Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J. Heering, P. Klint

A syntax definition formalism

Computer Science/Department of Software Technology Report CS-R8633 September

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

A Syntax Definition Formalism

J. Heering
Department of Software Technology, Centre for Mathematics and Computer Science

P. Klint
Department of Software Technology, Centre for Mathematics and Computer Science
Department of Computer Science, University of Amsterdam

The goal of the GIPE project is to generate interactive programming environments from formal language definitions. We currently envisage language definitions consisting of three parts: (a) a syntax section; (b) a static constraints section; (c) a dynamic semantics section. An initial version of the language definition formalism is being developed. In this paper we concentrate on the syntax section and introduce a new formalism which allows concrete and abstract syntax of specification (and other) languages to be defined simultaneously. The new formalism can be combined with a variety of specification languages. By doing so these obtain fully general user definable syntax.

1986 CR Categories: D.2.1 [**Software Engineering**]: Requirements/Specifications — *languages;* D.3.1 [**Programming Languages**]: Formal Definitions and Theory — *syntax;*

1980 Mathematics Subject Classification: 68B99 [**Software**].

Key Words & Phrases: user-definable syntax, abstract syntax, syntax definition formalism.

Note: Partial support received from the European Communities under ESPRIT project 348 (Generation of Interactive Programming Environments - GIPE).

Note: This paper will be published in *ESPRIT '86, Status Report of Continuing Work*, Amsterdam: North-Holland.

1. INTRODUCTION

1.1. Overall goals of the GIPE project

The goal of ESPRIT Project 348 (GIPE — Generation of Interactive Programming Environments) is to generate interactive programming environments from formal language definitions. In principle, a formal language definition need only contain a specification of the syntax and semantics of the language to be defined. As far as the language proper is concerned this is sufficient. If, however, a programming environment is to be derived from such a definition, it must contain additional information so as to enable the environment generator to produce language-dependent programming tools like a syntax-directed editor, type-checker, evaluator, symbolic tracer, and file system.

We currently envisage language definitions consisting of three parts:

(a) A *syntax section* containing a definition of the abstract and concrete syntax (including lexical syntax and prettyprinting) of the language to be defined. From this part the environment generator has to derive a syntax-directed editor.

(b) A *static constraints* section containing a definition of the *type constraints* or *static semantics* of the language. From this part the programming environment generator has to derive an incremental typechecker.

(c) A definition of the *(dynamic) semantics* of the language. From this part the environment generator has to derive an incremental evaluator and, at a later stage, a compiler.

Reflecting the above partitioning of language definitions, the overall language definition formalism we currently have in mind has three well-defined subformalisms. In this paper we concentrate on section (a) and introduce a new formalism which allows concrete and abstract syntax of

specification (and other) languages to be defined simultaneously.

1.2. The definition of syntax

It is common practice to make a distinction between the *parse tree* of a text in a programming or specification language L and its *abstract syntax tree*. The parse tree is the derivation showing how the text in question can be derived from the start symbol of the grammar of L. Its interior nodes are non-terminals of the grammar and its leaves are the lexical tokens, such as keywords, operator symbols, identifiers, etc., making up the original text. The abstract syntax tree only contains the essential information describing the text; its interior nodes are the constructors (also called operators) of the language and the direct descendants of each node are its operands. The leaves of the abstract syntax tree are identifiers, integer constants, etc.

Let us first of all summarize the arguments for using some structured representation of programs or specifications instead of the text itself:

(1) By using a structured representation one may abstract from the details of the text. A single structured representation may correspond to a whole class of variations of a text obtained by using "abstractly equivalent" but textually different lexical conventions or syntactic variations.

(2) Using the textual representation makes it difficult to give a semantics based on equations or rewrite rules, because in this case all semantic actions have to be expressed as string transformations. Such transformations easily become ambiguous, however: combinations of strings arising during processing may (erroneously) be interpreted as language constructs not occurring in the original text. These ambiguities are avoided by using a bracketed (and thus structured) representation of the text.

(3) A structured representation allows more efficient processing and editing since repeated parsing of source text can be avoided.

Which structured representation should one use? The main arguments in favour of using abstract syntax trees rather than parse trees are:

(1) Parse trees may contain chains of non-terminals that describe the intermediate steps necessary for deriving a string from a certain non-terminal. These derivation chains are mostly redundant and do not concisely express the essential structure of the string.

(2) The syntactic definition of (a) priority and associativity of operators, and (b) syntactic iteration, i.e. lists of items, introduce non-terminals which need not appear in the abstract syntax tree.

(3) The use of parser generators accepting restricted classes of grammars (e.g. LR(k), LALR(k), LL(k), etc.) may also lead to the introduction of "unnecessary" non-terminals.

(4) For different semantic purposes one may wish to associate different abstract syntax trees with one parse tree.

The major part of the translation between parse trees and abstract syntax trees is typically devoted to the elimination of non-terminals introduced in (1), (2) and (3).

In most compiler generation systems a distinction is made between the definition of lexical syntax and construction rules for lexical tokens, the definition of context-free syntax, and the translation rules from parse trees to abstract syntax trees. The formalisms used for defining them are completely different.

In this paper we develop a syntax definition formalism—called *SDF*—with the following properties:

(1) It allows the simultaneous definition of concrete (i.e. lexical and context-free) and abstract syntax in a uniform way.

(2) The formalism implicitly defines a translation from parse trees to abstract syntax trees. The latter are described by a (first-order) signature which is derived from the *SDF*-definition.

(3) The need for introducing "unnecessary" non-terminals is eliminated in many cases. This is achieved (a) by using *priority rules* to eliminate undesirable derivation chains; (b) by using

associativity attributes; (c) by allowing *arbitrary context-free grammars.*

(4) It can be used to add user definable syntax to any formalism based on first-order signatures.

In section 2 we discuss related work and describe two existing syntax definition formalisms: LEX/YACC and METAL. In section 3 we give an informal description of *SDF*. Next, in section 4, three simple examples are given using LEX/YACC, METAL and *SDF*. Some concluding remarks are made in section 5.

2. EXISTING FORMALISMS FOR THE DEFINITION OF SYNTAX

2.1. General

There have been many attempts to introduce user-definable syntax in programming languages. These ideas have led to user-definable syntax for operators in ALGOL68, PROLOG, SNOBOL4 and other programming languages, and to various styles of macro-definitions (PL/I, LISP). Around 1970, there was much interest in so-called *extensible languages* [8,15,16]. The aim of this line of research was to define a small base language in combination with a syntax definition mechanism. New language constructs could then be added to the base language by defining their syntax and by describing their semantics in terms of the base language. For various reasons, however, the overall goal of full syntactic and semantic language extensibility has never been completely achieved. Although it does not have a syntactic extension mechanism, SMALLTALK-80 may be viewed as the most successful extensible language in existence. (SMALLTALK-72—a predecessor of SMALLTALK-80—did support syntactic extensibility: each class had to parse the messages sent to it explicitly. This feature has been replaced by a more limited scheme of keyword parameters in SMALLTALK-80. This new scheme results in more readable programs and allows a more efficient implementation.)

A successful method for defining the syntax of language constructs is by means of *syntax-directed translations* [2,7]. LEX/YACC and METAL (to be discussed in the next sections) fall into this category. The LITHE system [14] combines syntax-directed translation with classes. Apart from the fact that its lexical syntax is fixed, LITHE has user-definable LR(1)-syntax. O'Donnell's notational specifications [13] constitute an alternative to syntax-directed translations.

In [3] we gave a complete algebraic specification of the syntax and semantics of a small programming language. Our dissatisfaction with the algebraic specification of syntax as presented there has motivated the development of *SDF*. Most closely related to our work are the user-definable *distfix* operators in HOPE [4] and OBJ2 [6].

2.2. LEX and YACC

LEX and YACC are, respectively, the scanner generator and parser generator of the UNIX system. We restrict our description of LEX and YACC to two small examples, since both systems have been described extensively in the literature [1,9,12].

LEX uses regular expressions to describe the syntax of lexical tokens. These are compiled to tables for a deterministic finite automaton. A typical LEX definition for identifiers is:

```
[a-z][a-z0-9]*  { id(yytext); }
```

The left-hand part of this rule describes the syntax of identifiers (i.e. a letter followed by zero or more letters or digits) and the right-hand part describes the action to be performed when the rule matches. Actions are written in the C programming language. In the above example the procedure `id` is called. The global variable `yytext` is part of the LEX/C interface and has as value a character array containing the text of the current token.

YACC uses a BNF-like notation for defining grammars and compiles these definitions to tables for a shift/reduce parser. Input grammars have to satisfy the LALR(1) restriction. A part of a typical YACC definition for defining a `while`-statement is:

```
%token WHILE, DO, ENDWHILE

 ... definitions for expr and series ...

while : WHILE expr DO series ENDWHILE
                { $$ = mk-while( $2, $4 ); }
```

The rule starting with `%token` declares the lexical tokens of the language (defined separately by means of LEX definitions). The second rule defines the syntax for a `while`-statement. Its left-hand part gives the syntax definition, and its right-hand part the code to be executed when a `while`-statement is recognized. The value of variable `$`*i* is the value returned by the *i*-th component of the rule after matching. The action associated with the rule returns a value in variable `$$`.

Syntax rules may contain alternatives (which are separated by the `|`-operator) but there is no mechanism for expressing repetition.

2.3. METAL

METAL [10] is the syntax-definition formalism of the MENTOR system. A language definition in METAL specifies:

(1) the lexical tokens of the language,

(2) its context-free syntax, and

(3) its abstract syntax.

The METAL compiler translates METAL specifications to specifications for an existing scanner/parser generator. The UNIX implementation of MENTOR, for instance, compiles METAL into input for LEX and YACC.

The actual form of lexical tokens is left unspecified in METAL definitions: these are specified in the language of the host scanner generator.

BNF-notation is used to define context-free syntax. A typical rule in a METAL definition is:

```
<while_stat> ::= while <exp> do <series> od ;
        while(<exp>, <series>)
```

Non-terminals are between angle brackets and terminals are just written as they are. Characters in terminals that conflict with the syntax notation have to be escaped (using a sharp sign). The formalism allows neither alternation nor repetition in syntax rules.

The part of the rule following the semicolon specifies the abstract syntax tree to be built for the construct. In the above example a tree labeled with the operator "`while`" with the abstract trees corresponding to `<exp>` and `<series>` as descendants. The abstract syntax of a language is defined by means of *functions* and *sorts* (or *operators* and *phyla* in METAL terminology). Functions are the constructors of abstract syntax trees. Functions with fixed arity are allowed to have descendants of different sorts. Functions with arity zero are the leaves of the abstract tree and represent the atoms of the language. Varyadic functions (also called *list functions*) are only allowed to have descendants of the same sort.

A typical definition of a function is:

```
while   -> EXP SERIES;
```

which defines the function `while` with two arguments which are respectively of sorts `EXP` and `SERIES`. The output sort of functions is specified in a separate definition (see below). The formalism allows the definition of list functions:

```
series  -> STATEMENT + ... ;
```

This defines the function `series` which may have one or more arguments, all of sort `STATEMENT`. Constructors for the empty list (e.g. `series-list`) and for prepending (e.g. `series-pre`) and appending (e.g. `series-post`) items to an existing list are implicitly defined.

The output sorts of functions are defined by enumerating all functions with a given output sort. For instance,

```
STATEMENT ::= assign if while;
```

specifies that the functions `assign`, `if` and `while` all have `STATEMENT` as output sort. Hence, the complete specification of the input and output sorts of, for instance, the function `while` expressed in METAL is

```
while     -> EXP SERIES
STATEMENT ::= while;
```

which is equivalent to

$$while : EXP \times SERIES \rightarrow STATEMENT$$

using conventional mathematical notation. In the sequel we will use # instead of $\times$ to indicate the Cartesian product.

3. INFORMAL DESCRIPTION OF THE SYNTAX DEFINITION FORMALISM

One may characterize *SDF* roughly as "signatures with an integrated concrete syntax definition mechanism": we start with signatures—being a convenient formalism for expressing abstract syntax—and add to them mechanisms for defining lexical and context-free syntax.

Signatures consist of definitions of sorts, and constants and functions over these sorts [5,11]. For each sort *s* in the signature, we define the derived sorts *s** and *s+*, denoting lists with elements of sort *s* containing respectively zero or more and one or more elements.

An example will illustrate the kind of signatures we have in mind. First, we define the signature of tables with three sorts `KEY`, `ENTRY`, and `TABLE`, constants `a, . . . , z`, `0`, `1`, and functions `pair`, `table`, `lookup`, and `delete`.

```
module Tables
begin
   sorts KEY, ENTRY, TABLE

   functions
      a      :                -> KEY
       ...
      z      :                -> KEY

      0      :                -> ENTRY
      1      :                -> ENTRY

      pair   : KEY # ENTRY    -> PAIR
      table  : PAIR*          -> TABLE
      lookup : KEY # TABLE    -> TABLE
      delete : KEY # TABLE    -> TABLE
```

```
end Tables
```

Signatures are extended with a mechanism for defining concrete syntax: definitions of constants and functions will not only have to specify their input and output sorts but also their concrete syntactic form. This syntactic information is mixed with the description of the input sorts. In this manner, the function definitions

```
"(" KEY "," ENTRY ")"  -> PAIR
"[" { PAIR "," }* "]"  -> TABLE
```

introduce, for instance, new syntactic forms for the functions `pair` and `table`. Instead of the prefix terms `pair(a,1)` and `table(pair(b,1),pair(a,0),pair(d,0))`, one should now write `(a,1)` and `[(b,1),(a,0),(d,0)]`. A complete *SDF*-definition for `Tables` might look as follows:

```
module Tables
begin
   lexical syntax
      sorts KEY, ENTRY
      layout SPACE

      functions
         [a-z]                    -> KEY
         0                        -> ENTRY
         1                        -> ENTRY
         " "                      -> SPACE

   context-free syntax
      sorts TABLE

      functions
         "(" KEY "," ENTRY ")"    -> PAIR
         "[" { PAIR "," }* "]"    -> TABLE
         lookup KEY in TABLE      -> TABLE
         delete KEY from TABLE    -> TABLE
end Tables
```

For derived sorts two forms are available:

(1) *s** (or *s*+) stands for zero or more (one or more) repetitions of *s*.

(2) {*s t*}* (or {*s t*}+) stands for zero or more (one or more) repetitions of *s separated by* the symbol *t*.

Several other aspects of the concrete syntactic form of functions may be defined:

(3) The lexical syntax is specified in a separate section. *Character classes*, like `[a-z]` in the above example, may be used to abbreviate lexical definitions. *Layout sorts* are used to define layout.

(4) The priority of a function declaration relative to other function declarations. The usual priorities of the arithmetic functions `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division) and `exp` (exponentiation) can, for instance, be defined by the following priority declaration:

```
priority ("+", "-") < ("*", "/") < exp
```

(5) If the syntactic representation of a function may be surrounded by parentheses (in order to change its priority), the attribute `par` is added to its definition.

(6) If a function is associative, left-associative, or right-associative, the attribute `assoc`, `left-assoc`, or `right-assoc` is added to its definition.

In section 4 we will give more examples of *SDF*-definitions.

4. THREE EXAMPLES

Three examples are now given to illustrate the descriptive power of our formalism. They demonstrate the syntax definition of expressions, statements and lexical constants. For each example we give

(1) the desired concrete and abstract syntax,

(2) a definition in LEX/YACC,

(3) a definition in METAL, and

(4) a definition in *SDF*.

4.1. Expressions

This example consists of a grammar for simple arithmetic expressions. It illustrates the definition of operator priorities in the various formalisms.

4.1.1. Concrete and abstract syntax

The concrete syntax for arithmetic expressions is:

```
expr   ::= term | expr "+" term
term   ::= factor | term "*" factor
factor ::= id | "(" expr ")"
```

where we leave `id` unspecified.

The desired abstract syntax is:

```
add    : EXPR # EXPR -> EXPR
mul    : EXPR # EXPR -> EXPR
id     : ID          -> EXPR
```

4.1.2. Definition in YACC

```
%token ID
%left '+'
%left '*'
%%
expr   : expr '+' expr          { $$ = add($1, $3); }
       | expr '*' expr          { $$ = mul($1, $3); }
       | '(' expr ')'           { $$ = $2; }
       | ID                     { $$ = id($1); }
       ;
```

YACC allows the definition of priority and associativity of operators. The lines `%left '+'` and `%left '*'` define `+` and `*` as left-associative operators. Consecutive lines define operators

with increasing priority. Hence, * has higher precedence than +. The definitions of the functions add, mul and id are not shown; they are written in C and construct nodes of the abstract syntax tree.

4.1.3. Definition in Metal

```
definition of EXPRESSIONS is

    chapter 'expr-rules'
        rules
            <exp>               ::= <term> ;
                <term>
            <exp>               ::= <exp> #+ <term> ;
                add(<exp>,<term>)
            <term>              ::= <factor> ;
                <factor>
            <term>              ::= <term> #* <factor> ;
                mul(<term>,<factor>)
            <factor>            ::= <id> ;
                <id>
            <factor>            ::= #( <exp> #) ;
                <exp>
            <id>                ::= %ID ;
                id-atom(%ID)
        abstract syntax
            add                 -> EXP EXP;
            mul                 -> EXP EXP;
            id                  -> implemented as IDENT;

            EXP                 ::= add mul;
            ID                  ::= id;
    end chapter;
end definition
```

Note that Metal does not provide means to define the associativity or priority of operators.

4.1.4. Definition in *SDF*

```
module exprs
begin
   lexical syntax
      sorts ID
   context-free syntax
      sorts EXPR
      priority "+" < "*"
      functions
         EXPR "+" EXPR -> EXPR   {par, left-assoc}
         EXPR "*" EXPR -> EXPR   {par, left-assoc}
         ID            -> EXPR
end exprs
```

4.2. Statements

The following grammar for simple statements illustrates the treatment of syntactic iteration, i.e. the description of lists of syntactic notions.

4.2.1. Concrete and abstract syntax

The concrete syntax for statements is:

```
if     ::= "if" expr "then" series "else" series "endif"
while  ::= "while" expr "do" series "endwhile"
stat   ::= if | while
series ::= { stat ";" }*
```

where we leave `expr` unspecified.

The notation `{ <stat> ';' }*` is used to indicate lists of `<stat>`s separated by `';'` and is equivalent to

```
series ::= "" | series1
series1::= stat | stat ";" series1
```

The desired abstract syntax for statements is:

```
if     : EXPR # SERIES # SERIES -> STAT
while  : EXPR # SERIES          -> STAT
series : STAT*                  -> SERIES
```

4.2.2. Definition in YACC

```
%token IF, THEN, ELSE, ENDIF, WHILE, DO, ENDWHILE
%%

stat   : if                     { $$ = $1; }
       | while                  { $$ = $1; }
       ;
if     : IF expr THEN series ELSE series ENDIF
                                { $$ = if($2, $4, $6); }
       ;
while  : WHILE expr DO series ENDWHILE
                                { $$ = while($2,$4); }
       ;
series :                        { $$ = emptyseries; }
       | series1                { $$ = $1; }
       ;

series1: stat                   { $$ = series($1, emptyseries); }
       | stat ';' series1       { $$ = series($1, $3); }
       ;
```

4.2.3. Definition in Metal

```
definition of STATEMENTS is

    chapter 'stat-rules'
        rules
            <stat>              ::= <if> ;
                <if>
            <stat>              ::= <while> ;
                <while>
            <if>                ::= if <exp> then <series>
                                                  else <series> endif ;
                if(<exp>,<series>.1,<series>.2)
            <while>             ::= while <exp> do <series> endwhile ;
                while(<exp>,<series>)
            <series>            ::=  ;
                series-list(())
            <series>            ::= <stat> ;
                series-list(<stat>)
            <series>            ::= <stat> #; <series> ;
                series-pre (<stat>,<series>)
        abstract syntax
            if                  -> EXP SERIES SERIES;
            while               -> EXP SERIES;
            series              -> STAT * ... ;

            SERIES              ::= series;
            STAT                ::= if while;
    end chapter;
end definition
```

4.2.4. Definition in *SDF*

```
module stats
begin
   context-free syntax
      sorts EXPR, STAT, SERIES
      functions
         if EXPR then SERIES else SERIES endif  -> STAT
         while EXPR do SERIES endwhile          -> STAT
         { STAT ";" }*                          -> SERIES
end stats
```

4.3. Lexical constants

This example illustrates the definition of natural numbers and identifiers as lexical constants.

4.3.1. Concrete and abstract syntax

The concrete syntax for natural numbers and identifiers is:

```
int      ::= digit+
id       ::= letter id-tail*
id-tail  ::= letter | digit
letter   ::= "a" | ... | "z"
digit    ::= "0" | ... | "9"
```

The desired abstract syntax is:

```
int : DIGIT+             -> INT
id  : LETTER # ID-TAIL*  -> ID
a   :                    -> LETTER
 ...
z   :                    -> LETTER
0   :                    -> DIGIT
 ...
9   :                    -> DIGIT
i1  : LETTER             -> ID-TAIL
i2  : DIGIT              -> ID-TAIL
```

Note that `i1` and `i2` are injection functions.

4.3.2. Definition in LEX

Both YACC and METAL rely on LEX for the definition of lexical syntax:

```
%%
[a-z][a-z0-9]*          { id(yytext); }
[0-9]+                  { int(yytext); }
%%
```

`id` and `int` are C functions (not shown here) which construct lexical tokens on the basis of the input text recognized.

4.3.3. Definition in *SDF*

```
module lexical-constants
begin
   lexical syntax
      sorts DIGIT, LETTER, INT, ID, ID-TAIL
      functions
         [a-z]              -> LETTER
         [0-9]              -> DIGIT
         DIGIT+             -> INT
         [a-z0-9]           -> ID-TAIL
         LETTER ID-TAIL*    -> ID
end lexical-constants
```

5. CONCLUDING REMARKS

In a forthcoming report we will give a formal definition of *SDF* and further examples. Additional reports by Jan Rekers en Paul Hendriks will cover the implementation of *SDF* and the combination of *SDF* with the algebraic specification formalism introduced in [3].

ACKNOWLEDGEMENTS

Comments made by Niek van Diepen, Paul Hendriks, Jan Rekers, and Ard Verhoog are gratefully acknowledged.

REFERENCES

[1] A.V. Aho, R. Sethi & J.D. Ullman, *Compilers. Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] A.V. Aho & J.D. Ullman, *The Theory of Parsing, Translation, and Compiling*, 2 Vols., Prentice-Hall, 1972.

[3] J.A. Bergstra, J. Heering & P. Klint, Algebraic definition of a simple programming language, Centre for Mathematics and Computer Science, Report CS-R8504, 1985.

[4] R.M. Burstall, D. MacQueen & D. Sanella, HOPE: an experimental applicative language, *Conf. Record of the 1980 LISP Conference*, Stanford University, 1980, pp. 136-143.

[5] H. Ehrig & B. Mahr, *Fundamentals of Algebraic Specifications*, Vol. I, *Equations and Initial Semantics*, Springer-Verlag, 1985.

[6] K. Futatsugi, J.A. Goguen, J.P. Jouannaud & J. Meseguer, Principles of OBJ2, *Conf. Record 12th Ann. ACM Symp. Principles of Programming Languages*, ACM, 1985, pp. 52-66.

[7] E.T. Irons, A syntax directed compiler for Algol 60, *Comm. ACM* **4** (1961) 1, pp. 51-55.

[8] E.T. Irons, Experience with an extensible language, *Comm. ACM* **13** (1970) 1, pp. 31-40.

[9] S.C. Johnson, YACC: yet another compiler-compiler, in: *UNIX Programmer's Manual*, Vol. 2B, Bell Laboratories, 1979.

[10] G. Kahn, B. Lang, B. Mélèse & E. Morcos, METAL: a formalism to specify formalisms, *Science of Computer Programming*, 3(1983), pp. 151-188.

[11] H.A. Klaeren, *Algebraische Spezifikation: Eine Einführung*, Springer-Verlag, 1983.

[12] M.E. Lesk & E. Schmidt, LEX - A lexical analyzer generator, in: *UNIX Programmer's Manual*, Vol. 2B, Bell Laboratories, 1979.

[13] M.J. O'Donnell, *Equational Logic as a Programming Language*, MIT Press, 1985.

[14] D. Sandberg, LITHE: A language combining a flexible syntax and classes, *Conf. Record 9th Ann. ACM Symp. Principles of Programming Languages*, ACM, 1982, pp. 142-145.

[15] T.A. Standish, Extensibility in programming language design, *SIGPLAN Notices* **10** (1975) 7, pp. 18-21.

[16] B. Wegbreit, *Studies in Extensible Programming Languages*, dissertation, Harvard, 1970 (reprinted by Garland Publishing, 1980).