A survey of adaptable grammars

Henning Christiansen
Roskilde University Centre
P.O. Box 260, DK-4000 Roskilde, Denmark
E-mail: henning@dat.ruc.dk

This paper is a comment on two recent contributions to Sigplan Notices.

In his paper, "The static semantics file", no. 25/4, Brian Meek discusses the relevance of the notion of "static semantics". The relation between a variable's declaration and the restrictions on its use, for example, is usually classified as static semantics. Meek finds the designation rather misleading since it is applied for concepts concerned with context-dependent syntax. The term "semantics" should properly only be used for aspects that have to do with real meaning, e.g., the association between program statements and their intended computation. Here I will show that this distinction between syntax and semantics can be made clearer using grammars which adapt themselves to the current program contexts. For example, declarations of new items can be described by adding new rules to the grammar and thus, within a given scope of a program, the set of valid phrases can be derived *freely* by means of the current set of grammar rules. This way, we get rid of some of those — often quite complicated — context constraints that are called static semantics.

In no. 25/5, Boris Buhrsteyn presents an article, "On the modification of the formal grammar at parse time". The author suggests an approach to language recognition in which declarations of, say, variables result in an adaptation of the grammar as outlined above and in turn in an adjustment of the parsing tables. The idea can be traced back to the early sixties, and over the years several proposals for adaptable grammar formalisms have been suggested. In the following, I complement Buhrsteyn's article giving an overview of the area and describe the advantages and disadvantages of describing programming language syntax this way.

Section 1 below describes the principle of adaptable grammars and how it can be used to characterize various context-dependent, syntactic aspects. The notation used will be my own generative clause grammars which are adaptable versions of Prolog's definite clause grammars. The formal definition is given in appendix A, appendix B shows some examples. Section 2 provides a short historic survey of approaches to extensible or adaptable grammar formalisms. The more problematic issues of the approach and how they have been solved or not solved in various formalisms are discussed in section 3. Visibility rules and recursive declarations represent syntactic aspects that are difficult to handle this way. Chapter 4 provides for a summary and a discussion of the distinction between syntax and semantics.

1 The principle

When describing the full syntax of a programming language, we have to take care of context-dependent issues introduced, among other things, by declarations. In a traditional grammar — an attribute grammar (Knuth, 1968), for example — a fixed set of (in principle) context-free rules are kept in check by additional context conditions. The general rule describing use of variables, for example,

variable $\rightarrow$ identifier

will typically have an associated context condition which states that the actually occurring identifier has to appear in a certain symbol table object. The designation of "static semantics" is often applied for such context conditions.[1]

These context conditions tend to be rather complicated. For simple variables, it works quite well, but the more sophisticated declaration mechanisms, the more inane grammar rules and complicated context conditions. This is indicated very clearly in the published attribute grammar for Ada (Uhl et al., 1982) by the two-and-a-half page context condition associated with the grammar rule for general procedure and function call (rule r_084). We can illustrate the phenomenon by a small experiment. Consider a language in which arbitrary new constructs can be declared in any syntactic category. In order to be prepared for all possibilities, an attribute grammar must, then, contain a rule of the following kind.

phrase $\rightarrow$ character*

The associated context — or "static semantic" — condition will, then, be where all real information about the language syntax must be encoded.

Instead we may suggest (as also Buhrsteyn, 1990, di Forini, 1963, just to mention a few) that we

- abolish the over-general rules with their context conditions,
- introduce mechanisms to add individual grammar rules for each declared notion.

[1]From where or from whom this designation originates is not clear to me. Perhaps it is due to a shallow interpretation of the title of Knuth's (1968) paper which introduced attribute grammars?

For example, for a variable, x, of type integer, the following rule should be added to the current grammar.

variable(integer) $\rightarrow$ [x]

The notation used here is similar to Prolog's definite clause grammars (Colmerauer, 1975, Pereira, Warren, 1980), terminal symbols are indicated by square brackets, the atomic value "integer" is an attribute of the nonterminal "variable".

Following this principle, a procedure declaration should give rise to a grammar rule describing calls of that particular procedure. For example, calls of a Pascal procedure, p, with a variable parameter of type real and a value parameter of type integer is described as follows.

statement $\rightarrow$ [p, '('], variable(real), [','], expression(integer), [')']

This as opposed to having a set of rules describing general procedure calls — with any possible name and any possible kind of parameter list — and associated attribute definitions which collect the information about the particular call and check its consistency with the symbol table.

In order to turn the idea into a formalism, two things must be provided,

– a method for synthesis of new grammar rules, and

– a mechanism for controlling their scope.

As an example of an adaptable grammar formalism, I will explain my own generative clause grammars, GCG, which are extensions of the well-known definite clause grammars (Colmerauer, 1975, Pereira, Warren, 1980).

Controlling grammatical scope

The syntactic derivation relation for GCG associates with each instance of a nonterminal a grammar which is called its *grammatical context*. This grammar determines the sentential forms which can be derived in one step from that nonterminal. In a GCG rule, a nonterminal can have a *context part* which gives expression to its grammatical context. Consider as an example the following GCG rule for a non-recursive block construct.

```
statement in *context →
        [declare], declarations(*news),
        [begin],
                statements in *news & *context,
        [end].
```

The asterisks indicate grammar variables, the "in" symbol connects nonterminals with their context parts.

Consider, now, a nonterminal, "statement" whose grammatical context, G, includes this rule. When the rule is applied to expand the nonterminal, the variable *context will refer to G. The declaration part is expected to generate an additional set of grammar rules, N, which henceforth is bound to the variable, *news. (The omitted context part for the declarations defaults to the current, i.e., *context). The grammatical context for the body statements is then the value of the expression, *news & *context, which is the union of N and G. A formal definition is given in appendix A.

Synthesizing new grammar rules

Buhrsteyn (1990) shows a notation based on Yacc's, (Johnson, 1975). The GCG formalism inherits its notation from its underlying logic programming framework. Consider as an example the following rule for variable declarations.

```
declaration(*use-var) →
        [var], identifier(*id), [:], type(*type)
where
        *use-var = (variable(*type) → [*id]).
```

In an application of this rule, the variables *id and *type will be instantiated to suitable terms which, then, serve as parts of the generated grammar rule. For example, in the derivation of the declaration,

```
var x: integer,
```

the grammar variable *id is assigned the value "x" and *type the value "integer". (How we distinguish between such meta-variables and possible new variables needed in the new rule is explained later). The *where* notation makes it possible to extract textually large attribute expressions and place them below the grammar rule.

The effect of the adaptable principle

The basic idea is that grammatical context is made an explicit data object open for modification, e.g., by extension with new rules. This way, we may write grammars which extract the contents of declarations, arrange it into new grammar rules which become valid in the scope of the declarations. Hence there is no need to attach any additional context restrictions to the grammar rule. In the body of a block everything is generated freely, perhaps supplied by yet other, locally generated grammar rules.

Above, the principle was applied to simple variables. We can continue with types. The declaration of a record type for, say, complex numbers, generates the grammar rules which so to speak brings the type into existence.

type(complex) $\rightarrow$ [complex].

variable(real) $\rightarrow$ variable(complex), ['.', re].

variable(real) $\rightarrow$ variable(complex), ['.', im].

The first rule giving the type identifier makes it possible to declare variables of complex type, the two latter for accessing the fields of such variables. The last rule, for example, states that anywhere (in its scope, of course) that a real number variable is expected, a complex number variable followed by "dot-im" is allowed.

If, furthermore, the language at hand allows the declaration of new operators, a declaration of an operator for complex multiplication could give rise the following rule.

expression(complex) $\rightarrow$ expression(complex), [*], expression(complex).

The rules for the declarations which generate these rules are found in appendix B.

As mentioned above, the principle of adaptable grammars reduces the need for the often quite heavy context conditions which — under the name of static semantics — otherwise is a burden on syntax descriptions for programming languages. There is also a conceptual advantage in the approach which seems more natural with respect to the way we *use* declarations in programming languages. Whenever a programmer has got used to a program package for complex arithmetic he can think of complex numbers as being an integral part of the programming language — equal to integers and booleans.

The idea looks captivating when applied to declarations as shown, however, there are other context-dependent issues of programming language syntax which are more problematic to express this way. We return to this in section 3.

2 Approaches to adaptable syntax

The following survey is based on my personal search for work related to my own.

The first to describe the idea of extending the grammar in order to model declarations seems to be A.C. di Forini (1963). The approach is motivated by the (at the time) lack of a formal definition method for the full syntax of programming languages as well as by interest in conceptual aspects. To support the viewpoint, di Forini gives citations from the original Algol 60 report (Naur, ed., 1960), e.g., "... [declarations] automatically introduce a new level of nomenclature". The paper gives no suggestion for a grammatical formalism or a notation, but it is very clearly written and may still provide valuable insight to the present-day reader.

In 1970, B. Wegbreit introduced a class of extensible context-free grammars, ECF's, in order to describe his extensible programming system ECL, (Wegbreit, 1970). An ECF consists of a context-free grammar and an associated finite state machine. The finite state machine analyzes the source text in parallel with the normal parsing, and whenever a pattern has been recognized as a grammar rule (which have to be enclosed in special brackets), it can be either added to or deleted from the grammar. Whether addition or deletion meant, is indicated using either an arrow or a crossed out arrow between the left and right hand sides of the grammar rule in the source text. However, the applications of ECF are rather limited; I will come back to this in section 3.

Hanford and Jones (1973) have suggested a concept of dynamic syntax which is a monstrous device based on the λ-calculus. Also here, the current set of valid rules is intended to be a data object open for modification. Unfortunately the approach is not given a proper formalization or accompanied with an appropriate notation. Therefore it is difficult to compare it with other approaches and the authors do not seem to have continued with the idea.

Yet another approach is Mason's (1984, 1987) dynamic template translators, DDT's, which can be seen as a generalization of Wegbreit's approach. A

DDT is basically a syntax-directed translation scheme which may have side-effects on itself. The language determined by a DDT is defined operationally in terms of an extended shift-reduce algorithm.

The first of my own published papers on these matters, (Christiansen, 1985), introduces an extension of attribute grammars which in their essence are equal to the generative clause grammars described here. A distinguished attribute is used for passing grammar objects around and the derivation relation is defined relative to these attributes. Furthermore, it is shown that these grammars also can express similarly adaptable semantics. The report (Christiansen, 1988) gives a more thorough treatment of these grammars. Buhrsteyn's recent approach, (Buhrsteyn, 1990) seems closely related. His modifiable grammars are also extensions of attribute grammars such that the general attribute mechanism can be used to extract the contents of declarations and put it together to form new rules. However, with respect to modifying the grammar, the approach is more related to Wegbreit's and Mason's, the syntax analyzer can add new rules to the grammar as a side-effect while scanning the program from left to right. This as opposed to our functional style of grammar adaptation.

In (Christiansen, 1986), the problems involved in adapting traditional parsing methods to languages defined by such grammars are discussed. However, no general solutions are given, the main problem seems to be that parsing tables are global properties of a grammar. This means that adding a single rule to a grammar may lead to a re-construction of the entire parsing table, or worse, that the LL(1) or LALR(1) or whatever property is relevant, is destroyed. However, it is worth mentioning here the recent work by Heering, Klint, and Rekers (1989) on incremental generation of parsing tables which looks quite promising. It will also be interesting to see the methods applied in the announced, forthcoming papers about Buhrsteyn's translator generator.

The recent work on generative clause grammars described in this paper has lead to an implementation in Prolog and a notational improvement compared with my earlier work referenced above. The implementation is based on a straightforward generalization of the implementation method used for definite clause grammars (see, e.g., Clocksin, Mellish, 1987). Here there is a one-to-one correspondence between a grammar rule and its compilation into a Prolog clause such that each rule can be compiled one at a time.[2] The price

[2]Some additional techniques are needed in order to represent the resulting hierarchies of programs in a common program space, see (Christiansen, 1990).

to be paid, on the other hand, is a large amount of backtracking and a potentiality for infinite loops. Interested readers are welcome to write to the present author for a copy of source text of the GCG implementation.

Since the very beginning of computer science there have been an interest in extensible languages and various sorts of generators of language processors. In this survey, I have concentrated on grammatical formalisms based on extensibility and generation of new rules. Readers interested in the mentioned, related subjects are referred to the existing bibliographies and surveys on these matters. Just to mention a few, I can refer to the following. Extensible languages: Solntseff, Yezerski (1974), Layzell (1985). Generator systems based on attribute grammars: Deransart, Jourdan, Lorho (1988).

Finally, it should be made very clear that adaptable grammars cannot describe any language which cannot be described by a conventional grammar. It is possible to write down a definite clause grammar which is able to simulate any generative clause grammar. It is based on a context-free grammar which can derive any string, generative clause grammars are passed around as attributes and the GCG derivation relation is encoded as a context condition.

Similar constructions are possible in other kinds of grammars, e.g., attribute grammars (Knuth, 1968) or two-level grammars (van Wijngaarden et al., 1975). Actually, the three mentioned "traditional" types of grammars are equivalent. Deransart, Maluszynski (1985) showed that definite clause grammars and attribute grammars can be considered notational variations over the same thing; Dembinski and Maluszynski (1978) has described the transformation of attribute grammars into two-level grammars that generate the same languages and vice versa.

3 Difficulties in the adaptable grammar approach

We have seen that some contextual issues in programming language syntax can be handled quite nicely by adaptable grammars. There are other, more problematic aspects which I will discuss in the following.

Removing rules at block exit

It seems quite obvious that a grammar formalism for programming languages possess a natural way for describing locality of declarations in block structured languages. However, this is not true for several proposals for adaptable grammars.

In Wegbreit's ECF's and the suggestion of Buhrsteyn, the grammar is a state which develops while the source program is scanned from left to right. Rules generated from declarations can be added during this process but neither of the systems has a way of marking rules for automatic removal when the block's final "end" is reached. It should be noted, however, that the ECF's are developed for describing one particular system and are not intended as a general descriptive tool.

Mason's DDT's are similar in their treatment of the grammar as a state. Here, translation rules may have side-effects on the current DDT. The rules to be removed can be encoded in a rule describing the derivation of the block's "end"; i.e., a rule which expands nonterminal END to terminal "end" and whose side-effect is to take out the rules. It is a quite tricky construction — and it is even more trickier to arrange these clean-up rules in order to avoid confusion in the case of nested blocks.

The context-dependencies around nested blocks follow the phrase structure and not the sequential ordering of the program text. Hence the grammar flow in an adaptable formalism should better follow the phrase structure as it is done in GCG, the context parts attached with each nonterminal can deal out individual grammars to each sub-phrase.

Delayed or indirect declarations

A "with" statement is an example of a construct which gives access to entities declared somewhere else in the program text. The declaration of a record type for, say, complex numbers, should in an adaptable grammar context give rise to a grammar rule describing the existence of a particular kind of "with" statement.

```
statement in *context $\rightarrow$
    [with], variable(complex), [do],
        statement in [*select$_1$, *select$_2$] & *context,
where
    *select$_1$ = (variable(real) $\rightarrow$ [re]),
    *select$_2$ = (variable(real) $\rightarrow$ [im]).
```

The problem is the notation to be used in the rule for the type declaration. Here we will have variables referring to information extracted from the declaration text as well as indications of variables to serve as such in the generated rule. If, furthermore, the rules for field selection needed variables of their own, yet another level would have been introduced.

Such problems are well-known and well-understood in the field of metaprogramming in logic (e.g., Bowen, Kowalski, 1982, Hill, Lloyd, 1988, Christiansen, 1990). It appears that a distinction between the different levels of variables is needed, otherwise the soundness of the formalism breaks down (Bowen, Kowalski, 1982).

GCG (and earlier work, Christiansen, 1985) is the only approach to adaptive grammars which to my knowledge has addressed the description of indirect declaration mechanisms. The notation used in GCG is inspired by the referenced work in logic programming. It is given in definition 2 of appendix A and exemplified in the grammar rule for record declarations in appendix B. Module declarations and generics as in Ada can be described in a similar way.

Recursive declarations

No known, adaptable grammars provide a satisfactory treatment of recursive declarations. Intuitively, the following GCG rule for a block with recursive declarations looks quite reasonable.

```
statement in *context →
    [declare, recursively],
        declarations(*news) in *news & *context,
    [begin],
        statements in *news & *context,
    [end].
```

However, this rule specifies much more than we want in that *news according to the formal definition can contain strange creatures capable of generating themselves.

In the implemented version of GCG, recursion is handled by a little hack in the shape of a multi-pass operator which makes it possible to analyze the same part of the input twice with two different grammars. In the first pass, we use a grammar extension which accepts as a statement anything which looks like a procedure call and so forth for other categories of declarations.

GCG's provide thus no satisfactory solution; none of the other, referenced approaches to adaptable grammars have tried to cope with recursion.

Visibility

In most languages, the declaration of a variable, x, disables any other variable declared in a surrounding scope with that name. Similarly, if a variable name coincides with the field of a record type or an element of a user-defined scalar type as in Pascal, some adjustment is needed. In general, we talk about the visibility of a declaration and each language has its rules of visibility. The design of visibility rules for real programming languages is very difficult and the rules are often quite complicated. Here we may refer to the problems in the design of the Pascal language (cf. Welsh, Sneeringer, Hoare, 1977) or to the visibility rules of Ada (U.S. Dept. Defence, 1983).

In an attribute grammar, information about declared constructs is stored in a symbol table. A symbol table is a structure, and the visibility rules can be implemented in the functions which access this structure. In the adaptable grammar approach, we made an effort to get rid of the symbol table in favour of a more homogeneous representation. Here the declared concepts lie in a collection — in principle something as unstructured as a set — of grammar rules. This implies that we have no counterpart to the symbol table access functions, disabling certain grammar rules must be done as set operations. Of course it is possible to specify such operations, but there is a real hard problem in devising the proper, descriptive notation.

Neither GCG nor any other adaptive grammars are sufficient at this point. In the implemented version of GCG, a few Prolog hacks can compensate a little. The readers familiar with Prolog will recognize how the following grammar rule disables any previous grammar rule concerned with variables named x.

$$\text{variable}(*\text{type}) \to [\text{x}],\ !,\ \{*\text{type} = \text{integer}\}$$

However, it should be made clear that such tricks by no means constitute a general method for expressing visibility rules.

Currently, our group is working on a GCG for Ada which should represent a reasonable test case concerned with visibility rules.

Preventing multiple declarations

In most languages, the following sequence of declarations is illegal.

```
integer x;
real x;
```

We may cite the Revised Algol 60 Report (Naur, ed., 1963): “No identifier may be declared more than once in any one block head”. The idea in the adaptable grammar approach is that a piece of program text, e.g., the one shown above, can be generated freely from the current set of grammar rules, which, however, may change for different sub-phrases. Taking the full consequence of the adaptable principle amounts to having one grammar rule for each possible variable name — and arrange the grammar rules for declarations such that the first declaration will result in a removal of the rule for the name x. And some additional fiddling for making x available again inside procedure declarations.

The only feasible solution here seems to go back to the traditional way, over-general grammar rules with additional context constraints (sic!). The rule for blocks should, then, include an overall judgement of the grammar synthesized from the declarations.

Error handling

A compiler designed on the background of an attribute grammar provides explanatory error messages such as the following.

```
Undeclared variable:  X
```

The offending token matches the context-free part of a grammar rule and the error is exposed by the context constraint. In an adaptable grammar of the sort discussed in the present paper, there is no rule at all to match the offending `X`, so the only possible message is something of the following sort.

```
Misplaced token:  X
```

Or as my Prolog implementation of GCG arrogantly states it: “`no`”.

4 Summary and some final remarks

The idea of programming language grammars able to adapt themselves to different scopes of a program text has been around for nearly thirty years. The principle is intuitively appealing, each declared concept in a program is captured by individually generated grammar rules and is thus treated similarly to any predefined concept in the language. In a given scope of the program the legal phrases are generated freely from the current set of grammar rules. Hence we eliminate some of those context constraints which otherwise are referred to as static semantics. The processing of declarations, on the other hand, filtering out their declarative contents, is similar to how it is done in an attribute grammar.

However, not all aspects of programming language syntax are described easily this way. Some suggested formalisms cannot even handle usually nested scopes. Recursive declarations and visibility rules are difficult to handle and no proposed, adaptable grammar formalism can do it in a sufficient way.[3] As a basis for developing production-quality compilers, there are also problems concerned with the quality of error messages and in the incremental construction of parsing tables.

For theoretical studies of programming languages and their use the approach seems quite feasible. Systems like the one reported to be under development by Buhrsteyn or my GCG implementation in Prolog, may serve as powerful research and prototyping environments. In such cases we often prefer to ignore difficult items such as visibility rules.

One of my motivations for writing this paper was a dispute about the relevance of the notion of "static semantics", (Meek, 1990). I will conclude by showing another phenomenon which might be called "dynamic syntax". Consider the following Lisp program.

```
(progn (if (read) (defun f () nil) nil)
       (f))
```

Whether the sub-phrase "(f)" is legal or not depends on the execution of the program. I.e., the context-dependent syntax is given by the dynamic semantics. Where does syntax end and semantics begin? One view might be that the meaning of a declaration, its semantics, is the creation of new syntactic

[3] On the other hand, these things are also difficult when using attribute grammars!

and semantic potentiality. Syntax is concerned with the forms in the language, whether they be defined in the reference manual or in the declarations in a program text. The semantics associated with program statements has to do with the intended computation. And this seems also to be the inherent viewpoint in the various approaches to adaptable grammars.

Appendix A. Formal definition of GCG's

Here we give formal definitions of generative clause grammars and their syntactic derivation relation. The *where*-notation, the omission of "obvious" context parts, and the ampersand function symbol are syntactic sugar not considered in the definition below. The usual Prolog concepts of *constants*, *variables*, *functors*, *terms*, and *ground terms* will be assumed (see, e.g., Clocksin, Mellish, 1987).

Definition 1. A *contexted nonterminal* is a term of the form

non in *context*.

The subterms, *non* and *context*, are called *nonterminal* and *context parts*, respectively.

A *generative clause grammar rule* is a term of the form

$cn \rightarrow item_1, \ldots, item_n$

where *cn* is a contexted nonterminal. An *item* is either

- a contexted nonterminal,
- a list of *terminal symbols*, $[\mathrm{a}_1, \ldots, \mathrm{a}_m]$, or
- *embedded code* which is a term enclosed in curly brackets, $\{\ldots\}$.

A *generative clause grammar* is a list of generative clause grammar rules. □

The examples in this paper do not show examples of embedded code. Embedded code makes it possible to apply the full computational power of the Prolog language for evaluating attributes or grammatical contexts (!); the notation for and the meaning of embedded code is the same as for definite clause grammars (Clocksin, Mellish, 1987). The view of grammars as lists of rules — as opposed to sets — simplifies the representation of grammars within themselves.

The binding time for a variable symbol — or its level of generation — is indicated by the number of prefixing asterisks. A variable in a rule is denoted, say, *x, whereas a piece of text which stands for a "future" variable in a rule being generated appears, e.g., as **x. An example is shown in appendix B. The precise meaning is defined as follows; we assume the existence of a bijective function, $\mathcal{V}$, from constants to variables.

Definition 2 The *syntactic denotation function* is the partial function, $\mathcal{D}$, from the set of ground terms to the set of terms defined inductively as follows.

$$\mathcal{D}[\![\, c]\!] = c, \quad \text{for any constant, } c,$$

$$\mathcal{D}[\![\, {*c}]\!] = \mathcal{V}[\![c]\!], \quad \text{for any constant, } c,$$

$$\mathcal{D}[\![\, {*^n c}]\!] = {*^{n-1}}c, \quad \text{for any constant, } c, \text{ and } n > 1,$$

$$\mathcal{D}[\![\, f(t_1, \ldots, t_n)]\!] = f(\mathcal{D}[\![\, t_1]\!], \ldots, \mathcal{D}[\![\, t_n]\!]), \quad f \text{ different from unary } *.$$

Whenever $\mathcal{D}[\![\, t]\!] = t'$, we say that *t denotes t′*. □

The syntactic derivation relation is defined as follows. The dot operator, "●", denotes construction and concatenation of strings of grammar symbols, ϵ is the empty string. We assume the presence of a global Prolog program, BASIS, giving the meaning of predicates applied in embedded code; "$\vdash$" represents the provability relation.

Definition 3 The syntactic derivation relation, $\Rightarrow$, is defined as follows.

- Whenever

  non in *context*,

  is a contexted nonterminal such that *context* denotes a grammar, and this grammar includes a rule which has an instance,

  non in *context* $\rightarrow$ $item_1, \ldots, item_n$,

  then

  non in *context* $\Rightarrow$ $item_1 \bullet \ldots \bullet item_n$,

- $[\mathrm{a}_1, \ldots, \mathrm{a}_m] \Rightarrow \mathrm{a}_1 \bullet \ldots \bullet \mathrm{a}_m$,

- $\{term\} \Rightarrow \epsilon$ if and only if BASIS $\vdash$ *term*.

The reflexive transitive closure of $\Rightarrow$ is denoted $\Rightarrow^*$. □

Appendix B. Examples

Loops with tagged exit statement.

```
statement in *context $\rightarrow$
    label(*id), [:],
    [loop],
        statements in [*exit] & *context,
    [end, loop]

*where* *exit = (statement $\rightarrow$ [exit, *id]).
```

Declaration of infix operators. New rules are generated for applying the declared operator, for accessing formal parameters inside the function body, and for function exit.

```
declaration(*fn-call) in *context $\rightarrow$
    [function], formal(*par$_1$, *par-type$_1$), operator(*op),
        formal(*par$_2$, *par-type$_2$), [:], type(*fn-type), [;],
    statement in [*access-formal$_1$, *access-formal$_2$, *return]
                                                    & *context
*where*
    *fn-call = (expression(*fn-type) $\rightarrow$
                 expression(*par-type$_1$), [*op], expression(*par-type$_2$)),
    *access-formal$_1$ = (expression(*par-type$_1$) $\rightarrow$ [*par$_1$]),
    *access-formal$_2$ = (expression(*par-type$_2$) $\rightarrow$ [*par$_2$]),
    *return = (statement $\rightarrow$ [return], expression(*fn-type)).
```

Declaration of record types. For simplicity we assume that records always have exactly two fields.

```
declaration([*type-rule, *full-select$_1$, *full-select$_2$, *with-rule]) $\rightarrow$
      [type], identifier(*new-type), [=],
      [record],
            identifier(*f$_1$), [:], type(*t$_1$), [;],
            identifier(*f$_2$), [:], type(*t$_2$),
      [end]
where
      *type-rule = (type(*new-type) $\rightarrow$ [*new-type]),
      *full-select$_1$ = (variable(*t$_1$) $\rightarrow$ variable(*new-type), ['.', *f$_1$])
      *full-select$_2$ = (variable(*t$_2$) $\rightarrow$ variable(*new-type), ['.', *f$_2$])
      *with-rule =
            (statement in **context $\rightarrow$
                  [with], variable(*new-type), [do],
                        statement in [**select$_1$, **select$_2$] & **context,
             where
                  **select$_1$ = (variable(*t$_1$) $\rightarrow$ [*f$_1$]),
                  **select$_2$ = (variable(*t$_2$) $\rightarrow$ [*f$_2$])).
```

References

Bowen, K.A. and Kowalski, R.A., Amalgamating language and meta-language in logic programming. *Logic Programming*, Clark, K.L. and TŁrnlund, S.., eds., pp. 153–172, Academic Press, 1982.

Buhrsteyn, B., On the modification of the formal grammar at parse time. *Sigplan Notices*, Vol. 25, No. 5, pp. 117–123, 1990.

Christiansen, H., Syntax, semantics, and implementation strategies for programming languages with powerful abstraction mechanisms. *Proc. 18th Hawaii International Conference on System Sciences*, vol. 2, pp. 57–66, 1985.

Christiansen, H., Recognition of generative languages. *Lecture Notes in Computer Science* 217, pp. 63–81, Springer-Verlag, 1986.

Christiansen, H., The syntax and semantics of extensible languages. *Datalogiske skrifter*, no. 14, Roskilde University Centre, 1988.

Christiansen, H., Declarative semantics of a meta-programming language. *Proc. META 90, Workshop on Meta-Programming in Logic*, Leuven, Belgium, pp. 159–168, 1990.

Clocksin, W.F. and Mellish, C.S., *Programming in Prolog, Third edition*. Springer-Verlag, 1987.

Colmerauer, A., *Les grammaires de metamorphose*, Groupe d'Intelligence Artificielle, Universit de Marseilles-Luminy, 1975. Appears as ""Metamorphosis grammars" in *Lecture Notes in Computer Science* 63, pp. 133–189, Springer-Verlag, 1978.

Dembinski, P. and Maluszynski, J., Attribute grammars and two-level grammars: A unifying approach, *Lecture Notes in Computer Science* 64, pp. 143–154, 1978.

Deransart, P., Jourdan, M., and Lorho, B., Attribute grammars. Definitions, systems and bibliography. *Lecture Notes in Computer Science* 323, Springer-Verlag, 1988.

Deransart, P. and Maluszynski, J., Relating logic programs and attribute grammars. *Journal of Logic Programming* 2, pp. 119–155, 1985.

di Forini, A.C., Some remarks on the syntax of symbolic programming languages. *Communications of the ACM* 6, pp. 456–460, 1963.

Hanford, K.V. and Jones, C.B., Dynamic syntax: A concept for the definition of the syntax of programming languages. *Annual Review in Automatic Programming* 7, pp. 115–142. Pergamon Press, Oxford, 1973.

Heering, J., Klint, P., and Rekers, J., Incremental generation of parsers. *Sigplan Notices* 24/7, pp. 179–191, 1989.

Hill, P.M. and Lloyd, J.W., Analysis of meta-programs. *Proc. META 88, Workshop on Meta-Programming in Logic Programming*, Bristol, England, pp. 27–42, 1988.

Johnson, S.C., Yacc — yet another compiler compiler. *Computing Science Technical Report* 32, AT&T Bell Laboratories, 1975.

Knuth, D.E., Semantics of context-free languages. *Mathematical Systems Theory* 2, pp. 127–145, 1968.

Layzell, P.J., The history of macro processors in programming language extensibility. *The Computer Journal* 28, pp. 29–33, 1985.

Mason, K.P., *Dynamic Template Translators: A useful model for the definition of programming languages*. Ph.D. thesis, University of Adelaide, Australia, 1984.

Mason, K.P., Dynamic template translators — A new device for specifying programming languages. *Intern. J. Computer Math* 22, pp. 199-212, 1987.

Meek, B., The static semantics file, *Sigplan Notices*, Vol. 25, No. 4, pp. 33–42, 1990.

Naur, P., *ed.*, Report on the algorithmic language Algol 60. *Communications of the ACM* 4, 1960.

Naur, P., *ed.*, Revised report on the algorithmic language Algol 60. *Communications of the ACM* 6, pp. 1–17, 1963.

Pereira, F.C.N. and Warren, D.H.D., Definite clause grammars for language analysis — A survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence* 13, pp. 231–278, 1980.

Solntseff, N. and Yezerski, A., A survey of extensible languages. *Annual Review of Automatic Programming* 7, pp. 267–307, 1974.

Uhl, J., Drossopoulos, S., Persch, G., Goos, G., Dausmann, M., Winterstein, G., and KirchgŁssner, W., An attribute grammar for the semantic analysis of Ada. *Lecture Notes in Computer Science* 139, Springer-Verlag, 1982.

U.S. Dept. Defence, *Reference manual for the Ada programming language*, ANSI/
MIL-STD-1815A, 1983.

Wegbreit, B., Extensible programming languages. *Harward University, Cambridge, Massachusetts*, 1970. (*Garland Publishing*, Inc., New York & London, 1980).

Welsh, J., Sneeringer, W.J., and Hoare, C.A.R., Ambiguities and insecurities in Pascal. *Software, Practice and Experience* 7, pp. 685–696, 1977.

van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., Koster, C.H.A., Sintzoff, M., Lindsey, C.H., Meertens, L.G.L.T., and Fisker, R.G., Revised report on the algorithmic language ALGOL 68. *Acta Informatica* 5, pp. 1–236, 1975.