F. Beichter
O. Herzog
H. Petzsch

SLAN-4: A Language for the Specification and Design of Large Software Systems

The language SLAN-4 has been defined in view of the need for formal tools supporting the specification and design of large software systems. It offers its users language constructs for algebraic and axiomatic specifications as well as for design in pseudocode. One of its major design goals has been to ease subsequent refinements of a (given) specification. The user can start his development with an informal high-level specification which can be formalized and implemented at a later date by using lower-level concepts. This paper provides the formal definitions of the SLAN-4 language, discusses the design decisions, and presents examples for the use of the syntactic constructs.

Introduction

When one talks about computer languages, one normally means the languages used to instruct and control computers, i.e., the languages for communication between programmers and machines. However, most of the time programmers do not communicate with computers but rather with their fellow programmers and other people, e.g., to design a piece of software, to implement it, or to document it. Because the normal computer languages are generally inadequate for this kind of communication, programmers have invented other communication vehicles. For casual discussion, they use a mixture of computer and natural languages; for written communication, they use restricted natural languages, formal design languages, graphic languages, etc. Since each of these languages was designed for a specific purpose, each is less well suited for other purposes. Usually a single piece of software is described by using three different types of languages (excluding casual discussions): a specification and design language, a compilable computer language, and a graphic documentation language. We believe that this situation should be improved, and that eventually the programmer needs but one language for most stages of software production.

In this paper we propose SLAN-4, a Software LANguage spanning the complete range from an almost natural language to an almost compilable language, which can be used as a software specification, design, communication, and documentation tool.

During the design of SLAN-4 we used the following guidelines:

- The language should allow the programmer to proceed in a uniform way from specification to implementation. Concepts that are initially vague ultimately become precise through formal specification, refining, and detailing.
- The language should allow one to define abstract data types and data objects with varying degrees of detail. It should emphasize the definition of data types together with the operations allowed on them.

A design principle of SLAN-4 is that informal descriptions in the form of comments may be used as placeholders for formal language constructs. This possibility allows a user of SLAN-4 to start with a specification written in natural language, but in a structural way. Thereafter, during the development cycle, the informal specifications without formal semantics can be made precise by formalizing the informal constructs, e.g., by writing out the axioms of an algebraic specification of an abstract data type, by providing

precise object type descriptions, or by supplying complete predicates for axiomatic specification. As the number of formally defined constructs increases, it is possible to check, for instance, the consistency of interfaces and the correct use of defined objects very early in the development process.

SLAN-4 incorporates four approaches to software specification:

- abstract data types,
- algebraic specifications,
- specification of operations by means of pre- and post-conditions, and
- design with pseudocode.

The concept of abstract data types represents the design philosophy of a hierarchical, data-oriented approach to specification. Algebraic specification, pre- and post-conditions, and pseudocode are formal tools for the description of design decisions incorporating different levels of abstraction. The algebraic method is not concerned with variables and other objects manipulated by a program. The pre- and post-conditions specify the result of a computation in a model-oriented axiomatic approach, yet in a nonoperational way. Pseudocode offers high-level language constructs including high-level data types for algorithmic specifications.

SLAN-4 was initially designed by F. Beichter, O. Buchegger, N. E. Fuchs, and O. Herzog [1] of the IBM System Products Division Laboratory in Boeblingen, Germany. It was designed as a formal language for use in the development of large software systems for specifications, design, communication, and documentation.

Lexical rules

Since computer terminals normally do not support different type styles, SLAN-4 refrains from using numerous font types for specifications (bold keywords, comments in italics, etc.) The only convention adopted is that keywords and other words with a predefined meaning in SLAN-4 are written with uppercase letters; user-defined names use only lowercase letters.

The syntactic notation uses the symbols shown with their definitions in Table 1.

The syntax follows the general rules that 1) nesting of SLAN-4 constructs is allowed to any depth; 2) optional terms may appear in any order; 3) the user may replace almost every formal syntactic entity with a comment; and 4) SLAN-4 does not require input formatting.

As is usual in computer languages, there are lexical rules and guidelines in addition to the rules given in Backus-Naur form, e.g., definition of delimiters, compound symbols, and comments. Appendix 1 lists these rules and guidelines.

Table 1 Symbols used in syntactic notation in SLAN-4.

Symbol	*Meaning*
:=	"is defined by"
$[\cdot]_{}$	an optional term
$[\cdot]^{0}$	optional repetition of a term; the 0 indicates that it may be left off
$[\cdot]^{1}$	optional repetition, but the 1 indicates that at least one term is needed
$\cdot \mid \cdot$	alternatives—one must be chosen
'empty'	the empty string (equivalent to an arbitrary number of blanks)
uppercase letters special characters	must be written as they stand
lowercase letters	variable terms, to be replaced

Classes and modules

The basic forms of specification in SLAN-4 are *classes* and *modules*. While classes define data types, modules represent operations such as procedures in high-level programming languages. Whereas classes are used to group several operations together, modules describe what will be perceived as a single action by future "callers" of the module.

Descriptions of classes and modules are self-explanatory because all information needed to understand a class or module is contained within the construct. This implies that no object being defined outside a module or class may be used within a module or class without explicitly "importing" the object and its definition. Given a class (module) consisting of the interface declaration and the class (module) specification, the task of refining the specification by giving more details can be performed without knowledge of other parts of the whole specification. Both constructs control the visibility of data and (inner) operations, thus serving the information-hiding principle.

Modules and classes may be nested inside one another (and of course within themselves). In this way, SLAN-4 supports structured top-down development of software as refinements of modules or classes can be developed to reflect the hierarchical levels of a software system.

- *The class construct*

The *class* construct defines both the data type, i.e., the basic types that make up its structure, and all the operations allowed on objects of this data type. The class construct incorporates two different concepts: classes as seen by Simula [2] and classes as (algebraically specified) abstract data types (see for example [3]). More details on algebraic specification methods are given in the section on class specification. For the rest of this discussion, we concentrate on the first concept of classes.

The class construct defines a data type, i.e., a data structure or a data structure together with associated procedures defining the operations on the data structure. The class concept enforces abstraction and information hiding because the use of operations is completely separated from their implementation. This is essential for the implementation of large systems because it increases their integrity and makes them easier to read and maintain:

```
class := class-name:CLASS
                    [interface-declaration]
                    [class-specification]
                    [declaration]
                    [class | module]°
                    [stmts]
                  ENDCLASS class-name
```

The interface-declaration part describes the data objects which are exchanged with the external environment of the class. The class specification, which is discussed more fully later, defines the operations of the class using equations of operations; furthermore, information on sequencing constraints may be included. The declarations may describe the data objects used internally within the class in terms of more basic types. The statement part of a class describes the actions being performed at the time of *instantiation*, i.e., declaration of an object of this (class) type. The module construct in a class definition is intended to be used as a refinement of the specification of a data type operation defined by that class. An example for the *class* construct is given later in connection with the definition of a class specification; "stmts" is defined in the section on pseudocode.

Through parameterization, classes allow the construction of generic data types. With their extended descriptive power (compared to subroutines) they are better suited for forming prebuilt libraries. Most of the discussion on the advantages of "packages" given in the ®Ada design rationale [4] is also valid for the SLAN-4 class construct. Therefore, we do not repeat the arguments but instead highlight the most important aspects of SLAN-4 by giving an example.

Since SLAN-4 does not introduce any dynamic objects such as pointers, there is no need to consider the semantics of dynamic classes. We are convinced that, for specification purposes, static classes are fully sufficient. As an example, we refer to the class *directory*, which is parameterized by the type of element entry. This type must be transferred upon instantiation of the class:

```
directory: CLASS
          << of all operations upon unordered directories >>
          INTERFACE
            entry_type : PARAMETER(TYPE)
                    << gives type of a directory element>>
            create,insert,
            delete,list : EXPORT(OPERATION)
                                  << allowed operations>>
          ENDINTERFACE
          DECLARATION
            entry,entry 1: entry_type
                    << names of directory elements>>
            dir: RECORD
                    <<data structure of the directory >>
                - dir_descriptor:
                    RECORD
                      - total_number_dir_entries: INTEGER.
                      - number_dir_entries: INTEGER.
                    ENDRECORD
                - dir_entries: SET OF entry_type.
              ENDRECORD.
          ENDDECLARATION
          << functional description of operations to follow >>
        ENDCLASS directory
```

- *The module construct*

The notion of *module* refers to a procedural or functional entity of a program, regardless of whether it is seen as a separately compilable unit or as a subroutine within the program. The nesting of modules can be regarded as an Algol-like block structure, with similar scope rules for names. A major difference from the Algol-60 concept, however, is that the declaration of each name used in the module must be stated explicitly in its declaration or interface part. Thus, a module acts like a self-contained unit, where the programmer must specify all objects used and imported by that module; this enforces a complete interface declaration which is essential to the specification, design, and implementation of large software systems.

A module represents a part of an algorithm which we want to regard as a functional entity (i.e., an indivisible operation to those who call it):

```
module  := module-name: MODULE
                        [interface-declaration]
                        [RESULT type .]
                        [module-specification]
                        [declaration]
                        [class | module]°
                        [stmts]
                      ENDMODULE module-name
```

The interface-declaration part describes the data objects which are exchanged with the external environment of the module. For example,

```
directory : CLASS
          ...
          create : MODULE
                    INTERFACE
                      entry-type : IMPORT(TYPE).
                      dir : IMPORT(WRITE).
                    ENDINTERFACE
                      << functional description to follow >>
                  ENDMODULE create
        ENDCLASS directory
```

If the module has to return a value (like a function in high-level programming languages) the type of the value is defined in the *result* clause. The module specification defines the effect which the execution of the module has on the variables of its environment. The declaration of inner objects, classes, and modules, together with a block of executable

statements (given in SLAN-4 pseudocode), describes a refinement of the module specification. An example for the module construct is given in connection with the definition of a module specification.

Declarations

While SLAN-4 is very strict in requiring the existence of declarations for each name occurring in a module/class, (type) attributes of a name may be given loosely as a comment at first. Later, in the software development cycle, the informal description can be sharpened to any degree of detail by referring either to basic data types or by detailing the structure of the given object.

Every name used in a specification must be defined as belonging to a class, module, parameter, type, or object. The definitions of (non-class) types and objects are given within the declaration:

```
declaration := DECLARATION
                    [type-declaration | object-declaration]^0
               ENDDECLARATION
```

All names which are defined within a declaration are local to the class/module in which the declaration resides. For example,

```
DECLARATION
    TYPE arr1 : ARRAY ( 1. .10) OF INTEGER.
    a1,a2 : arr1 << input/output-arrays>>
    c1,c2 : INTEGER << indices for a1/a2>>
ENDDECLARATION
```

- *Type declarations*

A data type is defined by the set of its elements, together with the operations which may be performed on the elements. A *type* definition has the form

```
type-declaration := TYPE name [,name]^0 : type.

type := type-name | record | array | set | list |
        semaphore | enumeration | subrange |
        class-name parameterlist | 'empty'
```

We distinguish among simple types, structured types, and class types. In the first (simple types), the elements and the operations are predefined. In structured types, the composition of elements is defined by the user, whereas the operations (i.e., selections of substructures) are predefined with the structure. In class types, both the construction of elements and the definition of operations are given explicitly.

The simple and structured types offered by SLAN-4 are well known from high-level programming languages like ®Ada or Pascal. The use of sets, lists, and arrays (mappings) is supported because they offer a framework for describing data concepts in a mathematical and abstract way. This allows programmers to deal with objects by using a conceptual view within the specification part, rather than by committing themselves early to an implementation-oriented data structure.

From the conceptual point of view, there is no difference between a general mapping and an array with an infinite index type. We therefore did not introduce separate keywords for "arrays" and "mappings," but included several predefined functions on arrays. These standard functions are introduced in [5] as operators on partial mappings. Their semantics may be defined in terms of arrays by introducing an "undefined value," which is used for initalization.

Semaphores are introduced because they are a very powerful and universal synchronization tool. The disadvantage stemming from the universality of semaphores is that their use can be quite "dangerous" for a program's control flow, because inadvertent deadlocks may be programmed. By using the results described in [6], such control-flow anomalies can be detected by the static analysis of the SLAN-4 text. In this way, one can prove automatically at specification time the absence of those control flow anomalies.

The possibility of defining new types via the class construct improves considerably the descriptive flexibility of SLAN-4. As far as declarations are concerned, a class name denotes a type just as INTEGER does. Difficulties may arise with respect to referencing the operations and objects defined in a class. There are two possibilities:

- Let x be an instance of a class. An operation (op) may depend on objects local to the class, as in the case of "push" on a stack; push will perform some changes on the local "store" of the class. It looks very natural to qualify the operation with the name of the data object either like Alphard [7] or CLU [8], namely op(x,. . .); or in the way of Simula [2] or ®Ada [9], x. op(. . .).
- The operation does not depend on any internal data of the class, as is the case with an operation which adds two complex values and returns the result in a third parameter. We feel that "complex.add ($x1$,$x2$,y)" is the most natural notation for this situation. This is much more readable than "x.add (. . .)" because "add" does not depend on the value of "x."

To be able to handle both situations appropriately SLAN-4 offers the concept of *anonymous declarations*: each time an operation is prefixed only with a class name, a new incarnation of the class is generated and referred to. In contrast to class operations incarnated in a declaration part, operations which belong to an anonymous incarnation are therefore not able to retain information between two calls. Thus the use of anonymous incarnations emphasizes that an operation does not depend on internal data of a given class (but of course it may use internal objects to store intermediate results). Note that it is not possible to use this concept in connection with parameterized classes. (For an example of a class declaration, see the last section on the module construct.)

Table 2 Basic data types and operations associated with them.

Operator	*Operation*	*Type of operand(s)*[a]	*Result_type*	*Priority*
arithmetic:				
+(unary)	identity	I, R	same as operand	1
−(unary)	sign inversion	I, R	same as operand	1
+	addition	I, R	I, R	4
*	multiplication	I, R	I, R	3
−	subtraction	I, R	I, R	4
/	division	I, R	R	3
MOD	modulus : a MOD b = a − [(a/b)*b]	I	I	3
**	exponentiation	I, R	I, R	2
relational:				
=	equality	I, R, C, S, B	B	5
<>, ¬=	inequality	I, R, C, S, B	B	5
<	less than	I, R, C, S	B	5
>	greater than	I, R, C, S	B	5
<=	less or equal	I, R, C, S	B	5
>=	greater or equal	I, R, C, S	B	5
logic:				
NOT, ¬	negation	B	B	1
AND	conjunction	B	B	6
OR	disjunction	B	B	6
XOR	exclusive or	B	B	6
=>	implication	B	B	6
‖	concatenation	C, S	S	4

[a]I = integer, R = real, B = Boolean, C = character, S = string.

Table 3 Operations on enumerations.

relational operations—equal and notequal (=, ¬=, <, >);	
FIRST ('enumeration')	: 'eename' << enumeration_element_name >>;
LAST ('enumeration')	: 'eename';
ORD ('eename')	: INTEGER << ORD(first_element) = 0 >>;
SUCC ('eename')	: 'eename';
PRED ('eename')	: 'eename'.

Simple types The basic data types, *integer*, *real*, *boolean*, *character*, and *string*, require the operations shown in Table 2. Further simple types are *subranges*, *enumerations*, *sets*, and *semaphores*.

Subranges may be used to restrict the elements of a previously defined type to those within a given range (including the boundaries):

```
subrange := RANGE constant. .constant
```

Operations on subranges are inherited from the base type. For example,

```
TYPE index : RANGE −5..+5. << index type for the array xyz >>
```

When the array xyz is defined, all operations on its indices may be applied to index.

Enumerations are defined by giving names to all the elements of the enumeration type:

```
enumeration := VALUES [constant [,constant]^0]
```

The enumeration defines an order for the enumerated elements. Operations on enumerations are summarized in Table 3. For example, the definition

```
TYPE color : VALUES red, green, blue, yellow.
```

implies that

```
FIRST(color) = red
ORD(BLUE)  = 2 <<since the first element is defined as 0>>
PRED(green) = red.
```

The values of SUCC(yellow) and PRED(red) are undefined because there are no given elements after "yellow" or before "red."

Table 4 Operations on sets.

'set' + 'set'	: 'set'	≪ union ≫
'set' * 'set'	: 'set'	≪ intersection ≫
'set' – 'set'	: 'set'	≪ set difference ≫
'element' IN 'set'	: BOOLEAN	≪ set membership ≫
'set' <= 'set'	: BOOLEAN	≪ first set subset of second set ≫
'set' >= 'set'	: BOOLEAN	≪ second set subset of first set ≫
CARD ('set')	: INTEGER	≪ number of elements ≫

Given a data type *t*, SET builds the powerset of the elements of *t*:

```
set := SET [OF type]
```

The operations on sets are summarized in Table 4. For example,

```
TYPE stopchar: SET OF CHARACTER . ≪ word terminators ≫
abc , xyz : stopchar .
xyz := ('.', ',', ';').
abc := ().            ≪ empty set ≫
```

Sets have many properties which correspond to properties of programming systems, e.g., the uniqueness of the elements of a set. This is one of the reasons—together with well-defined constructs—that sets can be very useful in a model-oriented specification approach.

Semaphores can be used to control the synchronization of modules:

```
semaphore := SEMAPHORE
```

The only operations on a semaphore are *wait* and *signal*. For example,

```
TYPE buffer_control : SEMAPHORE
                          ≪ controls buffer access≫
```

Semaphores were included in SLAN-4 because they are very powerful yet basic synchronization data types. If a SLAN-4 user wishes other synchronization constructs, e.g., monitors, he can construct them using semaphores and semaphore operations.

Structured types Structured types use previously defined types for the definition of the base set of a new type. For every structured type there is an operation for the decomposition of an element of that structured type. Structured types may be nested to any depth, and comprise *arrays*, *lists*, and *records*.

Arrays specify a sequence of fixed length of elements; the sequence is indexed by all the elements of the index type:

```
array := ARRAY [(indextype [,indextype]^0)
                              [OF type]
indextype := constant. .constant | type
```

For example,

```
TYPE table : ARRAY ( 1. .100,5. .50) OF INTEGER.
```

Table 5 Operations on arrays.

DOM ('array')	: 'set'	≪ DOMAIN ≫
RNG ('array')	: 'set'	≪ RANGE ≫
RESTRICT('array','set')	: 'array'	
OVERWRITE('array','array')	: 'array'	

If tab1 is an object of type "table," then tab1(15, 27) is an element of type "integer," whereas tab1(37) is an array of type ARRAY(5. .50) of "integer."

Operations which can be performed on arrays are shown in Table 5. These operations extend arrays to "mappings." The operation DOM defines the indices of an array, where the array has been initialized; RNG is the set of the values of all initialized array elements; RESTRICT forgets about the initialization of all elements whose index is not a member of a given set; and OVERWRITE(a, b)(i) becomes equal to b(i) wherever this is initialized, and remains a(i) elsewhere.

Lists specify a variable-length sequence of elements. Lists do not have a defined index type, but indexing a list with positive integers is an operation which is easy to define in terms of the given primitive operations:

```
list := LIST [OF type]
```

Operations on lists are given in Table 6. For example,

```
TYPE string1 : LIST OF CHARACTERS.
abc , xyz : string1
xyz := <|'a', 'b', 'c'|>
abc := <||>    ≪ empty list ≫
```

Records are used to group together several elements of (possibly) distinct types to form one new type:

```
record   := RECORD
              [ - object-name : type]^0
              [CASE name : type OF
                 [constant [,constant]^0 :
                    (   [ - object-name : type ]^1 ) ]^1 ]
            ENDRECORD
            [ IS record ]
```

For example, for

Table 6 Operations on lists.

HEAD('list')	: element_type	≪ first element ≫
TAIL('list')	: 'list'	≪ all but the first ≫
LENGTH('list')	: INTEGER	≪ number of elements ≫
'list' ‖ 'list'	: 'list'	≪ concatenation ≫
ELEMS('list')	: 'set'	≪ a set of all the elements in the list ≫

```
TYPE person : RECORD
                - name : STRING
                - age  : INTEGER
                CASE sex : VALUES(male,female) OF
                    male : ( - beard : BOOLEAN
                             - weight : REAL )
                    female : ( - maiden_name : STRING )
              ENDRECORD
```

if "p" is an object of *type* "person," then "p.name" and "p.maiden_name" are objects of type *string*. The value of "p.maiden_name" is undefined if "p.sex" is "male."

Class types A *class* gives an explicit definition of both the elements of the introduced type and of the operations belonging to this type. Class definitions may be parameterized; in a type declaration there must be an actual parameter for each formal parameter:

```
class-type := class-name parameterlist

parameterlist := [( act-param [,act-param]^0 )] |
                 [( parameter-name: act-param
                    [,parameter-name: act-param]^0 )]

act-param := object-name | constant | type-name | module-name
```

Thus, if c is an object of type class and x is an object (a module, inner class) defined in this class which appears in an export clause, then the (inner) object (module, ...) can either be denoted by "c.class-name.x" or by "c.x". For example,

```
TYPE bounded_stack : stack(max_depth : 100).
```

- *Object declarations*

Each data object used, together with its attributes, must be described in an *object* declaration. The attributes can be given very loosely, or to any degree of detail wanted. The default scope of an object name is the module or class in which it is defined. The initialization of an object may be specified within the declaration.

An *object* declaration has the form

```
object-declaration:=name   [,name]^0 : type
                           [INITIAL constant] .
```

For example,

```
escape1,escape2 : CHARACTER INITIAL '↓'.
```

Expressions

Expressions describe the computation of values starting with given objects. The syntax given here does not distinguish between set expressions, list expressions, integer expressions, etc. Where such a distinction is needed, it will be indicated in the accompanying text:

```
expression:=  IF expression : expression
              [ | expression : expression]^0
              [ ELSE expression ] ENDIF |
              ALL | EX var-name IN expression : expression |
              LET var-name = expression : expression |
              simple_expression | 'empty'
```

The syntactic form for expressions follows the standard which was set by Algol-60. The set of predefined operators is only slightly incremented. The need for complex predicates in pre- and post-conditions has led to the introduction of local declarations in combination with a single assignment within expressions (a well-known concept in functional languages) and quantified expressions using a notation similar to the mathematical "for all x elements of S holds . . . ," or "there exists an x element of S such that . . . ." The quantifiers ALL and EX represent the mathematical quantifiers "for all x in 'set' : . . ." and "there exists at least one x in 'set' such that . . . ." The first expression in such a construct must therefore be of type "set." For example,

```
EX x IN (|1..5|) : x**2 = 16
```

The conditional expression "IF expr. : . . . ENDIF" is given in the same syntax as the IF statement used within the pseudocode (see later section on pseudocode). This allows the designer to enumerate a group of "condition-resulting expression" pairs easily:

```
≪ assume as declaration :
   input_type : VALUES numeric,identifier,string,delimiter. ≫
   input_type := IF input IN (|'0'..'9'|) : numeric
                 | input IN (|'a'..'z'|)  : identifier
                 | input = ' '            : string
                 ELSE delimiter
                 ENDIF.
```

With the LET construct, local variables can be defined in expressions. The value of the local variable is computed *once* upon entering the expression. For example,

```
LET x = f(g(5),y) : h(x,x**5)
```

Additional syntax rules and examples of expressions can be found in Appendix 2, where the use of selectors for arrays and for records is covered and expressions are discussed which relate very closely to programming languages, e.g., Boolean, integer, and real number expressions.

Interfaces and scope rules

The region of a specification (program) where the definition of a name is known is called the *scope* of the name. Names may only be referenced within their scope. If a name is used within a module/class (with the exception of the class specification), it must be either declared or imported by the module/class. This informal definition is precise because the default scope of a name defined in a declaration consists of 1) the complete declaration, 2) the accompanying interface declaration of the class/module, 3) the interface declarations of inner classes/modules as long as no redefinition of the name makes the former definition inaccessible, 4) the accompanying module specification, if any, and 5) the accompanying statements. Here, accompanying means "being defined in the same specification part at the same level." A name may be defined only once within a class/module; it may be redefined within an inner class/module.

The scope rules chosen for SLAN-4 are similar to the scope rules of Euclid [10]; i.e., modules and classes define a *closed* scope. The difference between an open and a closed scope can be defined as follows: An identifier is accessible in an *open* scope if it is declared in that scope or accessible in the enclosing scope. An identifier is accessible in a closed scope if it is declared in that scope or accessible in some enclosing scope *and* explicitly imported via an IMPORT clause.

A closed scope has the advantage, compared to the commonly used open scope, that it is easier for the user of the module/class to determine which objects may be used within the (module) specification, declaration, and pseudocode part. In that way, the effects of changes of declarations can be more easily localized. The danger of introducing complexity through long transitive import lists [4] has been avoided by allowing the import of a variable from any enclosing scope, instead of only the directly enclosing one. A name which is used in n scopes must still be "declared" $n + 1$ times: once within a declaration part and n times within interface descriptions. However, the overhead seems to be smaller for a specification language than for a programming language. Within a specification, one usually gives one name to a complex data structure instead of naming all the lowercase elements of the representation of that structure. Thus, one has to import only the single name instead of a long list of names.

The default scope of a *formal parameter* is the scope of a name declared in the accompanying declaration of the class/module. For purposes of explicit references the formal parameter name may be used in the parameter list of the call of the class/module (see below). The default scope of a *class* or *module name* is the scope of a name declared in the accompanying declaration (i.e., at the same level) plus the scope of a name declared in the declaration part of the class, i.e., the module; recursive modules therefore do not have to import their own names.

If a class or a module has to exchange values (operations) with another specification, the transfer is described in the *interface* declaration:

```
interface-declaration := INTERFACE
                           [parameter-declaration]°
                           [import-declaration]°
                           [export-declaration]°
                         ENDINTERFACE
```

The *parameter* declaration describes the parameters and the way they are accessed:

```
parameter-declaration := parameter-name [,parameter-name]° :
                         PARAMETER qualification type.

qualification := (READ | WRITE | READ/WRITE | TYPE |
                  OPERATION).
```

Parameter passing for objects corresponds to a "call by reference" as used in high-level programming languages.

There are two possibilities for substituting actual parameters for formal parameters: The correspondence can be established by an explicit reference to the formal parameter name, or it can be established by conforming parameter positions as known from programming languages. One cannot mix the two forms. In both cases there has to be an actual parameter for each formal parameter. Lists of actual parameters come with parameterized *classes* when they are used to define objects, or with parameterized *modules* when they are called within the pseudocode.

The possibility of binding actual parameters to formal ones by giving an explicit reference to their names has been included because it can improve the readability of a specification. Binding by corresponding positions has been included as a possibility for situations where the correspondence is obvious, e.g., because there is only one parameter.

Any externally defined object (type, class, module) which is used within a class or module, but defined outside it, must appear in an *import* declaration along with information on how it can be accessed. For example,

```
import-declaration := name [,name]° : IMPORT qualification
type.
```

The (original) scope of the imported name must surround the import declaration; i.e., the name must have been defined on a higher level of the specification. Importing a name extends the scope of the name by the scope it would have if it were defined in the accompanying declaration.

Any object (class, module) that should be known externally, i.e., outside its default scope, must appear in an *export*

declaration. This declaration also indicates the access rights granted:

```
export-declaration :=
    name [,name]^0 : EXPORT [TO name [,name]^0 ] qualification.
```

It is possible to include the name of the specification to which the name is exported; note that the inclusion is treated as an informal comment in the semantics. Exporting a name from a module extends the scope of the name by the scope the name would have if it were defined on the next higher level of the specification, i.e., within the declaration part of the directly surrounding class or module. Exporting a name from a class does not extend the scope but is a prerequisite for referring to the name using the dot notation (compare with class types). It is not necessary to export names to inner classes or modules.

SLAN-4 offers very few rules for type compatibility; this omission is obvious in the case of *parameter*, *export*, and *import* declarations. On the one hand, we did not want to impose restrictions on the user which might be too strict for early system specification. On the other hand, strong typing has been proven to be a property of a language which enforces discipline in the use of data objects to make powerful "specification time" checks possible. That is why the user is responsible for indicating how to perform type conversions in nontrivial cases.

The export clause serves two different purposes. First, it defines which items of a class *c* can be used after the declaration of an object of type *c*. Second, it allows the definition of a variable within a module at a lower level, if we want to describe a situation wherein a variable is only changed by this module, but exists not only during its execution but is to be accessed by other modules or classes as well. In the second case, the lower-level module seems to be the natural owner of the variable. (Note that it may become difficult in this case to find the original definition of the variable if one starts at an *import* statement at the same syntactic level.) An imported name, therefore, may either be directly defined in the enclosing scope or may be indirectly defined by exporting it from a module at the same level as the import clause. We decided against offering two keywords for distinguishing between the two situations (e.g., "import" to refer to a corresponding *export*; "use" to refer to a declaration in the enclosing scope) because the design of the module may find it impossible to decide whether the name should be "imported" or "used."

The class specification

Algebraic specifications may be used to specify abstract data types. Their main advantage is that the specification is completely independent of any representation of the elements of the specified data type. This is achieved by giving information only about the relations which hold between the different operations defined on the elements of the data type. Abstract data types may be described by choosing a convenient representation of the data objects and by defining the operations by the effects the operations have on the chosen representation. It then becomes very difficult to distinguish between the properties of the representation and the properties of the class to be defined. This impairs the free choice of an implementation appropriate for the ideas behind the specification. The same holds for axiomatic specifications: While these abstract from any algorithms for computing the results of operations, they nevertheless depend on a representation of the input and output states for the operations. Therefore, the algebraic method appears to be well suited to the specification of systems in a representation-independent way.

On the other hand, sole use of the algebraic method for specifications presents several problems. First, if the representation of the data objects is fixed (e.g., because one wants to specify an operation "sort array"), it may be more convenient to use this representation instead of abstracting from it. Second, algebraic descriptions are given in an applicative language. Thinking in terms of applicative constructs is not very popular in many programming environments. Third, since we are dealing with second order (or predicate) logic, it is impossible, in the general case, to decide whether a specification is consistent (i.e., whether the relations given do not imply that TRUE = FALSE) or whether it is complete with respect to a base type. That is, we cannot decide in the general case whether an equivalent element of the base type exists for each operation. For some specifications, it is even necessary to introduce new operations to be able to specify the behavior of the wanted operations completely (see for instance [11]).

Therefore, SLAN-4 encourages the combination of both algebraic and axiomatic specifications. Whereas the axiomatic specifications describe the effects of an operation with regard to its input and output states, relations which hold between operations can be comfortably described using algebraic specifications. We do not suppose that either method is a complete specification on its own; each describes different aspects of the target system. It is, therefore, possible for one operation to have different types in the axiomatic and algebraic specifications; normally the types in the axiomatic part are a refinement of the types in the algebraic specification. Every operation used within a class specification therefore must be defined in the definition part. Correspondence to objects existing outside of the class specification is given only by name; no automatic check on compatibility can be performed.

The *class* specification describes the interactions of the modules defined within a class. It consists of three parts: 1) the definitions, where the domain and range of operations are specified; 2) the relations, where equalities holding between the operations are established; and 3) a specification concerning sequencing constraints. Note that the class specification is not part of the scope of names declared in the surrounding environment; every operation name to be used in specifying the relation or sequence must be defined in the definition part. Undefined names in the relation part are taken to be variables which must be used consistently within every single relation. Every name used in the sequence part must have been defined earlier. The correspondence to modules declared in the surrounding environment is given only by the equality of the names; operations within the class specification may have other arities (e.g., domain, range) than their counterparts in the environment. That is, the number and types of parameters of a module do not have to be the same within the class specification and the module interface description:

```
class-specification := SPECIFICATION
                         definition
                         [relation]
                         [sequence]
                       ENDSPECIFICATION

definition := DEFINITION
                [module-name : [typeref [,typeref]^0 ] => typeref.]'
              ENDDEFINITION

typeref. := type-name | *
```

The use of "*" instead of a type name is discussed later. The relation part describes (together with the definition of domain and range) an abstract algebraic model of the data type to be implemented:

```
relation  := RELATION
               [element = expression]'
             ENDRELATION,
```

where the variable in the expression is an implicitly defined object whose scope is only the equation in which it appears. Note that the syntax for the definition part allows only one output type to be specified for an operation. This may seem to forbid the specification of operations with side effects, but this is not quite true. If the output type should be a type which consists of several components (e.g., 'state_of_variables' * 'result_of_operation'), the user himself can introduce the pairing functions and projections needed; e.g.,

```
pair : state , integer => pair_type.
pro1 : pair_type => state.
pro2 : pair_type => integer.
op1 : . . . => pair_type.
    . . .
RELATION
    pro1(pair(s,i)) = s
    pro2(pair(s,i)) = i
    op1( . . . ) = pair( . . . , . . . )
```

etc.

The "*" as a type description has been introduced for distinguishing between data objects declared in the declaration part of the class construct and imported objects of the same type. A type name in the list is always a placeholder for an explicit parameter, whereas the "*" refers to the object to which the operation belongs when it is used. As further examples,

```
add : complex , complex => complex
```

is an operation with two parameters of type "complex" producing a value of type "complex";

```
add1 : complex , * => complex
```

is an operation with one parameter of type "complex" and an argument, which is also of type "complex" if the above definition appears within the specification of a class named "complex." If x is an object of type "complex," then x is an implicit parameter for the "x.add1" operation. Thus, "x.add1(y)" adds the value of x to the value of y and produces a value of type "complex." Finally,

```
add2 : complex , * => *
```

is an operation with one parameter and an implicit argument of type "complex"; it changes the value of the implicit argument. For example, "x.add2(y)" sets the value of x to the result of the addition of x and y.

For presenting the relations holding between the operations, SLAN-4 offers an equational notation. This is the most common method for algebraic specifications. More powerful methods, e.g., using conditional equations of the form p_1 and p_2 and . . . and $p_n \Rightarrow t_1 = t_2$, could be used (where p_i is a predicate), but we doubt whether the additional expressive power really adds to the readability of SLAN-4 specifications (see [12]). A possible alternative would be the restriction to operations specified by recursive definitions similar to the primitive recursive functions defined on natural numbers (see [13]). The advantages of such a restriction are easier semantics for the composition of several specifications and the possibility of executing such specifications for testing purposes (see [14]). Further research is necessary in this area.

Restricted- or unrestricted-execution sequences of operations can be specified with the sequence construct. This is of great value in cases where operations must be performed in a specific total ordering (i.e., sequentially) or where operations can happen in a partial ordering (i.e., concurrently). Campbell [15] proposes the use of "open path expressions" which are incorporated into Path Pascal in order to describe the synchronization of operations defined within an encapsulation mechanism, a restricted abstract data type construct. For SLAN-4, we use Campbell's concept of synchronization specification for *classes* and also his syntax of open path expressions:

```
sequence := SEQUENCE
              [PATH seqlist ENDPATH]'
            ENDSEQUENCE

seqlist    := itemlist [,itemlist]°

itemlist   := item [;item]°

item       := [expr :] (seqlist)|| seqlist || module-name.
```

Four types of constraints can be specified: 1) strict sequencing, denoted by the semicolon, ";"; 2) no sequencing, denoted by the comma, ","; 3) resource restriction, denoted by "expression :(. . .)"; and 4) "|. . .|" which denotes resource derestriction. A path can be composed of arbitrary subexpressions consisting of all of these elements. Module names may be repeated within one path. The specified synchronization constraints for each occurrence are evaluated from left to right.

The following example provides a flavor of the underlying ideas.

```
directory: CLASS
          << The INTERFACE declaration goes here >>
        SPECIFICATION
          DEFINITION
                << of arities, domains, and ranges >>
            create   :                  => *.
            insert   : *, entry_type    => *.
            delete   : *, entry_type    => *.
            is_elem  : *, entry_type    => BOOLEAN.
          ENDDEFINITION
          RELATION   << Essentially a set is defined >>
            is_elem(create,entry) = FALSE.
            is_elem(insert(drc,entry),entry1) =
                IF entry = entry1 : TRUE
                  ELSE is_elem(drc,entry1)
                ENDIF.
            delete(create,entry) = create.
            << delete is allowed on empty directories >>
            delete(insert(drc,entry),entry1) =
                IF entry = entry1 :
                        delete(drc,entry1)
                  ELSE
                        insert(delete(drc,entry1),entry)
                ENDIF.
          ENDRELATION
          SEQUENCE
            PATH create; | insert,delete,is_elem |
            ENDPATH
          ENDSEQUENCE
        ENDSPECIFICATION
      ENDCLASS directory
```

In the *definition* construct, the *insert* function is shown to operate on "directory × entry_type" as domain and "directory" as range. Correspondingly, the relation part shows the operation "insert" with two parameters, where the second axiom means that an insert function applied to a directory with an entry will result in bringing that entry into the directory. The sequence specification shows that first of all a "create" operation must take place, followed by an arbitrary number of "insert," "delete," and "list" operations, which may happen to occur concurrently in an unrestricted manner. The specification of the possible concurrent activation of these operations indicates that synchronization must be provided in the final implementation to ensure the integrity of a directory created by an instantiation of this class.

The module specification

In addition to the concept of pre- and post-conditions, SLAN-4 offers a framework for an implicit specification at a functional level. This specification was developed along the lines described in [5] and [16] and the basic Vienna development method (VDM). High-level data types such as sets, lists, mappings, etc. are also part of SLAN-4. In addition, SLAN-4 provides a more programming-style syntax (Pascal-like), together with an explicit strict data interface control. In [5] it is shown how the basic VDM constructs can be used to formally express the semantics of highly complicated software systems. The expression of operational constraints via a *pre-condition* on the variable state domain is proposed. A *post-condition* relates the initial and final variable states of an operation:

```
module-specification  := SPECIFICATION
                           PRE-module-name:   expression
                           [INTERMEDIATE      expression]°
                           POST-module-name:  expression
                           [EXCEPTIONS:       expression]
                         ENDSPECIFICATION
```

The user is responsible for checking that the supplied pre- and post-conditions of a module are compatible with the requirements of the module as given in a preceding class specification. The pre-condition corresponds exactly to [5], whereas the post-conditions are split into three parts: The *post*-condition expresses the non-exceptional semantics; the *intermediate* allows the designer to state explicitly intermediate states of a module (e.g., in order to be able to express a certain sequence of the internal behavior of a module which might be useful if one wants to provide synchronization points of a module); and the *exceptions* denote exceptional cases explicitly.

In the previous section, the class directory was specified using the algebraic specification method. Now the appropriate pre- and post-specification is given for the operation "insert." Again, it is assumed here that an object directory_entry is declared elsewhere before creating an instance of the class directory with these elements. The pre-/post-specification seems to be well suited as a refinement of the *class* specification. The semantic binding is performed by the object names common to both pre- and post-specifications or by the refinement of common objects in both. It should be noted in the example that follows that the value of an object in the final state is distinguished from the initial value by a prime (e.g., old_directory').

```
insert: MODULE          << inserts an entry into a directory >>
        INTERFACE
          entry_type: PARAMETER(TYPE).
          new_entry: PARAMETER (READ) entry_type.
          old_directory: IMPORT (WRITE) directory
                          << imported from class directory >>
          number_dir_entries: PARAMETER (WRITE) INTEGER.
          total_dir_entries: IMPORT (READ)
                                          << max. size >>
          return_code: IMPORT (WRITE)
                                  << indicates result of insert >>
        ENDINTERFACE
        SPECIFICATION
          PRE-insert: TRUE
          POST-insert:
            number_dir_entries < total_dir_entries =>
              number_dir_entries' = number_dir_entries+1
                                  AND
              old_directory' = old_directory + (| new_entry |)
                                  AND
              return_code' = "SUCCESSFULLY_DONE"
          EXCEPTIONS:
            number_dir_entries = total_dir_entries =>
            return_code' = "DIRECTORY_FULL"
                                  AND
            old_directory' = old_directory
        ENDSPECIFICATION
      ENDMODULE insert
```

Pseudocode

Besides offering language constructs for high-level specification methods, SLAN-4 allows the user to express his low-level design in a notation similar to high-level imperative programming languages. This sublanguage of SLAN-4 is called *pseudocode*, and contains control structures for sequential and concurrent processing besides assignment and procedure calls. The pseudocode part has been designed to offer a way of presenting algorithms independently of the language in which the final program is to be written.

For the description of a module decomposition into smaller parts as well as for the description of actions taking place at the instantiation of a class, SLAN-4 offers a notation which is similar to a conventional programming language. All statements are placed in the "body" of a class or a module:

```
stmts   := BODY
             [statement]^1
           ENDBODY
```

Statements represent the actions to be performed. The syntax rule for the general form of a statement specifies that it may be preceded by one or more labels and that it is always ended by a period. For the control of sequence, selection, and iteration, SLAN-4 includes concepts which are well-known from high-level programming languages. Besides the control of serial actions, SLAN-4 offers language elements for concurrent processing, e.g., the EXECUTE CONCURRENTLY, DO CONCURRENTLY statements and the WAIT and SIGNAL operations on semaphores:

```
statement := [label:]^0 [unlabeled-statement].
unlabeled-statement := assign | do | if | loop | for_loop |
                       execute | assert | return | goto
```

Labels must be unique within the body of a module or a class. The empty statement (i.e., [label:]^0.) has no effect on the values of variables; it may be used in any case where a formal statement is required but the user wants to remain informal.

- *ASSIGN*

The *assign* statement changes the value of an object:

```
assign := name := expression
```

For example,

```
newvalue := 17-4.
```

The *assign* statement does not use the equality sign for denoting an action; instead, it uses the composite sign. This is well-known from Algol-60, Pascal, and ®Ada. By now it should not be necessary to justify this distinction. We did not want to offer a multiple or a parallel assignment because of the difficulties concerning the understanding of the semantics of multiple (parallel) assignments in connection with arrays.

- *DO*

The *do* statement combines statements which have to be processed serially or, by specifying the keyword CONCURRENTLY, in parallel. In the latter case, the ENDDO statement acts like a WAIT statement for the termination of the execution of all the statements inside this *do* statement.

```
do   := DO [CONCURRENTLY]
          [statement]^1
        ENDDO
```

For example,

```
DO CONCURRENTLY
  EXECUTE read_buffer1.
  EXECUTE read_buffer2.
ENDDO.
```

The *do* statement has been introduced primarily for the description of concurrent execution of several statements. It can also be used as a "begin-end" bracket for a group of statements; however, most statement constructors accept groups of statements without brackets. That is, the DO/ENDDO can almost always be omitted without changing the semantics of a design.

- *IF*

The *if*-construct offered by SLAN-4 combines the functions of the usual IF and CASE statements known from other languages:

```
if   := IF expression : [statement]^1
        [| expression : [statement]^1 ]
          [ELSE [statement]^1 ]
        ENDIF
```

The conditions are tested serially from top to bottom. As soon as a true condition is found, the associated statement is

executed and the IF statement is left. If no condition is true the ELSE branch will be taken if present. The *if* statement is equivalent to the empty statement if no ELSE branch is given and no condition is true. For example,

```
IF x < a(m) : h := m.
  |x > a(m) : 1 := m.
ELSE h := m.
     1 := m.
ENDIF.
```

- *LOOP*

The *loop* statement provides a unifying concept for "while" and "repeat" loops and avoids redundant tests or duplicate code in situations where the decision to leave the loop is taken in the middle of the loop. We did not introduce a more sophisticated approach (e.g., using multiple exits as in the proposal of Knuth [17]) because we felt that the overhead for writing down "common loops" would become too big. "While" and "repeat" loops can be written by placing the exit test at the beginning or at the end of the loop:

```
loop   := LOOP
            [statement]^0
          EXITIF expression.
            [statement]^0
          ENDLOOP
```

For example,

```
LOOP
  << ask for next input >>
EXITIF input = 'quit'.
  << process input >>
ENDLOOP.
```

- *FOR*

Loops with an iteration counter, which takes values from a finite set, are represented by the *for* construct:

```
for_loop := FOR name IN [REVERSE]
                <<set>>expression DO [statement]^1
            ENDFOR
```

If REVERSE is not specified, values are selected in increasing order; otherwise they are selected in decreasing order. The variable "name" is implicitly defined with a scope comprising only the for_loop:

```
FOR i IN (|1..10|) DO
    a(i) := 0.
ENDFOR.
```

The *for* statement is more general than the usual form; the set of values of the control variable can be a discrete range or an arbitrary set. If the elements of the set are not ordered, the keyword REVERSE should not be used. Note that the control variable is implicitly defined and is local to the loop.

- *EXECUTE*

The *execute* statement starts the sequential or concurrent execution of a module. If a module represents a function procedure (i.e., returns a value), it may be called by using its name in an expression:

```
execute :=
  EXECUTE [CONCURRENTLY] module-name parameterlist
```

For a discussion of parameters see the section on interfaces and scope rules. For example,

```
EXECUTE insert (old_directory: macro_directory,
                new_entry: request_storage).
```

- *ASSERT*

The *assert* statement describes the conditions which have to be met at a certain point in the program flow. If the test fails, an error message is generated:

```
assert := ASSERT expression.
```

For example,

```
ASSERT 1E-20 < eps.
```

- *RETURN*

The *return* statement terminates execution of a module and transfers control back to the caller:

```
return := RETURN expression.
```

- *GOTO*

The *goto* statement transfers control to the point specified by the label. The label must appear within the same body of a module or a class as the *goto* statement:

```
goto := GOTO label
```

One cannot jump into a FOR-loop from the outside.

Conclusion

SLAN-4 offers a framework for data abstraction and formal specification techniques such as

- algebraic and axiomatic specification methods suited for a conceptual design;
- pre-defined data types and pseudocode for the stepwise refinement process of both data and control structures.

The connection between both design steps is given by the interface and data declaration parts of the syntactic structure of modules and classes.

The algebraic and axiomatic specification methods are used for different aspects of a system. While the algebraic approach documents the relations that exist between the components of a system, the axiomatic method describes the behavior of individual components.

SLAN-4 was used in several IBM locations to specify and design more then five software systems adding up to more than 100 000 lines of code. The experience using SLAN-4 as the specification and design language can be summarized as follows:

Table 7 SLAN-4 words with fixed meanings.

ALL	AND	ARRAY	ASSERT
BODY	CASE	CLASS	CONCURRENTLY
DECLARATION	DEFINITION	DO	ELSE
ENDBODY	ENDCLASS	ENDDECLARATION	ENDDEFINITION
ENDDO	ENDFOR	ENDIF	ENDINTERFACE
ENDLOOP	ENDMODULE	ENDPATH	ENDRECORD
ENDRELATION	ENDSEQUENCE	ENDSPECIFICATION	EX
EXCEPTIONS	EXECUTE	EXITIF	EXPORT
FOR	GOTO	IF	IMPORT
IN	INITIAL	INTERFACE	INTERMEDIATE
IS	LET	LIST	LOOP
MOD	MODULE	NOT	OF
OPERATION	OR	PARAMETER	PATH
POST	PRE	RANGE	READ
RECORD	RELATION	RESULT	RETURN
REVERSE	SEMAPHORE	SEQUENCE	SET
SPECIFICATION	TO	TYPE	VALUES
WRITE	XOR		

1. The rigorous use of SLAN-4 requires a thorough reflection of the designer's intentions leading to a concise documentation of it.
2. The precision introduced by pre- and post-conditions and interface descriptions eases inspections and maintenance.
3. The design contains fewer errors and the documentation avoids ambiguities.

SLAN-4 serves as a good basis for an integrated automatic software development environment that supports the programmer in writing, validating, and verifying specifications and documents for large software systems.

Acknowledgments

The efforts of many were involved in this paper; we especially thank B. Commentz-Walter, H.-G. Frischkorn, C. B. Jones, G. Kreissig, B. Schoener, F. Scholz, and the manuscript referees for their valuable criticism and suggestions.

Appendix 1: Lexical guidelines

In addition to the rules given in Backus-Naur form, the following guidelines must be observed in SLAN-4:

1. Every nonblank character which is not a letter, a digit, "'", or "_" is a *delimiter*.
2. The two-character combinations (|, |), <|, |>, **, ||, ¬=, <>, <=, >=, =>, :=, .., <<, and >>, are called *compound symbols*. Compound symbols are also delimiters.
3. A comment is started by "<<" and delimited by ">>".
4. Blanks and comments may appear everywhere within a specification, but not within names, compound symbols, numbers, or character strings delimited by "'" or "/" without changing the semantics of the specification.

Table 8 SLAN-4 words with meanings which are predefined but which may be redefined by the user.

BOOLEAN	CARD	CHARACTER	DOM
ELEMS	FALSE	FIRST	HEAD
INTEGER	LAST	LENGTH	ORD
OVERWRITE	PRED	REAL	RESTRICT
RNG	STRING	SUCC	TAIL
TRUE			

5. Identifiers (including reserved words) must be delimited with a blank, a comment, or another delimiter.
6. Identifiers and strings delimited with "'" must be further delimited by a blank (since identifiers could end with a "'").
7. The end of a line is only significant in combination with a period; in all other situations it is logically equivalent to a blank.

The length of a syntactic name is not restricted. The underscore character "_" can be used to generate meaningful names. Within the definition part of a class specification, the asterisk can be used as a placeholder for a name. Within pre- and post-conditions, names may have the suffix "'".

Table 7 lists words that have a fixed meaning in SLAN-4 (i.e., they may only be used in the contexts indicated in the syntax). Words that have predefined meanings but which may be redefined by the user are listed in Table 8.

The chosen syntactic notation is in some cases ambiguous. For example, one cannot distinguish between "A or (B followed by C)" and "(A or B) followed by C." Normally the

intended meaning can be grasped from the context; the syntax diagrams presented in Appendix 3 serve as an unambiguous reference.

The requirement that it should be possible to include SLAN-4 specification as a comment within the code (i.e., programs written in high-level or assembler language) has as consequences two syntactic peculiarities. First, the syntactic form of comments in SLAN-4 must be compatible with comments in the code language. For example, a SLAN-4 comment must *not* be used to terminate a comment in the code language. Because of this, delimiters were chosen that had not been used elsewhere. Second, we had to exclude the use of the semicolon for a statement or declaration terminator, because compilers for high-level languages often produce a warning when they find a semicolon within comments, assuming that an end_comment is missing. The only other character which seems to be acceptable as a terminator is ".", the dot or period.

Appendix 2: Expressions

A series of examples of various types of expressions follow.

```
1. simple_expression := term [AND | OR | XOR | => term]^0
```

AND, ..., and => are Boolean operators and therefore expect Boolean factors. For example,

```
p <= y AND y < z
input_empty => error_message_generated
```

```
2. term := simple_term [IN | = | ¬= | <> | <|
                        <= |> | >= simple_term ]
```

IN is the test for set membership; the possible operands for the relational operators <, <= ... are the basic data types (except BOOLEAN) and enumerations. The equality operators may be applied to all expressions. For example,

```
char IN (|'a' .. 'z'|)
X * Y + 1 = 12
```

```
3. simple_term := factor [+ | - | || factor]^0
```

Here, "+" and "−" are defined on INTEGER and REAL operands; || denotes the concatenation of strings. For example,

```
'fish' || ' and chips' << yields 'fish and chips' >>
```

```
4. factor := simple_factor [* | / | MOD simple_factor]^0
```

Here, "*" and "/" are defined on INTEGER and REAL operands; MOD takes two INTEGER operands and returns the remainder. For example,

```
i / (1 - i)
```

```
5. simple_factor := component [** component ]
```

Here, ** denotes exponentiation and accepts two operands of type INTEGER or REAL. For example,

```
sin(x) ** 1.5 .
```

```
6. component := [+ | - | NOT | ¬ ] element
```

The unary operators + and − are defined on INTEGER and REAL; NOT or ¬ denotes the complement on BOOLEAN values; e.g.,

```
NOT ( a**b = c ).
```

```
7. element := 'empty' | constant | variable | (expression) |
              module-name parameterlist |
              (| [expression | range [, expression | range]^0 ] |) |
              <| [expression | range [, expression | range]^0 ] |>
```

```
8. range := expression .. expression
```

The "(| ... |)" is used to denote sets, while "<| ... |>" yields a list. For example,

```
sqrt (x+y+z)
(| 1, 5, 10..19, 23 |)
<| <|1,2|>, <|1..3|> |> << a list of lists >>
<| |>                    << the empty list >>
```

```
9. variable :=
     var_name [(expression [, expression]^0) | . objectname ]^0
```

Array variables may be indexed by expressions to indicate the selection of a specific element. Object names may be applied to record names as selectors and module names may be used as selectors for class names.

```
X.FIRST(7) .
```

```
10. constant := int_const | real_const | char_const |
                string_const | constant-name
```

```
11. int_const := [digit ]^1
```

```
12. real_const := int_const [ .[digit]^1 ]
                              [E [+ | - ] int_const ]
```

```
13. char_const := 'character' | "character"
```

```
14. string_const := ' [character]^0 ' | " [character]^0 "
```

The "E" in a real constant (exponent) must be followed by a blank, a plus, or a minus sign:

```
12.3 E 7,
```

To represent the character "'" within a string delimited by "'" it must be written twice:

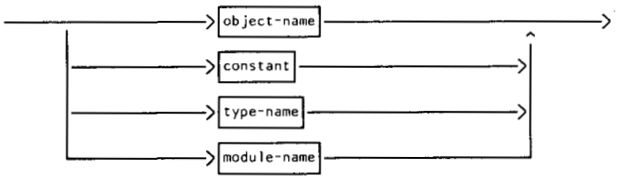
```
'THEO'S PIPE'.
```

Constant expressions are defined in terms of their possible constituents. A constant expression is an expression in which names of variables or modules do not appear. In addition, TYPE conversions between INTEGER and REAL, as well as between CHARACTER and STRING, are performed implicitly where needed and as late as possible.

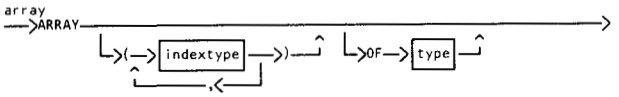
Appendix 3: Syntax diagrams

```
character := any representable letter
digit := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
label := alphameric string starting with a letter
(any-) name := string of letters, digits,
               or '_' starting with a letter.
```

act-param

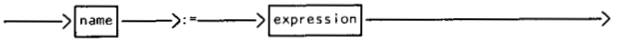


array

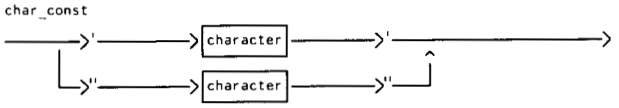


assert

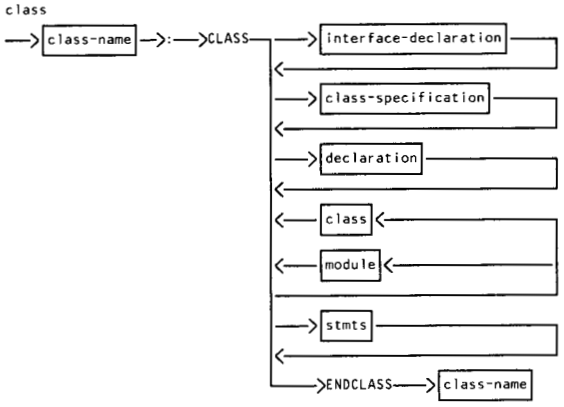
assign



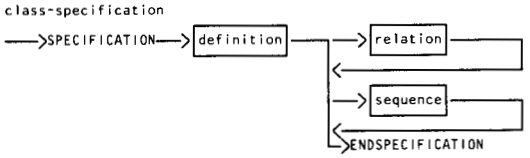
char_const



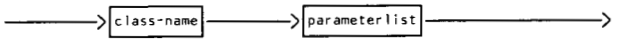
class



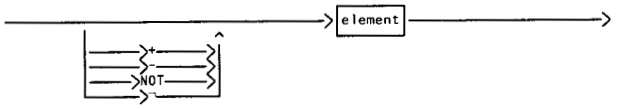
class-specification



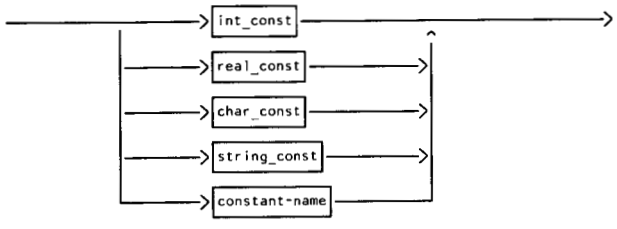
class-type



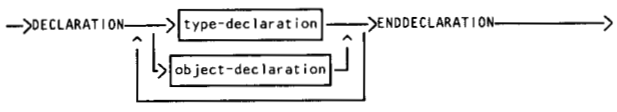
component



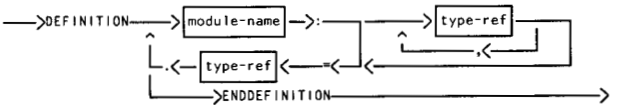
constant



declaration

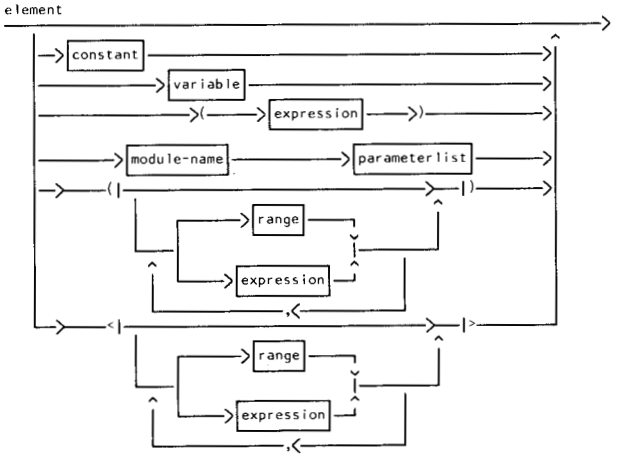


definition

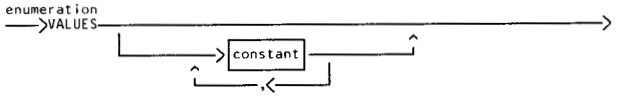


do

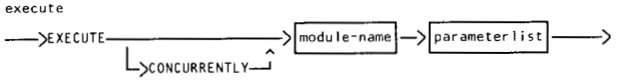
element



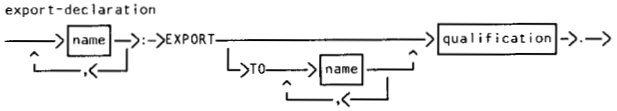
enumeration



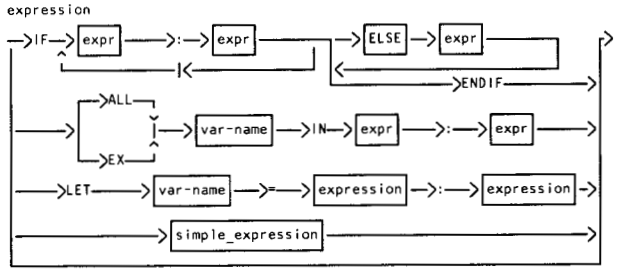
execute



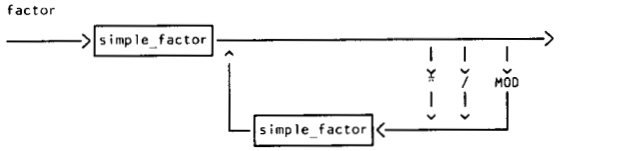
export-declaration



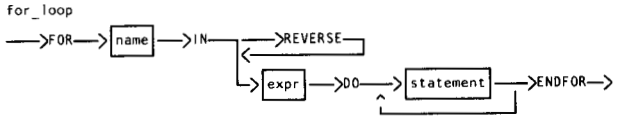
expression



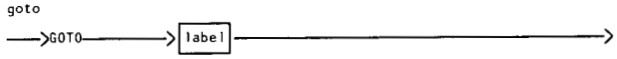
factor



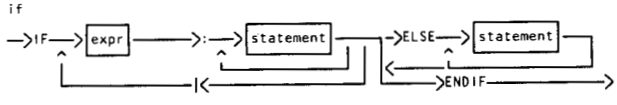
for_loop



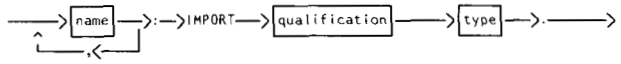
goto



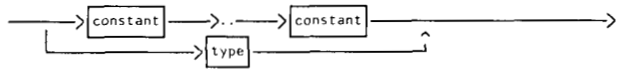
if



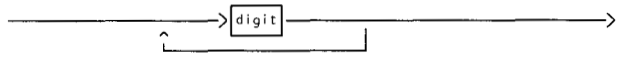
import-declaration



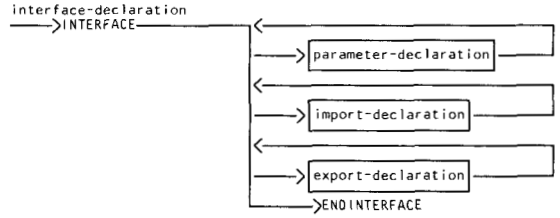
indextype



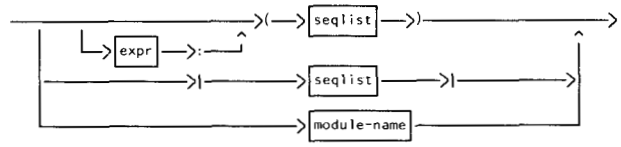
int_const



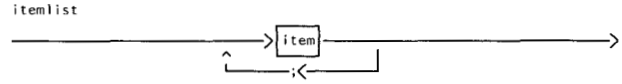
interface-declaration



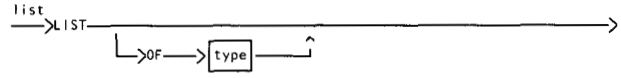
item



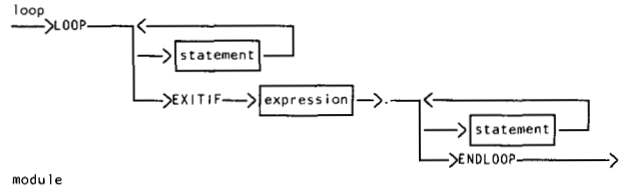
itemlist



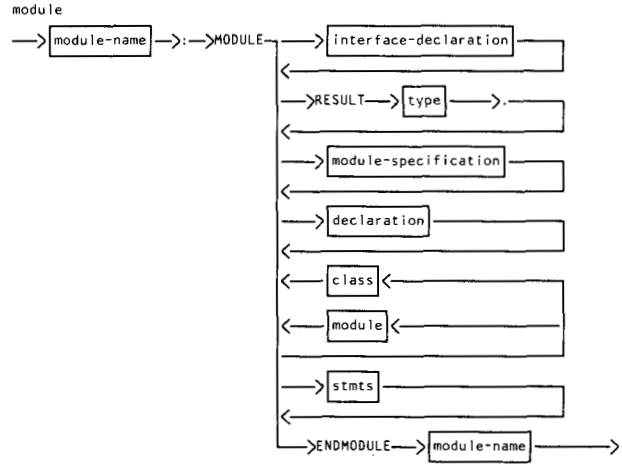
list



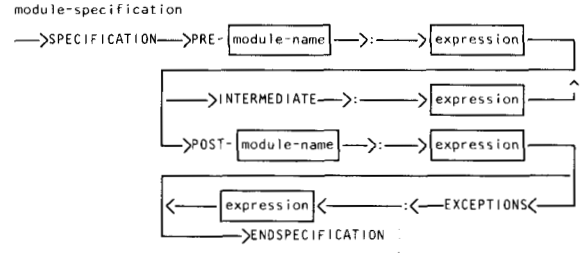
loop



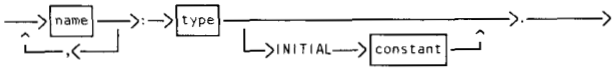
module



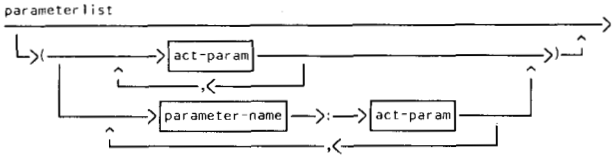
module-specification



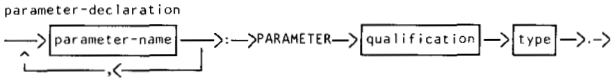
object-declaration



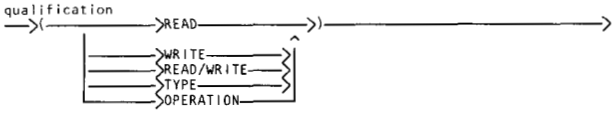
parameterlist



parameter-declaration

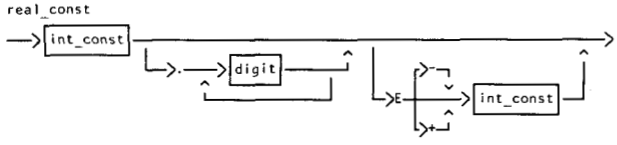


qualification

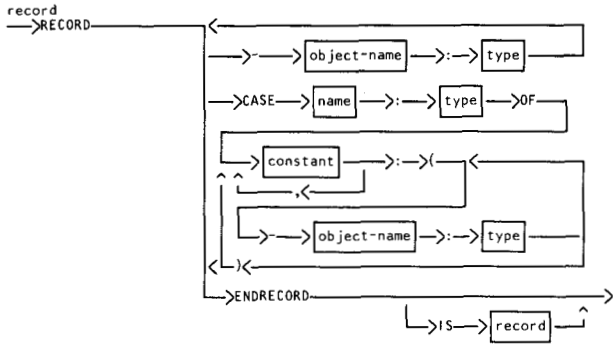


range

real_const



record

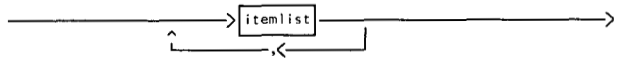


relation

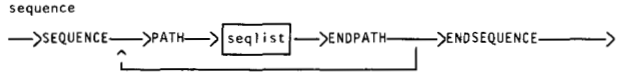
return

semaphore

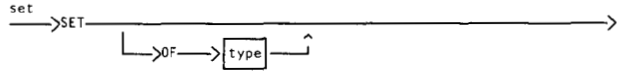
seqlist



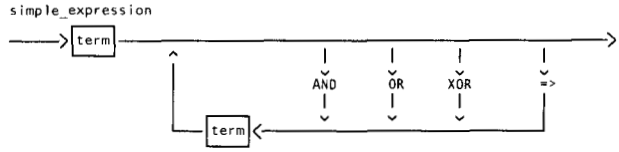
sequence



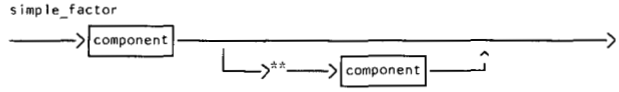
set

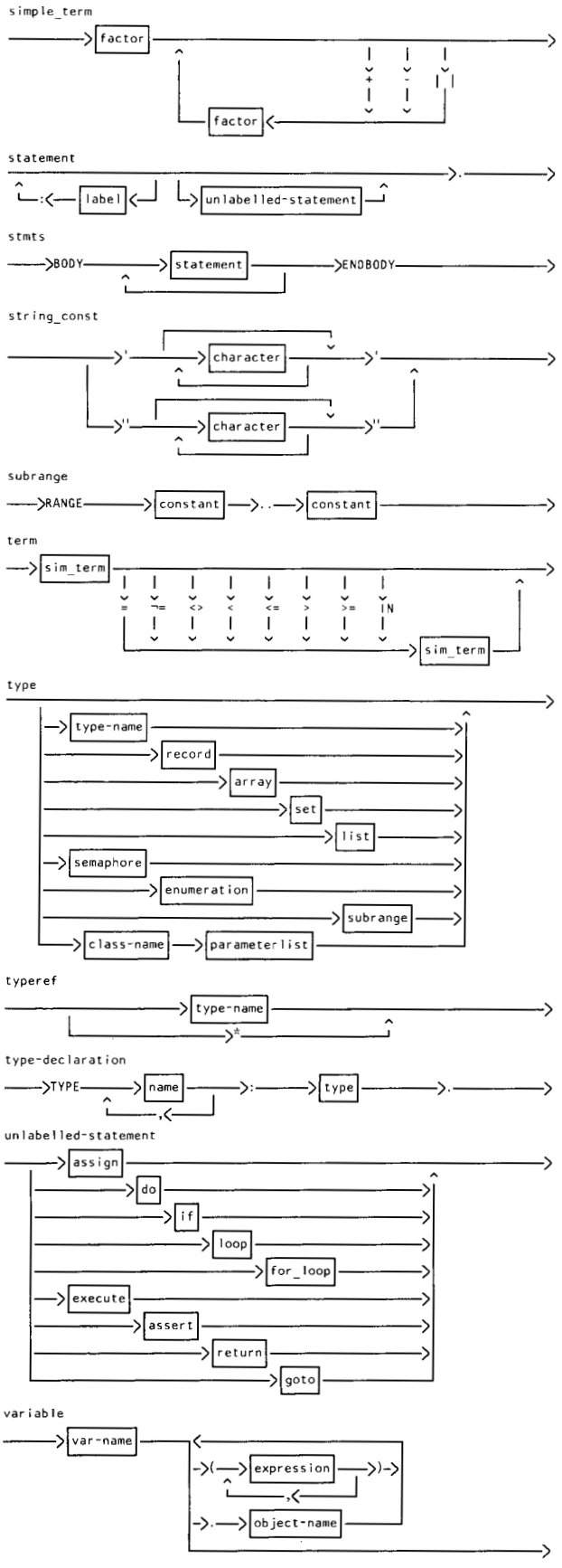


simple_expression



simple_factor





References and notes

1. F. Beichter, O. Buchegger, N. E. Fuchs, and O. Herzog, "SLAN-4: A Software Specification and Design Language," *Technical Report GTR-05.235*, IBM Laboratory, Boeblingen, Germany, December 1979; see also *Software Engineering—Entwurf und Spezifikation*, C. Floyd and H. Kopetz, Eds., Teubner Verlag, Munich, 1981, pp. 91–108.
2. O. J. Dahl, B. Myhrhaug, and U. Nygaard, "SIMULA 67, Common Base Language," *Publication S-22*, Norwegian Computing Center, Oslo, Norway, 1967.
3. J. V. Guttag, "Abstract Data Types and the Development of Data Structures," *Commun. ACM* **20,** 396–404 (1977).
4. J. D. Ichbiah, J. G. P. Barnes, J. C. Heliard, B. Krieg-Brueckner, O. Roubine, and B. A. Wichmann, "Rationale for the Design of the ®Ada Programming Language," *ACM Sigplan Notices* **14,** 6, Part B (1979).
5. C. B. Jones, *Software Development: A Rigorous Approach*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1980.
6. O. Herzog, "Static Analysis of Concurrent Processes for Dynamic Properties Using Petri Nets," *Proceedings of the International Symposium on Semantics of Concurrent Computation, Lecture Notes in Computer Science* **70,** Springer-Verlag, New York, 1979, pp. 60–90.
7. W. A. Wulf, R. L. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs," *IEEE Trans. Software Eng.* **2,** 253–265 (1976).
8. B. Liskov, A. Snyder, and R. Atkinson, "Abstraction Mechanisms in CLU," *Commun. ACM* **20,** 564–576 (1977).
9. J. D. Ichbiah, J. G. P. Barnes, J. C. Heliard, B. Krieg-Brueckner, O. Roubine, and B. A. Wichmann, "Preliminary ®Ada Reference Manual," *ACM Sigplan Notices* **14,** 6, Part A (1979). Note: Ada is a registered trademark of the U.S. Department of Defense.
10. B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek, "Report on the Programming Language Euclid," *ACM Sigplan Notices* **12,** 2 (1977).
11. J. Bergstra and J. V. Tucker, "Equational Specifications for Computable Data Types: Six Hidden Functions Suffice and Other Sufficiency Bounds," *Research Report IW 128*, Computer Science Department, Mathematical Centre, Amsterdam, 1980.
12. J. W. Thatcher, E. G. Wagner, and J. B. Wright, "Data Type Specifications: Parameterization and the Power of Specification Techniques," *Proceedings of the 10th Annual Symposium on the Theory of Computing, ACM SIGACT*, 119–132 (1978).
13. H. A. Klaeren, "A Simple Class of Algorithmic Specifications for Abstract Software Modules," *9th Mathematical Foundations of Computer Science, Lecture Notes in Computer Science* **88,** Springer-Verlag, New York, 1980.
14. H. Petzsch, "INTAS—Ein System zur Interpretation Algebraischer Spezifikationen," *Berichte des Lehrstuhls fuer Informatik* **II,** No. 5, Rheinisch-Westfaelische Technische Hochschule, Aachen, Germany, 1981.
15. R. H. Campbell and R. E. Kolstadt, "Path Expressions in PASCAL," *Proceedings of the 4th International Conference on Software Engineering*, IEEE, Munich, Germany, 1979, pp. 212–219.
16. "The Vienna Development Method: The Meta-Language," *Lecture Notes in Computer Science*, D. Bjorner and C. B. Jones, Eds., Springer-Verlag, Berlin, Vol. 61, 1978.
17. D. E. Knuth, "Structured Programming with Go To Statements," *ACM Computing Surv.* **6,** 261–301 (1974).

Received July 29, 1982; revised July 11, 1983

Friedrich W. Beichter *IBM System Products Division, 7030 Boeblingen 1, Federal Republic of Germany.* Mr. Beichter is a senior associate programmer in VSE operating system development.

After joining IBM in 1977 in Boeblingen, he worked on job control and program librarian programs; he is currently responsible for design, development, and testing of operating system functions. Mr. Beichter received a diploma (M.S.) in computer science from the University of Stuttgart, Germany, in 1977.

Otthein Herzog *IBM System Products Division, 7030 Boeblingen 1, Federal Republic of Germany.* Dr. Herzog joined IBM in 1977 and is currently manager of a group responsible for the quality assurance of program products developed in the System Products Division Laboratory in Boeblingen. He received a diploma (M.S.) in mathematics from the University of Bonn, Germany, in 1972 and the Ph.D. in computer science from the University of Dortmund, Germany, in 1976. From 1972 to 1976 he was a research and teaching assistant in the Computer Science Department of the Dortmund University. Prior to his present assignment, he worked on different projects in systems programming, mainly in the DOS/VSE area. Dr. Herzog is a member of the ACM, the Gesellschaft fuer Informatik (GI), and several ACM and GI SIGs. He is also one of the speakers of the GI SIG on Petri nets and related system models.

Heiko Petzsch *Technical University of Aachen, Federal Republic of Germany.* Mr. Petzsch is working as a research assistant on methods for the specification and development of software. He recently spent a year in the System Products Division Laboratory at Boeblingen implementing a computer support system for the language SLAN-4. He received a diploma (M.S.) in computer science in 1979 from the Technical University of Aachen. Mr. Petzsch is a member of the Gesellschaft fuer Informatik.