How to explicitly specify precedence and associativity in a grammar

As explained in the lecture and text, grammars are often ambiguous if they have production rules that are both left and right recursive. For example, consider the following expression grammar G1:

E → E + E
E → E - E
E → E * E
E → E / E
E → id
E → num
E → (E)

This grammar is ambiguous. One way to eliminate this ambiguity is to rewrite the grammar so the operator precedence and associativity is explicit in the grammar. Associativity is specified by making the grammar rule either left or right recursive, but not both (the rules in the above grammar are a problem because they are both left and right recursive). Precedence is specified by adding non-terminals for each precedence level.

Associativity and precedence determine which subexpression is evaluated first when there are no parentheses; that is, they eliminate the above ambiguity so that only one parse tree can be created for the expression as a whole.

Associativity determines which subexpression is evaluated first when the same operator occurs in two subexpressions! For example, 1 + 2 + 4 would be evaluated as (1 + 2) + 4 if + is left-associative, i.e., (1+2) is evaluated before 4 is added to that result. This is especially important for operators like subtraction and division that yield a different result depending on which subexpression is evaluated first, e.g., (4-2)-1 yields a different result than 4 - (2-1).

When there are two different operators, then precedence is used to determine which subexpression is evaluated first. For example, 1 + 2 * 3 would be evaluated as 1 + (2 * 3) if * has precedence over +.

The usual associativity for arithmetic operators is that they are all left associative; this is specified by making the rules left recursive. For example, the following rule is left recursive and is not right recursive because the non-terminal from the left-hand side appears to the left of the operator + on the right-hand side.

E → E + T

This rule defines non-terminal E (the symbol on the left-hand side) in terms of itself, i.e., E also appears on the right-chand side of the arrow (-->); therefore, it is a recursive production rule. On the right-hand side, the E appears to the left of the + operator; therefore the rule is left-recursive. This makes the operator + left associative.

To specify precedence, the right-hand side of production rules with lower precedence must reference rules with higher precedence. For example, the following rule references T in addition to the non-terminal E on the right hand side (however, when determining precedence, we don't consider the E since that non-terminal appears on the left hand side of the rule; it determines associativity).

E → E + T

Therefore, operator + will have lower precedence than any operators occurring on the right hand side of production rules for T. The following transformation of grammar G1 and explanation should clarify this. This new grammar will be referred to as grammar G2.

Notice that in the new grammar, G2, we have introduced non-terminals E1, E2, and P. These represent the three levels of precedence (P is the primitive level with no operators). Also, E is the start symbol for the grammar.

(0) E → E1

(1) E1 → E1 + E2
(2) E1 → E1 - E2
(3) E1 → E2

(4) E2 → E2 * P
(5) E2 → E2 / P
(6) E2 → P

(7) P → id
(8) P → num
(9) P → (E)

The usual precedence of arithmetic operators is that the additive operators (+ and -) have lower precedence than the multiplicative operators (* and /). Furthermore, all these operators are left associative. The above grammar specifies this precedence and associativity as explained below. I have also numbered the rules so they can be referenced in the explanation.

Notice that rules (1) and (2) are left recursive; this specifies that operators **+** and **-** are left associative. Similarly, operators ***** and **/** are also left associative. Therefore, an expression such as

x + y + z + 1

is equivalent to the parenthesized expression

((x + y) + z) + 1

Notice also that rules (1), (2), and (3) reference the non-terminal E2, but the rules for E2 do not reference E1. Therefore, operators **+** and **-** have lower precedence than operators ***** and **/**. Hence, an expression such as

x + y * z + 1

is equivalent to the parenthesized expression

((x + (y * z)) + 1

Finally, recursive rules must have a base rule, i.e. a rule that is non-recursive. Without such rules, no sentences can be derived from the grammar, i.e., the language cannot contain sentences generated by those rules. This is because there is no rule to terminate the recursion; such grammars are like programs containing an infinite loop. Rule (3) is the required base rule for the non-terminal E1, and rule (6) is the base rule for E2.

Note also that there are grammars for which it is not always this obvious that a rule is left or right-recursive just as a method M can be recursive by calling a method that directly or indirectly calls back to M; but we will not be concerned with such rules in this course since most programming languages have very few such operators.

In programming languages that allow assignment in expressions, the assignment operator (=) is an example of an operator that is right associative, e.g., languages like Java, C++, and C (JL does not allow assignment in expressions). For example, the grammar for assignment (if added to the above grammar G2) could be specified with the following right recursive rule:

(a) E → id = E

Therefore, an expression such as

x = y = z + 1

is equivalent to the parenthesized expression

x = (y = (z + 1))

Notice that rule (a) specifies that the assignment operator is right associative and has lower precedence than the additive operators (+ and -); this is because rule (0) (E → E1) of grammar G2 references E1 on its right-hand side. Therefore, any operators in the rules for E1 (+ and -) have higher precedence than all operators specified in the rules where E is on the left-hand side, e.g., rule (a) above.

An operator is non-associative if the rules where it is defined are neither left nor right recursive. For example, relational operators are non-associative and could be specified as in the following production rules:

(b) E → E1 < E1
(c) E → E1 > E1

Notice that the above two rules do not reference the non-terminal on the left-hand side of those rules (E) and thus the operators defined in rules (b) and (c) are non-associative. Furthermore, these two operators would have lower precedence than the operators on the right-hand side of the production rules for E1, i.e., lower than operators + and -.

Exercises

1. (a) Show that grammar G1 is ambiguous by giving two left-most derivations for the expression "x+y+z" using G1.

   (b) Give the two corresponding parse trees for the two derivations given in part (a).

2. (a) Using Grammar G2 (with E as the start symbol), give a derivation and the corresponding parse tree for the expression "x + y * z + 1".

   (b) Using Grammar G2 (with E as the start symbol), give a derivation and the corresponding parse tree for the expression "((x + (y * z)) + 1".

   (c) The two parse trees from parts (a) and (b) should have the same structure (if done correctly) except for the parentheses. What can we conclude from this?

3. (a) If rules (b) and (c) are added to grammar G2, is the expression "x < 1 < y < 2" in the language generated by that grammar, i.e., is there a derivation for that expression?

   (b) What can we conclude from part (a)?