This paper presents a formal method for describing programming languages independently of machine architectures and compiler implementations. The method, which was developed to describe PL/1, *is being applied to other programming languages and to compilers and operating systems.*

The definitional techniques are demonstrated using a simple programming language (SPL). *The paper has been written so that little knowledge of mathematics or formal logic is required.*

The formal description of programming languages

by E. J. Neuhold

The description of programming languages involves both the definition of the functions that can be expressed in the language (its semantics) and the notational rules governing the format to be used for requesting the functions (its syntax). A number of formal description methods exist for representing the syntax of programming languages,[1,2,6] but natural language (such as English) is still generally used to specify their semantics. We describe one technique that allows the complete formal description (syntax and semantics) of programming languages. The principal features of this definitional method were developed for the formal description of PL/1.[3-9] The publications of J. McCarthy,[10,11] P. J. Landin,[12,13] and C. C. Elgot[14] strongly influenced the early work leading to development of this method. So far, it has been used to at least partially describe ALGOL 60,[15] FORTRAN, APL, and BASIC, as well as PL/1. Efforts are also underway to apply the techniques to compilers and to operating systems.

The use of formal description methods allows programming languages to be defined in precise, universally understood terms, independently of machine architecture and compiler implementations. Such methods are also opening the way to further theoretical investigations of programming languages and compilers.[10,11,16,17]

The formal description method discussed in this paper is based on the notion that interpretive execution of a program in fact constitutes a semantical description of that program. In this case, the interpretation is conceptual rather than actual, the programs being in-

terpreted are abstract rather than real, and the interpretation must be seen as applying to the entire language rather than a subset of its statements included in a particular program.

The interpretation is performed by an *abstract interpreter* or *abstract machine*, which is not written in any programming or machine language but is specified in an artificial language based on abstract concepts of computing. The basic interpreter mechanisms are the same for all programming language definitions. However, for each language, different information must be kept during interpretation (values of variables, intermediate results, flow of control information) and different ways must be employed for handling this information (transfers of control, movements of data, procedure calls, arithmetic operations). Thus, formal language definition is concerned primarily with defining both the information to be retained by an abstract interpreter and the instructions and functions to be used by the interpreter in manipulating that information.

By interpreting an abstract form of a program, the interpretation process is not burdened with purely notational considerations, such as spacing requirements, choice of separators and delimiters, and parentheses requirements and priority rules in arithmetic expressions. Abstract programs are defined using the *abstract syntax*[10,11] of the programming language, which is designed to exhibit only those structural aspects of programs that are relevant to their interpretation. Thus the formal description of a programming language must include the specification of a translator that describes the mapping of source programs into their abstract form before interpretation.

Finally, to complete the description, the formal syntax (using Backus-Naur Form) of well-formed source programs must be supplied at some stage in the production of the formal description.

This paper is intended to provide a precise but not excessively formal presentation of the principles used for the complete formal description of programming languages. We develop a formal description of a simple programming language (SPL) as a vehicle for demonstrating the method. SPL has deliberately been kept very simple to avoid burdening the reader with learning a programming language as well as the formal description concepts.

We have attempted to write the paper so that little knowledge of abstract mathematical or logical concepts is needed. However, at least a superficial knowledge of Backus-Naur Form notation is required. Some of the definitions we introduce differ from those used in the references.[3-9] These changes were made partially to simplify the definitional technique itself and partially to simplify explanations; however, they do not fundamentally alter the descriptive method.

We first present a conventional description of SPL. We follow that with a more careful examination of the logical properties of SPL programs translated into their abstract form. The requirements of an abstract interpreter that can provide a formal description of SPL are developed next. Finally, in the concluding remarks, we demonstrate one important use of formal language definition—how to accommodate language extensions. A formal description of the translator is not included. It is of little interest for the definition of semantics, and its properties are implied in the relationship between the abstract form of SPL programs and their source language form.

A simple programming language

The basic components of SPL are *numbers* and *variables*. A number in SPL is always an integer, and a variable is a quantity that is identified by a *name* (an identifier consisting of one or more letters). An SPL variable may take on any of the integer values. SPL also provides the two *arithmetic operators*, + (add) and − (subtract).

The arithmetic operators, together with numbers and variables, are used to construct *expressions*. An expression may be simple (a single number or variable) or it may be any combination of numbers, variables, and operators, as allowed by the rules of mathematics. Using Backus-Naur Form, we are able to specify the syntax of expressions as

⟨expression⟩ : : = ⟨simple expression⟩ |

⟨simple expression⟩ ⟨operator⟩ ⟨expression⟩

⟨simple expression⟩ : : = ⟨number⟩ | ⟨variable⟩ |

(⟨expression⟩)

operator : : = + | −

Examples of well-formed SPL expressions are:

ALPHA
105
(A + X)
(SUM − A) + (B − 5 + D)
A + B − C + D

If an expression contains a sequence of additions and/or subtractions in which the order of evaluation is not specified by parentheses, the operations are performed from left to right. Thus the expression

A + B − C + D

is equivalent to the expression

((A + B) − C) + D

Note that the order in which additions and subtractions are performed is of importance as soon as upper and lower limits are placed on the numbers that can be handled by the computer.

An SPL *program* consists of *set-statements* and *goto-statements.*

A set-statement is used to assign a value to a variable. The syntactic form of a set-statement is given by the rule

⟨set-statement⟩ : : = SET ⟨variable⟩ TO ⟨expression⟩

The value of the expression is taken on by the variable that follows the word SET, and any value previously assigned to the variable is destroyed.

Normally, the statements in an SPL program are executed sequentially. However, the goto-statement can transfer control to some statement other than the next sequential statement. A label in the goto-statement identifies the destination of the transfer of control. Such a label is represented by an identifier consisting of one or more letters. To identify the target statement, the same label must appear exactly once as the prefix to some statement in the SPL program. The general form of the goto-statement is given by the syntax rule

⟨goto-statement⟩ : : = GOTO ⟨label⟩ IF ⟨expression⟩

The transfer of control only takes place if the value of the expression is greater than zero; otherwise, the statement following the goto-statement is executed.

The identifiers SET and TO in the set-statement and the identifiers GOTO and IF in the goto-statement are called keywords of SPL. The words SET and GOTO are also called statement identifiers.

To avoid ambiguities, at least one space (blank character) must be inserted between adjacent identifiers in SPL programs. Between the other basic components, the use of separating spaces is optional.

An SPL program that specifies the summation of the integers 1 through 10 is shown in Table 1.

Table 1 Summation program

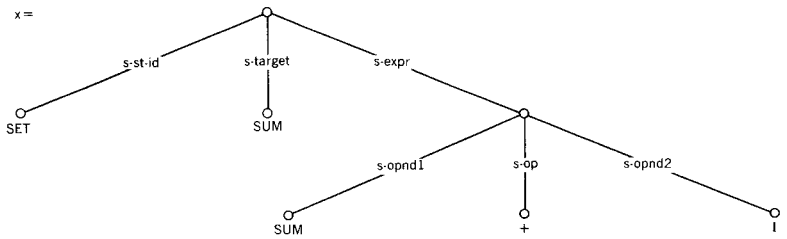
```
      SET SUM TO ZERO
      SET I TO 1
LOOP  SET SUM TO SUM + I
      SET I TO I + 1
      GOTO LOOP IF 11 - I
```

The abstract program

need for translation

When designing an interpreter for a programming language, source programs are usually not interpreted directly; instead a translated form of the program is used as the input to the interpreter. Basically, source programs may be considered as strings of characters. Interpreting these strings requires cumbersome scanning and sometimes complicated procedures for gathering the segments relevant for the interpretation. (By segments, we mean components of statements

Figure 1 Sample tree x



such as statement identifiers, variable names, or expressions.) For example, a set-statement in SPL contains two main parts, the target variable and the expression to be evaluated. However, the expression must be evaluated (and therefore located in the program) before the value replacement for the target variable can take place. For this reason, a translation should produce structures that allow ready access to the program segments needed for interpretation.

As previously indicated, the translation process also eliminates source program elements not required for the semantic interpretation. Usually, the structuring inherent in the translated program allows the elimination of some of the punctuation marks and keywords. For example, the words TO and IF in SPL are not required in the translated form; they are redundant and are used only to make the program more intelligible to human beings. However, the statement identifiers SET and GOTO are important; they distinguish between the two possible statement types in SPL. Also in the translation process, keywords are commonly translated into some internal form to distinguish them from variable and label names.

tree representation

These principles of source program translation have been applied in the design of the interpreters used for the formal description of programming languages. The translated (abstract) programs explicitly show the relevant program structure but do not contain redundant symbols and keywords. The form we selected to represent the abstract program is the *tree*. Names associated with the branches of the trees identify the various segments of the abstract programs. The leaves (terminal nodes) of the trees are formed by the *elementary components*; e.g., translated keywords, identifiers, constants, and operators in the programming language. A degenerated tree consisting of only a single elementary component (a tree without branches) is called an *elementary tree*. An example of a tree, which represents the SPL statement

SET SUM TO SUM + I

in abstract form, is shown in Figure 1. Note that the keyword TO has been eliminated in the abstract form of the set-statement.

Three functionally different segments in the set-statement are identified by the branch names:

s-st-id for the translated statement identifier SET
s-target for the target variable SUM
s-expr for the expression SUM + I

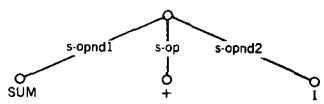
The expression itself forms an abstract tree (a subtree of the set-statement tree). Its parts are identified by:

*s-opnd*1 for the first operand SUM
s-op for the expression operator +
*s-opnd*2 for the second operand I

segment selection

To allow the interpreter ready access to the various parts of an abstract program, the names of the branches are used as *selectors* of tree segments. For example, the application of the selector *s-expr* to the sample tree *x* produces the subtree shown in Figure 2. The application of the selector *s-st-id* to the abstract set-statement produces the elementary tree SET, i.e., a degenerated tree consisting only of the leaf SET.

Figure 2 Subtree of x



Assuming that *x* identifies the abstract set-statement represented in Figure 1, we may use functional notation to represent the application of a selector. For example, *s-expr* (*x*), which is read "*s-expr* applied to *x*," also represents the subtree in Figure 2. Another example is given by

s-st-id(*x*) = SET

The result of the application of a selector is a tree, and another selector may be applied to this result-tree to produce some even smaller part of the total tree. For example, the application of the selector *s-op* to the subtree represented by *s-expr*(*x*) produces the elementary component +. Using the functional notation, we may write

s-op (*s-expr*(*x*)) = +

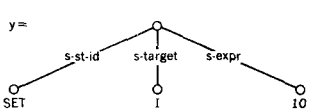
definition of predicates

The named branches in the abstract programs allow, by means of selector application, fast access to the various segments of the programs. However, an interpreter must accept any well-formed program and thus must handle a variety of different tree types. Because different actions are required of the interpreter depending on the type of tree, the interpreter must be able to distinguish among tree types. For example, translation of the SPL statement

SET I TO 10

produces the tree shown in Figure 3. Comparison of the tree in Figure 3 with the tree in Figure 1 shows that the interpreter must use different selectors to arrive at the elementary components of these trees. Thus a test of the tree type is required.

Figure 3 Sample tree y



Each type of tree can be distinguished from all other trees by its membership in a set of trees containing all the trees of that type. For example, all trees representing SPL expressions may form such a

class. All tests for a tree type are then tests for membership in the associated class. Therefore, with each class, a *test function*, termed a *predicate*, is defined. The predicate, when applied to a tree, is satisfied (*true*) if the tree is a member of the associated class; it is not satisfied (*false*) otherwise. We may also say that each predicate defines a class of trees whose members are those trees that satisfy the predicate.

Assuming that the predicate *is-expr* defines all SPL expressions, the following formulas referring to the trees in Figures 1 and 3 produce the result *true*.

is-expr (*s-expr*(x))
is-expr (*s-expr*(y))
is-expr (*s-target*(y))

Note that in the last formula the selector *s-target* produces the elementary tree I, i.e., a variable, and a variable is a well-formed SPL expression. However, the formula

is-expr (*s-st-id*(x))

is not satisfied. The function *s-st-id*(x) returns the translated statement identifier SET, which is not an expression.

To define the predicates used by the abstract machine, we start by introducing predicates that describe the elementary components. These predicates are then used to specify predicates describing more complex tree classes.

We introduce the predicate *is-constant*, which is satisfied only for the integer constants allowed in SPL. The predicate *is-name* is satisfied for all the variables and labels allowed in SPL. The special predicates *is-set*, *is-goto*, *is-add*, and *is-subtract* are satisfied for the elementary objects SET, GOTO, +, and −, respectively.

It may become necessary to specify predicates that define simple combinations of different tree classes. For example, a predicate defining all elementary components allowed as SPL expressions must combine variables and constants, since either of them may appear as a simple expression.

To define such a combination, the logical *or* operator $\vee$ is used. For example, the expression

is-simple-expr = *is-name* $\vee$ *is-constant*

specifies that the predicate *is-simple-expr*(x) is satisfied if either the predicate *is-name*(x) or the predicate *is-constant*(x) is satisfied for the tree x.

Predicates that define nonelementary trees are defined by incorporating the branch names of the trees into the definition. By using

the branch names that leave the root of the tree, all subtrees immediately subordinate to the root can be identified. Assuming that the predicates of these subtrees have already been defined, the new predicate can be formulated by specifying the predicates of the subtrees with the branch names. For example, a predicate *is-set-stmt* defining all abstract set-statements may be formulated as

$$\begin{aligned} \textit{is-set-stmt} = (&\langle \textit{s-st-id} : \textit{is-set}\rangle,\\ &\langle \textit{s-target} : \textit{is-name}\rangle,\\ &\langle \textit{s-expr} : \textit{is-expr}\rangle) \end{aligned}$$

All branches leaving the root of the set-statement tree are specified together with the predicates defining the subtrees that may appear at these branches. The general form of this construct is given by

$$\begin{aligned} \textit{is-pred} = (&\langle \textit{sel}_1 : \textit{is-pred}_1\rangle,\\ &\langle \textit{sel}_2 : \textit{is-pred}_2\rangle,\\ &\quad\vdots\\ &\langle \textit{sel}_n : \textit{is-pred}_n\rangle) \end{aligned}$$

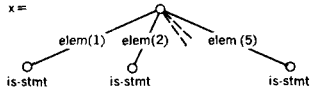
where *is-pred* is the newly defined predicate; sel_1, sel_2, $\cdots$, sel_n are the selectors leaving the roots of the trees described; and $is\text{-}pred_1$, $is\text{-}pred_2$, $\cdots$, $is\text{-}pred_n$ are the predicates defined for the subtrees. The predicate *is-pred* is satisfied only if all selectors sel_1, sel_2, $\cdots$, sel_n appear as branches leaving the root of a tree and if in addition all of the predicates $is\text{-}pred_1$, $is\text{-}pred_2$, $\cdots$, $is\text{-}pred_n$ as applied to the identified subtrees are satisfied. Note the correspondence of this definition to the logical *and* operation.

definition of sequences

To describe in tree form the sequences of SPL statements that constitute programs, a special convention about the branch names identifying elements in a sequence is introduced.

The branch names $elem(1)$, $elem(2)$, $\cdots$, $elem(n)$ are used to identify the first, second, $\cdots$, nth element in the sequence. For example, the SPL program in Table 1 is represented in abstract form as shown in Figure 4. Note that the predicates *is-stmt* are used as indicators for the subtrees that must appear in a well-formed abstract program. To complete the tree, we must insert all of the proper subtrees for statements 1 through 5 of the program.

Figure 4 Part of integer summation tree



Predicates that define sequences are always written as a predicate name followed by the affix-*list* (e.g., *is-stmt-list* for a sequence of statements). The general definition of such predicates is given by

$$\begin{aligned} \textit{is-pred-list} = (&\langle \textit{elem}(1) : \textit{is-pred}\rangle,\\ &\langle \textit{elem}(2) : \textit{is-pred}\rangle,\\ &\quad\vdots\\ &\langle \textit{elem}(n) : \textit{is-pred}\rangle) \end{aligned}$$

Figure 5 Tree representing A + (B — C)

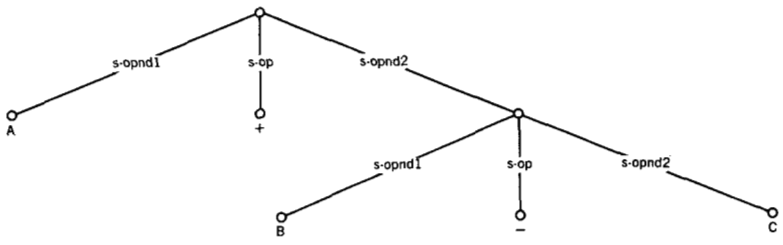
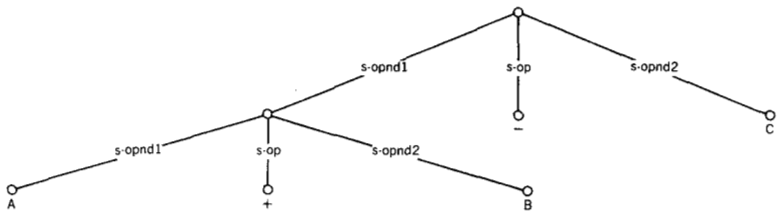


Figure 6 Tree representing A + B — C



The selectors *elem*(1), *elem*(2), $\cdots$, *elem*(*n*) identify the individual elements of the sequence. For example, the *i*th statement in the abstract tree *x* (Figure 4) is selected by using the selector function

elem(*i*) (*x*)

A special function *length* (*x*) is introduced to determine the number of elements contained in the abstract sequence *x*. We shall see that this function is needed to determine when the interpretation of a sequence is completed.

treatment of expressions

Consider now the representation of SPL expressions in abstract form. In SPL, parentheses and the left-to-right rule indicate the order in which the individual operations required to evaluate an expression must be performed. In the abstract program, this order can be reflected in the structure of the tree, which represents the abstract SPL expression. For example, the expression

A + (B — C)

is translated into the tree shown in Figure 5, whereas the expression

A + B — C

is translated according to the left-to-right rule into the tree in Figure 6. Each operator and its associated operands are easily identified in the tree structure using the branch names leaving the individual nodes in the expression tree.

the abstract syntax of SPL

In general, the structure of the trees satisfying a predicate is illustrated in the definition of the predicate. Branch names and predicates

corresponding to the immediate subcomponents of the trees appear in the definition. That is, the predicate definition provides a syntactic description of the associated trees. By using the definition of predicates, an *abstract syntax*, i.e., a syntax of abstract SPL programs, can be formulated.

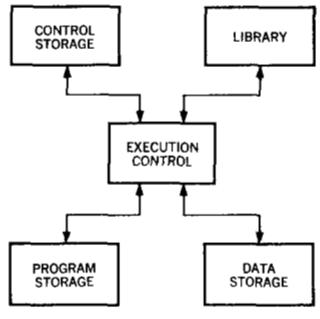
All well-formed abstract SPL programs satisfy the predicate *is-program*. This predicate is defined in terms of subordinate predicates corresponding to the relevant subcomponents of the abstract SPL programs, as follows:

is-program = *is-stmt-list*
is-stmt = *is-lab-stmt* ∨ *is-unlab-stmt*
is-lab-stmt = (⟨*s-label* : *is-name*⟩,
⟨*s-unlab-stmt* : *is-unlab-stmt*⟩)
is-unlab-stmt = *is-set-stmt* ∨ *is-goto-stmt*
is-set-stmt = (⟨*s-st-id* : *is-set*⟩,
⟨*s-target* : *is-name*⟩,
⟨*s-expr* : *is-expr*⟩)
is-goto-stmt = (⟨*s-st-id* : *is-goto*⟩,
⟨*s-label* : *is-name*⟩,
⟨*s-cond-expr* : *is-expr*⟩)
is-expr = *is-name* ∨ *is-constant* ∨ *is-infix-expr*
is-infix-expr = (⟨*s-opnd*1 : *is-expr*⟩,
⟨*s-op* : *is-operator*⟩,
⟨*s-opnd*2 : *is-expr*⟩)
is-operator = *is-add* ∨ *is-subtract*

The abstract interpreter

The abstract interpreters used for defining the semantics of programming languages have many similarities to conventional interpreters. Therefore, the main structural components of such interpreters are illustrated in Figure 7 and reviewed here using conventional terminology. The components of abstract interpreters are then considered in relation to those of conventional interpreters.

Figure 7 Interpreter components



The *execution control* in Figure 7 guides all actions of the conventional interpreter during program interpretation. To specify these actions, a set of *instructions* is provided with the interpreter. These instructions are used to formulate routines that scan and test the elements of the program to be interpreted and that describe all the calculations required for the interpretive execution of the program. The execution control selects from these routines the next instruction to be executed and performs its execution.

The *control storage* contains the instruction sequence to be used for the interpretation. To increase the flexibility of these routines, the instruction sequence is usually not fixed; instead it is created

during an interpretation via expansions of macroinstructions or via direct modifications of the instruction stream.

The *program storage* contains the program to be interpreted in internal format. As discussed in the preceding section, the internal format in general reflects the original source language program.

The *data storage* contains all program data and internal data needed for the interpretation process. To relate variable names encountered in the program with the values kept in the data storage, it is necessary to hold information that identifies the locations of the values, the length of the value items, the data types, etc. These *data descriptions* are also kept in the data storage, and they are accessed via the names of the program variables.

To specify an interpreter and its instruction set, some machine language is usually used. The execution control is represented by a machine language program. Each nonmacroinstruction of the interpreter is defined with a small machine language routine, which is kept in a *library*. The routine is called as soon as an execution of the instruction is required. For macroinstructions, the macro-definitions are kept in the library, and an execution of such an instruction results in a macro-expansion.

In abstract interpreters, a specific machine language is not used. Instead, we use the tree structure and tree manipulation functions as the bases for our interpreter construction.

All storage components of the abstract interpreter are represented in tree form. In addition, all information concerning the status of an interpretation is always kept in these storage components. No information is stored in the execution control of the interpreter. For example, the next instruction to be executed is identified by information in storage and not by a counter in the execution control.

The sum of all stored information determines the *state* (ξ) of the abstract interpreter. The execution of an instruction changes this information and hence the state. Thus, the interpreter instructions define the transitions of the interpreter between its various states.

The type of information included in the state of an abstract interpreter, of course, varies with the programming language described by the interpreter. For example, SPL allows only integer variables. Consequently, no description of the data types of variables is required. Each value of a variable can automatically be interpreted as an integer value. Conversions between different data types need never occur.

The information kept as the state of the SPL interpreter is classified into five different categories:

the SPL interpreter state

1. The *abstract program* to be interpreted
2. The *statement counter*, which identifies the abstract statement currently being interpreted
3. The *value storage*, which contains all the values of the program variables
4. The *control*, which contains the interpreter instruction sequence to be executed
5. The *library*, which contains the definitions of the macroinstructions and of the elementary instructions that form the instruction set of the SPL interpreter

A comparison of these state components with the general structure of an interpreter shown in Figure 7 provides that category 1 represents the program storage; category 2 together with category 3, the data storage; category 4, the control storage; and category 5, the library of the interpreter.

The predicate *is-state* is introduced to describe the state of the SPL interpreter. All well-formed states of the SPL machine satisfy the predicate *is-state*. The predicate *is-state* is defined in terms of subordinate predicates corresponding to the five information categories as follows:

$$\begin{aligned} \textit{is-state} = (&\langle \textit{s-pgm} : \textit{is-program}\rangle,\\ &\langle \textit{s-stc} : \textit{is-constant}\rangle,\\ &\langle \textit{s-vst} : \textit{is-val-stg}\rangle,\\ &\langle \textit{s-c} : \textit{is-control}\rangle,\\ &\langle \textit{s-lib} : \textit{is-library}\rangle) \end{aligned}$$

abstract program and statement counter

The predicate *is-program* describes all well-formed abstract SPL programs, as discussed in the previous section. To select the abstract SPL program represented by the state ξ of the SPL interpreter, the selector function *s-pgm* (ξ) must be used. Each abstract program is defined as a sequence of SPL statements. The ith statement is selected for interpretation by the function $elem(i)(s\text{-}pgm(\xi))$ where i represents an integer value.

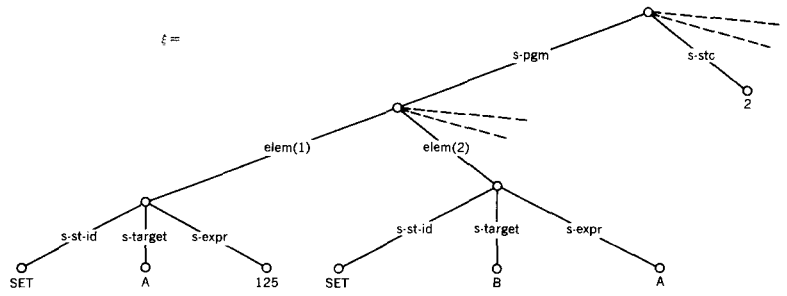
The statement counter is identified by the selector *s-stc* in the state of the SPL interpreter. The value of the statement counter is found by using the function *s-stc* (ξ). This value represents the sequence number of the statement currently being processed by the SPL interpreter. In other words, the current statement can be extracted from the interpreter state by the function $elem(s\text{-}stc(\xi))(s\text{-}pgm(\xi))$.

examples

Assume that the interpreter state has the form shown in Figure 8. Application of the selector functions $elem(1)$ $(s\text{-}pgm(\xi))$ to state ξ results in the subtree shown in Figure 9. Various other selector functions applied to this state produce the following results:

$$s\text{-}target\ (elem(s\text{-}stc(\xi))\ (s\text{-}pgm(\xi))) = \text{B}$$
$$elem(1)\ (\xi) = \Omega$$
$$elem(4)\ (s\text{-}pgm(\xi)) = \Omega$$

Figure 8 Interpreter state 1

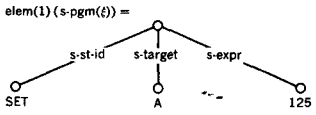


By definition, a selector function applied to a tree always returns the null-tree Ω if the identified selector does not appear as one of the names of the immediate subbranches of the tree.

value storage

The value storage component of the SPL interpreter state contains the values of the program variables encountered in the abstract SPL program. To identify the individual values, the names of the program variables are used as selectors. For example, a value storage may have the form shown in Figure 10. A, B, and SUM are variables with the values 124, 30, and 0, respectively.

Figure 9 Selector function applied to state 1



A special notation derived from the notation used for set definitions allows us to define the predicate *is-val-stg* describing all the well-formed value storage items of the SPL interpreter. For our purposes, it is only necessary to keep in mind that the *names* of variables are used as the selectors to gain access to their values. However, using the notation, which is further explained in Reference 8, the definition of *is-val-stg* appears as

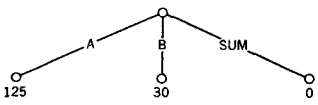
$$is\text{-}val\text{-}stg = (\{\langle name : is\text{-}constant\rangle \mid\mid is\text{-}name(name)\})$$

examples

Assume that the interpreter state ξ has the form shown in Figure 11. The results of various selector functions are as follows:

$$A(s\text{-}vst\ (\xi)) = 125$$
$$s\text{-}target\ (elem(1)\ (s\text{-}pgm(\xi)))\ (s\text{-}vst(\xi)) = 125$$
$$B(s\text{-}vst(\xi)) = \Omega$$

Figure 10 Value storage tree

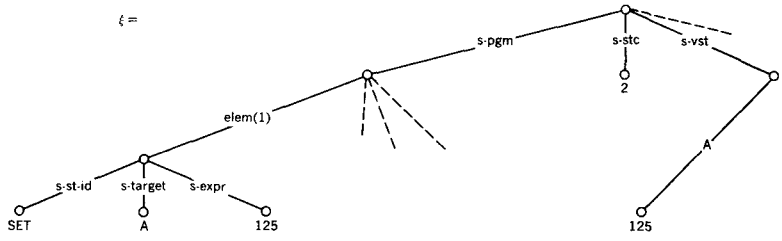


Note that the statement counter indicates that the first statement has already been executed by the interpreter. That is, the value of A in the value storage has been set to 125.

control

The actions of abstract interpreters are controlled by the interpreter instructions. However, arranging the instructions into a predetermined sequence for execution, as is done in conventional programming, does not provide enough flexibility to represent the semantics of programming languages.

Figure 11 Interpreter state 2



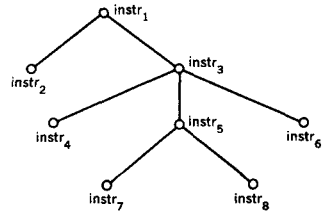
Assume, for example, the well-formed SPL expression

(A − B) + (C − D)

The rules governing parentheses require that the two subexpressions A − B and C − D be processed before the addition is performed. However, no rule of SPL specifies which of the two subtractions has to be performed first. This is often an important feature of programming languages, since it makes possible the optimization of expression evaluation. Depending on the technique chosen for the scanning of expressions, and for optimization, one conventional interpreter may execute A − B before C − D, and another interpreter may execute C − D before A − B. In other words, conventional interpreters introduce additional rules that select specific execution sequences for expression processing. The abstract interpreter is not allowed to add such rules. It must represent the meaning of expressions exactly as specified for the language. The additional rules are only allowed to be added for a specific implementation of the language. The abstract interpreter must describe all possible implementations, not select a specific one.

To allow the execution control of the abstract interpreter the required arbitrariness in the selection of the instruction to be executed, the control component of the interpreter state is represented by a finite tree (the control-tree), where each node of the tree is associated with an instruction, as shown in Figure 12. In addition, the convention is established that only the instructions associated with the leaves of the control tree are candidates for immediate execution. The execution control chooses randomly one of these terminal instructions for the next execution. In the example in Figure 12, the instructions $instr_2$, $instr_4$, $instr_6$, $instr_7$, and $instr_8$ are the candidates for being executed next.

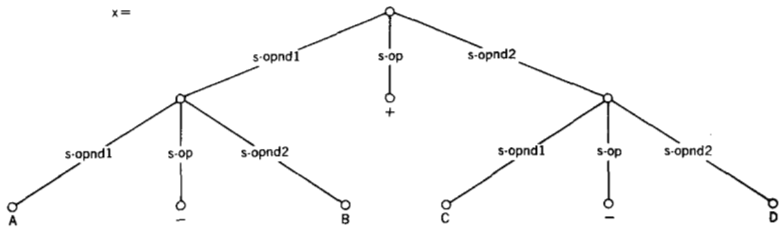
Figure 12 The control-tree



The execution of an instruction results in the deletion of that instruction from the control-tree, so that a different set of instructions is available for the next execution step. The interpretation process ends as soon as the control component becomes empty.

This generalization of the execution process in the abstract interpreter allows all possible execution sequences to be represented in

Figure 13 (A − B) + (C − D)



the control component. Thus the behavior of the interpreter in interpreting a program is defined to include all possible execution sequences, not just one of them. Consequently, all situations that may arise during the interpretation of a program are described by the abstract interpreter.

example

To illustrate this concept, we investigate once more the SPL expression

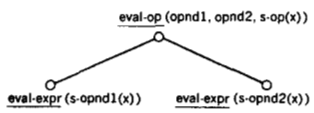
(A − B) + (C − D)

The abstract form of the expression is given in Figure 13. The two subexpressions (A − B) and (C − D) are represented as the subtrees *s-opnd*1(*x*) and *s-opnd*2(*x*).

The rules of SPL require that the subtrees *s-opnd*1(*x*) and *s-opnd*2(*x*) be processed before the addition can take place. To describe the execution, we introduce three instructions, which are described more fully later.

1. *eval-expr*(*t*), where *t* is an expression. Execution of the instruction causes evaluation of the expression.
2. *eval-op*(*opnd*1, *opnd*2, *op*), where *opnd*1 and *opnd*2 are operand values and *op* is one of the two infix operators (add or subtract) of SPL. Depending on the operator specified, execution of the instruction calculates the sum *opnd*1 + *opnd*2 or the difference *opnd*1 − *opnd*2.
3. *get-val*(*var*), where *var* is a program variable. Execution of the instruction extracts the value of the variable *var* from the value storage.

Figure 14 Control tree for expression evaluation



The proper execution sequence for the expression *x* can be specified with the control-tree shown in Figure 14.

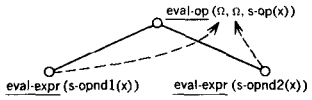
The instructions *eval-expr*(*s-opnd*1(*x*)) and *eval-expr*(*s-opnd*2(*x*)) are both candidates for execution. Either one of them may be chosen by the execution control to be processed first.

return information

The evaluation of the two subexpressions of *x* produces two intermediate result values. These values are to be used subsequently

during the execution of the instruction *eval-op*. Consequently, a mechanism for saving intermediate results must be provided. A simple technique, which is used sometimes in conventional interpreters, inserts intermediate results directly into the appropriate argument fields of the instructions that make use of them. (The much more widely used push-down stack (last-in, first-out) requires a predetermined sequence in the execution of interpreter instructions and is not applicable for our more general control-tree.) For this purpose, the so-called *return information* is associated with each node of the control-tree. The return information specifies the argument position into which the intermediate result produced by the instruction of the node is to be inserted. Figure 15 illustrates the control-tree and the return information required for the evaluation of the expression x. The return information is indicated by a dashed line, which leads from the node where the intermediate result is produced to the argument position into which the value is to be inserted. The argument values of the instructions, e.g., *s-op*(x) or *s-opnd*1(x), are abstract trees. An argument value is represented by the null-tree Ω until the interpreter establishes the intermediate result value.

Figure 15 Control tree with return information



state transitions

Each execution of an interpreter instruction produces changes in the state of the abstract interpreter. These *state transitions* always include changes in the control component of the state. For example, the insertion of an intermediate result into the specified argument position represents such a change. Another obvious requirement is that eventually all subtrees of a node in the control-tree must be eliminated, to allow the execution of the instruction associated with the node.

Corresponding to the two instruction types, the macroinstructions and the basic instructions, we define two types of transformations for the control component of the interpreter state, the *macro-expansion* and the *basic transformation*.

macro-expansion

The definition of a macroinstruction consists of a finite set of control-trees. During execution of the macroinstruction, i.e., during a macro-expansion, one of these control-trees is selected by the execution control of the interpreter. This selected tree replaces the executed instruction in the control part of the interpreter state. In the example illustrated in Figure 15, the execution of the instruction *eval-expr* (*s-opnd*1(x)) results in the new control-tree shown in Figure 16. Note that the replacement process does not change the return information associated with the node where the replacement takes place. Of course, new return information may be established for the newly created nodes of the control-tree.

In Figure 16, all three *eval-expr* instructions are candidates for the next execution step. Let us assume that the instruction *eval-expr* (*s-opnd*2(*s-opnd*1(x))) is selected for execution. The tree component

Figure 16 Control tree after evaluation of an expression

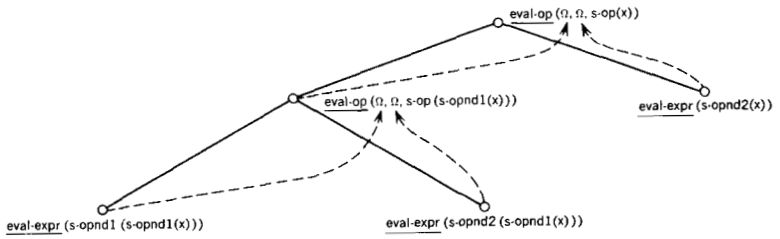


Figure 17 Replacement of eval-expr(s-opnd2(s-opnd1(x)))

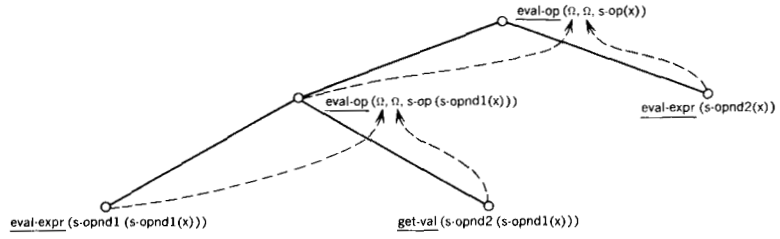
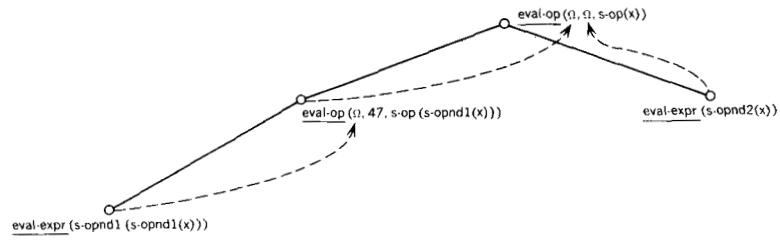


Figure 18 Execution of get-val(s-opnd2(s-opnd1(x)))



selected by *s-opnd2*(*s-opnd1*(*x*)) from the abstract expression *x* is the elementary tree B, i.e., a variable name. In this case, the macro-expansion results in the control-tree shown in Figure 17, in which the instruction *eval-expr*(*s-opnd2*(*s-opnd1*(*x*))) is replaced by the basic instruction *get-val*(*s-opnd2*(*s-opnd1*(*x*))).

basic transformations

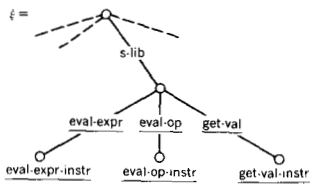
The executions of basic instructions, i.e., basic transformations, have in common that the node with the instruction is deleted from the control component of the state and that the intermediate result produced by execution of the instruction is inserted according to the return information associated with the node. Assuming that the value of the variable B in the value storage of the interpreter is 47, the execution of the instruction *get-val*(*s-opnd2*(*s-opnd1*(*x*))) in Figure 17 results in the modified control-tree shown in Figure 18.

library and the instructions

All instructions of the SPL interpreter are contained in the library component. To identify the individual instructions, the *instruction names* are used as selectors. For example, assume that the library only contains the three instructions *eval-expr*, *eval-op*, *get-val*. The state ξ has the form shown in Figure 19, where *eval-expr-instr*, *veal-op-instr*, and *get-val-instr* symbolize the instructions contained in the library. When an instruction is to be executed, the execution control of the interpreter

1. Selects the instruction from the library
2. Associates the arguments with the parameters by replacing each parameter occurrence in the instruction definition with the corresponding argument value
3. Performs the macro-expansion or the basic transformation specified by the instruction definition

Figure 19 State when library contains only three instructions



For convenience, a number of special notational conventions have been adopted in the representation of instruction definitions.

macro-definition

The format of a macro-definition is given in Table 2, where *instr* is the instruction name and $parm_1$, $parm_2$, $\cdots$, $parm_n$ are the parameters of the instruction. In an instruction definition without parameters, the parentheses do not appear.

One of the control-trees $c\text{-}tree_1$, $c\text{-}tree_2$, $\cdots$, $c\text{-}tree_m$ is selected during the macro-expansion to replace the executed instruction in the control component of the interpreter. For the selection of the control-tree, the conditional expressions $cond_1$, $cond_2$, $\cdots$, $cond_m$ are used. Note that the conditional expressions are not SPL expressions. They are used to control the actions of the abstract interpreter rather than being a part of the language to be interpreted. To distinguish expressions controlling the interpreter from SPL expressions, we shall use the term interpreter expressions where a conflict may arise.

Table 2 Macro-definition format

$$
\begin{aligned}
&instr\,(parm_1, parm_2, \cdots, parm_n) = \\
&\quad cond_1 \;\rightarrow\; c\text{-}tree_1 \\
&\quad cond_2 \;\rightarrow\; c\text{-}tree_2 \\
&\quad \vdots \\
&\quad cond_m \;\rightarrow\; c\text{-}tree_m
\end{aligned}
$$

The evaluation of a conditional expression produces either of the result values *true* or *false*. Conditional expressions may be as simple as a single logical constant, i.e., *true* and *false*, or a single test function, i.e., a predicate. More complex expressions may be formed using the symbols $+$, $-$, $<$, $\leq$, $=$, $\neq$, $\geq$, $>$ for the conventional arithmetic and comparison operators. For the definition of SPL, only these operators are required. Additional operators are defined in Reference 8. Examples of conditional expressions are

is-stmt(t)
s-st-id(t) = SET
true
s-stc(ξ) > *length* (*s-pgm*(ξ))

The first three expressions should be easy to understand; the fourth expression has the value *true* if the statement counter of the state ξ

Figure 20 Linear notation of control-tree

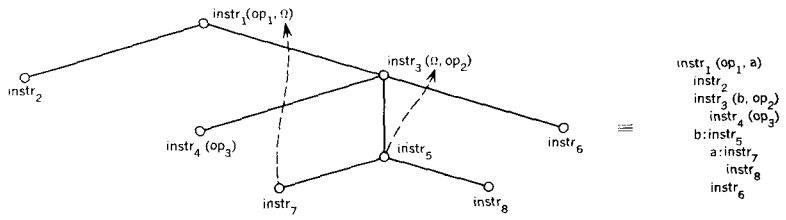
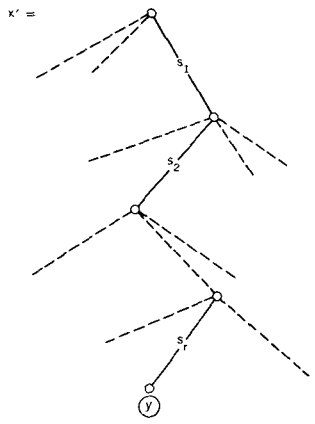


Figure 21 Results of function μ



indicates that the last statement in the program has been interpreted, i.e., its value is greater than the number of statements in the program. When executing a macroinstruction, the execution control evaluates the conditional expressions in the macro-definition in the order $cond_1$, $cond_2$, $\cdots$, until an expression evaluation produces the result value *true*. The control-tree associated with this expression is then used for the replacement in the control component of the state, and the execution of the macroinstruction ends.

To allow for a convenient representation of control-trees, a linear notation is used, as illustrated by the example in Figure 20.

In the linear form, indentation indicates the sub-levels of the control tree. *Dummy names*, *a* and *b* in the example, are used in the prefix of instructions that return values. The appearance of the dummy name in the prefix of an instruction and in an argument position of an instruction higher in the control tree indicates that the return value of the prefixed instruction is to be placed into the identified argument position. For example, the return value of the instruction $instr_7$ is to be placed into the second argument position of the instruction $instr_1$.

basic instruction

The format of a basic instruction definition is given in Table 3. The definition header for basic instructions has the same form and meaning as the definition header of macroinstructions. The expressions $t\text{-}expr_0$, $t\text{-}expr_1$, $\cdots$, $t\text{-}expr_m$ are interpreter expressions that produce an abstract tree when they are evaluated. The value of the expression $t\text{-}expr_0$ represents the intermediate result produced by the instruction. This result is inserted into the control component according to the return information of the node associated with the instruction. The selectors $s\text{-}sc_1$, $s\text{-}sc_2$, $\cdots$, $s\text{-}sc_m$ identify immediate components of the state that are to be replaced by the abstract trees produced by the expressions $t\text{-}expr_1$, $t\text{-}expr_2$, $\cdots$, $t\text{-}expr_m$. That is, the values of these expressions constitute the new or modified state components that result from the instruction execution.

Table 3 Basic instruction definition

$instr(parm_1, parm_2, \cdots, parm_n) =$	
PASS	: $t\text{-}expr_0$
$s\text{-}sc_1$	: $t\text{-}expr_1$
$s\text{-}sc_2$	: $t\text{-}expr_2$
.	
.	
.	
$s\text{-}sc_m$	: $t\text{-}expr_m$

The interpreter expressions specified in the instruction definition produce abstract trees as results; that is, they are specified using tree operations. The only tree operations discussed so far are the selector functions used for selecting subtrees from existing trees.

To specify the construction and the modification of trees, a new function is introduced. It is represented by the form

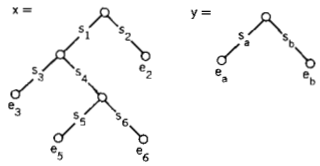
$\mu(x;\ \langle sel\text{-}sequ{:}y\rangle)$

where x and y are abstract trees and *sel-sequ* is a sequence of branch names, written as

$s_r \circ s_{r-1} \circ s_{r-2} \circ \cdots \circ s_1$

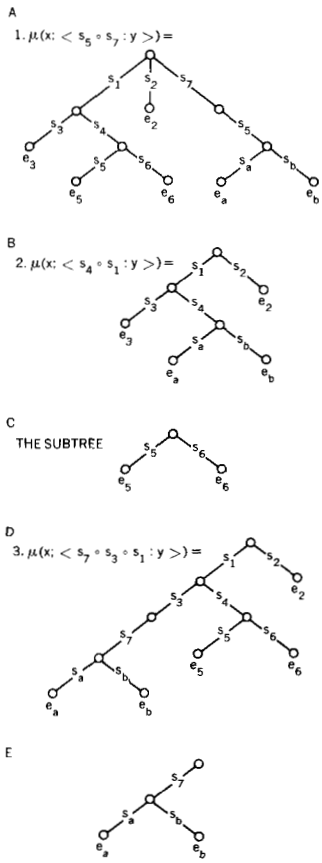
Using the branch names $s_1, s_2, \cdots, s_r$ for the identification, the function μ incorporates the tree y into the tree x. The resultant tree x' has the form shown in Figure 21, where any segments of the tree x that would lead to naming conflicts have been deleted from the tree. The order of the branch names in the function is reversed to indicate the order of selector functions $s_r(s_{r-1}(\cdots s_1(x')\ ,\ ,\ ,))$ used to select the tree y from the tree x'.

Figure 22 Trees x and y



As illustrative examples, consider the abstract trees shown in Figure 22. The result of the function μ for various choices of branch name sequences is shown in Figure 23. In the first example, Figure 23A, a complete new branch is added to the tree x, and no elements are deleted. In the second example, application of the function μ to tree x (shown at B) replaces the subtree of x (shown at C) by the tree y. The subtree at C must be eliminated from the tree x to avoid a naming conflict in selecting the new subtree y. In the third example (shown at D), elementary tree e_3 is replaced by the subtree shown at E to avoid conflicts in selecting the new tree components.

Figure 23 Function μ for various branch name sequences



Expressions used in the definition of the basic instructions may contain any of the tree manipulation functions in addition to the conventional arithmetic operators $+$ and $-$. (For the description of SPL, only these functions are required. For more complex definitions, additional operations are described in References 3 through 8.)

For convenience, state components not changed by execution of a basic instruction need not be specified in the list $s\text{-}sc_1, s\text{-}sc_2, \cdots, s\text{-}sc_m$.

If no intermediate result is produced by an instruction, the element

PASS: $t\text{-}expr_0$

may be deleted. If no parameters are required, the parentheses in the definition header must not be specified. For example, the basic instruction used to update the statement counter is defined as

up-stc =

$$s\text{-}stc : s\text{-}stc\ (\xi) + 1$$

As an additional convenience, the body of a basic instruction may appear instead of a control-tree in a macroinstruction. Assuming the macroinstruction format shown in Table 2, the appearance of the form

$$
\begin{aligned}
cond_i \rightarrow\ & \underline{PASS} : t\text{-}expr_0 \\
& s\text{-}sc_1 : t\text{-}expr_1 \\
& \quad \cdot \\
& \quad \cdot \\
& \quad \cdot \\
& s\text{-}sc_p : t\text{-}expr_p
\end{aligned}
$$

as one of the alternatives has the same meaning as the appearance of

$$cond_i \rightarrow \underline{b\text{-}instr}\ (parm_1, parm_2, \cdots, parm_n)$$

together with the appearance of the basic instruction

$$
\begin{aligned}
& \underline{b\text{-}instr}\ (parm_1, parm_2, \cdots, parm_n) = \\
& \qquad \underline{PASS} : t\text{-}expr_0 \\
& \qquad s\text{-}sc_1 : t\text{-}expr_1 \\
& \qquad\quad \cdot \\
& \qquad\quad \cdot \\
& \qquad\quad \cdot \\
& \qquad s\text{-}sc_p : t\text{-}expr_p
\end{aligned}
$$

Note that the new basic instruction uses all the parameters of the macroinstruction.

The SPL interpreter

The state components of the SPL interpreter and their general properties were introduced in the preceding section. We now investigate how the interpretation process of SPL programs progresses and which instructions are to be used by the SPL interpreter during interpretive execution.

initial state

To start the interpretation of an SPL program, initial information must be given to the abstract interpreter. This information constitutes what is called the *initial state* of the interpreter. The initial state of the SPL interpreter consists of

- The abstract program to be interpreted
- The value 1 for the statement counter
- The instruction *int-program* as the only instruction in the control component, i.e., *int-program* represents the top and only node in the control component
- The library containing all interpreter instructions

The value storage component of the state is empty.

The interpretation process starts with the execution of the instruction *int-program*, and it continues until the control component becomes empty.

instruction set

The instructions of the SPL interpreter can be classified into three principal groups, each one designed to handle a specific aspect of

SPL. For reference purposes, a sequential numbering of the instructions is introduced.

general control instructions

The interpretation of the SPL program starts with the macro-instruction

1. *int-program* =
   s-stc(ξ) ≤ *length*(*s-pgm*(ξ)) →
   int-program
   up-stc
   int-stmt(*elem*(*s-stc*(ξ))(*s-pgm*(ξ)))
   s-stc(ξ) > *length*(*s-pgm*(ξ)) → Ω

As long as the statement counter does not exceed the number of statements in the SPL program, the first alternative is chosen for the macro-expansion. It leads to the interpretation of one statement and to the subsequent updating of the statement counter. The macro-expansion also reproduces the instruction *int-program* in the top node of the control component. As soon as the statement counter exceeds the number of statements in the program, the instruction *int-program* is replaced by the null-tree, that is, the instruction is simply deleted from the control component. Since it is always associated with the top node of the control component, the control component becomes empty and the interpretation process stops.

2. *int-stmt* (*t*) =
   is-lab-stmt (*t*) → *int-stmt* (*s-unlab-stmt*(*t*))
   is-set-stmt (*t*) → *int-set-stmt* (*t*)
   is-goto-stmt (*t*) → *int-goto-stmt* (*t*)

where *t* is an abstract tree representing a labeled or unlabeled statement of SPL. The first alternative reproduces the instruction *int-stmt* in the control component, but the new argument of the instruction is the subtree representing the unlabeled statement part of the abstract tree *t*. The other alternatives lead to the execution of the specific statement.

3. *up-stc* =
   s-stc : *s-stc* (ξ) + 1

The instruction increases the value of the statement counter by one.

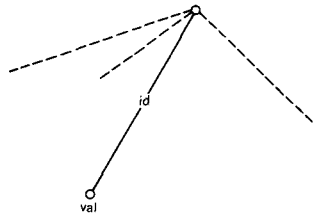
the set statement

4. *int-set-stmt* (*t*) =
   true → *assign-val* (*a*, *s-target* (*t*))
   a: *eval-expr* (*s-expr*(*t*))

where *t* is the abstract form of a set-statement. Note that the conditional expression *true* is always satisfied. That is, this alternative is always chosen for the macro-expansion. The intermediate value produced by the evaluation of *eval-expr* is returned to the first argument of the instruction *assign-val*.

5. *assign-val* (*val*, *id*) =
   s-vst : μ(*s-vst*(ξ); ⟨*id* : *val*⟩)

Figure 24 Value storage component



where *val* is an integer value and *id* is a variable name. After execution of the instruction, the value storage contains a component, as shown in Figure 24. If a component with the branch name *id* was already present in the storage before execution of the instruction, the old value associated with the branch name is replaced by the value *val*. If there was no such component, a new component of the form illustrated will have been added during the execution. In the initial state, the value storage is empty; consequently, a variable only appears in the value storage after at least one assignment to the variable has been encountered during interpretation.

6. *eval-expr* (*t*) =
   is-name (*t*) → *get-val* (*t*)
   is-constant (*t*) → *PASS*: *t*
   is-expr (*t*) → *eval-op* (*op*1, *op*2, *s-op* (*t*))
   *op*1 : *eval-expr* (*s-opnd*1 (*t*))
   *op*2 : *eval-expr* (*s-opnd*2 (*t*))

In this definition, the order of the conditional expressions is of importance. The predicate *is-expr* (*t*) is satisfied for all expressions, including simple variables and constants. Therefore, the tests for a variable and a constant must precede the test for an expression. Otherwise, the macro-expansions associated with the conditional expressions *is-name* (*t*) and *is-constant* (*t*) would never be utilized.

7. *get-val* (*id*) =
   is-const (*id*(*s-vst*(ξ))) → *PASS* : *id*(*s-vst*(ξ)))
   true → *error*

The first alternative is only chosen if a value associated with the name *id* is found in the value storage. If such a value does not exist, an assignment to the variable *id* has not been encountered during the interpretation and the program is in error. The error situation is indicated using the not further defined instruction *error*.

8. *eval-op* (*opnd*1, *opnd*2, *op*) =
   is-add (*op*) → *PASS*: *opnd*1 + *opnd*2
   is-subtract (*op*) → *PASS*: *opnd*1 − *opnd*2

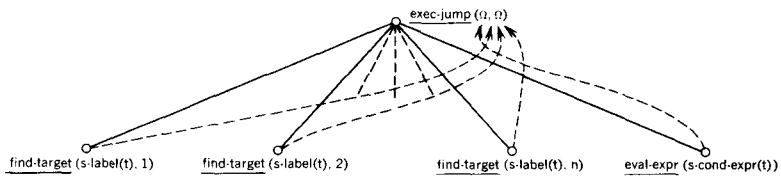
Depending on the operator *op*, the sum or the difference of the operand values *opnd*1 and *opnd*2 is returned as an intermediate result.

Note that if SPL imposed limitations on the size of the numbers that may be handled, the limitations would have to be included in the definition of the instruction *eval-op*, e.g., by specifying overflow or underflow actions for the abstract interpreter.

the goto statement

During the interpretation, the conditional SPL expression specified in the goto-statement must be evaluated, and the target label must be located in the statement sequence of the abstract program before a decision on the next statement to be interpreted can be made. SPL does not specify whether the conditional expression or the

Figure 25 *Go-to control-tree*



search for the target statement has to be performed first. In addition, a search algorithm for the target statement is not prescribed by SPL. The only condition to be considered is that the target label must appear exactly once in the prefix of one of the statements in the program.

9. *int-goto-stmt* $(t) =$
   true $\rightarrow$ *exec-jump* (*cond*, *target*)
   {*target*: *find-target* (*s-label* (t), *sc*)|
   $1 \leq sc \leq$ *length* (*s-pgm*(ξ))}
   cond: *eval-expr* (*s-cond-expr* (t))

The instruction definition utilizes notation commonly used in set theory when specifying that all the tests for the target statement and the evaluation of the conditional SPL expression may be processed in any order by the execution control of the interpreter. That is, the control-tree specified in the instruction definition has the form shown in Figure 25, where n is the number of statements in the abstract program.

A special convention is introduced that governs the processing of the return information provided for all the nodes *find-target*. Each of these nodes identifies the same argument (*target*) of the instruction *exec-jump*.

The definition of the instruction *find-target* specifies that either an elementary tree or a null-tree is produced as the intermediate result of the execution of the instruction. These two situations must be distinguished by the execution control of the interpreter as follows:

- If the intermediate result is the null-tree Ω, the return information is disregarded; no value is inserted into the second argument of the instruction *exec-jump*.
- If the intermediate result is not a null-tree, the execution control tests whether the argument identified by the return information is still the null-tree Ω, that is, whether no preceding execution of some other instruction has inserted an intermediate result into the argument position. If the argument is the null-tree, the intermediate result is inserted; if it is not, an error situation is encountered and the execution control automatically initiates the execution of the instruction *error*.

Note that this special convention provides a test, that a label must not appear more than once in the prefix of the statements in the abstract program.

10. *find-target* (*lab*, *sc*) =
    s-label (*elem* (*sc*)(*s-pgm*(ξ))) = *lab* $\rightarrow$ *PASS*: *sc*
    s-label (*elem* (*sc*)(*s-pgm*(ξ))) $\neq$ *lab* $\rightarrow$ *PASS*: Ω

The instruction returns the statement number *sc* if the label *lab* is found in the prefix of the investigated statement; otherwise, the null-tree is returned.

11. *exec-jump* (*val*, *tgt*) =
    tgt = Ω $\rightarrow$ *error*
    val > 0 $\rightarrow$ *s-stc*: *tgt*-1
    val < 0 $\rightarrow$ Ω

A value Ω for the second argument indicates that the label specified in the goto-statement could not be found during the executions of the instruction *find-target*; that is, the abstract program contains an error, and the instruction *error* is to be executed. If the first conditional expression is not satisfied, tests are made to determine whether the jump has to be performed or not.

A value *val* greater than zero leads to execution of the jump. This transfer of the interpretation process to the target statement is accomplished by replacing the statement counter component in the interpreter state by the value *tgt* $-$ 1, where *tgt* is the statement number of the statement containing the target label. The decrease by one is required because in the method chosen for incrementing the statement counter (see the definition of *int-program*), the counter contents is increased by one before the identified statement is actually interpreted.

Concluding remarks

The formal description methods introduced in this paper were related to our simple language SPL for explanation purposes. Actually, the methods (i.e., abstract trees, predicates, state components, control trees, etc.) apply to all programming languages. However, some programming languages contain facilities that cannot be expressed using the methods described in this paper alone. For example, additional definitions are required to represent the multitasking facilities of PL/1. To solve this particular problem, an extension to the execution control of the abstract interpreter was used in the formal description of PL/1.[3-9]

Similar problems are created when a language that has been formally defined is extended to provide additional facilities. However, formally defining language extensions helps to contain the often serious problems of identifying all parts of the language affected

and possibly changing them to accommodate the new facilities. Assume, for example, that floating-point arithmetic facilities are to be added to SPL. The conceptual approach to this problem is outlined in the following paragraphs (with the complete formal description of the extended SPL left as an exercise to the reader).

To allow floating-point operations in SPL, an analysis of the abstract interpreter shows that we shall at least have to expand the definition of the instruction *eval-op* to provide floating-point additions and subtractions and to provide conversions between integers and floating-point values when both appear in a single operation. Determining the needed conversions to the operands passed as parameters to the *eval-op* instruction requires, in turn, that these values be identified as either integers or floating-point numbers. Thus the elementary trees containing the operands in the original formal description must be changed to trees containing two subcomponents—one defining the data type, the other the value of the operand. Immediately it is clear that this change is not restricted to the instruction *eval-op*; any instruction handling arithmetic values will have to be changed to allow for the accompanying type information.

Some of the instructions so affected also refer to the storage component of the interpreter state. Thus we shall have to provide for a state component that allows values of variables in storage to be identified as either integers or floating-point numbers. For example, we could change the storage component using the abstract syntax

is-val-stg = ({⟨*name* : *is-elem*⟩ | | *is-name* (*name*)})
is-elem = (⟨*s-type* : *is-type*⟩,
⟨*s-val* : *is-constant*⟩)

where each storage element is represented as a tree composed of a type-describing component and the value component.

To characterize SPL variables as integer or floating-point, a *declaration* of the variables must appear in the abstract program. This declaration is then used by the abstract interpreter to construct the modified storage component of the interpreter state during interpretation. The abstract syntax of the *declare-statement* might have the form

is-dcl-stmt = (⟨*s-st-id* : DCL⟩,
⟨*s-dcl*: *is-dcl-var*⟩)
is-dcl-var = ({⟨*name* : *is-type*⟩ | | *is-name* (*name*)})
is-type = *is-integer* ∨ *is-float*

After adding new instructions to the interpreter to process the *declare-statement*, a concrete syntax of the *declare-statement* can be designed, and the necessary translator function can be defined to complete the formal definition of the extended SPL.

REFERENCES

1. P. Naur (Editor) "Revised report on the algorithmic language ALGOL 60," *Communications of the ACM* **6,** No. 1 (January 1963).
2. *American National Standard COBOL,* United States of America Standards Institute (1969).
3. M. Fleck and E. Neuhold, *Formal Definition of the PL/I Compile Time Facilities,* IBM Laboratory Vienna, Technical Report TR 25.080 (June 28, 1968).
4. K. Walk, K. Alber, K. Bandat, H. Bekic, G. Chroust, V. Kudielka, P. Oliva, and G. Zeisel, *Abstract Syntax and Interpretation of PL/I,* IBM Laboratory Vienna, Technical Report TR 25.082 (June 28, 1968).
5. P. Lucas, K. Laber, K. Bandat, H. Bekic, P. Oliva, K. Walk, and G. Zeisel, *Informal Introduction to the Abstract Syntax and Interpretation of PL/I,* IBM Laboratory Vienna, Technical Report TR 25.083 (June 28, 1968).
6. K. Alber, P. Oliva, and G. Urschler, *Concrete Syntax of PL/I,* IBM Laboratory Vienna, Technical Report TR 25.084 (June 28, 1968).
7. K. Alber and P. Oliva, *Translation of PL/I into Abstract Text,* IBM Laboratory Vienna, Technical Report TR 25.086 (June 28, 1968).
8. P. Lucas, P. Lauer, and H. Stigleitner, *Method and Notation for the Formal Definition of Programming Languages,* IBM Laboratory Vienna, Technical Report TR 25.087, (June 28, 1968).
9. P. Lucas and K. Walk, "On the formal description of PL/I" *Annual Review in Automatic Programming* **6,** Part 3, 105–181, Pergamon Press, New York (1970).
10. J. McCarthy, "Towards a mathematical science of computation," *Information Processing 1962,* 21–28, North-Holland Publishing Co., Amsterdam (1963).
11. J. McCarthy, "A formal description of a subset of ALGOL," *Proceedings of the IFIP Working Conference, Vienna 1964,* North-Holland Publishing Co., Amsterdam (1965).
12. P. J. Landin, "Correspondence between ALGOL60 and Church's Lambda-Notation, Part I," *Communications of the ACM* **8,** No. 2, 89–101 (1965).
13. P. J. Landin, "Correspondence between ALGOL60 and Church's Lambda-Notation, Part II," *Communications of the ACM* **8,** No. 3, 158–165 (1965).
14. C. C. Elgot and A. Robinson, "Random-access stored-program machines, an approach to programming languages," *Journal of the ACM* **11,** No. 4, 365–399 (1964).
15. P. Lauer, *Formal Definition of ALGOL 60,* IBM Laboratory Vienna, Technical Report TR 25.088 (December 13, 1968).
16. C. B. Jones and P. Lucas, "Proving correctness of implementation techniques," *Lecture Notes in Mathematics* **188,** Springer Verlag (1971).
17. H. Bekic and K. Walk "Formalization of storage properties" *Lecture Notes in Mathematics* **188,** Springer Verlag (1971).