Precedences in Specifications and Implementations of Programming Languages

Annika Aasa

Programming Methodology Group, Dept. of Computer Sciences
Chalmers University of Technology
S-412 96 Göteborg, Sweden
E-mail: annika@cs.chalmers.se

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.

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. 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 that 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.

```
E ::= E + E
    | E - E
    | E * E
    | E / E
    | int
    | ( E )
```

```
E ::= E + T
    | E - T
    | T

T ::= T * F
    | T / F
    | F

F ::= int
    | ( E )
```

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

If a language has user-defined operators as for example ML [16] and PROLOG [19] it is also very 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 should indicate where this production should be placed 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, 11, 20]. 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 that transforms a grammar with precedences to an unambiguous context-free grammar.

2 Notation

We will only consider grammars with one nonterminal and in which every left-hand-side is either *atomic* or is of one of the following forms: $A\ op\ A$, $A\ op$, $op\ A$, where A is the nonterminal and *op* a terminal. Atomic left-hand-sides either consists solely of terminals (for example `int`) or is a nonterminal surrounded by terminals (for example (A)). We will sometimes use AE as a shorthand for *all* atomic left-hand-sides. Furthermore, all terminals must be distinct.

This kind of grammars generates languages with infix, prefix and postfix operators, but it is trivial to extend the ideas to grammars with distfix operators. The first restriction, only 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 [2].

With precedence rules we mean both precedence and associativity rules. A grammar together with precedence rules will be called a *precedence grammar* and the notation is as follows.

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

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 $\mathsf{P}(op)$. The operators can be divided into four kinds: left associative infix, right associative infix, prefix and postfix, and operators of different kinds are not allowed to have the same precedence. In the algorithm described in section 4 we suppose that there is only one operator of each precedence but it is trivial to extend the algorithm to handle more. We let the variable H range over precedence grammars.