Journal of A Simple Compiler

Trganda

I. PART 0: INTRODUCTION

I've decided to go on a compiler writing journey. In the past I've written some assemblers, and I've written a simple compiler for a typeless language. But I've never written a compiler that can compile itself. So that's where I'm headed on this journey.

As part of the process, I'm going to write up my work so that others can follow along. This will also help me to clarify my thoughts and ideas. Hopefully you, and I, will find this useful!s

A. Goals of the journey

Here are my goals, and non-goals, for the journey:

- To write a self-compiling compiler. I think that if the compiler can compile itself, it gets to call itself a real compiler.
- To target at least one real hardware platform. I've seen a few compilers that generate code for hypothetical machines. I want my compiler to work on real hardware. Also, if possible, I want to write the compiler so that it can support multiple backends for different hardware platforms.
- Practical before research. There's a whole lot of research in the area of compilers. I want to start from absolute zero on this journey, so I'll tend to go for a practical approach and not a theory-heavy approach. That said, there will be times when I'll need to introduce (and implement) some theory-based stuff.
- Follow the KISS principle: keep it simple, stupid! I'm definitely going to be using Ken Thompson's Principe here: "When in doubt, use brute force."
- Take a lot of small steps to reach the final goal. I'll break the journey up into a lot of simple steps instead of taking large leaps. This will make each new addition to the compiler a bite-sized and easily digestible thing.

B. Target Language

The choice of a target language is diffcult. If I choose a high-level language like **Python**, **Go** etc., then I'll have to implement a whole pile of libraries and classes as they are built-in to the language.

I could write a compiler for a language like Lisp, but these can be done easily.

Instead, I've fallen back on the old standby and I'm going to write a compiler for a subset of C, enough to allow the compiler to compile itself. C is just a step up from assembly language (for some subset of C, not C18), and this will help make the task of compiling the C code down to assembly somewhat easier. Oh, and I also like C.

C. The Basic of a Compiler's Job

The job of a compiler is to translate input in one language (usually a high-level language) into a different output language (usually a lower-level language than the input). The main steps are:
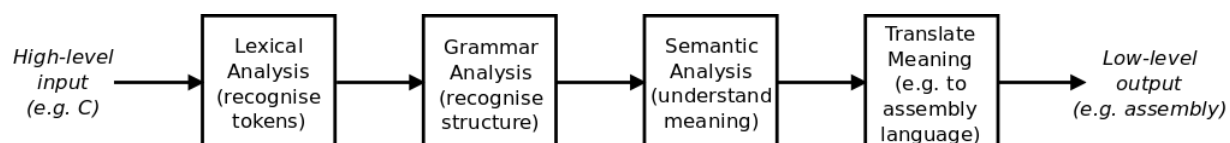
- Do lexical analysis to recognize the lexical elements. In several languages, = is different to ==, so you can't just read a single =. We call these lexical elements *tokens*.
- Parse the input, i.e. recognize the syntax and structural elements of the input and ensure that they conform to the *grammar* of the language. For example, your language might have this decision-making structure:

```
if (x < 23) {
    print("x is smaller than 23\n");
}
```

but in another language you might write:

```
if (x < 23)
    print("x is smaller than 23\n");
```

This is also the place where the compiler can detect syntax errors, like if the semicolon was missing on the end of first *print* statement.
- Do semantic analysis of the input, i.e. understand the meaning of the input. This is actually different from recognising the syntax and structure. For example, in English, a sentence might have the form **¡subject¿ ¡verb¿ ¡adjective¿ ¡object¿**.

```
David ate lovely bananas.
Jennifer hates green tomatoes.
```

- Translate the meaning of the input into a different language. Here we convert the input, parts at a time, into a lower-level language.

D. Resources

There's a lot of compiler resources out on the Internet. Here are the ones I'll be looking at.

1) Learning Resources: If you want to start with some books, papers and tools on compilers, I'd highly recommend this list:

- Curated list of awesome resources on Compilers, Interpreters and Runtimes by Ahmad Alhour

2) Existing Compilers: While I'm going to build my own compiler, I plan on looking at other compilers for ideas and probably also borrow some of their code. Here are the ones I'm looking at:

- SubC by Nils M Holm
- Swieros C Compiler by Robert Swierczek
- fbcc by Fabrice Bellard
- tcc, also by Fabrice Bellard and others
- catc by Yuichiro Nakada
- amacc by Jim Huang
- Small C by Ron Cain, James E. Hendrix, derivatives by others

In particular, I'll be using a lot of the ideas, and some of the code, from the SubC compiler.

E. Setting Up the Development Environment

Assuming that you want to come along on this journey, here's what you'll need. I'm going to use a Linux development environment, so download and set up your favorite Linux system: I'm using Lubuntu 18.04.

I'm going to target two hardware platforms: Intel x86-64 and 32-bit ARM. I'll use a PC running Lubuntu 18.04 as the Intel target, and a Raspberry Pi running Raspbian as the ARM target.

On the Intel platform, we are going to need an existing C compiler. So, install this package. If there are any more tools required for a vanilla Linux system let me know.

Finally, clone a copy of this Github repository.

F. The Next Step

In the next part of our compiler writing journey, we will start with the code to scan our input file and find the *tokens* that are the lexical elements of our language.

II. PART 1: INTRODUCTION TO LEXICAL SCANNING

We start our compiler writing journey with a simple lexical scanner. As I mentioned in the previous part, the job of the scanner is to identify the lexical elements, or *tokens*, in the input language.

We will start with a language that has only five lexical elements:

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

Each token that we scan is going to be stored in this structure (from **defs.h**)

```
// Token structure
struct token {
   int token;
   int intvalue;
};
```

where the **token** field can be one of these values (from **defs.h**):

```
// Tokens
enum {
   T_PLUS, T_MINUS, T_STAR, T_SLASH, T_INTLIT
};
```

When the token is a T_INTLIT (i.e. an integer literal), the $intvalue$ field will hold the value of the integer that we scanned in.

A. Functions in $scan.c$

The $scan.c$ file holds the functions of our lexical scanner. We are going to read in one character at a time from our input file. However, there will be times when we need to "put back" a character if we have read too far ahead in the input stream. We also want to track what line we are currently on so that we can print the line number in our debug messages. All of this is done by the $next()$ function:

```
static int next(void) {
   int c;

   if (Putback) {          // Use the character put
      c = Putback;         // back if there is one
      Putback = 0;
      return c;
   }

   c = fgetc(Infile);      // Read from input file
   if ('\n' == c)
   Line++;                 // Increment line count
   return c;
}
```

The $Putback$ and $Line$ variables are defined in $data.h$ along with our input file pointer:

```
extern_ int Line;
extern_ int Putback;
extern_ FILE *Infile;
```

All C files will include this where *extern_* is replaced with *extern*. But *main.c* will remove the *extern_*; hence, these variables will "belong" to *main.c*.

Finally, how do we put a character back into the input stream? Thus:

```c
// Put back an unwanted character
static void putback(int c) {
    Putback = c;
}
```

B. Ignoring Whitespace

We need a function that reads and silently skips whitespace characters until it gets a non-whitesapce character, and returns it. Thus:

```c
// Skip past input that we don't nee d to deal with,
// i.e. whitesapce, newlines. Return the first
// character we do need to deal with.
static int skip(void) {
    int c;

    c = next();
    while (' ' == c || '\t' == c || '\n' == c || '\r' == c || '\f' == c) {
        c = next();
    }
    return (c);
}
```

C. Scanning Tokens: *scan()*

So new we can read characters in while skipping whitespace; we can also put back a character if we read one character too far ahead. We can now write our first lexical scanner:

```c
// Scan and return the next token found in the input.
// Return 1 if token valid, 0 if no tokens left.
int scan(struct token *t) {
  int c;

  // Skip whitespace
  c = skip();

  // Determine the token based on
  // the input character
  switch (c) {
  case EOF:
   return (0);
  case '+':
   t->token = T_PLUS;
   break;
  case '-':
   t->token = T_MINUS;
   break;
  case '*':
   t->token = T_STAR;
   break;
  case '/':
   t->token = T_SLASH;
```

```
      break;
    default:
      // More here soon
    }

    // We found a token
    return (1);
  }
```

That's it for the simple one-character tokens: for each recognized character, turn it into a token. You may ask: why not just put the recognized character into the *structtoken*? The answer is that later we will need to recognized multi-character tokens such as $==$ and keywords like *if* and *while*. So it will make life easier to have an enumerated list of token values.

D. Integer Literal Values

In fact, we already have to face this situation as we also need to recognize integer literal values like 3827 and 87731. Here is the missing *default* code from the *switch* statement:

```
default:
    // If it's a digit, scan the
    // literal integer value in
    if (isdigit(c)) {
     t->intvalue = scanint(c);
     t->token = T_INTLIT;
     break;
    }

    printf("Unrecognised character %c on line %d\n", c, Line);
    exit(1);
```

Once we hit a decimal digit character, we call the helper function *scanint()* with this first character. It will return the scanned integer value. To do this, it has to read each character in turn, check that it's a legitimate digit, and build up the final number. Here is the code:

```
// Scan and return an integer literal
// value from the input file. Store
// the value as a string in Text.
static int scanint(int c) {
   int k, val = 0;

   // Convert each character into an int value
   while ((k = chrpos("0123456789", c)) >= 0) {
      val = val * 10 + k;
      c = next();
   }

   // We hit a non-integer character, put it back.
   putback(c);
   return val;
}
```

We start with a zero *val* value. Each time we get a character in the set 0 to 9, we covert this to an *int* value with *chrpos()*. We make *val* 10 times bigger and then add this new digit to it.

For example, if we have the character 3, 2, 8, we do:

- $val = 0 * 10 + 3$, i.e. 3

- $val = 3 * 10 + 2$, i.e. 32
- $val = 32 * 10 + 8$, i.e. 328

Right at the end, did you notice the call to $putback(c)$? We found a character that's not a decimal digit at this point. We can't simply discard it, but luckily we can put it back in the input stream to be consumed later.

You may also ask at this point: why not simply substract the ASCII value of 0 from c to make it an integer? The answer is that, later on, we will be able to do $chrpos("0123456789abcdef")$ to convert hexadecimal digits as well.

Here's the code for $chrpos()$:

```c
// Return the position of character c
// in string s, or -1 if c not found
static int chrpos(char *s, int c) {
  char *p;

  p = strchr(s, c);
  return (p ? p - s : -1);
}
```

And that's it for the lexical scanner code in *scan.c* for now.

E. Putting the Scanner to Work

The code in *main.c* puts the above scanner to work. The $main()$ function opens up a file and then scans it for tokens:

```c
void main(int argc, char *argv[]) {
    ...
    init();
    ...
    Infile = fopen(argv[1], "r");
    ...
    scanfile();
    exit(0);
}
```

And $scanfile()$ loops while there is new token and prints out the details of the token:

```c
// List of printable tokens
char *tokstr[] = { "+", "-", "*", "/", "intlit" };

// Loop scanning in all the tokens in the input file.
// Print out details of each token found.
static void scanfile() {
    struct token T;

    while (scan(&T)) {
        printf("Token %s", tokstr[T.token]);
        if (T.token == T_INTLIT)
            printf(", value %d", T.intvalue);
        printf("\n");
    }
}
```

F. Some Example Input Files

I've provided some example input files so you can see what tokens the scanner finds in each file, and what input files the scanner rejects.

```
$ make
cc -o scanner -g main.c scan.c

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

$ ./scanner input01
Token intlit, value 2
Token +
Token intlit, value 3
Token *
Token intlit, value 5
Token -
Token intlit, value 8
Token /
Token intlit, value 3

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

$ ./scanner input04
Token intlit, value 23
Token +
Token intlit, value 18
Token -
Token intlit, value 45
Unrecognised character . on line 3
```

G. Conclusion and What's Next

We've started small and we have a simple lexical scanner that recognizes the four main maths operators and also integer values. We saw that we needed to skip whitespace and put back characters if we read too far into the input.

Single character tokens are easy to scan, but multi-character tokens are a bit harder. But at the end, the $scan()$ function returns the next token from the input file in a *structtoken* variable.

In the next part of our compiler writing journey, we will build a recursive descent parser to interpret the grammar of our input files, and calculate & print out the final value for each file.

III. 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 recognize the syntax and structural elements of the input and ensure that they conform to the *grammer* of the language.

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

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

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

A. 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 recognize.

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, or
- 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 $non-termainl$ 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-recognized token in the language. Similarly, the four maths operator tokens are terminal symbols.

B. 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 $lookahead$ 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 look 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 $2 + 3 - 5 \quad T\_EOF$ 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 $2 + 3 - 5 \quad T\_EOF$.

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.

C. Abstract Syntax Trees

To do the semantic analysis, we need code that either interprets the recoginsed 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 tree
  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.

D. 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:

```c
// 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:

```c
// 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));
}
```

E. 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.

F. 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*:

```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:

```c
// 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*:

```c
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:

```c
// 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 parser 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 ge 23 insetad of doing $2 * 3$ to get 6.

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

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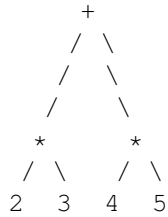
```
Then, interpret the right-hand sub-tree and get its value
Perform the operation in the node at the root of our tree
on the two sub-tree values, and return this value
```

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
```

H. Code to Interpret the Tree

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

```c
// 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.

I. *Building the Parser*

There is some other code here, it's looks 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 curse you can create you 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
```

J. Conclusion and What's Next

A parser recognises the grammar of the language and checks that the input to the compiler conforms to the 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 expression.

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.

IV. 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:
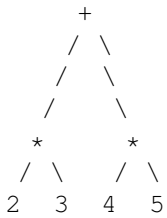
```
  *
 / \
2   +
   / \
  3   *
     / \
    4   5
```

instead of:

```
    +
   / \
  /   \
 /     \
 *       *
/ \   / \
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

A. *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 operator. 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 expression: *additive* expressions and *multiplicative* expressions. Note that the grammar now forces the numbers to be part of multiplicative expressions only. This forces the ∗ and / operator 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 precedence than the multiplicative expression.

B. 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 $-$ operator.

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.

C. additive_expr()

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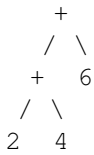
But if *multiplicative_expr*() had its own higher precedence operators, we would be combining sub-trees with multiple nodes in them.

D. multiplicative_expr()

```c
// 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.

E. 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.

F. The Alternative: Pratt Parsing (Operator-precedence parser)

One way to cut down on the amount of code is to use a Operator-precedence parser which has a table of precedence values associated with each token instead of having 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.

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 determin 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 $234 \quad 101 + 12$. 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 precedence as us. Now we have a single function to enforce the operator precedence, and thus implements the semantics of our language.

G. 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 parser 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
```

H. 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 recognizes and returns the tokens in our language
- a parser that recognizes 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 compilers! 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.

V. Part 4: An Actual Compiler

It's about time that I met my promise of actually writing a compiler. So in this part of the journey we are going to replace the interpreter in our program with code that generates x86-64 assembly code.

A. Revising the Interpreter

Before we do, it will be worthwhile to revisit the interpreter code in *interp.c*:

```c
int interpretAST(struct ASTnode *n) {
  int leftval, rightval;

  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);
  }
}
```

The *interpretAST()* function walks the given AST tree depth-first. It evaluates any left sub-tree, then the right sub-tree. Finally, it uses the *op* value at the base of the current tree to operate on these children. If the *op* value is one of the four math operators, then this math operation is performed. If the *op* value indicates that the node is simply an integer literal, the literal value is return. The function returns the final value for this tree. And, as it is recursive, it will calculate the final value for a whole tree one sub-tree at a time.

B. Changing to Assembly Code Generation

We are going to write an assembly code generator which is generic. This is, in turn, going to call out to a set of CPU-specific code generation functions. Here is the generic assembly code generator in *gen.c*:

```c
// Given an AST, generate
// assembly code recursively
static int genAST(struct ASTnode *n) {
  int leftreg, rightreg;

  // Get the left and right sub-tree values
  if (n->left) leftreg = genAST(n->left);
  if (n->right) rightreg = genAST(n->right);

  switch (n->op) {
    case A_ADD:  return (cgadd(leftreg,rightreg));
    case A_SUBTRACT: return (cgsub(leftreg,rightreg));
    case A_MULTIPLY: return (cgmul(leftreg,rightreg));
    case A_DIVIDE: return (cgdiv(leftreg,rightreg));
    case A_INTLIT: return (cgload(n->intvalue));

    default:
```

```
        fprintf(stderr, "Unknown AST operator %d\n", n->op);
        exit(1);
    }
  }
```

Looks familar, huh?! We are doing the same depth-first tree traversal. This time:

- A_INTLIT: load a register with the literal value
- Other operators: perform a math function on the two registers that hold the left-child's and right-child's value

Instead of passing values, the code in $genAST()$ passes around register identifiers. For example $cgload()$ loads a value into a register and returns the identify of the register with the loaded value. $genAST()$ itself returns the identify of the register that holds the final value of the tree at this point. That's why the code at the top is getting register identities:

```
    if (n->left) leftreg = genAST(n->left);
    if (n->right) rightreg = genAST(n->right);
```

C. Calling $genAST()$

$genAST$ is only going to calculate the value of the expression given to it. We need to print out this final calculation. We're also going to need to wrap the assembly code we generate with some leading code (the *preamble*) and some trailing code (the *postamble*). This is done with the other function in *gen.c*:

```
    void generatecode(struct ASTnode *n) {
      int reg;

      cgpreamble();
      reg= genAST(n);
      cgprintint(reg); // Print the register with the result as an int
      cgpostamble();
    }
```

D. The x86-64 Code Generator

That's the generic code generator ou of the road. Now we need to look at the generation of some real assembly code. For now, I'm targetting the x86-64 CPU as this is still one of the most common Linux platforms. So, open up *cp.c* and let's get browsing.

1) Allocating Registers: Any CPU has a limited number of registers. We will have to allocate a register to hold the integer literal values, plus any calculation that we perform on them. However, once we've used a value, we can often discard the value and hence free up the register holding it. Then we can re-use that register for another value. There are three functions that deal with register allocation:

- $freeall_registers()$: Set all registers as available
- $alloc_register()$: Allocate a free register
- $free_register()$: Free an allocated register

I'm not going to go through the code as it's straight forward but with some error checking. Right now, if I run out of registers then the program will crash. Later on, I'll deal with the situation when we have run out of free registers. The code works on generic registers: r0, r1, r2 and r3. There is a table of strings with the actual register name:

```
    static char *reglist[4]= { "%r8", "%r9", "%r10", "%r11" };
```

This makes these functions fairly independent of CPU architecture.

2) Loading a Register: This is done in $cgload()$: aregister is allocated, then a $movq$ instruction loads a literal value into the allocated register.

```c
// Load an integer literal value into a register.
// Return the number of the register
int cgload(int value) {

  // Get a new register
  int r= alloc_register();

  // Print out the code to initialise it
  fprintf(Outfile, "\tmovq\t$%d, %s\n", value, reglist[r]);
  return(r);
}
```

3) Adding Two Registers: $cgadd()$ takes two register numbers and generates the code to add them together. The result is saved in one of the two registers, and the other one is then freed for future use:

```c
// Add two registers together and return
// the number of the register with the result
int cgadd(int r1, int r2) {
  fprintf(Outfile, "\taddq\t%s, %s\n", reglist[r1], reglist[r2]);
  free_register(r1);
  return(r2);
}
```

Note that addition is *commutative*, so I could have added $r2$ to $r1$ instead $r1$ to $r2$. The identity of the register with the final value is returned.

4) Multiplying Two Registers: This is very similar to addition, and again the operation is *commutative*, so any register can be returned:

```c
// Multiply two registers together and return
// the number of the register with the result
int cgmul(int r1, int r2) {
  fprintf(Outfile, "\timulq\t%s, %s\n", reglist[r1], reglist[r2]);
  free_register(r1);
  return(r2);
}
```

5) Subtracting Two Registers: Subtraction is *not* commutative: we have to get the order correct. The second register is subtracted from the first, so we return the first and free the second:

```c
// Subtract the second register from the first and
// return the number of the register with the result
int cgsub(int r1, int r2) {
  fprintf(Outfile, "\tsubq\t%s, %s\n", reglist[r2], reglist[r1]);
  free_register(r2);
  return(r1);
}
```

6) Dividing Two Registers: Division is also not commutative, so the previous notes apply. On the x86-64, it's even more complicated. We need to load $\%rax$ with the *dividend* from $r1$. This needs to be extended to eight bytes with cqo. Then, $idivq$ will divide $\%rax$ with the divisor in $r2$, leaving the *quotient* in $\%rax$, so we need to copy it out to either $r1$ or $r2$. Then we can free the other register.

```c
// Divide the first register by the second and
// return the number of the register with the result
```

```c
int cgdiv(int r1, int r2) {
  fprintf(Outfile, "\tmovq\t%s,%%rax\n", reglist[r1]);
  fprintf(Outfile, "\tcqo\n");
  fprintf(Outfile, "\tidivq\t%s\n", reglist[r2]);
  fprintf(Outfile, "\tmovq\t%%rax,%s\n", reglist[r1]);
  free_register(r2);
  return(r1);
}
```

7) Printing A Register: There isn't an x86-64 instruction to print a register out as a decimal number. To solve this probelm, the assembly preamble contains a function called $printint()$ that takes a register argument and calls $printf()$ to print this out in decimal. I'm not going to give the code in $cgpreamble()$, but it also contains the beginning code for $main()$, so that we can assemble our output file to get a complete program. The code for $cgpostamble()$, also not given here, simply calls $exit(0)$ to end the program. Here, however, is $cgprintint()$:

```c
void cgprintint(int r) {
  fprintf(Outfile, "\tmovq\t%s, %%rdi\n", reglist[r]);
  fprintf(Outfile, "\tcall\tprintint\n");
  free_register(r);
}
```

Linux x86-64 expects the first argument to a function to be in the $\%rdi$ register, so we move our register into $\%rdi$ before we $callprintint$.

E. Doing Our First Compile

That's about it for the x86-64 code generator. There is some extra code in $main()$ to open out $out.s$ as our output file. I've also left the interpreter in the program so we can confirm that our assembly calculates the same answer for the input expression as the interpreter. Let's make the compiler and run it on $input01$:

```
$ make
cc -o comp1 -g cg.c expr.c gen.c interp.c main.c scan.c tree.c

$ make test
./comp1 input01
15
cc -o out out.s
./out
15
```

Yes! The first 15 is the interpreter's output. The second 15 is the assembly's output.

VI. PART 5: STATEMENTS

It's time to add some "proper" statements to the grammar of our language. I want to be able to write lines of code like this:

```
print 2 + 3 * 5;
print 18 - 6/3 + 4*2;
```

Of course, as we are Ignoring whitespace, there's no necessity that all the tokens for one statement are on the same line. Each statement starts with the keyword $print$ and is terminated with a semicolon. So these are going to become nwe tokens in our language.

A. BNF Description of the Grammar

We've already seen the BNF notation for expressions. Now let's define the BNF syntax for the above types of statements:

```
statements: statement
  | statement statements
  ;

statement: 'print' expression ';'
  ;
```

An input file consists of serveral statements. They are either one statement, or a statement followed by more statements. Each statement starts with the keyword *print*, then one expressioni, then a semicolon.

B. Changs to the Lexical Scanner

Before we can get to the code that parses the above syntax, we need to add a few more bits and pieces to the existing code. Let's start with the lexical scanner. Adding a token for semicolons will be easy. Now, the *print* keyword. Later on, we'll have many keywords in the language, plus identifiers for our variables, so we'll need to add some code which helps us to deal with them.

In *scan.c*, I've added this code which I've borrowed from the SubC compiler. It reads in alphanumeric character into a buffer until it hits a non-alphanumeric character.

```c
// Scan an identifier from the input file and
// store it in buf[]. Return the identifier's length
static int scanident(int c, char *buf, int lim) {
  int i = 0;

  // Allow digits, alpha and underscores
  while (isalpha(c) || isdigit(c) || '_' == c) {
    // Error if we hit the identifier length limit,
    // else append to buf[] and get next character
    if (lim - 1 == i) {
      printf("identifier too long on line %d\n", Line);
      exit(1);
    } else if (i < lim - 1) {
      buf[i++] = c;
    }
    c = next();
  }
  // We hit a non-valid character, put it back.
  // NUL-terminate the buf[] and return the length
  putback(c);
  buf[i] = '\0';
  return (i);
}
```

We also need a function to recognize keywords in the language. One way would be to have a list of keywords, and to walk the list and *strcmp()* each one against the buffer from *scanident()*. The code from SubC has an optimisation: match against the first letter before doing the *strcmp()*. This speeds up the comparison against dozens of keywords. Right now we don't need this optimisation but I've put it in for later:

```c
// Given a word from the input, return the matching
// keyword token number or 0 if it's not a keyword.
// Switch on the first letter so that we don't have
```

```c
// to waste time strcmp()ing against all the keywords.
static int keyword(char *s) {
  switch (*s) {
    case 'p':
      if (!strcmp(s, "print"))
        return (T_PRINT);
      break;
  }
  return (0);
}
```

Now, at the bottom of switch statement in $scan()$, we add this code to recognize semicolons and keywords:

```c
case ';':
  t->token = T_SEMI;
  break;
default:

  // If it's a digit, scan the
  // literal integer value in
  if (isdigit(c)) {
    t->intvalue = scanint(c);
    t->token = T_INTLIT;
    break;
  } else if (isalpha(c) || '_' == c) {
    // Read in a keyword or identifier
    scanident(c, Text, TEXTLEN);

    // If it's a recognised keyword, return that token
    if (tokentype = keyword(Text)) {
      t->token = tokentype;
      break;
    }
    // Not a recognised keyword, so an error for now
    printf("Unrecognised symbol %s on line %d\n", Text, Line);
    exit(1);
  }
  // The character isn't part of any recognised token, error
  printf("Unrecognised character %c on line %d\n", c, Line);
  exit(1);
```

I've also added a global $Text$ buffer to store the keywords and identifiers:

```c
#define TEXTLEN    512          // Length of symbols in input
extern_ char Text[TEXTLEN + 1]; // Last identifier scanned
```

C. Changes to the Expression Parser

Up to now our input files have contained just a single expression; therefore, in our Pratt parser code in $binexpr()$ (in $expr.c$), we had this code to exit the parser:

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

With our new grammar, each expression is terminated by a semicolon. Thus, we need to change the code in the expression parser to spot the T_SEMI tokens and exit the expression parsing:

```c
// 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 we hit a semicolon, return just the left node
  tokentype = Token.token;
  if (tokentype == T_SEMI)
    return (left);

    while (op_precedence(tokentype) > ptp) {
     ...

        // Update the details of the current token.
    // If we hit a semicolon, return just the left node
    tokentype = Token.token;
    if (tokentype == T_SEMI)
      return (left);
    }
}
```

D. Changes to the Code Generator

I want to keep the generic code generator in *gen.c* separate from the CPU-specific code in *cg.c*. That also means that the rest of the compiler should only ever call the functions in *gen.c*, and only *gen.c* should call the code in *cg.c*. To this end, I've defined some new "front-end" funcitons in *gen.c*:

```c
void genpreamble()  { cgpreamble(); }
void genpostamble() { cgpostamble(); }
void genfreeregs()  { freeall_registers(); }
void genprintint(int reg) { cgprintint(reg); }
```

E. Adding the Parser for Statements

We have a new file *stmt.c*. This will hold the parsing code for all the main statements in our language. Right now, we need to parse the BNF grammar for statements which I gave up above. This is done with this single function. I've converted the recursive definition into a loop:

```c
// Parse one or more statements
void statements(void) {
  struct ASTnode *tree;
  int reg;

  while (1) {
    // Match a 'print' as the first token
    match(T_PRINT, "print");
```

```
    // Parse the following expression and
    // generate the assembly code
    tree = binexpr(0);
    reg = genAST(tree);
    genprintint(reg);
    genfreeregs();

    // Match the following semicolon
    // and stop if we are at EOF
    semi();
    if (Token.token == T_EOF)
      return;
  }
}
```

In each loop, the code finds a T_PRINT token. It then calls $binexpr()$ to parse the expression. Finally, it finds the T_SEMI token. If a T_EOF token follows, we break out of the loop. After each expression tree, the code in $gen.c$ is called to convert the tree into assembly code and to call the assembly $printint()$ function to print out the final value.

F. Some Helper Functions

There are a couple of new helper functions in the above code, which I've put into a new file, $misc.c$:

```
// Ensure that the current token is t,
// and fetch the next token. Otherwise
// throw an error
void match(int t, char *what) {
  if (Token.token == t) {
    scan(&Token);
  } else {
    printf("%s expected on line %d\n", what, Line);
    exit(1);
  }
}

// Match a semicon and fetch the next token
void semi(void) {
  match(T_SEMI, ";");
}
```

These form part of the syntax checking in the parser. Later on, I'll add more short functions to call $match()$ to make our syntax checking easier.

G. Changes to $main()$

$main()$ used to call $binexpr()$ directly to parse the single expression in the old input files. Now it does this:

```
scan(&Token);          // Get the first token from the input
genpreamble();         // Output the preamble
statements();          // Parse the statements in the input
genpostamble();        // Output the postamble
fclose(Outfile);       // Close the output file and exit
exit(0);
```

H. Tring It Out

That's about it for the new and changed code. Let's give the new code a whirl. Here is the new input file, *input*01:

```
print 12 * 3;
print
   18 - 2
     * 4; print
1 + 2 +
  9 - 5/2 + 3*5;
```

Yes I've decide to check that we have tokens spread out across multiple lines. To compile and run the input file, do a *maketest*:

```
$ make test
cc -o comp1 -g cg.c expr.c gen.c main.c misc.c scan.c stmt.c tree.c
./comp1 input01
cc -o out out.s
./out
36
10
25
```

And it works!

I. Conclusion and What's Next

We've added our first "real" statement grammar to our language. I've defined it in BNF notaion, but it was easier to implement it with a loop and not recursively. Don't worry. we'll go back to doing recursive parsing soon. Along the way we had to modify the scanner, add support for keywords and identifiers, and to more cleanly separate the generic code generator and the CPU-specific generator. In the next part of our compiler writing journey, we will add variables to the language. This will require a significant amout of work.