Teaching Compiler Design

1

Compiler Course for Undergraduates

- A complete study of compilers could easily fill several graduate-level courses.
- Simplifications and compromises are necessary for a one-semester course that is accessible to undergraduate students.
  - Narrow focus
    - a project-oriented course (the "fun" part of studying compilers)
  - Relatively simple source language
    - but powerful enough to be interesting and challenging
  - Target language is assembly language for a virtual machine with stack-based architecture
    - simplifies code generation
    - eliminates having to deal with general-purpose registers

2

Decisions, Decisions, Decisions

- Target Audience?
  - undergraduate students?
  - graduate students?
  - compiler professionals?

  This course:
  - advanced undergraduate students
  - also useful as basic introduction to graduate students
- Prerequisite?

  This course:
  - programming experience in both high-level and low-level languages (e.g., Kotlin and assembly language)
  - basic knowledge of algorithms and data structures (recursion, lists, stacks, maps, etc.)

3

Decisions, Decisions, Decisions (continued)

- Amount of Theory?
  - comprehensive versus minimal?

  This course:
  - minimal (but must be able to understand and analyze context-free grammars)
- Tools?
  - scanner generators?
  - parser generators?
  - examples: Antlr, Coco/R, Flex/Bison, Lex/Yacc, JavaCC

  This course:
  - no tools other than compiler and IDE (e.g., Kotlin and IntelliJ IDEA)

4

Decisions, Decisions, Decisions (continued)

- Approach to parsing (checking program for valid syntax)?
  - top down versus bottom up (or both)?
  - hard-coded versus table driven (or both)?
  - number of lookahead tokens?

  This course:
  - hard-coded
  - recursive descent (top down) with one token lookahead
- Intermediate Representation(s)?
  - high-level versus low level (or both)?

  This course:
  - abstract syntax trees (high-level, target machine independent)

5

Decisions, Decisions, Decisions (continued)

- Source Language?
  - real programming language (e.g., C++, Java, Python, etc.)?
  - subset of real programming language?
  - simple language designed for teaching basics of compiler design

  This course:
  - simple language designed for teaching basics of compiler design
  - CPRL (**C**ompiler **PR**oject **L**anguage)
- Implementation Language (language used to write the compiler)?

  This course:
  - flexible, but Kotlin is recommended. Slides, handouts, and skeletal code all use Kotlin.

6

Decisions, Decisions, Decisions (continued)

- Target Language?
  - real machine versus virtual machine (e.g., JVM)
  - machine code versus assembly language

  This course:
  - assembly language (simplifies code generation)
  - virtual machine (similar to JVM but simpler)

7

A Quote on LL(1) Recursive Descent Parsers

"This pattern shows how to implement parsing decisions that use a single token of lookahead. It's the weakest form of recursive-descent parser, but the easiest to understand and implement. If you can conveniently implement your language with this LL(1) pattern you should do so." – Terence Parr

8

Course Project

Implementation of a compiler for a small programming language

- Simple source language (CPRL)
- Simple target language (assembly language for CVM, a simple stack-based virtual machine)
- Build a compiler **one step at a time**.
  - series of 8 smaller subprojects
- Lots of template Kotlin code to guide you through the process

9

Challenging Project Variations (for ambitious undergraduates or graduate students)

- Add one or more new features to the language
  - enum types
  - records/structures (or classes)
  - references/pointers and dynamic memory allocation (heap)
  - predefined environment with builtin procedures/functions (make Boolean a predefined enum type)
- Modify target language/machine
  - real machine, or assembly language for a real machine (e.g., Intel x86)
  - JVM or assembly language for JVM
  - Common Language Runtime (part of Microsoft's .NET Framework)
  - C programming language (e.g., first C++ "compilers")

10

Challenging Project Variations (continued)

- Implement the project in a language other than Kotlin (e.g., Java, C++, Python, or C#)
- Implement constraint analysis and code generation using the visitor design pattern
- Redesign code generation to allow for multiple targets
  - use a universal, machine-independent back end (e.g., LLVM)
  - use design patterns to create a code-generation factory

11

Compiler Implementation Resources

- Kotlin source code for
  - CVM (virtual machine target for compiler
  - Assembler for CVM
  - Disassembler for CVM machine code
- Kotlin source code or skeletal source code for various parts of the compiler
- Language documentation files for the full CPRL compiler
- Correct and incorrect CPRL programs for testing your compiler
- Sample Windows command files and Bash shell scripts for running and testing various stages of your compiler

12