Lecture Notes on Compiler Design: Overview

15-411: Compiler Design
Frank Pfenning*

Lecture 1
August 24, 2010

1 Introduction

This course is a thorough introduction to compiler design, focusing on more low-level and systems aspects rather than high-level questions such as polymorphic type inference or separate compilation. You will be building several complete end-to-end compilers for successively more complex languages, culminating in a mildly optimizing compiler for a (subset of a) safe variant of the C programming language to x86-64 assembly language. For the last project of this course you will have the opportunity to optimize more aggressively, to implement a garbage collector, or retarget the compiler to an abstract machine. The safe variant of the C programming language in which is called C0 and has been developed at Carnegie Mellon University by Frank Pfenning et al. based on the reference compiler from last year's Compiler Design course.

In this overview we review the goals for this class and give a general description of the structure of a compiler. Additional material can be found in the optional textbook [App98, Chapter 1].

*These course notes are by Frank Pfenning from previous versions of the Compiler Design course. This version of the notes are minor edits for the course 15-411 on Compiler Design in Fall'10 taught by André Platzer.

2 Goals

After this course you should know how a compiler works in some depth. In particular, you should understand the structure of a compiler, and how the source and target languages influence various choices in its design. It will give you a new appreciation for programming language features and the implementation challenges they pose, as well as for the actual hardware architecture and the runtime system in which your generated code executes. Understanding the details of typical compilation models will also make you a more discerning programmer.

You will also understand some specific components of compiler technology, such as lexical analysis, grammars and parsing, type-checking, intermediate representations, static analysis, common optimizations, instruction selection, register allocation, code generation, and runtime organization. The knowledge gained should be broad enough that if you are confronted with the task of contributing to the implementation of a real compiler in the field, you should be able to do so confidently and quickly.

For many of you, this will be the first time you have to write, maintain, and evolve a complex piece of software. You will have to program for correctness, while keeping an eye on efficiency, both for the compiler itself and for the code it generates. Because you will have to rewrite the compiler from lab to lab, and also because you will be collaborating with a partner, you will have to pay close attention to issues of modularity and interfaces. Developing these software engineering and system building skills are an important goal of this class, although we will rarely talk about them explicitly.

3 Compiler Requirements

As we will be implementing several compilers, it is important to understand which requirement compilers should satisfy. We discuss in each case to what extent it is relevant to this course.

Correctness. Correctness is absolutely paramount. A buggy compiler is next to useless in practice. Since we cannot formally prove the correctness of your compilers, we use extensive testing. This testing is end-to-end, verifying the correctness of the generated code on sample inputs. We also verify that your compiler rejects programs as expected when the input is not well-formed (lexically, syntactically, or with respect to the static semantics),

and that the generated code raises an exception as expected if the language specification prescribes this. We go so far as to test that your generated code fails to terminate (with a time-out) when the source program should diverge.

Emphasis on correctness means that we very carefully define the semantics of the source language. The semantics of the target language is given by the GNU assembler on the lab machines together with the semantics of the actual machine. Unlike C, we try to make sure that as little as possible about the source language (C0) remains undefined. This is not just for testability, but also good language design practice since an unambiguously defined language is portable. The only part we do not fully define are precise resource constraints regarding the generated code (for example, the amount of memory available).

Efficiency. In a production compiler, efficiency of the generated code and also efficiency of the compiler itself are important considerations. In this course, we set very lax targets for both, emphasizing correctness instead. In one of the later labs in the course, you will have the opportunity to optimize the generated code.

The early emphasis on correctness has consequences for your approach to the design of the implementation. Modularity and simplicity of the code are important for two reasons: first, your code is much more likely to be correct, and, second, you will be able to respond to changes in the source language specification from lab to lab much more easily.

Interoperability. Programs do not run in isolation, but are linked with library code before they are executed, or will be called as a library from other code. This puts some additional requirements on the compiler, which must respect certain interface specifications.

Your generated code will be required to execute correctly in the environment on the lab machines. This means that you will have to respect calling conventions early on (for example, properly save callee-save registers) and data layout conventions later, when your code will be calling library functions. You will have to carefully study the ABI specification **[?]** as it applies to C and our target architecture.

Usability. A compiler interacts with the programmer primarily when there are errors in the program. As such, it should give helpful error messages. Also, compilers may be instructed to generate debug information together

with executable code in order help users debug runtime errors in their program.

In this course, we will not formally evaluate the quality or detail of your error messages, although you should strive to achieve at least a minimum standard so that you can use your own compiler effectively.

Retargetability. At the outset, we think of a compiler of going from one source language to one target language. In practice, compilers may be required to generate more than one target from a given source (for example, x86-64 and ARM code), sometimes at very different levels of abstraction (for example, x86-64 assembly or LLVM intermediate code).

In this course we will deemphasize retargetability, although if you structure your compiler following the general outline presented in the next section, it should not be too difficult to retrofit another code generator. One of the options for the last lab in this course is to retarget your compiler to produce code in a low-level virtual machine (LLVM). Using LLVM tools this means you will be able to produce efficient binaries for a variety of concrete machine architectures.

4 The Structure of a Compiler

Certain general common structures have arisen over decades of development of compilers. Many of these are based on experience and sound engineering principles rather than any formal theory, although some parts, such as parsers, are very well understood from the theoretical side. The overall structure of a typical compiler is shown in Figure 1.

In this course, we will begin by giving you the front and middle ends of a simple compiler for a very small language, and you have to write the back end, that is, perform instruction selection and register allocation. Consequently, Lectures 2 and 3 will be concerned with instruction selection and register allocation, respectively, so that you can write your own.

We then turn to the front end and follow through the phases of a compiler in order to complete the picture, while incrementally complicating the language features you have to compile. Roughly, we will proceed as follows, subject to adjustment throughout the course:

1. A simple expression language
2. Loops and conditionals

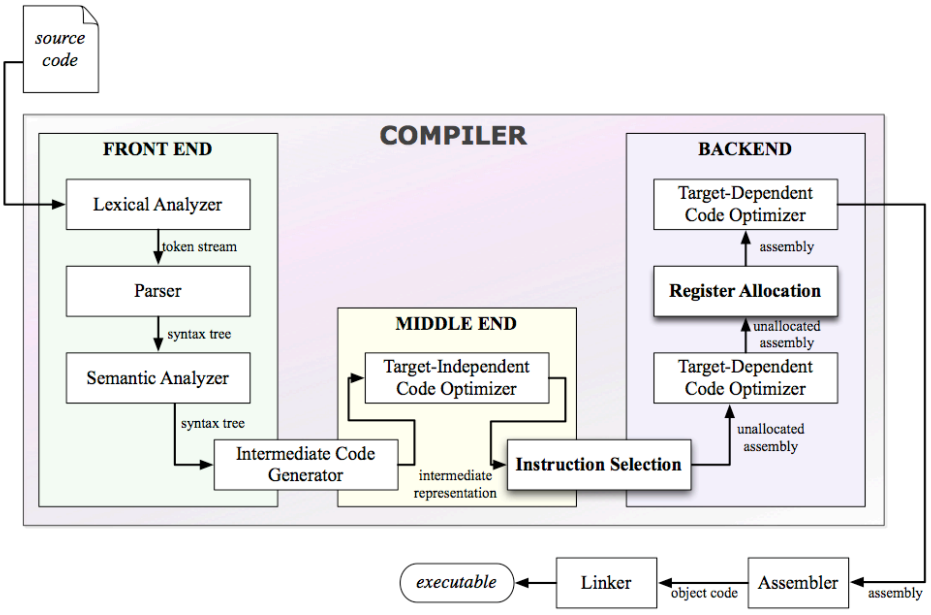

Figure 1: Structure of a typical compiler[1]

3. Functions
4. Structs and arrays
5. Memory safety and basic optimizations

The last lab is somewhat open-ended and allows either to implement further optimizations, a garbage collector, or a new back end which uses the low-level virtual machine (LLVM)[2].

5 Bibliographical Notes

Additional material about compiler design can be found in the optional textbook [App98]. Other standard references on compiler construction include the "dragon book" [ALSU06] and Muchnik's book on optimizing

[1] Thanks to David Koes for this diagram.
[2] See `http://llvm.org`

compilers [Muc97]. A book that also covers functional, logic-based, and object-oriented languages is by Wilhelm and Maurer [WM95].

References

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman, Boston, MA, USA, 2nd edition, 2006.

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.