UNION COLLEGE

CSC-375: Compiler Design

Course: *CSC-375 Compiler Design* – Professor: *Aaron Cass*
Due date: *DUE DATE*

Objectives

- Learn and implement the different stages of a compiler for a subset of the C programming language (C99 standard)
- Construct a functioning compiler capable of compiling the given C files
- Work in groups of 3-4

Compilers! In the beginning with CSC-120 and further into your CS career here at Union, you may have noticed that in languages like C and Java, unlike Python, your code had to *compile* before you could run it. Compilation is the process of taking your code file as text, processing it, and turning it into instructions that are directly usable by your machine's CPU.

You might have wondered, "why does this need to happen if my C code is well formatted?" and to answer, we have to step back and think about what a programming language actually does for a human, and what language a computer uses. A computer's CPU only accepts a series of bits which it takes in electrically and uses to compute results, and to us humans that means a series of binary (or hexadecimal for short) numbers. Many of us wouldn't be here if programming was sitting for hours writing and attempting to read hexadecimal, so programmers designed assembly code to make these numbers more readable and easily composed into usable binary. Unfortunately, we quickly discovered that writing in assembly is only slightly less horrible, so we then designed programming languages to make programming much more human readable, and these languages produce assembly for us. Think about this, though. The computer's CPU has no clue what an "int" is, or a "for", or a "{" even, all it wants is numbers. This means that we have to take steps to convert this text that we as humans can read into executable code, because otherwise our CPU, which does all the work of computing for us, couldn't care less what a printf is.

In this course you'll get a chance to implement every step of a compiler including a lexer, a parser, a symbol table stack, a semantic analyzer, and finally a code generator which will allow you to compile from a C file to something called LLVM IR, a language that is just one readable step above assembly and can be converted to assembly and run on many different CPUs.

Lexer

This is the first stage of a compiler. A lexer's job is fairly simple, the human readable program text is given to the lexer and the lexer takes this giant long string of code and reads through it character by character, generating a list of markers that mark what it saw, these markers are called "lexical tokens". For example, if a lexer encountered the string "a = 1 + 2" the resulting list of lexical tokens would look something like "[variable:a, equals, number:1, plus, number:2]". This might seem trivial, but it makes the next stage much, much easier.

Parser

A parser is the second stage of a compiler. A parser's job is to take the list of lexical tokens you got from the last step, and look over them to build a tree of logic called an abstract syntax tree or AST. We've given you a completed representation of an AST and AST nodes that you will use for this. This AST will define the order of operations to do all of the operations you wrote your code to do in the correct order, essentially representing your original program code as a tree of steps to follow.

Semantic Analyzer

The semantic analyzer is the third step in code compilation, and it works with the AST generated by parsing. The semantic analyzer essentially does two functions at once. For one thing, it acts as a sort of a sanity check to make sure the logic of a program is acceptable. The parser doesn't really have an understanding of this when it's creating the AST, it knows it's creating logic structure but it doesn't know if that structure makes sense or not, and the semantic analyzer checks if it make sense. For example, on one line the parser may parse "a « 2" into a tree just fine, but it doesn't know that this isn't valid because a was defined earlier as a float, not an int. The semantic analyzer will catch this error. The second thing a semantic analyzer does is it creates a symbol table stack for the AST, speaking of...

Symbol Table Stack

The symbol table stack is a key component of a compiler, as it is essentially a way to keep track of variables and functions and their scopes, building a representation of scopes that can be used for later reference. A "symbol table" is a structure that keeps track of variable names, types, and other information for a given scope, a stack of them lets you keep track of many scopes!

IR Generator

This is the last stop for compilation here, generating LLVM Intermediate Representation code (IR). At this point what really usually happens is the AST and STS are used together to produce directly machine runnable assembly, but this isn't very versatile so instead you'll be writing LLVM IR which is essentially its own language one step above machine runnable assembly that can be compiled and run on its own for many different CPUs. Generating this means you can run it!

Follow these guidelines to build each part effectively

1. Do not change any of the function signatures, i.e. leave all parameters, parameter types, and return types as you found them.
2. Follow all instructions given in the comments where they exist, and take any potential hints into consideration when making your design.
3. Start early. These are not the kind of projects you can begin and finish last minute. They take a lot of work and may present challenges that you did not anticipate. You're given ample time to finish, but it cannot be stressed enough that you will likely not finish if you start working close to the deadline.
4. Never be afraid to ask for help. You're in groups for a reason, and you will fair much better asking questions and working with your group than struggling in silence. These projects are difficult, and teamwork will be the key to success.
5. Make sure all tests work before moving on to the next section. Each level of this compiler depends on the previous one being correctly functional.