Part 3: Operator Precedence

We saw in the previous part of our compiler writing journey that a parser doesn't necessarily enforce the semantics of our language. It only enforces the syntax and structural rules of the grammar.

We ended up with code that calculates the wrong value of expressions like `2 * 3 + 4 * 5`, because the code created an AST that looks like:

```
   *
  / \
 2   +
    / \
   3   *
      / \
     4   5
```

instead of:

master acwj / 03_Precedence / Readme.md Top

Preview Code Blame Raw

```
     *       *
    / \     / \
   2   3   4   5
```

To solve this, we have to add code to our parser to perform operator precedence. There are (at least) two ways of doing this:

- Making the operator precedence explicit in the language's grammar
- Influencing the existing parser with an operator precedence table

Making the Operator Precedence Explicit

Here is our grammar from the last part of the journey:

```
expression: number
          | expression '*' expression
          | expression '/' expression
          | expression '+' expression
          | expression '-' expression
          ;

number:  T_INTLIT
         ;
```

Note that there is no differentiation between any of the four maths operators. Let's tweak the grammar so that there is a difference:

```
expression: additive_expression
    ;

additive_expression:
      multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;

multiplicative_expression:
      number
    | number '*' multiplicative_expression
    | number '/' multiplicative_expression
    ;

number:  T_INTLIT
         ;
```

We now have two types of expressions: *additive* expressions and *multiplicative* expressions. Note that the grammar now forces the numbers to be part of multiplicative expressions only. This forces the '*' and '/' operators to bind more tightly to the numbers on either side, thus having higher precedence.

Any additive expression is actually either a multiplicative expression by itself, or an additive (i.e. multiplicative) expression followed by a '+' or '-' operator then another multiplicative expression. The additive expression is now at a much lower predencence than the multiplicative expression.

Doing The Above in the Recursive Descent Parser

How do we take the above version of our grammar and implement it into our recursive descent parser? I've done this in the file `expr2.c` and I'll cover the code below.

The answer is to have a `multiplicative_expr()` function to deal with the '*' and '/' operators, and an `additive_expr()` function to deal with the lower precedence '+' and '-' operators.

Both functions are going to read in something and an operator. Then, while there are following operators at the same precedence, each function will parse some more of the input and combine the left and right halves with the first operator.

However, `additive_expr()` will have to defer to the higher-precedence `multiplicative_expr()` function. Here is how this is done.

`additive_expr()`

```
// Return an AST tree whose root is a '+' or '-' binary operator
struct ASTnode *additive_expr(void) {
  struct ASTnode *left, *right;
  int tokentype;

  // Get the left sub-tree at a higher precedence than us
  left = multiplicative_expr();

  // If no tokens left, return just the left node
  tokentype = Token.token;
  if (tokentype == T_EOF)
    return (left);

  // Loop working on token at our level of precedence
  while (1) {
    // Fetch in the next integer literal
    scan(&Token);

    // Get the right sub-tree at a higher precedence than us
    right = multiplicative_expr();

    // Join the two sub-trees with our low-precedence operator
    left = mkastnode(arithop(tokentype), left, right, 0);
```

```
    // And get the next token at our precedence
    tokentype = Token.token;
    if (tokentype == T_EOF)
      break;
  }

  // Return whatever tree we have created
  return (left);
}
```

Right at the beginning, we immediately call `multiplicative_expr()` in case the first operator is a high-precedence '*' or '/'. That function will only return when it encounters a low-precedence '+' or '-' operator.

Thus, when we hit the `while` loop, we know we have a '+' or '-' operator. We loop until there are no tokens left in the input, i.e. when we hit the T_EOF token.

Inside the loop, we call `multiplicative_expr()` again in case any future operators are higher precedence than us. Again, this will return when they are not.

Once we have a left and right sub-tree, we can combine them with the operator we got the last time around the loop. This repeats, so that if we had the expression `2 + 4 + 6`, we would end up with the AST tree:

```
      +
     / \
    +   6
   / \
  2   4
```

But if `multiplicative_expr()` had its own higher precedence operators, we would be combining sub-trees with multiple nodes in them.

multiplicative_expr()

```
// Return an AST tree whose root is a '*' or '/' binary operator
struct ASTnode *multiplicative_expr(void) {
  struct ASTnode *left, *right;
  int tokentype;

  // Get the integer literal on the left.
  // Fetch the next token at the same time.
  left = primary();

  // If no tokens left, return just the left node
```

```
  tokentype = Token.token;
  if (tokentype == T_EOF)
    return (left);

  // While the token is a '*' or '/'
  while ((tokentype == T_STAR) || (tokentype == T_SLASH)) {
    // Fetch in the next integer literal
    scan(&Token);
    right = primary();

    // Join that with the left integer literal
    left = mkastnode(arithop(tokentype), left, right, 0);

    // Update the details of the current token.
    // If no tokens left, return just the left node
    tokentype = Token.token;
    if (tokentype == T_EOF)
      break;
  }

  // Return whatever tree we have created
  return (left);
}
```

The code is similar to `additive_expr()` except that we get to call `primary()` to get real integer literals! We also only loop when we have operators at our high precedence level, i.e. '*' and '/' operators. As soon as we hit a low precedence operator, we simply return the sub-tree that we've built to this point. This goes back to `additive_expr()` to deal with the low precedence operator.

Drawbacks of the Above

The above way of constructing a recursive descent parser with explicit operator precedence can be inefficient because of all the function calls needed to reach the right level of precedence. There also has to be functions to deal with each level of operator precedence, so we end up with lots of lines of code.

The Alternative: Pratt Parsing

One way to cut down on the amount of code is to use a Pratt parser which has a table of precedence values associated with each token instead of having functions that replicate the explicit precedence in the grammar.

At this point I highly recommend that you read Pratt Parsers: Expression Parsing Made Easy by Bob Nystrom. Pratt parsers still make my head hurt, so read as much as you can and get comfortable with the basic concept.

`expr.c` : Pratt Parsing

I've implemented Pratt parsing in `expr.c` which is a drop-in replacement for `expr2.c`. Let's start the tour.

Firstly, we need some code to determine the precedence levels for each token:

```
// Operator precedence for each token
static int OpPrec[] = { 0, 10, 10, 20, 20,    0 };
//                    EOF  +   -   *   /  INTLIT

// Check that we have a binary operator and
// return its precedence.
static int op_precedence(int tokentype) {
  int prec = OpPrec[tokentype];
  if (prec == 0) {
    fprintf(stderr, "syntax error on line %d, token %d\n", Line, tokentype);
    exit(1);
  }
  return (prec);
}
```

Higher numbers (e.g. 20) mean a higher precedence than lower numbers (e.g. 10).

Now, you might ask: why have a function when you have a look-up table called `OpPrec[]`? The answer is: to spot syntax errors.

Consider an input that looks like `234 101 + 12`. We can scan in the first two tokens. But if we simply used `OpPrec[]` to get the precedence of the second `101` token, we wouldn't notice that it isn't an operator. Thus, the `op_precedence()` function enforces the correct grammar syntax.

Now, instead of having a function for each precedence level, we have a single expression function that uses the table of operator precedences:

```
// Return an AST tree whose root is a binary operator.
// Parameter ptp is the previous token's precedence.
struct ASTnode *binexpr(int ptp) {
  struct ASTnode *left, *right;
  int tokentype;
```

```
  // Get the integer literal on the left.
  // Fetch the next token at the same time.
  left = primary();

  // If no tokens left, return just the left node
  tokentype = Token.token;
  if (tokentype == T_EOF)
    return (left);

  // While the precedence of this token is
  // more than that of the previous token precedence
  while (op_precedence(tokentype) > ptp) {
    // Fetch in the next integer literal
    scan(&Token);

    // Recursively call binexpr() with the
    // precedence of our token to build a sub-tree
    right = binexpr(OpPrec[tokentype]);

    // Join that sub-tree with ours. Convert the token
    // into an AST operation at the same time.
    left = mkastnode(arithop(tokentype), left, right, 0);

    // Update the details of the current token.
    // If no tokens left, return just the left node
    tokentype = Token.token;
    if (tokentype == T_EOF)
      return (left);
  }

  // Return the tree we have when the precedence
  // is the same or lower
  return (left);
}
```

Firstly, note that this is still recursive like the previous parser functions. This time, we receive the precedence level of the token that was found before we got called. `main()` will call us with the lowest precedence, 0, but we will call ourselves with higher values.

You should also spot that the code is quite similar to the `multiplicative_expr()` function: read in an integer literal, get the operator's token type, then loop building a tree.

The difference is the loop condition and body:

```
multiplicative_expr():
  while ((tokentype == T_STAR) || (tokentype == T_SLASH)) {
    scan(&Token); right = primary();
```

```
    left = mkastnode(arithop(tokentype), left, right, 0);

    tokentype = Token.token;
    if (tokentype == T_EOF) return (left);
  }

binexpr():
  while (op_precedence(tokentype) > ptp) {
    scan(&Token); right = binexpr(OpPrec[tokentype]);

    left = mkastnode(arithop(tokentype), left, right, 0);

    tokentype = Token.token;
    if (tokentype == T_EOF) return (left);
  }
```

With the Pratt parser, when the next operator has a higher precedence than our current token, instead of just getting the next integer literal with `primary()`, we call ourselves with `binexpr(OpPrec[tokentype])` to raise the operator precedence.

Once we hit a token at our precedence level or lower, we will simply:

```
  return (left);
```

This will either be a sub-tree with lots of nodes and operators at a higher precedence that the operator that called us, or it might be a single integer literal for an operator at the same predence as us.

Now we have a single function to do expression parsing. It uses a small helper function to enforce the operator precedence, and thus implements the semantics of our language.

Putting Both Parsers Into Action

You can make two programs, one with each parser:

```
$ make parser                                   # Pratt Parser
cc -o parser -g expr.c interp.c main.c scan.c tree.c

$ make parser2                                  # Precedence Climbing
cc -o parser2 -g expr2.c interp.c main.c scan.c tree.c
```

You can also test both parsers with the same input files from the previous part of our journey:

```
$ make test
(./parser input01; \
 ./parser input02; \
 ./parser input03; \
 ./parser input04; \
 ./parser input05)
15                                      # input01 result
29                                      # input02 result
syntax error on line 1, token 5         # input03 result
Unrecognised character . on line 3      # input04 result
Unrecognised character a on line 1      # input05 result

$ make test2
(./parser2 input01; \
 ./parser2 input02; \
 ./parser2 input03; \
 ./parser2 input04; \
 ./parser2 input05)
15                                      # input01 result
29                                      # input02 result
syntax error on line 1, token 5         # input03 result
Unrecognised character . on line 3      # input04 result
Unrecognised character a on line 1      # input05 result
```

Conclusion and What's Next

It's probably time to step back a bit and see where we've got to. We now have:

- a scanner that recognises and returns the tokens in our language
- a parser that recognises our grammar, reports syntax errors and builds an Abstract Syntax Tree
- a precedence table for the parser that implements the semantics of our language
- an interpreter that traverses the Abstract Syntax Tree depth-first and calculates the result of the expression in the input

What we don't have yet is a compiler. But we are so close to making our first compiler!

In the next part of our compiler writing journey, we will replace the interpreter. In its place, we will write a translator that generates x86-64 assembly code for each AST node that has a maths operator. We will also generate some assembly preamble and postamble to support the assembly code that the generator outputs. Next step