SPECIFICATIONS FOR PARTITION ANALYSIS

Debra J. Richardson

COINS Technical Report TR-81-34
September 1981

Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

This research was funded in part by the National Science Foundation under grant NSFMCS 81-04202.

ABSTRACT

This report introduces a specification language, called SPA (Specifications for Partition Analysis), that has been used in conjunction with the partition analysis method. The SPA language is based on the operational approach to formal specification, whereby a procedure is described by an algorithm that models the desired behavior. The operational approach was chosen because it is both easy to use and amenable to analysis. Furthermore, the SPA language has abstract constructs that enhance the comprehensibility of operational specifications by facilitating the description of the behavior of a procedure at a high level of abstraction. The wide spectrum of descriptive capabalities provided by SPA enables the use of SPA throughout the pre-implementation phases of the software development process.

An overview of the software development process is provided, along with descriptions of some of the specifications that are developed throughout this process. The general role of program validation within software development is discussed, as well as the specific role of the partition analysis method. The SPA language is outlined and an example of the development of a procedure using SPA is provided.

TABLE OF CONTENTS

INTRODUCTION 1

PARTITION ANALYSIS IN SOFTWARE DEVELOPMENT 3

The Software Development Process 3
Program Validation 6
Module Specifications 8
Example of Module Development 13

THE SPA LANGUAGE 18

Language Overview 20
Syntax Notation 22
Type Declarations 23
Object Declarations 25
Expressions and Operations 26
Statements 38
Procedure Declarations 45
Abstract Data Type Declarations 46

REFERENCES 48

INTRODUCTION

To demonstrate the reliability of a program, there must be some mechanism that enables the determination of whether the program performs correctly. Often this mechanism is merely a prospective user who, when asked whether the results are correct, answers "yes" or "no" based on unstated requirements of the program. An alternative mechanism, and often a more dependable and versatile one, is the use of an independent description of the desired program behavior. Such a description is referred to as a problem specification and is often a product of the software development process. A problem specification provides an independent description of the external behavior of a program and thus provides an alternative, and usually more concise, representation to which an implementation of the program can be compared. The partition analysis method [RICH81a, RICH81c] incorporates information derived from such a specification with information derived from an implementation to develop common representations of the two descriptions of a program. These representations are then compared through the application of verification and testing techniques. By verifying that the implementation is consistent with the specification, which is assumed to be correct, and ascertaining that consistency through actual execution on a comprehensive set of test data, the partition analysis method provides assurance in the reliability of the implementation.

This report first provides an overview of the software development process, along with descriptions of some of the problem specifications that are developed throughout this process. Then, the general role of program validation within software development is discussed, as well as the specific role of the partition analysis method. Next, several approaches to formally specifying the problem to be solved are described and the types of problem specifications to which partition analysis is applicable are discussed. A specification language that has been used in conjunction with partition analysis is introduced. This language, which is called SPA (Specifications for Partition Analysis), enables descriptions of a program at varying levels of abstraction throughout the pre-implementation phases of software development. An example of the development of a procedure using SPA is provided. Finally, the SPA language is thoroughly described.

PARTITION ANALYSIS IN SOFTWARE DEVELOPMENT

The Software Development Process

Software development is a process of iterative refinement in which problem specifications of graduated levels of abstraction are developed. This process consists of several consecutive phases, where each phase refines the results of the previous phase. These phases are not always clearly distinguishable, but a useful delineation is to consider software development as a four-phase process -- requirements analysis, specification, design, and implementation. It may not always be possible to adhere to the ordering of these phases. The problem to be solved is not always fully understood throughout its development and it may become necessary to backtrack to correct any untimely decisions that result from a lack of understanding. Assuming a complete understanding of the problem, however, the problem to be solved is represented by successively more elaborate descriptions of the desired program behavior throughout the software development process. Even abandoning this idealistic view, these successive problem specifications provide a sequence of benchmarks that should be achieved in developing a program.

A program begins as a concept of some task that a prospective user would like to have performed by a computer. During the requirements analysis phase, this concept is put onto paper in the form of a requirements document. The analysis of the requirements is usually done by, or at least

in collaboration with, the prospective user, so this document is generally the user's perspective of the task the program should perform. The requirements document also describes the external program behavior expected by the user and thus defines the interface between user and program. The remainder of the software development process takes a divide-and-conquer approach in attempting to satisfy the requirements.

The specification phase is concerned with describing the program as a system of interacting modules. Current philosophy (which originated with Dijkstra [DIJK68]) proposes that a program be decomposed into hierarchical levels of abstraction. Each level is considered an abstract machine, which is composed of modules that are used to implement (by being called) the machine on the next higher level in the hierarchy. A clean decomposition of each abstract machine is achieved when each module corresponds to a concept that is useful in solving the problem posed by the next higher machine. Two types of modules have been found useful in decomposing the solution to a problem -- the procedure and the abstract data type. A module that corresponds to a procedure is a mapping from a set of input values to a set of output values. A module that corresponds to an abstract data type consists of a data type and a collection of allowable operations for objects of that type. The operations of an abstract data type are mappings and thus will eventually be implemented as procedures. The intended behavior of each module is described by a <u>module</u>

specification, which dictates the external behavior of a module, while the range of possible implementations, which describe how that behavior can be efficiently produced, remains broad. There is a wide variety of approaches for formally specifying both procedures and abstract data types that may be employed during the specification phase. Many of these approaches are discussed later in this section. The result of the specification phase is a system specification, which represents the hierarchical structure and the interactions among the modules and includes the module specifications.

During the design and implementation phases, the system specification is refined by choosing efficient data structures and selecting and optimizing algorithms for each of the modules. The refinement of a module is usually a step-wise process through which module designs of graduated levels of detail are developed. The system design consists of a set of corresponding module designs. In the implementation phase, the module designs are further refined and translated into the appropriate programming language, providing module implementations. This phase results in a set of module implementations that make up the final implementation of the system.

Thus, throughout the software development process, problem specifications are developed that proceed from the abstract to the concrete -- from a requirements document, to a system specification, to a system design, and finally to the system implementation. While there is a wide variety of

alternative views of the software development process in the literature on programming methodology, and the terminology varies greatly from one to another, the description of software development given here is fairly typical of most of the approaches that have been proposed.

Program Validation

In the traditional view of software development, program validation follows the implementation phase. The task of program validation is the demonstration of the reliability of the program. The two primary methods for accomplishing this task are program testing and program verification. Partition analysis is a program validation method that integrates verification and testing techniques. In applying these techniques, partition analysis utilizes both the problem specification as well as the implementation. The task of demonstrating program reliability is divided on the basis of the hierarchical decomposition of the problem. Partition analysis is applied to each module independently by comparing a module implementation to the associated module specification. The validation of the modules at each level in the hierarchy rests on the assumption that the modules at the lower level -- that is, those modules that are called -- are reliable. When each module corresponds to a clean abstraction, the interactions between modules are kept to a minimum, and demonstrating the reliability of an entire program amounts to demonstrating the reliability of its modules as well as

the interfaces between them. In this fashion, partition analysis demonstrates the reliability of the implementation with respect to the specification. The implementation can be deemed reliable, therefore, only if the reliability of the specification has been demonstrated. The correctness of the specification is a fundamental assumption of partition analysis.

It has become increasingly apparent, however, that in practice errors are more likely to be introduced during the pre-implementation phases of software development than during the implementation phase. Therefore, validation must be done, not only following the implementation phase, but throughout the entire software development process. In fact, it has been shown that design errors are inherently more difficult to detect and correct than implementation errors [BOEH75]. Clearly, it is important to develop capabilities to eliminate design errors when they occur rather than after the implementation phase has been completed. With this in mind, the reliability of each successive description of the problem -- be it the implementation, the system design, the system specification, or the requirements document -- should be checked. In this view, validation is an integral part of the software development process, where validation involves demonstrating that each newly-developed problem specification is consistent with the previous one.

Partition analysis is envisioned as a validation method that can be applied throughout the design and implementation phases. Although the method is typically used to demonstrate the consistency of a module implementation with a module specification, the techniques employed are equally applicable to a comparison of a module specification and a module design, a comparison of two module designs at different levels of detail, and a comparison of a module design and a module implementation. This extended application of partition analysis enables the demonstration of reliability throughout the design and implementation phases.

Module Specifications

The partition analysis method, whether applied to a module implementation and a module specification or to two module designs, relies on the use of a formal specification language. A formal specification language is one whose syntax and semantics can be precisely defined. There is a variety of approaches to formally specifying the behavior of modules, some of which depend on whether the module is an abstract data type or a procedure [LISK79]. The partition analysis method is appropriate only for certain types of module specifications. This section outlines several of the varied approaches for writing module specifications. The type for which partition analysis is currently most suitable is then discussed as well as why that particular type was chosen.

The approaches for formally specifying the behavior of a procedure fall into two main categories -- input/output specifications and operational specifications. By either approach, the specification describes the relations between the inputs and outputs of the module, and should be interpreted as what must be accomplished by a correct implementation but not how it must be accomplished. The input/output approach describes the behavior of a procedure by specifying pairs of assertions that constrain the relations between the inputs and outputs. Whenever the input values satisfy an input assertion, the corresponding output assertion specifies intended output values. Input/output specifications are generally expressed in the standard notation of mathematical logic and their use has been most extensive in proving program correctness [FLOY67, HOAR69, LOND75, NAUR66]. In the operational approach, the desired behavior of a procedure is described by an algorithm that models the function corresponding to that behavior. The operational approach is widely used in professional software development environments [CAIN75, DAVI77].

An abstract data type is specified by describing the functionality of the operations allowed on the data type. The three most widely studied approaches for specifying abstract data types are abstract model specifications, state machine model specifications, and axiomatic specifications. In the abstract model approach [HOAR72], the data type is explicitly defined by a representation in terms of another data type whose properties are well understood or specified

in advance. The operations of the abstract data type are specified in terms of the operations defined for the data type that is used as the model. Using the state machine model approach [PARN72, ROBI77, SILV79], the abstract data type is viewed as a machine and the behavior is specified by characterizing the states of the machine. In the axiomatic approach, of which algebraic specifications [GOGU75, GOGU79, GUTT75, ZILL75] are the most popular, an abstract data type is specified by listing axioms that characterize the operations on the data type in terms of each other. The effect of any sequence of operations must be deducible from the axioms provided.

A critical review of these approaches to formal specification led to the conclusion that the most appropriate language to use in conjunction with partition analysis is one that follows the operational approach. Employing the operational approach has some disadvantages. Most notably, this approach tends to result in specifications that are less concise and more apt to bias the implementation than specifications developed by the other approaches. On the other hand, operational specifications have numerous advantages. In particular, the algorithmic representations provided by operational specifications are amenable to the analysis techniques that have been established for procedural programming languages. Moreover, the operational approach appears to be more widely used outside of the academic community than the other approaches. It is easier for software developers to write

and comprehend specifications in operational languages because they are similar to programming languages and because the operational approach, unlike the other approaches, is not based on sophisticated mathematical theory [DAVI77]. Furthermore, a good operational specification language not only has constructs that are similar to programming language constructs but also has more abstract constructs that enhance the comprehensibility of operational specifications by facilitating the description of the behavior of a procedure at a high level of abstraction. This wide spectrum of descriptive capabilities enables the use of the operational approach throughout the pre-implementation phases of the software development process.

Several operational languages that are suitable for specifying procedures have been described. PDL [CAIN75] allows the specification of procedures in structured English, but also has facilities for describing a procedure in whatever level of detail is appropriate to the current phase in software development. PDL can be used, therefore, throughout the specification and design phases to create operational specifications. LAMBDA [JORD77] is another language that effectively permits the development of operational specifications of graduated levels of detail. LAMBDA has constructs for structuring data and can thus be used to specify abstract data types with the associated operations described in an operational way. Both LAMBDA and PDL are specification languages that are independent of any

programming language and can be used in the specification and design phases of software development but not in the implementation phase. Other operational approaches to specifying procedures have proposed the use of programming languages to write abstract implementations, which are later optimized to produce a concrete implementation. SETL [DEWA78] has been used as an operational specification language to describe algorithms with minimum attention to the description of data structures; all data are modelled by sets. Alphard [WULF76], Clu [LISK77], Gypsy [AMBL77], and Ada [ADA80] are high-level programming languages that attempt to promote the hierarchical development of programs and have been suggested for use in the pre-implementation phases of software development. Another operational specification language, CIP-L [BAUE79], has been designed to cover the spectrum of software development from specification through implementation. CIP-L contains a variety of constructs that provide a descriptive power richer than those provided by existing high-level programming languages. It allows formal problem specifications to be formulated in which operational descriptions can coexist with non-operational ones, which are gradually refined.

Although any of these operational specification languages could be used with partition analysis, none had all of the features that were deemed desirable. Consequently, SPA (Specifications for Partition Analysis) was defined. The SPA language incorporates constructs

capable of describing conditional values, finite summation and product, existential and universal quantification, assertions, non-determinism, and abstract data types, as well as the standard implementation-oriented constructs. Thus, although the language is basically operational, it has facilities for expressing specifications similar to those described by the other approaches. A more thorough description of SPA is provided in the next section. The SPA language is intended to be usable throughout the specification and design phases of software development. It is not proposed that SPA serve as the language of implementation -- that is, there is no plan to compile the low-level specifications -- but design can proceed to the point where the implementation phase is merely a translation of the lowest-level module designs into the appropriate programming language. The use of a single specification language throughout the specification and design phases greatly facilitates the extended application of partition analysis throughout the pre-implementation phases of software development.

Example of Module Development

To illustrate both the step-wise refinement process in software development and the SPA specification language, this section provides an example of the development of a single module, which is presumably part of a larger program.

Figure 1 gives the requirements for a procedure to determine whether a positive integer is prime. The module specification of PRIME, which is provided in Figure 2, is developed by specifying the mathematical properties of both a positive integer and a prime number. Figure 3 provides a refinement of this initial module specification, which makes use of the fact that if N has a factor greater than the square root of N then there is a corresponding factor less than the square root of N. Thus, if N has no factor less than or equal to its square root, it has no factor. The next module design, which is shown in Figure 4, takes advantage of the fact that odd numbers have only odd factors. Thus by first returning false if N is even, only the possibility of an odd factor must be checked for the remaining positive integers. The Ada implementation, which is taken from [WEGN80] and appears in Figure 5, is yet more efficient. By first returning false if N is divisible by either two or three, only the possibility of an odd factor that is not divisible by three must be checked for the remaining natural numbers. The application of the partition analysis method to the initial module specification and the module implementation of PRIME is provided in [RICH81b, RICH81c].

Input: positive integer N

Output: PRIME returns true if N is a prime number
or false if N is not a prime number.

Figure 1.
Requirements for Procedure PRIME

```
    procedure PRIME( N: in integer inset {1...})
              return boolean =
      -- specification
      -- PRIME returns true if N is a prime number
      -- or false if N is not a prime number
s   begin
      case
1     N = 1:
2       return false;

3     N = 2:
4       return true;

5     otherwise:
        -- if N has no factor < N-1, then N has no factor
6       return forall< i: integer inset {2..N-1} |
                       (N mod i /= 0) >;
      endcase;
f   end PRIME;
```

Figure 2.
Module Specification of PRIME

```
procedure PRIME( N: in integer inset {1...})
          return boolean =
begin
  case
  N = 1:
    return false;

  N = 2:
    return true;

  otherwise:
    -- if N has no factor <= sqrt(N), N has no factor
    return forall< I: integer inset {2..sqrt(N)}|
                    (N mod I /= 0 >;
  endcase;
end PRIME;
```

Figure 3.
Module Design of PRIME

```
procedure PRIME( N: in integer inset {1...})
          return boolean =
  FAC:integer;
  ISPRIME:boolean;
begin
  case
  N = 1:
    ISPRIME:= false

  N mod 2 = 0:
    -- if N is even and N /= 2, N is not PRIME
    ISPRIME:= (N = 2);

  otherwise:
    -- if N has no FACtor <= sqrt(N), N has no FACtor
    -- if N is odd, any FACtor of N is odd
    ISPRIME:= true;
    for FAC:= 3 to sqrt(N) by 2 loop
      if N mod FAC = 0 then
        ISPRIME:= false;
        exit;
      endif
    endloop;
  endcase;
  return ISPRIME;
end PRIME;
```

Figure 4.
Module Design of PRIME

```
     function PRIME( N: in integer range 2..max'int)
              return boolean is
       -- implementation in Ada
       -- PRIME returns true if N is a prime number
       -- or false if N is not a prime number
       FAC: integer;
       ISPRIME: boolean;

s    begin
1      if N mod 2 = 0 or N mod 3 = 0 then
         -- if N is even and N /= 2, N is not prime
         -- if N is divisible by 3 and N /= 3, N is not prime
2        ISPRIME:= (N < 4);

       else
         -- if N is odd, any FACtor of N is odd
         -- if N is not divisible by 3,
         --   N has no FACtor in the sequence 9,15,21,...
         -- if N has no FACtor <= sqrt(N), N has no FACtor
3        ISPRIME:= true;
4        FAC:= 5;
5        while FAC**2 <= N loop
6          if N mod FAC = 0 or N mod (FAC+2) = 0 then
7            ISPRIME:= false;
             exit;
           else
8            FAC:= FAC + 6;
           endif;
9        endloop;
       endif;
10     return ISPRIME;
f    end PRIME;
```

Figure 5.
Module Implementation of PRIME

THE SPA LANGUAGE

SPA (Specifications for Partition Analysis) is a specification language to be used in conjunction with the partition analysis method. The SPA language is considered an operational specification language because the sequence of statements implies an order in their evaluation. SPA, however, is more than just a high-level programming language. It incorporates facilities for describing conditional values, finite summation and product, existential and universal quantification, assertions, and nondeterminism. SPA is thus capable of representing both operational and input/output specifications for procedures. In addition, abstract data types can be described in SPA by specifying the behavior of the operations performed on those types. This section describes the SPA language. The constructs that are common to most specification and programming languages are described only briefly, while those that might be unfamiliar to the reader are discussed in greater detail along with examples.

The conception of the SPA language was influenced by the partition analysis method, and thus was designed with several goals in mind. First and foremost, the language is capable of specifying a large class of problems to be analyzed by the partition analysis method. There are constructs in SPA that can not currently be evaluated by the partition analysis method, but have been included with the intended purpose of allowing the expansion of the class of

problems. Second, the language is abstract enough to enable the specification of problems throughout the specification and design phases of program development. Thus, SPA can be used to create specifications of procedures that do not in any way constrain the implementation or to create low-level designs that closely resemble an implementation of the procedure. SPA is intended to enable the description of procedures that could be written in any algorithmic programming language, although certain capabilities -- e.g., concurrent activity, performance restrictions, and realtime constraints -- have been excluded because their analysis by the method was never intended. The language has been designed to facilitate the application of partition analysis. Although most programming languages are not as restrictive, SPA requires explicit control over the visibility of identifiers. One final design consideration was the ability to formally define the semantics of SPA. Although this has not yet been done, a well-defined semantics would be required for the automation of the partition analysis method. Several existing specification and programming languages have also influenced the design of SPA. These include SPECIAL [SILV79], CIP-L [BAUE78], Gypsy [AMBL77], Alphard [WULF76], Euclid [POPE77], CLU [LISK77], and Ada [ADA80].

Language Overview

A problem specification written in SPA is composed of one or more module specifications. A module specification is a procedure specification or an abstract data type specification.

A procedure specification is the unit for specifying the behavior of a function by a sequence of statements, which may correspond to actions or assertions. A procedure specification may reference formal input parameters, imported identifiers, and locally declared identifiers. A procedure may modify locally declared variables and formal output parameters. A value returning procedure has no formal output parameters and thus may modify only locally declared variables.

An abstract data type specification is the unit for specifying a data type and a collection of operations that can be performed on objects of that type. An abstract data type specification may reference imported identifiers and locally declared identifiers. An abstract data type exports a list of identifiers, which can be referenced by other modules to access objects of the data type. The behavior of the abstract data type is described by procedure specifications for the exported operations.

A module specification has two essential parts: a description of the behavior of the module and a description of the data that can be manipulated by the module. Behavior is described by statements. Data are described by type

definitions and expressions.

A data type defines a set of values and associated operations that can be performed on objects of that type. SPA has three classes of types: simple types, composite types, and abstract data types. The simple types include the predefined types, scalar types, and subtypes. The composite types include arrays, records, and sets. The abstract data type is a data type (either simple or composite) with an associated set of user-defined operations for objects of that type.

The statement part of a module specifies the behavior of the module. There are both simple statements and compound statements, which are sequences of statements. The simple statement is the assignment statement, which specifies that the value of an expression be assigned to a variable. A procedure invocation specifies the evaluation of a sequence of statements enclosed by the designated procedure. Loop statements specify the repeated evaluation of an enclosed sequence of statements. Case statements and if statements specify the selection of a sequence of statements based on the values of conditions.

The power in the SPA language lies in the expressions that can be created using the predefined operators. Along with the typical arithmetic, relational, and logical operators, SPA provides a wide variety of operations on sets. The usual set operators of union, intersection, and set difference are included, as is set membership. SPA also supports several operations that return a value when applied

to a finite set of values. These include the minimum and maximum value of a set, the sum and product of the values in a set, and some value and the unique value satisfying a specified condition. In addition, both universal and existential quantifiers can be applied to a set of values. SPA also provides for conditional expressions, which return a value depending on the values of the designated conditions. Expressions are built up in SPA from constants and variables using this wide variety of predefined operators along with user-defined procedures.

Syntax Notation

The notation used to describe the SPA syntax is an extended BNF (Backas-Naur Form). In this notation:

- upper case words, possibly containing hyphens, denote syntactic categories, where underscores indicate that the category is a terminal;
- lower case words denote reserved words;
- special characters that are terminals are enclosed in single quotes;
- the left side of a rule is separated from the right side by "::=";
- square brackets enclose optional items;
- curly brackets enclose repeated items, which may appear zero or more times;
- a vertical bar separates alternative items.

Type Declarations

A type determines the set of values that objects of that type may assume and a set of basic operations that can be performed on those objects. A type declaration associates an identifier with a type. There are three classes of types supported by SPA: simple types, composite types, and abstract data types. The abstract data type is a module in itself and is treated later in this appendix.

```
TYPE-DECLARATION::= type IDENTIFIER = TYPE-DEFINITION
TYPE-DEFINITION::= SIMPLE-TYPE-DEF
                 ¦ COMPOSITE-TYPE-DEF
                 ¦ ABSTRACT-DATA-TYPE-DEF
```

Simple Types

Simple types include predefined types, scalar types, and subtypes. The predefined types are integer, real, boolean, and character. A scalar type defines an unordered set of values by enumeration of the identifiers that denote the values. The only operations permitted for scalar types are equality and nonequality. A subtype is either a subrange type or a subset type. A subrange type is a subrange of another simple type defined by indicating the smallest and largest element in the subrange (an infinite subrange can be specified by replacing the largest element by '.'). A subset type is a subset of a simple type defined by indicating the values in the subset with a set expression (the constructs for set expressions are outlined below). A subtype of a simple type inherits the operations of that type.

```
SIMPLE-TYPE-DEF::= PREDEFINED-TYPE
                 ¦ SCALAR-TYPE-DEF
                 ¦ SUB-TYPE-DEF
PREDEFINED-TYPE::= integer
                 ¦ real
                 ¦ boolean
                 ¦ character
SCALAR-TYPE-DEF::= (IDENTIFIER, IDENTIFIER-LIST)
SUB-TYPE-DEF::= SUBRANGE-TYPE-DEF
              ¦ SUBSET-TYPE-DEF
SUBRANGE-TYPE-DEF::= [TYPE-INDICATION] RANGE-CONSTRAINT
SUBSET-TYPE-DEF::= SET-EXP
TYPE-INDICATION::= IDENTIFIER
                 ¦ TYPE-DEF
RANGE-CONSTRAINT::= ARITHMETIC-EXP '..' ARITHMETIC-EXP
                  ¦ ARITHMTIC-EXP '...'
```

Examples of Simple Type Declarations

Example of scalar type declaration:

```
type MONTHS = (JAN,FEB,MAR,APR,MAY,JUN,
               JUL,AUG,SEP,OCT,NOV,DEC)
```

Example of subrange type declaration:

```
type SUMMER = MONTHS JUN..AUG;
```

Example of subset type declaration:

```
type ODD = {I: integer ¦ I mod 2 /= 0};
```

Composite Types

The composite types include arrays, sets, and records. An array type is a structure consisting of components that are all of the same type. The elements of the array are designated by indices, which must be a subtype of type integer. A set type is the powerset of its component type, i.e., the set of all subsets of values of that type. A record type is the same as the Pascal record.

```
COMPOSITE-TYPE-DEF::= ARRAY-TYPE-DEF
                    ¦ SET-TYPE-DEF
                    ¦ RECORD-TYPE-DEF
ARRAY-TYPE-DEF::= array '['INDEX-TYPE {,INDEX-TYPE}']'
                  of COMPONENT-TYPE
INDEX-TYPE::= SUBRANGE-TYPE-DEF
```

```
COMPONENT-TYPE::= TYPE-INDICATION
SET-TYPE::= set of COMPONENT-TYPE
RECORD-TYPE::= record
                  IDENTIFIER':' COMPONENT-TYPE;
                 {IDENTIFIER':' COMPONENT-TYPE;}
               endrecord
```

Examples of Composite Type Declarations

Examples of array type declarations:

```
type LINE = array [integer 1..80] of character;

type      = array [integer 1..60] of LINE;

type MATRIX = array [integer 1..3,integer 1..3]
                 of real;
```

Examples of record type declarations:

```
type DATE = record
               DAY: integer 1..31
               MONTH: MONTHS
               YEAR: integer 1500..2000
            endrecord;

type COMPLEX = record
                  REALPART: real
                  IMAGPART: real
               endrecord;
```

Examples of set type declarations:

```
type DATESET = set of DATE;

type MATRICES = set of MATRIX;
```

Object Declarations

An object is an entity that contains a value of a given type. Object declarations introduce one or more named objects of a designated type. An object may be variable or constant. If constant, the initial value of an object is specified in the declaration and may not be modified.

```
OBJECT-DECLARATION::= IDENTIFIER-LIST':'TYPE-INDICATION
                         [constant][':=' EXPRESSION]
```

Expressions and Operations

An expression is a formula that specifies the computation of a value. Expressions consist of operands, operators, and functions. The rules of computation specify operator precedences according to eight classes of operators. Set intersection has the highest precedence, then set union and set difference, then the multiplying operators, then the adding operators, then the relational operators, then negation, then conjunction, and finally disjunction has lowest precedence. Sequences of operators with the same precedence are evaluated from left to right. The precedence may be overridden by the use of parentheses.

```
EXPRESSION::= LOGICAL-EXP
LOGICAL-EXP::= CONJUNCTION
             ¦ LOGICAL-EXP or CONJUNCTION
CONJUNCTION::= NEGATION
             ¦ CONJUNCTION and NEGATION
NEGATION::= RELATIONAL-EXP
          ¦ not RELATIONAL-EXP
RELATIONAL-EXP::= ARITHMETIC-EXP
                ¦ RELATIONAL-EXP REL-OP ARITHMETIC-EXP
ARITHMETIC-EXP::= SUM
SUM::= TERM
     ¦ ADD-OP SUM
     ¦ SUM ADD-OP TERM
TERM::= FACTOR
      ¦ TERM MULT-OP FACTOR
FACTOR::= PRIMARY
        ¦ PRIMARY EXP-OP PRIMARY
PRIMARY::= LITERAL
         ¦ OBJECT
         ¦ COMPOSITE-EXP
         ¦ DESCRIPTIVE-EXP
         ¦ CONDITIONAL-EXP
         ¦ FUNCTION-CALL
         ¦ ADD-OP FACTOR
         ¦ '(' EXPRESSION ')'
```

Object Denotations

Denotations of objects either designate an entire object or a component of an object. An entire object is denoted by its identifier. A component or group of components of an object is denoted by the identifier of the object followed by a selector specifying the components. The selector may specify an index of an array, a slice of a one-dimensional array, a field of a record.

```
OBJECT::= IDENTIFIER
        ¦ IDENTIFIER
            '['ARITHMETIC-EXP{,ARITHMETIC-EXP}']'
        ¦ IDENTIFIER '['RANGE-CONSTRAINT']'
        ¦ IDENTIFIER '.' IDENTIFIER
```

Composite Expressions

Composite expressions denote values of composite types and are formed by objects of those types and constructors. The composite expressions include array expressions, record expressions, and set expressions.

```
COMPOSITE-EXP::= ARRAY-EXP
               ¦ RECORD-EXP
               ¦ SET-EXP
```

Array expressions are specified with three types of array constructors, two of which are applicable only to one-dimensional arrays. Expressions for one- or multi-dimensional arrays can be constructed with an indexed constructor, which allows the array to be viewed as an unordered collection of components identified by index selectors. Thus, an array expression is constructed by associating a value with each index value. One-dimensional arrays can be constructed by a positional constructor, which requires that the array be viewed as an ordered collection

of components. An array expression is thus constructed by listing values by position to be associated with the indexed components of an array. SPA provides a shorthand notation for the positional construction of a one-dimensional array with component type character whereby the characters need not be separated -- that is, ('abc') is the same as ('a','b','c'). In addition, a one-dimensional array expression can be constructed with an iterative constructor, which supplies an index variable and an expression to be evaluated for each value designated for the index variable.

```
ARRAY-EXP::= IDENTIFIER
           | '(' ARRAY-CONSTRUCTOR ')'
ARRAY-CONSTRUCTOR::= INDEX-ASSOCIATION
                        {,INDEX-ASSOCIATION}
INDEX-ASSOCIATION::= [INDEX-RANGE '=>'] EXPRESSION
INDEX-RANGE::= '['ARITHMETIC-EXP{,ARITHMETIC-EXP}']'
             | CHOICE {,CHOICE}
             | OBJECT-DECLARATION
CHOICE::= ARITHMETIC-EXP
        | RANGE-CONSTRAINT
```

Record expressions can be constructed with two forms of record constructors. A field constructor allows the record to be viewed as an unordered collection of components identified by field selectors. Thus, a record expression is constructed by associating a value with each selector name, and does not depend on the order of the components in the record type definition. A positional constructor requires the record to be viewed as an ordered collection of components and disregards selector names. It constructs a record expression by listing values by position.

```
RECORD-EXP::= IDENTIFIER
            | '(' RECORD-CONSTRUCTOR ')'
```

```
RECORD-CONSTRUCTOR::= FIELD-ASSOCIATION
                          {, FIELD-ASSOCIATION}
FIELD-ASSOCIATION::= EXPRESSION
                   ¦ IDENTIFIER'=>' EXPRESSION
```

Set expressions are constructed from sets and set operators. Sets are specified by set constructors. An enumeration constructor enables the listing of the individual elements in a set. The elements of an enumeration constructor must be expressions of the same type. A property constructor supplies a type and a necessary and sufficient property of the elements. A subrange constructor is used to specify sets of numbers by indicating the smallest and largest elements in the subrange (infinite sets can be specified by replacing the largest element by a '.'). The set operators are set union (union), set intersection (inter), set difference (diff). Each set operator takes two set-valued arguments with the same component type as operands and results in a set-valued type with the same component type.

```
SET-EXP::= IDENTIFIER
         ¦ '{' SET-CONSTRUCTOR '}'
         ¦ SET-EXP SET-OP SET-EXP

SET-CONSTRUCTOR::= EXPRESSION {,EXPRESSION}
                 ¦ IDENTIFIER ':' TYPE '¦' LOGICAL-EXP
                 ¦ TYPE-INDICATION RANGE-CONSTRAINT
SET-OP::= union ¦ inter ¦ diff
```

Examples of Composite Expressions

Examples of indexed array expressions:

```
([0,0],[1,1]=>1.0, [0,1],[1,0]=>0.0)

(0=>0, 1=>2, 2=>4, 3=>6, 4=>8,
 5=>1, 6=>3, 7=>5, 8=>7, 9=>9)
```

Example of positional array expression:

```
(0,2,4,6,8,1,3,5,7,9)
```

Example of intensional array expression:

```
(I: integer 0..4 => 2*I,
 I: integer 5..9 => 2*(I-4)-1)
```

Examples of field record expressions:

```
(DAY=>4, MONTH=>JULY, YEAR=>1776)

(MONTH=>JULY, DAY=>4, YEAR=>1776)

(IMAGPART=>-1.5, REALPART=>6.75)
```

Examples of positional record expressions:

```
(4, JULY, 1776)

(6.75, -1.5)
```

Example of extensional set expression:

```
{2, 4, 6, 8, 10}
```

Examples of intensional set expressions:

```
{I: integer ¦ 0 < i < M}

{J inset LESS10 ¦ J mod 2=0}
```

Example of subrange set expression:

```
{I: 1..M-1}
```

Examples of set operations:

```
NLESS10 inter {J: integer ¦ J mod 2 = 0}

{1..10} diff {J: integer ¦ J mod 2 /= 0}
```

Descriptive Expressions

Descriptive expressions allow the specification of a value by describing its properties. Each descriptive expression describes a set that qualifies the range of values over which the properties are evaluated. This set may be described by local variable declarations, whose types

qualify the range of values. Any qualification variables serve as iteration counters for the expression and take on each value specified in the qualification. There are three types of descriptive expressions: the characterization expressions, the finite repetition expressions, and the quantification expressions.

```
DESCRIPTIVE-EXP::= CHARACTERIZATION-EXP
                 ¦ QUANTIFICATION-EXP
                 ¦ REPETITION-EXP
```

The characterization expressions (some, that, min, max) characterize one or more elements in a set by a stated logical expression. The characterization expressions operate on finite sets and provide a single-valued result, which is a value of a member of the set. The value of a some expression is the value of some element in the set that satisfies the stated logical expression. If the logical expression does not uniquely characterize an element, then the some expression may assume the value of any such element. If there is no element satisfying the logical expression, then the some expression is undefined. The value of a that expression is the value of the unique element in the set that satisfies the stated logical expression. If the logical expression does not uniquely characterize an element or if there is no element satisfying the logical expression, then the that expression is undefined. The value of a min expression is the minimum value in the set that satisfies the stated condition, while the value of a max expression is the maximum value in the set that satisfies the stated condition. If no logical

expression is stated in any of these characterization expressions, the logical expression is the value true and it is satisfied by all values in the set.

```
CHARACTERIZATION-EXP::= CHARACTERIZE-OP'<'QUALIFICATION
                        ['¦' LOGICAL-EXP] '>'
CHARACTERIZE-OP::= some ¦ that ¦ min ¦ max
QUALIFICATION::= SET-EXP
                 ¦ OBJECT-DECLARATION
                    {,OBJECT-DECLARATION}
```

The finite repetition expressions (sum, product) characterize the result of an operation repeated over all values in a set. The finite repetition expressions operate on finite sets and provide a single-valued result whose type is the component type of the set. The value of a sum expression is the sum of the values resulting from evaluating the stated arithmetic expression over each value in the set, while the value of a product expression is the product of the values resulting from evaluating the stated arithmetic expression over each value in the set.

```
REPETITION-EXP::= REPETITION-OP '<' QUALIFICATION
                  '¦' ARITHMTIC-EXP '>'
REPETITION-OP::= sum ¦ product
```

The quantification expressions (forall, exists) describe a formula in the first order logic. The quantification expressions operate on finite sets and provide a single-valued result. The value of a forall expression is true if the stated logical expression is true for all elements of the set and false otherwise. The value of an exists expression is true if the stated logical expression is true for some element of the set and false otherwise.

```
QUANTIFICATION-EXP::= QUANTIFY-OP '<' QUALIFICATION
                                  '¦' LOGICAL-EXP '>'
QUANTIFY-OP::= exists ¦ forall
```

Examples of Descriptive Expressions

Examples of characterization expressions:

```
some< J: integer ¦ j inset DIV2 >

that< Q,R: integer ¦ (0 <= Q <= N) and (0 <= R <= Q-1)
                     and (N=Q*D+R)>

min< X: real A..B ¦ F(X) >

max< G: integer inset GRADES >
```

Examples of finite repetition expressions:

```
sum <DIV2>

product< I: integer 1..N ¦ I >
```

Examples of quantified expressions:

```
forall< D: integer 2..N-1 ¦ N mod D /= 0>

exists< P: integer inset NLESSM ¦
       forall< D: integer 2..P-1 ¦ P mod D /= 0 > >
```

Conditional Expressions

A conditional expression (if, case) designates the selection of one of its component expressions for evaluation. Each component expression must be of the same type and that type is the type of the result. There are three types of conditional expressions: the if expression, the deterministic case expression, and the nondeterministic case expression.

```
CONDITIONAL-EXP::= IF-EXP
                 ¦ DETERMINISTIC-CASE-EXP
                 ¦ NONDETERMINISTIC-CASE-EXP
```

An if expression designates the evaluation of one or none of a number of expressions, depending on the truth value of the corresponding conditions. The logical expression specified after if and any logical expression specified after elseif are evaluated in succession until one evaluates to true; then the corresponding expression is evaluated and provides the value of the if expression. An else clause may be given as the last alternative and the corresponding expression is evaluated if all the logical expressions evaluate to false. If no else clause is given and all the logical expressions evaluate to false, the if expression is undefined.

```
IF-EXP::= if LOGICAL-EXP then
               EXPRESSION';'
          {elseif LOGICAL-EXP then
               EXPRESSION';'}
          [else EXPRESSION';']
           endif
```

The deterministic case expression designates the evaluation of one or none of a number of expressions depending on the value of the selector expression. The selector expression must be of a discrete type. Each alternative expression is preceded by a list of choices specifying the values of the selector for which the alternative is selected. The choice values must be of the type of the selector expression and must be mutually exclusive. An otherwise clause may be given as the choice for the last alternative to cover all values not given in previous choices.

```
DETERMINISTIC-CASE-EXP::=
     case EXPRESSION of
        CHOICE {','CHOICE}'=>' EXPRESSION';'
       {CHOICE {','CHOICE}'=>' EXPRESSION';'}
       [otherwise'=>'          EXPRESSION';']
     endcase
CHOICE::= ARITHMETIC-EXP
        | RANGE-CONSTRAINT
```

The nondeterministic case expression designates the evaluation of one or none of a number of expressions depending on the truth values of the corresponding conditions. Each alternative expression is preceded by a logical expression specifying the condition under which the alternative might be selected. The logical expressions need not be mutually exclusive. When more than one of the logical expressions evaluate to true, any of the corresponding expressions might be selected for evaluation and provide the value of the case expression, hence allowing for nondeterminism. An otherwise clause may be given as the last alternative, and the corresponding expression is evaluated only if all other logical expressions evaluate to false.

```
NONDETERMINISTIC-CASE-EXP::=
     case
        LOGICAL-EXP'=>' EXPRESSION';'
       {LOGICAL-EXP'=>' EXPRESSION';'}
       [otherwise'=>'   EXPRESSION';']
     endcase
```

Examples of Conditional Expressions

Example of if expression:

```
if N mod 2 = 0 then
   false;
else
   true;
endif
```

Example of deterministic case expression:

```
case DAY of
   MON..FRI=> RATE;
   SAT,SUN=>  RATE*1.5;
endcase
```

Example of nondeterministic case expression:

```
case
   (HOURS>40)=>                RATE*1.5;
   (DAY=SAT) OR (DAY=SUN)=> RATE*1.5;
   otherwise=>                 RATE;
endcase
```

Procedure Designators

A procedure designator specifies the evaluation of a value-returning procedure. It consists of the identifier of the procedure followed by a list of arguments. The arguments are expressions and their values are substituted for the coresponding formal parameters. The type of the value returned by the procedure is specified by the procedure declaration. The intrinsic functions, which are value-returning procedures, defined in SPA are trunc, which returns the conversion of a real number into an integer by truncation, round, which returns the conversion of a real number into an integer by rounding, abs, which returns the absolute value of a number, and sqrt, which returns the sqrt of a number.

```
PROCEDURE-DESIGNATOR::= IDENTIFIER
                        ['(' IDENTIFIER-LIST')']
```

Arithmetic Expressions

Arithmetic expressions are formed by function calls, descriptive expressions, objects, literals, and arithmetic operators. The arithmetic operators are the exponentiation operator (**), the multiply operators (*, /, mod, div), and the addition operators (+, -). Each operates on numbers, which are subtypes of type integer or real. Objects of both numeric types can be mixed. An operator having only integer arguments has an integer result; otherwise the result is real. The mod and div operators are defined only for integers.

```
ARITH-OP::= EXP-OP
          | MULT-OP
          | ADD-OP
EXP-OP::= '**'
MULT-OP::= '*' | '/' | mod | div
ADD-OP::= '+' | '-'
```

Relational Expressions

Relational expressions are formed by relational expressions, arithmetic expressions, set expressions, and relational operators. SPA allows a conjunction to be expressed as a single relational expression. The relational operators are the equality relations (=, /=), the inequality relations (<, <=, >, >=), and the set membership relations (inset, subset). The equality relations apply to any type. The inequality relations apply only to numbers, which are subtypes of type integer or real. The inset relation operates on one object of any type and a set-valued object with components of that type. The subset relation operates on two set-valued objects of the same component type. The

result of any relational expression is boolean.

```
REL-OP::= EQUALITY-OP
        ¦ INEQUALITY-OP
        ¦ SETMEMBER-OP
EQUALITY-OP::= '=' ¦ '/='
INEQUALITY-OP::= '<' ¦ '<=' ¦ '>' ¦ '>='
SETMEMBER-OP::= inset ¦ subset
```

Examples of Relational Expressions

Example of a conjunction expressed as a relational expression:

```
A <= B <= C
```

Examples of set memberships:

```
N inset DIV2

PLESS10 subset NLESS10
```

Logical Expressions

Logical expressions are formed by relational expressions and logical operators. The logical operators are negation (not), conjunction (and), and disjunction (or). The result of any logical expression is boolean.

```
LOG-OP::= not ¦ and ¦ or
```

Statements

Statements denote algorithmic actions and assertions. A statement may be simple or compound. A simple statement contains no other statement. A compound statement may contain simple statements and other compound statements.

```
STATEMENT-SEQUENCE::= STATEMENT
                      {STATEMENT}
STATEMENT::= SIMPLE-STATEMENT
           ¦ COMPOUND-STATEMENT
```

```
SIMPLE-STATEMENT::= ASSIGNMENT-STATEMENT
                  ¦ PROCEDURE-STATEMENT
                  ¦ EXIT-STATEMENT
                  ¦ RETURN-STATEMENT
                  ¦ ASSERT-STATEMENT
COMPOUND-STATEMENT::= CONDITIONAL-STATEMENT
                    ¦ LOOP-STATEMENT
```

Assignment Statements

The assignment statement serves to replace the current value of a variable by a new value specified as an expression. The value of an expression can be assigned to more than one variable by means of a multiple assignment statement.

```
ASSIGNMENT-STATEMENT::= OBJECT ':='
                        {OBJECT ':='} EXPRESSION ';'
```

The variable and the expression must be of the same type, with one exception; if the types of the variable and the expression are both subranges of the same type, the assignment is legal as long as the value of the expression is within the subrange of the variable.

For an assignment to an entire array variable or a slice of a one-dimensional array, each component of the array expression is assigned to the matching component of the array variable. For each component of the array variable, there must be a matching component in the array expression, and vice versa. Otherwise, the array variable becomes undefined.

For an assignment to an entire record variable, each component of the record expression is assigned to the matching component of the record variable. For each component of the record variable, there must be a matching

component of the record expression, and vice versa. Otherwise, the record variable becomes undefined.

Procedure Statements

A procedure statement designates the evaluation of the procedure denoted by the procedure identifier. The statement specifies the association of any actual arguments with formal parameters of the procedure. The correspondence is established by the postitions of the arguments and paramters in the lists.

```
PROCEDURE-CALL::= IDENTIFIER ['(' [EXPRESSION]
                              {, EXPRESSION} ')'] ';'
```

Exit Statements

An exit statement indicates that further evaluation should continue at the statement following the innermost enclosing loop statement. The when clause, if present, makes the transfer of control conditional.

```
EXIT-STATEMENT::= exit [when LOGICAL-EXP] ';'
```

Return Statements

A return statement indicates that control should return from the procedure currently being evaluated. The when clause, if present, makes the transfer of control conditional. A return statement in a value-returning procedure must include an expression whose value is the result returned by the procedure. The expression must be of the type specified in the return class of the procedure.

```
RETURN-STATEMENT::= return [EXPRESSION]
                           [when LOGICAL-EXP] ';'
```

Assert Statements

An assert statement specifies a logical expression that must be satisfied whenever the statement is evaluated. At any point, if the logical expression specified evaluates to false, the evaluation of the procedure is terminated; otherwise, evaluation continues under the specified assertions. When appearing as the first statement in a procedure, an assert statement specifies initial assertions that must be guaranteed by any procedure calling the given procedure. In this sense, the assert statement specifies assumptions on the input values that are taken for granted each time the procedure is invoked. When appearing as the last statement in a procedure, an assert statement specifies final assertions that the output values must satisfy.

ASSERT-STATEMENT::= assert '<' LOGICAL-EXP '>'

Conditional Statements

A conditional statement selects for evaluation one of its component statements. There are three types of conditional statements: the if statement, the deterministic case statement, and the nondeterministic case statement.

```
CONDITIONAL-STATEMENT::=
    IF-STATEMENT
  ¦ DETERMINISTIC-CASE-STATEMENT
  ¦ NONDETERMINISTIC-CASE-STATEMENT
```

An if statement designates the evaluation of one or none of a number of sequences of statements, depending on the truth value of the corresponding conditions. The logical expression specified after if and any logical expressions specified after else if are evaluated in

succession until one evaluates to true; then the corresponding sequence of statements is evaluated. An else clause may be given as the last alternative and the corresponding sequence of statements is evaluated if all the logical expressions evaluate to false.

```
IF-STATEMENT::= if LOGICAL-EXP then
                     STATEMENT-SEQUENCE
                {elseif LOGICAL-EXP then
                     STATEMENT-SEQUENCE}
                [else
                     STATEMENT-SEQUENCE]
                 endif
```

The deterministic case statement designates the evaluation of one or none of a number of sequences of statements depending on the value of the selector expression. The expression must be of a discrete type. Each alternative sequence of statements is preceded by a list of choices specifying the values of the selector for which the alternative is selected. The choice values must be of the type of the selector expression and must be mutually exclusive. An otherwise clause may be given as the choice for the last alternative to cover all values not given in previous choices.

```
DETERMINISTIC-CASE-STATEMENT::=
     case EXPRESSION of
        CHOICE{','CHOICE}'=>' STATEMENT-SEQUENCE';'
       {CHOICE{','CHOICE}'=>' STATEMENT-SEQUENCE';'}
       [otherwise'=>'          STATEMENT-SEQUENCE';']
     endcase';'
CHOICE::= ARITHMETIC-EXP
        | RANGE-CONSTRAINT
```

The nondeterministic case statement designates the evaluation of one or none of a number of sequences of statements depending on the truth values of the corresponding conditions. Each alternative sequence of statements is preceded by a logical expression specifying the condition under which the alternative might be selected. The logical expressions need not be mutually exclusive. When more than one of the logical expressions evaluates to true, any of the corresponding sequences of statements might be selected for evaluation; hence allowing for nondeterminism. An otherwise clause may be given as the last alternative and the corresponding sequence of statements is evaluated only if all other logical expressions evaluate to false.

```
NONDETERMINISTIC-CASE-STATEMENT::=
     case
        LOGICAL-EXP'=>' STATEMENT-SEQUENCE';'
       {LOGICAL-EXP'=>' STATEMENT-SEQUENCE';'}
       [otherwise'=>'    STATEMENT-SEQUENCE';']
     endcase
```

Loop Statements

Loop statements specify that a sequence of statements is to be evaluated zero or more times. A loop may have an iteration clause, which controls the repeated evaluation of the loop. A loop may also be left as the result of an exit or return statement. Three iteration clauses are provided: the for iteration clause, the while iteration clause, and the until iteration clause.

```
LOOP-STATEMENT::= [ITERATION-CLAUSE]LOOP-BODY
```

```
ITERATION-CLAUSE::= for IDENTIFIER GENERATOR
                  ¦ while LOGICAL-EXP
                  ¦ until LOGICAL-EXP
GENERATOR::= in DISCRETE-RANGE
           ¦ in SET-EXP
           ¦ ':=' ARITHMETIC-EXP to ARITHMETIC-EXP
                    by ARITHMETIC-EXP
LOOP-BODY::= loop
                 STATEMENT-SEQUENCE
             endloop
```

The evaluation of a loop statement with a for iteration clause begins with the evaluation of the for clause, which serves to declare the specified identifier. The identifier represents the loop parameter, which is a constant within the loop body. The loop body is evaluated once for each value specified by the generator (subject to the loop body being left by an exit or return statement). Prior to each such iteration, a designated value is assigned to the loop parameter.

In a loop statement with a while iteration clause, the logical expression is evaluated before each iteration (including the first) of the loop body. If it evaluates to false, the loop statement is terminated.

In a loop statement with an until iteration clause, the logical expression is evaluated after each iteration of the loop body. If it evaluates to true, the loop statement is terminated.

Procedure Declarations

A procedure declaration serves to define a module that corresponds to a sequence of statements and to associate with it an identifier so that it can be activated by procedure statements.

```
PROCEDURE-DECLARATION::= PROCEDURE-HEADING
                         LOCAL-DECLARATIONS
                         PROCEDURE-BODY
PROCEDURE-HEADING::=
     procedure IDENTIFIER ['FORMAL-PARAMETER-LIST']
               [imports IMPORT-LIST]
               [return TYPE] '='
FORMAL-PARAMETER-LIST::= '('FORMAL-PARAMETER
                           {,FORMAL-PARAMETER}')'
FORMAL-PARAMETER::= IDENTIFIER ':' MODE TYPE
MODE::= in | out | in out
IMPORT-LIST::= IDENTIFIER-LIST
LOCAL-DECLARATIONS::= {TYPE-DECLARATION}
                      {OBJECT-DECLARATION}
PROCEDURE-BODY::= begin
                    STATEMENT-SEQUENCE
                  end';'
```

The formal parameters of a procedure are considered local to the procedure. A parameter whose mode is in acts as a local constant whose value is provided by the corresponding actual argument before evaluation of the procedure. A parameter whose mode is out acts as a local variable whose value is assigned to the actual argument after evaluation of the procedure. A parameter whose mode is in out acts as a local variable whose initial value is provided by the corresponding actual argument before evaluation of the procedure and whose value is assigned to the actual argument after evaluation of the procedure. If no mode is explicitly given for a parameter, the mode in is assumed.

The import list of a procedure specifes identifiers that are not local to the procedure but may be referenced (but not defined) within the procedure. These identifiers may denote objects, procedures, or types accessible to the calling procedure.

A procedure heading with a return clause designates a value-returning procedure (function). Such a procedure's formal parameters must all have mode in.

Abstract Data Type Declarations

An abstract data type declaration serves to define a module that corresponds to a data type and a set of allowable operations on that type and to associate with it an identifier so that objects of the type can be declared and manipulated.

```
ABSTRACT-DATA-TYPE-DECLARATION::=
       type IDENTIFIER '=' ABSTRACT-DATA-TYPE
ABSTRACT-DATA-TYPE::= ADT-HEADING
                      LOCAL-DECLARATIONS
                      PROCEDURE-DECLARATIONS
                      ADT-BODY
ADT-HEADING::= TYPE-DEFINITION
               [imports IMPORT-LIST]
               [exports EXPORT-LIST]
EXPORT-LIST::= IDENTIFIER-LIST
PROCEDURE-DECLARATIONS::= {PROCEDURE-DECLARATION}
ADT-BODY::= begin
              STATEMENT-SEQUENCE
            end
```

The import list of an abstract data type specifies identifiers that are not local to the module but may be referenced (but not defined) within the module. These identifiers may denote objects, procedures, or types accessible to the procedure in which the abstract data type

is declared.

The export list specifies the identifiers that can be accessed outside of the abstract data type. The procedures designated in the export list are thus the operations allowed for objects of the type.

The sequence of statements in the body designate actions that serve to initialize an object of this type. The statements are evaluated when an object is declared (just as an initial value can be assigned for a simple type).

REFERENCES

ADA80 Reference Manual for the Ada Programming Language, United States Department of Defense, July 1980.

AMBL77 A. Ambler, D. Good, J. Browne, W. Burger, R. Cohen, C. Hoch and R. Wells, "GYPSY: A Language for Specification and Implementation of Verifiable Programs," Proceedings of the ACM Conference on Language Design for Reliable Software, March 1977, 1-10.

BAUE79 F.L. Bauer, M. Broy, R. Gnatz, W. Hesse, B. Krieg-Bruckner, H. Partsch, P. Pepper, and H. Wossner, "Towards a Wide Spectrum Language to Support Program Specification and Program Development," Program Construction, Lecture Notes in Computer Science, Springer-Verlag, 1979.

CAIN75 S.H. Cain and E.K. Gorden, "PDL -- A Tool for Software Design," Proceedings of the National Conference on Computers 75, 1975, 271-276.

DAVI77 C.G. Davis and C.R. Vick, "The Software Development System," IEEE Transactions on Software Engineering, SE-3, 1, January 1977, 69-84.

DEWA78 R.B.K. Dewar, A. Grand, S. Liu, E. Schonberg, J.T. Schwartz, "SETL as a Tool for Generation of Quality Software," Constructing Quality Software, editors P.G. Hibbard and S.A. Schuman, North Holland Publishing Company, 1978.

FLOY67 R.W. Floyd, "Assigning Meaning to Programs," Proceedings of a Symposium in Applied Mathematics, 19, American Mathematical Society, 1967, 19-32.

GOGU75 J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright, "Abstract Data Types as Initial Algebras and the Correctness of Data Representations," Proceedings of the Conference on Computer Graphics, Pattern Recognition and Data Structures, 1975, 89-93.

GOGU79 J.A. Goguen and J.J. Tardo, "An Introduction to OBJ:," Proceedings of the Conference on Specifications of Reliable Software, 1979, 170-189.

GUTT75 V. Guttag, "The Specification and Application to Programming of Abstract Data Types," Ph.D. Dissertation, University of Toronto, Canada, 1975.

HOAR69 C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," Communications of the ACM, 12, 10, October 1969, 576-583.

HOAR72 C.A.R. Hoare, "Proof of Correctness of Data Representations," Acta Informatica, 1, 1972, 271-281.

JORD77 E.A. Jordan, "A Support for Program Design with Abstract Machines," Information Processing 77, North Holland Publishing Company, 1977.

LISK79 B.H. Liskov and V. Berzins, "An Appraisal of Program Specifications," Research Directions in Software Engineering, editor P. Wegner, MIT Press, 1979.

LOND75 R.L. London, "A View of Program Verification," Proceedings International Conference on Reliable Software, April 1975, 534-545.

PARN72 D.L. Parnas, "A Technique for Software Module Specification with Examples," Communications of the ACM, 15, 5, May 1972, 330-336.

POPE77 G.J. Popek, J.J. Horning, B.W. Lampson, J.G. Mitchell, and R.L. London, "Notes on the Design of Euclid," Proceedings of the ACM Conference on Language Design for Reliable Software March 1977, 11-18.

RICH81a D.J. Richardson, L.A. Clarke, "A Partition Analysis Method to Increase Program Reliability," Fifth International Conference on Software Engineering, March 1981, 244-253.

RICH81b D.J. Richardson, "Examples of the Application of the Partition Analysis Method," Department of Computer and Information Science, University of Massachusetts, TN-48, August 1981.

RICH81c D.J. Richardson, "A Partition Analysis Method to Demonstate Program Reliability," Ph.D. Dissertation, University of Massachusetts, September 1981.

ROBI77 L. Robinson and K.N. Levitt, "Proof Techniques for Hierarchically Structured Programs," Communications of the ACM, 20, 4, April 1977, 271-283.

SILV79 B.A. Silverburg, L. Robinson, and K.N. Levitt, "The Languages and Tools of HDM," Stanford Research Institute Project 4828, June 1979.

WULF76 W.A. Wulf, R.L. London, and M.A. Shaw, "An Introduction to the Construction and Verification of ALPHARD Programs," IEEE Transactions on Software Engineering, SE-2, 4, December 1976, 253-265.

ZILL75 S.N. Zilles, "Data Algebra: A Specification Technique for Data Structures," Ph.D. Dissertation, Massachusetts Institute of Technology, Cambridge, MA, 1975.