A style for writing the syntactic portions of complete definitions of programming languages*

F. G. Pagan

Department of Computer Science, Southern Illinois University, Carbondale, IL 62901, USA

A strategy for the formalisation of the syntactic aspects—abstract syntax, textual (or concrete) syntax, and context conditions (or 'static semantics')—of programming languages is described. The strategy emphasises the centrality of abstract syntax in a complete language definition and is compatible with the principle of using a general purpose programming language as a metalanguage. The proposed technique for specifying (using a set of mutually recursive procedures) textual syntax and its relationship to abstract syntax provides a sort of programming language counterpart of BNF as far as overall structure is concerned, and the technique for specifying (using another set of mutually recursive procedures) context conditions parallels the use of attribute grammars or two-level grammars for the same purpose. The techniques are illustrated using the miniature language Asple.

(Received October 1979)

1. Introduction

The principle of using general purpose programming languages as metalanguages for the formal specification of programming languages (Pagan, 1976; 1979a; 1979b; 1980a) has two major potential advantages. The first stems from the fact that programming languages in general are familiar entities to all concerned. Formal specifications should be more acceptable and more widely understood by the community of language designers, implementers, and users if they are expressed in a familiar, general purpose programming language than if they are expressed in an esoteric, specialised, formal metalanguage.

The second advantage is that specifications expressed in a programming language can easily and immediately be subjected to computer aided analysis and testing—they can simply be fed to a compiler for the (meta-)language and perhaps even executed. One facet of the pervasive analogy between programming and language definition is that in both cases the 'software engineering' problem becomes more and more serious as the size and complexity of the product increase. It is, of course, extremely difficult to achieve full correctness in a large program without the aid of some debugging runs on a computer, no matter how sophisticated the methods and tools used or how thorough the proofreading. It is just as difficult to achieve total completeness and consistency in a set of language specifications that is not readily amenable to mechanical checking. Language specifications are 'metasoftware', and one can be much more confident of their validity if they have been computer tested than one ever could be otherwise.

All this presupposes that the metalinguistic use of a programming language is feasible with respect to expressive adequacy and qualities of specification such as clarity and conciseness. Evidence that such feasibility is possible in the contexts of operational semantics (Pagan, 1976) and denotational semantics (Pagan, 1979a; 1979b) has been presented elsewhere. Those studies largely disregarded the question of how syntax (context sensitive as well as context free aspects) should be defined. Of course, syntax usually presents a much less difficult problem than semantics as far as formalisation is concerned, but syntactic specifications are nevertheless an essential part of a *complete* definition of a language.

2. The syntactic aspects of formal definitions of programming languages

In the area of formal specification of programming languages, the following concepts relating to syntax are relevant: (*a*) abstract syntax, (*b*) textual or concrete syntax, (*c*) correspondence between abstract syntax and textual syntax, and (*d*) context sensitive conditions. In many approaches to formal specification, a complete language definition would include all four of these components in recognisable form.

An abstract syntax for a language highlights those structural features that are semantically relevant and forms the basis for the assignment of meaning to programs by a set of formal semantic specifications. Almost all approaches to formal semantics, whether they be categorised as operational, denotational, or axiomatic in nature, employ some sort of abstract syntax. The concept is present even in the technique of defining semantics by means of two-level grammars (Cleaveland and Uzgalis, 1977), where metalinguistic analogues of program constructs occur as elements of the grammar vocabulary. A scheme for abstract syntax must allow for the unique naming and selection of semantically relevant components of constructs. Some of the most flexible and detailed schemes, in which constructs are represented as structured, mathematical objects, are those used in the Vienna methods (Lucas *et al.*, 1968; Bjørner and Jones, 1978) for formal definition. The technique of specifying abstract syntax by means of the data structuring facilities of a modern, general purpose programming language (Pagan, 1976; 1979a) is closely analogous.

It is often useful to think of the abstract form of a program as the 'fundamental' form and the textual version as a derived form, rather than vice versa. The textual form is then regarded merely as a linear encoding of the abstract form as a character string. Among the textual properties that need not be incorporated into an abstract syntax are defaults and optional parts of constructs, parenthesisation and operator precedence conventions, and delimiter symbols.

Assuming that a language definition scheme does differentiate between abstract and textual syntax, the correspondence between the two must be formalised if the language definition is to be complete. (Many published language definitions are intentionally incomplete in this respect, sometimes even omitting the textual syntax specifications, in order to focus on techniques for the specification of semantics.) In the two-level grammar scheme, the correspondence is inherent in the grammar mechanism. In the more common schemes, there are basically two possibilities: specifications are supplied either for mapping textual programs to abstract programs or for mapping abstract programs to textual programs. The textual-to-abstract strategy is analogous to compilation and involves (usually) definition of a many-to-one transformation. Examples

*This material is based upon work supported by the National Science Foundation under Grant No. MCS-7902962.

CCC-0010-4620/81/0024-0143 $02.50

of its adoption may be found in the Semanol system (Anderson *et al.*, 1976) and the VDL definition of Asple (Marcotty *et al.*, 1976). The abstract-to-textual strategy is analogous to decompilation (e.g. prettyprinting) and involves (usually) a one-to-many transformation. An example of a definition using this strategy is the VDL definition of ALGOL 60 (Lauer, 1968). Clearly, the latter strategy is more compatible with the idea that abstract as opposed to textual syntax should be considered as fundamental. It also circumvents matters such as lexical scanning and symbol table lookup which, although of interest to the compiler writer, can add unnecessary complexity to a formal definition.

The enforcement of context conditions (noncontext free aspects of a language's syntax, also known as static semantics) has been handled in a variety of ways. One strategy (used in Lauer, 1968, for example) is to combine the specification of all such properties with the specification of (dynamic) semantics. The objections to this approach are that context conditions are more properly considered to be syntactic than semantic in nature and that the semantic component of a language definition has enough to deal with without piling on more. Another strategy is to embed the context sensitive restrictions in the specifications for the abstract-to-textual mapping or the textual-to-abstract mapping (e.g. the VDL definition of Asple (Marcotty *et al.*, 1976)). The trend in several recent language definition studies (e.g. Bjørner and Jones, 1978; Anderson *et al.*, 1976; Tennent, 1977), including some of the denotational variety, appears to be toward the specification of context conditions by means of a separate set of predicates or functions operating on abstract programs. This strategy improves the modularity of a language definition as a whole.

3. A proposed scheme

The general philosophy being advocated here is that a complete, formal definition of a programming language should consist of, first and foremost, a set of abstract syntax specifications together with (*a*) a set of semantic specifications for assigning meaning to abstract programs and (*b*) two sets of syntactic specifications for (i) mapping abstract programs to textual representations and (ii) enforcing context sensitive restrictions. Ideally, the abstract syntax and other syntactic components should be formulated so as to be independent of the type of semantics (operational, denotational, etc.) used.

Applying this philosophy to the case where a programming language is used as the metalanguage, and for the sake of concreteness taking that language to be ALGOL 68, the overall layout of a complete set of language specifications could be as follows:

```
mode prog = ...                    A
  ...;
proc progtext = ...                B
  ...;
proc wfprog = ...                  C
  ...;
proc interpret = ...               D
  ...
```

Here A defines the abstract syntax in terms of a set of mutually recursive modes as described elsewhere. D defines either the operational (Pagan, 1976) or the denotational (Pagan, 1979a; 1979b) semantics in terms of a set of mutually recursive procedures. B defines the abstract-to-textual mapping in terms of mutually recursive procedures, and C expresses the context conditions in terms of yet more mutually recursive procedures.

Given an abstract program *p* (a value of mode **prog**), the call *progtext*(*p*) yields the textual form of *p*. The meaning of *p* is defined only if *wfprog*(*p*) yields "true" and is then given by *interpret*(*p*,). (Here it is assumed that the mode of *interpret* is **proc(prog,file)file**, where the mode **file** characterises the data sets manipulable by the language being defined, so that when it is partially parametrised with *p*, it yields a routine of mode **proc(file)file** (Pagan, 1980a). Partial parametrisation is an extension of standard ALGOL 68.)

The next two sections illustrate how abstract-to-textual mappings (B) and context conditions (C) can be specified, using the miniature language Asple as an example. Asple has been the object of many previous case studies (Cleaveland and Uzgalis, 1977; Marcotty *et al.*, 1976) with which the present study may be compared. The choice of ALGOL 68 as the metalanguage is for the sake of concreteness only and is not meant to imply that this is an ideal programming language for such a role.

4. Correspondence between abstract and textual syntax

The abstract syntax of Asple may be formulated in ALGOL 68 as follows:

```
mode prog = struct (decl de, stm st),
 decl = union (declist, dec),
  declist = struct (dec de, ref decl next),
  dec = struct (type ty, ref[]id ids),
  type = struct (int refcount # >= 0 #, string primtype
    # "bool" or "int" #),
 stm = union (stmlist, cmd),
  cmd = union (asgt, cond1, cond2, loop, inp, outp),
  stmlist = struct (cmd st, ref stm next),
  asgt = struct (id dest, exp source),
  cond1 = struct (exp cond, ref stm st),
  cond2 = struct (exp cond, ref stm st1, st2),
  loop = struct (exp cond, ref stm body),
  inp = struct (id var),
  outp = struct (exp ex),
 exp = union (id, const, infix, compare),
  infix = struct (ref exp opd1, opd2, string opr # "+" or
    "*" #),
  compare = struct (ref exp comp1, comp2, string rel #
    "=" or "/=" #),
  const = union (bconst, iconst),
   bconst = bool,
   iconst = union (diglist, dig),
    diglist = struct (dig d, ref iconst next),
    dig = char # "0" . . . "9" #,
 id = string # of chars in A . . . Z #
```

It can be seen that a program contains only one level of declarations. The data type of a variable consists of one or more "ref" indicators together with a 'primitive type' of "bool" or "int".

The problem in this section is to specify a mapping from **prog** values to textual programs that conform to the following BNF grammar:

```
⟨program⟩ ::= begin ⟨dcl train⟩; ⟨stm train⟩ end
⟨dcl train⟩ ::= ⟨declaration⟩ | ⟨declaration⟩; ⟨dcl train⟩
⟨declaration⟩ ::= ⟨mode⟩ ⟨idlist⟩
⟨mode⟩ ::= bool | int | ref ⟨mode⟩
⟨idlist⟩ ::= ⟨id⟩ | ⟨id⟩, ⟨idlist⟩
⟨stm train⟩ ::= ⟨statement⟩ | ⟨statement⟩; ⟨stm train⟩
⟨statement⟩ ::= ⟨id⟩ := ⟨exp⟩ |
 if ⟨exp⟩ then ⟨stm train⟩ fi |
 if ⟨exp⟩ then ⟨stm train⟩ else ⟨stm train⟩ fi |
 while ⟨exp⟩ do ⟨stm train⟩ end | input ⟨id⟩ | output ⟨exp⟩
⟨exp⟩ ::= ⟨factor⟩ | ⟨exp⟩ + ⟨factor⟩
⟨factor⟩ ::= ⟨primary⟩ | ⟨factor⟩ * ⟨primary⟩
⟨primary⟩ ::= ⟨id⟩ | ⟨constant⟩ | (⟨exp⟩) | (⟨compare⟩)
⟨compare⟩ ::= ⟨exp⟩ = ⟨exp⟩ | ⟨exp⟩ ≠ ⟨exp⟩
⟨constant⟩ ::= ⟨bool constant⟩ | ⟨int constant⟩
⟨bool constant⟩ ::= true | false
```

⟨int constant⟩ ::= ⟨digit⟩ | ⟨int constant⟩ ⟨digit⟩
⟨digit⟩ ::= 0 | 1 | . . . | 9
⟨id⟩ ::= ⟨letter⟩ | ⟨id⟩ ⟨letter⟩
⟨letter⟩ ::= *A* | *B* | . . . | *Z*

(This grammar is given here for expository purposes only; it will not form part of the final specifications, since the information it contains will be implicit in the mapping definition.)

The first question to be settled is whether a program in its textual form is to be considered as a sequence of characters or as a sequence of tokens. With some (but not all) languages, the latter possibility could carry more information about just where spaces and line boundaries can or must be inserted in programs. Such matters are often left to the individual implementation, however, and in any case are generally not dealt with in the formal syntax (e.g. BNF). For this reason, and because a sequence of tokens is a more complicated structure, a textual program will be regarded as a simple of nonblank characters; layout rules will be assumed to be separately formalised if necessary. Thus the mode of the procedure *progtext* will be **proc(prog)string.**

As a starting point, to give the basic flavour of the technique, the following procedures define textual representations of Asple statements and expressions:

```
proc stmtext = (stm s) string:
 case s in
  (stmlist s):
   stmtext (st of s) + ";" + stmtext (next of s),
  (asgt s):
   dest of s + ":=" + exptext (source of s),
  (cond1 s):
   "if" + exptext (cond of s) + "then" +
   stmtext (st of s) + "fi",
  (cond2 s):
   "if" + exptext (cond of s) + "then" +
   stmtext (st1 of s) + "else" + stmtext (st2 of s) + "fi",
  (loop s):
   "while" + exptext (cond of s) + "do" +
   stmtext (body of s) + "end",
  (inp s):
   "input" + var of s,
  (outp s):
   "output" + exptext (ex of s)
 esac,
proc exptext = (exp e) string:
 case e in
  (id e): e,
  (const e): consttext (e),
  (infix e): "(" + exptext (opd1 of e) + opr of e +
   exptext (opd2 of e) + ")",
  (compare e): "(" + exptext (comp1 of e) + rel of e +
   exptext (comp2 of e) + ")"
 esac,
proc consttext = (const c) string:
 case c in
  (bconst c):
   (c | "true" | "false"),
  (iconst c):
   case c in
    (diglist c): consttext (d of c) + consttext (next of c),
    (dig c): c
   esac
 esac
```

Such procedures can be read in much the same way as a grammar, and are much simpler than a set of 'compilation' (textual-to-abstract) specifications would be.

As it stands, the procedure *exptext* does not completely define the textual syntax of Asple expressions. According to the BNF rules, any expression may be parenthesised, whereas the procedure employs parentheses only for enclosing infix expressions (and comparisons). Thus *exptext* defines only one of the possible texts for each expression, and not necessarily the best one at that, since the parentheses it does specify are often redundant.

The need to express the fact that any expression can be (perhaps multiply) parenthesised is a special case of the general problem posed by 'optionality' in the textual syntax of programming languages. The procedures must somehow map each abstract program on to a whole set of textual representations. A natural way of programming such multiple-valued 'functions' is to make use of a suitable form of nondeterminism. (The next best thing to use is, probably, the metalanguage's facilities for random number generation, and this may work better if the specifications are to be executed for testing purposes.) In ALGOL 68, for example, if a conformity clause contains more than one specification matching the mode of the object being tested, then it is undefined which case is chosen.

The procedure *exptext* can be refined in the following manner. (The use of **union(exp, void)** instead of just **exp** as the parameter mode is a technicality of the metalanguage.)

```
proc exptext = (union(exp, void) e) string:
 case e in
  (id e): e,
  (const e): consttext (e),
  (infix e): (
   proc sum on left = (infix e) bool:
    (opd1 of e | (infix o1): opr of o1 = "+" | false),
   proc prod on right = (infix e) bool:
    (opd2 of e | (infix o2): opr of o2 = "*" | false),
   proc simple = (exp e) bool:
    (e | (infix): false | true);
   string left = (opr of e = "*" and sum on left (e) |
    "(" + exptext (opd1 of e) + ")" | exptext (opd1 of e)),
   string right = (opr of e = "+" and prod on right (e) or
                                  simple (opd2 of e) |
    exptext (opd2 of e) | "(" + exptext (opd2 of e) + ")");
   left + opr of e + right),
  (compare e):
   "(" + exptext (comp1 of e) + rel of e +
    exptext (comp2 of e) + ")",
  (exp e): "(" + exptext (e) + ")"
 esac
```

An infix expression is now definitely parenthesised only if (*a*) its operator is '+' and either it is the left operand of a '*' or it is the right operand of any operator, or (*b*) its operator is '*' and it is the right operand of another '*'. However, *any* expression may be *optionally* parenthesised (to any depth, because of the recursion). Now the procedure completely and accurately characterises the textual syntax of Asple expressions together with its relationship to the abstract syntax. The set of textual representations of an abstract expression *e* is defined to be the set of all possible values of *exptext*(*e*).

It remains to define the textual form of declarations and complete programs. The procedure *typetext* specifies that the first "ref" of a mode does not appear in the textual form:

```
proc typetext = (type t) string:
 if refcount of t <= 1
 then primtype of t
 else "ref" + typetext (type (refcount of t − 1, primtype of t))
 fi
```

Thus in the case of the program

```
begin
 int I, J;
 ref int P; ref ref bool B;
 . . .
end
```

the relevant types are (1, "int") for *I* and *J*, (2, "int") for *P*, and (3, "bool") for *B*.

The ordering of the declarations in a program is arbitrary. Two of the alternative orderings for the declarations in the above example are

int I; *int J*; *ref ref bool B*; *ref int P*
ref ref bool B; *int J*; *ref int P*; *int I*

In this paper, it is assumed for simplicity that all these possibilities correspond to distinct (though semantically equivalent) abstract programs. A more elaborate scheme whereby each abstract program in such an equivalence class has all permutations of declarations in its set of textual representations is given elsewhere (Pagan, 1980b). The remaining procedures for the abstract-to-textual mapping are thus quite straightforward:

```
proc decltext = (decl d) string:
 case d in
  (declist d):
   decltext (de of d) + ";" + decltext (next of d),
  (dec d): (
   string text := typetext (ty of d) + (ids of d)[1];
   for i from 2 to upb ids of d do
    text := text + "," + (ids of d)[i] od;
   text)
 esac,

proc progtext = (prog p) string:
 "begin" + decltext (de of p) + ";" + stmtext (st of p) +
 "end"
```

The specification of Asple's textual syntax and its relationship to the abstract syntax is now complete. The set of textual representations of an abstract program *p* is defined to be the set of all possible values of *progtext(p)*.

5. Context conditions

Not all values of mode **prog** are valid programs; that is to say, the syntax of Asple has various context sensitive aspects. The major context conditions may be informally stated as follows:

1. All identifiers used in a program must be declared.
2. No identifier may be declared more than once.
3. The two sides of an assignment statement must have the same primitive type and the number of "refs" in the left type must not exceed the number of "refs" in the right type by more than one.
4. The primitive type of the expression in a conditional or loop statement must be "bool".
5. The primitive types of the operands of an infix expression must be the same.
6. The primitive types of the operands of a comparison must both be "int".

The proposed technique for formalising such context conditions is similar to that used in the Vienna Development Method (Bjørner and Jones, 1978), except of course that here the metalanguage is a programming language. The conditions are characterised by a set of mutually recursive predicate routines (procedures returning boolean values) together with some additional procedures needed by the predicate routines. The various procedures need to manipulate new types of structured values which are closely analogous to the attributes of an attribute grammar or the metanotions of a two-level grammar in a definition of the context conditions expressed in either of these formalisms. In the case of Asple, the only additional modes needed are

```
mode item = struct (type ty, id id, table rest),
 table = ref item
```

A **table** value, which is a linked list of nodes of mode **item**, will serve to record the name and type information corresponding to a set of declarations. The 'well-formedness' (adherence to context conditions) of many of the constructs in an Asple program can only be defined relative to a table containing the name and type information corresponding to all the program's declarations. The following procedure augments a table *tbl* with the information specified by a declaration *d*:

```
proc newtable = (table tbl, decl d) table:
 case d in
  (declist d):
   newtable (newtable (tbl, de of d), next of d),
  (dec d): (
   table t := tbl;
   for i to upb ids of d do
    t := heap item := (ty of d, (ids of d)[i], t) od;
   t)
 esac
```

The procedure *consistent* enforces the restriction against multiply-declared identifiers:

```
proc consistent = (table tbl) bool:
 if tbl is nil then true
 elif table (rest of tbl) is nil then true
 else table p := rest of tbl, bool con := true;
  while table (p) isnt nil do
   if id of tbl = id of p then con := false fi;
   p := rest of p od;
  con and consistent (rest of tbl)
 fi
```

The well-formedness of a complete program is then defined by

```
proc wfprog = (prog p) bool: (
 table tbl = newtable (nil, de of p);
 consistent (tbl) and wfdecl (de of p) and wfstm (st of p, tbl))
```

(In an attribute grammar, "table" would be a synthesised attribute of ⟨dcl train⟩ and an inherited attribute of ⟨statement⟩; the function for evaluating it would be the counterpart of the procedure *newtable* here.)

The predicate for declaration well-formedness is very simple:

```
proc wfdecl = (decl d) bool:
 case d in
  (declist d): wfdecl (de of d) and wfdecl (next of d),
  (dec d): refcount of ty of d >= 1
 esac
```

In order to define the predicates for statements and expressions, auxiliary procedures are needed for determining the types of variables and expressions and checking that identifiers have been declared:

```
proc vartype = (id i, table tbl) type:
 if tbl isnt nil then
  if i = id of tbl then ty of tbl
  else vartype (i, rest of tbl)
  fi
 fi,

proc exptype = (exp e, table tbl) type:
 case e in
  (id e): vartype (e, tbl),
  (const e): (e | (bconst): (0, "bool"),
                  (iconst): (0, "int")),
  (infix e): (0, ptype (opd1 of e, tbl)),
  (compare e): (0, "bool")
 esac,

proc ptype = (exp e, table tbl) string:
 primtype of exptype (e, tbl),
```

```
proc declared = (id i, table tbl) bool:
 if tbl is nil then false
 elif i = id of tbl then true
 else declared (i, rest of tbl)
 fi
```

The procedures *wfstm* and *wfexp* may now be written as follows:

```
proc wfstm = (stm s, table tbl) bool:
 case s in
  (stmlist s):
   wfstm (st of s, tbl) and wfstm (next of s, tbl),
  (asgt s): (
   type td = vartype (dest of s, tbl),
        ts = exptype (source of s, tbl);
   declared (dest of s, tbl) and wfexp (source of s, tbl) and
    refcount of td - 1 <= refcount of ts),
  (cond1 s):
   wfexp (cond of s, tbl) and wfstm (st of s, tbl) and
    ptype (cond of s, tbl) = "bool",
  (cond2 s):
   wfexp (cond of s, tbl) and wfstm (st1 of s, tbl) and
    wfstm (st2 of s, tbl) and
    ptype (cond of s, tbl) = "bool",
  (loop s):
   wfexp (cond of s, tbl) and wfstm (body of s, tbl) and
    ptype (cond of s, tbl) = "bool"
  (inp s):
   declared (var of s, tbl),
  (outp s):
   wfexp (ex of s, tbl)
 esac,
proc wfexp = (exp e, table tbl) bool:
 case e in
  (id e): declared (e, tbl),
  (const): true,
  (infix e):
   wfexp (opd1 of e, tbl) and wfexp (opd2 of e, tbl) and
    ptype (opd1 of e, tbl) = ptype (opd2 of e, tbl),
  (compare e):
   wfexp (comp1 of e, tbl) and wfexp (comp2 of e, tbl) and
    ptype (comp1 of e, tbl) = "int" and
    ptype (comp2 of e, tbl) = "int"
 esac
```

6. Conclusion

Additional examples of the use of the techniques illustrated in this paper are given in Pagan (1980b). The techniques implement a comprehensive strategy for syntactic specification which emphasises the centrality of abstract syntax in a complete language definition, is largely independent of the approach used to define semantics, and is compatible with the principle of using a general purpose programming language as a metalanguage. Roughly speaking, the technique for specifying textual syntax and its relationship to abstract syntax constitutes a programming language counterpart of BNF grammars as far as overall structure is concerned, while the technique for specifying context conditions parallels the use of attribute grammars or two-level grammars for the same purpose. The examples suggest that a satisfactory degree of clarity and naturalness of the specifications can be achieved.

References

ANDERSON, E. R., BELZ, F. C. and BLUM, E. K. (1976). SEMANOL (73) A Metalanguage for Programming the Semantics of Programming Languages, *Acta Informatica*, Vol. 6, pp. 109-131.

BJØRNER, D. and JONES, C. B. (eds.) (1978). *The Vienna Development Method: The Meta-Language*, Springer-Verlag Lecture Notes in Computer Science, No. 61.

CLEAVELAND, J. C. and UZGALIS, R. C. (1977). *Grammars for Programming Languages*, Elsevier North-Holland, New York.

LAUER, P. (1968). Formal Definition of ALGOL 60, TR 25.088, IBM Laboratory Vienna.

LUCAS, P., LAUER, P. and STIGLEITNER, H. (1968). Method and Notation for the Formal Definition of Programming Languages, TR.087, IBM Laboratory Vienna, 1968 and 1970.

MARCOTTY, M., LEDGARD, H. F. and BOCHMANN, G. V. (1976). A Sampler of Formal Definitions, *Computing Surveys*, Vol. 8, pp. 191-276.

PAGAN, F. G. (1976). On Interpreter-Oriented Definitions of Programming Languages, *The Computer Journal*, Vol. 19, pp. 151-155.

PAGAN, F. G. (1979a). ALGOL 68 as a Metalanguage for Denotational Semantics, *The Computer Journal*, Vol. 22, pp. 63-66.

PAGAN, F. G. (1979b). Studies in the Metalinguistic Use of a General-Purpose Programming Language for the Specification of Denotational Semantics, Technical Report 79-01, Dept. of Computer Science, Southern Illinois Univ. at Carbondale.

PAGAN, F. G. (1980a). On the Generation of Compilers from Language Definitions, *Inf. Process. Lett.*, Vol. 10, pp. 104-107.

PAGAN, F. G. (1980b). Programming Languages as Metalanguages: A Style for Specification of Syntax and Context Conditions, Technical Report 80-81, Dept. of Computer Science, Southern Illinois Univ. at Carbondale.

TENNENT, R. D. (1977). A Denotational Definition of the Programming Language PASCAL, Technical Report 77-47, Dept. of Computing and Information Science, Queen's Univ., Kingston, Ontario.