaration both in number and in sequence. Actual parameters may not, therefore, be omitted. However, a formula or variable as an actual parameter may designate or specify a nonscalar value with a varying number of elements, and it may be necessary to use parentheses to establish the desired correspondence. In addition, an actual parameter must be compatible with its definition and/or use within the subroutine declaration.

Examples:

call B (T)

call G (GAMMA, sin, V*PI)

A procedure subroutine is invoked by a procedure call statement according to the following steps:

- a. Any formal value argument parameters are assigned the values of the corresponding actual parameters.
- b. In effect, the corresponding actual parameters are substituted for any formal expression and name parameters.
- c. The subroutine is executed, and if it completes its operation (i.e., does not stop or transfer control outside the subroutine), the following steps are done;
- d. The values of any formal value result parameters are assigned to the corresponding actual parameters;
- e. Control is returned to the statement following the subroutine call statement.

When a procedure subroutine is invoked by a parallel statement (i.e., "do call"), steps a-b are done in sequence, steps c-e in parallel.

A procedure call statement invoking a hardware operator will follow the same steps, except that the subroutine executed will generally be a single machine instruction. Indeed, with appropriate actual parameters (in many cases, hardware operands), such a procedure call statement may be entirely equivalent to single machine instruction.

5.1.2 Entry and Exit Statements

In order to enter a procedure at some point other than the beginning, the entry statement may be used. One or more entry statements may be defined within a

procedure to define secondary entry points. Like the heading statement of a procedure, each of the entry statements must have a label to serve as the entry name for that point, and each may specify a list of formal parameters which need not be the same as for the procedure or as for other entry points. At the point it is desired to leave a procedure, the exit statement should be invoked. An exit will be generated after the last statement of a procedure but, in order for alternate exits to be taken, an exit statement is required.

SYNTAX

entry-statement ::= name Δ entry <(parameter-declaration) | nothing>

exit-statement ::= exit

Examples:

SEC entry (A, B)

THIRD. entry (C)

exit

5.2 FUNCTION SUBROUTINES

Function subroutines are defined by declaration and invoked by functions. Function declarations are very similar to procedure declarations, except that a function may have parameters preceding the function name, a function has only one result parameter, a value parameter designated by the function name and declared in the function heading, to which a value may be assigned by an abbreviated assignment statement.

SYNTAX

function-declaration ::= function-heading Δ <nothing | parameter-declaration-string> Δ <= Δ formula | complex-statement | compound-statement>

function-heading ::= function Δ <nothing | formal-parameter | (formal-parameter-string)> Δ name of-function Δ <nothing | formal-parameter | (formal-parameter-string)> Δ <item-description | array-description | group-description> Δ <nothing | recursive | reentrant>

Note: The (complex) statement in a function declaration may not be a chronic statement.

Examples:

function RANDOM. real reentrant begin ... end

function F. (A,B) integer recursive

The rules governing procedure declarations also apply to function declarations. In addition, for a function subroutine to compute a functional value, the statement comprising the subroutine body must assign a value to the formal result parameter designated by the function name and also, of course, complete its operation.

5.3 RECURSIVE AND REENTRANT SUBROUTINES

Recursive or reentrant subroutines may be declared in SPL. A recursive subroutine is one that, directly or indirectly, invokes itself. A reentrant subroutine is one that is compiled into "read-only" code, so that it may be invoked by several parallel processes or tasks before it has finished the computation required by a previous invocation, without confusing the data associated with the various invocations.

Both recursive and reentrant subroutines must be explicitly declared as such in the procedure or function heading. A subroutine may not be both recursive and reentrant; in addition, a reentrant subroutine may not involve a non-reentrant subroutine.

5.4 BUILT-IN FUNCTIONS

Basic SPL contains a minimum number of built-in functions. They specify a numeric result.

5.4.1 Trigonometric Function

The trigonometric function sin, cos, and tan specify, for any real scalar argument x expressed in radians, the sine, cosine, and tangent of x . The arctan function specifies, for any real scalar argument x , the arctangent of x in radians.

SYNTAX

function_{of-numeric-type} ::= <sin | cos | tan | arctan> Δ numeric-formula

Example:

sin (tan X)

Complex arguments of these functions are converted to real mode by disregarding their imaginary parts. Nonscalar arguments specify identically-structured nonscalar results.

5.4.2 Absolute Value

The absolute value function specifies the positive value of an integer or real scalar argument, and the positive magnitude of a complex scalar argument.

SYNTAX

function_{of-numeric-type} ::= abs Δ numeric-formula

Example:

abs (I-J)

Nonscalar arguments of the absolute value function specify identically structured nonscalar results.

5.4.3 Base e Exponential.

The base e exponential and logarithm functions specify, for any numeric scalar argument x, the values e^x and $\log_e x$.

SYNTAX:

function_{of-numeric-type} ::= <exp | log.e> Δ numeric-formula

Examples:

exp X

log.e (exp X)

Nonscalar arguments of these functions specify identically-structured nonscalar results.

5.4.4 Base 2 and Base 10 Logarithm.

The base 2 and base 10 logarithm functions specify, for any real scalar argument x, the value $\log_2 x$ and $\log_{10} x$.

SYNTAX:

function_{of-numeric-type} ::= <log.2 | log.10> Δ numeric-formula

Examples:

log.2 (2**X)

log.10 (log.10X)

Complex arguments of these functions are converted to real mode by disregarding their imaginary parts. Nonscalar arguments specify identically-structured nonscalar results.

6. COMMANDS

Every SPL program is composed of declarations which generally describe the form of the data to be operated upon by the program and statements which provide the rules for operating upon that data. Commands are an additional language features which provide the programmer in SPL additional controls in the areas of language definition, editing, debugging, code optimization, time and storage control and program execution. Each one of these categories is described in the sections that follows.

Commands permit the programmer to command the compiler to: translate some portion of the program according to a defined notational extension; execute statements immediately; show (either in typeout or display) any defined data element values; direct the compiler for code optimization; calculate time required for code execution for generated instructions.

SYNTAX

command ::= define-command | execute-command | debug-command | optimization-command | count-command

6.1 DEBUG COMMAND

A debug command serves to print or display (either in the listing produced by the compiler or on a display) any previously provided (input, entered, or executed) lines of program text, or any defined data element value.

SYNTAX

debug-command ::= show Δ <symbolic | nothing> location-identifier Δ <number | nothing> <thru | nothing> Δ <location-identifier | nothing>

location-identifier ::= <name | cell Δ name> Δ <code-name | nothing>

The symbolic attribute indicates if the display of data is to be in the form in which it was declared. The location identifier can be any name defined in the program or compool. The number following indicates the number of cells to be displayed. The display area can be identified by bracketing with names or cell numbers.

6.2 OPTIMIZATION COMMAND

The optimization command serves to indicate the type of code optimization preferred for a portion of the program. The SPL user has some control over the type of code optimization which will be in effect during the compilation of a specified segment of code.

SYNTAX

optimization-command ::= optimize Δ <time | space | none> Δ statement

The primitive time specifies that the statement which follows, simple or compound, is to be optimized for a minimum operating time. The analogous capability for space is indicated if the primitive space precedes the compound statement. If none is specified, this would negate any rearrangement of code for optimization. If the optimized command is not specified, the normal optimization algorithm is used (one which appears optimum for a majority of cases).

6.3 COUNT COMMAND

The count command serves to indicate the execution time for a sequence of code. The statement, whether simple or compound, following a count command is operated with data values declared in the data declarations and the amount of execution time recorded for output.

SYNTAX

count-command ::= count Δ time Δ <statement | declaration>

6.4 DEFINE COMMAND

SPL is an extendable language. Not only may subroutines be defined to extend the language's computational capabilities, but notational extensions may be defined for any other statement or declaration format that is required. These extensions may range from the definition of simple synonyms and abbreviations, to the definition of complicated new data structures and the operations on them.

Indeed, with the subroutine definition capability described in SPL and the notational definition capability described here, it is likely that, in any implementation of SPL, a great many of its features will be implemented either as built-in subroutines, or built-in notation definitions, leaving only a relatively simple language kernel for the compiler to implement.

A notational extension is given in a define command and is applied to the source program text prior to compilation. A define command in effect, serves to translate some new language form into an equivalent SPL representation. This translation is done at the source level in what effectively is a pre-pass over the source language statements if the define command capability is specified by the SPL user.

The define command may be followed by a series of definition rules which apply to the program which follows them, or it may simply specify the name of a set of rules, previously defined, which are brought in by the system from the library and similarly applied.

Notational Definitions can be thought of as having two parts. The first half of the define command begins with the word where and describes the code to be sought out of the source program code for translation into SPL. The second half of the define command begins with the word then and describes the equivalent SPL code for the non-SPL form.

SYNTAX

define-command ::= <name. | nothing> Δ where Δ definition Δ then Δ translation Δ
 <end | define-command-string>

definition ::= textual-formula | term | textual-pattern | not
 Δ definition | definition Δ < or | and > Δ definition

define-command-string ::= begin Δ define-command <nothing | define-command> Δ end

pattern-declaration ::= pattern Δ name_{of-pattern} Δ definition

define-command-call ::= define Δ name_{of-define-command}

textual-pattern ::= pattern < (textual-constant) | (term) | (name-of-pattern) >

translation ::= declaration | statement | command

term ::= character | letter | digit | name | null | constant |
 simple-statement | compound-statement | comment | item-declaration |
 array-declaration | group-declaration | storage-declaration |
 variable | hardware-operand | formula

Examples:

10A. where 'Z' then 'P**2-P+expP' end

ASIGN. where '=' variable ',' constant '+' variable then variable (1) '='
 constant '+' variable (2) end

Note: The term null denotes an empty text--the character string of length zero.

FOR. begin where 'DO' pattern GEO. (statement-name) variable ', ' '=' variable ', ' nothing or variable then 'FOR' variable (1) '=' variable (2) 'BY' if variable (4) then variable (4) else '1' ', ' variable (3) where statement-label and GEO statement then GEO statement 'END' end

In the first example a substitution is made for 'Z' and in the second example an assignment statement of a non-SPL form is mapped into the SPL assignment statement form. In the last example, a FORTRAN "DO" statement is transformed into an SPL "FOR" statement.

A define command may employ just a definition (which may itself be composed of a series of elementary definitions) or, for more elaborate notational extensions, it may employ a definition string, which may contain declarations and other definitions. A definition may contain pattern declarations.

A textual pattern may be specified by enclosing a textual constant or a term in parenthesis. Enclosing a single term indicates the textual pattern represented by the term is being defined as a pattern. The elements of a pattern, then, are alphanumeric textual formulas (and in particular, alphanumeric constants), terms (which are the names used in this report to identify the textual patterns of SPL), and the names of declared patterns.

The logical operators and, or, and not may also be used in specifying patterns, where they have the set-theory meanings of intersection, union, and complementation. Thus, A and B specifies any pattern that is at once both an A pattern and a B pattern. A or B specifies any pattern that is either an A pattern or a B pattern. And not A specifies any pattern except an A pattern.

A define command may involve a definition string that exists only at compile time. The declarations in a definition string establish patterns, data elements, subroutines files, etc. that exist only at compile time--they may not be referenced

by the program at execution time. In addition, the names thus declared are defined only for the definition string in which they are declared--and for any definitions or definition strings appearing within the program text that is affected by the define command containing this definition string.

The most general case is shown below:

```
where begin1 A. ...  
where ... A ...B. ...  
.  
.  
.  
where3 ... A ... B. ... C. ... end1
```

The first define command applies to and contains the next (define) command, which applies to and contains the following statement, which in turn contains a define command. Anything, for example, A, declared in the first definition string may be referenced in any definition or definition string between begin₁ and end₁.

It must also be noted that definitions may contain compound statements, which may naturally contain other declarations. These also establish compile time entities, whose names are defined for the compound statement containing their declarations.

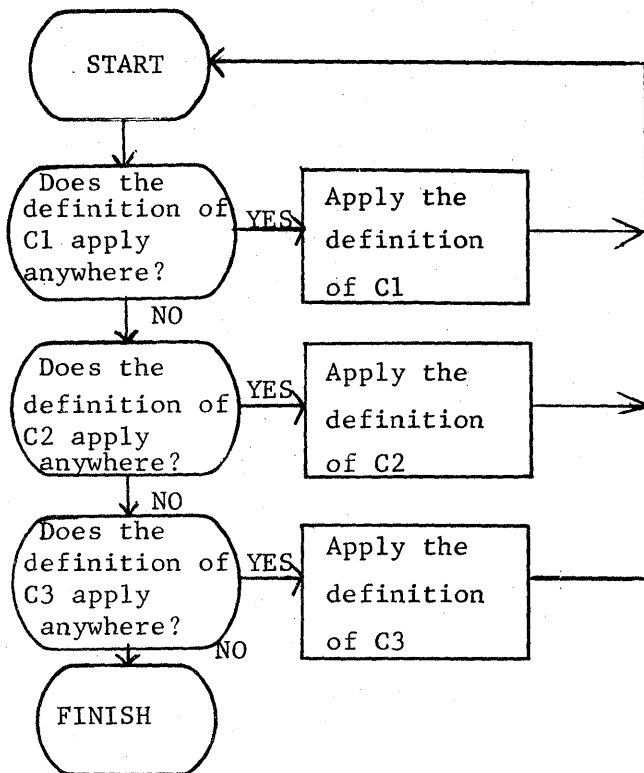
A definition in a define command specifies an alphanumeric textual pattern and indicates some action to be taken by the compiler whenever it encounters that pattern in the program text to which the define command applies. This action may be just a replacement of the encountered matching text by the value of a textual formula, or it may be a more complicated action, as specified by a compound statement, which is executed when the elementary definition is applied.

When used as a textual variable in the textual formula or compound statement of an elementary or single element definition, a pattern name or term refers to the program text to which the define command containing the elementary definition applies. Specifically, for each application of an elementary definition, any term or pattern name it establishes as a textual variable will designate the matching part (or parts) of that segment of the statement, or declaration that matches the pattern given in the elementary definition.

The pertinent rules in applying notational definitions are described in the following rules:

- Rule 1. Where define commands are nested within begin and end brackets, the first define command is applied first, and subsequent nested define commands are applied to text immediately following the symbol of the preceding define command.

- Rule 2. In applying a define command, its definitions are repeatedly applied, in the order given, to the entire program text to which the define command applies--until they are no longer applicable.



Rule 3. In any single application of a definition to a program text, it is applied to the leftmost part that (1) matches the pattern, and (2) is not followed by a part such that the two parts together would also match the pattern.

6.5 EXECUTE COMMAND

An execute command may utilize a definition to identify a set of code which is to be replaced by a value which obtained by executing a formula prior to compilation. An execute command serves to provide a capability to compute compile-time calculated constants. This capability will serve to reduce the size of the stored object program required. The execute command can be thought of as having two parts. The first half begins with where and describes the code to be sought out of the source program for substitution by the calculated value. The second half of the execute command begins with the word execute and can be followed by an item declaration, which is only active for the execute command, and a formula.

SYNTAX

execute-command ::= <name. | nothing> Δ where Δ definition Δ execute Δ item-declaration Δ formula

Examples:

10B. where 'Z' execute item P. integer/4 (P**2P)-4P/P

10C. where 'A**' '+' 'B' execute item C. integer/7
item D. integer/16
C/D * (D+1.4)

An execute command operates at compile time and must result in a single value.

7. EXTENDED SPL

An extension to SPL is described here--for Mission Development Programming and for Support Programming. The defined SPL extension does and any other SPL extension should have as their base "basic SPL". Discussions in this section will assume the existence of basic SPL.

7.1 COMPLEX VALUES

The extended SPL programs may manipulate complex values (i.e., imaginary numeric values). Imaginary constants denote numeric values in the conventional decimal sense.

SYNTAX

numeric-constant ::= imaginary-constant | number | real-constant | binary-constant | octal-constant | decimal-constant | hexadecimal-constant

imaginary-constant ::= <number | real-constant> i

Examples:

2.i

88i

Integer numbers and real and imaginary constants denote numeric values in the conventional, decimal sense.

In extended SPL, a complex item, i.e., one having imaginary parts, may be declared. The description of a complex item applies to both the real and imaginary parts.

SYNTAX

complex-item-description ::= complex Δ number₁ Δ minimum Δ <bit | digit | nothing> Δ <nothing | -> number₂ of-fractional-or-exponent <bit | digit> Δ <nothing | float | fixed> Δ <nothing | signed | unsigned> Δ <nothing | round | truncate>

7.2 CODE DECLARATIONS

A code declaration serves to name and define a coded alphabet. To each character in a declared alphabet corresponds a machine-language code, denoted in the declaration (according to the rules of the assignment statement) by a number or by a textual constant from another alphabet, or deduced by the mechanism of Note b, following. The code string in the declaration establishes these correspondences. In a code declaration, the code name identifies the alphabet being declared. The code size is indicated in number of bits, digits, or characters per character. The right or left descriptor indicates whether shorter character strings are to be right or left justified in relation to longer strings.* This indication may be omitted if the character strings used are such that justification is never required.

SYNTAX

code-declaration ::= code Δ name_{of-code} Δ code-description

code-description ::= <nothing | code-string> Δ number Δ <bit | digit | character> Δ <nothing | left | right>

* Binary, octal, decimal, and hexadecimal texts are right-justified; alphanumeric texts are left-justified.

code-string ::= textual-constant Δ <nothing | is code> Δ <nothing |
code-string>

code ::= number | textual-constant

Notes:

- a. Each textual constant in a code string that is not serving as a code is understood to contain only one character of the alphabet being declared. The different characters must be uniquely represented.
- b. A code may be omitted if the "natural successor" to the previous code is meant, or if zero is meant when there is no previous code. (The natural successor to a textual constant may be derived by replacing its last character with the next character in the collating sequence of the alphabet from which the textual constant is constructed; or, if the constant's last character is also the last character of the alphabet, by replacing it with the first character in the alphabet and then replacing the constant's next-to-last character, and so on.)
- c. It is not necessary for each code in a code string to be different; many-to-one codings are often quite useful.
- d. It is not necessary for the codes in a code string to be in numeric order. Indeed, the collating sequence for a declared alphabet is determined, not by the numeric encoding, but by the sequence of characters as given in the code string, with the first character used as the filler character when justification is required.
- e. If the code string contains no codes that are textual constants, then either the bit descriptor or the digit descriptor is appropriate

for specifying the units of code size. If the code string does contain a code that is a textual constant, the character descriptor must be used in specifying the units of code size. Textual constants from more than one alphabet may not be used as codes in a code string.

- f. It sometimes cannot be determined from the code declaration alone if a space or a comma is intended as a character separator. However, this ambiguity can easily be resolved by looking at a multi-character textual constant in that alphabet.
- g. The code string may be omitted from a code description for those alphabets where textual constants are never used in denoting the value of textual items with that code.

Examples:

<u>code</u>	L.	'	'	'A'	'B'	'C'	'D'	'E'	'F'	'G'	'H'	'I'	'J'
				'J'	'L'	'M'	'N'	'O'	'P'	'Q'	'R'	'S'	'T'
				'U'	'V'	'W'	'X'	'Y'	'Z'	5	<u>bit</u>	<u>left</u>	

code GREEK. ' ' is L' / 'ALPHA' is L'A' / 'BETA'
is L'B' / 'GAMMA' is L'G' / 'DELTA'
is L'D' / 'EPSILON' is L'E' / 'ZETA'
is L'Z' / 'ETA' is L'EY' / 'THETA' is
L'TH' / 'IOTA' is L'I' / 'KAPPA' is
L'K' / 'LAMBDA' is L'L' / 'MU' is
L'M' / 'NU' is L'N' / 'XI' is L'X' /
'OMICRON' is L'O' / 'PI' is L'P' /
'RHO' is L'RH' / 'SIGMA' is L'S' /
'TAU' is L'T' / 'UPSILON' is L'U' /
'PHI' is L'PH' / 'CHI' is L'CH' /
'PSI' is L'PS' / 'OMEGA' is L'OH'

2 character left

code DIRECTION. 'NORTH' 'SOUTH' 'EAST' 'WEST' 2 bit

code COIN. 'PENNY' is 1, 'NICKEL' is 5, 'DIME' is 10, 'QUARTER' is
25, 'HALF' is 50, 'DOLLAR' is 100, 3 digit

code WORD. 36 bit

7.3 LIST DECLARATIONS

Lists are collections of similar data elements--items, arrays, or groups--that are linked together in memory by pointer items. A list may have several pointer items linking its elements together in several separate sequences. Several different but similarly described lists may be declared at once.

SYNTAX

list-declaration ::= list Δ name-string Δ list-description

list-description ::= full-list-description

full-list-description ::= <item-description | array-description | group-
description> Δ name-string

Examples:

list AFTER. array PLACE. real 3

list Q. integer M.

list LAST.

The name string in the list description serves to declare the pointer items that link the list elements together.

Abbreviated descriptions are possible for lists, according to the declared mode declaration.

7.4 LIST PROCESSING STATEMENTS

The list processing statements in extended SPL provide a rudimentary but basically adequate capability for list processing. List processing statements are available for referencing, linking, and freeing elements of lists.

SYNTAX

list-processing-statement ::= reference-statement | link-statement |
free-statement

7.4.1 Reference Statements

A reference statement places an element of a list "under consideration," so that is--and its components and associated pointers--may be subsequently designated without being explicitly located.

SYNTAX

reference-statement ::= see Δ list-element-reference

list-element-reference ::= name_{of-list} Δ <nothing | at Δ pointer-formula>

- NOTE:
- a. A list element may be located anywhere, as specified by an arbitrary (scalar) pointer formula (see Section 8.2.4 on located variables). However, a list element that has been automatically linked into a list, by a link statement, may be reliably referenced, after the execution of another link statement, not necessarily for the same list or in the same process, only by one of the list's pointers, since such list elements are subject to automatic reallocation of storage and concomittant adjustment of linkages by a built-in routine, usually called a "garbage collector", which may be automatically called during the execution of a link statement.
 - b. Where a pointer formula is omitted in a list-element reference, the element currently under consideration is assumed.

Examples:

see LETTER

see WORD at NEXT-WORD

see SYMBOL at cell sub Q

see SYMBOL at null

Although several lists may each have an element under consideration at once, no more than one element in any single list can be considered at any given time--regardless of how many processes are active--and unless a pointer value locating another list element is explicitly specified, defined references to data in a list element including points, pertain to the element currently under consideration.

A new element in a list may be considered by another execution of a reference statement. By giving the name of a list and the name of a pointer, for example, the programmer may place under consideration with a reference statement either the first or the next element in the list according to that pointer,* depending on whether or not an element in that list was previously under consideration. (When no element is under consideration for a list, its pointers each designate the location of the first element, if any, (or the null pointer value if none), in the corresponding element sequences they link.)

A list element can be removed from consideration by the execution of a reference statement wherein the pointer formula specifies the null pointer value.

* Note that a list element may have several pointers associated with it, so that it may have as many (or fewer) successors.

For lists with only one pointer, the pointer formula may be omitted from a reference statement, with the name of the lone pointer being assumed. With the list, LETTER. 1 character text NEXT.ELEMENT. list, for example, see LETTER \equiv see LETTER at NEXT.ELEMENT.

7.4.2 Link Statements

A link statement serves to allocate storage for a new list element, and to link it into the list as a successor to other elements in the list. A link statement may also serve to dynamically allocate storage for an item, an array, or a group.

SYNTAX

link-statement ::= link Δ <linkage-set | name of-item-array-or-group>

linkage-set ::= linkage Δ <nothing | linkage-set>

linkage ::= <nothing | pointer-set Δ from> Δ list-element-reference

pointer-set ::= name of-pointer Δ <nothing | pointer-set>

- Notes:
- a. A linkage set will ordinarily reference only one list, but may reference several identically-declared lists to permit inter-list linkages.
 - b. The pointers in a pointer set must all belong to the list named in the list-element reference in the linkage. Moreover, a pointer should not be named in a pointer set more than once.
 - c. A pointer need not be named in a linkage for a list with only one pointer.

Examples:

link LETTER

link NEXT.ELEMENT from LETTER

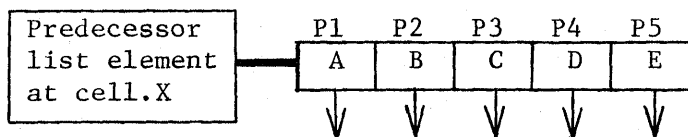
link NEXT.ELEMENT from LETTER at NEXT.ELEMENT

link PREVIOUS.WORD from WORD at NEXT.WORD, NEXT.WORD from WORD

A link statement containing a linkage set allocates storage for a single list element and its pointers,* and links that element into the list, as indicated by the set of linkages. Each linkage references a single list element as predecessor for the new element and, by naming the pointers, indicates the sequences in which the new element is to be the successor of the referenced element. Several linkages in a link statement should specify several predecessors for the new element; the actual linking, though, is done in the order in which the linkages are written.

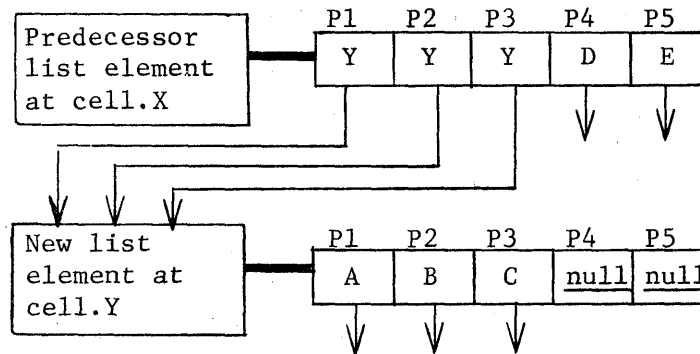
For each linkage, a link statement will assign the named pointers in the new element the values of those same pointers in the predecessor element, and then assign the named pointers in the predecessor element the pointer value of the new element. Consequently, the named pointers in the predecessor element will point to the new element, and the named pointers in the new element will point to the predecessor element's previous successors (for those pointers).

The mechanics of establishing a linkage are perhaps best explained with an illustrated example. Consider the following list, ELEMENT. begin ... end P1. P2. P3. P4. P5. list, where the element at cell X is shown below:



*When necessary, a link statement may automatically invoke "garbage collection."

The situation after the execution of the link statement link P1, P2, P3 from ELEMENT at CELL.X, is shown next:



The example shows that pointers not named in a link statement are assigned the null pointer value. Of equal importance though, is the fact that none of the components of a newly allocated and linked list element have defined values until these values are later assigned.

As in the reference statement, if no pointer-formula is given in a list-element reference in a linkage, the element currently under consideration is assumed. And if no element is currently under consideration for the list, then the newly linked element becomes the first element in the list--at least for the named pointers.

The execution of a link statement does not affect which list element (if any) is currently being considered.

The link statement may be applied to other data elements besides lists, i.e., items, groups, and arrays. When this is done, the data element is considered as a pointerless, zero- or one-element list, and linking accomplishes the dynamic allocation of storage for that element. Where storage is already allocated, however, a link statement has no effect.

7.4.3 Free Statements

A free statement serves to unlink one or more elements from one or more sequences in a list, perhaps deleting some elements from the list entirely, thus freeing their storage for later reallocation. A free statement may also serve to dynamically free storage for an item, an array, or a group.

SYNTAX

free-statement ::= free Δ <linkage-set | name_{of-item-array-or-group}>

Examples:

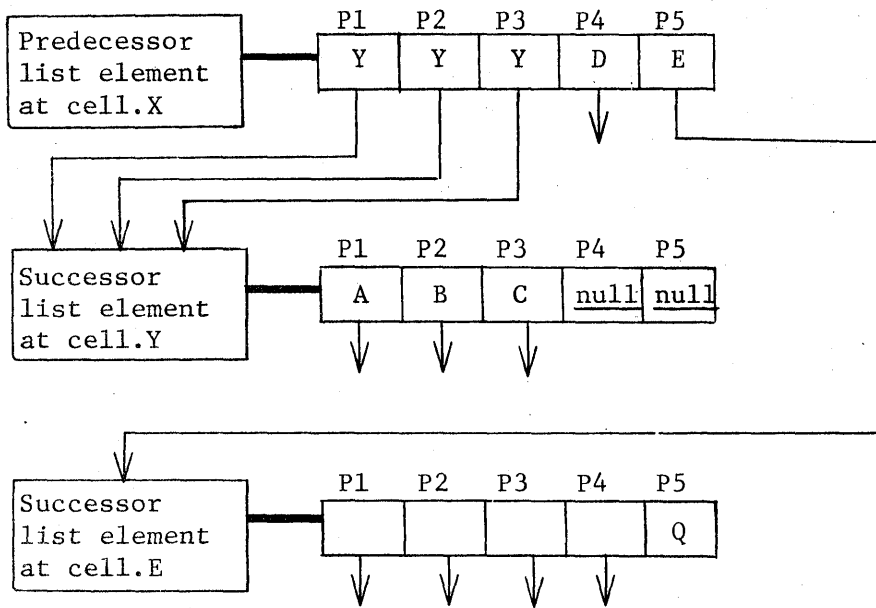
free LETTER

free NEXT.ELEMENT from LETTER

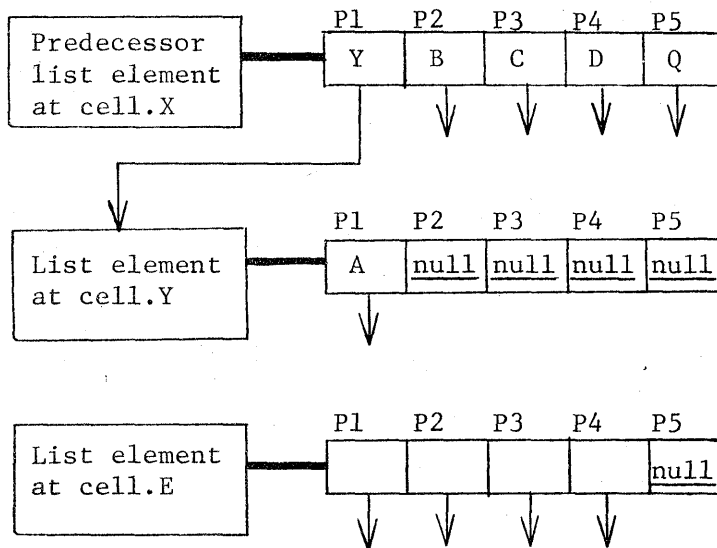
free NEXT.ELEMENT from LETTER at NEXT.ELEMENT

free NEXT.WORD from WORD, PREVIOUS.WORD from WORD at NEXT.WORD

A free statement may contain a linkage set, indicating which of the list's linkages are to be unlinked. Each linkage in such a free statement contains a reference to a single list element, which is the predecessor to the one or more successor elements that are to be freed (unlinked) from the predecessor element. These successor elements are specified by the values in the predecessor element of the pointers named in the linkage's pointer set. Unlinking occurs as follows. Each successor element is specified by a pointer in the predecessor element. The value of that same pointer in the successor element is assigned as the pointer's value in the predecessor element, and the pointer in the successor element is assigned the null pointer value. This is done for each successor element in the linkage, and then for each linkage in the free statement, in the order written. The result is that the named pointers in the referenced predecessor elements now point to the successors of the successor elements they previously specified. Again, an illustrated example is probably helpful. Consider then the next situation.



The situation after the execution of the free statement, free P2, P3, P5 from ELEMENT at CELL.X, is shown next.



As in the reference and link statements, if no pointer-formula is given in a list-element reference in a linkage, the element currently under consideration for the list is assumed. And if no element is currently under consideration for the list, then the unlinked elements are the first in their respective sequences.

The execution of a free statement does not affect which list element (if any) is currently being considered.

The free statement may be applied to other data elements besides lists, i.e., items, groups, and arrays. When this is done, the data element is considered as a pointerless, zero- or one-element list, and freeing accomplishes the dynamic freeing of storage for that element. Where no storage is allocated, however, the free statement has no effect.

7.5 ENCODING AND DECODING

Encoding and decoding are conversion operations ordinarily used in extended SPL in conjunction with reading and writing externally formatted record. Nevertheless, these operations are useful for converting other texts besides records, and even for conversions where neither operand is textual.

The encode and decode statements both have the same three parts: a (nominally) textual operand; a (nominally) nontextual operand; and a format, composed of conversion procedure calls that may be grouped by parentheses and unconditionally, conditionally and repetitively catenated.

SYNTAX

encode-decode-statement ::= <encode | decode> Δ variable Δ = Δ formula Δ by Δ
format

format ::= conversion-procedure-call | (Δ format Δ) | catenated-format |
repeated-format | conditional-format

conversion-procedure-call ::= name of-procedure Δ <nothing | actual-parameter |
(Δ actual-parameter-string Δ)>

catenated-format ::= format Δ <nothing | catenation-operator> Δ format

repeated-format ::= format Δ <nothing | catenation-operator> Δ (Δ repetition-
clause Δ)

conditional-format ::= closed-conditional-format | open-conditional-format

closed-conditional-format ::= if Δ condition Δ then Δ format₁ Δ else Δ format₂

open-conditional-format ::= if Δ condition Δ then Δ <format₃ | open-conditional-
format | Δ else Δ format₄>

- Notes:
- a. The discussion of encode and decode statements is based on the assumption that the variable in an encode statement and the formula in a decode statement are both textual in type. This need not generally be so, and the syntactically indicated extension to more general operands should be obvious.
 - b. Format₃ may be any format. Format₁, format₂, and format₄ may be of any format but an open conditional format.

The encode statement converts the value specified by a (nominally) nontextual formula, assigning the results as the value designated by a (nominally textual) variable. The encode statement works by invoking, in turn, each individual conversion procedure from the format, first automatically providing it with its major, actual parameters. (Any minor parameters must be supplied in the conversion procedure call.)

A conversion procedure in an encode statement may have several, minor argument parameters, but it has two major result parameters--an expression parameter, which is the (nominally textual) variable itself, and another expression parameter, which is an index variable designating the origin within this variable that is to receive the results of the conversion--and it may or may not have a major argument parameter.* If it does, it will be either a value or an expression parameter specifying some part of the formula's value, to be converted.

The segmentation of the formula's value into a sequence of actual major argument parameters for the format-specified sequence of conversion-procedure invocations, is done according to the rules given on repeated statements.

That is, where C is the formal major argument parameter of the conversion procedure being invoked, and A is the remaining, unconverted part of the formula, of dimension equal to or greater than C, the part of A that is to be the actual parameter corresponding to C is: A sub ((if (*) C size gr 1 then ((1 (C size size times)) to C size)), 1).

Where a catenation operator is encountered in an encode format, the effect is to pad the variable, where necessary, with the appropriate filler character, and to increment by one the corresponding dimension of the actual index parameter (which gives the origin within the variable for the results of the next conversion) and to reset its lower dimensions to one. Where, for example,

*A conversion procedure in an encode statement that merely inserts punctuation or control characters into the variable, may not require a major argument parameter.

the variable is a three-dimension textual operand (whose dimensions correspond to (1) characters in a line, (2) lines in a page, and (3) pages in a report), then the row catenation operator has the effect of inserting a single filler character in the variable and incrementing the character dimension of the index parameter by one; the // catenation operator has the effect of padding the remainder of the line with filler characters, incrementing the line dimension of the index parameter by one and resetting the character dimension to one; and the /// catenation operator has the effect of padding the remainder of the page with filler characters, incrementing the page dimension of the index parameter by one, and resetting both the character and the line dimension to one.

In summary, then, an encode statement invokes each individual conversion procedure in turn from the format, and where the conversion procedure has a major argument parameter, supplies it with the next unconverted portion of the value specified by the formula, and then inserts the converted results into the variable at the indicated origin.

The decode statement, on the other hand, also invokes each individual conversion procedure in turn from the format. But here, the implicit index parameter applies not to the variable, but to the formula.

A decode statement, then, converts the value specified by a (nominally textual formula), assigning the results as the value designated by a (nominally non-textual) variable. The decode statement invokes each individual conversion procedure in turn from the format, first automatically providing it with its major, actual parameters. (Any minor parameters must be supplied in the conversion procedure call.)

A conversion procedure in a decode statement may have several, minor argument parameters, but it has just one major argument parameter--an expression parameter--which is the (nominally textual) formula itself. A decode conversion procedure also has at least one major result parameter--an expression parameter--which is an index variable designating the origin within the formula of the data that is to be converted, and it may or may not have another major result parameter.* If it does, it will be either a value or an expression parameter designating some part of the variable that is to receive the converted results.

The segmentation of the variable into a sequence of actual major result parameters for the format-specified sequence of conversion-procedure invocations, is done according to the segmentation rule already described for the encode statement (and for the repeated statement).

In summary, a decode statement invokes each individual conversion procedure in turn from the format, and automatically supplies it with its major actual parameters: (1) as an argument, the formula given in the decode statement itself; (2) as a result, an implicit index variable designating the origin in the formula of the data to be converted; and in most cases (3) as another result, the next part of the variable given in the decode statement that is to receive the results of the conversion.

The format in an encode and decode statement gives a sequence of conversion-procedure calls. These may be grouped by parentheses and unconditionally, repetitively, and conditionally catenated.

*A decode conversion procedure that merely interprets punctuation or control characters in the formula, may not require another major result parameter.

So far as the conversion procedures themselves are concerned, SPL allows for a complete set of "built-in" encode and decode procedures for converting between, on the one hand, alphanumeric texts representing numeric, textual, pointer, and Boolean constants, and nonscalar formulas containing only these constants, and on the other hand, the internal representations of these as data values.

The programmer may, of course, declare other conversion procedures in addition to those built-in. An encode conversion procedure has at least two formal parameters: an expression result parameter whose corresponding actual parameter is the variable given in the encode statement; and an expression result parameter whose corresponding actual parameter is the implicit index variable generated by the encode statement. Most encode conversion procedures also have at least a third formal parameter: an expression or value argument parameter whose corresponding actual parameter, automatically supplied by the encode statement, is (or specifies) some portion of the value that is specified by the formula given in the encode statement. These are the major parameters of an encode conversion procedure.

A decode conversion procedure also has at least two formal parameters: an expression argument parameter whose corresponding actual parameter is the formula given in the decode statement; and an expression result parameter whose corresponding actual parameter is the implicit index variable generated by the decode statement. Most decode conversion procedures also have at least a third formal parameter: an expression or value result parameter whose corresponding actual parameter, automatically supplied by the decode statement, is (or designates) some portion of the variable given in the decode statement. These are the major parameters of a decode conversion procedure.

Conversion procedures may also be declared with additional, so-called minor parameters: argument or result parameters; value, expression or name parameters. Any minor parameters must precede the major parameters in the formal parameter string, however.

When a conversion procedure is invoked by an encode or decode statement, the actual, major parameters are omitted from the conversion procedure call, since these are supplied automatically by the encode or decode statement. Any actual minor parameters must, however, be supplied.

A conversion procedure may also be invoked, however, outside an encode or decode statement, by a procedure call statement. And in this case, all actual parameters, both major and minor, must be explicitly supplied. Conversion procedures may, of course, invoke other conversion procedures, either directly, or by encode and decode statements. The only restriction on the computation done by a conversion procedure is that it must update the implicit index variable generated by the encode or decode statement so that, on completion, it designates the origin for the next conversion.

7.6 ALGEBRAIC FORMULA MANIPULATION

In SPL, algebraic variables may be declared, and algebraic formulas may be symbolically manipulated, and where they are equivalent to defined numeric formulas, evaluated. An algebraic variable or formula is represented as a tree-like list of elements, any of which may be an algebraic variable or formula or an alphanumeric text, representing an algebraic operator, a numeric constant, or an atomic operand. Actually, all algebraic variables and formulas declared and specified in a program are represented in a single list. Where AF1, AF2, ..., AFn are the declared algebraic variables, this list might be declared as follows:

ALGEBRAIC.FORMULA.LIST.

begin

L. integer

ATOM. L character text

OP. L character text

store ATOM/OP

ATOMIC. Boolean

end

T.AF1. T.AF2. ... T.AFn.

N.AF1. N.AF2. ... N.AFn.

list

In the above algebraic formula list, N.AFi points to the next element of AFi. If the current element of AFi is an alphanumeric text, then L ng 0 and T.AFi eq null, and the Boolean item ATOMIC indicates whether the text represent an atomic operand, or an algebraic operator or numeric constant. On the other hand, if the current element of AFi is an algebraic variable or formula, then L eq 0, and T.AFi points to this element of T.AFi. Any given implementation of this algebraic formula manipulation capability may include additional pointers and items to simplify processing.* The programmer who wants to construct his own, special-purpose manipulation procedures, however, may reference those given above.

An alphanumeric text as an element of an algebraic formula may represent: a defined numeric formula specifying a numeric value; an undefined numeric formula--actually, any sign-string; a numeric constant; or an algebraic operator. In this context, a defined numeric formula is one whose operands are all either numeric constants, declared numeric items or arrays, or numeric-valued functions. Whether defined or undefined, though, the algebraic formula manipulation operations in SPL treat numeric-formula text as atomic operands; that is, the operations do not apply to the components of the numeric formulas. A numeric formula may be represented as an algebraic formula where it is desired to manipulate it algebraically.

Algebraic variables and arrays of algebraic variables may be declared (though only scalar algebraic formulas are manipulated), and algebraic formulas may be assigned to algebraic variables. In addition, a variety of built-in functions take algebraic arguments and produce numeric, textual, and Boolean as well as algebraic results. Algebraic functions and procedures may also be declared by the programmer, wherein the formal parameters corresponding to actual algebraic parameters are formal name parameters.**

*In particular, pointers for an algebraic "accumulator" may be included.

**This is already adequately accommodated by the subroutine capability described in Section 9, so no more need be said about it here.

SYNTAX

algebraic-declaration ::= name-string Δ algebraic Δ <nothing | dimension-string
 Δ array>

algebraic-variable ::= name_{of-algebraic-variable} Δ <nothing | sub |
 Δ index-string>

algebraic-formula ::= numeric-constant | algebraic-variable | atomic-operand |
(Δ algebraic-formula Δ) | <- | + > Δ algebraic-formula |
algebraic-formula Δ <- | + | * | / | ** | rem>
 Δ algebraic-formula | <abs | floor | ceiling | exp | log.e |
log.2 | log.10 | sin | cos | tan | arctan | cosh | sinh |
tanh> Δ algebraic-formula | <min | max > Δ (Δ algebraic-
formula-string Δ)

algebraic-formula-string ::= algebraic-formula Δ <nothing | algebraic-formula>

atomic-operand-string ::= atomic-operand Δ <nothing | atomic-operand-string>

atomic-operand ::= 'numeric-formula' | 'sign-string' | textual-formula

algebraic-assignment-statement ::= <nothing | let> Δ algebraic-variable
 Δ = Δ <algebraic-formula |
function_{of-algebraic-type}>

algebraic-operator ::= (|) | + | - | * | ** | / | abs | floor |
ceiling | rem | exp | log.e | log.2 | log.10 | sin |
cos | tan | arctan | cosh | tanh | sinh | min | max

Notes:

- a. The index string in a subscripted algebraic variable must specify a scalar algebraic variable.
- b. A numeric-formula text as an atomic operand in an algebraic formula should probably--but not necessarily--specify a scalar value, since algebraic formula manipulations are done as if on scalar values, and in some cases these manipulations are not mathematically valid for nonscalar operands.
- c. A numeric-formula text as an atomic operand in an algebraic formula should contain no textual constants--either as numeric operands or as arguments to numeric functions.
- d. A sign string (nominally representing an undefined numeric formula) as an atomic operand in an algebraic formula, must not contain the ' sign.
- e. Numeric constants and algebraic operators, though represented as alphanumeric texts in the algebraic formula list, are not written in an algebraic formula as textual constants.
- f. The statement prefix let is a "noise" word and may be omitted, though it improves readability in certain contexts.
- g. Algebraic functions are not permissible elements of algebraic formulas, contrary to what might be expected, since they involve algebraic manipulations.

Examples:

array N A. B. C. algebraic

D. algebraic

let D = A sub I - (D**2 + B sub I/C sub I)

let A sub I = 'ALPHA sub I' * 'THIS OPERAND SHOULD FACTOR OUT'

let C sub 8 = expand D

C sub K = 27

An algebraic declaration serves to declare one or more algebraic variables. In addition, each algebraic variable is given an initial, atomic value: an undefined text typographically identical to the algebraic variable used to designate the value. Thus, D has the initial value 'D', C sub 1 has the initial value 'C sub 1' and so on.

The algebraic operators are quite analogous to the arithmetic operators and numeric functions they typographically resemble. Nevertheless, they do not automatically invoke numeric operations--or algebraic operations for that matter--since they are basically just textual symbols in a symbolic formula.

The algebraic assignment statement operates in a straightforward way, as if by linking the elements of the algebraic formula together in some algebraic accumulator, freeing the elements of the algebraic variable, and then linking to it the elements in the algebraic accumulator, or by first doing the manipulations specified by the algebraic function, and then using the results as an algebraic formula for assignment.

7.6.1 The Evaluation Function

The evaluate function takes an algebraic formula as argument--one containing no undefined atomic elements--and converts it to a numeric value by first evaluating the atomic elements and then evaluating the resulting formula by considering the algebraic operators as arithmetic operators and numeric functions.

SYNTAX

function_{of-numeric-type} ::= eval Δ algebraic-formula

Examples:

eval D

eval (A sub I - (D**2+B sub I/C sub I))

The result of evaluating an algebraic formula with undefined atomic elements is undefined. The result of evaluating an algebraic formula with defined but nonscalar atomic elements is, in general, nonscalar.

7.6.2 The Represent Function

The represent function takes an algebraic formula as argument and converts it to an alphanumeric textual value--an equivalent algebraic formula containing no non-atomic operands (i.e., no algebraic variables).

SYNTAX

function_{of-textual-type} ::= rep Δ algebraic-formula

Examples:

rep D \equiv (initially)* 'D'

rep (A sub I - (D ** 2 + 'Q' * B sub I))

*Before any assignments to D.

7.6.3 The Defined Function

The defined function takes an algebraic formula as argument and produces a scalar Boolean value as result: true if the algebraic formula contains no undefined atomic elements; false if it does.

SYNTAX

function_{of-Boolean-type} ::= algebraic-formula Δ defined

Examples:

'Q' defined

D defined

(A sub I - (D * 2 + 'Q' * B sub I)) defined

7.6.4 The Identity Function

The identity function takes a pair of algebraic formulas as arguments and produces a scalar Boolean value as result: true if the two formulas are found to be identical or mathematically equivalent; false otherwise.*

SYNTAX

function_{of-Boolean-type} ::= algebraic-formula Δ ident Δ algebraic-formula

Examples:

D ident 27

C sub 1 ident (D ** 2 + 'Q' * D)

*This, of course, is not a sure test, since only the more common equivalences are employed. Another good procedure here is to evaluate the two formulas for a range of values and compare the results.

7.6.5 The Approximate Function

The approximate function takes as its arguments a pair of algebraic formulas and a scalar numeric formula specifying a tolerance value, and produces a scalar Boolean value as result: true if the two formulas are found to be approximately identical or mathematically equivalent; false otherwise.

SYNTAX

```
functionof-Boolean-type ::= algebraic-formula  $\Delta$  approx  $\Delta$  (<numeric-formula |  
algebraic-formula>)
```

Examples:

D approx (1e-4, 'Q')

(27*'Q') approx (28, 'Q') \equiv true

The approximate comparison is performed in almost exactly the same way as the identity comparison, and using the same equivalences, except that two algebraic formulas are approximate if matching terms in each have constant coefficients that do not differ by an amount whose absolute value is greater than the absolute value of the tolerance value.

7.6.6 The Reduce Function

The reduce function takes as its arguments an algebraic formula and a binary textual formula, specifying a 22-bit binary text with a special meaning: each bit of the text corresponds to an algebraic operator, and a zero bit means the corresponding algebraic operation is to be reduced while a one bit means the corresponding algebraic operation is not to be reduced. The reduce function produces a "simplified" algebraic formula. However, only the indicated algebraic operations are employed in arriving at this simplified result the others are "not reduced."

SYNTAX

function of-algebraic-type ::= reduce Δ (algebraic-formula Δ textual formula)

Examples:

reduce (D, null)

reduce (D ** 2 + 'Q' * D, bit '1000100110110110111110')

The reduce function operates as follows:

- a. all defined atomic operands are evaluated.
- b. where their operands have been numerically evaluated, all indicated algebraic operations (except those specified as not to be reduced) are numerically performed.
- c. like terms and factors are combined.

7.6.7 The Expand Function

The expand function takes an algebraic formula as argument and removes the parentheses from it by applying the distributive law and/or the multinomial theorem, thus producing as a result, an "expanded" algebraic formula.

SYNTAX

function of-algebraic type ::= expand Δ algebraic-formula

Examples:

expand D

expand (B sub I * 'if T then 0 else Q')

7.6.8 The Coefficient Function

The coefficient function takes a pair of algebraic formulas as its arguments, and the result is an algebraic formula that is the coefficient of the second argument as it appears within the first argument.

SYNTAX

function of-algebraic-type ::= algebraic-formula₁ Δ coeff Δ algebraic-formula₂

Examples:

D coeff 'Q'

('Q' * 'X' ** 2 + 'P' * 'X' ** 3) coeff ('X' ** 2) \equiv ('Q' + 'P' * 'X')

Where the second argument is not an element of the first, the result of the coefficient function is, of course, zero.

7.6.9 The Differentiation Function

The differentiation function takes the (full or partial) derivative of an algebraic formula with respect to one or more other algebraic formulas. The result is an algebraic formula.

To permit differentiation where functional relationships among atomic operands are not explicitly given (at least in terms of the operations and functions allowed in an algebraic formula), implicit dependence relationships among atomic operands may be given as an adjunct to the differentiation function.

SYNTAX

function of-algebraic-type ::= algebraic-formula Δ deriv Δ <algebraic-formula
| (algebraic-formula-string)> <nothing |
(where Δ dependencies)>

dependencies ::= atomic-operand Δ is Δ f (Δ atomic-operand-string Δ)
 Δ <nothing | dependencies>

Examples:

D deriv 'Q'

('X' ** 2) deriv 'X' \equiv (2 * 'X')

D deriv ('Y sub 1', 'Y sub 2', 'Y sub 3')
(where 'X' is f ('Y sub 1', 'Y sub 3'))

The differentiation function works as follows: the derivative of the algebraic formula preceding deriv is first taken with respect to the first algebraic formula following deriv, then the derivative of the resulting formula is taken with respect to the second algebraic formula following deriv (if any), and so on. Where dependencies among atomic operands are specified for a differentiation, the atomic operand preceding is depends on (is some unspecified function of) the atomic operand or operands within the f (and) brackets.

Certain of the algebraic operators are not differentiable, and thus care should be exercised in differentiating an algebraic formula containing them.

7.7 INTERACTIVE PROGRAMMING

Interactive, on-line programming is possible in extended SPL with an interpreter or incremental (line-at-a-time) compiler running on a time shared computer. The commands used to control the on-line compilation and execution of a program are also useful in controlling program compilation and execution off-line, and the results are completely similar, except for the lack of quick interaction.

Each statement, declaration, and command input or entered is immediately checked by the compiler for formal errors and, to some limited extent, for logical errors. Diagnostics or warnings are automatically incorporated in the program listing immediately after the line causing them (but without a line number). These should emphasize intelligibility, yet be reasonably brief. Should a novice programmer, on-line, require further explanation, he may be able to retrieve a tutorial text, cataloged under some appropriately descriptive title. The system's diagnostic, warning, and advisory messages are printed without line numbers.

To facilitate the writing in extended SPL of programs intended to communicate interactively with an on-line teleterminal, extended SPL includes a pair of built-in-procedures--accept and display--which perform the necessary read-decode, encode-write operations on an implementation defined file employing the appropriate device: the user's own terminal in a multi-access system, the operator's terminal otherwise. The calls for these procedures have the following syntax.

SYNTAX

```
procedure-call-statement ::= <nothing | call> Δ <accept Δ variable |  
                           display Δ formula>
```


Examples:

accept I

accept (J, A sub I // B)

display ('PI = ' // 4*arctan 1)

At the terminal, the effect of the three procedure call statements above would be to print:

```
set I =  
set (J, A sub I // B) =  
PI = 3.1415927...
```

After each of the first two printouts, the user or operator would be expected to enter an appropriate SPL formula--containing only constant operands, though--which the accept procedure would read, decode, and assign to the designated variable.

Insofar as possible, the message produced by the display procedure will be tabular. But where the size and/or dimension of the value specified by the formula preclude this, the displayed message will utilize the linear notation of SPL formulas.

7.8 COMMANDS

7.8.1 Edit Commands

Programs are written to be executed but, unfortunately, modification or editing is a far more common operation. Programs are considered to be input or entered a line at a time. Each line of program text--whether it contains a statement, a declaration, a command, or a comment--is automatically given a serial line number, for editing purposes, by the compiler. The editing command permits lines to be inserted, deleted, replaced, and renumbered.

SYNTAX

`edit-command ::= edit Δ lines Δ <nothing | out | is Δ character-string`

`lines ::= <all | line-number | to Δ line-number | line-number Δ to
Δ line number> Δ <nothing | lines>`

`line-number ::= number <nothing | line-number>`

Examples:

`edit all`

`edit all out`

`edit 38 to 38.19 is begin ... end`

`edit 17.1 out`

`edit to 74.6.9`

Lines input or entered are automatically given serial numbers by the compiler: 1, 2, 3, 4, 5, 6, etc. Line numbers are automatically printed at the beginning of each line, effectively, as part of the line-feed/carriage-return action.

Assume 100 lines of text have been entered; these would be numbered

1.
2.
3.
⋮
99.
100.

in the left margin. To insert new lines between 57 and 58, say, the edit command,

101. `edit 57.1 is ...`

would be used. The remainder of the edit command on line 101 would logically become line 57.1, and the automatic line numbering would resume with 57.2, 57.3, and so on. When the insertion is done and it is desired to resume the program where it had been left off, the edit command

57.23. `edit 102 is ...`

could be used. To insert text at the beginning of a program, an edit command without a line number is used. Thus,

138. edit is ...

would cause the remainder of line 138 to be given the number 0.1, and the next line number would be 0.2, and so on. Another such edit command, e.g.,

0.28. edit is ...

would cause the remainder of line 0.28 to have the line number 0.0.1, and the next line number would be 0.0.2, and so on.

To delete a line, or a series of lines, the primitive out must be explicitly used. The edit command,

0.0.3. edit 75 out

would delete line 75 from the previously provided text. The edit command

0.0.4. edit 70 to 88.6 out

would then delete lines 70-74 and 76-88.6; line 75 having of course been previously deleted. It should be noted that a subsequent edit command

0.0.5. edit 0.0.3 out

would not have the effect of replacing line 75, just the effect of deleting line 0.0.3 from any subsequent listing of the text. Nor, to use an earlier example, would

0.0.6. edit 101 out

have any effect on what the edit command on that line had already caused to be inserted at line 57.1.

All previously provided text may be deleted by the all-consuming command

0.0.7 edit all out

After such a command, the next line number would automatically be 1. But assume such drastic steps are unnecessary, and it is only desired to replace a line.

A line, say 27, may be replaced with the following edit command:

0.0.7. edit 27 is ...

This is entirely equivalent to

0.0.7. edit 27 out, edit 27 is ...

With either command, the remainder of line 0.0.7. is given the number 27, and the next line would automatically be 28, so that anything entered or input there would replace any previous line 28, and so on. (But it would not replace any line 27.1 or any other line between 27 and 28.) If such subsequent automatic replacement of lines 28, 29, et. seq. were not desired, the command

0.0.7. edit 27.0 is ...

This would also replace line 27 (\equiv 27.0) with the remainder of line 0.0.7, but the next line would automatically be 27.1.

A sequence of lines may also be replaced with an edit command, say

27.1. edit 70 to 88.6 is ...

And this is exactly equivalent to

27.1. edit 70 to 88.6 out, edit 70 is ...

Not that such an edit command deletes all lines between 70 to 88.6, inclusive, no matter how deeply they may be numbered.

After a great deal of the kind of editing exemplified above, line numbers are likely to be in a hodge-podge, with gaps in the sequence, and seven- or eight-level line numbers in places. This can be corrected by renumbering, with an edit command, such as

367.6.54.2. edit all

which renumbers all previously provided and remaining lines: 1, 2, 3, and so on. Of course, any sequence of lines can also be renumbered, with an edit command like

59.6. edit 23.4 to 38.7.12

which would renumber all lines between 23.1 and 38.7.12, inclusive, no matter how deeply numbered. These would be renumbered: 23.4, 23.5, 23.6, and so on.

An edit command to renumber can also have the effect of replacing lines (though this can easily be avoided, with a little care, where replacement is not wanted). For example

38.5. edit 16 to 25

would cause the indicated lines to be renumbered 16, 17, 18,--and if there were more than ten lines in the sequence--25, 26, 27, ... and so on, causing any lines previously numbered 26, 27, etc. to be replaced.

To renumber just some initial sequence of lines, the command

57. edit to 31.6

could be used to renumber all lines with numbers less than or equal to 31.6. And to renumber some final sequence, the command

58. edit 40

might be used (since there is no point, of course, in renumbering a single line) to renumber all lines with numbers greater than or equal to 40. (This last interpretation does not apply to deletion. Thus,

59. edit 40 out

would delete just a single line.)

7.8.2 Save Commands

A save command serves to store and catalog, under a user supplied title, the current values of any data elements, or any lines of previously supplied program text--including texts composed entirely of commentary.

SYNTAX

save-command ::= Δ title $\Delta = \Delta$ <formula | lines>

title ::= name Δ <nothing | (all)>

Note: There is syntactic ambiguity between some numeric formulas and some line numbers. In a save command, in such cases, the lines number interpretation will prevail. And if, for some strange reason, the programmer wants to save the integers 1 to 100 rather than lines 1 to 100, he must use parentheses; e.g., (1 to 100).

Examples:

save EXPLANATION.OF.SAVE = 'A SAVE COMMAND SERVES TO STORE AND CATALOG,
UNDER A USER SUPPLIED TITLE, THE CURRENT ...'

save JONES.PROGRAM.EPHEMERIDES.03 = all

save D.7 = to 132.8

Used as a title in a save, show, or get command, a name exhibits a hierarchic structure, with the embedded periods delimiting the various levels. A variety of names can be given to the different levels. Library-file-section-shelf-volume-book-chapter-page-paragraph might be one such (improbable) sequence, so that A.B.C.D.E.F.G.H.I, as a title, would be interpreted: library A, file B, section C, shelf D, volume E, book F, chapter G, page H, paragraph I. A less improbable interpretation of one of the preceding examples might be: Jones' library, program file, Ephemerides routine, 3rd version. The point of all this is that an abbreviated title, say JONES.PROGRAM, refers to all the routines in the program file of Jones' library.

A save command may replace an existing element in a library or add a new element (or even a new library) to the system, depending on whether or not an element cataloged under that title already exists in the system. In either case, the saved element will be cataloged in the appropriate place in the hierarchy.

Whenever a name is used in a title that refers to an existing element that is not at the bottom level of its hierarchy--i.e., the name refers to a number of bottom level elements--the primitive all must be added in parentheses after the name as part of the title, to make it less likely that unintentional replacement or purging of whole files will occur. Any other safeguards to prevent unauthorized or unintentional replacement, purging, or access to saved elements are implementation defined.

Any bottom level element, say Jones' program, Ephemerides, version 03, may be purged (along with its name) by a save command like the following:

```
save JONES.PROGRAM.EPHEMERIDES.03 = null
```

Any bottom level element or collection of bottom level elements, say all versions of Jones' program, Ephemerides (with their names) may be purged with a save command like the following:

```
save JONES.PROGRAM.EPHEMERIDES (all) = null
```

It is a useful and relatively simple and straightforward practice for the user to construct and maintain an index--with save commands--for any level of a library hierarchy. The system does not, however, do this automatically.

7.8.3 Get Commands

A get command serves to retrieve any previously saved (and retained) data or lines of program text.

SYNTAX

```
get-command ::= get Δ <variable Δ = Δ title | title>
```

Examples:

get JONES.PROGRAM.EPHEMERIDES.03

get COMPOOL.21 (all)

get BETA = N1626.MATRIX (all)

get D.7

Where a set command retrieves data, the effect is that of assignment to a variable. Thus:

save Q = 1.693**I sub (1 to N), get P = Q, save Q = null

has exactly the same effect as:

set P = 1.693**I sub (1 to N)

Where a get command retrieves lines of program text, the effect is exactly the same as if the lines of text were input or entered--any commands in the retrieved text will be obeyed. The line numbers of the retrieved text, however, will all be prefixed with the line number of the get command itself. Thus,

61.13 get SYSTEM.SUBROUTINE.OBOE

The lines of system subroutine OBOE would be inserted as 61.13.1, 61.13.2, 61.13.3, etc. And if for some reason this is not desired, for example, because of the existence of another line 61.13.1 that is to be retained, the following commands could be used:

61.13. edit 61.13.0 is get SYSTEM.SUBROUTINE.OBOE

so that OBOE would be inserted as 61.13.0.1, 61.13.0.2, 61.13.0.3, and so on.

Where a `get` command retrieves several separate sets of program text, as in

17. `get` COMPOOL.21 (all)

the line numbers in each set are prefixed with a unique serial number for the set, before being prefixed by the `get` command's line number. Thus, in the above example, the line numbers for the first part of COMPOOL.21 would be 17.1.1, 17.1.2, 17.1.3, and so on, and for the *i*th part, they would be 17.i.1, 17.i.2, etc.

7.9 BUILT-IN FUNCTIONS

Extended SPL contains a number of built-in functions in addition to those described for basic SPL.

7.9.1 Functions

7.9.1.1 Minimum and Maximum Functions. The minimum and maximum functions are used to specify the minimum and maximum scalar value in a (nonscalar) formula.

7.9.1.2 Remainder Function. The remainder function specifies the remainder, after division, of the real scalar dividend *x* by the real divisor *y*. The remainder function may be generally defined as: $x \text{ rem } y \equiv x - y * \text{floor}(x/y)$. Complex arguments of the remainder function are converted to real mode by disregarding their imaginary parts. Nonscalar arguments specify identically structured nonscalar results.

7.9.1.3 Conjugate Function. The conjugate function, for any complex scalar argument ($a+b*li$), specifies ($a-b*li$). Real arguments of the complex conjugate function are converted to complex mode by assuming imaginary parts of zero. Nonscalar arguments specify identically-structured nonscalar results.

7.9.1.4 Floor Function. The floor function, for any integer or real scalar argument *x*, specifies the largest integer not exceeding *x*. Complex arguments of the floor function are converted to real mode by disregarding their imaginary parts. Nonscalar arguments specify identically-structured nonscalar results.

7.9.1.5 Ceiling Function. The ceiling function, for any integer or real scalar argument x , specifies the smallest integer not exceeded by x . Complex arguments of the ceiling function are converted to real mode by disregarding their imaginary parts. Nonscalar arguments specify identically-structured nonscalar results.

7.9.1.6 Hyperbolic Functions. The hyperbolic functions sinh, cosh, and tanh specify, for any real scalar argument x , the hyperbolic sine, cosine, and tangent of x . Complex arguments of these functions are converted to real mode by disregarding their imaginary parts. Nonscalar arguments specify identically-structured nonscalar results.

7.9.1.7 Identity Matrix. The identity-matrix function specifies an m by n numeric matrix whose elements have the value one along the main diagonal and zero elsewhere. The number of rows in the identity matrix is specified by the scalar numeric-formula₁, the number of columns by the scalar numeric-formula₂. Either or both arguments may be omitted where the number of rows or columns can be determined by compatibility considerations of context.

The elements of the identity matrix may be defined as follows:

id sub (I,J) \equiv if I eq J then 1 else 0.

7.9.1.8 Determinant Function. The determinant function specifies, for any square n by n numeric matrix A , the determinant of A .

7.9.1.9 Size Function. The size function specifies, for any k -dimensional formula F , the number of elements along each of the dimensions of F . The size function, for a k -dimensional formula, specifies an index value--an integer-valued k -element vector (integer k array). For a rectangular, nonscalar formula, size sub 1 specifies the number of rows, size sub 2 the number of columns, size sub 3 the number of planes, and so on (assuming the

number of dimensions exceeds 3). For a nonrectangular, nonscalar formula, the number of elements along any given dimension may vary. Here, the corresponding element of the size specified vector is the maximum number. For a scalar argument, the size function specifies the value one and for a null argument, for example, a text of length zero, the size function specifies zero.

7.9.1.10 Origin Function. The origin function specifies, for any pair of formulas X and Y, where the value of X is an element of the value of Y, the index of the first origin--first in the sequence (1,1,...,1) to (Y size)--of X in Y. Where the value of X is not an element of the value of Y, the origin function specifies the value zero. Where X is an element of k-dimensional Y, the origin function specifies an index value--an integer-valued k-element vector. Also, $X \text{ eq } Y \text{ sub } ((X \text{ origin } Y) \text{ to } (X \text{ origin } Y + X \text{ size}))$.

7.9.1.11 Coordinate Transformations. The coordinate transformation functions specify the transformations among real-valued 3-vectors representing Polar, Cartesian, and direction cosine coordinates in 3-space (syntax unspecified).

8. LISTING OF SYNTAX EQUATIONS

8.1 ALPHABET, VOCABULARY AND PROGRAM STRUCTURE (Ref. Section 2)

character ::= letter | digit | mark

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

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

mark ::= space | (|) | + | - | * | / | . | , | ' | = | \$

symbol ::= name | primitive | constant

name ::= <letter | name> <nothing | letter | digit | .letter | .digit>

primitive ::= delimiter | operator | descriptor

delimiter ::= . | , | / | \$ | (|) | ' | '' | begin | end | term | program

operator ::= catenation-operator | repetition-operator | conditional-operator |
arithmetic-operator | define-operator | logical-operator |
relational-operator | assignment-operator | functional-operator |
discrimination-operator | sequential-operator | input-output-operator |
location-operator | editing-operator | compile-operator

arithmetic-operator ::= - | + | * | / | **

logical-operator ::= not | and | or

relational-operator ::= eq | nq | gr | ls | gq | lq | equiv

conditional-operator ::= if | then | else

assignment-operator ::= = | set

repetition-operator ::= times | by | while | until | for

catenation-operator ::= // | ///

sequential-operator ::= goto | stop | when | on | call | entry | exit | for | do

input-output-operator ::= open | close | read | write | assign | status | position

location-operator ::= store | at
 editing-operator ::= out | is | all | to | show | thru
 compile-operator ::= optimize | count
 define-operator ::= execute | where | then | names of-terms*
 functional operator ::= log.e | log.10 | sin | cos | tan | abs
 descriptor ::= integer | real | pointer | boolean | array | mode | procedure |
 function | pattern | file | dec | oct | hex | bit | text | fixed |
 float | cell | true | false | ready | busy | error | addr |
 item | minimum | digit | maximum | signed | unsigned | round |
 truncate | group | compool | full | unready | value | result |
 recursive | reentrant | time | space | none
 constant ::= numeric-constant | textual-constant | pointer-constant |
 boolean-constant
 numeric-constant ::= number | real-constant | binary-constant | octal-constant |
 decimal-constant | hexadecimal-constant
 numeral ::= digit <nothing | numeral>
 signed ::= + | -
 number ::= numeral <nothing | e_{xponent-base-10} numeral>
 real-constant ::= <numeral . | . numeral | numeral . numeral> <nothing |
 e_{xponent-base-10} numeral | e_{xponent-base-10} signed numeral>
 binary-constant ::= <name. | nothing> Δ bit ' binary-string '
 binary-string ::= <0 | 1> <nothing | binary-string>
 octal-constant ::= <name. | nothing> Δ oct ' octal-string '
 octal-string ::= <0 | 1 | 2 | 3 | 4 | 5 | 6 | 7> <nothing | octal-string>
 decimal-constant ::= <name. Δ nothing> Δ <nothing | dec> ' numeral '
 hexadecimal-constant ::= <name. | nothing> Δ hex ' hexadecimal-string '
 hexadecimal-string ::= <numeral | A | B | C | D | E | F> <nothing |
 hexadecimal-string>

* Listing of terms in Section 7.

```

textual-constant ::= <name. | nothing> <nothing | text> ' character string '
character string ::= character <nothing | character-string>
pointer-constant ::= cell Δ name | addr Δ name
boolean-constant ::= true | false
statement ::= simple-statement | compound statement
compound-statement ::= <name. | nothing> Δ begin Δ statement-string Δ end Δ
                        <name | nothing>
statement-string ::= statement | declaration | command <statement-string |
                        nothing>
declaration ::= data-declaration | subroutine-declaration | file-declaration

simple statement ::= simple-control-statement | input-output-statement |
                    procedure-call-statement | assignment-statement
comment ::= "character-string"
program ::= program Δ name.Δstatement-string Δ term <name | nothing>

```

8.2 DATA DEFINITIONS (Ref. Section 3)

```

data-declaration ::= item-declaration | array-declaration | group-declaration |
                    storage-declaration | mode-declaration
item-declaration ::= <item | nothing> Δ name-string Δ item-description Δ <nothing |
                    initial-value-string>
name-string ::= name. Δ <nothing | name-string>
initial-value-string ::= /item-value Δ <nothing | initial value string>
item-value ::= numeric-constant | pointer-constant | textual-constant |
                    boolean-constant
item-description ::= numeric-item-description | textual-item-description |
                    pointer-item-description | boolean-item-description
numeric-item description ::= full-integer-item-description | full-real-item-
                        description

```

full-integer-item-description ::= integer Δ number₁ Δ <bit | digit> Δ <minimum |
 nothing> Δ <nothing | number₂ Δ maximum> <nothing |
signed | unsigned> Δ <nothing | round | truncate>

full-real-item-description ::= real Δ number₁ Δ <bit | digit> Δ <nothing | minimum>
 Δ <nothing | -> number₂ <bit | digit> Δ <nothing |
float | fixed> Δ <nothing | signed | unsigned> Δ
 <nothing | round | truncate>

textual-item-description ::= text Δ <nothing | number Δ character |
 name
 of-integer-item Δ character>

pointer-item-description ::= pointer

boolean-item-description ::= boolean

array-declaration ::= array Δ name-string Δ array-description

array-description ::= <item-description | group-description> Δ dimension-string
 Δ <initial-value-string | nothing>

dimension-string ::= <number | name
 of-integer-item> Δ <nothing | by Δ dimension-
 string>

group-declaration ::= group Δ <nothing | name-string> Δ group-description

group-description ::= begin Δ declaration-string Δ end Δ <nothing | (name)

declaration-string ::= <item-declaration | array-declaration | group-
 declaration | function-declaration | mode-declaration |
 storage-declaration> Δ <nothing | declaration-string>

mode-declaration ::= mode Δ <numeric-item-description | array-description |
 full-file-description>

storage-declaration ::= store Δ block-description Δ at Δ pointer-formula

block-description ::= name
 of element Δ <nothing | block-description>

variable ::= name | subscripted-variable | conditional-variable |
catenated-variable | hardware-operand

subscripted-variable ::= variable <(index-string) | Δ sub Δ index-string>

index-string ::= index Δ <nothing | <>// | to> Δ index-string>

index ::= <numeric-formula | index-string> Δ <nothing | index> | (index)

catenated-variable ::= variable Δ <nothing | catenation-operator> Δ variable

conditional-variable ::= if Δ condition Δ then Δ variable₁ Δ <nothing | else Δ variable₂>

hardware-operand ::= \$ name

compool-declaration ::= compool Δ name_{of-compool}

8.3 STATEMENTS (Ref. Section 4)

assignment-statement ::= <nothing | set> Δ variable Δ = Δ formula

formula ::= numeric-formula | textual-formula | pointer-formula |
boolean-formula | (formula)

numeric-formula ::= constant | function | variable | hardware-operand
Δ <nothing | arithmetic-operator> Δ <nothing |
numeric-formula> | n-ary-arithmetic-operator Δ numeric-
formula | numeric-formula Δ matrix-operator Δ
numeric-formula | boolean-formula

n-ary arithmetic-operator ::= (arithmetic-operator)

matrix-operator ::= arithmetic-operator₁ . arithmetic-operator₂

textual-formula ::= textual-constant Δ <catenation-operator | subscription-
operator>

pointer-formula ::= name Δ <sub Δ index-string | nothing > | cell Δ sub Δ
index-string | cell Δ variable | bit Δ sub Δ
index-string

condition ::= boolean-formula

boolean-formula ::= boolean-constant | not Δ boolean-formula | boolean-
formula Δ <and | or | equiv> Δ boolean-formula |
relational-formula | n-ary-logical-operator Δ boolean-
formula

relational-formula ::= <nothing | n-ary-relational-operator> Δ formula
 Δ <nothing | relational-operator Δ relational-formula>

n-ary-logical-operator ::= (<and | or | equiv>)

n-ary-relational-operator ::= (relational operator)

direct-code-statement ::= \$name_{of-hardware-operator} Δ (actual-parameter-string)

control-statement ::= simple-control-statement | complex-control-statement
 simple-control-statement ::= transfer-statement | stop-statement | procedure-call-statement

complex-control-statement ::= repeated-statement | conditional-statement |
 parallel-statement | delayed-statement |
 chronic-statement

transfer-statement ::= <go Δ to | goto> Δ pointer-formula

repeated-statement ::= for Δ repetition-clause Δ statement

repetition-clause ::= variable Δ = Δ value-sequence

value-sequence ::= formula | numeric-formula₁ Δ by Δ numeric-formula₂ Δ
 <while | until> Δ condition

conditional-statement ::= if Δ condition Δ then Δ statement₁ <nothing |
 Δ else Δ statement₂>

parallel-statement ::= do Δ statement

stop-statement ::= stop

delayed-statement ::= when Δ condition Δ statement

chronic-statement ::= on Δ condition Δ statement

input-output-statement ::= open-statement | close-statement | read-statement |
 write-statement

file-declaration ::= file Δ name-string Δ file-description

file-description ::= device-name Δ <nothing | \$(character-string)> Δ <nothing |
 dimension-string> Δ <nothing | code-name>

device-name ::= name. | device-name. <nothing | number> Δ <nothing | module-name.
 number>

code-name ::= <bin | oct | dec | hex | text>
assign-declaration ::= assign Δ name to Δ device-name
open-statement ::= open Δ device-name Δ <nothing | \$(character-string) Δ <nothing |
dimension-string> Δ <nothing | code-name> Δ file-designation
close-statement ::= close Δ <nothing | out | out Δ module-name> Δ file-designation
file-designation ::= name_{of-file} | file Δ at Δ pointer-formula
function ::= file-designation Δ <status | position>
functional-variable ::= file-designation Δ position
read-statement ::= read Δ variable Δ into Δ file-designation
write-statement ::= write Δ file-designation Δ from Δ textual-formula

8.4 PROCEDURES AND FUNCTIONS (Ref. Section 5)

procedure-declaration ::= procedure-heading Δ <nothing | parameter-declaration-
string> Δ statement
procedure-heading ::= procedure Δ name_{of-procedure} Δ <nothing | (formal-
parameter-string)> Δ <nothing | formal-parameter-string>
formal-parameter-string ::= formal-parameter Δ <nothing | formal-parameter-string>
formal-parameter ::= name
parameter-declaration-string ::= parameter-declaration Δ <nothing | parameter-
declaration-string>
parameter-declaration ::= <item-declaration | array-declaration | group-declaration>
Δ <nothing | value> Δ <nothing | result> <procedure-
heading | function-heading Δ <nothing | parameter-
declaration-string> | file-declaration
procedure-call-statement ::= call Δ name_{of-procedure}
Δ <nothing | actual-parameter | (actual-parameter-
string)>

actual-parameter-string ::= actual-parameter Δ <nothing | (actual-parameter-string)>

actual-parameter ::= variable | formula | name

entry-statement ::= name Δ entry <(parameter-declaration) | nothing>

exit-statement ::= exit

function-declaration ::= function-heading Δ <nothing | parameter-declaration-string> Δ <= Δ formula | complex-statement | compound-statement>

function-heading ::= function Δ <nothing | formal-parameter | (formal-parameter-string)> Δ name_{of-function} Δ <nothing | formal-parameter | (formal-parameter-string)> Δ <item-description | array-description | group-description> Δ <nothing | recursive | reentrant>

function_{of-numeric-type} ::= <sin | cos | tan | arctan> Δ numeric-formula

function_{of-numeric-type} ::= abs Δ numeric-formula

function_{of-numeric-type} ::= <exp | log.e> Δ numeric-formula

function_{of-numeric-type} ::= <log.2 | log.10> Δ numeric-formula

8.5 COMMANDS (Ref. Section 6)

command ::= define-command | execute-command | debug-command | optimization-command | count-command

debug-command ::= show Δ <symbolic | nothing> location-identifier Δ <number | nothing> <thru | nothing> Δ <location-identifier | nothing>

location-identifier ::= <name | cell Δ name> Δ <code-name | nothing>

optimization-command ::= optimize Δ <time | space | none> Δ statement

count-command ::= count Δ time Δ <statement | declaration>

define-command ::= <name. | nothing> Δ where Δ definition Δ then Δ translation Δ
<end | define-command-string>

definition ::= textual-formula | term | textual-pattern | not
Δ definition | definition Δ < or | and > Δ definition

define-command-string ::= begin Δ define-command <nothing | define-command> Δ end

pattern-declaration ::= pattern Δ name_{of-pattern} Δ definition

define-command-call ::= define Δ name_{of-define-command}

textual-pattern ::= pattern < (textual-constant) | (term) | (name-of-pattern)>

translation ::= declaration | statement | command

term ::= character | letter | digit | name | null | constant |
simple-statement | compound-statement | comment | item-declaration |
array-declaration | group-declaration | storage-declaration |
variable | hardware-operand | formula

execute-command ::= <name. | nothing> Δ where Δ definition Δ execute Δ item-
declaration Δ formula

8.6 EXTENDED SPL (Ref. Section 7)

numeric-constant ::= imaginary-constant | number | real-constant | binary-
constant | octal-constant | decimal-constant |
hexadecimal-constant

imaginary-constant ::= <number | real-constant> i

`complex-item-description ::= complex Δ number1 Δ minimum Δ <bit | digit |
nothing> Δ <nothing | -> number2 of-fractional-
or-exponent <bit | digit> Δ <nothing | float |
fixed> Δ <nothing | signed | unsigned> Δ <nothing
round | truncate>`

`code-declaration ::= code Δ nameof-code Δ code-description`

`code-description ::= <nothing | code-string> Δ number Δ <bit | digit | character
Δ <nothing | left | right>`

`code-string ::= textual-constant Δ <nothing | is code> Δ <nothing |
code-string>`

`code ::= number | textual-constant`

`list-declaration ::= list Δ name-string Δ list-description`

`list-description ::= full-list-description`

`full-list-description ::= <item-description | array-description | group-
description> Δ name-string`

`list-processing-statement ::= reference-statement | link-statement |
free-statement`

`reference-statement ::= see Δ list-element-reference`

`list-element-reference ::= nameof-list Δ <nothing | at Δ pointer-formula>`

`link-statement ::= link Δ <linkage-set | nameof-item-array-or-group>`

`linkage-set ::= linkage Δ <nothing | linkage-set>`

`linkage ::= <nothing | pointer-set Δ from> Δ list-element-reference`

`pointer-set ::= nameof-pointer Δ <nothing | pointer-set>`

`free-statement ::= free Δ <linkage-set | nameof-item-array-or-group>`

encode-decode-statement ::= <encode | decode> Δ variable Δ = Δ formula Δ by Δ
format

format ::= conversion-procedure-call | (Δ format Δ) | catenated-format |
repeated-format | conditional-format

conversion-procedure-call ::= name of-procedure Δ <nothing | actual-parameter |
(Δ actual-parameter-string Δ)>

catenated-format ::= format Δ <nothing | catenation-operator> Δ format

repeated-format ::= format Δ <nothing | catenation-operator> Δ (Δ repetition-
clause Δ)

conditional-format ::= closed-conditional-format | open-conditional-format

closed-conditional-format ::= if Δ condition Δ then Δ format₁ Δ else Δ format₂

open-conditional-format ::= if Δ condition Δ then Δ <format₃ | open-conditional-
format | Δ else Δ format₄>

algebraic-declaration ::= name-string Δ algebraic Δ <nothing | dimension-string
Δ array>

algebraic-variable ::= name of-algebraic-variable Δ <nothing | sub |
Δ index-string>

algebraic-formula ::= numeric-constant | algebraic-variable | atomic-operand |
(Δ algebraic-formula Δ) | <- | + > Δ algebraic-formula |
algebraic-formula Δ <- | + | * | / | ** | rem>
Δ algebraic-formula | <abs | floor | ceiling | exp | log.e |
log.2 | log.10 | sin | cos | tan | arctan | cosh | sinh |
tanh> Δ algebraic-formula | <min | max > Δ (Δ algebraic-
formula-string Δ)

algebraic-formula-string ::= algebraic-formula Δ <nothing | algebraic-formula>

atomic-operand-string ::= atomic-operand Δ <nothing | atomic-operand-string>

atomic-operand ::= 'numeric-formula' | 'sign-string' | textual-formula

algebraic-assignment-statement ::= <nothing | let> Δ algebraic-variable
 Δ = Δ <algebraic-formula |
function_{of-algebraic-type}>

algebraic-operator ::= (|) | + | - | * | ** | / | abs | floor |
ceiling | rem | exp | log.e | log.2 | log.10 | sin |
cos | tan | arctan | cosh | tanh | sinh | min | max

function_{of-numeric-type} ::= eval Δ algebraic-formula

function_{of-textual-type} ::= rep Δ algebraic-formula

function_{of-Boolean-type} ::= algebraic-formula Δ defined

function_{of-Boolean-type} ::= algebraic-formula Δ ident Δ algebraic-formula

function_{of-Boolean-type} ::= algebraic-formula Δ approx Δ (<numeric-formula |
algebraic-formula>)

function_{of-algebraic-type} ::= reduce Δ (algebraic-formula Δ textual formula)

function_{of-algebraic-type} ::= expand Δ algebraic-formula

function_{of-algebraic-type} ::= algebraic-formula₁ Δ coeff Δ algebraic-formula₂

function_{of-algebraic-type} ::= algebraic-formula Δ deriv Δ <algebraic-formula
| (algebraic-formula-string)> <nothing |
(where Δ dependencies)>

dependencies ::= atomic-operand Δ is Δ f (Δ atomic-operand-string Δ)
 Δ <nothing | dependencies>

procedure-call-statement ::= <nothing | call> Δ <accept Δ variable |
display Δ formula>

edit-command ::= edit Δ lines Δ <nothing | out | is Δ character-string

lines ::= <all | line-number | to Δ line-number | line-number Δ to
 Δ line number> Δ <nothing | lines>

line-number ::= number <nothing | line-number>
save-command ::= Δ title Δ = Δ <formula | lines>
title ::= name Δ <nothing | (all)>
get-command ::= get Δ <variable Δ = Δ title | title>

INDEX

	<u>Page</u>
Actual Parameter	72
Actual Parameter String	72
Algebraic Assignment Statement	109
Algebraic Declaration	109
Algebraic Formula	109
Algebraic Formula Manipulation	106
Algebraic Formula String	109
Algebraic Operator	109
Alphabet	13
Arithmetic Operator	15
Array	25
Array Declaration	25
Array Description	25
Assign Declaration	65
Assignment Operator	15
Assignment Statement	39
Atomic Operand	109
Atomic Operand String	109
Binary Constant	18
Binary String	18
Bit	18
Block Description	28
Boolean Constant	19
Boolean Formula	50
Boolean Item Description	23
Built-In Functions	76
Catenated Format	101
Catenated Variable	36
Catenation Operator	16
Character	13
Character String	18

	<u>Page</u>
Chronic Statement	60
Close Statement	66
Code	89
Code Declaration	88
Code Description	80
<hr/>	
Code Name	63
Code String	89
Commands	78
Comment	20
Complex Item Description	88
<hr/>	
Compound Statement	80
Compool-Declaration	38
Condition	51
Conditional Format	100
Conditional Operator	15
<hr/>	
Conditional Statement	56
Conditional Variable	37
Constant	17
Control Statement	54
Conversion Procedure Call	101
<hr/>	
Complex Control Statement	54
Count Command	79
Data Declaration	22
Debug Command	78
Decimal Constant	18
<hr/>	
Declaration	20
Declaration String	26
Define Command	80
Define Command Call	81
Define Command String	81
<hr/>	
Definition	80
Delayed Statement	59
Delimiter	26

	<u>Page</u>
Descriptor	16
Device Name	63
Digit	13
Dimension String	25
Discrimination Operator	16
<hr/>	
Edit Command	119
Editing Operator	16
Encode Decode Statement	100
Entry Statement	73
Execute Command	86
<hr/>	
Exit Statement	73
File Declaration	63
File Description	63
File Designation	66
Formal Parameter	70
<hr/>	
Formula	42
Free Statement	98
Full Integer Item Description	
Full Real Item Description	23
Function	67
<hr/>	
Function Declaration	74
Function Heading	74
Functional Operator	16
Functional Variable	67
Get Command	125
<hr/>	
Group	26
Group Declaration	26
Group Description	26
Hardware Operand	37

	<u>Page</u>
Hexadecimal Constant	18
Hexadecimal String	18
Imaginary Constant	87
Index	32
Index String	32
<hr/>	
Initial Value String	22
Input-Output Operator	16
Input-Output Statement	62
Item	22
Item Declaration	22
<hr/>	
Item Description	22
Item Value	22
Letter	13
Line Number	120
Lines	120
<hr/>	
Link Statement	95
Linkage	95
Linkage Set	95
List Declaration	92
List Description	92
<hr/>	
List Element Reference	93
List Processing Statement	93
Location Identifier	78
Location Operator	16
Logical Operator	15
<hr/>	
Mode Declaration	27
Mark	13
Matrix Operator	43
Name	14
Name String	22

	<u>Page</u>
Number	17
Numeral	17
Numeric Constant	17,87
Numeric Formula	43
Numeric Item Description	23
<hr/>	
N-ary Arithmetic Operator	43
N-ary Logical Formula	51
N-ary Relational Formula	51
Oct	18
Octal Constant	18
<hr/>	
Octal String	18
Open Conditional Format	101
Open Statement	66
Optimization Command	79
Operator	15
<hr/>	
Parallel Statement	57
Parameter Declaration	70
Parameter Declaration String	70
Pattern	81
Pattern Declaration	81
<hr/>	
Pointer Constant	19
Pointer Formula	49
Pointer Item Description	23
Pointer Set	95
Primitive	15
<hr/>	
Procedure Call Statement	72,120
Procedure Declaration	70
Procedure Heading	70
Program	21
Read Statement	68
Real Constant	17

	<u>Page</u>
Recursive	75
Reentrant	75
Reference Statement	93
Relational Formula	51
Relational Operator	15
<hr/>	
Repeated Format	101
Save Command	123
Sequential Operator	16
Signed	22,17
Simple Control Statement	54
<hr/>	
Space	13
Statement	20
Statement String	20
Stop Statement	58
Storage Declaration	28
<hr/>	
Store	28
Subscripted Variable	32
Symbol	13
Term	81
Textual Constant	18
<hr/>	
Textual Formula	49
Textual Item Description	23
Textual Pattern	81
Transfer Statement	54
Translation	81
<hr/>	
Value Sequence	54
Variable	30
Write Statement	68

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 <i>(Corporate author)</i> System Development Corporation 2500 Colorado Avenue Santa Monica, California 90406		2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	
		2b. GROUP	
3. REPORT TITLE Specification for Space Programming Language (SPL)			
4. DESCRIPTIVE NOTES <i>(Type of report and inclusive dates)</i>			
5. AUTHOR(S) <i>(First name, middle initial, last name)</i> Levi J. Carey, Al E. Kroger			
6. REPORT DATE August 1967		7a. TOTAL NO. OF PAGES 153	7b. NO. OF REFS
8a. CONTRACT OR GRANT NO.		9a. ORIGINATOR'S REPORT NUMBER(S) SAMSO TR-67-23	
b. PROJECT NO.		9b. OTHER REPORT NO(S) <i>(Any other numbers that may be assigned this report)</i> SDC TM-3719/000/00	
c.			
d.			
10. DISTRIBUTION STATEMENT This document has been approved for public release and sale; its distribution is unlimited			
11. SUPPLEMENTARY NOTES		12. SPONSORING MILITARY ACTIVITY Space and Missile Systems Organization Air Force Systems Command Los Angeles, California 90045	
13. ABSTRACT This document contains a complete specification of the Space Programming Language (SPL) in Backus-Naur form. A description of basic SPL and extensions is given. SPL is a space application language with a large array of capabilities. It is further an extendable language with punctuation rules and vocabulary designed for ease of learning and programming.			

14.

KEY WORDS

LINK A

LINK B

LINK C

ROLE

WT

ROLE

WT

ROLE

WT

Software
Spaceborne Software
Computer Programming
Computer Program Languages
Procedure-Oriented Languages
Space Programming Language
Programming Language Implementation

UNCLASSIFIED

Security Classification