Introduction, Scanning

Lecture #01

Charles Averill

Practical Compiler Design
The University of Texas at Dallas

Spring 2023

Introduction

- Welcome to Practical Compiler Design!
- Info about me -
  - 3rd-year undergraduate student at UTD
  - Planning for a PhD in Computer Science - likely PL field
  - Have been studying compiler design since Fall 2021 - check out Purple, WIP compiler for a language I'm designing. Also prettybird, a compiler for a language I submitted to PLDI 2023 (hopefully waiting on acceptance notification!)

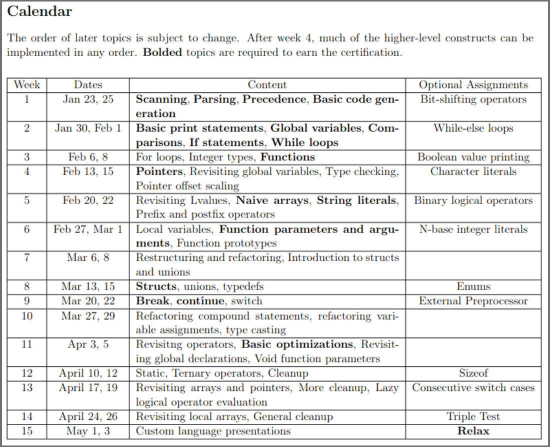
Course Information

- This course is for people passionate about programming and interested in learning about language design - if you don't like programming you will not enjoy this course
- This course is not for credit, I am partnering with the CS department to offer a new certification - "University of Texas at Dallas Certification in Compiler Design"
- You don't have to know much to participate in this class, the most important thing is a mature programming ability. Familiarity with assembly is very helpful but not necessary. Students should be familiar with trees and tree descent. Submissions will be done through GitHub or Gitlab. These topics are covered in the review video I posted

Passing this Course

- This course is built around the completion of a hand-written compiler, with many concessions given:
  - You may write the compiler in any language you choose. Please choose something mainstream so I can grade it. I will not grade your compiler if it is written in an esoteric language. Ask me if you're unsure your desired language is allowed
  - You may compile any imperative language you choose. As long as it meets base requirements and your compiler uses the methods we discuss in class - or sufficiently complicated alternate methods. Turing machine languages like BF are too simple and not allowed
  - There are only two submission deadlines - the midterm deadline and the final deadline. How fast you complete the course is up to you
- We cover many topics. A total of 75% of the topics are required to implement into your compiler to pass. 50% are required topics, and the remaining 25% must be chosen from the remaining topics.

Schedule



Calendar

The order of later topics is subject to change. After week 4, much of the higher-level constructs can be implemented in any order. **Bolded** topics are required to earn the certification.

Week	Dates	Content	Optional Assignments
1	Jan 23, 25	**Scanning**, **Parsing**, **Precedence**, **Basic code generation**	Bit-shifting operators
2	Jan 30, Feb 1	**Basic print statements**, **Global variables**, **Comparisons**, **If statements**, **While loops**	While-else loops
3	Feb 6, 8	For loops, Integer types, **Functions**	Boolean value printing
4	Feb 13, 15	**Pointers**, Revisiting global variables, Type checking, Pointer offset scaling	Character literals
5	Feb 20, 22	Revisiting Lvalues, **Naive arrays**, **String literals**, Prefix and postfix operators	Binary logical operators
6	Feb 27, Mar 1	Local variables, **Function parameters and arguments**, Function prototypes	N-base integer literals
7	Mar 6, 8	Restructuring and refactoring, Introduction to structs and unions	
8	Mar 13, 15	**Structs**, unions, typedefs	Enums
9	Mar 20, 22	**Break**, **continue**, switch	External Preprocessor
10	Mar 27, 29	Refactoring compound statements, refactoring variable assignments, type casting	
11	Apr 3, 5	Revisitng operators, **Basic optimizations**, Revisiting global declarations, Void function parameters	
12	April 10, 12	Static, Ternary operators, Cleanup	Sizeof
13	April 17, 19	Revisiting arrays and pointers, More cleanup, Lazy logical operator evaluation	Consecutive switch cases
14	April 24, 26	Revisiting local arrays, General cleanup	Triple Test
15	May 1, 3	Custom language presentations	**Relax**

This info and more available via the syllabus

Course Goals

- Design a compiler for either a subset of C or an imperative language of your choice
- Implement extra features as optional homework
- Learn how common high-level structures such as loops, conditional statements, functions, and more can be parsed into Abstract Syntax Trees and mapped to generated LLVM-IR pseudo-assembly code.
- Learn how to apply simple optimizations to parsed high-level code and abstract syntax trees in order to improve generated pseudo-assembly code
- Compare the (relatively) naive approaches presented in this course to production-level approaches found in compilers like `gcc` and `clang`

Starting out

- Spend today/tomorrow thinking about what language you want to compile and what language you want to write the compiler in. I'll be doing a C compiler written in Python (C is easy to compile, Python is easy to write)
- Get the project base - https://github.com/CharlesAverill/ecco
- If you're running Linux/Mac (I suggest you do) you can just run this with python after installing poetry, a package development system. If using Windows, the project base has a Dockerfile, so use Docker. I have tested Docker on Linux and it works fine, if it doesn't for you then contact me. WSL should work fine
- "Why do we have a project base? I thought we were writing everything from scratch?" - I wanted to remove the complications of non-compiler-specific related features, like the Python import system, argument parsing, error handling, etc.

Compiler Architecture

- Generally, a compiler is a program that receives a program in the "source language" as input, then translates that program to a new program in the "target language", reporting errors in the source program as it encounters them.
- Most compilers explicitly follow the "Analysis-Synthesis Model of Compilation" (ours will muddy the distinctions a little bit, but still follow the general idea):
    1. Lexical Analysis - scan source program into Tokens, throw error if encountered a non-token sequence of characters
    2. Semantic Analysis - ensure tokenized source program follows syntax rules of source language, throw error if syntax is not followed
    3. Abstract Syntax Tree (AST) generation - parse tokens into a tree that can be optimized through recursive descent, and manipulated in other ways
    4. Code generation - walk the AST and generate code in the target language as you do so

Scanning

- Another word for Lexical Analysis
- We will start out by parsing simple arithmetic expressions with no operator precedence.
- Let's walk through how our compiler runs, open `ecco/ecco.py` (https://github.com/CharlesAverill/ecco/blob/project_base/ecco/ecco.py)
- When we call `ecco` (or `./scripts run`) in the terminal, `main` is the function that gets called
- First, we check command line arguments. In `ecco/utils/arguments.py` we can see that `PROGRAM` (our source file's filename) is a required argument. These args get parsed and returned as an object in our main function

Scanner class

- Back in our main function, we see that we initialize a `Scanner` object with our source file's filename as an argument. If we look in `ecco/scanning/ecco_scanner.py` we see that we store the filename, and also a `TextIO` object from the standard library
- In `__enter__` and `__exit__` (the functions that allow us to use the `with Scanner(...) as ...:` syntax) we handle opening and closing the source program file. The Scanner class holds onto this file descriptor so we can do stuff with it. We could scan the entire program into a string, but that's resource-intensive for big source programs
- Back in main, we can see that we call the Scanner's `scan_file` method. Let's take a look at that method.

Scanner class

- `scan_file` will step through every Token in the file (using `scan`) and print them out one by one
- `scan` will:
  1. Ensure we're not at the end of the file
  2. Compare the first character of each TokenType to the character we're currently reading, store any TokenTypes that match
  3. If we didn't find any matches, we can assume that we encountered a number (unsigned integer, no decimal points or unary negative operators) rather than an operator. We parse and store this integer into an `INTEGER_LITERAL` Token
  4. Otherwise, we stored an operator token (or a multiple-character token, but we won't worry about that now)