acwj

Trganda

May 31, 2021

1 Part 0: Introducation

I've decided to go on a compiler writing journey. In the past I've written some assemblers, and I've written a simaple 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!

1.1 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.
- Pratical 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.

1.2 Target Language

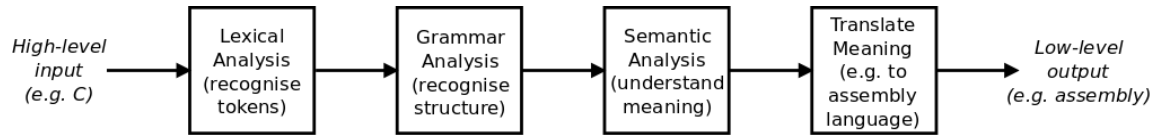
The choice of a target language is diffcult. If I choose a high-level language like **Python**, **Go** etc., then I'll have to implement a whole pile of libraries and classes as they are built-in to the language.

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.

1.3 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 recognise 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. recognise 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 cna 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.

1.4 Resources

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

1.4.1 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

1.4.2 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.

1.5 Setting Up the Development Enviroment

Assuming that you want to come along on this journey, here's what you'll need. I'm going to use a Linux development enviroment, so download and set up your favourite 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.

1.6 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.

2 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.

2.1 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:

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

2.2 Ignoring Whitespace

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

```
// 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);
}
```

2.3 Scanning Tokens: *scan()*

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

```
    // 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 recognised character, turn it into a token. You may ask: why not just put the recognised character into the *structtoken*? The answer is that later we will need to recognised 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.

2.4 Integer Literal Values

In fact, we already have to face this situation as we also need to recoginse 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*("0123456789*abcdef*") to convert hexadecimal digits as well.

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

```
// 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.

2.5 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:

```
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:

```
// 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");
    }
}
```

2.6 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
```

2.7 Conclusion and What's Next

We've started small and we have a simple lexical scanner that recognises 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 grammer of our input files, and calculate & print out the final value for each file.

3 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 *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 grammer for the language that our parser will recognise.

3.1 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 grammer to express maths expressions with whole numbers. Here is the BNF description of the grammer:

```
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. Howerer, 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.

3.2 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 *semanticanalysis* 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 interwine the syntax analysis and semantic analysis.

3.3 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.

3.4 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));
}
```

3.5 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.

3.6 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: