The syntax and semantics of extensible languages

Henning Christiansen
http://www.ruc.dk/~henning

Reference as:
Datalogiske Skrifter no. 14, 1988
(Technical report series)
Computer Science Section
Roskilde University, Roskilde, Denmark

Comment added 2006: The article introduces a grammar formalism called "Generative Grammars". This name is a bit misleading, and later other authors have named these grammars "Adaptive Grammars" and "Christiansen Grammars".

The syntax and semantics of extensible languages

Henning Christiansen

Roskilde University Centre
P.O. Box 260, DK-4000 Roskilde, Denmark

Abstract. It is generally agreed that conceptual and linguistic enrichment is a central concept in computer programming. In the last three decades or more, still more advanced abstraction mechanisms inside programming languages and various suggestions for extensible languages have emerged. However, there has been a lack of appropriate tools for describing the syntax and semantics of linguistic extensibility.

This paper is concerned with generative grammars, a general, syntactic formalism directed towards programming languages with a high degree of extensibility. A generative grammar is a generalization of an attribute grammar: The attribute machinery is made responsible for the creation and management of syntactic rules and the syntactic derivation relation is defined relative to a specialized, inherited attribute. The resulting, syntactic descriptions are simple and transparent, each syntactic construct — predefined or declared by the user — is captured by an explicit, grammatical rule. As for attribute grammars, semantic descriptions can be merged into the attributes.

In this paper, I give a presentation of generative grammars and make a comparison with other syntactic formalisms. I consider two kinds of semantic descriptions, compositional semantics and semantic specifications based on interpretation. The relation between generative grammars and an extended Horn clause logic is also covered.

• CONTENTS •

1 Introduction

Any interesting programming language includes some kind of extension mechanism ranging from simple declarations to the definitions of new control structures or even to the introduction of new computational paradigms. The purpose of this declaration or extension potentiality is, of course, to be able to adapt the language to a given purpose — or, in other words, to make it possible to write more abstract and expressive programs.

The term, 'extensible language', has over the years been used with various, more or less clarified, meanings. It has been used in order to characterize macro processors serving as front-end to compilers for, say, Algol60; the similarity between the language's own declaration mechanisms and the external macros has not been recognized. The word 'extensible' has also been used with respect to Algol68 for facilities now considered standard in programming languages such as Pascal and Ada — languages usually not referred to as extensible. Rather, the use of the word 'extensible' refers to the authors objects and attention.

When I refer to 'extensible languages' or 'linguistic extensibility', I refer to any such possibility for enrichment of a language with new concepts in the shape of new linguistic forms. This means that the programmer — within the programming language — is given the opportunity to describe new entities and to use these entities. This wide and admittedly fuzzy definition of extensibility covers traditional programming languages with simple declaration mechanisms, e.g., Pascal, more powerful programming languages such as LISP, various declared extensible languages, and generators of, for example, compilers and editors.

Concerning the description of programming languages, it is well-known that a context-free grammar is well-suited for describing the non-extensible aspects of programming language syntax: Each linguistic construct is captured by its individual and concise rule. When it comes to the description of extensibility, however, a method for collecting the user's specifications and merge their contents into the existing language description must be provided. Obviously, context-free grammars are not sufficient, but with respect to transparency and succinctness, context-free grammars may serve as an ideal.

In this paper, I present a syntactic formalism, generative grammars, intended for the description of extensible languages. Generative grammars are a generalization of attribute grammars: The attributes and their mutual dependencies appear to be an appropriate medium for the generation and management of new syntactic descriptions. The syntactic derivation relation is defined according to the grammar found in a specialized, inherited attribute attached to each node in a syntax tree. Hence at each program point, the available language is determined by a set of concise rules — similar to a context-free grammar.

In chapter 2, I give a brief and selective review of the history of syntactic descriptions of programming languages. My suggestion for an extensible language formalism, the generative grammars, is presented in chapter 3 by means of a pragmatic discussion and a definition; a useful generalization of the basic, generative grammar framework is also described. A sample grammar for a Pascal-like language equipped with various extension mechanisms can be found in appendix A. A comparison with other selected syntactic formalism is given in chapter 4.

The semantic description of extensible programming languages by means of generative grammars is considered in chapter 4: It appears that the attributes are well-suited for describing semantics, as is well-known from attribute grammars, and for the generation of new semantic descriptions. A full description of the intertwined syntax and semantics of a LISP-like language is given in chapter 6. Chapter 7 covers the representation of generative grammars as Prolog programs. The well-known relation between attribute grammars and Horn clause programs has a natural generalization to generative grammars. Appendix B

gives a translation of the LISP grammar, chapter 6, into a Prolog program — which may be used as an effective LISP interpreter. The final chapter 8 provides for a summary.

Finally, I will stress the pragmatic character of this paper, it concerns the use of grammatical formalisms in the service of describing extensible languages. Formal properties of grammars and languages are of minor interest here.

2 Historical overview

Algol60 was the first well-known programming language to be presented together with a formal, syntactic description. The overall structure of Algol60 programs is described in the Algol60 report, (Naur,1963), by a context-free grammar, (Chomski, 1956). The context-free grammar provides for a modular and transparent description of the superficial phrase structure of a language, but it cannot cover the full syntax — the restrictions imposed by declarations are outside scope of a context-free grammar. In the Algol60 report, these restrictions are described by additional, informal explanations.

The problem of formally describing the full syntax of languages such as Algol60 has motivated a large amount of research work. The developments within the area of syntactic descriptions can be roughly separated into two directions. Much effort has been made in order to provide suitable syntactic descriptions that cover the full syntax of Algol60-like languages and, what is especially of interest in this paper, for the description of extensibility. Grammars and classes of languages have also become research object of their own right; this direction concerns formal properties of grammars, often with a bias towards automata theory and most often separated from the description of programming languages. Examples from the sixties are (Aho, 1968, 1969), (Greibach, Hopcroft, 1969), (Kasai, 1970), (Rosenkrantz, 1969). This direction is still very active — and manifests itself in several volumes of Lecture Notes in Computer Science each year.

For the purpose of programming language description, various syntactic formalisms have been suggested. Chomski's context-sensitive grammars, (Chomski, 1956), posses the power to describe the full syntax of a typical programming language; however, this formalism is simply too powerful and general and no tractable notation for this purpose has been suggested.

The most successful grammatical formalism covering the full syntax of Algol60-like languages is attribute grammars, (Knuth, 1968): An attribute grammar is a context-free grammar in which each nonterminal is equipped with data object, so-called attributes, determined as functions of the neighbouring attributes in a derivation tree. In an attribute grammar, the context information created at the declarations and applied in the program statements can be wrapped into symbol tables and passed around in the derivation tree as attributes. The exclusion of, say, references to undeclared variables is described according to the symbol table. The reason for the success of attribute grammars is that they provide for very readable, and hence useful, descriptions of programming languages — and at the same time, these descriptions are well-suited as guide-lines for implementations: The information flow in the attributes represents an abstraction of the data flow in a compiler. Chapter 4 gives a more detailed account on attribute grammars.

In the sixties, various extension mechanisms were suggested for Algol60, for example (McIlroy, 1960), (Galler, Perlis, 1967). However, these mechanisms have only been described by means of the macro processors that implemented them, no explicit, syntactic descriptions have been given.

The intended successor of Algol60, Algol68, on the other hand, has been designed together with a grammatical formalism, two-level grammars, (van Wijngaarden et al., 1975). In Algol68, it is possible to declare new types (called 'modes') with associated operator systems. In a two-level grammar, information about declared objects is coded in the language of a (context-free) metagrammar; terms of this 'meta-language' are in turns

used to instantiate so-called hyperrules into context-free rules which then may be used for the generation of actual program text. Despite its simplicity, being based on derivations in context-free grammars only, the two-level grammar formalism suffers from severe notational problems and attribute grammars have, in fact, ousted two-level grammars. A more detailed discussion concerning the relation between these formalisms and generative grammars is found in chapter 4.

Attribute grammars and two-level grammars can only describe language extensions as instances of in advance fixed, syntactic patterns. This is reasonable for programming languages such as Algol60 with a modest degree of extensibility, but for more powerful syntactic extensions the principle is rather inadequate; this claim is discussed more thoroughly in chapter 4.

In the extensible language system, ELSA, (Lindsey, 1983), a dialect of two-level grammars, (Turner, 1979), has been used as a specification language: The user describes the syntax of a new language by means of a two-level grammar; the formalism is not used to cover the combination of language definition and use, i.e., it is not used to describe an extensible language.

In order to describe extension mechanism in general, it appears to be natural to understand declarations as creators of new, explicit syntax rules. A.C. di Forini, (1963), seems to be the first to have presented this idea. His paper is a conceptually clear, and still worth reading, argument that declarations of any kind produce new linguistic material and should be treated as such. For example, the declaration of a variable should result in the creation of syntax rule stating the existence of exactly that variable. However, the paper gives no method for actually generating these rules or to control their application (scope rules). Although this idea appears to be strong, it has had a quiet life — probably because context-free grammars equipped with more or less formally specified context conditions, (for example, the Algol60 report's style or an attribute grammar), is reasonable for the description of traditional, i.e. Algol60-like, languages. The idea has been adapted in Wegbreit's extensible context-free grammars in order to describe the extensible programming system ECL, (Wegbreit, 1970). However, this class of grammars is biased towards the description of exactly that system and cannot be viewed as a tool for description of extensible languages in general; even the context-conditions of Algol60 cannot be described. More precise comments will be given in chapter 4.

With a few exceptions, the idea appears to have been resting until the invention of generative grammars: Hanford and Jones, (1973), have suggested a so-called 'dynamic syntax', a monstrous device based on λ-calculus, which also generates new descriptions from declarations. Unfortunately, the suggestion is not given a proper formalization or accompanied with an appropriate notation. In fact, it seems that the authors have been aware of neither attribute grammars nor two-level grammars.

In 1985, I published the first paper on generative grammars, which seems to be the first, general approach to the syntactic and semantic description of extensible languages based on explicit generation of syntax rules from declarations, (Christiansen, 1985).

Mason, (1984, 1987), has independently developed a generalization of syntax-directed translation schemes, called dynamic template translators, in which the analysis of an input string may result in side-effects on the current translation scheme. A comparison with generative grammars is given in chapter 5.

The LISP programming language, (McCarthy, 1960), has not been mentioned in this survey yet — it has, somehow, followed its own historical direction. LISP is both a very early language and at the same time it contains the very idea of linguistic extensibility. In fact, LISP is still one of the most powerful programming languages concerning extensibility, if not *the* most powerful one — at least at the semantic level. A characterization of LISP by means of a generative grammar is given chapter 6.

3 Generative grammars

The generative grammar formalism is a generalization of attribute grammars: The attribute machinery is made responsible for the generation and management of new syntactic rules — and the derivation relation is replaced by a more powerful and context-sensitive one.

In this chapter, I will give a presentation of the generative grammar formalism. To begin with, generative grammars are presented by means of an example, section 3.1. The precise definition of generative grammars, their derivation relation, and a useful generalization are given in section 3.2. Section 3.3 is a pragmatic discussion of the design and use of the generative grammar formalism, The limitations of the generative grammar model, as it is presented here, are discussed in section 3.4. The explanations in this chapter are in principle self-contained although a basic knowledge of attribute grammars is preferred.

Appendix A gives an example grammar for a Pascal-like language equipped with various extension mechanisms. A generative grammar for a quite different language, LISP, can be found in chapter 6.

3.1 Informal presentation

Here I will show an example generative grammar for the following language.

$\{ww \mid w \text{ is a string of letters}\}$

This is a prototype extensible language, the first occurrence of the string, w, can be thought of as the declaration of a new syntactic concept, the second occurrence as its application. A generative grammar for this language is as follows, the notation is adapted from the extended attribute grammars of Watt and Madsen, (1979).

$\langle \text{program}\downarrow G\rangle ::= \langle \text{dcl}\downarrow G\uparrow w\rangle\ \langle \text{body}\downarrow\{w\text{-}rule\}\rangle$
where $w\text{-}rule = \langle \text{body}\downarrow G'\rangle ::= w$
$\langle \text{dcl}\downarrow G\uparrow ch\bullet w\rangle ::= \langle \text{char}\downarrow G\uparrow ch\rangle\ \langle \text{dcl}\downarrow G\uparrow w\rangle$
$\langle \text{dcl}\downarrow G\uparrow(\,)\rangle ::= \langle \text{nothing}\rangle$
$\langle \text{char}\downarrow G\uparrow \text{a}\rangle ::= \text{a}$
. . .

The operator '•' is a generalized operator for constructing and appending arbitrary lists and is used as such throughout this paper. The symbol ⟨nothing⟩ is a meta-symbol which denotes the empty string. The inherited attribute (preceded by ↓) of each nonterminal, in most cases described by the attribute variable G, should be ignored for the moment. Consider, now, the program in our tiny, extensible language,

hubahuba,

and assume that the first 'huba' is generated as a declaration, represented by the nonterminal 'dcl' in the grammar. The synthesized attribute of the declaration collects the following character string.

w = huba.

Now, take a look at the rule for programs; it will tell us that a grammar consisting of a single rule,

$\langle \text{body}\downarrow G'\rangle ::=$ huba,

is given as the inherited attribute to the body. (The variable G' in the rule for programs is covered by a notational convention described below). This rises a question of how this generated rule actually happen to be applied. This reveals the difference between a generative grammar and a plain attribute grammar: In an attribute grammar, derivation,

i.e., the replacement of a nonterminal by a string of nonterminal and terminal symbols, is defined according to the given, fixed set of rules given by the grammar: Find an instance of a rule that matches the given attributed nonterminal and replace the latter by the right hand side of that rule instance. In a generative grammar, on the other hand, the first attribute position is by convention inherited and holds an entire grammar; the derivation relation is defined according to the actual contents of that attribute. The derivation of an entire program is initiated by assigning the initial grammar to this attribute in an initial symbol.

Now, let us return to our favourite, extensible language. In the generation of the declaration, the original grammar is copied around by the attribute variables, all denoted *G* — and everything works out as in an attribute grammar. When it comes to the generation of the body, the available grammar is specific to the given program text.

3.2 Definitions

The concepts are defined more precisely as follows.

Definition 1. A *generative grammar* is a set of *rules* of the form

$$\langle N_0 \dagger exp_{0,1} \dagger \ldots \dagger exp_{0,n0} \rangle ::= \alpha_0 \langle N_1 \dagger exp_{1,1} \dagger \ldots \dagger exp_{1,n1} \rangle \alpha_1 \ldots \alpha_{n-1} \langle N_n \dagger exp_{n,1} \dagger \ldots \dagger exp_{n,nn} \rangle \alpha_n.$$

where

- $N_0, \ldots, N_n$ are *nonterminal symbols*,
- $\alpha_1, \ldots, \alpha_n$ sequences of *terminal symbols*,
- $\dagger$ stands for either $\uparrow$, preceding an inherited attribute (position), or $\downarrow$, preceding a synthesized attribute (position),
- $exp_{0,1}, \ldots, exp_{n,nn}$ are *attribute expressions* which are terms built out of typed function symbols, constants, and variables.

Each inherited attribute expression of N_0 and each synthesized attribute expression of $N_1, \ldots, N_n$ consists of a single, distinct variable; these attribute positions are called *defining* positions. By convention, the first attribute position of any nonterminal symbol should be inherited and its type should represent the set of all generative grammars; it is called the *language attribute* (position).
□

Definition 2. An *instance* of a generative grammar rule is the structure, if any, that results from a substitution of its attribute expression by their respective values assuming a consistent assignment of values to variables throughout the rule. Hence, a rule instance is of the form

$$A_0 ::= \alpha_0 A_1 \ldots A_n \alpha_n$$

where $\{A_i\}$ are *attributed nonterminal symbols*,

$$A_i = \langle N_i \dagger a_{i,1} \dagger \ldots \dagger a_{i,ni} \rangle.$$

The values, $\{a_{i,j}\}$, are called *attributes*.
□

As long as the restrictions on attribute variables and expressions are observed, a 'where' notation, as in the sample grammar in section 3.1, may be used. A pattern matching notation for structured attribute variables may also be used. By convention, a variable that

appears in a where-clause, but does not appear in any defining position in the rule, e.g., *G'* in the sample grammar, is a constant symbol whose value is a variable. Hence such variables remain unbound in the instantiation of rules. (Note the analogy to the treatment of variables in Prolog). This notational convenience is useful for describing generation of new, syntactic rules.

Attribute variables in a generative grammar can be distinguished from constant symbols due to their appearances in defining positions. However, in order to facilitate the reading, variables will be written in italics. In the sample grammar above, for example, it should be clear that the symbol 'a' represents a concrete letter whereas *ch* is a variable that ranges over the set of all letters.

Definition 3. The *derivation relation*, $\Rightarrow$, is defined as follows. Consider a string,

$$\alpha\, A\, \beta$$

where α and β are strings of terminal and nonterminal symbols and A an attributed nonterminal. Whenever there is a rule in A's language attribute and an instance of that rule,

$$A ::= \alpha_0\, A_1\, \alpha_1 \ldots \alpha_{n-1}\, A_n\, \alpha_n,$$

we say that

$$\alpha\, A\, \beta \Rightarrow \alpha\, \alpha_0\, A_1\, \alpha_1 \ldots \alpha_{n-1}\, A_n\, \alpha_n\, \beta.$$

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

The language generated by a generative grammar, viewed either as a set of terminal strings or a family of syntax trees, is defined as usual. By convention, the derivation of a program text is initiated from an initial, attributed nonterminal whose language attribute is the entire, initial grammar.

Finally, I give the definition of a useful variant of generative grammars.

Definition 4. A *representation-based generative grammar* is defined similar to a generative grammar except that the language attribute holds an arbitrary *representation* of a grammar. Associated with a given grammar representation is an *abstraction function* which yields grammars for the elements in the representation.

The derivation relation for a representation-based generative grammar is defined by means of its associated abstraction function.
□

Representation-based generative grammars are useful, for example, in order to apply a stack-based protocol in the description of scope rules: The top-most rule describing a given identifier is the one that counts. In a 'normal' generative grammar in which new rules are added to an existing grammar by means of set union, such scope rules are likely to be unspecified, cf. the examples below and in appendix A. No specialized notation is developed for representation-based generative grammars, all examples are described either informally or in terms a Prolog programs, cf. chapter 7.

3.3 Pragmatics

Generative grammars are intended — not only to appear as a formalism of significant, expressive power — but also, somehow, to reflect the *use* of extension mechanisms such as declarations and abstraction mechanisms. I will illustrate what I mean in discussion around a little toy example.

Consider a simple programming language of integer expressions that allows for the declaration of named constants; the following is a sample program.

1 + **let** two = 2 **in** two + 3 **end**

Suppose, now, our chief programmer has produced the following, partial program.

let two = 2 **in** . . . **end**

The idea of defining the constant is to create an entity, two, that can be applied equally to, say, '3', '7', or '666', in the scope indicated by the ellipsis. This viewpoint materializes in the following generative grammar rule.

⟨exp↓*G*⟩::= **let** ⟨identifier↓*G*↑*id*⟩ = ⟨integer↓*G*⟩ **in** ⟨exp↓*G*∪{*new-rule*}⟩ **end**
where *new-rule* = ⟨integer↓*G'*⟩::= *id*

The partial sample program gives rise to a specialized environment in the shape of a very high-level and special-purpose language. The generated grammar is a perfect starting point for writing a user's manual for this environment. And if semantic descriptions are merged into the grammar as synthesized attributes, this starting point is made even more perfect.

⟨exp↓*G*↑*val*⟩::= **let** ⟨identifier↓*G*↑*id*⟩ = ⟨integer↓*G*↑*n*⟩
in ⟨exp↓*G*∪{*new-rule*}↑*val*⟩ **end**
where new-rule = ⟨integer↓*G'*↑*n*⟩::= *id*

From this rule it immediately follows that the new and highly specialized environment includes a brand-new, special-purpose concept described by the following generated rule.

⟨integer↓*G'*↑2⟩::= two

Generative grammars allow for a conceptually nice treatment of types: Each expression type in a language can be described by its individual nonterminal symbol. This means that the class of expressions of a given type constitutes a well-defined sublanguage described by a well-defined subgrammar. Hence there is no need — as in an attribute grammar — for an explicit level of type checking. Consider, for example, the following rules describing a set of integer expressions.

⟨integer↓*G*⟩::= 0
⟨integer↓*G*⟩::= 1
. . .
⟨integer↓*G*⟩::= ⟨integer↓*G*⟩ + ⟨integer↓*G*⟩

These rules may be used as freely as the rules of a context-free grammar — the explicit generation of new rules for declarations ensures that each available construct is described by its individual syntax rule. In order to use variables in expressions, for example, the grammar may include a rule of the following form.

⟨integer↓*G*⟩::= ⟨integer-variable↓*G*⟩

However, initially there are no rules for 'integer-variables' so generation of references to undeclared variables is impossible. The declaration of a variable may, then, give rise to the generation of a new rule of exactly that variable, e.g.,

⟨integer-variable↓*G*⟩::= n.

These rules, (apart from the last one, of course), are excerpts from the example grammar in appendix A which also describes the actual generation of rules for variables.

This treatment of types is also discussed further in the comparison with attribute grammars, section 4.1.

3.4 Limitations

Here I will discuss two aspects of extensible language syntax which are not obviously covered by the basic, generative grammar model. This concerns the description of user-extensible, polymorphic type systems and of recursive definitions.

Polymorphism

One of the qualities of generative grammars is that a data type can be represented by its own nonterminal symbol. As discussed above, the set of all expressions of a given type will constitute a well-defined sublanguage described by a well-defined subgrammar. A polymorphic operator, on the other hand, is an operator that applies to a class of types, perhaps all types as is the case for the traditional assignment operator. In a generative grammar it is not possible to capture a polymorphic operator by means of a single rule if we insist on the correspondence between expression types and nonterminals.

In the sample grammar in appendix A, a notational trick is used in order to make a polymorphic operator appear as one description. For any given substitution of the meta-variable, *type*, the notation

expression-tools(*type*)

represents a set of rules available for that type:

$$\{\langle \text{stm}{\downarrow}G'\rangle ::= \langle \textit{type}\text{-variable}{\downarrow}G'\rangle := \langle \textit{type}{\downarrow}G'\rangle,$$
$$\langle \textit{type}{\downarrow}G'\rangle ::= \langle \textit{type}\text{-variable}{\downarrow}G'\rangle,$$
$$\ldots$$
$$\}$$

This notational shorthand is then referred to in the description of any type — predefined or declared by the user. (Note that this notation implies constructed nonterminals such as 'integer-variable'). However, this notation is not a part of the formalism so it is impossible to describe the generation of new and user-defined polymorphic operators this way. Furthermore, the trick only works for explicitly defined and named types.

In order actually to model user-defined polymorphic operators by means of generative grammars (based on a finite number of rules), the direct relation between types and nonterminals must be sacrificed: All expressions are clashed into one nonterminal symbol and the type information is moved to the attributes. For example, the following rule may be generated from the declaration of a general list append operator.

$$\langle \text{exp}{\downarrow}G'{\uparrow}\text{list}(X)\rangle ::= \text{append}(\langle \text{exp}{\downarrow}G'{\uparrow}\text{list}(X)\rangle, \langle \text{exp}{\downarrow}G'{\uparrow}\text{list}(X)\rangle)$$

For any expression type, X, this rule implies an 'append' operator for expressions of type list(X). (The notation used here is actually beyond what has been defined and used elsewhere in this paper; the more general attribute expressions used by (Madsen, Watt, 1979) are necessary).

Another possible solution that preserves the desired correspondence between nonterminals and types would be to generalize generative grammars with generic nonterminals: A nonterminal is now a term (over some signature); a class of nonterminals may be specified by means of nonterminal variables. The 'append' operator may then be described as follows.

$$\langle \text{list}(X){\downarrow}G'\rangle ::= \text{append}(\langle \text{list}(X){\downarrow}G'\rangle, \langle \text{list}(X){\downarrow}G'\rangle).$$

Any application of such a rule should imply a full instantiation of nonterminal as well as attribute variables.[1)]

Finally, it should be observed that, formally, there is only a cosmetic difference between these two suggestions. From a pragmatic viewpoint, on the other hand, the use of generic nonterminal symbols is superior — especially when several other attributes are involved.

Recursive declarations

Consider the following, very reasonable and trustworthy, rules describing a block-construct a with recursive procedure declaration.

⟨stm↓G⟩::= **begin** ⟨dcl↓$G \cup G_1$↑G_1⟩; ⟨stm↓$G \cup G_1$⟩ **end**
⟨dcl↓G↑{*new-rule*}⟩::= **procedure** ⟨identifier↓G↑*proc-id*⟩; ⟨stm↓G⟩
where *new-rule* = ⟨stm↓G'⟩::= *proc-id*

In a similar, circular attribute grammar, the attributes may describe symbol tables and a declaration create a new symbol table entry. In this case, the grammar consists of a fixed set of rules and, for a given program structure, the attribute expressions describe a system of equations having a unique solution which, furthermore, is the intuitively correct solution; see, e.g., (Madsen, 1980). So, for example,

```
begin
  procedure p; p;
  p
end,
```

is a correct 'stm' whereas

```
begin
  procedure p; p;
  q
end
```

is not (assuming no external declaration of 'q').

Now, let us return to the generative grammar case and consider the following partially generated 'stm', assuming an external definition of the null statement.

```
begin
  procedure p; null;
  . . .
end
```

Now, the grammar,

{⟨stm↓G'⟩::= p},

is a legal value for G_1 in the rule for the block-construct.

However, let 'bad-grammar' be a constant attribute symbol whose value is the following grammar.

{⟨dcl↓G↑bad-grammar⟩::= **procedure** ⟨identifier↓G↑*proc-id*⟩; ⟨stm↓G⟩,
⟨stm↓G⟩::= nonsense
}

1. Actually, a grammar involving generic nonterminals can be viewed as a representation-based generative grammar, cf. definition 4, chapter 3: An infinite, generative grammar is represented as a finite set of rules, some of which include nonterminal variables.

This value is unfortunately also a legal value for G_1 and hence, according to the generative grammar, the text

```
begin
  procedure p; null;
  nonsense
end
```

is a legal 'stm'.

A set of suitable restrictions should be developed in order to exclude the generation of such undesirable grammars; this will involve a detailed study the set theoretic consequences of recursive rules as the one for the block-construct above.

The LISP grammar in chapter 6 describes recursive functions by multiple analysis of function bodies so no recursive rule as the one above is needed. In a similar way, recursive declarations based on so-called forward declarations — as in Pascal, (Jensen, Wirth, 1974) — can be described directly in a generative grammar without any complications.

4 Comparison with other syntactic approaches

In this chapter, I give a more detailed presentation of selected syntactic formalisms that cover certain context-sensitive aspects of programming language syntax. These formalisms will be evaluated according to their qualities for the description of extensible language syntax and compared with generative grammars.

Most space is spent on attribute grammars, (Knuth, 1968), for two reasons: Generative grammars are a generalization of attribute grammars and, secondly, attribute grammars are the most widespread formalism for describing the full, context-sensitive syntax of traditional programming languages.

Another well-known syntactic tool that has been used for describing certain extensible language features is the two-level grammar formalism, (van Wijngaarden, et al., 1975), which has received much attention for its theoretical properties. I have not investigated the formal relationship between my generative grammars and two-level grammars, the comparison is made on a more pragmatic level.

The ECL system, (Wegbreit, 1970), is a so-called extensible programming system often referred to in the literature and one of the few such presented together with a formal, syntactic description. The syntactic aspects of ECL are described by a so-called extensible context-free grammar which appears to be a special, but rather restricted, case of a generative grammar.

The use of operator precedence is also considered and a few comments are given to the descriptive tools suggested by Hanford and Jones, (1973), and Mason, (1987). The final section of this chapter provides for a summary.

4.1 Attribute grammars

Attribute grammars were first described by Knuth, (1968), originally intended for the specification of semantics of context-free languages. However, attribute grammars have proved to be the most successful tool for describing the full, context-sensitive syntax of Algol-like languages.

An attribute grammar is an extension of a context-free grammar: To each node in a syntax tree is associated a number of attributes, each determined as a function of attributes of its neighbouring nodes. Inherited attributes are functions of attributes of sibling and father nodes in the tree and thus inherited attributes may be used to pass contextual information into a subtree. Synthesized attributes are functions of attributes of offspring nodes and hence tend to pass descriptions of a given tree to its context.

Consider, for example, the Algol60 block-construct whose superficial structure is described in the following context-free rule. (For reasons discussed in section 3.4, recursive declarations are not considered in this discussion).

⟨stm⟩::= **begin** ⟨dcl⟩ ; ⟨stm⟩ **end**

The declaration part is understood according to a global symbol table which it receives as an inherited attribute. The contents of the declaration is described by a local symbol table, generated as a synthesized attribute, which is merged together with the global table and in turns given to the body of the block as an inherited attribute. This can be written as a formula using the notation of Watt and Madsen, (1979), as follows; as in the notation for generative grammars, cf. the previous chapter, the arrows, ↓ and ↑, indicate inherited and synthesized attributes, resp.

⟨stm↓*global-symbol-table*⟩::=
begin ⟨dcl↓*global-symbol-table*↑*local-symbol-table*⟩ ;
⟨stm↓*global-symbol-table*⊔*local-symbol-table*⟩ **end**

Certain context-sensitive aspects of programming language syntax tend to have a character of constraints such as type requirements or the ban on declarations of more than one item bearing the same name at a given scope level. Such constraints can be embedded in an attribute grammar as explicit conditions or by partial functions.

⟨stm↓*symbol-table*⟩::=
⟨variable↓*symbol-table*↑$type_{var}$⟩ := ⟨exp↓*symbol-table*↑$type_{exp}$⟩
provided that $type_{var} = type_{exp}$
⟨dcl↑combine($table_1$, $table_2$)⟩::= ⟨dcl↑$table_1$⟩ ; ⟨dcl↑$table_2$⟩
where the combine function is undefined whenever the
two tables contain bindings for the same identifier,
otherwise it specifies the union of the tables.

In the notation used by Knuth, (1968), the value of an attribute is described by a single function symbol, i.e., it is determined by a monolithic function which cannot be analyzed further within the syntactic framework. The notation suggested by Watt and Madsen, (1979), for their so-called extended attribute grammars, which I also have used in this paper, shows two advantages over the notation of Knuth, which for some reason still is the most commonly used notation for attribute grammars:

- It has a compact expression and a concise reading.
- It accentuates the algebraic properties of the attribute dependencies: An attribute is viewed as the value of an explicit term whose variables stand for other attributes.

This notation also allows constraints to be expressed indirectly by the use of identical attribute variables, for example:

⟨stm↓*symbol-table*⟩::=
⟨variable↓*symbol-table*↑*type*⟩ := ⟨exp↓*symbol-table*↑*type*⟩

In fact, this is only a simple example of the more general pattern matching mechanism in extended attribute grammars.

The relation between attribute grammars and generative grammars

Formally, an attribute grammar is a special case of a generative grammar in which the language attribute is constant. However, the different styles of syntactic specification in the two approaches need to be illustrated by an example. Consider the following context-free rule that describes the shape of a variable in a traditional programming language.

⟨variable⟩::= ⟨identifier⟩

For any application of this rule, the actual identifier is expected to be declared as a variable in an appropriate context and the type of the variable is given in the declaration. In an attribute grammar, the context condition is conveniently captured by a symbol table attribute (created at the declarations) that in turns is used to determine the actually allowed instances of the rule.

⟨variable↓*table*↑determine-type(*id*, *table*)⟩::= ⟨identifier↑*id*⟩
where determine-type is undefined if *id* is unbound in *table*,
otherwise it yields the type bound to *id* in *table*.

In a generative grammar, on the other hand, the context information is naturally merged into the grammar: For each declaration of a variable, a specific syntactic rule is created. The declaration

var n: integer

may thus by understood as the creation of a rule

⟨variable↓*G*↑integer-type⟩::= n

or, expressed slightly different,

⟨integer-variable↓*G*⟩::= n.

The general pattern for variables does not appear as a rule in the grammar — but each actual variable is captured by a rule which is a particular instance of that pattern. Hence, the generative grammar above is more transparent than its attribute grammar counterpart in the sense that, for each nonterminal in a syntax tree, the permissible derivations from it are determined by the superficial structure of the rules in the (local) grammar. In other words, the generative grammar descriptions require fewer constraints and are thus of a more generative nature. However, in order to express the requirement that two variables declared at the same block level cannot have identical names, even a generative grammar must use a constraint — which is conveniently wrapped as a partial function.

There is another essential difference between the two: In a generative grammar we can do more than just creating instances of general patterns, we are free to introduce completely new, syntactic structures — not foreseen by the designer of the initial grammar. Declaration of new control structures consisting of arbitrary sequences of formal arguments with interspersed keywords provides for a good example; cf. appendix A.

However, a certain degree of extensibility can be expressed in an attribute grammar. Consider the language

{*ww* | *w* is a string of letters}

As observed in chapter 3, this is a simple, extensible language, the first *w* being a declaration of a new, syntactic pattern; the second *w* is its application. The language can be described by the following (extended) attribute grammar:[1)]

1. As elsewhere in this paper, the • symbol is a combined operator for construction and concatenation of arbitrary lists.

⟨program⟩::= ⟨dcl↑*w*⟩ ⟨body↓*w*⟩
⟨dcl↑*ch*•*w*⟩::= ⟨dcl-char↑*ch*⟩ ⟨dcl↑*w*⟩
⟨dcl↑⟨⟩⟩::= ⟨nothing⟩
⟨dcl-char↑a⟩::= a
. . .
⟨body↓*ch*•*w*⟩::= ⟨body-char↓*ch*⟩ ⟨body↓*w*⟩
⟨body↓⟨⟩⟩::= ⟨nothing⟩
⟨body-char↓a⟩::= a
. . .

The superficial structure of this grammar, i.e., the underlying context-free grammar, states that any character string is a potential body — and this enormous space is thinned out by a constraint coded into the attribute expressions.

On the other hand, consider the generative grammar for this language presented in chapter 3. In the generation of the string 'hubahuba', the following explicit rule is created.

⟨body↓*G*⟩::= huba

As opposed to the attribute grammar above, the generative grammar description is truly generative, no constraints are needed.

The question as to whether this example can be generalized in order to transform a practical class of generative grammars in to equivalent attribute grammars have not been investigated. If, in fact, this really happens to be the case, the actual construction is probably of no use: Extensibility, the creation of new syntactic possibilities, is described in a rather indirect and 'coded' fashion.

It should be emphasized, however, that attribute grammars are the most used and most successful tool for the full description the syntax of traditional programming languages: It provides for a precise and readable notation — and at the same time the attributes and their dependencies represent an abstraction over the data structures and the data flow in a compiler. Many practical classes of attribute grammars associated with efficient implementation techniques have been isolated, cf. the bibliography (Deransart, Jourdan, Lorho, 1985).

For traditional programming languages, i.e., those usually not thought of as extensible, there is really no need for the invention of generative grammars: They represent a conceptually slightly nicer view of certain linguistic aspects, all right, but their use as a basis for implementations is less obvious — and their power and generality are not used.

4.2 Two-level grammars

The grammatical formalism known as two-level grammars has been developed in order to describe the syntax of the programming language Algol68, cf. (van Wijngaarten et al., 1975). The novel features in Algol68 that motivated the development of a new kind of grammar was extensibility over expression types and their operators. It was also a goal to be able to describe the full syntax of the language, including the context-sensitive aspects introduced by declaration of variables, etc.

As an attribute grammar, a two-level grammar is a generalization of context-free grammars. A two-level grammar consists of a set of hyperrules and a metagrammar. A hyperrule serves as an abbreviation of a potentially infinite set of context-free rules that may be used in the generation of actual program text. Hyperrules are parameterized by metanotions and these metanotions are to be replaced by hypernotions generated by the metagrammar.

This is illustrated by a description of the sample language,

$\{ww \mid w \text{ is a string of letters}\}$.

The hyperrules for this language are as follows.

(1) **program: W, W.**

(2a) **letter a W endw:** a, **W endw.**

(2b) **letter b W endw: b, W endw.**

. . .

(3) **endw: .**

The right and left hand sides are separated by the **':'** symbol, the elements of the right hand side by commas, and the period indicates the end of a rule. Terminal symbols, such as 'a', 'b', etc., are written in normal font.

Consider rule (1). Its right hand side consists of two identical metanotions denoted **'W'**. This metanotion (indicated by boldface, capital letters) must be replaced consistently throughout that rule by a so-called protonotion in order to apply it, the rule, in the generation of terminal strings. The resulting rule instance can be thought of as a context-free rule; the role of nonterminal symbols is performed by the protonotions that appear as strings of lowercase, boldface letters.

The protonotions are described by an additional metagrammar. The metagrammar for our sample language consists of the following rules.

W:: LETTER, W; endw.

LETTER:: letter a; letter b; . . . .

Metarules are distinguished from hyperrules by the use of **'::'** to separate the left and right hand sides; the semicolons indicate alternative productions. The metanotions (capital letters) serve as nonterminal symbols in this grammar and hence the following protonotion is a legal **'W'**.

letter h letter u letter b letter a endw

New, returning to the hyperrules, this protonotion is a legal substitute for **'W'** in rule (1), which yields the following, fully instantiated rule.

program: letter h letter u letter b letter a endw,
letter h letter u letter b letter a endw.

The right hand side of this grammar consists of two, fairly large, nonterminal symbols. In order to derive anything from these nonterminals, we must find an appropriate hyperrule and instantiation of it. Hyperrule (3h) with the substitution

W = letter u letter b letter a endw

yields the following rule.

letter h letter u letter b letter a endw:
h, letter u letter b letter a endw.

So now, we may write down the following derivation:

program ⇒

letter h letter u letter b letter a endw, letter h letter u letter b letter a endw ⇒

h, letter u letter b letter a endw, letter h letter u letter b letter a endw ⇒

h, letter u letter b letter a endw, h, letter u letter b letter a endw

Continuing this way using appropriate instantiations of the rules (2u), (2b), (2a), and finally (3) we derive the terminal string

hubahuba.

Two-level grammars are closely related to attribute grammars: The metarules of a two-level grammar correspond to the rules of an attribute grammar, the protonotions to the attributes. In two-level grammars, the protonotions are described by derivations in a context-free grammar as opposed to an attribute grammar, in which arbitrary functions can be used to construct attribute values. However, the attribute grammar's distinction between inherited and synthesized attributes has no counterpart in two-level grammars. The precise relationship between the two formalisms is described in (Dembinski, Maluszynski, 1978).

Viewed in isolation, relieved from its applications, the two-level grammar formalism appears to be a clearer and conceptually simpler system than attribute grammars: The process of generating program text from a two-level grammar can be understood solely by derivations from context free rules: Hypernotions are generated by a context-free metagrammar, these hypernotions are used for instantiating hyperrules which then appear as context-free rules that generate strings of terminal symbols. Two-level grammars, despite their simple, formal concepts, are extremely powerful, even the ban on multiple declarations for the same identifier and user-defined operator precedence can be described within the pure two-level framework, cf. (van Wijngaarten et al., 1975). In fact, Sintzoff, (1976), has demonstrated that any recursively enumerable set can be described by a two-level grammar.

However, in the service of describing programming languages, two-level grammars are somewhat disappointing: The necessary contextual information can only be embedded in the protonotions in a very tricky manner. Furthermore, in order to express the desired relations between these protonotions, additional meta- and hypernotions that generate the empty terminal string (or, in the case of an 'error', no string at all) must appear in the hyperrules. In other words, a complicated and not very obvious coding is needed and the resulting syntactic descriptions tend to be extremely hard to read. The Algol68 report, (van Wijngaarten et al., 1975), is a perfect evidence for this claim! Similar conclusions are given in the survey paper by Marcotty, Ledgard, and Bochmann, (1976).

In an evaluation of two-level grammars as a general tool for describing extensible languages, the conclusion is analogous to attribute grammars, namely that language extensions must be understood by means of instances of general grammatical rules: Hence extensions that have a character of 'unforeseen' grammatical constructs must be coded into the meta- and hypernotions in a rather speculative manner.

4.3 Wegbreit's extensible context-free grammars

In order to describe the extensible programming system ECL, Wegbreit invented a formalism called extensible context-free grammars, (Wegbreit, 1970). ECL is user-extensible over the fixed set of syntactic categories in the 'ground' language, EL1. Types in ECL are considered to be a dynamic concepts not covered by the extension mechanism.

An extensible context-free grammar consists of a context-free grammar describing the original language and a finite state transducer. The set of syntactic rules available at a given program point is found as the initial grammar with some rules added or deleted, these rules being specified by the output of the finite state transducer applied to the text to the left of that program point.

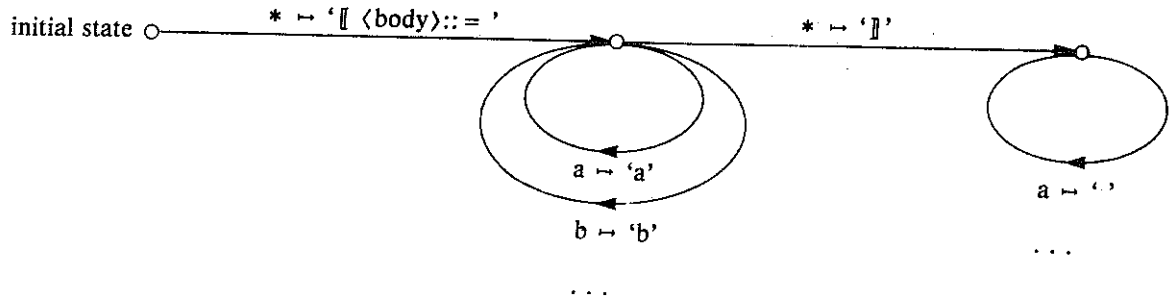
Consider, as an example, the following variant of our favourite extensible language. The '*' symbols serve as special markers to direct the finite state transducer.

$\{ *w*w \mid w$ is a string of letters$\}$

The initial grammar consists of the following rules.

⟨program⟩::= * ⟨dcl⟩ * ⟨body⟩
⟨dcl⟩::= ⟨char⟩ ⟨dcl⟩
⟨dcl⟩::= ⟨nothing⟩
⟨char⟩::= a
. . .

The finite state transducer is specified by the following diagram. Each transition is represented as an edge in the diagram labelled by a translation of an input symbol to a (possibly empty) string of output symbols. The output symbols '⟦ ' and '⟧' signifies the beginning and end of a generated syntax rule.



For the following, partially specified program,

huba⟨body⟩,

the finite state transducer produces the following output from the text up to the residual nonterminal.

⟦ ⟨body⟩::= huba⟧

The generation of terminal strings and the finite state transducer's activities are syncronized by only allowing expansion of the leftmost nonterminal in a partially generated program text. Returning to the sample program text, the termination of the generated rule is signified by the output symbol '⟧', and the completed rule is added to the grammar and is available for the generation of the ⟨body⟩.

Rules to be deleted from a grammar are produced in a similar manner by the finite state transducer except that another special symbol is used for '::='. Any nonsense produced by the finite state transducer up to a certain point that cannot be recognized as well-born syntactic rules is ignored.

It can be seen that the formalism of extensible context-free grammars is simple and constructive — and hence may seem reasonable from an implementation point of view.

However, an extensible context-free grammar possesses a fixed set of nonterminals and hence it is impossible to express extensible type systems. Due to the nature of finite state transducers, extensible context-free grammars cannot either cope with traditional, static scope rules. Neither is it possible to describe 'higher-order' constructs such as Pascal's 'with'-statement — that would require a modification of the finite state transducer.

In can be concluded about this grammatical formalism, that one can describe systems in which syntactic rules explicitly mentioned in the source text (qua a finite state translation) can be added or deleted. Furthermore, an extensible language described this way is tied to a fixed, finite set of syntactic categories.

Extensible context-free grammars can be seen as a special case of generative grammars: Consider an extensible context-free grammar consisting of a context free grammar, G_0, and a finite state transducer characterized by a function, T, such that the transducer's output

for a string, s, is $T(s)$. For each rule in G_0, we construct a similar generative grammar rule as follows: Each nonterminal is equipped with

- a language attribute, of course,
- an inherited attribute that holds the text to left of the nonterminal in a given derivation, and
- a synthesized attribute that holds the text to the left of and including the text generated from this nonterminal.

The transformation of rules of extensible context-free grammars into equivalent generative grammar rules is described by means of an example. Let the following be such a rule.

$$A ::= \alpha\ B\ \beta$$

The corresponding generative grammar rule is as follows where 'P' is a suitable operator for combining the output of the finite state transducer with an existing grammar.

$$\langle A{\downarrow}G{\downarrow}\mathit{textuptoA}{\uparrow}\mathit{textincludingB}{\bullet}\beta\rangle ::= \alpha\ \langle B{\downarrow}P(G_0,\ T(\mathit{textuptoA}{\bullet}\alpha)){\downarrow}\mathit{textuptoA}{\bullet}\alpha{\uparrow}\mathit{textincludingB}\rangle\ \beta$$

In the input languages for parser generators, compiler generators, etc., it is custom to separate the specifications of abstract, concrete, and lexical syntax. Such conventions are also outside the scope of extensible context-free grammars: The formalism has no means for collecting and merging together information scattered over the program text, syntactic rules must be 'read in' in a strictly sequential order.

It appears that extensible context-free grammars are too weak and too restricted in order to qualify as a general formalism for describing the syntax of extensible languages.

4.4 Precedence grammars

An inherent problem in language design, and especially in extensible languages, is the question of ambiguity: The language designer may accidentally introduce, say, a system of operators whose (generated) grammar contains semantically significant ambiguities. One way of solving this problem is to specify for each operator, its precedence and associativity in order to achieve a unique interpretation of the program text.

This was first suggested by Floyd, (1963), and used among others by Wirth and Weber, (1966), for the specification of the Euler language; however, this work was not biased towards user-extensible languages. Turner, (1979), has used a version of this scheme in an extensible language system — although not in the shape of a formalism for describing the extensible language, i.e., the combination of definitions and applications of language constructs. The definition of Algol68, (van Wijngaarden et. al., 1975), shows the use of two-level grammars for describing user-defined operator precedence.

The now most widespread and well-known application of user-defined precedence and associativity is probably the operator definitions in Prolog, see, e.g., (Clocksin, Mellish, 1984). For example, with the following declarations,

```
op( 2, xfx, are).
op( 1, xfy, and).
```

the phrase,

```
girls are nice and pretty
```

is a convenient writing for the proper Prolog phrase

```
are( girls, and ( nice, pretty)).
```

This form of notational extensibility can be described in a generative grammar by attaching the precedence and associativity information as synthesized attributes to the generated syntax rules. A constraint, implemented as a partial function, can ensure the observance of the given precedence and associativity. In a generative grammar for Prolog, the rule for operator declarations should generate the appropriate rules: For the example operators above, the following rules should be generated.

$\langle \text{term}{\downarrow}G{\uparrow}\text{precedence}(\langle 2, \text{xfx}\rangle, p_1, p_2)\rangle ::=$
$\quad \langle \text{term}{\downarrow}G{\uparrow}p_1\rangle \text{ are } \langle \text{term}{\downarrow}G{\uparrow}p_2\rangle$
$\langle \text{term}{\downarrow}G{\uparrow}\text{precedence}(\langle 1, \text{xfx}\rangle, p_1, p_2)\rangle ::=$
$\quad \langle \text{term}{\downarrow}G{\uparrow}p_1\rangle \text{ and } \langle \text{term}{\downarrow}G{\uparrow}p_2\rangle$

The 'precedence' function is defined as follows.

$\text{precedence}(<p_0, a_0>, p_1, p_2) =$
$\quad p_0 \text{ if } a_0 = \text{xfx and } p_0 > p_1 \text{ and } p_0 > p_2, \text{ or}$
$\quad\quad a_0 = \text{xfy and } p_0 > p_1 \text{ and } p_0 \geqslant p_2, \text{ or}$
$\quad\quad a_0 = \text{yfx and } p_0 \geqslant p_1 \text{ and } p_0 > p_2;$
$\quad \text{undefined otherwise}$

In should be clear that other ways of expressing precedence and associativity, that may appear more appealing to the novice user, can be described in generative grammars in a similar manner. An interesting treatment of syntactic ambiguity and how to resolve it is given by Aho, Johnson, and Ullman, (1975).

4.5 Other grammatical formalisms

Here, I will comment on two other suggestions for extensible language formalisms in which, similar to generative grammars, declarations are viewed as creators of new rules of the same sort as those found in the initial description.

Hanford and Jones, (1973), propose a fairly complicated device based on λ-calculus, called dynamic syntax. However, due to the lack of a proper formalization in the referenced paper it is not possible to judge whether dynamic syntax is a special case of generative grammars — although it actually seems to be the case. A comparison of the two approaches at a superficial level reveils that my generative grammars fully satisfy the conceptual requirements posed by the authors for an extensible syntactic formalism and that generative grammars are superior concerning the clarity of the applied notation.

Mason, (1987), presents a generalization of context-free translation schemes called dynamic template translators: The application of a translation rule will trigger off an action sequence that works by side-effects on the current translation scheme. These action may add or delete translation rules or modify their action sequence. This imperative nature of the extension mechanism implies the need to provide for a fixed order of applications of these rules. The actual derivation relation for dynamic template translators is defined in terms of a recognition device based on a shift-reduce parsing algorithm. In other words, as in Wegbreit's extensible context-free grammars, cf. section 4.3, only left-most derivations are allowed.

A dynamic template translator represents a special case of a generative grammar: Consider a dynamic template translator rule which represents a context-free rule,

$$N_0 ::= \alpha_1 N_1 \alpha_2 \ldots \alpha_{n-1} N_n \alpha_n,$$

with an associated action sequence, **S**, viewed here as a function from translation schemes to translation schemes, and an output string, w. This dynamic translation rule is easily converted to the following generative grammar rule.

$$\langle N_0{\downarrow}G_0{\uparrow}S(G_n){\uparrow}w_1 \bullet \ldots w_n \bullet w\rangle ::= \alpha_1 \langle N_1{\downarrow}G_0{\uparrow}G_1{\uparrow}w_1\rangle \alpha_2 \ldots \alpha_{n-1} \langle N_n{\downarrow}G_{n-1}{\uparrow}G_n{\uparrow}w_n\rangle \alpha_n$$

However, I find that the imperative action sequences and their control structure do not serve as an appropriate medium for the specification of language extensions. For example, in order to model static scope rules, a rather intricate construction is needed in order to remove the generated rules at block exit. The applicative style of specification applied in a generative grammar seems more suited for the specification of grammars and their elaboration.

4.6 Summary

In this chapter, I have described a number of syntactic formalisms that allow certain kinds of context-sensitive syntax to be specified. Such descriptive tools may roughly be separated in to two classes which may be characterized as the restrictive and the generative approach, respectively.

- Those that rest on fixed sets of (context-free-like) rules whose application is restricted in various ways; attribute grammars and two-level grammars belong to this class.
- Those that explicitly extend or modify the set of available rules; generative grammars and the proposals of Wegbreit, Hanford and Jones, and Mason belong to this class.

It appears that the generative principle is the right one for extensible languages, otherwise the set of rules in a syntactic description becomes too general and the system of restrictions too complicated. The problem with the generative approach, on the other hand, has been the lack of a suitable framework and notation for describing language extensions — a problem I have tried to solve with my generative grammars. In the following, I will review the conclusions of the previous sections.

An attribute grammar is well-suited for describing the context-sensitive aspects of traditional programming languages. Consider, for example, the class of variables or the class of procedure calls in languages such as Pascal or Ada. Each such class corresponds to a general, syntactic rule and the validity of an application of that rule, the context condition, is conveniently described in the attributes. When it comes to more powerful extension mechanisms, the attribute grammars loose their qualities of succinctness and transparency: The underlying context-free rules tend to be too general and not very informative — and the context conditions must be implemented by heavily loaded attributes and strong constraints.

In a generative grammar, on the other hand, a syntactic extension, say the declaration of a new control structure, is 'coded' in its most natural form, namely as a syntactic rule — and this rule is merged into the generative, grammatical machinery equally to any other syntax rule of the language. Compared with attribute grammars, generative grammars provide for a more generative — and thus more transparent — descriptions with fewer constraints.

Concerning types of expressions, generative grammars allow for a radically different treatment than attribute grammars. In a generative grammar for a language such as Pascal, it is possible to let each type, built-in or user-defined, correspond to a unique nonterminal in the grammar. The class of expressions of that type is generated *freely* from a well-defined subgrammar. In an attribute grammar, on the other hand, the class of all expressions must be captured by a fixed set of general rules — and hence an explicit level of type checking must appear in the grammar in the shape of certain constraints or conditions that must hold among the attribute values.

The only typical programming language aspect for which a generative grammar must use a constraint is the ban on declarations of entities with coincident names at the same block

level. This constraint can be represented in a partial function that refuses to combine sets of rules that include elements resembling each other too much.

Most of the comments on attribute grammars goes for two-level grammars as well. In a two-level grammar, context information is coded into the hyper- and protonotions, and the less trivial context information, the more complicated 'coding' is necessary. These arguments should also be seen in the light of the severe notational problems with two-level grammars, even for languages with a modest degree of extensibility, Algol68 for example.

Wegbreit's extensible context-free grammars share the explicit generation of syntactic rules with generative grammars but they are of a rather limited use. It is only possible to read in syntactic rules literally from the program text, no elaborate analysis, and thus no sophisticated declaration mechanisms, are within reach. Furthermore, the set of nonterminals in a language described by an extensible context-free grammar is fixed and traditional scope rules cannot be described either.

Finally, I find that the extensible context-free grammars' use of a finite state transducer for generation of new rules is rather awkward: The attribute machinery, which generative grammars has adapted from attribute grammars, is both more powerful and results in much cleaner descriptions.

The suggestion for a dynamic syntax given by Hanford and Jones, (1973), seems to be closely related to my generative grammars. However, the presentation of dynamic syntax suffers from the lack of a proper formalization and an appropriate notation. Mason's, (1987), dynamic template translators are an 'extensible' version of syntax-directed translation schemes, the extension mechanism being based on imperative action sequences. A dynamic template translator appears to be a special case of a generative grammar. Furthermore, the applicative specification style applied in generative grammars, vs. action sequences, seems more appropriate for operating on grammars as data objects.

An inherent problem in extensible languages is the question of ambiguity: The user may accidentally introduce, say a system of operators, whose generated grammar contains semantically significant ambiguities. One way of solving this problem is to require that the user explicitly specifies the precedence and associativity of each operator in order to achieve a unique interpretation of expressions. Such conventions are easily embedded in a generative grammar, cf. section 4.4.

5 Semantics of extensible languages

The semantics of traditional, almost-context-free programming languages is an extensively studied and reasonably understood field in which several descriptive tools have been developed, Hoare's axiomatic specifications, (Hoare, 1969; Meyer, Halpern, 1982), and denotational semantics, (Milne, Strachey, 1976), for example. A generalization of a semantic formalism to extensible languages should provide for a means for picking up the user's semantic specifications — formulated in the given programming language — and converting them into the given formalism.

As for the syntactic descriptions, I also aim at a uniform treatment at the semantic level of the predefined language elements and the novel constructs defined by the user. Hence the approach taken in denotational semantics for, say, procedures is not sufficient: Each procedure declaration gives rise to a new entry in a symbol table, usually called an environment, and the meaning of any program phrase is a function of, among other things, such an environment. The meaning of each original language construct is described by its particular semantic equation whereas the class of all potential procedure calls is compressed into one futile equation.

I do not intend to present one style of semantic descriptions for extensible languages and claim it to be valid for all extensible systems and languages. Rather, it is my aim to demonstrate that generative grammars allow for a wide spectrum of semantic descriptions.

Here, I will show two kinds of semantic specifications by means of generative grammars, compositional and interpretive semantics.

Compositional semantics is a generalization of denotational semantics to extensible languages. The meaning of a program phrase is an abstract expression of a computation and the semantic specification for each syntactic operator describes a combination of the computations for its arguments. Compositional semantics resembles programs in a procedural language such as Algol60: New meanings in the shape of procedures are composed from existing procedures and primitive language constructs. An example of a compositional semantic specification is shown in section 5.1.

Defining the semantics of a new language by means of an interpreter, on the other hand, provides for much more flexibility and generality. Such semantic specifications allow for the introduction of new paradigms, i.e, not only new meanings but new kinds of meanings. Using interpreters one can, for example, add 'goto's and coroutines to a strictly sequential language — or add a layer of logic programming to a procedural language. Section 5.2 shows an example of a semantic specification based on interpreters embedded in a generative grammar.

Compositional semantics and semantics based on interpretation are only intended to serve as examples of possible semantic styles in generative grammars; the generative grammar for LISP found in chapter 6 is neither one or the other, rather it is some kind of a mixture that reflects the fuzzy border between syntax and semantics in LISP.

Section 5.3 provides for a comparison with other work related to the semantics of extensible languages.

Finally, it must be admitted that the terms 'compositional semantics' and 'interpretive semantics' have been used quite ambiguously; it may refer either to the style used within a language for describing (in the source program text) the semantics of new language elements — or to the style applied in the underlying, generative grammar. This is not by accident; these two levels are coupled tightly together in that the underlying formal framework determines the scope of the potential extension mechanisms.

5.1 Compositional semantics

It is well-known that any denotational semantics can be formulated as an attribute grammar in which the (synthesized) attributes are used to convey the meanings of program phrases, cf. (Mayoh, 1981), (Madsen, 1980). Compositional semantics is the generalization of denotational semantics to extensible languages based on generative grammars.

In the following, I will show a generative grammar that incorporates a compositional semantic description for declarations of new infix operators. For example, a new operator, '+ +', may be defined as follows.

new operator a + + b = a + b + b

The intended meaning of this declaration is that new expressions such as

5 + + 2

can be written, the sample expression is supposed to represent the value 5 + 2 + 2 = 9.

It is convenient to use textual substitution in the generation of new semantic descriptions and hence semantic objects are considered as being terms (as opposed to functions). A formal parameter is specified as an identifier which gives the name of that parameter; the nonterminal symbol, 'symbol', is expected to represent a class of lexical symbols distinct from identifiers. Now, the generative grammar rule for operator declaration is as follows.

⟨dcl↓G↑*new-rule*⟩::= **new operator** ⟨parameter↓G↑$name_1$⟩ ⟨symbol↓G↑*operator*⟩
⟨parameter↓G↑$name_2$⟩ = ⟨exp↓$G \cup \{par_1, par_2\}$↑*sem-term*⟩

where

par_1 = ⟨exp↓G'↑param($name_1$)⟩::= $name_1$
par_2 = ⟨exp↓G'↑param($name_2$)⟩::= $name_2$
new-rule = ⟨exp↓G'↑*new-sem-term*⟩::=
⟨exp↓G'↑*sem-term*$_1$⟩ *operator* ⟨exp↓G'↑*sem-term*$_2$⟩

where *new-sem-term* is a copy of *sem-term* in which each occurrence of 'param($name_1$)' is replaced by the attribute variable '*sem-term*$_1$' and 'param($name_2$)' by the attribute variable '*sem-term*$_2$'

The semantic operator 'param(−)' serves as a special marker whose only purpose is to be replaced textually by attribute variables; note that the substitution takes place at 'declaration time' and that 'param(−)' does not appear in the generated rule.

The parameter transmission described in the rule above is analogous to in-line compilation of procedure calls with embedded code for the actual parameters. Assume, for example, that the plus operator is defined as follows.

⟨exp↓G↑a_1; a_2 ! plus⟩::= ⟨exp↓G↑a_1⟩ + ⟨exp↓G↑a_2⟩

The attribute expression, 'a_1; a_2 ! plus', represents a semantic term parameterized by the variables 'a_1' and 'a_2'. The semantic term is written in the signature of Mosses' (1983) abstract semantic algebras. With this definition of plus, the declaration

new operator a + + b = a + b + b

will result in the generation of the following rule.

⟨exp↓G'↑*sem-term*$_1$; *sem-term*$_2$! plus; *sem-term*$_2$! plus⟩::=
⟨exp↓G'↑*sem-term*$_1$⟩ + + ⟨exp↓G↑*sem-term*$_2$

Various other parameter passing mechanisms can be described by the generation of other semantic terms. Gordon, (1979), provides for a catalogue of denotational characterizations of various parameter passing mechanisms. In should be noted, that the textual substitution applied in the example above is specific to the actual example and not a characteristics of compositional semantics; however, textual substitution and renaming are useful techniques in order to generate compact and concise semantic descriptions.

The paper, (Christiansen, 1985), provides for a comprehensive example of a compositional semantic specification written as a generative grammar.

5.2 Interpretive semantics

An extensible language which allows its user to describe semantics by means of interpreters should, of course, contain a set of basic facilities that makes it well-suited for writing interpreters. This includes a representation of program text as data objects within the language.

In the following, I will sketch a generative grammar for a language with an explicitly represented phrase structure and semantic specifications based on interpretation. In this language, one can define new operators and either specify their semantics immediately in terms of an interpretation procedure or defer the semantic specification to another level. In this example, all expressions are expected to be of the same type but the method can easily be generalized to typed expressions.

The basic idea in this generative grammar is to assign to each expression phrase a synthesized attribute holding a representation of its syntax tree — and to each node in this

representation attach a, perhaps void, interpretation procedure (represented as a function). This will allow programs to activate or interpret a given phrase, decompose a phrase, and maybe interpret parts of it or simply pass it on as a data value.

A phrase is activated or interpreted by means of an explicit operator described as follows.

⟨exp↓G↑*value*⟩::= **evaluate** ⟨phrase↓G↑⟨*interp*, *tree*⟩⟩
where *value* = *interp*(*tree*)

It appears in this rule that an expression denotes a value and that phrases denote particularly decorated syntax trees and that the 'evaluate' operator converts phrases to expressions. (So in a certain sense, with a rather specialized meaning of the word 'meaning', an interpretive semantics of this sort is a special case of a compositional semantics).

The rule for operator definition is as follows.

⟨dcl↓G↑*new-rule*⟩::= **new operator** ⟨parameter↓G↑$name_1$⟩ ⟨symbol↓G↑*operator*⟩
⟨parameter↓G↑$name_2$⟩ = ⟨procedure↓$G \cup \{par_1, par_2\}$↑*interp-code*⟩

where

par_1 = ⟨phrase↓G'↑⟨proc($name_1$), tree($name_1$)⟩⟩::= $name_1$
par_2 = ⟨phrase↓G'↑⟨proc($name_2$), tree($name_2$)⟩⟩::= $name_2$
new-rule = ⟨phrase↓G'↑⟨*interp*, *operator*(⟨p_1, t_1⟩, ⟨p_2, t_2⟩)⟩⟩::=
⟨phrase↓G'↑⟨p_1, t_1⟩⟩ *operator* ⟨phrase↓G'↑⟨p_2, t_2⟩⟩

where *interp* is a function derived from *interp-code*

The intermediate representation of the interpreter, *interp-code*, may be some term that contains occurrences of the placeholders, proc($name_1$), etc.

5.3 Relation to other work

I have not seen any other general mechanism applied to the description of extensible language semantics, the semantics of declared extensible languages presented in the literature has been described either informally or in terms of their implementation. Mason's dynamic template translators, (Mason, 1987), is a generalization of syntax-directed translation schemes for extensible languages, cf. section 4.5. In principle, the output produced by a translation scheme may be used to describe semantics. However, there is no interaction between the output strings produced for given input strings and the mechanisms for creating new rules. Hence it is impossible to collect the semantic content of a declaration for representation in new, generated rules. Here I will discuss how semantics is specified within various, more or less, extensible languages and other related systems.

In the ECL system, (Wegbreit, 1970), the meaning of a new language construct is defined by means of an interpretation procedure: Together with the syntactic specification, the user must also supply a definition of a data structure for representation of abstract syntax trees and an interpretation procedure that refers to this representation. There is no syntactic coupling between these three levels, it is the user's responsibility to ensure consistency. I also see a problem in ECL due to its dynamic type concept — data types are treated as data objects in ECL — together with a static extension mechanism: The interpretation procedures tend to include extensive and explicitly programmed checking of the types of their arguments.

Several extensible languages and programming systems have been developed based on the so-called compiler-compiler described in (Brooker et al., 1963). In the two systems described by Napper and Fisher, (1976, 1980), ALEC and RCC, semantics is specified in terms of compilation: The semantic specification for a construct is an abstract description of how to compile it. The ELSA system, (Lindsey, 1983), provides for an interesting combination of compositional and interpretive semantics specifications. The language

includes specialized pattern matching operators for the analysis and synthesis of program phrases: The evaluation or execution or a formal argument (or any other, denoted phrase) is specified simply by mentioning it, just as in a traditional procedural language; if that does not suffice, the argument may be decomposed or transformed and perhaps later executed. An example in the paper, (Lindsey, 1983), shows these facilities applied in order to ensure an efficient compilation of arithmetic expressions — and that in a rather clean and abstract way.

Abstraction mechanisms in the languages of the Algol family, Algol60, Algol68, Pascal, Ada, etc., reflect with no exception the compositional view. A procedure declaration, for example, consists of a syntactic pattern, the procedure heading, and a semantic specification, the procedure body. The execution of a procedure is understood as the execution of the instructions in the procedure body with a suitable substitution of actual parameters for their formal counterpart, it is impossible to introduce any radically new concept by a procedure declaration.

LISP, on the other hand, allows for both compositional and interpretive semantic specifications for new function calls in that a LISP function may receive its arguments either as their (evaluated) values or in their unevaluated, textual form. Unevaluated parameters can be made subject of arbitrary transformation using the full power of the LISP language and then interpreted by a call of the eval function. The following chapter includes a generative grammar describing the syntax and semantics of LISP.

Various generators of compilers, editors, user application programs, (e.g., the so-called 4th generation systems), prototype interpreters, etc., can be viewed as extensible languages: Each such system is characterized by a specialized input language for writing language definitions — and the system generates some representation of the language. The semantic parts of a language specification are for the most part of such generators expressed compositionally in some fixed semantic meta-language. This may be machine language, calls of screen management routines, data base query languages, or some other abstract machine language. The SIS system, (Mosses, 1978), for example, is intended for prototype implementation of programming languages whose semantics is specified solely in terms of the λ-calculus or auxiliary function symbols whose meaning is specified by λ-expressions. The CERES compiler generation system, (Christiansen, Jones, 1983), produces recursive-descent compilers from denotational semantic specifications based on a simplified, semantic language called imperative semantics, (Christiansen, 1981).

Comparing to Standish', (1975), classification of language extensions due to the character of their semantic specification, his metaphrase extensions correspond to my interpretive semantics. Standish points out two other classes covered by compositional semantics, paraphrase and orthophrase extensions. The former concerns semantic specifications in terms of the concepts found inside a given language, as is the case with traditional procedure abstraction. 'Orthophrase' refers to extensions whose semantics needs to be expressed compositionally in terms of some 'external' language, for example calling machine language routines from a high-level language.

6 A comment on Lisp

LISP, (McCarthy, 1960), is one of the earliest, if not the first, language with a clear stress on extensibility. In LISP, a user can define new functions which, then, cannot be distinguished from the primitive LISP functions. This is also reflected in the semantic specification of LISP, see (McCarthy, 1960), in which the meaning of any function call is described uniformly in terms of the universal functions, eval and apply. In fact, this description of LISP is probably the existing language description which has most in common with a generative grammar concerning a unified treatment of existing and user-defined language constructs.

Furthermore, LISP gains its flexibility from the symmetry between programs and data together with the explicit representation of the language semantics inside the language itself, in the shape of the eval function. The predominant convention in the various dialects of the language has, until recently, been dynamic binding and function definition which also contributes to the frightening power of LISP.

In this chapter, I give a generative grammar that specifies the full (and intertwined!) syntax and semantics of a LISP-like language that includes all the above mentioned aspects and also user-defined macros. However, it will be practical to extend the model of generative grammars slightly. Until now, the attribute values have been determined as *functions* of other attributes, but in order to describe the introspective function, eval, it is necessary to use relations for this purpose, more specifically the syntactic derivation relation, denoted $\Rightarrow^*$.

In fact, the generative grammar for LISP reveals a close correspondence between LISP's eval functions and the generative grammars' syntactic derivation relation. In chapter 7, the derivation relation — and hence eval — is modelled by an introspective prove predicate in Prolog.

In order to describe the dynamic nature of declarations, the execution state is identified with the language attribute.

The language includes the following forms.

atoms
: atomic values and identifiers,

nil
: an atom whose evaluated value is nil,

(prog2 – –)
: evaluate two expressions in sequence,

(if – – –)
: conditional expression,

(quote –)
: return the argument unevaluated,

(eval –)
: evaluate the value of the argument,

(defun *fn-name* (*param-name*) *fn-body*)
: define a new function, the text for function body is not evaluated,

(defmacro *macro-name* (*param-name*) *macro-body*)
: define a new macro, the text for the function body is not evaluated,

(primitive$_i$ – . . .)
various primitive functions or special forms, the latter receiving their arguments in an unevaluated form.

The addition of an assignment operator, i.e., setq, will be discussed later.

The grammar has three nonterminal symbols, atoms, expressions and quoted-text. The difference between expressions and quoted text is that the former denote their evaluated values whereas the latter denotes itself. The meaning of the attached attributes is indicated as follows.

⟨atom↓*language*↑*name of atom*⟩

⟨quoted-text↓*language*↑*the text itself*⟩

⟨expression↓*language*↑*value*↑*a possible changed language*⟩

The rules in the generative grammar are as follows.

Atoms denote nothing but themselves, for example:

⟨atom↓G↑hubahop⟩::= hubahop

Quoted text also denotes itself:

⟨quoted-text↓G↑($text_1$ . . . $text_n$)⟩::=
(⟨quoted-text↓G↑$text_1$⟩ . . . ⟨quoted-text↓G↑$text_n$⟩)

⟨quoted-text↓G↑*name*⟩::= ⟨atom↓G↑*name*⟩

I will consider the primitive expressions first since any other expressions are instances of one of the two patterns, one for functions, and one for special forms.

⟨expression↓G↑val_x↑G_x⟩::=
(function$_i$ ⟨expression↓G↑val_1↑G_1⟩ . . . ⟨expression↓G_{n-1}↑val_n↑G_n⟩)
where the relation between
⟨val_x, G_x⟩ and ⟨val_1, . . . ,val_n, G_n⟩
is specific to the given function.

⟨expression↓G↑val_x↑G_x⟩::=
(special$_i$ ⟨quoted-text↓G↑$text_1$⟩ . . . ⟨quoted-text↓G↑$text_n$⟩)
where the relation between
⟨val_x, G_x⟩ and ⟨$text_1$, . . . , $text_n$, G⟩
is specific to the given special form.

Sequencing, the nil and quote expressions are straightforward examples of the above:

⟨expression↓G↑val_2↑G_2⟩::=
(prog2 ⟨expression↓G↑val_1↑G_1⟩ ⟨expression↓G_1↑val_2↑G_2⟩)

⟨expression↓G↑nil↑G⟩::= nil

⟨expression↓G↑*text*↑G⟩::= (quote ⟨quoted-text↓G↑*text*⟩)

In a conditional expression, the first subexpression is evaluated and depending on its value, one out of the two following expressions is selected for evaluation. The text in the remaining expression should not be considered at all, it may contain any correctly paranthesized nonsense. The linguistic equivalent to the eval function, the derivation relation, is necessary in order to characterize this in a generative grammar rule.

⟨expression↓G↑val_x↑G_x⟩::=
(if ⟨expression↓G↑val_1↑G_1⟩ ⟨quoted-exp↓G↑$text_{then}$⟩ ⟨quoted-text↓G↑$text_{else}$⟩)
where, if val_1 = nil then
⟨exp↓G_1↑val_x↑G_x⟩ ⇒ * $text_{else}$,
or, otherwise
⟨exp↓G_1↑val_x↑G_x⟩ ⇒ * $text_{then}$.

The effect of a function definition is to add to the grammar a new rule for application of the declared function. This rule should describe the meaning of an application as an evaluation of the function body in the 'application-time' environment extended with a suitable rule for the function parameter.

⟨expression↓G↑*fn-name*↑G∪{*fn-rule*}⟩::=
(defun ⟨atom↓G↑*fn-name*⟩ (⟨atom↓G↑*param-name*⟩)
⟨quoted-text↓G↑*fn-body*⟩)
where *fn-rule* =
⟨expression↓G'↑*result-value*↑G_2'\{*param-rule*}⟩::=
(*fn-name* ⟨expression↓G'↑*actual-param*↑G_1'⟩)
where
⟨expression↓G_1'∪{*param-rule*}↑*result-value*↑G_2'⟩ ⇒ * *fn-body*
and *param-rule* =
⟨expression↓G''↑*actual-param*↑G''⟩::= *param-name*

Macro definitions are analogous to function definitions, except that the actual parameter (in the generated rule) should not be evaluated and that parameter transmission is performed by textual substitution.

⟨expression↓G↑*macro-name*↑G∪{*macro-rule*}⟩::=
(defmacro ⟨atom↓G↑*macro-name*⟩ (⟨atom↓G↑*param-rule*⟩)
⟨quoted-text↓G↑*macro-text*⟩)
where macro-rule =
⟨expression↓G'↑*result-value*↑G_1'⟩::=
(macro-name ⟨quoted-text↓G'↑*param-text*⟩)
where
⟨expression↓G'↑*result-value*↑G_1'⟩ ⇒ * *expanded-text*
where *expanded-text* is derived from *macro-text* in that each
each occurrence of *param-name* is replaced by *param-text*.

It is left as an exercise for the reader to implement the indicated textual substitution by means of a few generative grammar rules.

The final rule in the grammar is for the eval function.

⟨expression↓G↑val_2↑G_2)::= (eval ⟨expression↓G↑val_1↑G_1⟩)
where
⟨expression↓G_1↑val_2↑G_2⟩ ⇒ * val_1

The LISP description given above contains serious ambiguities concerning functions or variables with identical names. This can be remedied in a representation-based generative grammar (definition 4, chapter 3): The grammar is represented in a state having two components,

- a table mapping function or macro names to their definitions, and
- a stack of variable bindings.

The abstraction function, i.e., the function which yields back a grammar, should ignore all but the top-most bindings to a given variable name. This representation provides a suitable medium for specifying a more accurate behaviour of the binding mechanisms:

- Adding a new function or macro is described as the addition of a new entry to the table overriding any previous definition for the given name.
- Actual parameter bindings are added to the top of the stack and removed at function exit.
- The explicit assignment of a value to a variable, i.e., setq, should change the top-most binding for the given variable name; if none such exists, i.e., it is a global variable, a binding is added at the bottom of the stack.

A yet more detailed representation of the execution state based on structure sharing would, furthermore, allow for a description of the LISP functions rplaca and rplacd.

7 The relation to logic

In this chapter, I consider the relation between the generative grammar formalism and the logic programming language Prolog, see, e.g. (Clocksin, Mellish, 1984).

The core of Prolog is the subset of first-order logic known as Horn clauses which — as is well-known — are closely related to the syntax of non-extensible languages. Furthermore, the full Prolog language is well-suited for writing programs that manipulate programs — which suggests that Prolog might be useful with respect to a class of grammars in which grammars themselves are treated as data objects.

It will appear that any generative grammar can be rewritten in a quite straightforward manner as a Prolog program. This transformation is valuable for prototype implementation of languages described by generative grammars and, furthermore, useful representation-based grammars can be expressed quite naturally using Prologs non-logical facilities. Section 7.1 is concerned with the representation of 'pure' generative grammars; section 7.2 discusses various kinds of representation-based grammars.

7.1 Generative grammars in extended Horn clauses

To begin with the simplest case, we observe the corresponding between Horn clause logic and non-extensible languages. Consider, for example, the following context-free grammar.

⟨exp⟩::= ⟨exp⟩ + ⟨exp⟩
⟨exp⟩::= ⟨exp⟩ * ⟨exp⟩
⟨exp⟩::= 2

It is equivalent to the following Horn clause program, where equivalence means that the syntax trees of the former are isomorphic to the proof trees of the latter.

```
exp:- exp, exp.
exp:- exp, exp.
exp.
```

Deransart and Maluszynski, (1985), have demonstrated the correspondence between Horn clause programs and attribute grammars whose attributes are terms over free algebras, again here, the associated classes of trees are isomorphic. Furthermore, additional predicates can be used to calculate more interesting attribute relations, as observed, among others, by Abramson, (1984). Consider, for example, an attribute grammar for the simple expression language having a synthesized attribute which describes the expression values. It may be written as a Horn clause program as follows.

$exp(v) :- exp(v_1), exp(v_2), v \text{ is } v_1 + v_2.$
$exp(v) :- exp(v_1), exp(v_2), v \text{ is } v_1 * v_2.$
exp(2).

(Here, the operator 'is' should be understood as a predicate and not as its imperative Prolog counterpart).

Considered as a procedural Prolog program, however, these clauses are not very interesting, but they may serve as a parsing program if the concrete, syntactic strings are merged into the predicates. For example as follows.

```
exp(E, V):- decomp(E, [E1,+,E2]),
            exp(E1 ,V1), exp(E2, V2), V is V1 + V2.
```

where `decomp` is defined such that, for example,

```
decomp(E, [E1,+,E2])
```

is equivalent to

```
append(E1, [+|E2], E).
```

The use of `append` or its syntactically dressed up companion, `decomp`, is rather inefficient, but that is out of interest in this chapter. (The definition of `decomp` written in a rather imperative Prolog style can be found in Christiansen, 1986c).

I will now consider how extensible languages — in the appearance of generative grammars — can be represented in Prolog. Bowen and Kowalski, (1982), has argued that a representation of Prolog within itself, by means of a meta-predicate

`prove(` *goal, program*)

is essential in order to provide for a more flexible, logic programming style.[1)]

Using this meta-predicate, it is easy to express the fact that different, syntactic phrases are generated by different grammars. Consider, for example, the following generative grammar rule for a traditional block construct.

$\langle block{\downarrow}G \rangle ::=$ **begin** $\langle dcl{\downarrow}G{\uparrow}G_1 \rangle$; $\langle stm{\downarrow}G \cup G_1 \rangle$ **end**

It is easily translated into the following Prolog rule.

```
block(B, G):- decomp(B, [begin, D, ;, S, end]),
              prove( dcl(D, G, G1), G),
              append(G1, G, GG),
              prove( stm(S, GG), GG).
```

In order to prove that a string, `B`, can be generated as a block in a given grammar, `G`, the expected declaration and statement parts, `D` and `S`, are extracted from `B`. Now, `D` is proved to be a declaration in the given program, `G`; `G` is also given as a parameter to the `dcl` predicate such that further, explicit manipulations of the grammar can be described. The synthesized attribute of the declaration, `G1`, represents a grammar, i.e., a program, which is a list of goals. The string, `S`, is now proved to be a statement in the extended grammar or program, `GG`. In other words, the correspondence between the derivation relation, $\Rightarrow^*$, and the `prove` predicate is such that

1. The fact that `prove` is true for given goal and program, the latter represented as a list of Prolog rules, is intended to mean that the goal is true according to the given program. The `prove` predicate can be defined in a reasonable manner in Prolog, cf. (Bowen, Kowalski, 1982); an implementation of `prove` appears in appendix B.

⟨*nonterm*↓*G* . . .⟩ ⇒ * *text*

if and only if

`prove(` *nonterm*(*text*, *G*, ...), *G*),

assuming a straightforward transformation of nonterminal symbols into predicate symbols as indicated above; the ellipsis stand for possible attributes apart from the language attribute.

It should be clear that this transformation into a (meta-) Prolog program can be applied to any generative grammar whose attribute expressions apply functions that can be implemented in Prolog — and as is well-known, these functions cover the spectrum of computable functions, see, for example, (Shapiro, 1984).[1)]

The initial call of a parsing program derived from a generative grammar in this way is as follows.

```
G = [ rules for the initial grammar,
      the decomp predicate,
      the prove predicate,
      additional predicates used for the attribute expressions ],
prove( program( program-text, G, ... other attributes), G).
```

The `prove` predicate provides for an implementation of the reflexive use of the derivation relation in the LISP grammar given in chapter 6. For example, the generative grammar rule for the eval function can be written as follows; Prolog's list structures are used to indicate the syntactic phrase structure so no extra predicate for syntactic decomposition is needed.

```
exp( [eval, E1], G, V2, G2):-
    exp( E1, G, V1, G1),
    % where
    prove( exp(E2, G1, V2, G2), G1).
```

The appendix gives the full text of the Prolog representation of the LISP grammar together with an implementation of `prove`; this Prolog program constitutes a slow, but effective, LISP interpreter.

7.2 Representation-based grammars in Prolog

The imperative aspects of Prolog, its deterministic proof strategy, the cut operator (usually denoted '!'), etc., give rise to an implementation of representation-based generative grammars (definition 4, chapter 3).

Consider, for example, a block-structured language whose basic block-construct is as described in section 7.1. Assume, furthermore, that the rules for declarations are such that a variable, 'n' say, is described by a generated rule of the form,

```
variable(V):- decomp( V, [n]), !.
```

This rule will inhibit any other rule for a variable named 'n' since new rules are added to the beginning of existing programs. If the grammatical rule, furthermore, contains type information and other attributes, any reference to their values should appear after the cut symbol, e.g.,

```
variable(V, Type, Meaning):- decomp(V, [n]), !,
                             Type = integer,
                             Meaning = yksikaksikolme.
```

1. However, generative grammars which allow for recursively defined attribute values may cause problems.

Hence, the indicated technique provides for a characterization of traditional scope conventions for block-structured languages by means of a representation-based generative grammar.

In certain cases, the cut symbol is unnecessary: Consider the LISP description in Prolog considered in 7.1 and whose full text is given in appendix B. If only the first value produced by the Prolog interpreter for a given expression is accepted, the Prolog program appear as a representation-based generative grammar describing the scope rules we would expect for LISP!

In the special case of block-structured languages, it is possible to replace the inefficient meta-level interpretation by the use of Prolog's imperatives for dynamic assertion and retraction of rules. In the paper (Christiansen, 1986c), I have introduced a logical operator that allows for a structured use of these imperatives. This operator introduces a new form of goals.

goal `assuming` *list of rules*

An instance of this form is intended to be provable according to a logical program, P, if *goal* is provable in P extended with *list of rules*. The familiar block construct can now be described as follows.

```
block(B):- decomp(B, [begin, D, ;, S, end]),
           dcl( D, Extension),
           stm( S) assuming Extension.
```

The declaration generates, as its synthesized attribute, a list of new rules which is available, together with the already existing program, for the proof of the statement goal. And again here, the imperative aspects of Prolog may be used in order to describe the more detailed scope rules.

This method provides, in fact, for a concise notation for generative grammars for block structured languages: The existence of a language attribute is made implicit except in those rules that really manipulate the language. In (Christiansen, 1986c), additional properties of the full Prolog language are used in order to characterize other aspects of programming language syntax.

8 Summary

In this paper, I have presented the generative grammar formalism which is a tool for the specification of the syntax and semantics of extensible languages. Generative grammars are a simple generalization of attribute grammars: The possible derivations from a given attributed nonterminal symbol are defined relative to the grammar found in a specialized, inherited attribute — as opposed to the attribute grammars' fixed and 'global' set of syntactic rules. It appears that the creation and management of new syntactic rules can be described quite naturally in the attributes and their defining expressions.

Generative grammars have been demonstrated for a wide range of extension mechanism and work well for both the static and dynamic bindings, the latter illustrated by a description of LISP in which syntax and program execution are coupled tightly together.

In the same way as attribute grammars are well-suited for describing semantics of (almost) context-free languages, semantic descriptions for extensible languages can be merged into generative grammars. Two 'styles' of semantics descriptions have been discussed, compositional semantics which is a generalization of denotational semantics and the more powerful interpretive semantics. It appears that textual substitution and renaming may be useful techniques in the generation of concise semantic descriptions from the specifications in the given programming language. The grammar describing LISP's intertwined syntax and semantics serves as an example of a non-trivial semantic description.

Furthermore, I have shown how in principle any generative grammar may be transformed in a quite straightforward manner into a Prolog program. These Prolog programs may serve as (quite slow) prototype implementations of extensible languages — and the construction exposes a relation between extensible languages and an extensible logic based on 'meta-interpretation'.

Generalizations

The basic generative grammar formalism is sufficient from a theoretical point of view, however, a few pragmatic extensions of this model may be useful.

Representation-based generative grammars.

- The specialized inherited attribute, the language attribute, is here a representation of a grammar. Hence generated grammars may be described and elaborated in terms of that representation. For example, a stack-based based protocol is useful in order to describe scope rules in which one syntax rule temporarily may override other rules. Such scoping principles are difficult to describe in the pure generative grammar framework in which a grammar is understood as a *set* of rules.

Reflexive use of the derivation relation.

- In a normal generative grammar, the attributes are determined as functions of other attributes. However, explicit reference to the derivation relation in the generative grammar rules may give rise to quite natural descriptions of reflexive or introspective language constructs such as LISP's eval function.

Concluding remarks

The generative grammars presented here provide a framework for the understanding of extensible languages — in the shape of a descriptive tool for their syntax and semantics. However, the formalism still needs some refinement in order to give a satisfactory treatment of recursive declarations, cf. section 3.4.

It would have been desirable with an accompanying catalogue of implementation techniques for languages described in this way. However, no systematic treatment of this topic has been given yet. The report, (Christiansen, 1986b), excerpts published in (Christiansen, 1986a), describes the problems in the adaptation of traditional parsing and compilation techniques at an overall and theoretical level. In (Christiansen, 1988), I have summarized my experience with implementation of generative grammars which includes various experiments in incremental editing.

Appendix A — An example grammar

This appendix shows a generative grammar for a Pascal-like language. Each expression type is represented by its own nonterminal symbol and new types can be introduced by means of record declarations. The language, furthermore, allows for the declarations of new statement and expression forms.

Built-in statement forms

The language includes statement sequencing, a block-construct with non-recursive declarations, and a conditional statements. Assignment statements are described later on.

⟨stm↓G⟩::= ⟨stm↓G⟩ ; ⟨stm↓G⟩
⟨stm↓G⟩::= **begin** ⟨dcl↓G↑G_1⟩ ; ⟨stm↓$G \cup G_1$⟩ **end**
⟨stm↓G⟩::= **if** ⟨bool↓G⟩ **then** ⟨stm↓G⟩ **else** ⟨stm↓G⟩ **fi**

Built-in types and operators

The set of operators common to all expressions consists of

- assignment statement,
- the use of variables in expressions,
- a conditional expression,
- an equality test, and
- a **valof** expression which specifies a value in terms of a statement; the result is returned by a **result is** statement.

In order to describe these operators for the built-in types and to generate versions for new types, the following notational shorthand is used. For any term, *type*, (which is expected to represent a nonterminal symbol), the notation

expression-tools(*type*)

refers to the following set of rules.

{⟨stm↓G'⟩::= ⟨*type*-variable↓G'⟩:= ⟨*type*↓G'⟩,
⟨*type*↓G'⟩::= ⟨*type*-variable↓G'⟩,
⟨*type*↓G'⟩::= **if** ⟨bool↓G'⟩ **then** ⟨*type*↓G'⟩ **else** ⟨*type*↓G'⟩ **fi**,
⟨bool↓G'⟩::= ⟨*type*↓G'⟩ = ⟨*type*↓G'⟩
⟨*type*↓G'⟩::= **valof** ⟨stm↓$G' \cup$ {*exit-rule*}⟩ **end**
where *exit-rule* = ⟨stm↓G''⟩::= **result is** ⟨*type*↓G''⟩
}

Note that this implies the existence of constructed nonterminals such as 'integer-variable' for variables of type integer.

The language features two basic types, booleans and integer numbers:

⟨type↓G↑bool⟩::= bool
⟨type↓G↑integer⟩::= integer

These two rules allow the basic types to be used in declarations of, for examples, variables and new types. To each of these types is associated a set of type specific rules, for example:

⟨bool↓G⟩::= true

⟨bool↓G⟩::= ⟨bool↓G⟩ **and** ⟨bool↓G⟩

⟨integer↓G⟩::= 1

⟨integer↓G⟩::= ⟨integer↓G⟩ + ⟨integer↓G⟩

As for any type, we assume the following.

expression-tools(bool)

expression-tools(integer)

Identifiers

For simplicity, each identifier is assumed to be defined by its own rule, for example:

⟨identifier↓G↑huba⟩::= huba

Declarations

A declaration may be a sequence of declarations. The first part of such a sequence is made available for the subsequent declarations.

⟨dcl↓G↑$G_1 \cup G_2$⟩::= ⟨dcl↓G↑G_1⟩ ; ⟨dcl↓$G \cup G_1$↑G_2⟩

Variables of specified type can be declared as follows.

⟨dcl↓G↑{*new-rule*}⟩::= **var** ⟨identifier↓G↑*name*⟩ : ⟨type↓G↑*typeid*⟩
where *new-rule* = ⟨*typeid*-variable↓G'⟩::= *name*

Type declarations

New types can be declared as records consisting of named fields with specified types. For each new type are generated rules for using the type in other declarations, for creation of values of that type and selection of their components, and a rule describing a 'with' statement. The latter, when applied, will in turns generate specialized selection rules.

⟨dcl↓G↑{*type-rule*, *create-rule*, *with-rule*,
access$_1$, . . ., *access*$_n$}∪expression-tools(*new-type*)⟩::=
type ⟨identifier↓G↑*new-type*⟩ =
record ⟨identifier↓G↑*field*$_1$⟩ : ⟨type↓G↑*type*$_1$⟩,
. . .
⟨identifier↓G↑*field*$_n$⟩ : ⟨type↓G↑*type*$_n$⟩ **end**

where

type-rule = ⟨type↓G'↑*new-type*⟩::= *new-type*

create-rule = ⟨*new-type*↓G'⟩::=
create *new-type* (⟨*type*$_1$↓G'⟩, . . ., ⟨*type*$_n$↓G'⟩)

for i = 1, . . ., n,
access$_i$ = ⟨*type*$_i$↓G'⟩::= ⟨*new-type*↓G'⟩ . *field*$_i$

with-rule =
⟨stm↓G'⟩::= **with** ⟨*new-type*↓G'⟩ **do**
⟨stm↓$G' \cup$ {*quick-access*$_1$, . . ., *quick-access*$_n$}⟩
where, for i = 1, . . ., n,
quick-access$_i$ = ⟨*type*$_i$↓G''⟩ ::= *field*$_i$

Declaration of new statement forms

Instead of traditional procedure declarations, the sample language includes a more general declaration mechanism: New statement forms are described as patterns specifying arbitrary sequences of terminal symbols and parameters which may be expressions or statements. For example, an if-then statement may be defined as follows.

operation if [b: bool] then [s] fi =
  if b **then** s **else** null **fi**

The declaration mechanism is described as follows.

⟨dcl↓G↑{*new-rule*}⟩::=
  operation ⟨pattern↓G↑*syntax-spec*↑*parameter-rules*⟩
    = ⟨stm↓G∪*parameter-rules*⟩
where *new-rule* = ⟨stm↓G'⟩::= *rhs*
  where *rhs* is a copy of *syntax-spec* in which each occurrence
    of the special marker, @CONTEXT, is replaced by
    the attribute variable G'

⟨pattern↓G↑$syntax\text{-}spec_1$ • $syntax\text{-}spec_2$↑$parameter\text{-}rules_1$∪$parameter\text{-}rules_2$⟩::=
  ⟨pattern↓G↑$syntax\text{-}spec_1$↑$parameter\text{-}rules_1$⟩
  ⟨pattern↓G↑$syntax\text{-}spec_2$↑$parameter\text{-}rules_2$⟩

⟨pattern↓G↑*atom*↑∅⟩::= ⟨identifier↓G↑*atom*⟩

⟨pattern↓G↑*nonterm*↑{*parameter-rule*}⟩::=
  [⟨identifier↓G↑*par-name*⟩ : ⟨type↓G↑*par-type*⟩]
where *parameter-rule* = ⟨*par-type*↓G'⟩::= *par-name*
  nonterm = ⟨*par-type*↓@CONTEXT⟩

⟨pattern↓G↑*nonterm*↑{*parameter-rule*}⟩::= [⟨identifier↓G↑*par-name*⟩]
where *parameter-rule* = ⟨stm↓G'⟩::= *par-name*
  nonterm = ⟨stm↓@CONTEXT⟩

Declaration of new expression forms

The declaration mechanism for statement forms may be generalized to cover declaration of operators of any type. The description of such declarations should be merged into the set of operators valid for all expression types as follows.

expression-tools(*type*) =
  { . . .,
   ⟨dcl↓G↑{*new-rule*}⟩::=
     operation ⟨pattern↓G↑*syntax-spec*↑*parameter-rules*⟩ : *type*
       = ⟨*type*↓G∪*parameter-rules*⟩
   where *new-rule* = ⟨*type*↓G'⟩::= *rhs*
     where *rhs* is . . .
  }

Appendix B — Lisp in Prolog

This appendix gives the full text of a Prolog implementation of the generative grammar for LISP presented in chapter 6. The method used for translation of generative grammars into Prolog programs is described in chapter 7.

In order to simplify the prove predicate, the bodies of clauses given to this predicate are written as Prolog lists. The LISP grammar is defined in the predicate named lisp. The 'top-level' predicate, prove_lisp, can be used as a LISP interpreter; the notation used for LISP source code is illustrated in the predicate sample_program.

```
prove_lisp(Text, Result):- lisp(G), prove( [exp(Text, G, Result, _)], G).
sample_program( [prog2,[defun,fak,[n],
                        [if,[equal,n,0],1,[times,n,[fak,[minus,n,1]]]]],
                  [fak,5]
                ]
              ).
lisp( [ latom( A, G, A):-
                  [atomic( A)],
        qtext( Q, G, Atom):-
                  [latom( Q, G, Atom)],
        qtext( [Q | MoreQ], G, [T | MoreT]):-
                  [qtext( Q, G, T),
                   qtext( MoreQ, G, MoreT)],
        exp( [quote, Q], G, Qtext, G):-
                  [qtext( Q, G, Qtext)],
        exp( [], G, [], G):- [],
        exp( t, G, t, G):- [],
        exp( N, G, N, G):- [integer(N)],
        exp( [plus,N1,N2], G, V, G2):-
                  [exp( N1, G, V1, G1),
                   prove( [exp( N2, G1, V2, G2)], G1),
                   V is V1 + V2],
       exp( [minus,N1,N2], G, V, G2):-
                  [exp( N1, G, V1, G1),
                   prove( [exp( N2, G1, V2, G2)], G1),
                   V is V1 - V2],
       exp( [times,N1,N2], G, V, G2):-
                  [exp( N1, G, V1, G1),
                   prove( [exp( N2, G1, V2, G2)], G1),
                   V is V1 * V2],
      exp( [equal, E1, E2], G, V, G2):-
                  [exp( E1, G, V1, G1),
                   prove( [exp( E2, G1, V2, G2)], G1),
                   if( (V1 = V2), (V = t), (V = []))],
     exp( [cons, E1, E2], G, [V1 | V2], G2):-
                  [exp( E1, G, V1, G1),
                   prove( [exp(E2, G1, V2, G2)], G1)],
```

```
            exp( [car, E1], G, V, G1):-
                      [exp( E1, G, V1, G1),
                       V1 = [V |_]],
            exp( [cdr, E1], G, V, G1):-
                      [exp( E1, G, V1, G1),
                       V1 = [_ | V]],
           exp( [prog2, E1, E2], G, V2, G2):-
                     [exp( E1, G, V1, G1),
                      prove( [exp( E2, G1, V2, G2)], G1)],
          exp( [if, Econd, Ethen, Eelse], G, Vx, Gx):-
                     [exp( Econd, G, Vcond, Gcond),
                      qtext( Ethen, G, Tthen),
                      qtext( Eelse, G, Telse),
                      % where
                        if( Vcond = [],
                              prove( [exp( Telse, Gcond, Vx, Gx)], Gcond),
                              prove( [exp( Tthen, Gcond, Vx, Gx)], Gcond))],
         exp( [eval, E1], G, V2, G2):-
                    [exp( E1, G, V1, G1),
                     % where
                       prove( [exp( V1, G1, V2, G2)], G1)],
        exp( [defun,A1,[A2],Q], G, Fn_name, [Fn_rule | G]):-
                   [latom( A1, G, Fn_name),
                    latom( A2, G, Param_name),
                    qtext( Q, G, Fn_body),
                    % where
                      Fn_rule = (exp( [Fn_name,E1_], G_, V_, G4_):-
                                  [exp( E1_, G_, Actual, G1_),
                                   Par_rule =
                                     (exp( Param_name, G__, Actual, G__)
                                                          :- []),
                                   G2_ = [Par_rule | G1_],
                                   prove( [exp( Fn_body, G2_, V_, G3_)],
                                                                  G2_),
                                   remove_rule( Par_rule, G3_, G4_)])
                  ],
       exp( [defmacro,A1,[A2],Q]  . . . left as an exercise
      ]).
prove( [], _).
prove( [First_goal | Rest_goals], Program):-
    member( Clause, Program),          % select a clause
    rename( Clause, Head:- Body),      % rename variables
    First_goal = Head,                 % match first goal with
                                       %   head of selected goal
    append( Body, Rest_goals, New_goals),
    prove( New_goals, Program).
% Escape to globally defined predicates:
prove( [First_goal | Rest_goals], Program):-
   is_defined(First_goal), First_goal, prove( Rest_goals, Program).
```

```
% Renaming of variables in clauses:
rename( Clause, Renamed_clause):-
    asserta( little_green_man( Clause)),
    clause( little_green_man( Renamed_clause), _),
    retract( little_green_man( Clause)), !.
% Globally defined predicates used in the grammar:
is_defined( prove( _, _) ).
is_defined( _ is _ ).
is_defined( _ = _ ).
is_defined( atomic( _ ) ).
is_defined( integer( _ ) ).
is_defined( if( _, _, _) ).
  % Cond, Then, and Else should be calls of prove or elementary
  % tests or assignments.
  if( Cond, Then, Else):-
      Cond, !, Then; Else.
is_defined( remove_rule( _, _, _) ).
  remove_rule( R, [R1 | Prog], Prog):- R == R1, !.
  remove_rule( R, [R1 | Prog], [R1 | Prog1]):-
      remove_rule( R, Prog, Prog1), !.
```

References

Abramson, H., Definite clause translation grammars, *Proceedings of the 1984 International Symposium on Logic Programming*, Atlantic City, pp. 233-240, 1984.

Aho, A.V., Indexed grammars — An extension of context-free grammars, *Journal of the ACM* 15, pp. 647-671, 1968.

Aho, A.V., Nested stack automata, *Journal of the ACM* 16, pp. 383-406, 1969.

Aho, A.V., Johnson, S.C., and Ullman, J.D., Deterministic Parsing of Ambiguous Grammars, *Communications of the ACM* 8, pp. 441-452, 1975.

Bowen, K.A. and Kowalski, R.A., Amalgamating Language and Meta-Language in Logic Programming, *Logic Programming*, Clark, K.L., and Tärnlund, S.Å., eds., Academic Press, pp. 153-172, 1982.

Brooker, R.A., MacCallum, I.R., Morris, D., and Rohl, J.S., The Compiler Compiler, *Annual Review in Automatic Programming* 3, pp. 229-275, 1963.

Christiansen, H., *A new approach to Compiler Generation*, Master's thesis, Aarhus University, Computer Science Department, 1981.

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, 1986a.

Christiansen, H., Parsing and Compilation of Generative Languages, *Datalogiske skrifter* 3, Roskilde University Centre, 1986b.

Christiansen, H., Context-sensitive Parsing in Full Prolog, *Datalogiske skrifter* 5, Roskilde University Centre, 1986c.

Christiansen, H., Programming as language development, *Datalogiske skrifter* 15, Roskilde University Centre, 1988.

Christiansen, H. and Jones, N.D., Control flow in a simple semantics-directed compiler generator, *Proc. Formal Description of Programming Concepts II*, pp. 73-97, North-Holland, 1983.

Chomski, N., Three models for the description of languages, *IEEE Transactions on Information Theory* 2, pp. 113-124, 1956.

Clocksin, W.F and Mellish, C.S., *Programming in Prolog*, second edition, Springer-Verlag, 1984.

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., A Survey on Attribute Grammars, Part III: Classified Bibliography, *INRIA, Rapports de Recherce* 417, 1985.

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

Floyd, R.W., Syntactic analysis and operator precedence, *Journal of the ACM* 3, pp. 316-333, 1962.

Galler, B.A. and Perlis, A.J., A proposal for definitions in Algol, *Communications of the ACM* 10, pp. 204-219, 1967.

Gordon, M.J.C., *The denotational description of programming languages, An introduction*, Springer-Verlag, 1979.

Greibach, S. and Hopcroft, Scattered Context Grammars, *Journal of Computer and System Sciences* 3, pp. 233-247, 1969.

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.

Hoare, C.A.R., An axiomatic basis for computer programming, *Communications of the ACM* 12, pp. 576-583, 1969.

Jensen, K. and Wirth, N., Pascal, User Manual and Report, *Springer Lecture Notes in Computer Science* 18, 1974.

Kasai, T., A Hierarchy between Context-Free and Context-Sensitive Languages, *Journal of Computer and Systems Sciences* 4, pp. 492-508, 1970.

Knuth, D.E., Semantics of Context-Free Languages, *Mathematical Systems Theory* 2, pp. 127-125, 1968.

Lindsey, C.H., Elsa — An extensible programming system, *Programming languages and system design* (ed. Borman, J.), North-Holland, 1983.

Madsen, O.L., On defining semantics by means of extended attribute grammars, *Lecture Notes in Computer Science* 94, pp. 259-299, 1980.

Marcotty, M., Ledgard, H.F., and Bochmann, G.V., A Sampler of Formal Definitions, *Computing Surveys* 8, pp. 191-276, 1976.

Mason, K.P., *Dynamic Template Translators: A Useful Model for the Definition of Programming Languages*, Ph.D. thesis, University of Adelaide, 1984.

Mason, K.P., Dynamic Template Translators — A New Device for Specifying Programming Languages, *Intern. J. Computer Math* 22, 199-212, 1987.

Mayoh, B.H., Attribute grammars and mathematical semantics, *SIAM Journal of Computing* 10, pp. 503-518, 1981.

McCarthy, J., Recursive Functions of Symbolic Expressions and their Computation by Machine, *Communications of the ACM* 3, pp. 184-195, 1960.

McIlroy, M.D., Macro instruction extensions of compiler languages, *Communications of the ACM* 3, pp. 214-220, 1960.

Meyer, A.R., Halpern, J.Y., Axiomatic definitions of programming languages: A theoretical assessment, *Journal of the ACM* 29, pp. 555-576, 1982.

Milne, R. and Strachey, C., *A theory of programming language semantics*, Chapman and Hall, London, 1976.

Mosses, P., SIS — Semantics Implementation System: Reference Manual and User Guide, *DAIMI MD*-30, Aarhus University, 1978.

Mosses, P., Abstract semantic algebras!, *Proc. Formal Description of Programming Concepts II*, pp. 45-70, North-Holland, 1983.

Napper, R.B.E. and Fisher, R.N., ALEC — A user extensible scientific programming language, *Computer Journal* 19, pp. 25-31, 1976.

Napper, R.B.E. and Fisher, R.N., RCC — A user-extensible systems implementation language, *Computer Journal* 23, pp. 212-222, 1980.

Naur, P., Revised report on the algorithmic language Algol60, *Communications of the ACM* 6, pp. 1-17, 1963.

Rosenkrantz, D.J., Programmed grammars and classes of formal languages, *Journal of the ACM* 16, pp. 107-131, 1969.

Shapiro, E.Y., Alternation and the computational complexity of logic programs, *Journal of Logic Programming* 1, pp. 19-33, 1984.

Sintzoff, M., Existence of a van Wijngaarden syntax for every recursively numerable set, *Ann. Soc. Scientifique de Bruxelles*, pp. 115-118, 1976.

Turner, J.S., *Unambigous definition mechanisms for extensible programming languages*, Ph.D. Thesis, Manchester, 1979.

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 of the algorithmic language ALGOL 68, *Acta Informatica* 5, pp. 1-236, 1975.

Watt, D.A. and Madsen, O.L., Extended Attribute Grammars, *DAIMI PB*-105, Computer Science Department, Aarhus University, 1979.

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

Wirth, N. and Weber, H., Euler: a generalization of Algol and its formal definition: Part 1, *Communications of the ACM* 9, pp. 12-23, 1966.