Operator Precedence and Associativity

May 21, 2010

Expression languages often come with infix operators. For example:

```
x + y * z
```

It would be useful to be able to write a language module that abstracts the key properties of expression languages so that the module can be reused in different contexts. One of the challenges is to abstract the precedence and associativity rules in order to reflect the following two parse outcomes:

```
x + (y * z)
(x + y) * z
```

Traditional approaches to parsing often use two approaches: (1) encode the precedence rules into the grammar rules; (2) write the parser so that it knows about certain types of operator. Both approaches have disadvantages: (1) complex grammar rules; (2) complex parsing machinery. The XPL parser, in conjunction with the language module approach, allows this to be achieved fairly painlessly without having to (1) encode the precedence in the rules; or (2) requiring the parser to know about operators.

The approach separates the syntax of operators from their semantics. As an example we will encode a language called `exp` with three operators. The syntax of exp is defined as follows:

```
( 1) exp.syntax(semantics) =
( 2)  import semantics { {
( 3)    root       -> e=exp ';' ?legal(e,10,semantics) {e};
( 4)    exp        -> a=atom postExp^(a);
( 5)    postExp(l) -> o=operator r=exp postExp^(binExp(l,o,r));
( 6)    postExp(e) -> { e };
( 7)    atom       -> whitespace c=['a','z'] { var(asString([c])) };
( 8)    operator   -> whitespace o=. op={asString([o])} ?isOp(op) { op};
( 9)    whitespace -> (32 | 10 | 9 | 13)*
(10)  }
(11) }
```

The syntax operator is supplied with a package of semantic definitions at (1) so that the semantics can be changed when the module is instantiated. The semantics package is imported at (2) to allow field reference without further qualification in the grammar definition. The starting non-terminal (3) parses

an expression, a terminator (;) and then checks that the expression is legal before returning it. An expression (4) is an atom followed by a post-atom check (for operators); an atom (7) is just a single character variable name.

The key processing takes place at (5) and (6) where postExp takes an expression and returns another expression (possibly after consuming more input). At (5), if an operator is detected then a right-hand operand expression is consumed. Then to allow for further operators, the postExp rule is called again. The rule (7) returns e with no further input being consumed.

The effect of (5) and (6) is to specify that all possible configurations of atomic expressions and operators are legal expressions. Recall that postExp can succeed without consuming any input *even if an operator is present.*Therefore, postExp is non-determinstic: we can think of it as performing *all possible parses*, which in this case means all possible associativity groupings of infix operators.

The legality check at (3) permits only those expressions e that satisfy the test. Since (5) and (6) have generated all possible groupings, the predicate `legal` at (3) is satisfied by only those that satisfy the test.

Now set up the semantics for the module. The default constructors will use XPL syntax constructors:

```
exp.semantics.binExp(l,o,r) = BinExp(l,o,r)
exp.semantics.var(n) = Var(n)
```

Recall that we are defining a language module for infix arithmetic expressions and that the operator set has been defined using a predicate `isOp`:

```
exp.semantics.isOp(o) =
  case o {
    '+' -> true;
    '*' -> true;
    o   -> false
  }
```

Furthermore, the operator associativities and precedences are defined as part of the semantics. The associativity is either left or right which determines whether parentheses are inserted on the left or the right of an operator when 2 or more operators of the same precedence occur at the same level in an expression. The precedence of an operator determines how tightly it binds its operands: the lower the numeric value the tighter it binds:

```
exp.semantics.isLeft(o) =
  case o {
    '*' -> true;
    '+' -> false
  }

exp.semantics.prec(o) =
  case o {
    '*' -> 9;
    '+' -> 8
  }
```

Finally, we need to define the predicate `legal`. This must check that the associativity rules are correct and that the operator precedence rules are correct in the supplied expression. The definition is by case analysis on the supplied expression and is explained in the collents given below:

```
// supply the expression, the current precedence level and the
// package of semantics...
exp.semantics.legal(exp,p,semantics) =
  // open the package of semantic operators to get unqualified
  // access to prec, isLeft, etc...
  import semantics {
    case exp {
      // two operators of the same precedence must be
      // right-associative to have the following shape...
      BinExp(l,o,BinExp(ll,oo,rr)) when prec(o) = prec(oo) ->
        isLeft(o) = false and
        (prec(o) < p or prec(o) = p) and
        legal(l,prec(o),semantics) and
        legal(BinExp(ll,oo,rr),prec(oo),semantics);
      // two operators of the same precedence must be left
      // associative to have the following shape...
      BinExp(BinExp(l,o,r),oo,rr) when prec(o) = prec(oo) ->
        isLeft(o) and
        (prec(o) < p or prec(o) = p) and
        legal(BinExp(l,o,r),prec(o),semantics) and
        legal(rr,prec(oo),semantics);
      // a binary operator must have a binding precedence
      // that is tighter than or equal to p...
      BinExp(l,o,r) ->
        if prec(o) < p or prec(o) = p
        then legal(l,prec(o),semantics) and legal(r,prec(o),semantics)
        else false;
      // otherwise we are ok...
      x -> true
    }
  }
}
```

Given the definitions above:

```
exp.syntax(exp.semantics).parse('a␣+␣b␣+␣c;')
=> [| a + (b + c) |]
```

Now change the associativity:

```
exp.semantics.isLeft('+')
=> true
exp.syntax(exp.semantics).parse('a␣+␣b␣+␣c;')
=> [| (a + b) + c |]
```

Now change the precedence and introduce another operator into the expression:

```
exp.syntax(exp.semantics).parse('a␣+␣b␣*␣c␣+␣d;')
```

```
=> [| (a + b) * (c + d) |]
```

Change the precedence:

```
exp.semantics.prec('+')
=> 9
exp.semantics.prec('*')
=> 8
exp.syntax(exp.semantics).parse('a␣+␣b␣*␣c␣+␣d;')
=> [| (a + (b * c)) + d |]
```

Finally, change the associativity back to right:

```
exp.semantics.isLeft('+')
=> false
exp.syntax(exp.semantics).parse('a␣+␣b␣*␣c␣+␣d;')
=> [| a + ((b * c) + d) |]
```

In conclusion we have shown that the XPL parser and language modules allow a convenient separation of concerns between the syntactic structure of infix binary operators and their semantics, where the semantics includes the associativity and operator precedence.