Writing a C Compiler, Part 1

This is the first post in a series on writing your own C compiler. Here are some reasons to write a compiler:

1. You'll learn about abstract syntax trees (ASTs) and how programs can represent and manipulate other programs. Handy for working with linters, static analyzers, and metaprogramming of all sorts.
2. You'll learn about assembly, calling conventions, and all the gritty, low-level details of how computers, like, do stuff.
3. It seems like an impossibly hard project (but isn't!), so writing one will make you feel like a badass.

I've been working on my own C compiler, nqcc for the past several weeks, using Abdulaziz Ghuloum's An Incremental Approach to Compiler Construction as a roadmap. I really like Ghuloum's approach: you start by compiling a tiny, trivial subset of your source language all the way down to x86 assembly. Then you add new language features, one step at a time. In step one, you just return constants; in a later step you handle addition and subtraction; and so on. Every step is small enough to feel manageable, and at the end of the every step you have a working compiler.

This series is adapted from Ghuloum's paper - the original paper is about compiling Scheme, so I had to make some adjustments to compile C instead. I'll cover arithmetic operations, conditionals, local variables, function calls, and perhaps more. I've also written some test programs that you can use to validate that each stage of your compiler works correctly.

Preliminaries

Before you start, you need to decide on two things: what language to write your compiler in, and how to handle parsing and lexing. You can implement the compiler in whatever language you like, but I'd recommend using a language with sum types and pattern matching[1], like OCaml, Haskell, or Rust. It will be SO MUCH EASIER to build and traverse an AST if you do. I started writing nqcc in Python, which I know very well, then got fed up and switched to OCaml, which I didn't know well at all, and it was definitely worth it.

You also need to decide whether to write your own parser and lexer or use automatic parser and scanner generators (e.g. flex and bison). In this series of posts, I'll show you how to write a lexer (or scanner) and recursive descent parser by hand. Using a parser generator is probably easier, but I haven't tried it so I could be wrong. You could probably also use a scanner generator for lexing, but hand-write your own parser. Basically, do whatever you like, but I'm only going to talk about hand-writing a lexer and parser for the rest of this series, so if you want to use bison and flex you're on your own.

Update 2/18/19

There's one more thing you need to decide on: whether to target 32-bit or 64-bit assembly. This series uses 32-bit architecture because that's what Ghuloum's paper used. However, I've realized since starting the series that this was a bad call. Because 32-bit architecture is increasingly obsolete, compiling and running 32-bit binaries can be a headache. I've decided to go back and add 64-bit examples to this series when I get the chance. Until I do that, you have one of two options:

1. Figure out on your own how to adapt these posts to a 64-bit instruction set. If you're at all familiar with assembly, this isn't too hard and it's what I'd recommend.
2. Stick with the 32-bit instruction set I've used in these posts. This will require a

little extra work up front, depending on your OS:

  - On Linux, you'll need to install some extra libraries in order to turn your 32-bit assembly into an executable. This Dockerfile lists the libraries you'll need (plus some Scheme-related stuff you can ignore). Many thanks to Jaseem Abid, who had previously worked through Ghuloum's paper, for creating this Dockerfile and telling me about it.
  - 32-bit support is being phased out on macOS, and the next version probably won't let you run 32-bit binaries at all. At the moment, the gcc binary that ships with XCode won't compile 32-bit applications by default. You can just install GCC from Homebrew and use that instead, or you can futz around with XCode and figure out how to make it build 32-bit binaries. I went with the former.

Week 1: Integers

This week, we'll compile a program that returns a single integer. We'll also set up the three basic passes of our compiler. This will be a lot of work for not that much payoff, but the architecture we define now will make it easy to add more language features later on.

Here's a program we'd like to compile - we'll call it return_2.c:

```
int main() {
    return 2;
}
```

We'll only handle programs with a single function, main, consisting of a single return statement. The only thing that varies is the value of the integer being returned. We won't handle hex or octal integer literals, just decimal. To verify that your compiler works correctly, you'll need to compile a program, run it, and check its return code:

```
$ ./YOUR_COMPILER return_2.c # compile the source file shown above
$ ./gcc -m32 return_2.s -o return_2 # assemble it into an executable
$ ./return_2 # run the executable you just compiled
$ echo $? # check the return code; it should be 2
2
```

Your compiler will produce x86 assembly. We won't transform the assembly into an executable ourselves - that's the job of the assembler and linker, which are separate programs[2]. To see how this program looks in assembly, let's compile it with gcc[3]:

```
$ gcc -S -O3 -fno-asynchronous-unwind-tables return_2.c
$ cat return_2.s
    .section __TEXT,__text_startup,regular,pure_instructions
    .align 4
    .globl _main
_main:
    movl    $2, %eax
    ret
    .subsections_via_symbols
```

Now, let's look at the assembly itself. We can ignore the .section, .align and .subsections_via_symbols directives - if you delete them, you can still assemble and run return_2.s[4].

.globl _main indicates that the _main symbol should be visible to the linker; otherwise it can't find the entry point to the program. (If you're on a Unix-like system other than OS X, this symbol will just be main, no underscore.)

Finally, we have our actual assembly instructions:

```
_main:                  ; label for start of "main" function
    movl    $2, %eax    ; move constant "2" into the EAX register
    ret                 ; return from function
```

The most important point here is that when a function returns, the EAX register[5] will contain its return value. The main function's return value will be the program's exit code.

An important side note: throughout this tutorial, I'll use AT&T assembly syntax, because that's what

GCC uses by default. Some online resources might use Intel syntax, which has operands in the reverse order from AT&T syntax. Whenever you're reading assembly, make sure you know what syntax it's using!

The only thing that can change in the snippet of assembly above is the return value. So one very simple approach would be to use a regular expression to extract the return value from the source code, then plug it into the assembly. Here's a 20-line Python script to do that:

```
import sys, os, re

#expected form of a C program, without line breaks
source_re = r"int main\s*\(\s*\)\s*{\s*return\s+(?P<ret>[0-9]+)\s*;\s*}"

# Use 'main' instead of '_main' if not on OS X
assembly_format = """
    .globl _main
_main:
    movl    ${}, %eax
    ret
"""

source_file = sys.argv[1]
assembly_file = os.path.splitext(source_file)[0] + ".s"

with open(source_file, 'r') as infile, open(assembly_file, 'w') as outfile:
    source = infile.read().strip()
    match = re.match(source_re, source)

    # extract the named "ret" group, containing the return value
    retval = match.group('ret')
    outfile.write(assembly_format.format(retval))
```

But parsing the whole program with one big regular expression isn't a viable long-term strategy. Instead, we'll split up the compiler into three stages: lexing, parsing, and code generation. As far as I know, this is a pretty standard compiler architecture, except you'd normally want a bunch of optimization passes between parsing and code generation.

Lexing

The lexer (also called the scanner or tokenizer) is the phase of the compiler that breaks up a string (the source code) into a list of tokens. A token is the smallest unit the parser can understand - if a program is like a paragraph, tokens are like individual words. (Many tokens *are* individual words, separated by whitespace.) Variable names, keywords, and constants, and punctuation like braces are all examples of tokens. Here's a list of all the tokens in return_2.c:

- int keyword
- Identifier "main"
- Open parentheses
- Close parentheses
- Open brace
- return keyword
- Constant "2"
- Semicolon
- Close brace

Note that some tokens have a value (e.g. the constant token has value "2") and some don't (like parentheses and braces). Also note that there are no whitespace tokens. (In some languages, like Python, whitespace is significant and you do need tokens to represent it.)

Here are all the tokens your lexer needs to recognize, and the regular expression defining each of them:

- Open brace {
- Close brace }
- Open parenthesis \(
- Close parenthesis \)
- Semicolon ;
- Int keyword int
- Return keyword return
- Identifier [a-zA-Z]\w*
- Integer literal [0-9]+

If you want, you could just have a "keyword" token type, instead of a different token type for each keyword.

☑ *Task:*

Write a *lex* function that accepts a file and returns a list of tokens. It should work for all stage 1 examples in the test suite, including the invalid ones. (The invalid examples should raise errors in the parser, not the lexer.) To keep things simple, we only lex decimal integers. If you like, you can extend your lexer to handle octal and hex integers too.

You might notice that we can't lex negative integers. That's not an accident - C doesn't have negative integer constants. It just has a negation operator, which can be applied to positive integers. We'll add negation in the next post.

Parsing

The next step is transforming our list of tokens into an abstract syntax tree. An AST is one way to represent the structure of a program. In most programming languages, language constructs like conditionals and function declarations are made up of simpler constructs, like variables and constants. ASTs capture this relationship; the root of the AST will be the entire program, and each node will have children representing its constituent parts. Let's look at a small example:

```
if (a < b) {
    c = 2;
    return c;
} else {
    c = 3;
}
```

This code snippet is an if statement, so we'll label the root of our AST "if statement". It will have three children:

- The condition (a < b)
- The if body (c = 2; return c;)
- The else body (c = 3;)

Each of these components can be broken down further. For example, the condition is a binary < operation with two children:

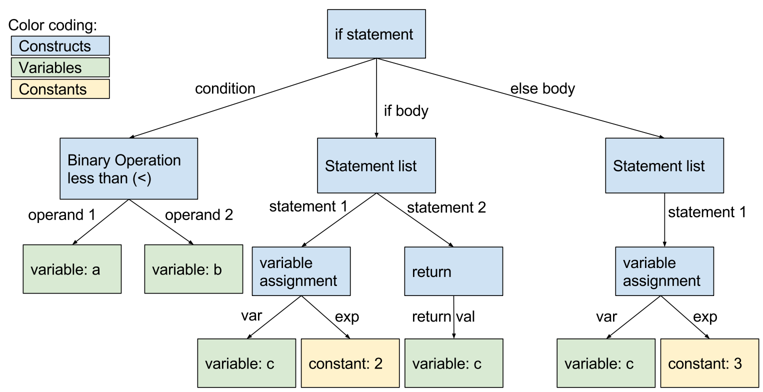
- The first operand (variable a)

- The second operand (variable b)

An assignment statement (like c=2;) also has two children: the variable being updated (c), and the expression assigned to it (2).

The if body, on the other hand, can have an arbitrary number of children - each statement is a child node. In this case it has two children because there are two statements. The children are ordered - c=2; precedes return c; because it comes first in the source code.

Here’s the full AST for this code snippet:



- if statement
    - condition: binary operation (<)
        - operand 1: variable a
        - operand 2: variable b
    - if body: statement list
        - statement 1: assignment
            - variable: c
            - right-hand side: constant 2
        - statement 2: return
            - return value: variable c
    - else body: statement list
        - statement 1: assignment
            - variable: c

- right-hand side:
  constant 3

And here’s pseudocode for constructing this AST:

```
//create if condition
cond = BinaryOp(op='>', operand_1=Var(a), operand_2=Var(b))

//create if body
assign = Assignment(var=Var(c), rhs=Const(2))
return = Return(val=Var(c))
if_body = [assign, return]

//create else body
assign_else = Assignment(var=Var(c), rhs=Const(3))
else_body = [assign_else]

//construct if statement
if = If(condition=cond, body=if_body, else=else_body)
```

For now, though, we don’t need to worry about conditionals, variable assignments, or binary operators. Right now, the only AST nodes we need to support are programs, function declarations, statements, and expressions. Here’s how we’ll define each of them:

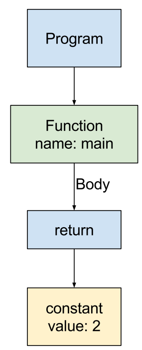
```
program = Program(function_declaration)
function_declaration = Function(string, statement) //string is the function
statement = Return(exp)
exp = Constant(int)
```

Right now, a program consists of a single function, main. Later on we’ll define a program as a list of functions. A function has a name and a body. Later, a function will also have a list of arguments. In a real C compiler, we’d also need to store the function’s return type, but right now we only have integer types. A function body is a single statement; later it will be a list of statements. There’s only one type of statement: a return statement. Later we’ll add other types of statements, like conditionals and variable declarations. A return statement has one child, an expression - this is the value being returned. For now an expression can only be an integer constant. Later we’ll let expressions include arithmetic operations, which will allow us to parse statements like return 2+2;.

As we add new language constructs, we'll update the definitions of our AST nodes. For example, we'll eventually add a new type of statement: variable assignment. When we do, we'll add a new form to our statement definition:

```
statement = Return(exp) | Assign(variable, exp)
```

Here's a diagram of the AST for return_2.c:



- Program
    - Function (name: main)
        - body
            - return statement
                - constant (value: 2)

Finally, we need a formal grammar, which defines how series of tokens can be combined to form language constructs. We'll define it here in Backus-Naur Form:

```
<program> ::= <function>
<function> ::= "int" <id> "(" ")" "{" <statement> "}"
<statement> ::= "return" <exp> ";"
<exp> ::= <int>
```

Each of the lines above is a *production rule*, defining how a language construct can be built from other language constructs and tokens. Every symbol that appears on the left-hand side of a production rule (i.e. <program>, <function>, <statement>) is a non-terminal symbol. Individual tokens (keywords, ids, punctuation, etc.) are terminal symbols. Note that, while this grammar tells us what sequence of tokens constitutes a valid C program, it *doesn't* tell us exactly how to transform that program into an AST - for example, there's no production rule corresponding to the Constant node in the AST. We could rewrite our grammar to have a production rule for constants, but we don't have to in order to parse the program.

Right now the grammar is extremely simple; there's only one production rule for each non-terminal symbol. Later, some non-terminal symbols will have multiple production rules. For example, if we added support for variable declarations, we could have the following rule for deriving statements:

```
<statement> ::= "return" <int> ";" | "int" <id> "=" <int> ";"
```

To transform a list of tokens into an AST, we'll use a technique called recursive descent parsing. We'll define a function to parse each non-terminal symbol in the grammar and return a corresponding AST node. The function to parse symbol *S* should remove tokens from the start of the list until it reaches a valid derivation of *S*. If, before it's done parsing, it hits a token that isn't in the production rule for *S*, it should fail. If the rule for *S* contains other non-terminals, it should call other functions to parse them.

Here's the pseudocode for parsing a statement:

```
def parse_statement(tokens):
    tok = tokens.next()
    if tok.type != "RETURN_KEYWORD":
        fail()
    tok = tokens.next()
    if tok.type != "INT"
        fail()
    exp = parse_exp(tokens) //parse_exp will pop off more tokens
    statement = Return(exp)

    tok = tokens.next()
    if tok.type != "SEMICOLON":
        fail()
```

    return statement

Later, the production rules will be recursive (e.g. an arithmetic expression can contain other expressions), which means the parsing functions will be recursive too - hence the name recursive descent parser.

☑ *Task:*

Write a *parse* function that accepts a list of tokens and returns an AST, rooted at a Program node. The function should build the correct AST for all valid stage 1 examples, and raise an error on all invalid stage 1 examples. If you want, you can also have your parser fail gracefully if it encounters integers above your system's INT_MAX.

There are a lot of ways to represent an AST in code - each type of node could be its own class or its own datatype, depending on what language you're writing your compiler in. For example, here's how you might define AST nodes as OCaml datatypes:

```
type exp = Const(int)
type statement = Return(exp)
type fun_decl = Fun(string, statement)
type prog = Prog(fun_decl)
```

Code Generation

Now that we've built an AST, we're ready to generate some assembly! Like we saw before, we only need to emit four lines of assembly. To emit it, we'll traverse the AST in roughly the order that the program executes. That means we'll visit, in order:

- The function name (not really a *node*, but the first thing in the function definition)
- The return value
- The return statement

Note that we often (though not always) traverse the tree in post-order, visiting a child before its parent. For example, we need to generate the return value before it's referenced in a return statement. In later posts, we'll need to generate the operands of arithmetic expressions before generating the code that operates on them.

Here's the assembly we need:

1. To generate a function (e.g. function "foo"):
   ```
    .globl _foo
   _foo:
    <FUNCTION BODY GOES HERE>
   ```
2. To generate a return statement (e.g. return 3;):
   ```
    movl    $3, %eax
    ret
   ```

☑ *Task:*

Write a *generate* function that accepts an AST and generates assembly. It can return the assembly as a string or write it directly to a file. It should generate correct assembly for all valid stage 1 examples.

(Optional) Pretty printing

You'll probably want a utility function to print out your AST, to help with debugging. You can write it now, or wait until you need it. Here's what nqcc's pretty printer outputs for return_2.c:

```
FUN INT main:
    params: ()
    body:
        RETURN Int<2>
```

This example includes some information your AST doesn't need, like the return type and list of function parameters.

☑ *Task:*

Write a *pretty-print* funcion that takes an AST and prints it out in a readable way.

Putting it all together

☑ *Task:*

Write a program that accepts a C source file and outputs an executable. The program should:

1. Read in the file
2. Lex it
3. Parse it
4. Generate assembly
5. Write the assembly to a file
6. Invoke GCC command to convert the assembly to an executable:
   gcc -m32 assembly.s -o out
   In this command, "assembly.s" is the name of the assembly file and "out" is the name of the executable you want to generate. The -m32 option tells GCC to build a 32-bit binary. You can omit that option and build 64-bit binaries if you want, but you'll need to make some changes to the code generation steps later on (e.g. using 64-bit registers).

7. (Optional) Delete the assembly file.

Testing

You can test that your compiler is working properly with the test script here. It will compile a set of test programs using your compiler, execute them, and make sure they return the right value.

To invoke it:

./test_compiler.sh /path/to/your/compiler
In order to test it with the script, your compiler needs to follow this spec:

1. It can be invoked from the command line, taking only a C source file as an argument, e.g.: ./YOUR_COMPILER /path/to/program.c
2. When passed program.c, it generates executable program in the same directory.
3. It doesn't generate assembly or an executable if parsing fails (this is what the test script checks for invalid test programs).

The script doesn't check whether your compiler outputs sensible error messages, but you can use the invalid test programs to test that manually.

Up Next

In the next post, we'll add three unary operators: -, ~, and !. Stay tuned!

If you have any questions, corrections, or other feedback, you can email me or open an issue.

Further Reading

- Baby Steps to a C Compiler - a post about another C compiler inspired by Ghuloum's paper.
- The C11 Standard, the current C language specification. Annex A is a summary of C's grammar, so it's a good reference for parsing. You probably don't need to read this all the way through.

1 If you're not familiar with sum types or pattern matching, there's a good introduction here.↩

2 An assembler converts a bunch of human-readable assembly instructions (like inc) into binary opcodes (like 1000000). A linker combines multiple object files (the files produced by the assembler) into a single executable. Even though return_2.c doesn't reference any external libraries, we still need the linker, for two reasons:

- Object files produced by the assembler aren't in the right file format.
- The linker includes some initialization code, called crt0, even though it's not explicitly referenced. ↩

3 In case you're curious about these GCC options: -S tells GCC to generate an assembly file (return_2.s) instead of an executable. -O3 turns on a bunch of compiler optimizations - this removes a lot of boilerplate and makes the code easier to read - at least for extremely simple programs like this one. -fno-asynchronous-unwind-tables tells it not to generate an unwind table, which contains information needed to generate stack traces. Hiding the unwind

table also makes the code smaller and more readable.↩

[4] I think these directives will vary between platforms; these were generated using Homebrew gcc 7.2.0 on OS X. Here's what they mean:

- .section __TEXT,__text_startup,regular,pure_instructions tells the assembler that this is the text section, which contains assembly instructions (other sections might contain string literals, initialized data, debug information, etc.).
- .align 4 tells the assembler to align all the instructions at 16-byte intervals - the 4 here is a power of 2, 2^4 = 16. On some architectures .align 4 would mean align at 4-byte intervals. This directive is important because instructions (and data) can be fetched more quickly from word-aligned addresses on most CPU architectures. On a 64-bit machine, a word is only 4 bytes, but some SIMD x86 instructions require data to be 16-byte aligned; I think that's why GCC emits .align 4.
- .subsections_via_symbols is used to eliminate dead code. It indicates that each chunk of assembly beginning with a symbol can be treated as an individual block, and removed if it's not used by any other block. More info in the documentation here.↩

[5] A register is a very tiny, very fast memory cell that sits right on the CPU and has a name you can refer to in assembly. ↩