ELSEVIER Theoretical Computer Science 142 (1995) 3–26

Theoretical Computer Science

Precedences in specifications and implementations of programming languages

Annika Aasa*

Department of Computer Sciences, Programming Methodology Group, Chalmers University of Technology, S-412 96 Göteborg, Sweden

Abstract

Although precedences are often used to resolve ambiguities in programming language descriptions, there has been no parser-independent definition of languages which are generated by grammars with precedence rules. This paper gives such a definition for a subclass of context-free grammars. The definition is shown to be equivalent to the implicit definition an operator precedence parser gives.

A problem with a language containing infix, prefix and postfix operators of different precedences is that the well-known algorithm, which transforms a grammar with infix operator precedences to an ordinary unambiguous context-free grammar, does not work. This paper gives an algorithm that works also for prefix and postfix operators, and the correctness of it is proved. An application of the algorithm is also presented.

1. Introduction

Precedences are used in many language descriptions to resolve ambiguities. The reason for resolving ambiguities with precedences, instead of using an unambiguous grammar, is that the language description often becomes shorter and more readable. An unambiguous grammar which reflects different precedences of operators usually contains a lot of nonterminals and single productions. Consider, for example, an ambiguous grammar for simple arithmetic expressions and the unambiguous alternative.

$$\begin{array}{llllll} E ::= & E+T & T ::= & T*F & F ::= & \text{int} \\ & \mid E-T & & \mid T/F & & \mid (E) \\ & \mid T & & \mid F & & \end{array}$$

*E-mail: annika@cs.chalmers.se.

$$
\begin{aligned}
E ::=\ & E+E \\
\mid\ & E-E \\
\mid\ & E*E \\
\mid\ & E/E \\
\mid\ & \texttt{int} \\
\mid\ & (E)
\end{aligned}
$$

If the language contains also prefix and postfix operators, then the unambiguous grammar will be surprisingly large.

If a language has user-defined operators, as, for example, ML [15] and PROLOG [20] it is also convenient to use precedences. When a new operator is introduced, the grammar is augmented with a new production, and it is hard to imagine how a user would be able to indicate where to place this production in an unambiguous grammar with different nonterminals.

When dealing with precedences, at least two questions arise. First, although precedences are used in many situations, there is no adequate definition of what it means for a production in a grammar to have higher precedence than another production. Precedences are only used to guide which steps a parser would take when there is an ambiguity in the grammar [3, 9, 19, 21]. It is not always easy, given an ambiguous grammar and a set of disambiguating precedence rules, to decide if a parse tree belongs to the language. The second question is if it is possible to transform a grammar with precedence rules to an ordinary context-free grammar. This is surprisingly complicated for grammars containing prefix and postfix operators of different precedences.

For a subclass of context-free grammars, we will give a parser-independent definition of precedences and an algorithm which transforms a grammar with precedences to an unambiguous context-free grammar.

2. Distfix grammars and precedence

Let us first define what kind of grammars we will consider. In the definition op stands for an arbitrary operator word, in analogy with int and id.

Definition 1. A *distfix grammar* is a grammar of the form

$$
\begin{array}{lllll}
E ::= & E\ \texttt{op}\ E & \mid\cdots\mid & E\ \texttt{op}\ E \cdots \texttt{op}\ E & \text{(infix distfix operators)} \\
\mid & \texttt{op}\ E & \mid\cdots\mid & \texttt{op}\ E \cdots \texttt{op}\ E & \text{(prefix distfix operators)} \\
\mid & E\ \texttt{op} & \mid\cdots\mid & E\ \texttt{op} \cdots E\ \texttt{op} & \text{(postfix distfix operators)} \\
\mid & \texttt{op} & \mid\cdots\mid & \texttt{op}\ E\ \texttt{op} \cdots E\ \texttt{op} & \text{(closed distfix operators)} \\
\mid & \texttt{int} & & & \\
\mid & \texttt{id} & & &
\end{array}
$$

where an initial operator word does not work also as a subsequent operator word and no whole sequence of operator words are an initial sequence of operator words of another operator.

We can divide distfix operators into five kinds: left associative infix distfix, right associative infix distfix, prefix distfix, postfix distfix and closed distfix. We will sometimes use AE as a shorthand for all atomic expressions such as integers and identifiers.

An example of a prefix distfix operator is if–then–else. As examples of what the extra requirements imply we consider which productions are allowed if the following production is already in the grammar:

E ::= if E then E else E

The following productions are then illegal:

E ::= if E then E

E ::= olle E if E erik

We will here concentrate on the special case with infix, prefix and postfix operators but the ideas can easily be extended to include distfix operators, and we will indicate how that can be done. The requirements on the operators when we only consider infix, prefix and postfix operators mean that all operators must be distinct.[1]

The requirement that distfix grammars only have one nonterminal is not as hard as it seems. In many language descriptions, precedences are used to resolve ambiguity in just one part of the language and that part can be described by a grammar with only one nonterminal. The same ideas of defining precedences can also be extended to more general grammars as shown in [2].

Definition 2. A *precedence grammar* is a distfix grammar together with precedence rules.

With precedence rules we mean both precedence and associativity rules. We will denote precedence grammars as follows.

```
E ::= $E    3
    | E+E   2  (left associative)
    | #E    1
    | int
```

[1] To allow both unary and binary minus in a language we may assume that the lexical analyzer translates them to different operators.

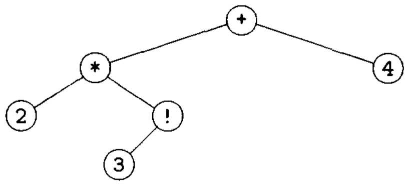
The precedences are given as numbers together with the productions. For these simple grammars we can just as well say that it is the operators that have precedence. The precedence of an operator *op* will be denoted P(*op*). Operators of different kinds are not allowed to have the same precedence. We do not, for example, allow a prefix operator to have the same precedence than a postfix operator. We let the variable H range over all precedence grammars which satisfy the requirements above.

We use the convention that a production with higher precedence has less binding power than one with lower precedence. Thus, for the usual arithmetic operators the addition operator $+$ has higher precedence than the multiplication operator $*$. This convention is used, for example, in PROLOG [20] and OBJ [12]. This convention is unusual, most other languages use the opposite convention, but we have chosen it to make the algorithm in Section 4.1 and the proof of it more clearer.

Since precedences have to do with structure we have to consider parse trees or syntax trees instead of strings when we talk about which language a precedence grammar defines. We will use syntax trees and we will, for example, picture the derivation

$$E \to E+E \to E*E+E \to E*E!+E \to^* 2*3!+4$$

as



Note that the sentence can easily be obtained by flatting the syntax tree. A *syntax tree for an operator* is a syntax tree with that operator as root.

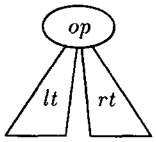
3. Definition of precedence and associativity

One obvious question to ask is which language we define with a precedence grammar. The language is of course a subset of the language generated by the ambiguous grammar without precedence rules. The precedence rules throw away some parse trees. We will call the parse trees we keep *precedence correct.*

It is unsatisfactory to define the precedence correct trees in terms of a specific parsing method. A specification of a language should not involve a method to recognize it, because if the language is defined by one parsing method it could be hard to see if a parser which uses another method is correct.

We will define a predicate Pc_H which given a precedence grammar H defines the precedence correct trees. So, $\mathsf{Pc}_H(t)$ holds if and only if the syntax tree t is correct according to the disambiguating rules in the grammar H. The predicate is defined in such a way that syntax trees built by an operator precedence parser [4, 11] are precedence correct. This and the converse, i.e. that every precedence correct tree can be recognized by an operator precedence parser is proved in a later section.

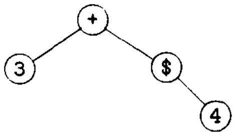
Let us first make some reflections. A syntax tree with an infix operator as root has the following form:



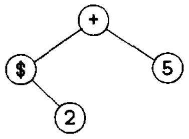
If it would be precedence correct, both the subtrees *lt* and *rt* must of course be precedence correct. Furthermore, there must be some requirements involving the precedence of the root operator. For languages with only infix operators it is enough to look at the precedences of the roots of the subtrees. They must be less than the precedence of the root. This is however not enough if the language contains also prefix and postfix operators. Consider the precedence grammar

E ::=	$\$E$	3	
		$E+E$	2	(left associative)
		$\#E$	1	
		int		

Are the following syntax trees precedence correct?



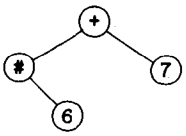
3+$4



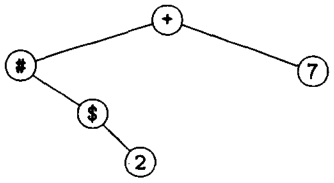
$2+5

We want to consider the left syntax tree as precedence correct but not the syntax tree to the right. This illustrates that prefix operators with higher precedence than an infix operator must be allowed to occur in the right subtree.

Furthermore, consider the two syntax trees below, generated from the same grammar:



#6+7



#$2+7

We want to consider the left syntax tree as precedence correct but not the syntax tree to the right. This illustrates that even if the precedence of the root operator of a subtree is less than the precedence of the whole tree, the syntax tree need not be precedence correct. To solve this problem we introduce two different kinds of precedence weights of a syntax tree, the left weight, Lw, and the right weight, Rw. Prefix operators have precedence only to the right, postfix operators only to the left and infix operators in both directions. The weights depend both on the root operator and the weights of the subtrees, and we define them as follows.

Definition 3.

$$\begin{array}{ll}
\mathsf{Lw}(AE) = 0 & \mathsf{Rw}(AE) = 0 \\
\mathsf{Lw}(t\,op) = max(\mathsf{P}(op), \mathsf{Lw}(t)) & \mathsf{Rw}(t\,op) = 0 \\
\mathsf{Lw}(op\,t) = 0 & \mathsf{Rw}(op\,t) = max(\mathsf{P}(op), \mathsf{Rw}(t)) \\
\mathsf{Lw}(lt\,op\,rt) = max(\mathsf{P}(op), \mathsf{Lw}(lt)) & \mathsf{Rw}(lt\,op\,rt) = max(\mathsf{P}(op), \mathsf{Lw}(rt))
\end{array}$$

It is easy to realize that the right weight of a syntax tree is the maximal precedence of the infix and prefix operators in the chain to the right, and the left weight of a syntax tree is the maximal precedence of the infix and postfix operators in the chain to the left as pictured below. The tree t' is either atomic or a tree for a prefix operator, and the tree t'' is either atomic or a tree for a postfix operator; see Fig. 1. We can now give the definition of the predicate Pc_H that defines the precedence correct syntax trees.

Definition 4. Given a precedence grammar H, the following rules define the predicate Pc_H, where *Left*, *Right*, *Pre* and *Post*, respectively, denote the set of left associative infix operators, right associative infix operators, prefix operators and postfix operators.

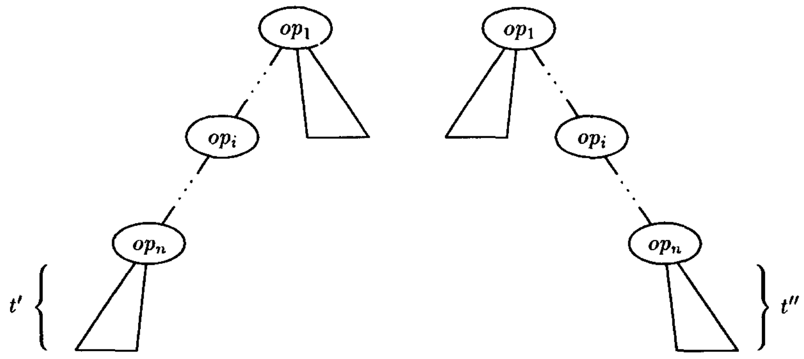


Fig. 1.

atomic expressions:

$$\mathsf{Pc}_H(AE)$$

left associative infix operators:

$$\frac{op \in \mathit{Left} \quad \mathsf{Pc}_H(lt) \quad \mathsf{Pc}_H(rt) \quad \mathsf{Rw}(lt) \leqslant \mathsf{P}(op) \quad \mathsf{Lw}(rt) < \mathsf{P}(op)}{\mathsf{Pc}_H(lt\ op\ rt)}$$

right associative infix operators:

$$\frac{op \in \mathit{Right} \quad \mathsf{Pc}_H(lt) \quad \mathsf{Pc}_H(rt) \quad \mathsf{Rw}(lt) < \mathsf{P}(op) \quad \mathsf{Lw}(rt) \leqslant \mathsf{P}(op)}{\mathsf{Pc}_H(lt\ op\ rt)}$$

prefix operators:

$$\frac{op \in \mathit{Pre} \quad \mathsf{Pc}_H(t) \quad \mathsf{Lw}(t) < \mathsf{P}(op)}{\mathsf{Pc}_H(op\ t)}$$

postfix operators:

$$\frac{op \in \mathit{Post} \quad \mathsf{Pc}_H(t) \quad \mathsf{Rw}(t) < \mathsf{P}(op)}{\mathsf{Pc}_H(t\ op)}$$

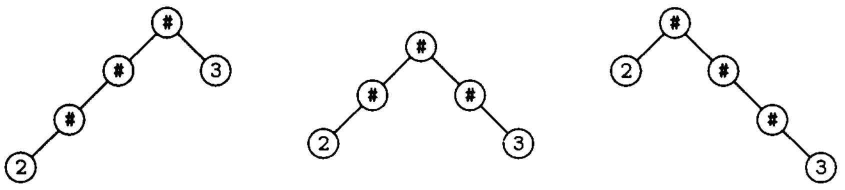
In the rest of this paper, a *precedence correct* tree is assumed to be precedence correct according to this definition. The definition can easily be extended to distfix operators. We just notice that the subtrees between operator words of the same operator are allowed to have arbitrary precedence weights as long as they are precedence correct. The precedence weights of the subtrees outside the leftmost and rightmost operator word must satisfy the same conditions as infix, prefix and postfix operators. If we let *op* denote a complete distfix operator while $op_1, \ldots, op_n$ denote the operator words in *op*, then, for example, the rule for infix distfix can be written as follows.
left associative infix distfix operators:

$$\frac{op \in \mathit{Left} \quad \mathsf{Pc}_H(t_0) \cdots \mathsf{Pc}_H(t_n) \quad \mathsf{Rw}(t_0) \leqslant \mathsf{P}(op) \quad \mathsf{Lw}(t_n) < \mathsf{P}(op)}{\mathsf{Pc}_H(t_0\ op_1\ t_1 \cdots t_{n-1}\ op_n\ t_n)}$$

The only requirement for a closed distfix operator is that the subtrees are precedence correct.
closed diftfix operators:

$$\frac{op \in \mathit{Closed} \quad \mathsf{Pc}_H(t_1) \cdots \mathsf{Pc}_H(t_{n-1})}{\mathsf{Pc}_H(op_1\ t_1 \cdots t_{n-1}\ op_n)}$$

The requirement that the operators are distinct is important. Assume that we have an operator # that is both a prefix, postfix and infix operator. Consider the sentence 2 # # # 3 and the three possible trees:

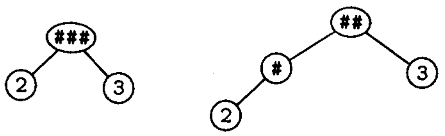


All of them are precedence correct regardless of which precedences we give to the productions. This arise from the fact that we could not know for each occurrence of the operator # if it is a prefix, postfix or infix operator. If we annotate each occurrence of the operator with which kind it is, then there is only syntax tree for the sentence.

Another ambiguity problem can arise if we have operators with different lengths of the same character. Consider, for example, the following grammar.

$$\begin{aligned} E ::= &\ E\#\#\#E \\ &\mid E\#\#E \\ &\mid E\# \\ &\mid \text{int} \end{aligned}$$

The sentence 2 # # # 3 has two different syntax trees:



Both are precedence correct regardless of which precedences we give to the productions. We think that this restriction should be taken care of in the lexical analyzer. A lexical analyzer usually finds the longest possible token.

An alternative way to define the precedence correct trees is to define which operators are allowed to occur in each subtree. To say that, we need a new definition which we will also use later.

Definition 5. An occurrence of an operator in a syntax tree t is *covered* if it occurs in a subtree of an operator with higher precedence than itself. An occurrence of an operator is *uncovered* if it is not covered.

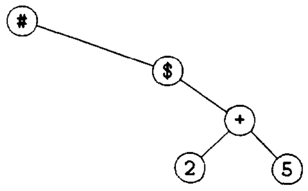
It is possible for an operator with higher precedence to occur in a subtree of an operator with lower precedence if it is covered. An example of this is the precedence correct syntax tree below generated from the same precedence grammar as discussed in the beginning of this section:

$E ::= \$E \quad 3$

$\quad \mid E+E \quad 2$ (left associative)

$\quad \mid \#E \quad 1$

$\quad \mid$ int



#$2+3

The prefix operator \$ *covers* the infix operator +. Postfix operators can be in the left subtree of an infix operator node independently of their precedence but not in the right subtree. Analogously, prefix operators can be in the right subtree of an infix operator node independently of their precedence but not in the left subtree. The conclusion of this is that if a syntax tree *lt op rt* (where *op* is left associative) must be precedence correct both *lt* and *rt* must be precedence correct, all infix and prefix operators in *lt* with higher precedence than *op* must be covered and all infix and postfix operators in *rt* with higher or equal precedence than *op* must be covered.

3.1. "Correctness" of the definition

That the definition is sensible is motivated by three theorems. The first one states that there is exactly one precedence correct tree for each sentence generated by a distfix grammar. This is desirable since we want to use precedences to throw away some syntax trees but not all. Note that this implies that a precedence grammar is unambiguous. The other two theorems motivate that it is the "correct" syntax tree that is precedence correct. They state that an operator precedence parser [4, 11] gives as result exactly the precedence correct trees.

3.1.1. Uniqueness of precedence correct trees

We will prove that there is exactly one precedence correct tree for each sentence generated by a distfix grammar. This is Theorem 12. To prove the theorem we need some definitions. The first two can be compared to the definition of covering.

Definition 6. An operator *op* is *postfix captured* in a sentence if there is a postfix operator to the right of *op* with higher precedence.

$$\cdots op \cdots postop \cdots \quad \mathsf{P}(op) < \mathsf{P}(postop)$$

Definition 7. An operator *op* is *prefix captured* in a sentence if there is a prefix operator to the left of *op* with higher precedence:

$$\cdots preop \cdots op \cdots \quad \mathsf{P}(op) < \mathsf{P}(preop)$$

The next definition characterizes the operator in a sentence which would be the root in a precedence correct syntax tree.

Definition 8. A *top operator* in a sentence generated by a distfix grammar is either
1. A postfix operator *postop* such that
   (a) there are not any operators to the right of *postop*, and
   (b) all infix and prefix operators in the subsentence w' to *postop* with higher precedence than *postop* are postfix captured in w'.

or

2. A prefix operator *preop* such that
   (a) there are not any operators to the left of *preop*, and
   (b) all infix and postfix operators in the subsentence w' to *preop* with higher precedence than preop are prefix captured in w'.

or

3. A left associative infix operator *inl* such that
   (a) all infix and prefix operators in the left subsentence w' to *inl* with higher precedence than inl are postfix captured in w'.
   (b) all infix and postfix operators in the right subsentence w'' to *inl* with higher or equal precedence than *inl* are prefix captured in w''.

or

4. A right associative infix operator *inr* such that
   (a) all infix and prefix operators in the left subsentence w' to *inr* with higher or equal precedence than inr with higher or equal precedence than inr with or equal precedence than inr are postfix captured in w'.
   (b) all infix and postfix operators in the right subsentence w'' to *inr* with higher precedence than *inr* are prefix captured in w''.

The definition can be used also for distfix operators if we regard all operator words of an operator and the enclosed expressions as a whole. In, for example, the expression **if** E **then** E **else** E, we regard **if** E **then** E **else** as a whole and thus the top operator is either the **if–then–else** operator or can be found in the E outside the operator. In the proof of Lemma 9 there is an algorithm that finds the top operator in a sentence.

Using the definition of top operator and the Lemmas 9, 10 and 11 below we can prove Theorem 12. The proofs of the lemmas are given in [2].

Lemma 9. *Every sentence generated by a distfix grammar with at least one operator has one and only one top operator.*

Lemma 10. *A syntax tree is precedence correct if it*
- *is without operators.*
- *has the top operator as root and precedence correct subtrees.*

Lemma 11. *A syntax tree in which the root operator is not the top operator is not precedence correct.*

Theorem 12. *There is exactly one precedence correct tree for each sentence generated by a distfix grammar.*

Proof. The proof is by induction on the structure of a sentence w.

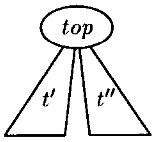
Base: There are not any operators in w. Clearly, there is exactly one precedence correct syntax tree for w.

Induction step: We assume that all subsetences of w, belonging to the same language as w, have exactly one precedence correct syntax tree and show that w has exactly one precedence correct syntax tree.

Lemma 9 gives that w contains one and only one top operator *top*. Let us first assume that *top* is an infix operator and therefore appear somewhere in the middle of w.

$$\underbrace{w' \ top \ w''}_{w} \tag{1}$$

The two subsentences w' and w'' belong to the same languages as w, and thus the induction assumption gives that there is exactly one precedence correct syntax tree for w' and w'', respectively. We call these syntax trees t' and t''. Lemma 10 gives, since *top* is a top operator in w and both t' and t'' are precedence correct, that the syntax tree below is precedence correct:



There cannot be any other precedence correct tree as follows from Lemma 11 and thus there is exactly one precedence correct tree for w. If we instead assume that *top* is a prefix or postfix operator, we only get one subsentence. Otherwise, the reasoning is the same. □

Note that if we extend precedence grammars to include also nonassociative infix operators then Theorem 12 no longer holds, since there can be sentences which do not have a precedence correct tree. Take, for example, the ususally nonassociative operator =. There is no precedence correct tree for the sentence 1=2=3. A weaker formulation of Theorem 12, that each sentence has at most one precedence correct tree can be shown for precedence grammars including nonassociative infix operators.

3.1.2. Comparison with operator precedence parsing

It is easy to translate a precedence grammar to an operator precedence table used in operator precedence parsing. An algorithm is given in the "dragon" book by Aho et al. [4, Ch. 4.6].

Theorem 13. *Parsing a sentence generated from a precedence grammar with an operator precedence parser gives a precedence correct tree as result.*

Proof. The proof is by induction on the number of reductions in the parsing process.

Base: No reduction are used. Trivial.

Induction step: We show that if we have done n reductions and the trees built from these reductions are precedence correct then the resulting tree after one more reduction is precedence correct.

Case analysis: On the possible handles in an operator precedence parser.

$\lessdot E\ post \gtrdot$: To show that the tree after the reduction is precedence correct we must, according to Definition 4, show that $\mathsf{Rw}(E) < \mathsf{P}(post)$. We know that $\mathsf{Rw}(E)$ is the maximal precedence of the chain of prefix and infix operators to the right. Assume that op is the one with highest precedence. It is not possible that op has higher precedence than $post$ because then we would have reached the configuration $\cdots op \lessdot E'\ post \gtrdot \cdots$ some time earlier in the parsing process. In such a configuration, $E'\ post$ would have been chosen for reduction and we would never reach the configuration $\cdots \lessdot E\ post \gtrdot \cdots$.

$\lessdot pre\ E \gtrdot$: Analogous to the postfix case.

$\lessdot E_1\ inl\ E_2 \gtrdot$: The proof of $\mathsf{Rw}(E_1) \leqslant \mathsf{P}(inl)$ is analogous with the postfix case. The proof of $\mathsf{Lw}(E_2) < \mathsf{P}(inl)$ is analogous with the prefix case. □

Theorem 14. *An operator precedence parser can give all precedence correct trees as result.*

Proof. Take an arbitrary precedence grammar H and an arbitrary precedence correct syntax tree t generated by H. We will prove that t can be a result from an operator precedence parse. Call the sentence of t for w. Parsing w by an operator precedence parser does not give raise to a syntax error since w is a correct sentence so the result of the parsing is a syntax tree t'. Since we have shown in Theorem 13 that operator precedence parsers only gives precedence correct trees as result then t' must be precedence correct. Theorem 12 says that two different syntax trees for the same sentence cannot both can be precedence correct. This means that t' must be equal to t and thus, since t was arbitrary, an operator precedence parser can generate all precedence correct trees. □

4. Transformation to an unambiguous grammar

Besides the theoretical interest of knowing whether a precedence grammar can be transformed to an unambiguous context-free grammar, such an algorithm is sometimes needed in practice. For example, if we want to describe a language with a precedence grammar but parse the language with a method that cannot handle precedence rules, then the algorithm is definitely needed. One such commonly used parsing method is recursive descent [7], and another is DCG [17]. It is not obvious how to use precedence rules in Earley's algorithm [8] even if it is possible as shown in [2].

For grammars with only infix operators, there is a well-known algorithm [4, Ch. 2.2] that transforms them to ordinary unambiguous context-free grammars by introducing one nonterminal for each precedence level. But if the language contains also prefix and postfix operators, this method does not work. Consider the precedence grammar

$$\begin{array}{lll} E ::= E? & 4 & \\ \quad\mid E+E & 3 & \text{(left associative)} \\ \quad\mid E! & 2 & \\ \quad\mid E*E & 1 & \text{(left associative)} \\ \quad\mid \mathtt{int} & & \end{array}$$

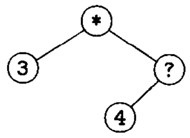
For this grammar the method of introducing one nonterminal for each precedence level does not work. Using the method naïvely would give the grammar

$$\begin{array}{lll} E(4) ::= & E(4)? & \mid E(3) \\ E(3) ::= & E(3)+E(2) & \mid E(2) \\ E(2) ::= & E(2)! & \mid E(1) \\ E(1) ::= & E(1)*E(0) & \mid E(0) \\ E(0) ::= & \mathtt{int} & \end{array}$$

But this grammar is incorrect since it does not generate *all* precedence correct syntax trees. It does not generate all sentences as the original grammar, for example, are not 7?+8, 3?! and 9+6?*8 derivable. There exists an unambiguous grammar which generates the same sentences as the precedence grammar above:

$$\begin{array}{ll} E(3) ::= & E(3)+E(1) \mid E(1) \\ E(1) ::= & E(1)*E(0) \mid E(0) \\ E(0) ::= & \mathtt{int} \mid E(0)! \mid E(0)? \end{array}$$

But this grammar is not correct since it generates syntax trees which are *not* precedence correct, for example,



3*4?

Another attempt to construct a grammar from which precisely the precedence correct syntax trees are derivable is

$$E(4) ::= E(3)$$

$$E(3) ::= E(3)+E(2) \mid E(2)$$

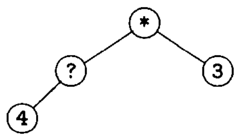
$$E(2) ::= E(1)$$

$$E(1) ::= E(1)*E(0) \mid E(0)$$

$$E(0) ::= \texttt{int} \mid E(2)! \mid E(4)?$$

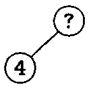
Here we have tried to incorporate the idea that a postfix operator forms a closed expression. This grammar is also incorrect since it is ambiguous and derives both precedence correct syntax trees and incorrect ones. This illustrates that we must construct the grammar in such a way that, for every production $E ::= E_\ell * E_r$, it is not possible to derive a syntax tree for a postfix operator with higher precedence than $*$ from E_r. Syntax trees for postfix operators with lower precedence than $*$ must of course be derivable from E_r.

Let us now make some reflections about the syntax trees which must be derivable from the nonterminal E_ℓ in the production $E ::= E_\ell * E_r$. This is harder because we sometimes want syntax trees for postfix operators with higher precedence than $*$ to be derivable from E_ℓ and sometimes not. Consider the syntax tree:

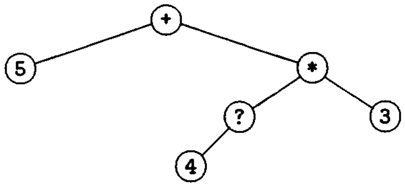


4?*3

It is precedence correct and has as left subtree:

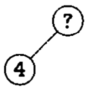


Therefore, we must ensure that the subtree is derivable from E_ℓ in a production $E ::= E_\ell * E_r$. We have shown that there must be at least one production $E ::= E_\ell * E_r$ such that we can derive syntax trees for postfix operators with higher precedence than $*$ from E_ℓ. We will now show that we must also have productions $E ::= E_\ell * E_r$ such that we cannot derive syntax trees for postfix operators with higher precedence than $*$ from E_ℓ. Consider the following syntax tree in which the syntax tree above is a subtree:

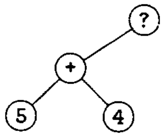


5+4?*3

This syntax tree is not precedence correct since ? has higher precedence than +. So in this case we must ensure that the syntax tree below is *not* derivable from E_ℓ:

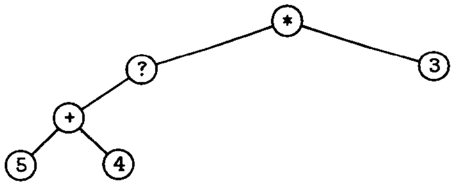


Note that the occurrence of + is covered in the syntax tree



5+4?

This syntax tree must be derivable from E_ℓ, since the precedence correct syntax tree for the sentence 5+4?*3 is



That is, if a postfix operator is allowed to occur in a syntax tree derived from a nonterminal, it can cover also other operators. The reasoning for the case where prefix operators are allowed to occur is analogous.

We have indicated that we must have more than one nonterminal from which it is possible to derive a syntax tree with a specific infix operator as root. In the nonterminals there must be information about which postfix and prefix operators are allowed to occur in the left (right) subtree. If a postfix (prefix) operator is allowed then it can cover also other operators which otherwise are not allowed to occur in the left (right) subtree. So, the number of different nonterminals we need for each infix operator depends on how many postfix and prefix operators with higher precedence there are in the precedence grammar.

4.1. *The algorithm $\mathscr{M}$*

In this section we transform our observations to an algorithm that generates a context-free grammar where the precedence rules in a precedence grammar are incorporated. For simplicity, the algorithm handles exactly one operator on each precedence level. This is not a severe restriction. We could easily extend the algorithm to allow several operators on each level or extend the resulting grammar with more operators. Another restriction is that we do not handle distfix operators but only infix, prefix and postfix operators. Nor is this a severe restriction. We could easily extend the algorithm to allow distfix operators.

Our algorithm generates a grammar with nonterminals of the form $E(n,p,q)$ where the indices are natural numbers and show which operators are allowed to occur in the syntax trees derived from the nonterminal. Before giving the algorithm we introduce some notation.

inl_i	the ith left associative infix operator
inr_i	the ith right associative infix operator
pre_i	the ith prefix operator
$post_i$	the ith postfix operator
$\mathsf{P}_{\mathsf{pre}}(n,p)$	the precedence of the pth prefix operator with higher precedence than n
$\mathsf{P}_{\mathsf{post}}(n,q)$	the precedence of the qth postfix operator with higher precedence than n.

Given a grammar, left associative infix operators, right associative infix operators, prefix operators and postfix operators are numbered separately in increasing precedence order. We define $\mathsf{P}_{\mathsf{post}}(x,0) = x$ and $\mathsf{P}_{\mathsf{pre}}(x,0) = x$. Examples are given in Section 4.2. Now we can define which operators must not occur in a syntax tree derived from $E(n,p,q)$.

1. An operator op for which $\mathsf{P}(op) > max\,\mathsf{P}_{\mathsf{pre}}(n,p)$, $\mathsf{P}_{\mathsf{post}}(n,q))$.
2. An uncovered prefix operator op for which $\mathsf{P}(op) > \mathsf{P}_{\mathsf{pre}}(n,p)$.
3. An uncovered postfix operator op for which $\mathsf{P}(op) > \mathsf{P}_{\mathsf{post}}(n,q)$.
4. An uncovered infix operator op for which $\mathsf{P}(op) > n$.

As mentioned earlier there must be more than one production for each operator, the number depends on the number of prefix and postfix operators with higher precedence. The algorithm generates the grammar by five rules which introduce the nonterminals $E(n, p, q)$ and the production for them.

1. The rule for left associative infix operators:

$$\begin{aligned} E(\mathsf{P}(inl_i), p, q) &::= E(\mathsf{P}(inl_i), 0, q)\ inl_i\ E(\mathsf{P}(inl_i) - 1, p, 0) \\ &\quad\mid E(\mathsf{P}(inr_i) - 1, p, q) \end{aligned}$$

where $1 \leqslant i \leqslant$ number of left associative infix operators, $0 \leqslant p \leqslant$ number of prefix operators with higher precedence than inl_i, $0 \leqslant q \leqslant$ number of postfix operators with higher precedence than inl_i.

2. The rule for right associative infix operators:

$$\begin{aligned} E(\mathsf{P}(inr_i), q, q) &::= E(\mathsf{P}(inr_i) - 1, 0, q)\ inr_i\ E(\mathsf{P}(inr_i), p, 0) \\ &\quad\mid E(\mathsf{P}(inr_i) - 1, p, q) \end{aligned}$$

where $1 \leqslant i \leqslant$ number of right associative infix operators, $0 \leqslant p \leqslant$ number of prefix operators with higher precedence than inr_i, $0 \leqslant q \leqslant$ number of postfix operators with higher precedence than inr_i.

3. The rule for prefix operators:

$$E(\mathsf{P}(pre_i), p, q) ::= E(\mathsf{P}(pre_i) - 1, p + 1, q)$$

where $1 \leqslant i \leqslant$ number of prefix operators, $0 \leqslant p \leqslant$ number of prefix operators with higher precedence than pre_i, $0 \leqslant q \leqslant$ number of postfix operators with higher precedence than pre_i.

4. The rule for postfix operators:

$$E(\mathsf{P}(post_i), p, q) ::= E(\mathsf{P}(post_i) - 1, p, q + 1)$$

where $1 \leqslant i \leqslant$ number of postfix operators, $0 \leqslant p \leqslant$ number of prefix operators with higher precedence than $post_i$, $0 \leqslant q \leqslant$ number of postfix operators with higher precedence than $post_i$.

5. The A-rule:

$$\begin{aligned} E(0, p, q) &::= AE \\ &\quad\mid pre_i\ E(\mathsf{P}(pre_i), p - i, 0) \quad \text{where } 1 \leqslant i \leqslant p \\ &\quad\mid E(\mathsf{P}(post_j), 0, q - j)\ post_j \quad \text{where } 1 \leqslant j \leqslant q \end{aligned}$$

where $0 \leqslant p \leqslant$ number of prefix operators, $0 \leqslant q \leqslant$ number of postfix operators.

The start symbol in the resulting grammar is the nonterminal $E(m, 0, 0)$ where m is the highest precedence.

4.2. Example

Let us use the method to construct an unambiguous grammar for the language generated by the precedence grammar

$$\begin{array}{lll} E ::= E? & 4 & \\ \quad \mid E+E & 3 & \text{(left associative)} \\ \quad \mid E*E & 1 & \text{(left associative)} \\ \quad \mid \texttt{int} & & \end{array}$$

For this grammar we have

$$\begin{array}{lll} inl_1 = * & post_1 = ! & \mathsf{P}_{\mathrm{post}}(1,2) = ? \\ inl_2 = + & post_2 = ? & \mathsf{P}_{\mathrm{post}}(2,1) = ? \end{array}$$

The rule for left associative infix operators yields the following productions since $\mathsf{P}(+) = 3$ and there is one prefix operator with higher precedence than $+$ but no prefix operators:

$$E(3,0,0) ::= E(3,0,0)+E(2,0,0) \mid E(2,0,0)$$

$$E(3,0,1) ::= E(3,0,1)+E(2,0,0) \mid E(2,0,1)$$

The rule for left associative infix operators yields also the following productions since $\mathsf{P}(*) = 1$ and there are two postfix operators with higher precedence than $*$ but no prefix operators:

$$E(1,0,0) ::= E(1,0,0) * E(0,0,0) \mid E(0,0,0)$$

$$E(1,0,1) ::= E(1,0,1) * E(0,0,0) \mid E(0,0,1)$$

$$E(1,0,2) ::= E(1,0,2) * E(0,0,0) \mid E(0,0,2)$$

The rule for postfix operators yields the following productions:

$$E(4,0,0) ::= E(3,0,1)$$

$$E(2,0,0) ::= E(1,0,1)$$

$$E(2,0,1) ::= E(1,0,2)$$

The first production arise since $\mathsf{P}(?) = 4$ and there are neither prefix operators nor postfix operators with higher precedence than ?. The last two productions arise since $\mathsf{P}(!) = 2$ and there is one postfix operator with higher precedence than ! but no prefix operators.

Finally the A-rule yields the following productions since we have two postfix operators, ! and ?:

$$E(0,0,0) ::= \texttt{int}$$

$$E(0,0,1) ::= \texttt{int} \mid E(2,0,0)\ !$$

$$E(0,0,1) ::= \texttt{int} \mid E(2,0,1)\ !\ E(4,0,0)?$$

The resulting grammar contains some useless[2] nonterminals and a lot of single productions. These could easily be eliminated and algorithms for that is, for example, given by Grune and Jacobs [13]. We can eliminate 8 of the 19 productions from the grammar above.

If we augment the grammar with a prefix operator having greater precedence than all other operators then the unambiguous grammar will consist of 42 productions and even if we eliminate all useless and all single productions there is still 26 left.

4.3. *Correctness of algorithm $\mathscr{M}$*

The correctness of algorithm $\mathscr{M}$ is shown by proving that every precedence grammar H generates the same language (the same set of syntax trees) as the grammar we obtain by applying the algorithm to H. We cannot consider the set of strings the two grammars generate since we are interested in the structure of the expressions. Neither could we consider parse trees since the nonterminals in the parse trees have different names and there are chains of single productions in the parse trees for the grammars that the algorithm produces. The correctness is formulated by the following theorem.

Theorem 15. *If $\mathscr{T}(\mathscr{M}(H))$ is the generated language of syntax trees from the grammar $\mathscr{M}(H)$ and $\mathscr{T}(H) = \{t\colon \mathsf{Pc}_H(t)\}$ then, for every precedence grammar H, the language $\mathscr{T}(H)$ is equal to the lanugage $\mathscr{T}(\mathscr{M}(H))$.*

Proof. In the proof we use induction on the precedence grammar. By "induction on a precedence grammar" we mean that we show a statement for a grammar consisting of zero operators, and under the assumption that a statement holds for a grammar consisting of m operators, we show that it holds if we extend the grammar with one more operator. The operators are introduced in increasing precedence order. We use the notation H_m for a precedence grammar where the highest precedence of the operators is m. Let H_m be equal to H_{m-1} plus the production for a new operator with precedence m. Then we have two important properties:

1. $\mathscr{T}(H_{m-1}) \subseteq \mathscr{T}(H_m)$,
2. $\mathscr{P}(\mathscr{M}(H_{m-1})) \subseteq \mathscr{P}(\mathscr{M}(H_m))$, where $\mathscr{P}(G)$ denote the set of productions in the grammar G.

[2] A nonterminal is useless if it does not appear in any derivation of any sentence.

The proof of (Theorem 15) $\mathscr{T}(H) = \mathscr{T}(\mathscr{M}(H))$ is divided into two parts:

1. $\mathscr{T}(H) \subseteq \mathscr{T}(\mathscr{M}(H))$,
2. $\mathscr{T}(\mathscr{M}(H)) \subseteq \mathscr{T}(H)$.

In the first part we show that if a syntax tree is precedence correct, that is, if it can be generated from a precedence grammar H, then it can be generated from the grammar we obtain by applying the precedence removing algorithm $\mathscr{M}$ on H. In the second part we show that if a syntax tree is generated from a grammar we have obtained by applying the algorithm on a precedence grammar, then the syntax tree is precedence correct.

In both parts we use a predicate $Q_H(t, n, p, q)$ which informally holds if the syntax tree t is precedence correct and some operators given by the natural numbers n, p and q do not occur in t. We will define Q precisely later. In each part we show one direction of

$$Q_H(t, n, p, q) \Leftrightarrow E(n, p, q) \rightarrow^* t. \tag{2}$$

We define Q in such a way that

$$t \in \mathscr{T}(H_m) \Leftrightarrow Q_{H_m}(t, m, 0, 0). \tag{3}$$

Since $E(m, 0, 0)$ is the start symbol in the grammar $\mathscr{M}(H_m)$ we have

$$E(m, 0, 0) \rightarrow^* t \Leftrightarrow t \in \mathscr{T}(\mathscr{M}(H_m)). \tag{4}$$

So, from (3) and (4) it follows that if we prove (2) we have then shown

$$t \in \mathscr{T}(H_m) \Leftrightarrow t \in \mathscr{T}(\mathscr{M}(H_m)). \tag{5}$$

From (5), the theorem follows immediately.

Let us turn to the definition of the predicate Q.

Definition 16. $Q_H(t, n, p, q)$ holds if and only if

1. $t \in \mathscr{T}(H)$
2. The following operators do not occur in the syntax tree t.
   (a) An operator *op* for which $\mathsf{P}(op) > max(\mathsf{P}_{\mathsf{pre}}(n, p), \mathsf{P}_{\mathsf{post}}(n, q))$.
   (b) An uncovered prefix operator *op* for which $\mathsf{P}(op) > \mathsf{P}_{\mathsf{pre}}(n, p)$.
   (c) An uncovered postfix operator *op* for which $\mathsf{P}(op) > \mathsf{P}_{\mathsf{post}}(n, q)$.
   (d) An uncovered infix operator *op* for which $\mathsf{P}(op) > n$.

Recall that an occurrence of an operator in a syntax tree t is uncovered if it does not occur in a subtree of an operator with higher precedence than itself.

Example 17. We illustrate which operators are allowed in a syntax tree t if $Q_{H_m}(t, n, p, q)$ would hold in Fig. 2. Clearly, $Q_{H_m}(t, m, 0, 0)$ holds if and only if $t \in \mathscr{T}(H_m)$.

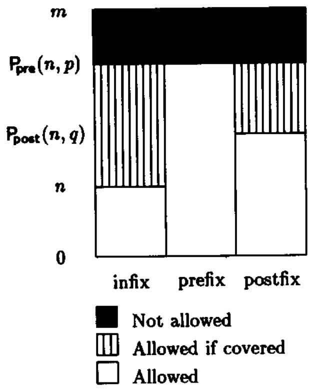


Fig. 2.

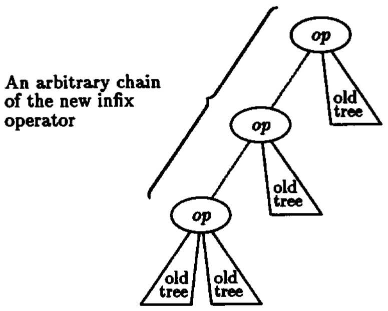


Fig. 3.

Proof of $\mathscr{T}(H) \subseteq \mathscr{T}(\mathscr{M}(H))$. Here, we only give an overview of the proof. The whole proof is found in [2].

First we use induction on the precedence grammar H.

Base: There are no operators in H. Trivial.

Induction step: Under the assumption that the transformation is correct for a precedence grammar with $m - 1$ operators we show that it is correct if we extend the grammar with one new operator. We prove this by case analysis on the new operator.

1. *Left associative infix operator*: All trees have the form shown in Fig. 3.

We use induction on the length of the chain of new operator.

Base: The length is zero, that is the tree is an old one.

Induction step: Under the assumption that we can derive every tree with l occurrences of the new infix operator, we show that we can derive every tree with $l + 1$ occurrences of the new infix operator.

2. *Right associative infix operator*. Analogous with a left associative infix operator.
3. *Prefix operator*. All trees have the following form (Fig. 4), where *pin* is either a prefix or infix operator. We use induction on the length of the chain of infix and prefix operators.

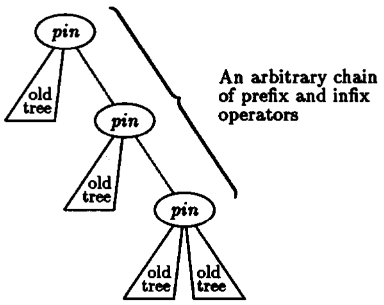


Fig. 4.

Base: The length is zero, that is the tree is an old one.

Induction step: Under the assumption that we can derive every tree with a chain of l occurrences of infix and prefix operators, we show that we can derive every tree with a chain of $l+1$ occurrences of infix and prefix operators.

We show the two cases that the tree has an infix operator as root and a prefix operator as root.

4. *Postfix operator.* analogous with a prefix operator. □

Proof of $\mathscr{T}(\mathscr{M}(H)) \subseteq \mathscr{T}(H)$. We again only sketch the proof here. The whole proof is found in [2]. To prove

(C6) $\forall t\ \forall(n,p,q)\quad E(n,p,q) \rightarrow^{*} t \ \Rightarrow\ Q_H(t,n,p,q)$,

we use induction on the length of a derivation. Under the induction assumption

(A7) $\forall t\ \forall(n,p,q)\quad E(n,p,q) \rightarrow^{*} t \ \Rightarrow\ Q_H(t,n,p,q)$ where $1 \leqslant r < \gamma$,

we show

(C8) $\forall t\ \forall(n,p,q)\quad E(n,p,q) \rightarrow^{\gamma} t \ \Rightarrow\ Q_H(t,n,p,q)$.

To show

(C9) $Q_H(t,n,p,q)$

for arbitrary t and (n,p,q) and given the assumption

(A10) $E(n,p,q) \rightarrow^{\gamma} t$

we use case analysis on the first step in the derivation

1. $E(n,p,q) \rightarrow E(n-1,p+1,q)$
2. $E(n,p,q) \rightarrow E(n-1,p,\ q+1)$
3. $E(n,p,q) \rightarrow E(n,0,q)\ inop\ E(n-1,p,0)$
4. $E(n,p,q) \rightarrow preop_i\ E(\mathsf{P}(preop_i),\ p-i,0)$
5. $E(n,p,q) \rightarrow E(\mathsf{P}(postop_i),\ 0,\ q-i)\ postop_i$
6. $E(n,p,q) \rightarrow E(n-1,p,q)$. □

5. Practical use of algorithm $\mathscr{M}$

We have used the algorithm to implement an experimental language with user-defined distfix operators [1], also described in [2]. A distfix operator is specified by the operator words and optionally precedence and associativity. The parser is written in ML [16] and uses parser constructors due to Burge [5] and Fairbairn [10] and Kent Petersson and Sören Holmström [18]. Using these parser constructors it is easy to write a parser given a grammar, since there are constructors that recognize terminal symbols, sequences, and alternatives and other constructors that introduce actions during the parsing.

The parser constructors construct a recursive descent parser and therefore the grammar must not be left recursive and it must express precedences of the involved operators.

In the parser for user-defined distfix operators we use the rules in the algorithm described above. We have to do some changes in order to remove left recursion, and we never generate the entire grammar with all different nonterminals. Instead, we see the rules as production schemas in a way that is similar to the hyper rules in two-level grammars [6], and instantiate the rules during the parsing. Hanson [14] describes another technique for parsing expressions using recursive descent without introducing additional nonterminals, but this technique does not handle prefix and postfix operators of different precedence as our method.

References

[1] A. Aasa, Recursive descent parsing of user defined distfix operators, Licentiate Thesis, Department of Computer Sciences, Chalmers University of Technology, S-412 96 Göteborg, Sweden, May 1989.

[2] A. Aasa, User defined syntax, Ph.D. Thesis, Department of Computer Sciences, Chalmers University of Technology, S-412 96 Göteborg, Sweden, October 1992.

[3] A.V. Aho, S.C. Johnson and J.D. Ullman, Deterministic Parsing of ambiguous grammars, *Commun. ACM* **18** (1975) 441–452.

[4] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, Tools* (Addison-Wesley, Reading, MA, 1986).

[5] W.H. Burge, *Recursive Programming Techniques* (Addison-Wesley, Reading, MA, 1975).

[6] J.C. Cleaveland and R.C. Uzgalis, *Grammars for Programming Languages* (Elsevier, Amsterdam, 1977).

[7] A.J.T. Davie and R. Morrison, *Recursive Descent Compiling* (Ellis Horwood, Chichester, UK, 1981).

[8] J. Earley, An efficient context-free parsing algorithm, *Comm. ACM* **13** (1970) 94–102.

[9] J. Earley, Ambiguity and precedence in syntax description, *Acta Inform.* **4** (1975) 183–192.

[10] J. Fairbairn, Making form follow function: an exercise in functional programming style, *Software-Practice and Experience* **17** (1987) 379–386.

[11] R.W. Floyd. Syntactic analysis and operator precedence, *J. ACM* **10** (1963) 316–333.

[12] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud and J. Meseguer, Principles of OBJ2, in: *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, LA (1985) 52–66.

[13] D. Grune and C.J.H. Jacobs, *Parsing Techniques a Practical Guide* (Ellis Horwood, limited, 1990.

[14] D.R. Hanson, Compact recursive-descent parsing of expressions, *Software-Practice and Experience* **15** (1985) 1205–1212.

[15] R. Milner, Standard ML proposal, *Polymorphism: The ML/LCF/Hope Newsletter* 1(3) (1984).

[16] R. Milner, M. Tofte and R. Harper, *The Definition of Standard ML* (MIT Press, Cambridge, MA, 1990).

[17] F.C.N. Pereira and D.H.D. Warren, Parsing as deduction, in: *Proc. 21st Ann. Meeting of the Association for Computational Linguistics*, (1983) 137–144.

[18] K. Petersson, LABORATION: Denotationsemantik i ML, Department of Computer Sciences, University of Göteborg and Chalmers University of Technology, S-412 96 Göteborg, Sweden, 1985.

[19] M. Share, Resolving ambiguities in the parsing of translation grammars, *ACM Sigplan Notices* **23** (1988) 103–109.

[20] L. Sterling and E. Shapiro, *The Art of Prolog* (MIT Press, Cambridge, MA, 1986).

[21] R.M. Wharton, Resolution of ambiguity in parsing, *Acta Inform.* **6** (1976) 387–395.