Course Introduction
Lecture 1

Kevin Leach

January 4, 2018
Course Introduction

Lecture 1

Compilers, what a concept

Kevin Leach

January 4, 2018
Cunning Plan

▶ Administrivia
▶ Compilers overview
▶ Compilation example
▶ Course motivation
▶ Course overview
Course Information

- Instructor: Kevin Leach (kjleach@umich.edu)
  - Please put EECS483 in the subject (or just use Piazza)
- Office Hours: TBA (see Piazza, prefer Thursday)
- Meeting Time: MW 2:00–3:30, EECS 1311
  - Bonus Discussion: F: 12:00–1:00, FXB 1012
- Books (optional)
  - *Programming Language Pragmatics* by Michael Scott
  - *Compilers: Principles, Techniques, and Tools* by Alred Aho et al., a.k.a. the Dragon Book
WHAT I LEARNED IN COMPILERS IS.....
What is a compiler?

The thing Eclipse runs when you hit ‘Play’?
What is a compiler?

...then you add `try/catch` to make the red go away?
What is a compiler?

A tool that turns your C code into segfaults?
What is a compiler?

A tool that turns your C code into segfaults?
What is a compiler?

A tool that turns your C code into segfaults?
What is a compiler?

A program that produces incomprehensible warnings?
What is a compiler?

- A language processing tool
  - Input in one language is **transformed** to an **equivalent** output in another language

```
Structured Input                        Structured Output

Source Code     Compiler     Program

firefox.c       gcc -o firefox.exe firefox.c  firefox.exe
```
Compiler Workflow

- **Goal** Developer can access language features
  - Object-oriented paradigms (classes)
Compiler Workflow

- **Goal** Developer can access language features
  - Object-oriented paradigms (classes)
  - Polymorphism (Cat inherits Animal)
Compiler Workflow

- **Goal** Developer can access language features
  - Object-oriented paradigms (classes)
  - Polymorphism (Cat inherits Animal)
  - Variables (human-readable naming)
Compiler Workflow

- **Goal** Developer can access language features
  - Object-oriented paradigms (classes)
  - Polymorphism (Cat inherits Animal)
  - Variables (human-readable naming)
  - Functional paradigms (functions and immutable state)
Compiler Workflow

- **Goal** Developer can access language features
  - Object-oriented paradigms (classes)
  - Polymorphism (Cat inherits Animal)
  - Variables (human-readable naming)
  - Functional paradigms (functions and immutable state)
  - Data types (int/string/char)
Compiler Workflow

- **Goal** Developer can access language features
  - Object-oriented paradigms (classes)
  - Polymorphism (Cat inherits Animal)
  - Variables (human-readable naming)
  - Functional paradigms (functions and immutable state)
  - Data types (int/string/char)
  - Control flow (if/then/else)
Compiler Workflow

- **Goal** Developer can access language features
  - Object-oriented paradigms (classes)
  - Polymorphism (Cat inherits Animal)
  - Variables (human-readable naming)
  - Functional paradigms (functions and immutable state)
  - Data types (int/string/char)
  - Control flow (if/then/else)

- **Constraint** fast execution on native platform (e.g., x86 CPU)
Compiler Workflow

- **Goal** Developer can access language features
  - Object-oriented paradigms (classes)
  - Polymorphism (Cat inherits Animal)
  - Variables (human-readable naming)
  - Functional paradigms (functions and immutable state)
  - Data types (int/string/char)
  - Control flow (if/then/else)

- **Constraint** fast execution on native platform (e.g., x86 CPU)
  - Need to “remove” features so that a CPU circuit can understand
Compiler Input

- Input is structured
  - Amenable to automated analysis and processing
  - Developer can write a *specification* for program behavior

- This class: Use COOL programming language

```coolk
1 (* hello-world.cl *)
2 class Main inherits IO {
3     main() : Object {
4         out_string("Hello, world.\n");
5     }
6 }
```

2,3 All
The compiler gives a promise to developer: If you supply a valid program according to the language’s rules, the compiler provides a valid assembly output.
The compiler gives a promise to developer:
If you supply a valid program according to the language’s rules, the compiler provides a valid assembly output

Implication: Developer promises to provide valid program
The compiler gives a promise to developer: If you supply a valid program according to the language’s rules, the compiler provides a valid assembly output.

Implication: Developer promises to provide valid program.

If they don’t, compiler gets to complain!
Compiler Output

- Output is also structured
  - Windows PE, Linux ELF, etc...
  - Usually a *lowered format*
- This class: Produce assembly (either RISC or x86)
Compiler performs *semantics preserving* transformations
Issues Guiding Compiler Design

Correctness

- Compiler performs \texttt{semantics-preserving} transformations
Issues Guiding Compiler Design

Correctness
  - Compiler performs \textit{semantics-preserving} transformations

Speed
  - Run time
  - Compilation time
Issues Guiding Compiler Design

Correctness
- Compiler performs semantics-preserving transformations

Speed
- Run time
- Compilation time

Space
- How large is the output?
Issues Guiding Compiler Design

- **Correctness**
  - Compiler performs *semantics-preserving* transformations

- **Speed**
  - Run time
  - Compilation time

- **Space**
  - How large is the output?

- **Feedback to user**
  - On a scale of LaTex to Python, how good is your feedback?
Issues Guiding Compiler Design

- **Correctness**
  - Compiler performs *semantics-preserving* transformations

- **Speed**
  - Run time
  - Compilation time

- **Space**
  - How large is the output?

- **Feedback to user**
  - On a scale of LaTeX to Python, how good is your feedback?

- **Debugging**
  - Can your output help the developer identify problems with their desired program?
Compilers vs. Interpreters

- Compilers lower code to assembly
  - A compiler produces a program that runs later (e.g., after compilation)

Interpreters directly execute the code provided
- Common in scripting languages
- Just-in-time compilation?
  - Secretly, languages like Python, Lua, and Java are all compiled at runtime!

Why would you compile vs. interpret?
Compilers vs. Interpreters

- **Compilers lower code to assembly**
  - A compiler produces a program that runs later (e.g., after compilation)

- **Interpreters directly execute the code provided**
  - Common in scripting languages
Compilers vs. Interpreters

- **Compilers** lower code to assembly
  - A compiler produces a program that runs later (e.g., after compilation)
- **Interpreters** directly execute the code provided
  - Common in scripting languages

- Just-in-time compilation?
  - Secretly, languages like Python, Lua, and Java are all compiled at runtime!
Compilers vs. Interpreters

- **Compilers** lower code to assembly
  - A compiler produces a program that runs later (e.g., after compilation)
- **Interpreters** directly execute the code provided
  - Common in scripting languages

- Just-in-time compilation?
  - Secretly, languages like Python, Lua, and Java are all compiled at runtime!

- Why would you compile vs. interpret?
Stages of Compilation

- Compilers are well-established

Input Code (Cool Language)

Tokenization (Lexical Analysis)

Parser (Syntactic Analysis)

Semantic Analysis

Code Generation

Optimization

Output Code (assembly)
Front End vs. Back End

- Lexing, Parsing, and Type-checking are the *front end*
- Code generation and optimization are the *back end*
Front End vs. Back End

- Lexing, Parsing, and Type-checking are the *front end*
- Code generation and optimization are the *back end*
- Modularity
Compiling by Analogy

- We’re going to make an English compiler.
- It will turn English into Spanish.

Example Input
I enjoy compilers.

Example Output
Yo disfruto compiladores.
Compiling by Analogy

- We’re going to make an English compiler.
- It will turn English into Spanish.

- **Example Input**
  I enjoy compilers.

- **Example Output**
  Yo disfruto compiladores.

- Compiling is a **5 step** process
- Each step takes an input and creates an *intermediate representation*
1: Tokenization or Lexical Analysis

- Break input into tokens and lexemes that are part of the language’s lexicon
  - keywords (if, while)
  - operators (+, -)
  - variable names
  - constants ("hello, world", 1, 0x40)

- English tokens = words

<table>
<thead>
<tr>
<th>Valid Inputs</th>
<th>Invalid Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>I am handsome.</td>
<td>我很帅。</td>
</tr>
<tr>
<td>He is pony.</td>
<td>Èl es poni.</td>
</tr>
</tbody>
</table>

Output is a list of lexemes and tokens.
“I am handsome.” → “I”, “am”, “handsome”, “.”
1: Tokenization or Lexical Analysis

- Break input into tokens and lexemes that are part of the language’s lexicon
  
  - keywords (if, while)
  - operators (+, -)
  - variable names
  - constants (“hello, world”, 1, 0x40)

- English tokens = words

**Valid Inputs**

I am handsome.
He is pony.

Output is a list of lexemes and tokens.

“I am handsome.” → “I”, “am”, “handsome”, “.”
2: Parsing or Syntactic Analysis

- Ensure tokens follow the rules of a grammar
  - + operator should have expressions on both sides...
    - \[ 2 + 3; \checkmark \quad 2 + ; \times \]

- English parsing = is it grammatically correct?

<table>
<thead>
<tr>
<th>Valid Input</th>
<th>Invalid Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>I am handsome.</td>
<td>I handsome am:</td>
</tr>
<tr>
<td>I be short.</td>
<td>Is he correct!!</td>
</tr>
</tbody>
</table>

Output is a parse tree or abstract syntax tree (on board).
3: Semantic Analysis (type checking)

- Ensure input makes sense
  - Does it make sense to add strings to integers?

- English = Does it make sense?

<table>
<thead>
<tr>
<th>Valid Input</th>
<th>Invalid Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>I am handsome.</td>
<td>I am short.</td>
</tr>
<tr>
<td>College is expensive.</td>
<td>I am rich.</td>
</tr>
</tbody>
</table>

Output is an annotated abstract syntax tree (AST, on board)
4: Code Generation

- Create code that exhibits the required behavior
  - `let x <- 2 + 3` becomes `add t0 <- 2, 3`

- English/Spanish example: Start generating Spanish

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>I am handsome.</td>
<td>Yo soy guapo.</td>
</tr>
</tbody>
</table>

Output is straight-line assembly code
5: Optimization

- Analyze generated code, change it to execute faster
  - `let x <- 2 + 3` becomes `li t0 <- 5`

- English/Spanish example: Remove superfluous words

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yo soy guapo.</td>
<td>Soy guapo.</td>
</tr>
</tbody>
</table>

Output is optimized straight-line assembly code
Compilers Today

- Early on: Lexing, Parsing most complex and expensive

- Today: optimization dominates everything many times over, both in execution time and community interest
  - Scientific programs
  - Advanced processors (DSP, speculative execution)
  - Small devices (speed = more time idle = longer battery)
Compilers Today

- Early on: Lexing, Parsing most complex and expensive

- Today: optimization dominates everything many times over, both in execution time and community interest
  - Scientific programs
  - Advanced processors (DSP, speculative execution)
  - Small devices (speed = more time idle = longer battery)

- We will not focus on lexing/parsing optimizations

- Support for languages with new features
  - MIMD architectures
Who cares?

Many upper-level electives correspond to a natural job choice

- Databases → Oracle
- Embedded Computing → Samsung
- E-commerce → Amazon
- Graphics → Nvidia
Who cares?

Many upper-level electives correspond to a natural job choice

- Databases → Oracle
- Embedded Computing → Samsung
- E-commerce → Amazon
- Graphics → Nvidia

What about compilers?

Reasonable Initial Skepticism
Visual Studio!

Visual Studio Team Services Jobs

Become a member of the Visual Studio Team Services Team

✓ Work with the latest technologies
✓ Work on a fast-paced agile team
✓ Have a big impact on the software industry through building an innovative service

Join us!

Become a key member of the Visual Studio Team Services (VSTS) service team led by Brian Harry and build the next generation of development services in the cloud! VSTS provides software development teams with version control, build automation, agile work management, social experiences and more to nearly 3,000,000 users.
Oracle

Java!

Create the Future
Java is the world's #1 programming language

Overview Roles Technologies Get Started

Java for Developers Java for Consumers

Java Embedded for IoT (2:51)
Leadership Application Performance

- Boost C++ application performance
- Future-proof code by making code that scales
- Plugs right into your development environment

If you are here, you are looking for ways to make your application run faster. Boost performance by augmenting your development process with the Intel® C++ Compiler. The Intel C++ Compiler plugs right into popular development environments like Visual Studio®, Eclipse®, XCode®, and Android Studio®. The Intel C++ Compiler is compatible with popular compilers including Visual C++® (Windows®) and GCC (Linux®, OS X® and Android®).

The Intel C++ Compiler is available in four products based on your application development needs:

- Intel® C++ Compiler in Intel® Parallel Studio XE
- Intel® C++ Compiler in Intel® System Studio
- Intel® C++ Compilers in Intel® INDE (support only)
- Intel® Bi-Endian Compiler
Google

Go, Dart, etc.

Scala, productive development

The Go Programming Language

Try Go

// You can edit this code!
// Click here and start typing.
package main
import "fmt"
func main() {
    fmt.Println("Hello, World!")
}

Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.

Download Go

Binary distributions available for Linux, Mac OS X, Windows, and more.
Wind River, Green Hills

▶ Embedded!

DIAB COMPILER

For over 25 years, Wind River Diab Compiler has been helping industrial, medical, and aerospace industries. Diab Compiler footprint, and produce high-quality, standards-compliant code.

Big Performance Tiny Footprint

Diab Compiler's unique optimization technology generates extremely fast, high-quality object code in the smallest possible footprint.

The Latest!

Due to collaboration with the latest compiler is a microcontroller processor, allowing the development of embedded applications.

Jobs - Opportunities in the USA

Green Hills Software is always looking for qualified engineering, sales, and marketing talent. Please submit your resume to the Corporate Office where it will be processed and reviewed by the hiring manager.

Click on a job title below to complete the description of the position:

- Corporate Field Applications Engineer (Santa Barbara, CA)
- Embedded Software Consultant (Santa Barbara, CA)
- Embedded Solutions Test Engineer (Santa Barbara, CA)
- Field Engineer (Santa Barbara, CA)
- Field Services Engineer (Santa Barbara, CA)
- Functional Safety Software Engineer (Santa Barbara, CA)
- Product Engineer (Santa Barbara, CA)
- Sales Manager Solution T&D
- Software Development Engineer (Santa Barbara, CA)
- Technical Marketing Engineer (Santa Barbara, CA)

Click here for information on applying.

Green Hills Software is an Equal Opportunity / Affirmative Action Employer.

Getting Ready Engineer (Santa Barbara, CA)

Job Description

A software engineer has complete engineering responsibility for one or more major components of the Green Hills product line. For an experienced engineer, this is a satisfying position in which you have personal responsibility for creating a tool used by thousands of programmers around the world. Our engineers are involved in language front ends, Core Generation, Real-time Operating Systems, our MULTI Development Environment, our Secure Workstation, and Target Systems.

Here are the responsibilities for the positions available:

- Compiler: Design, update, and maintain a language front end or a target architecture backend for the highly-optimized family of Green Hills compilers. A compiler engineer might work on new language extensions, specific cutting-edge optimizations for the latest chips to hit the market, or on general optimizations that will remain useful in future products. An ideal candidate understands low-level microarchitecture designs and is comfortable working with assembly code, yet can also code tools written in high-level languages.

Mars rover, cell phones, satellites, engine controllers, jets, cameras, game consoles...
Photoshop contains interpreters...

2 Photoshop Scripting Basics

This chapter provides an overview of scripting for Photoshop, describes scripting support for the scripting languages AppleScript, VBScript, and JavaScript, how to execute scripts, and covers the Photoshop object model. It provides a simple example of how to write your first Photoshop script.

If you are familiar with scripting or programming languages, you most likely will want to skip much of this chapter. Use the following list to locate information that is most relevant to you.

- For more information on the Photoshop object model, see "Photoshop Object Model" on page 11.
- For information on selecting a scripting language, refer to the Introduction to Scripting guide.
- For examples of scripts created specifically for use with Photoshop, see Chapter 3, "Scripting Photoshop" on page 21.
- For detailed information on Photoshop objects and commands, please use the reference information in the three reference manuals provided with this installation: Adobe Photoshop CC 2015 AppleScript Scripting Reference, Adobe Photoshop CC 2015 Visual Basic Scripting Reference, and Adobe Photoshop CC 2015 JavaScript Scripting Reference.

**NOTE:** You can also view information about the Photoshop objects and commands through the object browsers for each of the three scripting languages. See "Viewing Photoshop Objects, Commands, and Methods" on page 21.

**Scripting Overview**

A script is a series of commands that tells Photoshop to perform a set of specified actions, such as applying different filters to selections in an open document. These actions can be simple and affect only a single object, or they can be complex and affect many objects in a Photoshop document. The actions can call Photoshop alone or invoke other applications.
The list goes on

- Mozilla’s SpiderMonkey JS Engine
- Unreal Engine Blueprints Scripting
- Cadence HSPICE Circuit simulation language
What About This Class?

- You will turn in pieces of a compiler
What About This Class?

- You will turn in pieces of a compiler
- Compilers are large software systems
  - This is a challenging project that will require significant planning and developing multiple integrated components
What About This Class?

- You will turn in pieces of a compiler
- Compilers are large software systems
  - This is a challenging project that will require significant planning and developing multiple integrated components
- Former students contend it is very worthwhile
Course Expectations

► **Short version:** Students will deliver a compiler
► There are 6 assignments to help you progress towards this end
  1. PA1 Toposort implementation
  2. PA2 Lexical Analyzer
  3. PA3 Syntactic Analyzer
  4. PA4 Semantic Analyzer
  5. PA5 Compiler
  6. PA6 Optimizing compiler (extra credit)

N.B. Assignments are time-consuming. Start Early
Course Expectations

▶ **Short version**: Students will deliver a compiler
▶ There are 6 assignments to help you progress towards this end
  1. PA1 Toposort implementation
  2. PA2 Lexical Analyzer
  3. PA3 Syntactic Analyzer
  4. PA4 Semantic Analyzer
  5. PA5 Compiler
  6. PA6 Optimizing compiler (extra credit)
▶ N.B. Assignments are time-consuming. **Start Early**
Late policy

- Assignments turned in will be penalized by 1% per hour
- 48 hours late = 48% off that assignment
Exams

- Two midterms, one final exam
- 10% each, 15% for final
- You can have one page of notes, front and back
- Closed test otherwise
- Let me know as early as possible about conflicts
Academic Honesty

- Please don’t cheat
- We often use plagiarism detection software
Academic Honesty

- Please don’t cheat
- We often use plagiarism detection software
- Ample resources available!
Course Resources

- This course is based on Stanford’s Compilers course
Homework 1

- Write a short (50-100) line program in two languages: Cool and OCaml
- First implementation in your favorite language (PA1c) due 1/10!
- Both implementations in your language + Cool (PA1) due 1/17!