Introduction

COMP 520: Compiler Design
Chris Pickett
MWF 11:35-12:35 ENGTR 0060

Purpose:
- This course covers modern compiler techniques and their application to both general purpose and domain specific languages.
- The practical aspects focus on current technologies, primarily Java and interactive web services.

Contents:
- Deterministic parsing: LR parsers, the flex/bison and SableCC tools.
- Semantic analysis: abstract syntax trees, symbol tables, type checking, resource allocation.
- Virtual machines and run-time environments: stacks, heaps, objects.
- Code generation: resources, templates, optimizations.
- Surveys on: garbage collection, native code generation, static analysis.

Schedule:
- Lectures: 3 hours/week.

Prerequisites:
- COMP 273, COMP 302, (COMP 330), ability to read and write “large” programs.
- Students without COMP 330 should read the background material indicated in Week 1 of the web page ASAP.

Lecturer:
- Chris Pickett, McConnell 226, Office Hours MW 12:30–13:30

TA:
- Peng Zhang, McConnell 226, Office Hours TJ 16:00–17:30
Marking Scheme:

- 10% midterm, 25% final exam, 65% assignments and project
- the 65% for assignments and projects will be divided as follows:
  - 5% for each of the first 3 JOOS deliverables (15% total)
  - 10% for the JOOS peephole optimizer
  - 5% for meeting deadlines and milestones
  - 5% for completing weekly exercises
  - 30% for the WIG compiler and report
- Group members may be given different grades on the project work if the contributions are not reasonably equal.

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.
- In terms of this course, part of your responsibility is to ensure that you put the name of the author on all code that is submitted. By putting your name on the code you are indicating that it is completely your own work. If you use some third-party code you must have permission to use it and you must clearly indicate the source of the code.

Course material:

- course pack readings;
- slides for the lectures; and
- extensive documentation on the course web pages.

The course pack:

- is mainly background reading;
- does not discuss the JOOS and WIG projects used in this course; and
- is required for the exercises.

The slides:

- are quite detailed; and
- are available online just before class in 1-up and 4-up formats.

The web pages:

- aim to contain all information;
- provide on-line documentation; and
- may be updated frequently.
New programming languages per year:

The compiler for the FORTRAN language:
- was implemented in 1954–1957;
- was the world’s first compiler;
- was motivated by the economics of programming;
- had to overcome deep skepticism;
- paid little attention to language design;
- focused on efficiency of the generated code;
- pioneered many concepts and techniques; and
- revolutionized computer programming.

General purpose languages:
- allow for arbitrarily useful programs to be written; and
- are the focus of most programming language courses.

Prominent examples are:
- C
- C++
- FORTRAN
- Java
- ...

General purpose languages fairly obviously require full-scale compiler technology. (What about Lisp?)

Domain specific languages:
- extend software design; and
- are concrete artifacts that permit representation, optimization, and analysis in ways that low-level programs and libraries do not.

Prominent examples are:
- \LaTeX
- yacc and lex
- HTML
- XML
- ...

Domain specific languages also require full-scale compiler technology.
Reasons to learn compiler technology:
- understand existing languages;
- appreciate current limitations;
- talk intelligently about language design;
- implement your very own general purpose language; and
- implement lots of useful domain specific languages.

The phases of a modern compiler:

The individual phases:
- are modular software components;
- have their own standard technology; and
- are increasingly being supported by automatic tools.

Advanced backends may contain an additional 5–10 phases.

The project:
- **Java’s Object-Oriented Subset**
- is compiled to Java bytecode;
- illustrates a general purpose language;
- allows client-side programming on the web;
- is used to teach by example;
- has source code available;
- and will be upgraded by you into an A+ version.

The project:
- **Web Interface Generator**
- is compiled to C-based CGI-scripts (or other targets...);
- illustrates a domain specific language;
- allows server-side programming on the web;
- is used to get hands-on experience;
- and will be implemented from scratch by you!
The top 10 list of 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 learn C at some point;
4) it teaches discipline (the hard way);
3) methodology is language independent;
2) we have flex and bison; and
1) you can say that you have implemented a large project in C.

The top 10 list of 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 strongly typed, so many errors are caught at compile time;
7) you can use the large Java libraray (hash tables ...);
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; and
1) you can say that you have implemented a large project in Java.

How to bootstrap a compiler:

- we are given a machine M; and
- a programming language L.

We need the following:

\[
\begin{array}{c}
\text{source} \\
\text{implementation} \\
\end{array}
\begin{array}{c}
L \\
M \\
\end{array}
\begin{array}{c}
\text{target} \\
\end{array}
\begin{array}{c}
M \\
\end{array}
\]

The direct approach is hard and difficult, and we really want to implement L in L itself:

\[
\begin{array}{c}
\text{source} \\
\text{implementation} \\
\end{array}
\begin{array}{c}
L \\
M \\
\end{array}
\begin{array}{c}
\text{target} \\
\end{array}
\begin{array}{c}
L \\
\end{array}
\]

Define the following:

- \(L^\downarrow\) is a simple subset of L; and
- \(M^\downarrow\) is inefficient M-code.

We can easily implement:

\[
\begin{array}{c}
\text{source} \\
\text{implementation} \\
\end{array}
\begin{array}{c}
L^\downarrow \\
M^\downarrow \\
\end{array}
\begin{array}{c}
\text{target} \\
\end{array}
\begin{array}{c}
M \\
\end{array}
\]

and (in parallel) implement:

\[
\begin{array}{c}
\text{source} \\
\text{implementation} \\
\end{array}
\begin{array}{c}
L^\downarrow \\
M \\
\end{array}
\begin{array}{c}
\text{target} \\
\end{array}
\begin{array}{c}
L^\downarrow \\
\end{array}
\]

using basically our favourite language.
Combining the two compilers, we get:

which is an inefficient compiler generating efficient code.

A final combination gives us what we want:

the bootstrapping of an efficient compiler.