Compiler Design
Fall 2024
Course Information

Prof. Kaplan
Last updated: 2024-Aug-17

Be sure to read **all** of this document!

1 Overview

1.1 What's a compiler?

Consider the code that you have written in any of the high-level languages used in our CS curriculum, such as *Java* and *C*. It contains variables, types, arbitrary arithmetic expressions, method/procedures and procedure calls, and sometimes much more (e.g., objects, modules/packages). There are many tools for giving the code a logical and visual structure. Just as importantly, there are tools that catch certain types of errors: uninitialized variables, type mismatches, call argument and return value mismatches, incomplete structures and expressions.

Now consider what you've learned in the systems courses about what *assembly language* and *machine code* look like. There are no variables; the programmer must keep track of where values are stored at any given moment (e.g., in a register, on the stack). There are no types; perform an integer operation on floating point value, or try to dereference a character as though it were a string, and nothing in the language or the assembler will catch that mistake. There are no arithmetic expressions; omplex operations on data must be broken down into a carefully managed sequence of operations. Procedures and calls are not provided by assembly language; rather, the programmer must create them by explicitly writing the code to manage the stack, perform argument passing, and handle returns and return values.

It is therefore no surprise that we do not write code in assembly unless we must. Yet, for higher-level languages to exist—to make it possible to write programs with variables and types and procedures—we must be able to translate automatically those higher-level programs into low-level assembly, since only assembly can be transformed into the machine code that processors can execute.

As you have likely known since you started programming, a *compiler* performs this translation. To design a compiler, one must confront a series of problems and potential solutions. The design of programming languages themselves are intimately tied to what a compiler can (and cannot) do, thus defining how the language will be used by programmers. In this course, we will move through the problems that must be solved to write a compiler that translates a simple, C-like programming language into assembly code that we can run.

1.2 The concepts and components

The components of a compiler can be broken down into a well-established series of tasks:

- **Lexical analysis:** Turn the text of the source program into a seuquence of *tokens*, where each token is a legal operator, value, or name (i.e., variable or procedure name).
- **Parsing:** Turn the sequence of tokens into an *internal representation* that captures the structure and operations of the program, verifying that the tokens correctly follow the *grammar* of the language.
- **Semantic and type analysis:** Have all of the variables used been declared? Are operations and procedure calls performed on the correct types? Do procedures always return? Are variables initialized before they are used? The internal representation is traversed to prove that these properties hold throughout the program.
- **Code generation:** Emit a sequence of assembly operations that execute the program represented by the internal representation.

Even this group of tasks is simplified: compilers often first generate an *intermediate representation*, emitting a sequence of operations for an abstract (virtual) machine. That abstracted reprensentation can then be subjected to a number of *optimizations* (such as register allocation), which is then translated into an *architecture specific representation* (e.g., *x86*, *RISC-V*, *ARM*). Even that final representation can be further optimized with *architecture specific optimizations* that yield the final assembly code that is assembled into machine code.

And threaded throughout all of these stages of compilation is the problem of *error identification and reporting*. Knowing when an error has been identified is difficult: false negatives (failing to detect an error) and false positives (incorrectly flagging correct code as flawed) are both unacceptable. Providing meaningful information to the user can be exceedingly difficult and add complexity to each of the stages.

1.3 The projects

Given all of this detail, we observe two things about how our course will proceed:

1. **Real compilers are complex.** Grammars are a large topic, and there are many forms of parsing with varying degrees of efficiency and complexity (both to write/debug and to execute). Type theory is an entire field of study, and object orientation (which is a branch of type theory) adds complexity. Internal representations yield *control flow graphs* that are used to find optimizations on *basic blocks*. Making the compiler capable of generating output for multiple processor architectures requires the abstraction of an intermediate presentation and then independent optimization and generation units for each target.

   Which is all to say, *we cannot possibly do it all*. In fact, we are unlikely even to be able even to talk about all of it. The goal will be to discuss enough to understand the design of a compiler, and the many places at which problems are abstracted, simplified, and solved.

2. **We can only make a simple compiler.** The only way to understand these concepts is to write a compiler. With that said, the compiler we write will be greatly simplified. The language we compile will have a reduced set of operations and use a simple grammar; the will only allow a few types; we will forego an intermediate representation and move directly to real code generation with a simplified use of our target processor architecture.

The projects will begin somewhat more structured and we get familiar with the territory, but they will become more like group-project adventures as we get deeper into the semester. The goal will be a working compiler (or two) for the class, discovering for ourselves the challenges of getting these ideas to work in practice.

2 Logistics

Lectures: Our class meetings are on **Monday, Wednesday, and Friday** of each week, from **10:00 pm to 10:50 pm** in **SCCE E108**.

You are expected to be present for **all of the class meetings**. Some of our meetings will be primarily lectures, where I will lead us through the topics outlined above. Some class meetings—increasingly so near the end of the semseter—will be focused on the group development of our compiler. In any case, this course should be highly interactive. There will be no replacement for the presentations and the discussions that we have in class.

Office hours: If you seek assistance, reinforcement, review, or other opportunities to discuss the course material or assignments, you should see me. There is a link on the course web page for office hour slots. If none of the available times work, contact me to find another time to meet.

Announcements and questions: For communication outside of class meetings, we will use `Slack`. You should download and install it. There will be a class channel, `#cosc-371-24f`, where I will post class announcements, updates about assignments, and pointers to useful information. That channel will also be the place for you to post questions that others may wish to see, and to get those questions answered by myself or other students. Finally, via direct messaging, Slack will be the quickest way to send me a question and get an answer.

There is, of course, also a class website: `bit.ly/cosc-371-24f`. It will also have announcements, as well as all the of the assignments and documents needed for the class.

3 Texts and materials

This course will use the textbook *Modern Compiler Implementation in Java* by Andrew Appel, 2^{nd} edition. This textbook will be provided through the college's textbook program.[1] We will not follow the book explicitly, but use it as reinforcement of the ideas presented in class and practiced during labs and projects.

The tools needed for this course should be familiar to you. You should install them on your own computer (if you haven't already):

- A **Java development kit (JDK)**, which provides the compiler (`javac`) and Java virtual machine (`java`).
- A **text editor or IDE**, such as *VSCode*, *Sublime*, *Emacs*—whatever works for you.
- `git`, since we will use *GitLab* to keep repositories for the the projects.
- I recommend an environment that provides standard UNIX tools such as `bash`, `make`, `grep`, `awk`, etc. *macOS* comes standard with some of these; for others, you may need install a set of developer tools. A little internet searching will help with that task if needed. For Windows users, I recommend installing *Windows Subsystem for Linux (WSL)*, which provides a Linux installation that runs alongside Windows itself and provides all the basic tools.

4 Assignments, the final exam, and grading

This course is substantially project-based. It is critical that you not fall behind on the projects. Better to have completed a part of the project imperfectly and move onto the next part than to get stuck. Engaging with all of the parts of the compiler is essential.

Early portions of the projects will be done individually, and with clear, fixed deadlines. Later elements of the compiler will involve group projects. Deadlines for these will be set as the groups present their work and the development progresses.

During the final exam period, each of you will schedule an individual meeting time for an oral exam with me. For this exam, I will ask you about the group project work, asking questions about the code and about related topics discussed in class.

Your grade will be determined in small part by your individual projects, in larger part by your participation in class discussions, and mostly by your oral final exam.

[1] Well, I hope it is! I put in the request for it very, very late. It's unlikely to be ready at the start of the semester, but I hope it will get added shortly after.

5 Academic dishonesty

You will be expected to do **your own work** on all the individual projects in this course. It is acceptable to discuss any assignment for the class with a classmate. You may even discuss your approach to a particular problem, or review relevant material for a problem with another person. However, you **may not show another student your code, nor see another student's code. If in doubt, contact me and ask for clarification**.

For the group projects, the submitted work must be written by the members of the group. Copying and submitting code from another student, a web site, or an AI system (e.g., Chat-GPT) is misrepresenting that code as your own work, and is therefore plagiarism.

Regarding the use of AI's or other internet sources: Don't. The core experience of this course is to internalize the concepts and algorithms involved in compilation by writing an actual compiler. Engage fully with that project and you will understand much more clearly and deeply.