CS403: Programming Languages

Implementing Associativity and Precedence in Recursive Descent Expression Grammars

Associativity and Precedence

Suppose you wish to implement a grammar for C-like expressions. It is not easy for a number of reasons. One is that operators in C have specific associativities. For example, addition is left-associative while assignment is right-associative. Another problem is precedence. There are many levels of precedence in a C expressions. Handling associativity and precedence at the same time can be tricky. For example, the expression:

```
a = b = c + d * e / 3 - 2 - q
```

should be parsed the same as:

```
(a = (b = (((c + ((d * e) / 3)) - 2) - q)))
```

but it is not obvious how to accomplish this feat. First, we'll look at incorporating associativity into a grammar.

Associativity

Consider the following grammar:

```
expression : expression operator expression | unary

operator : PLUS | MINUS | TIMES | DIVIDES

unary : VARIABLE | NUMBER | ( expression ) | - unary
```

which implements a significant subset of C expressions. This grammar is easy to come up with, but it has a few drawbacks. For one, it is ambiguous. With this grammar, an expression such as:

```
x - y - 3
```

can be recognized as:

```
((x - y) - 3)
```

or:

```
(x - (y - 3))
```

corresponding to left-associativity and right-associativity, respectively. We can remove the ambiguity by rewriting the ambiguous rule as:

```
expression : unary operator expression | unary
```

This forces right associativity. We can also rewrite the rule to force left associativity:

```
expression : expression operator unary | unary
```

If we wish to implement the left-associative rule using recursive descent parsing techniques, we quickly run into trouble:

```
function expression()
    {
    var tree;
    if (expressionPending())
        {
        var temp = expression();    //infinite recursion!
        var op = operator();
        op.left = temp;
```

```
            op.right = unary();
            tree = op;
            }
        else
            tree = unary();
        return tree;
        }
```

We will fall into an infinite recursive loop because *expressionPending* will always return true; what begins an *expression* also begins a *unary*. The right associative version of expression *is*:

```
    function expression()
        {
        tree = unary();
        if (operatorPending())
            {
            var temp = operator();
            temp.left = tree;
            temp.right = expression();
            tree = temp;
            }
        return tree;
        }
```

This works fine, assuming all the operators recognized by the function operator are right-associative operators. What if the operators should be left-associative? Curiously, if we replace the *if* with a *while* and the recursive call to *expression* with a call to *unary*, as in:

```
    function expression()
        {
        tree = unary();
        while (operatorPending())
            {
            temp = operator();
            temp.left = tree;
            temp.right = unary();
            tree = temp;
            }
        return tree;
        }
```

we still recognize the same language. More curiously, the iterative function now builds a left-associative parse tree!

Precedence

If we are to follow the mathematical rules we learned earlier in life, we would like multiplication to happen before addition regardless of where the multiplications and additions occur in the expression. Using the left associative grammar above, the expression:

```
2 + 3 * 5 + 6
```

would be equivalent to:

```
(((2 + 3) * 5) + 6)
```

The structure we would like to see is:

```
((2 + (3 * 5)) + 6)
```

We can rewrite our grammar to accomplish this goal:

```
expression1 : expression2 operator1 expression1 | expression2

operator1 : PLUS | MINUS

expression2 : unary operator2 expression2 | unary

operator2 : TIMES | DIVIDES

unary  : VARIABLE | NUMBER | (expression1) | MINUS unary
```

Each numeric suffix in *expression1* and *expression2* correspond to a level of precedence. In this case, the *expression1* operators are at a lower level than the *expression2* operators. These rules, of course, are naturally right associative. If we wish operators at a given level to be right associative, we use the normal recursive implementation of the rule. On the other hand, if we wish to have all the operators at a level be left associative, we can implement the rule using the iterative approach described above.