Part 2: Introduction to Parsing

In this part of our compiler writing journey, I'm going to introduce the basics of a parser. As I mentioned in the first part, the job of the parser is to recognise the syntax and structural elements of the input and ensure that they conform to the *grammar* of the language.

We already have several language elements that we can scan in, i.e. our tokens:

- the four basic maths operators: `*`, `/`, `+` and `-`
- decimal whole numbers which have 1 or more digits `0` .. `9`

Now let's define a grammar for the language that our parser will recognise.

BNF: Backus-Naur Form

You will come across the use of BNF at some point if you get into dealing with computer languages. I will just introduce enough of the BNF syntax here to express the grammar we want to recognise.

We want a grammar to express maths expressions with whole numbers. Here is the BNF description of the grammar:

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

number:  T_INTLIT
         ;
```

The vertical bars separate options in the grammar, so the above says:

- An expression could be just a number, or
- An expression is two expressions separated by a '*' token, or
- An expression is two expressions separated by a '/' token, or
- An expression is two expressions separated by a '+' token, or
- An expression is two expressions separated by a '-' token
- A number is always a T_INTLIT token

It should be pretty obvious that the BNF definition of the grammar is *recursive*: an expression is defined by referencing other expressions. But there is a way to *bottom-out" the recursion: when an expression turns out to be a number, this is always a T_INTLIT token and thus not recursive.

In BNF, we say that "expression" and "number" are *non-terminal* symbols, as they are produced by rules in the grammar. However, T_INTLIT is a *terminal* symbol as it is not defined by any rule. Instead, it is an already-recognised token in the language. Similarly, the four maths operator tokens are terminal symbols.

Recursive Descent Parsing

Given that the grammar for our language is recursive, it makes sense for us to try and parse it recursively. What we need to do is to read in a token, then *look ahead* to the next token. Based on what the next token is, we can then decide what path we need to take to parse the input. This may require us to recursively call a function that has already been called.

In our case, the first token in any expression will be a number and this may be followed by maths operator. After that there may only be a single number, or there may be the start of a whole new expression. How can we parse this recursively?

We can write pseudo-code that looks like this:

```
function expression() {
  Scan and check the first token is a number. Error if it's not
  Get the next token
  If we have reached the end of the input, return, i.e. base case
```

```
    Otherwise, call expression()
  }
```

Let's run this function on the input `2 + 3 - 5 T_EOF` where `T_EOF` is a token that reflects the end of the input. I will number each call to `expression()`.

```
expression0:
  Scan in the 2, it's a number
  Get next token, +, which isn't T_EOF
  Call expression()

    expression1:
      Scan in the 3, it's a number
      Get next token, -, which isn't T_EOF
      Call expression()

        expression2:
          Scan in the 5, it's a number
          Get next token, T_EOF, so return from expression2

      return from expression1
  return from expression0
```

Yes, the function was able to recursively parse the input `2 + 3 - 5 T_EOF`.

Of course, we haven't done anything with the input, but that isn't the job of the parser. The parser's job is to *recognise* the input, and warn of any syntax errors. Someone else is going to do the *semantic analysis* of the input, i.e. to understand and perform the meaning of this input.

> Later on, you will see that this isn't actually true. It often makes sense to intertwine the syntax analysis and semantic analysis.

Abstract Syntax Trees

To do the semantic analysis, we need code that either interprets the recognised input, or translates it to another format, e.g. assembly code. In this part of the journey, we will build an interpreter for the input. But to get there, we are first going to convert the input into an abstract syntax tree, also known as an AST.

I highly recommend you read this short explanation of ASTs:

- Leveling Up One's Parsing Game With ASTs by Vaidehi Joshi

It's well written and really help to explain the purpose and structure of ASTs. Don't worry, I'll be here when you get back.

The structure of each node in the AST that we will build is described in `defs.h` :

```
// AST node types
enum {
  A_ADD, A_SUBTRACT, A_MULTIPLY, A_DIVIDE, A_INTLIT
};

// Abstract Syntax Tree structure
struct ASTnode {
  int op;                               // "Operation" to be performed on this tr
  struct ASTnode *left;                 // Left and right child trees
  struct ASTnode *right;
  int intvalue;                         // For A_INTLIT, the integer value
};
```

Some AST nodes, like those with `op` values `A_ADD` and `A_SUBTRACT` have two child ASTs that are pointed to by `left` and `right` . Later on, we will add or subtract the values of the sub-trees.

Alternatively, an AST node with the `op` value A_INTLIT represents an integer value. It has no sub-tree children, just a value in the `intvalue` field.

Building AST Nodes and Trees

The code in `tree.c` has the functions to build ASTs. The most general function, `mkastnode()` , takes values for all four fields in an AST node. It allocates the node, populates the field values and returns a pointer to the node:

```
// Build and return a generic AST node
struct ASTnode *mkastnode(int op, struct ASTnode *left,
                          struct ASTnode *right, int intvalue) {
  struct ASTnode *n;

  // Malloc a new ASTnode
  n = (struct ASTnode *) malloc(sizeof(struct ASTnode));
  if (n == NULL) {
    fprintf(stderr, "Unable to malloc in mkastnode()\n");
    exit(1);
  }
  // Copy in the field values and return it
  n->op = op;
  n->left = left;
  n->right = right;
  n->intvalue = intvalue;
  return (n);
}
```

Given this, we can write more specific functions that make a leaf AST node (i.e. one with no children), and make an AST node with a single child:

```
// Make an AST leaf node
struct ASTnode *mkastleaf(int op, int intvalue) {
  return (mkastnode(op, NULL, NULL, intvalue));
}

// Make a unary AST node: only one child
struct ASTnode *mkastunary(int op, struct ASTnode *left, int intvalue) {
  return (mkastnode(op, left, NULL, intvalue));
```

Purpose of the AST

We are going to use an AST to store each expression that we recognise so that, later on, we can traverse it recursively to calculate the final value of the expression. We do want to deal with the precedence of the maths operators. Here is an example.

Consider the expression `2 * 3 + 4 * 5`. Now, multiplication has higher precedence that addition. Therefore, we want to *bind* the multiplication operands together and perform these operations before we do the addition.

If we generated the AST tree to look like this:

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

then, when traversing the tree, we would perform `2*3` first, then `4*5`. Once we have these results, we can then pass them up to the root of the tree to perform the addition.

A Naive Expression Parser

Now, we could re-use the token values from our scanner as the AST node operation values, but I like to keep the concept of tokens and AST nodes separate. So, to start with, I'm going to have a function to map the token values into AST node operation values. This, along with the rest of the parser, is in `expr.c` :

```
// Convert a token into an AST operation.
int arithop(int tok) {
  switch (tok) {
    case T_PLUS:
      return (A_ADD);
    case T_MINUS:
      return (A_SUBTRACT);
    case T_STAR:
      return (A_MULTIPLY);
    case T_SLASH:
      return (A_DIVIDE);
    default:
      fprintf(stderr, "unknown token in arithop() on line %d\n", Line);
      exit(1);
  }
}
```

The default statement in the switch statement fires when we can't convert the given token into an AST node type. It's going to form part of the syntax checking in our parser.

We need a function to check that the next token is an integer literal, and to build an AST node to hold the literal value. Here it is:

```
// Parse a primary factor and return an
// AST node representing it.
static struct ASTnode *primary(void) {
  struct ASTnode *n;

  // For an INTLIT token, make a leaf AST node for it
  // and scan in the next token. Otherwise, a syntax error
  // for any other token type.
  switch (Token.token) {
    case T_INTLIT:
      n = mkastleaf(A_INTLIT, Token.intvalue);
      scan(&Token);
      return (n);
    default:
      fprintf(stderr, "syntax error on line %d\n", Line);
      exit(1);
    }
  }
```

This assumes that there is a global variable `Token`, and that it already has the most recent token scanned in from the input. In `data.h`:

```
extern_ struct token    Token;
```

and in `main()`:

```
  scan(&Token);                 // Get the first token from the input
  n = binexpr();                // Parse the expression in the file
```

Now we can write the code for the parser:

```
// Return an AST tree whose root is a binary operator
struct ASTnode *binexpr(void) {
  struct ASTnode *n, *left, *right;
  int nodetype;

  // 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
  if (Token.token == T_EOF)
    return (left);

  // Convert the token into a node type
  nodetype = arithop(Token.token);

  // Get the next token in
  scan(&Token);

  // Recursively get the right-hand tree
  right = binexpr();

  // Now build a tree with both sub-trees
  n = mkastnode(nodetype, left, right, 0);
  return (n);
}
```

Notice that nowhere in this naive parser code is there anything to deal with different operator precedence. As it stands, the code treats all operators as having equal precedence. If you follow the code as it parses the expression `2 * 3 + 4 * 5`, you will see that it builds this AST:

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

This is definitely not correct. It will multiply `4*5` to get 20, then do `3+20` to get 23 instead of doing `2*3` to get 6.

So why did I do this? I wanted to show you that writing a simple parser is easy, but getting it to also do the semantic analysis is harder.

Interpreting the Tree

Now that we have our (incorrect) AST tree, let's write some code to interpret it. Again, we are going to write recursive code to traverse the tree. Here's the pseudo-code:

```
interpretTree:
  First, interpret the left-hand sub-tree and get its value
  Then, interpret the right-hand sub-tree and get its value
  Perform the operation in the node at the root of our tree
  on the two sub-tree values, and return this value
```

Going back to the correct AST tree:

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

the call structure would look like:

```
interpretTree0(tree with +):
  Call interpretTree1(left tree with *):
     Call interpretTree2(tree with 2):
       No maths operation, just return 2
     Call interpretTree3(tree with 3):
       No maths operation, just return 3
     Perform 2 * 3, return 6
```

```
Call interpretTree1(right tree with *):
   Call interpretTree2(tree with 4):
     No maths operation, just return 4
   Call interpretTree3(tree with 5):
     No maths operation, just return 5
   Perform 4 * 5, return 20

Perform 6 + 20, return 26
```

Code to Interpret the Tree

This is in `interp.c` and follows the above pseudo-code:

```
// Given an AST, interpret the
// operators in it and return
// a final value.
int interpretAST(struct ASTnode *n) {
  int leftval, rightval;

  // Get the left and right sub-tree values
  if (n->left)
    leftval = interpretAST(n->left);
  if (n->right)
    rightval = interpretAST(n->right);

  switch (n->op) {
    case A_ADD:
      return (leftval + rightval);
    case A_SUBTRACT:
      return (leftval - rightval);
    case A_MULTIPLY:
      return (leftval * rightval);
    case A_DIVIDE:
      return (leftval / rightval);
    case A_INTLIT:
      return (n->intvalue);
    default:
      fprintf(stderr, "Unknown AST operator %d\n", n->op);
      exit(1);
  }
}
```

Again, the default statement in the switch statement fires when we can't interpret the AST node type. It's going to form part of the sematic checking in our parser.

Building the Parser

There is some other code here and the, like the call to the interpreter in `main()` :

```
  scan(&Token);                 // Get the first token from the input
  n = binexpr();                // Parse the expression in the file
  printf("%d\n", interpretAST(n));      // Calculate the final result
  exit(0);
```

You can now build the parser by doing:

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

I've provided several input files for you to test the parser on, but of course you can create your own. Remember, the calculated results are incorrect, but the parser should detect input errors like consecutive numbers, consecutive operators, and a number missing at the end of the input. I've also added some debugging code to the interpreter so you can see which AST tree nodes get evaluated in which order:

```
$ cat input01
2 + 3 * 5 - 8 / 3

$ ./parser input01
int 2
int 3
int 5
int 8
int 3
8 / 3
5 - 2
3 * 3
2 + 9
11

$ cat input02
13 -6+  4*
5
       +
08 / 3

$ ./parser input02
int 13
int 6
int 4
int 5
int 8
int 3
8 / 3
5 + 2
4 * 7
6 + 28
13 - 34
-21

$ cat input03
12 34 + -56 * / - - 8 + * 2

$ ./parser input03
unknown token in arithop() on line 1

$ cat input04
23 +
18 -
45.6 * 2
/ 18

$ ./parser input04
Unrecognised character . on line 3

$ cat input05
23 * 456abcdefg

$ ./parser input05
Unrecognised character a on line 1
```

Conclusion and What's Next

A parser recognises the grammar of the language and checks that the input to the compiler conforms to this grammar. If it doesn't, the parser should print out an error message. As our expression grammar is recursive, we have chosen to write a recursive descent parser to recognise our expressions.

Right now the parser works, as shown by the above output, but it fails to get the semantics of the input right. In other words, it doesn't calculate the correct value of the expressions.

In the next part of our compiler writing journey, we will modify the parser so that it also does the semantic analysis of the expressions to get the right maths results. Next step