Assessatity 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 expression. 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:

\[
\text{expression} : \text{expression} \ \text{operator} \ \text{expression} \ | \ \text{unary} \\
\text{operator} : \text{PLUS} \ | \ \text{MINUS} \ | \ \text{TIMES} \ | \ \text{DIVIDES} \\
\text{unary} : \text{VARIABLE} \ | \ \text{NUMBER} \ | \ ( \ \text{expression} \ ) \ | \ - \ \text{unary}
\]

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:

\[
\text{expression} : \ \text{unary} \ \text{operator} \ \text{expression} \ | \ \text{unary}
\]

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

\[
\text{expression} : \ \text{expression} \ \text{operator} \ \text{unary} \ | \ \text{unary}
\]

If we wish to implement the left-associative rule using recursive descent parsing techniques, we quickly run into trouble:
Remember, the `cons` function constructs a new lexeme with the first argument giving the type of the lexeme, the second argument giving the left pointer and the third argument giving the right pointer.

With the above code, 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:

```javascript
function expression()
{
    var u,o,e;
    u = unary();
    if (operatorPending())
    {
        o = operator();
        e = expression();
        return cons(o.type,u,e);
    }
    else
    return u;
}
```

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:

```javascript
function expression()
{
    var u,o,e;
    u = unary();
    while (operatorPending())
    {
        o = operator();
        e = unary();
        u = cons(o.type,u,e); //reset u to the growing parse tree
    }
    return u;
}
```

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:

\[ 2 + 3 \times 5 + 6 \]

would be equivalent to:

\[ (((2 + 3) \times 5) + 6) \]

The structure we would like to see is:

\[ ((2 + (3 \times 5)) + 6) \]

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. Suppose, rather than have the operators be right associative, we wish them default to left associativity. We simply flip expression1 and expression2 in the rules, yielding:

expression1 : expression1 operator1 expression2 | expression2

operator1 : PLUS | MINUS

expression2 : expression2 operator2 unary | unary

operator2 : TIMES | DIVIDES

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

If we wish to implement this grammar via recursive descent techniques, we transform all the rules to be right recursive, and then we use the associativity tricks describe above for those particular levels we wish to be right-associative. But, simpler is to have the rules naturally right associative and convert particular levels to left-associativity using loops, as described above.

Object orientation (encapsulation)

We would like to have are expression grammar handle the instance variables of objects (which in turn may be objects). Specifically, we would like to parse expressions like:

a . b . c = w . x . y . z;

The dot operator is typically left associative and has the highest precedence. A left-associative parsing function would look something like:

```javascript
function exprDot()
{
  var u,o,e;
  u = unary();
  while (check(DOT))
  {
    o = match(DOT);
    e = unary();
    u = cons(DOT,u,e);
  }
  return u;
}
```

A slight wrinkle occurs when one wishes to make a method call, as in:

p.q(x);

Since a function call is considered a unary, this parses out to:

```
DOT     /
\       /
  p  CALL
   /
    \  /
      q   ARGS
       /
         \  /
           x     null
```

which says use the return value of the call to q to access p. The intended parse would be:

```
CALL     /
\       /
  DOT    ARG
   /
    \  /
      q    x
       /
         \  /
           null
```

which says get the method q from p and then call it. We can fix the incorrect parse by reassembling the parse tree in `exprDot` if the right pointer is a function call:
function exprDot()
{
    var u,o,e;
    u = unary();
    while (check(DOT))
        {
            o = match(DOT);  
            e = unary();    
            u = cons(DOT,u,e);
        }
    if (u.right == CALL)
        return cons(CALL,cons(DOT,u.left,u.right.left),u.right.right);
    else
        return u;
}

This assumes a parse tree for a function call is tagged CALL with the function to be called to the left and the argument list to the right.