Algorithmic Languages, de Bakker/van Vliet (eds.)

Formal Language Definitions Can Be Made Practical

Paul Klint

Mathematical Centre, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

If some formal method is used to define a programming language, the problem arises that individuals with different backgrounds and intentions have to learn a notation and definition method they are unfamiliar with. The various uses of formal definitions are summarized in this paper and an improved method for operational language definitions is presented. This method aims at language descriptions that are understandable and useful for both designer, implementor and user of a defined language. The method has been used in the definition of the SUMMER programming language. Various examples of that definition are given and the method as a whole is assessed.

> "... *The metalanguage of a formal definition must not become a language known to only the priests of the cult. Tempering science with magic is a sure way to return to the Dark Ages.*" [6]

1. The Problem

Programming languages are being designed using pre-scientific methods. Of course, there is no substitute for experience, taste, style and intuition but a scientific design methodology to support them is lacking. Methods for describing programming languages are somewhat more developed, but most definitions are either ambiguous and inaccurate, or excessively formal and unreadable. In general, a language definition method should:

(1) help the language *designer* by giving insight in the language he or she is designing and by exposing interactions that might exist between language features. The definition should at the same time be a pilot implementation of the defined language or it should at least be convertible into one. It is assumed here, that design and definition can best be carried out simultaneously.

(2) help the language *implementor* by providing him with an unam-

biguous and complete definition that is capable of "executing" small programs in cases where the implementor is in doubt about the meaning of a particular language feature.

(3) help the *user* by providing him with a precise definition in a language he is not too *un*familiar with.

These three goals impose different and to a certain extent contradictory requirements on the definition method to be used. In particular, it seems difficult to combine precision and readability in one method, since a precise definition has to use some formalism to which the reader has to be initiated and such a definition will have a tendency to become long and unreadable. This paper reports on an experiment with a language definition method that may be considered as a first step in satisfying the above requirements.

The *defined* language is SUMMER [3, 4] an object-oriented string processing language. The definition method is similar in spirit to the SECD method [5], i.e. it is an operational language definition method which uses recursive functions and syntactic recognition functions to define a finite state machine that associates semantic actions with all constructs in the grammar of the language. In the method presented in this paper readability has been considerably enhanced by using a few imperative constructs and by introducing a very concise notation for parsing and decomposing the source-text of programs in the defined language. SUMMER, extended with such parsing and decomposing operations, is used as *defining language.* The definition is hence circular (see Sections 2.1 and 3).

A complete description of the definition method can be found in [4]. The next section gives only a birds-eye view of the description method and shows some illustrative examples from the SUMMER definition. In Section 3 the method as a whole and its application to SUMMER are assessed.

2. The Method

2.1. *Introduction*

An *evaluation process* or *interpreter* (with the name "eval") will be defined that takes an arbitrary source text ("the source program") as input and either computes the result of the execution of that program (if it is a legal program in the defined language) or detects a syntactic or semantic error. The evaluation process operates directly on the text of the source

program and the process as a whole may be viewed as performing a series of string transformations on that text. During this process a global *environment* may be inspected or updated. An environment is a mapping from identifiers in the source program to their actual values during the evaluation process. Environments are used to describe concepts such as variables, assignment and scope rules.

A fundamental question arises here: in which language do we write the definition? Several choices can be made, such as the formalism used in denotational semantics ([1], this boils down to mathematical notation for recursive functions and domains) or the Vienna Definition Language ([8], a programming language designed for the manipulation of trees). This is not the right place to discuss the merits of these formalisms, but none has the desired combination of properties as described in the previous paragraph. Instead of designing yet another definition language, the defined language itself (this is SUMMER in the examples given in this paper) will be used as definition language. This choice has the obvious disadvantage that the definition is circular, but it has the practical advantage that readers who have only a moderate familiarity with the defined language will be able to read the definition without great difficulty. An extensive discussion of circular language definitions can be found in [7]. It should be emphasized that there is no *fundamental* reason to make the definition circular. The definition method described here would also work if, for example, ALGOL 68 was used as defining language. In any case, it is essential that the defining language has powerful string operations and allows the creation of data structures (of dynamically determined sizes). This requirement makes, for example, PASCAL less suited as defining language. Choosing SUMMER as defining language gave us the opportunity to investigate the suitability of that language in the area of language definition (see Section 3).

In the following sections the definition method and an example of its application (in the SUMMER definition) are described simultaneously. In Section 2.2 some aspects of the use of SUMMER as a metalanguage are described. The definition method can be subdivided in the definition of semantic domains (Section 2.3) and of the evaluation process (Section 2.4). Some more detailed examples from the SUMMER definition are given in Section 2.5.

2.2. *SUMMER as metalanguage*

This paragraph focuses on some aspects of SUMMER that are used in the

formal definition. Most of these constructs have some similarity with constructs in, for instance, PASCAL and are assumed to be self-explanatory. Only less obvious constructs that are essential for the understanding of the definition are mentioned here.

SUMMER is an object-oriented language with pointer semantics. This means that an object can be modified by assignment and that such modifications are visible through all access paths to that object. For example,

$$s := stack(10)$$

assigns a *stack* object of size 10 to the variable *s,* and

$$s\,.\,push(v)$$

pushes the value of v on the stack s. As a side-effect the stack s is modified such that subsequent operations on s may perceive the effect of that modification. In the formal definition this is relevant for the concepts "state" and "environment", which are modified in this way.

The language is dynamically typed, i.e. the type of variables is not fixed statically (as in PASCAL) but is only determined during the execution of the program (as in LISP or SNOBOL4). Moreover, generic operations on data structures are allowed. If an operation is defined on several data types, then the procedure to be executed when that operation occurs is determined by the type of the (left) operand of that operation.

Control structures and data structures are self-explanatory except possibly *arrays* and *for-statements.*

Arrays are vectors of values, indexed by $0, \ldots, N-1$, where N is the number of elements in the array. If A is an array then the operation $A\,.\,size$ will yield the number of elements in the array. A new array is created by

$$[V_0, \ldots, V_{N-1}]$$

or

$$array(N, V).$$

In the former case, an array of size N is created and initialized to the values $V_0, \ldots, V_{N-1}$. In the latter case, an array of size N is created and all elements are initialized to the value V. Array denotations are also allowed as left hand side of assignments. This provides a convenient notation for multiple assignments. For example,

$$[x, y, z] := [10, 20, 30]$$

is completely equivalent with

$$x := 10;\ y := 20;\ z := 30$$

and, more generally,

$$[x_0, \ldots, x_k] := a$$

is equivalent with

$$x_0 := a[0]; \ldots ; x_k := a[k].$$

The general form of a for-statement is:

for *V* **in** *G* **do** *S* **od**

where *V* is a variable, *G* is an expression that has as value an object capable of generating a sequence of values VAL_i and where *S* is an arbitrary statement. For each VAL_i the assignment $V := VAL_i$ is performed and *S* is evaluated. In this paper, the expression *G* will be used in two forms: the value of *G* is either an array (in which case consecutive array elements are generated) or *G* is an array on which the operation *index* has been performed (in which case all indices of consecutive array elements are generated). For example, in

a := [144, 13, 7];
for *x* **in** *a* **do** *print*(*x*) **od**

an array object is assigned to the variable *a* and the values *144, 13* and *7* will be printed, while

for *i* **in** *a*. *index* **do** *print*(*i*) **od**

will print the values *0, 1* and *2*. Further examples of for-statements will be found in the following paragraphs.

2.3. *Semantic domains*

A semantic domain is a set, whose elements either describe a primitive notion in the defined language (like "variable" or "procedure declaration") or have some common properties as far as the language definition is concerned. The relationship between these domains is given by a series of domain equations.

In the remainder of this paragraph the domains in the SUMMER definition

are briefly described. The abstract properties of these domains are given in [4]. Here, they are only introduced informally. First, the domain equations are given. Next, the meaning of each domain is described.

The relationship between the domains BASIC-VALUES, DENOTABLE-VALUES, STORABLE-VALUES, ENVIRONMENT, LOCATIONS, STATE, PROC, CLASS and INSTANCE is as follows

BASIC-VALUES = STRING ∪ INTEGER ∪ UNDEFINED
DENOTABLE-VALUES = LOCATIONS ∪ INSTANCE ∪ PROC ∪ CLASS ∪ BASIC-VALUES
STORABLE-VALUES = INSTANCE ∪ BASIC-VALUES
ENVIRONMENT = ID → DENOTABLE-VALUES
STATE = LOCATIONS → (STORABLE-VALUES ∪ {unused})
PROC = PROC-DECL × ENVIRONMENT
CLASS = ID × CLASS-DECL
INSTANCE = ID × CLASS-DECL × ENVIRONMENT

Here, ID, PROC-DECL and CLASS-DECL are the sets of string values that can be derived from the syntactic notions ⟨*identifier*⟩, ⟨*procedure-declaration*⟩ and ⟨class-declaration⟩ in the SUMMER grammar. BASIC-VALUES is the domain of primitive values in the language. DENOTABLE-VALUES is the domain of values which can be manipulated by the evaluation process. STORABLE-VALUES is the domain of values which can be assigned to variables in the source program. The domain LOCATIONS is used to model the notion "address of a cell capable of containing a value". Inspection of the contents of a location does not affect the contents of that location itself or of any other location. Modification of the contents of a location does not affect the contents of any other location. STATE is the domain that consists of functions that map locations on actual values or unused.

PROC is the domain of procedures. Each element of this domain describes a procedure declaration and contains a literal copy of the text of the procedure declaration itself and an environment that reflects all names and values available at the point of declaration.

CLASS is the domain of classes. Each element of this domain describes one class declaration and contains the name of the class and a literal copy of the text of the class declaration. INSTANCE is the domain of class instances. All values that are created by a SUMMER program are instances

of some class. An instance consists of the name of the class to which it belongs, the literal text of the declaration of that class and an environment that has to be used to inspect or update components from the instance. Operations are defined on elements in PROC, CLASS and INSTANCE to manipulate the components of an element in these domains. For completeness, these domains are mentioned here, but they will not be used in the remainder of this paper.

STRING, INTEGER and UNDEFINED are the domains modeling the values and operations for the built-in types *string, integer* and *undefined* respectively. UNDEFINED is the domain consisting of undefined values. All variables are initialized to an undefined value. Operations are defined on elements in STRING, INTEGER and UNDEFINED that model the primitive operations on the data types *string, integer* and *undefined.*

ENVIRONMENT is the domain of environments. Environments administrate the binding between names and values and the introduction of new scopes (i.e. ranges in the program where names may be declared). The operations defined on environments modify, in general, the environment to which they are applied.

The definitions given in following sections are centered around operations on elements of these semantic domains, but we will see relatively few of them in the examples. Operations will only be explained when they occur in an example.

2.4. *Evaluation process*

An extended form of BNF notation is used to describe the syntax of the defined language. The extensions aim at providing a concise notation for the description of repeated or optional syntactic notions. A syntactic notion suffixed with "+" means one or more repetitions of that notion. A notion suffixed with "*" stands for zero or more repetitions of that notion. The notation

{notion separator} replicator

i.e. a *notion* followed by a *separator* enclosed in braces followed by a *replicator,* is used to describe a list of notions separated by the given separator. A replicator is either "+" or "*". The replicator "+" indicates that the list consists of one or more notions. The list begins and ends with a notion. The replicator "*" indicates that the list consists of zero or more notions.

An optional syntactic notion is indicated by enclosing that notion in square brackets, e.g. "[*notion*]". The terminal symbols of the grammar are either enclosed in single quotes (for example: ',' or ':=') or written in upper case letters if the terminal symbol consists solely of letters (for example: *IF* may be used to denote the terminal symbol **if**). Where necessary, parentheses are used for grouping.

Some parts of a syntax rule may be labeled with a ⟨*tag*⟩; their meaning will become clear below.

The evaluation process is described in SUMMER extended with *parse expressions*[1] of the form

'{{' ⟨*identifier*⟩ '==' ⟨*syntax-rule*⟩ '}}'

which are used as a very concise notation for parsing and extracting information from the text of the source program. A parse expression succeeds if the identifier at the left hand side of the '==' sign has a string as value and if this string is of the form described by the ⟨*syntax-rule*⟩ at the right hand side of the '==' sign. All ⟨*tag*⟩s occurring in the ⟨*syntax-rule*⟩ should have been declared as variables in the program containing the parse expression, in this case the evaluation process. Substrings of the parsed text are assigned to these variables. If the recognized part of the text is a list or repetition, then an array of string values is assigned to the variable corresponding with the *tag*. Consider, for example, the following program fragment:

```
if {{e== WHILE t:⟨test⟩ DO b: ⟨body⟩ OD}}
then
    put ('e is a while expression')
fi
```

The parse expression will succeed if *e* has the form of a while expression; the literal text of the ⟨*test*⟩ is then assigned to variable *t* and the text of the ⟨*body*⟩ is assigned to variable *b*. Repetition occurs in

```
if {{e== VAR list:⟨test⟩ DO b:⟨body⟩ OD}}
```

[1] There is no *fundamental* reason to introduce this language extension. However, the disadvantage of introducing such an ad-hoc extension is more than compensated by the fact that we use a notation which is sufficiently similar to BNF notation to be almost self-explanatory. The effect of introducing a language extension as proposed here is interesting in its own right but falls outside the scope of the current discussion.

```
then
    put('e is a variable declaration containing:');
    for l in list do put (l) od
fi
```

The parse expression succeeds if *e* has the form of a "variable declaration" (i.e. the keyword **var** followed by a list of ⟨*identifier*⟩s separated by commas) and in that case an array of string values corresponding to the ⟨*identifier*⟩s occurring in the declaration is assigned to the variable *list,* which is printed subsequently.

Parse expressions may be used as test in if statements or may stand on their own. In the latter case, the string to be parsed *has* to be of the form described by the parse expression. In this way, parse expressions can be used to decompose a string with a known form into substrings.

In the case of the SUMMER definition, the overall structure of the evaluation process is:

```
var E;
var S;
var varinit;
proc ERROR
    ...;
proc eval(e)
(var value, signal, ...;
 if {{e == ⟨program-declaration⟩}}
 then
     ...
     return([value, signal])
 fi;
 if {{e == ⟨variable-declaration⟩}}
 then
     ...
     return([value, signal])
 fi;
 ...
 if {{e == ⟨empty⟩}}
 then
     ...
     return([value, signal])
```

```
    fi;
    ERROR
);
```

The variable *E* has as value the current environment and *S* has as value the current state. The variable *varinit* has as value a string consisting of the text of all ⟨*variable-initialization*⟩s in the current ⟨*block*⟩.

The procedure *ERROR* is called when a syntactic or semantic error is detected during evaluation. In that case, the whole evaluation process is aborted immediately. The main defining procedure is *eval,* which selects an appropriate case depending on the syntactic form of its argument *e*. Some examples of these various cases will be given in Section 2.5. Note that each of these cases involves a complete syntactic analysis of the string *e*. The evaluation process is initiated by creating an initial, empty environment *E* and by calling *eval* with the text of the source program as argument. If the evaluation process is not terminated prematurely (by the detection of a semantic error) the result of the evaluation of the source program can be obtained by inspecting the resulting environment *E*. Note how syntactically incorrect programs are intercepted in *eval* by *ERROR,* which is called if none of the listed cases applies.

The procedure *eval* delivers as result an *array* of the form [*value, signal*], where *value* is the actual result of the procedure and *signal* is a success/fail flag that indicates how *value* should be interpreted. SUMMER uses a success-directed evaluation scheme: an expression can either *fail* or *succeed.* These success/fail signals are used by language constructs like ⟨*if-expression*⟩ and ⟨*while-expression*⟩ to determine the flow-of-control. The *signal* delivered by *eval* is used to model this evaluation mechanism. This *signal* may have the following values:

N: evaluation terminated normally.
F: evaluation failed.
NR: normal return; a ⟨*return-expression*⟩ was encountered during evaluation.
FR: failure return; a failure return was encountered during evaluation.

The signal is tested after each (recursive) invocation of *eval.* In most cases eval performs an immediate return if the signal is not equal to *N* after the evaluation of a subexpression. Exceptions are cases such as ⟨*if-expression*⟩ and ⟨*return-expression*⟩ in which the signal is used to determine how evaluation should proceed. This organization has the effect that aborting the evaluation of the "current" expression, which is necessary if failure

occurs in a deeply nested subexpression, can be achieved by passing a signal upwards until it reaches an incarnation of *eval* that can take appropriate measures. The difference between *F* and *FR* lies in the language constructs that handle these cases. For example, consider ⟨*if-expression*⟩s. An *F* signal generated in the ⟨*test*⟩ part of an ⟨*if-expression*⟩ can be treated by the semantic rule associated with ⟨*if-expression*⟩s. But an *FR* signal generated during the evaluation of the ⟨*test*⟩ can only be treated by the semantic rule associated with the invocation of the procedure in which the ⟨*if-expression*⟩ occurs. In general, the signals *NR* and *FR* are only *generated* by return-expressions and are only *handled* by the semantic rules associated with procedure calls. The latter rules turn *NR* into *N* and *FR* into *F* before the evaluation process is resumed at the point where it left off to perform the (now completed) procedure call. All other semantic rules return immediately when an *NR* or *FR* signal occurs.

Note that the [*value, signal*] artifact is induced by the specific form of expression evaluation in SUMMER and has nothing to do with the definition method itself. We have just chosen one particular way to describe a form of goto statement.

2.5. *Some examples*

2.5.1. *If expressions*

⟨*if-expression*⟩s correspond to the if-then-else statement found in most programming languages. If evaluation of the ⟨*test*⟩ immediately contained in the ⟨*if-expression*⟩ terminates successfully, the ⟨*block*⟩ following **then** is evaluated. Otherwise, the successive ⟨*test*⟩s following subsequent *elif*s are evaluated until one such evaluation terminates successfully (in which case the following ⟨*block*⟩ is evaluated) or the list is exhausted. In the latter case, the ⟨*if-expression*⟩ may contain an **else** and then the ⟨*block*⟩ following that **else** is evaluated. The formal definition is:

```
1 if {{e == IF t: ⟨test⟩ THEN b: ⟨block⟩
2             elifpart: (ELIF ⟨test⟩ THEN ⟨block⟩)*
3             elsepart: [ELSE ⟨block⟩] FI}}
4 then
5     [v, sig] := eval(t);
6     if sig = N then return(eval(b))
7     elif sig ≠ F then return([v, sig])
8     else
```

```
 9          for ei in elifpart
10          do {{ei == ELIF t: ⟨test⟩ THEN b: ⟨block⟩}};
11             [v, sig] := eval(t);
12             if sig = N then return(eval(b))
13             elif sig ≠ F then return([v, sig]) fi
14          od;
15          if {{elsepart == ELSE b: ⟨block⟩}}
16          then
17             return(eval(b))
18          else
19             return([a_undefined, N])
20          fi
21       fi
22 fi;
```

The parse expression in lines *1–3* decomposes the string value of *e* in several parts. In line *5* the ⟨*test*⟩ of the ⟨*if-expression*⟩ is evaluated. Note how the occurrence of non-standard (i.e. *sig* = *NR* or *sig* = *FR*) signals terminates the evaluation of the ⟨*if-expression*⟩ (lines *7, 13*). This is particularly relevant for the evaluation of the ⟨*test*⟩ part. SUMMER allows the occurrence of a return statement in a ⟨*test*⟩. This is reflected in the above definition.

For a better understanding of the above definition, it may be useful to note that parts of the source program are parsed *repeatedly* during *one* evaluation of a given ⟨*if-expression*⟩. For example, the ⟨*block*⟩ following an **elif** is parsed both in lines *2* and *10*. (This explains, by the way, why the parse expression in line *10* needs not be contained in an if statement, see Section 2.4.) In general, the source text of the ⟨*if-expression*⟩ is parsed *each* time that it is evaluated.

2.5.2. *Variable declarations*

A ⟨*variable-declaration*⟩ introduces in the current environment a series of new variables, i.e. names of locations whose contents may be inspected and/or modified. The declaration may contain ⟨*expression*⟩s whose value is to be used for the initialization of the declared variables. First, these initializing expressions are evaluated. Next, the ⟨*expression*⟩s following the ⟨*variable-declaration*⟩s are evaluated. In the formal definition this is described by appending all variable initializations in the current ⟨*block*⟩ to the variable *varinit* and by evaluating the string value of that variable

before the evaluation of the subsequent ⟨*expression*⟩s in the ⟨*block*⟩. The formal definition of ⟨*variable-declaration*⟩s is:

```
 1 if {{e == VAR vi : {⟨variable-initialization⟩ ',' } + ';' }}
 2 then
 3     for v in vi
 4     do if {{v == x : ⟨identifier⟩ ':=' ⟨expression⟩}} then
 5           varinit := varinit ‖v‖ ';' ;
 6           E . bind(x, S . extend(a_undefined));
 7        else
 8           {{v == x : ⟨identifier⟩}};
 9           E . bind(x, S . extend(a_undefined))
10        fi
11     od;
12     return([a_undefined, N])
13 fi;
```

In line *1, e* is decomposed into an array of strings which have the form of a ⟨*variable-initialization*⟩. These string values are considered in succession in the for loop in lines *3–11*. If the ⟨*variable-initialization*⟩ contains an initializing expression, that expression is appended to *varinit* (line *5*) using the string concatenation operator "‖". In both cases, the state *S* is *extend*ed with a location containing an undefined value, and that new location is *bound,* in the current environment *E,* to the identifier being declared. Note that, in line *8, v* is known to have the form of an ⟨*identifier*⟩.

2.5.3. *Blocks*

A ⟨*block*⟩ introduces a new scope to be used for the declaration of new variables and constants. It consists of a (perhaps empty) list of declarations followed by a sequence of expressions separated by semicolons. A ⟨*block*⟩ is evaluated as follows:

(1) Evaluate all declarations.

(2) Evaluate all variable-initializations resulting from the evaluation of the declarations.

(3) Evaluate the sequence of expressions in the ⟨*block*⟩. (Note that SUMMER forbids the failure of an expression inside a sequence of expressions. Only the last expression in a sequence is allowed to fail; this failure is passed upwards to enclosing language constructs.)

The formal definition is:

```
1 if {{e == dlist: ⟨variable-declaration⟩*
2             elist: {[⟨expression⟩] ';' }*}}
3 then
4     var E1, varinit1;
5     E1 := E;
6     E.new_inner_scope;
7     varinit1 := varinit;
8     varinit := '' ;
9     for d in dlist
10    do [v,sig] := eval(d);
11       if sig ≠ N then ERROR fi
12    od;
13    [v,sig] := eval(varinit);
14    varinit := varinit1;
15    if sig ≠ N then E := E1; return([v,sig]) fi;
16    for i in elist.index
17    do
18       [v,sig] := eval(elist[i]);
19       case sig of
20       N:,
21       F: if i ≠ elist.size - 1 then ERROR fi,
22       NR: FR: (E := E1; return([v,sig]))
23       esac
24    od;
25    E := E1;
26    return([v,sig])
27 fi;
```

In lines *5–8* local copies are made of *E* and *varinit* and new values are assigned to them. In lines *9–13* the list of ⟨*variable-declaration*⟩s in the ⟨*block*⟩ and the resulting ⟨*variable-initialization*⟩s are evaluated. In lines *16–24* the list of ⟨*expression*⟩s in the ⟨*block*⟩ are evaluated. Note how failure of an expression in the middle of the list is treated (line *21*, see above).

3. Assessment

The formal language definition presented in the previous section will now be assessed. It is tempting to try to get statements like:

> *"Users can answer* 87% *of their questions on language issues within five minutes if they have access to a formal language definition of the kind described in this article."*

or

> "35% *of all run-time errors in user programs are directly related to anomalies in the language definition*".

In the absence of such results and with the methods to obtain them lacking, we have to live with qualitative and more or less speculative observations.

A rough indication for the *conciseness* of the definition can be obtained by comparing various sizes as they apply to the SUMMER definition:

formal definition	20 pages
reference manual	100 pages
implementation	200 pages

These figures show that the implementation is ten times larger than the formal definition. This is not surprising, since the implementation has to be efficient while the formal definition does not have to be. In this light the "a-language-is-defined-by-its-implementation" approach can be rephrased as: "*if* a language is defined by its implementation, *then* that implementation had better be small".

The definition is *precise* and *complete,* in the sense that *all* semantic operations associated with a particular language construct *have* to be specified to allow the construction of an *executable* version of the definition. The number of *operational details,* i.e. details in the definition which stem from the chosen definition method and have no inherent meaning in the defined language, is surprisingly small. This is a consequence of the choice of the defining language (which should have powerful data types and string manipulation operations) and the choice of high-level environment manipulation primitives which correspond directly to operations in the defined language and which are not (yet) perverted by implementational details. SUMMER extended with parse expressions seems a quite reasonable vehicle for language definition. It is, however, not possible to make continuation-style (see [1]) definitions, since higher-order functions are lacking.

It is difficult to give an objective judgement on the *readability* of the definition, but we have observed that only a moderate effort (a few days) is required on the part of a programmer without any training in formal semantics and without any previous exposure to the language to learn SUMMER using only the (annotated) formal definition.

The advantages and disadvantages of the formal definition for designer, implementor and user will now be discussed in some detail.

The advantages for the *designer* are:

(1) Anomalies in the design are magnified. It is a general rule that ill-formed entities can only be described by ill-formed descriptions or by descriptions which list many exceptional cases. It is easier to locate such exceptions or anomalies in a concise formal definition than in an ambiguous natural language definition or in a bulky implementation. In the SUMMER definition, for example, a very specific operation on environments is needed ("partial-state-copy") to accommodate the definition of just one language feature ("try-expression"). It turned out that a slight modification of that feature would at the same time simplify the definition and improve the feature.

(2) Exhaustive enumeration of language features. A formal definition method forces the designer to enumerate all language features in the same framework and this may help him to find omissions in the design.

(3) Interactions between language features can be studied. In the SUMMER definition, for example, the designer is forced to decide what happens when a ⟨*return-expression*⟩ is evaluated during the evaluation of any other expression. There is, however, no guarantee that all interactions can be found, since the formal definition may still contain hidden interactions between language features. The use of auxiliary functions in the definition is an aid in making interactions explicit. One may even apply techniques such as calling graph analysis and data flow analysis to the definition to discover clusters of interacting features and to establish certain properties of the definition.

(4) An executable formal definition can be tested and used. This may help eliminate clerical and gross errors from the definition. An executable definition allows the designer to play with (toy) programs written in the language he is designing. Here is, however, a problem with circular definitions: some implementation of the defined language has to exist before the definition itself can be made executable.

Disadvantages for the *designer* are:

(1) A considerable effort is required to construct a formal definition.

(2) A general problem is that there are no canned, satisfactory definition methods available and that the designer has to begin with either creating a new method or adapting and extending an existing one.

Advantages for the *implementor* are:

(1) Unambiguous language definition.

(2) The implementor may be in doubt as to the meaning of a certain combination of features. Such cases can be executed both by the implementation and by the definition and the results can be compared.

Disadvantages for the *implementor* are:

(1) The implementor must be familiar with the definition method or become acquainted with it. This is only a minor effort if one compares it with the total effort required to implement the language.

(2) It is non-trivial to derive an implementation strategy from the language definition. This is a problem shared by all "abstract" language definitions, in which no attempt is made to use primitives in the definition with a direct counterpart in an implementation. This leads to the conclusion that such abstract definitions should be accompanied by an "annotation for implementors", which states where well-known implementation techniques can be used and where certain optimizations are possible.

Advantages for the *user* are:

(1) Unambiguous and concise language definition.

(2) The user is used to reading programs and the formal definition can be read as such. In the case of a circular definition, the formal definition may be considered as a very informative example program.

Disadvantages for the *user* are:

(1) The user must be exposed to the definition method.

(2) A formal definition is harder to read than a "natural language" definition.

(3) In the case of the SUMMER definition, the circularity may be confusing for the naive user.

In retrospect, it seems justified to conclude that the method presented in this paper is a first step in satisfying the requirements given in Section 1. However, many problems remain to be investigated. Does the given method lend itself to mathematical analysis? How can the "complexity" of a language be derived from its definition? Is it possible to "optimize" the executable version of definitions? (Attempts in this direction can be found in [2].) What is the relationship between this definition method and extensible languages? Answers to these questions will provide more insight in the structure of programming languages and the methods for defining them.

Acknowledgement

J. Heering, H.J. Sint and A.H. Veen made useful comments on various drafts of this paper. Parts of it were discussed with L.J.M. Geurts, F.E.J. Kruseman Aretz and L.G.L.T. Meertens. I am grateful for their support.

References

[1] M.J.C. Gordon, The Denotational Description of Programming Languages (Springer, Berlin, 1979).
[2] N.D. Jones, Semantics-directed Compiler Generation (Springer, Berlin, 1980).
[3] P. Klint, An overview of the SUMMER programming language, in: Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages (Jan. 1980) pp. 47–55.
[4] P. Klint, SUMMER Reference Manual, Mathematical Centre, to appear.
[5] P.J. Landin, The mechanical evaluation of expressions, Comput. J. 6 (1964) 308–320.
[6] M. Marcotty, H.F. Ledgard and G.V. Bochmann, A sampler of formal definitions, Comput. Surveys 8 (1976) 191–276.
[7] J.C. Reynolds, Definitional interpreters for higher-order languages, Proceedings ACM Annual Conference (Aug. 1972) pp. 717–740.
[8] P. Wegner, The Vienna Definition Language, Comput. Surveys 4 (1972) 5–63.