Introduction

COMP 520: Compiler Design (4 credits)

Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 10:30-11:30, TR 1100

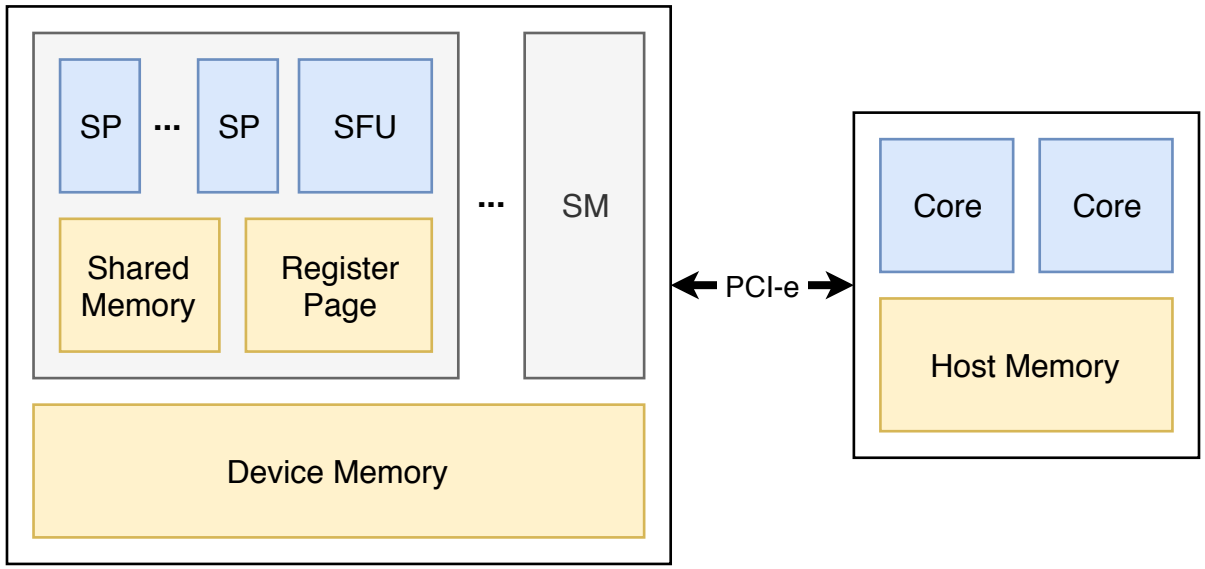
http://www.cs.mcgill.ca/~cs520/2020/

I'm Alex!

- Current PhD student in compilers;
- Undergrad & masters at McGill in computer science;
- Part of McGill's compiler research lab since 2014;
  - Speculative multithreading
  - Dependence analysis
  - GPU compilers
- Originally took COMP 520 in Winter 2015;
- Taught COMP 520 since 2017; and
- *Not* a morning person; but *definitely* a winter person.

GPU Compiler Research

A very different target architecture



How do we write efficient parallel programs for the GPU?

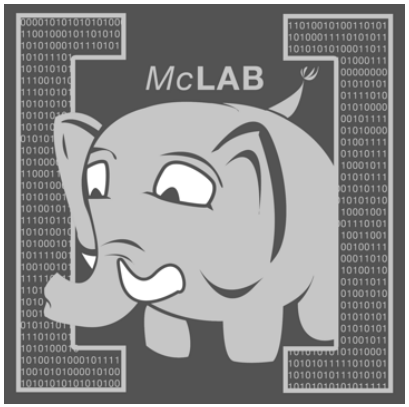
Furthermore, how do we translate single-threaded CPU programs to GPU programs?

My research: JIT compilation of SQL queries for GPUs

Compiler Research at McGill (McLAB)

Current Projects

- Dynamic GPU compilation and optimization (r3d3)
- Array language optimization for SQL queries (HorseIR)
- Sparse matrix operations (SpMV fait maison)
- JavaScript and WebAssembly



Past Work

- MATLAB analysis frameworks (McSAF, Tamer)
- MATLAB static compilers (Mc2For, MiX10, Matjuice)
- JIT compilation (McVM, Velociraptor)
- Programing languages and tools (AspectMATLAB)

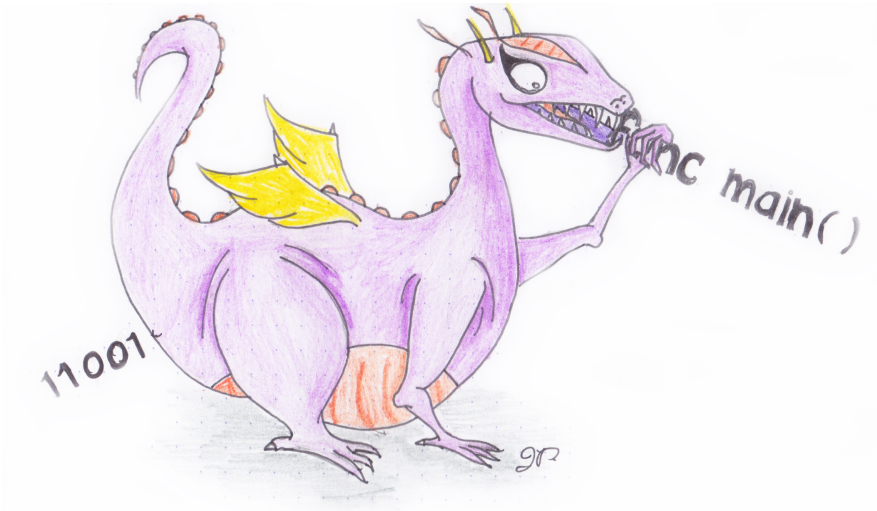
In Memory of Professor Laurie Hendren

Introduction

Course Outline

Compilers Overview

Bootstrapping



What is this class?

Purpose

This course is an introduction to the full pipeline of modern compilers

- It covers all aspects of the compiler pipeline for modern languages (C, Python, etc.);
- Explores its application to both general-purpose and domain-specific languages; and
- Discusses virtual machines and runtime environments.

End Goal

By the end of this class you will have a working knowledge of compilers that allows you to

- Effectively use compiler tools for general-purpose and domain-specific projects; and
- Produce fully functional compilers for general-purpose languages.

Contents

Topics covered in class

- **Deterministic parsing:** Scanners, LL/LR parsers, `flex`/`bison` tools
- **Semantic analysis:** Abstract syntax trees, symbol tables, type checking
- **Virtual machines and run-time environments:** JVM (stack machines), virtual register machines, garbage collection
- **Code generation:** Resources, templates, optimizations
- **Special topics:** GPUs, Rust/modern languages

Class Information

Schedule

- Lectures: 3 hours/week (MWF 10:30-11:30)

Prerequisites

- COMP 273, COMP 302, (COMP 330), ability to read and write “large” programs
- Students without COMP 330 should read the background material available at:
  `http://www.cl.cam.ac.uk/teaching/2003/RLFA/notes.pdf`

Lecturer

- Alexander Krolik, McConnell 234
  Office Hours: Monday 5:00-6:00, Wednesday 11:30-12:30

TAs

- Adrian Koretski (`adrian.koretski@mail.mcgill.ca`) McConnell 235
  Office Hours: Tuesday/Thursday 14:45-15:45
- Jason Pizzuco (`jason.pizzuco@mail.mcgill.ca`) McConnell 234
  Office Hours: Friday 12:00-13:00

If you have class at these times, send one of us an email to arrange another meeting time.

Marking Scheme (Outdated, see next slide)

Assignments

- 10% for individual assignments (2 x 5%)
- 10% for the JOOS peephole optimizer (group)

GoLite Project

- 35% for content submitted at milestones (group)
- 10% for the final compiler and report (group)
- Group members may be given different grades if the contributions are not reasonably equal

Midterm: 10% midterm (before reading week)

Final exam: 25% final exam (during exam period)

- *A 25% supplemental exam is offered*

Assignments and project milestones are due at midnight of the due date. A penalty of 10% per day late is given. Assignments will not be accepted after solutions have been circulated.

Marking Scheme (New)

Assignments

- 10% for individual assignments (2 x 5%)
- 15% for the JOOS peephole optimizer (group)

GoLite Project

- 30% for content submitted at milestones (group)
- 25% for the final compiler and report (group)
- Group members may be given different grades if the contributions are not reasonably equal

Midterm: 20% midterm (before reading week)

Final exam: Cancelled

Assignments and project milestones are due at midnight of the due date. A penalty of 10% per day late is given. Assignments will not be accepted after solutions have been circulated.

Academic Integrity

- McGill University values academic integrity. Therefore all students must understand the meaning and consequences of cheating, plagiarism and other academic offences under the Code of Student Conduct and Disciplinary Procedures.

  `https://www.mcgill.ca/students/srr/honest/`

- In terms of this course, your responsibility is to ensure that all work is correctly attributed. Code without attribution must be entirely your own work.
- If you use some third-party code you must have permission to use it and you must clearly indicate the source.
- You **may not** use previous years' solutions or grading material.
- An honesty statement must be included with all assignment/project submissions.

Other Required McGill Statements

- In accord with McGill University's Charter of Students' Rights, students in this course have the right to submit in English or in French any written work that is to be graded.

Course Material

Textbook and online readings

- Recommended Textbook, *Crafting a Compiler* by Fischer, Cytron and LeBlanc, available in hardcopy at the McGill Bookstore or `www.amazon.ca`. You can find a support web site with additional tutorial and reference information at `http://www.cs.wustl.edu/~cytron/cacweb/`. There should be a copy on reserve in the library, and you may find other sources for the book.
- Alternate Textbook, *Modern compiler implementation in Java (2nd edition)* by Appel and Palsberg. Free online version available via McGill Library (access via a McGill IP or VPN), `http://mcgill.worldcat.org/title/modern-compiler-implementation-in-java/oclc/56796736`.
- Optional but recommended background readings; and
- Required for additional exercises.

Course Material

Slides

- Detailed (and hopefully complete) explanation of the course contents; and
- Available via the web site, either just before or after the lecture.

Course website

- `http://www.cs.mcgill.ca/~cs520/2020/`
- Contains links to examples, documentation and readings; and
- Updated frequently with slides, due dates, and assignment specifications.

Facebook group

- `https://www.facebook.com/groups/COMP520W2020/`

Course Material

GitHub organization

- `https://github.com/comp520`
- Contains examples for scanning and parsing, and assignment starter code.

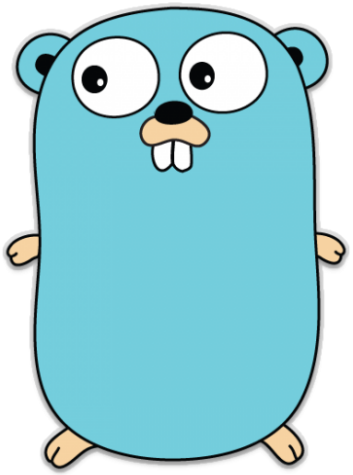
JOOS language and compiler

- Java's Object-Oriented Subset
- Is compiled to Java bytecode;
- Illustrates a fully fledged general purpose language;
- Is used to teach by example and will be useful for your assignments and project; and
- Has source code available on GitHub;

GoLite Project

Go Language

- `https://golang.org`
- Created by Rob Pike, Ken Thompson and Robert Griesemer
- Google employees
- Not a Google project like Gmail; open source
- Initial release in 2009
- 1.0 release in 2012



Motivation

- Simplify development
- Built-in concurrency support
- Faster compilation

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

GoLite Project

Goal

- Design a complete compiler for a significant, non-trivial subset of Go
- Learn a compiler toolkit (C tools shown in class)
- Produce any kind of output code, high-level code [C, JavaScript, Python, ...], or low-level code [assembly, LLVM, Java bytecode, ...]
- Fun stuffed Gophers!

GitHub and Groups

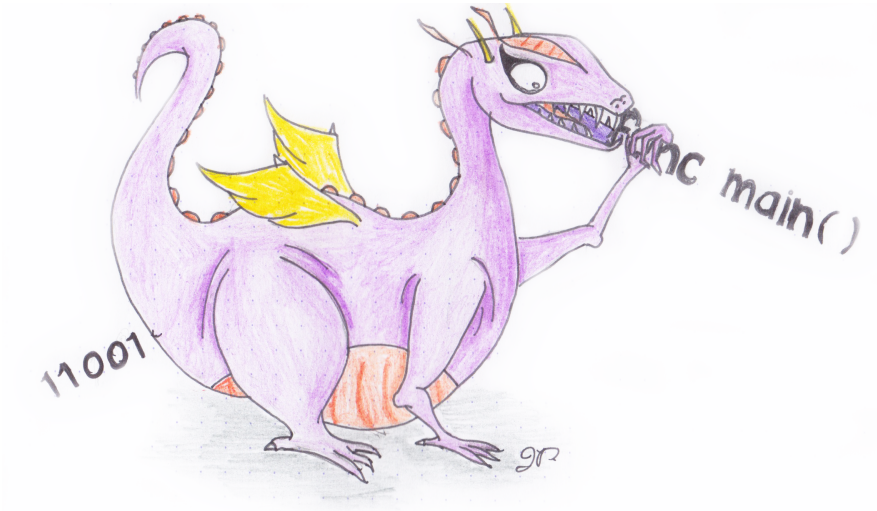
- The project and Peephole Optimizer are done in groups of 2 or 3
  - *Undergraduates are **strongly** recommended to work in groups of 3*
  - Picking a good group is important. If your group dynamics fail, you can either work things out, or work alone for the rest of the project
- We will use GitHub for the GoLite project
- Start looking for your teammates! You will have to declare your group by the add/drop deadline

Introduction

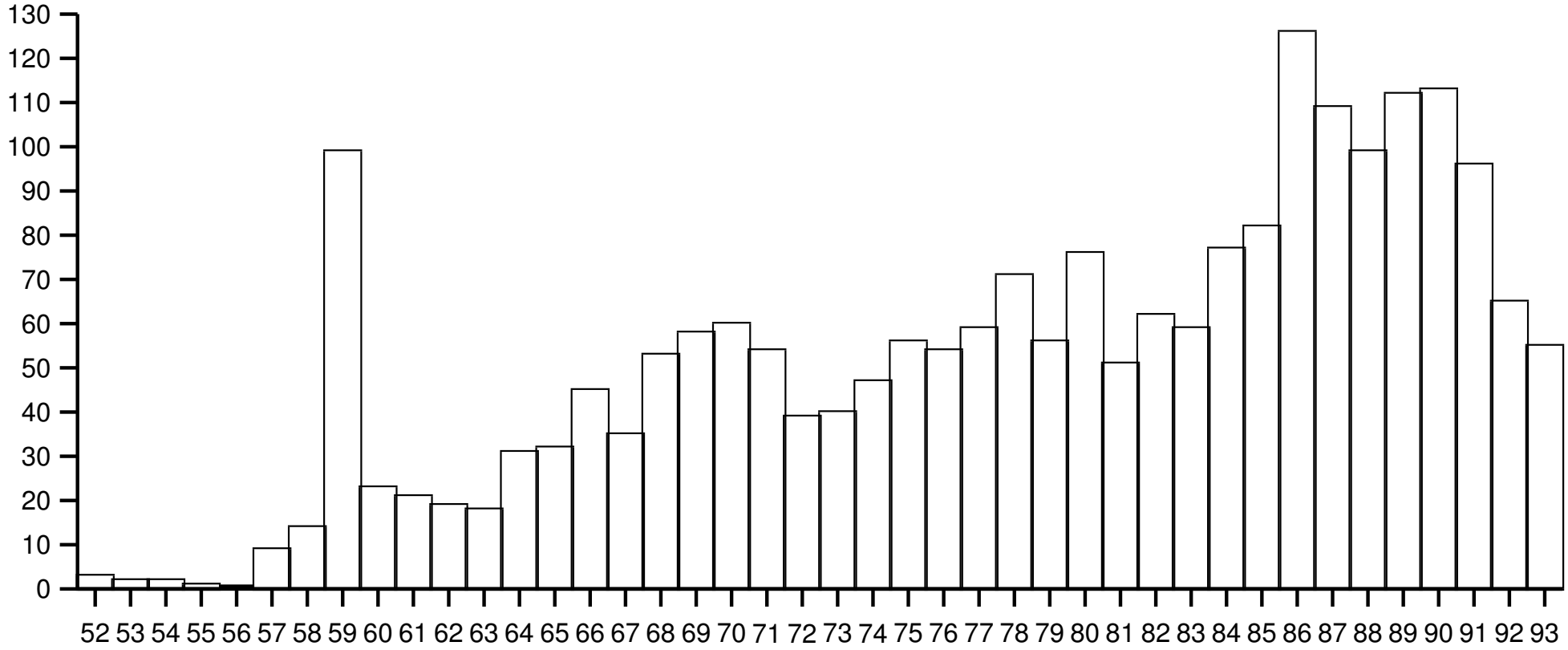
Course Outline

Compilers Overview

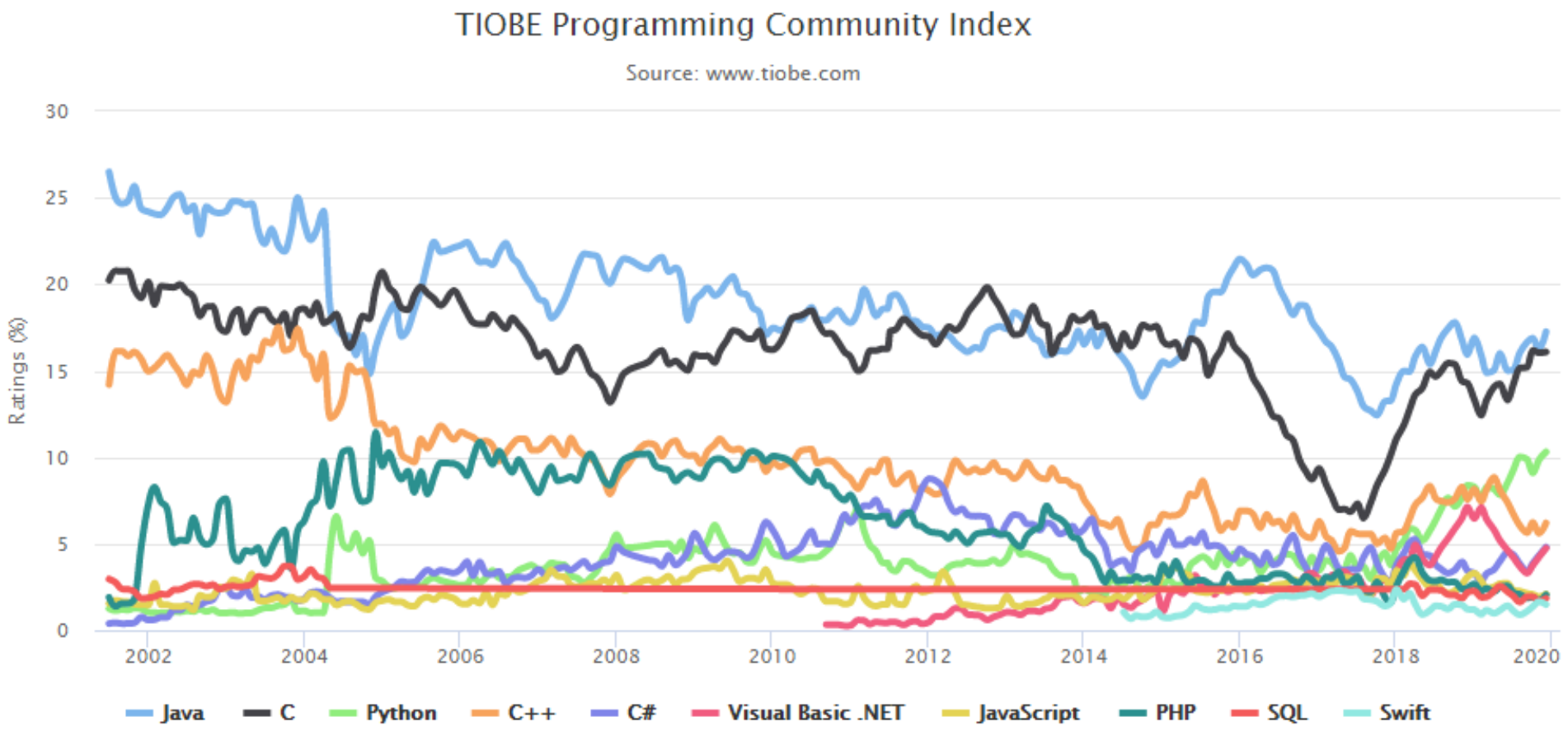
Bootstrapping



New Programming Languages per Year



Popular Programming Languages



http://www.tiobe.com/tiobe-index/

General Purpose Languages

- Allow for arbitrarily useful programs to be written;
- Are Turing-complete in the theoretical sense; and
- Are the focus of most programming language courses.

Prominent examples

- C
- C++
- Java
- JavaScript
- Rust
- . . .

General purpose languages require full-scale compiler technology to run efficiently.

Domain-Specific Languages

- Extend software design;
- Permit representation, optimization, and analysis in ways that low-level languages and libraries do not; and
- They may even be visual! (e.g. boxes & arrows)

Prominent examples

- LaTeX
- `yacc` and `lex`
- Makefiles
- HTML
- SVG
- . . .

Domain-specific languages also require full-scale compiler technology.

What is a compiler?

When talking about compilers you might think of...

- GCC
- clang/LLVM
- javac
- rustc
- . . .

A modern compiler is much more general

- Take source code written in programming language **S**
- Produce equivalent code in target language **T**
- May perform optimizations (performance, space, size) [COMP 621]

The target language **T** can be another high-level language or machine code.

FORTRAN: The First Compiler

Beforehand, programs were

- Written directly in machine code; and
- Developed by low-level programmers.

The first compiler

- Implemented in 1954–1957;
- The world's first complete compiler;
- Motivated by the economics of programming;
- Had to overcome deep skepticism;
- Paid little attention to language design;
- Focused on efficiency of the generated code; and
- Pioneered many concepts and techniques.

Revolutionized computer programming

Example FORTRAN Program

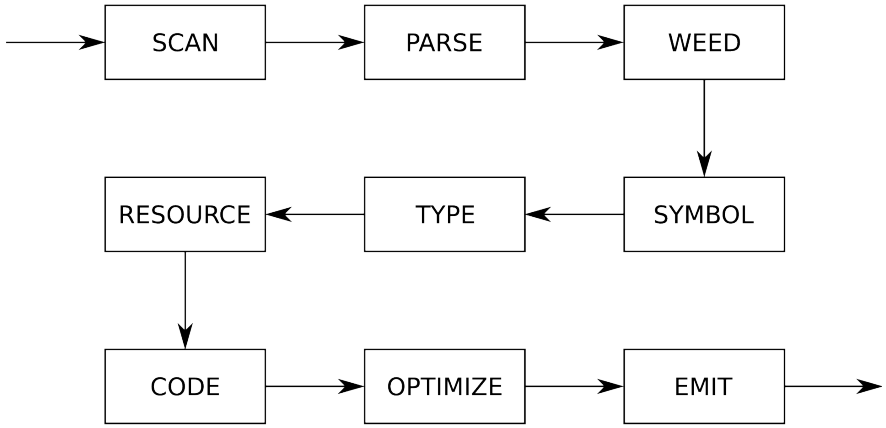
```
C AREA OF A TRIANGLE WITH A STANDARD SQUARE ROOT FUNCTION
C INPUT - CARD READER UNIT 5, INTEGER INPUT
C OUTPUT - LINE PRINTER UNIT 6, REAL OUTPUT
C INPUT ERROR DISPLAY ERROR OUTPUT CODE 1 IN JOB CONTROL LISTING
      READ INPUT TAPE 5, 501, IA, IB, IC
  501 FORMAT (3I5)
C IA, IB, AND IC MAY NOT BE NEGATIVE
C FURTHERMORE, THE SUM OF TWO SIDES OF A TRIANGLE
C IS GREATER THAN THE THIRD SIDE, SO WE CHECK FOR THAT, TOO
      IF (IA) 777, 777, 701
  701 IF (IB) 777, 777, 702
  702 IF (IC) 777, 777, 703
  703 IF (IA+IB-IC) 777,777,704
  704 IF (IA+IC-IB) 777,777,705
  705 IF (IB+IC-IA) 777,777,799
  777 STOP 1
C USING HERON'S FORMULA WE CALCULATE THE
C AREA OF THE TRIANGLE
  799 S = FLOATF (IA + IB + IC) / 2.0
        AREA = SQRT( S * (S - FLOATF(IA)) * (S - FLOATF(IB)) *
     +     (S - FLOATF(IC)))
      WRITE OUTPUT TAPE 6, 601, IA, IB, IC, AREA
  601 FORMAT (4H A= ,I5,5H  B= ,I5,5H  C= ,I5,8H  AREA= , F10.2,
     +        13H SQUARE UNITS)
      STOP
      END
```

Phases of a Modern Compiler

Modern compilers are modular, consisting of multiple *phases* which form the compilation pipeline.

Each individual phase

- Is a modular component;
- Uses input from the previous phase;
- Outputs data for the next phase;
- Transforms between internal representations;
- Has its own standard technology; and
- May be supported by automated tools.



Advanced compilers may contain additional phases, including multiple levels of optimization.

Phases of a Modern Compiler

Illustrated in more detail (textbook "Crafting a Compiler")

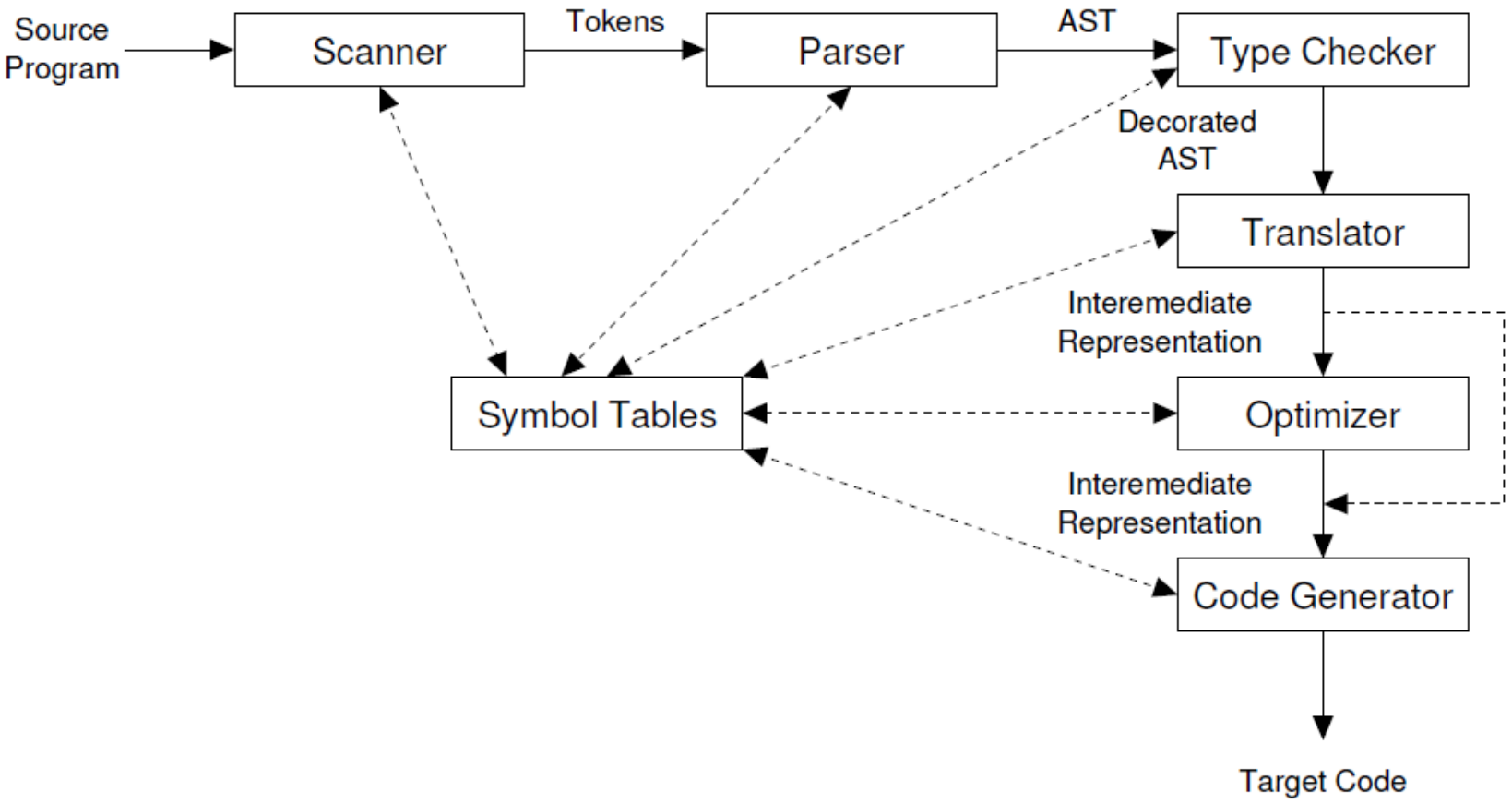


Figure 1.4: A syntax-directed compiler. AST denotes the Abstract Syntax Tree.

Why learn about compilers?

- Understand existing languages and their implementations;
- Appreciate current limitations;
- Talk intelligently about language design;
- Implement your very own general purpose language; and
- Implement lots of useful domain-specific languages.

And in other realms

- Improve programming ability;
- Understand compiler error messages;
- Tackle (some) performance bottlenecks; and
- View programming from a more abstract stance.

The top 10 reasons why we use C for compilers

10. It's tradition
9. It's (truly) portable
8. It's efficient
7. It has many different uses
6. ANSI C will never change
5. You must (almost always) learn C at some point
4. It teaches discipline (the hard way)
3. Methodology is language independent
2. We have `flex` and `bison`
1. You can say that you have implemented a large project in C

The top 10 reasons why we use Java for compilers

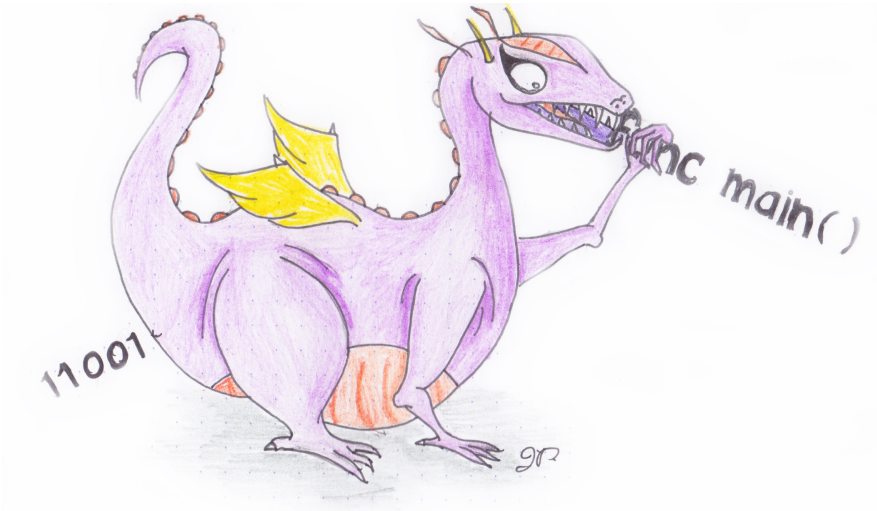
10. You already know Java from previous courses
9. Run-time errors like null-pointer exceptions are easy to locate
8. It is relatively strongly typed, so many errors are caught at compile time
7. You can use the large Java library (hash maps, sets, lists, . . .)
6. Java bytecode is portable and can be executed without recompilation
5. You don't mind slow compilers
4. It allows you to use object-orientation
3. Methodology is language independent
2. We have `SableCC`, developed at McGill
1. You can say that you have implemented a large project in Java

Introduction

Course Outline

Compilers Overview

Bootstrapping



Bootstrapping

How did the first compilers come about, if no compiler existed?

or

How can we design a compiler for a new language **L**, written in **L**?

Bootstrapping

What we want

- A compiler for language **L**, written in language **L**
- A compiler that compiles itself!

Bootstrapping is the process by which we produce a self-compiling compiler for a new language.

Idea: Implement 3 compilers!

(a) Implement (1) a basic compiler for a subset of **L** in an *existing programming language*

(b) Implement (2) a full compiler for **L** using the subset defined in (a)

(c) Compile (b) using (a). We now have a small (but working) compiler for **L**!

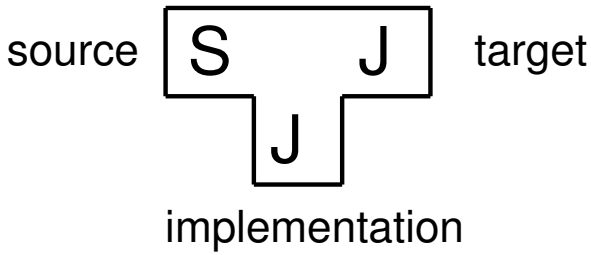
(d) Implement (3) a full compiler for **L** in **L**

How to Bootstrap a Compiler (SCALA)

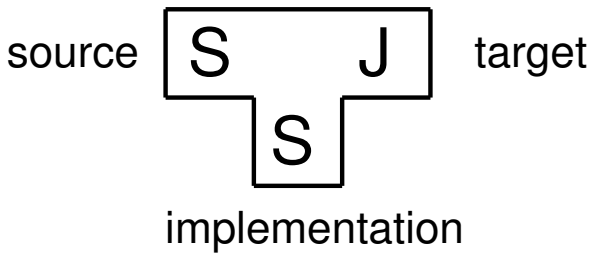
As an example, we will bootstrap a SCALA to execute on the JVM and illustrate with puzzle pieces.

- Source language: SCALA (**S**)
- Target language: Java (**J**)

We need the following to execute on the JVM



We would rather implement SCALA in itself

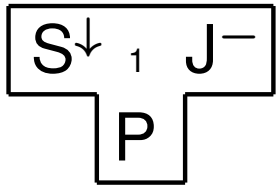


How to Bootstrap a Compiler (SCALA)

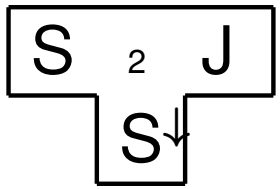
Define the following

- $S^{\downarrow}$ is a simple subset of SCALA
- J^{-} is inefficient Java code
- P is our favourite programming language, here “Pizza”.

We can easily implement a compiler for a subset of SCALA in Pizza

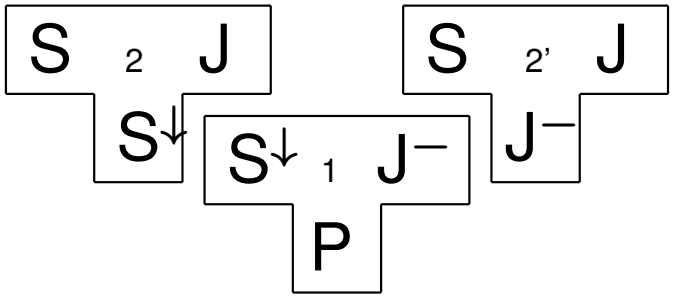


In parallel, using $S^{\downarrow}$, we implement a compiler for full SCALA



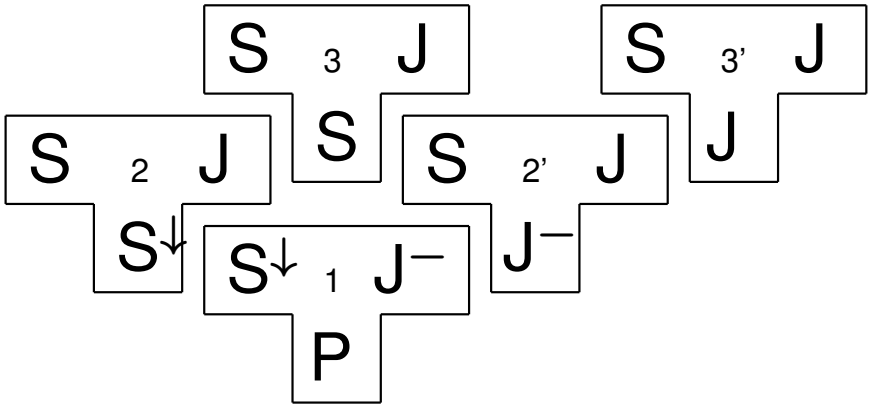
How to Bootstrap a Compiler (SCALA)

Combining the two compilers (compile ② with ①), we get ②'



Which is an inefficient SCALA compiler generating efficient Java code.

A final combination (compiling ③ with ②') gives us what we want, an efficient SCALA compiler, written in SCALA, running on the Java platform ③'.



Bootstrapping

As illustrated in the textbook, "Crafting a Compiler"

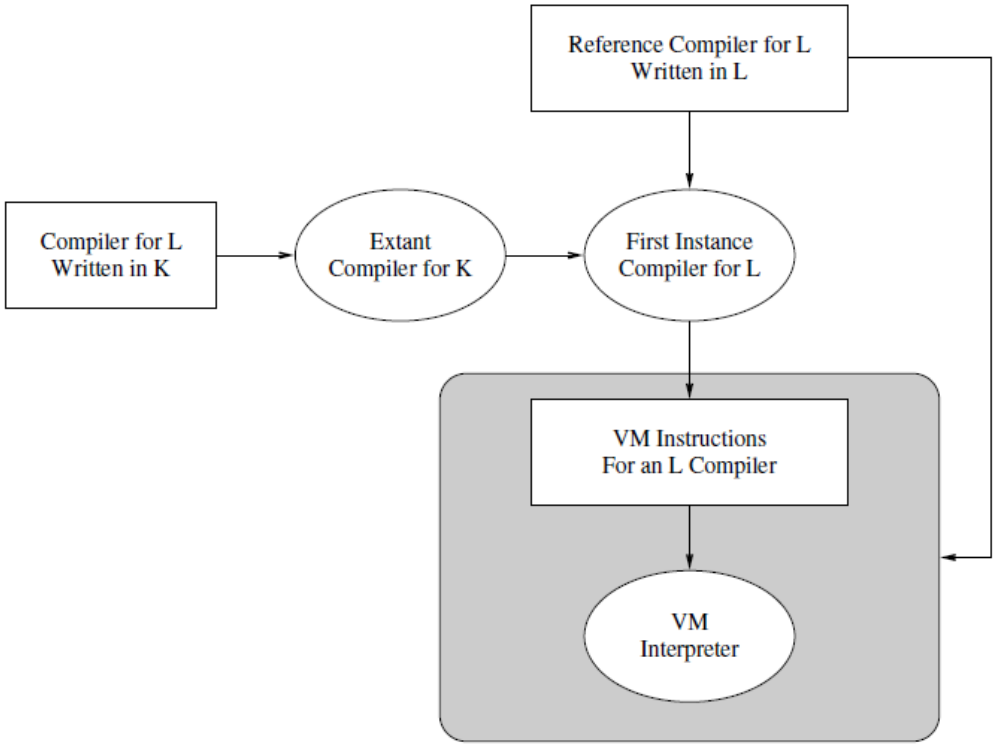


Figure 1.2: Bootstrapping a compiler that generates VM instructions. The shaded portion is a portable compiler for L that can run on any architecture supporting the VM.