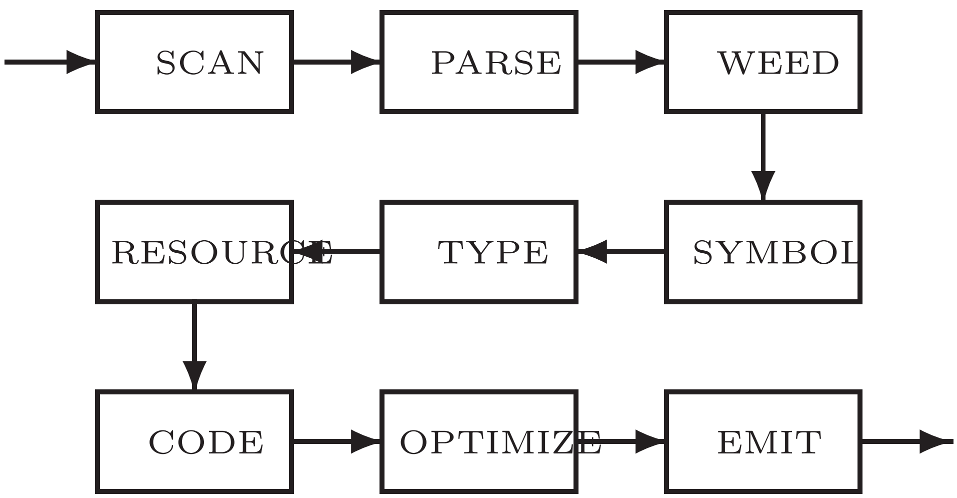
Introduction

COMP 520 Compiler Design

Nomair A. Naeem

MWF 11:35-12:35



Purpose:

- The course will teach modern compiler techniques applied to both general-purpose and domain-specific languages.

  The examples chosen will also convey a detailed knowledge of state-of-the-art based WWW technology such as Java and CGI-based interactive 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 program “large” programs.
- Students without COMP 330 should do background reading as indicated in Week 1 of the web page (ASAP).

Lecturer:

- Nomair A. Naeem, McConnell 234
- Office Hours MW 12:30-13:30

Teaching Assistants:

- To be Announced

Marking Scheme:

- 15% midterm, 25% final exam, 60% assignments and project
- the 60% for assignments and projects will be divided approximately as follows:
  - 5 points for each of the first 3 JOOS deliverables,
  - 10 points for the 4th JOOS deliverable (peephole optimizer),
  - 5 points for meeting milestone deadlines,
  - and 30 points for the WIG compiler/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:

- Readings;
- slides for the lectures; and
- extensive documentation on the course WWW pages.

Course Text

The course text is a special collection of chapters of two very good text books, plus some supplementary material not available in standard text books. The text is required for the course and is available as an **Eastman Course** Pack which can be purchased in the basement of the McGill Book Store (look under the stairs going to the basement for the Eastman Custom Publishing area). The course text contains extracts from:

1. Compiler Construction, Kenneth C. Louden, 582 pages
2. Modern Compiler Implementation in C, Andrew W. Appel, 544 pages
3. SableCC: An Object-Oriented Compiler Framework, M.Sc. Thesis, McGill Unviersity, Etienne Gagnon

plus, the following documents:

1. Flex, version 2.5, by Vern Paxson
2. Bison, The YACC-compatible Parser Generator, November 1995, Version 1.25
3. A Beginner's Guide to HTML, NCSA
4. An instantaneous introduction to CGI scripts and HTML forms, The University of Kansas, Michael Grobe

The order of the readings is:

1. Introduction, Louden (Chapter 1), pages 1-30
2. Lexical Analysis, Appel (Chapter 2) pages 16-38
3. Flex, version 2.5, by Vern Paxson
4. Context-Free Grammars and Parsing, Louden (Chapter 3), pages 95-142
5. Parsing, Appel (Chapter 3.2 - 3.5), pages 46-87
6. Bison, The YACC-compatible Parser Generator
7. SableCC Gagnon (Chapters 3-6)
8. A Beginner's Guide to HTML
9. An instantaneous introduction to CGI scripts and HTML forms
10. Symbol Tables, Louden (Chapter 6.3.1 - 6.3.4), pages 295-308
11. Data Types and Type Checking, Louden (Chapter 6.4), pages 313-334
12. Garbage Collection, Appel (Chapter 13), pages 273-298
13. Liveness Analysis, Appel (Chapter 10), pages 218-234

The book:

- is mainly background reading; and
- does not discuss the projects used in this course

The slides:

- are quite detailed; and
- will be available at the EUS Copy Center in McConnell by the beginning of the second week of lectures.

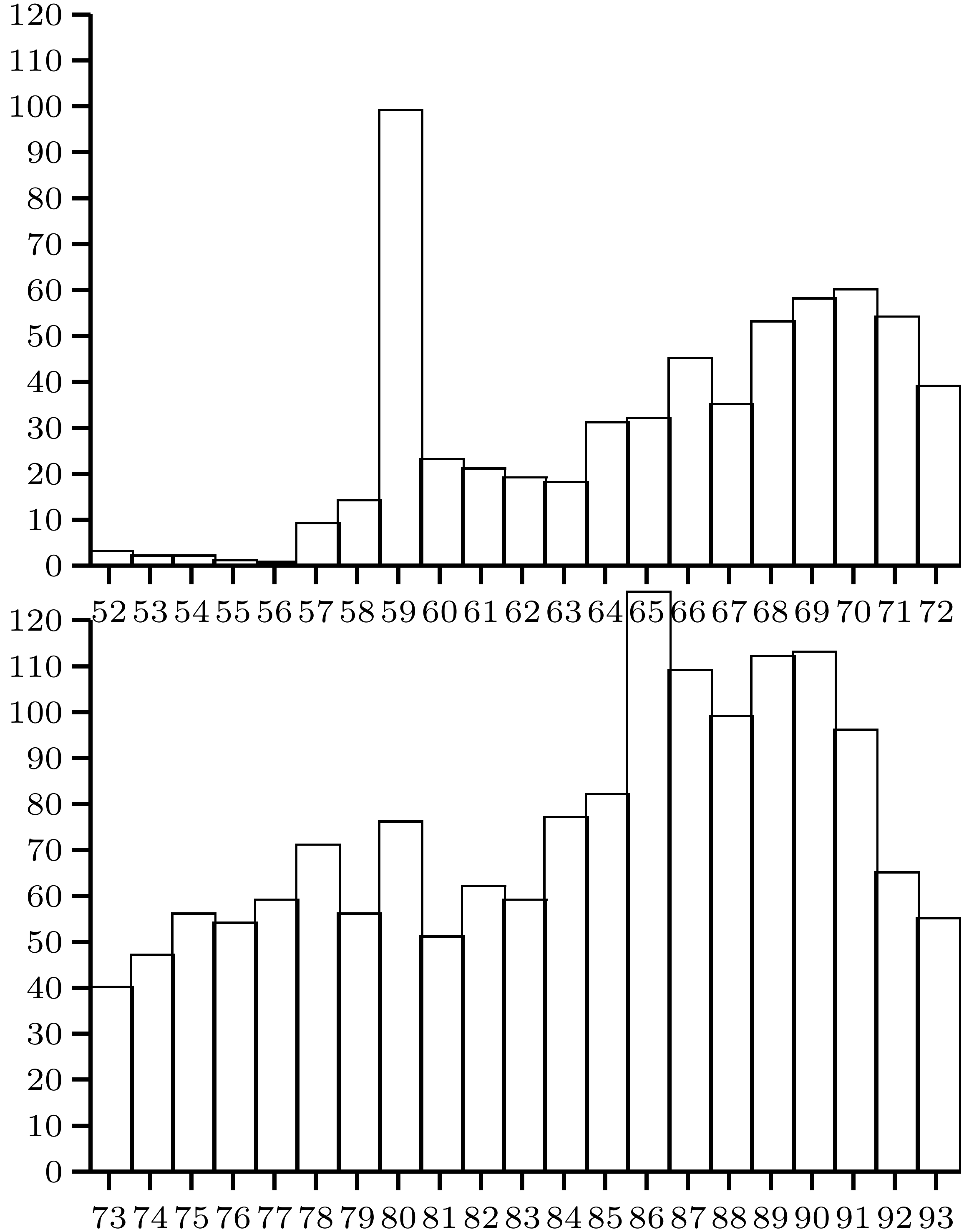
The WWW pages:

- aim to contain all information;
- provide on-line documentation; and
- may be updated frequently. Specifically check the Announcments section every morning before the class

Reasons to learn compiler technology:

- understand exisiting languages;
- appreciate current limitations;
- talk intelligently about language design;
- implement your very own general-purpose language; and
- implement lots of useful domain-specific languages.

New programming languages per year:



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:

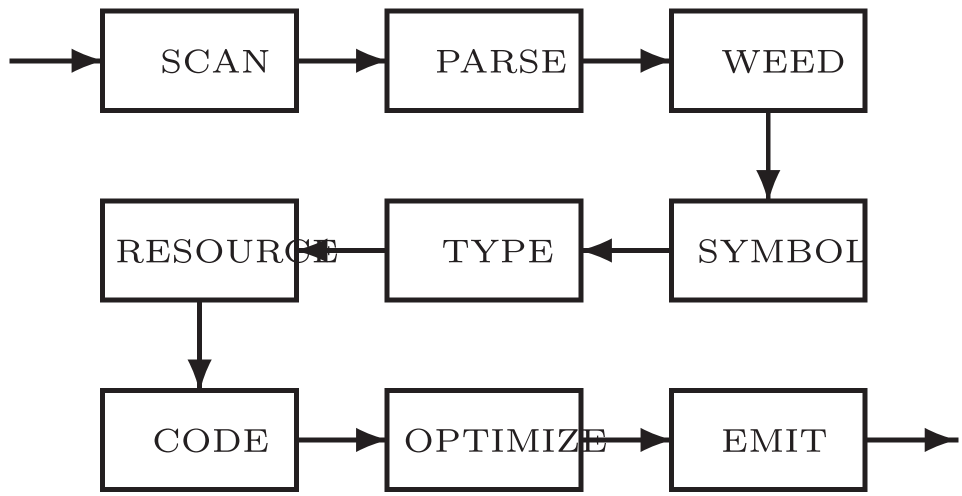
- LaTeX;
- `yacc` and `lex`;
- HTML;
- XML;
- ...

Domain-specific languages require full-scale compiler technology.

The FORTRAN compiler:

- implemented in 1954–1957;
- the world's first compiler;
- was motivated be 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.

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 JOOS project:

- Java's Object-Oriented Subset;
- compiled into Java Virtual Machine code;
- illustrates a general-purpose language;
- shows client-side programming on the WWW;
- used to teach by example;
- the A− source code will be studied;
- and you will upgrade it into an A+ version.

The WiG project:

- Web Interface Generator;
- compiled into C-based CGI-scripts;
- illustrates a domain-specific language;
- shows server-side programming on the WWW;
- used to get hands-on experience;
- and you will implement the language from scratch.

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.

Should you program in "Optimized C"?

If you want a fast C program, should you use `LOOP #1` or `LOOP #2`?

```
/* LOOP #1 */
for (i = 0; i < N; i++) {
    a[i] = a[i] * 2000;
    a[i] = a[i] / 10000;
}

/* LOOP #2 */
b = a;
for (i = 0; i < N; i++) {
    *b = *b * 2000;
    *b = *b / 10000;
    b++;
}
```

What would the expert programmer do?

If you said `LOOP #2` ... you were wrong!

`LOOP`	opt. level	SPARC	MIPS	Alpha
`#1` (array)	no opt	20.5	21.6	7.85
`#1` (array)	opt	8.8	12.3	3.26
`#1` (array)	super	7.9	11.2	2.96
`#2` (ptr)	no opt	19.5	17.6	7.55
`#2` (ptr)	opt	12.4	15.4	4.09
`#2` (ptr)	super	10.7	12.9	3.94

- Pointers confuse most C compilers; don't use pointers instead of array references.
- Compilers do a good job of register allocation; don't try to allocate registers in your C program.
- In general, write clear C code; it is easier for both the programmer and the compiler to understand.

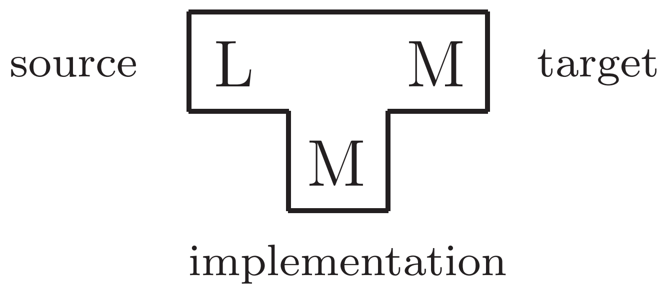
The top 10 list of reasons why we use Java for compilers:

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` which was developed at McGill;
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:

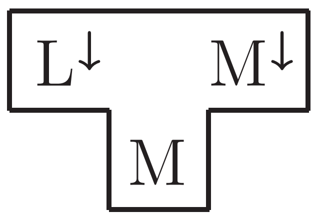


The direct approach is hard and difficult, and we really want to implement L in L itself.

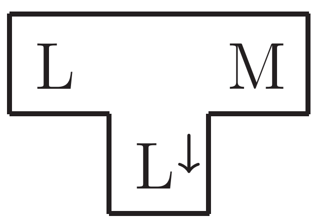
Define the following:

- $L\downarrow$ is a simple subset of L; and
- $M\downarrow$ is inefficient M-code.

We can easily implement:

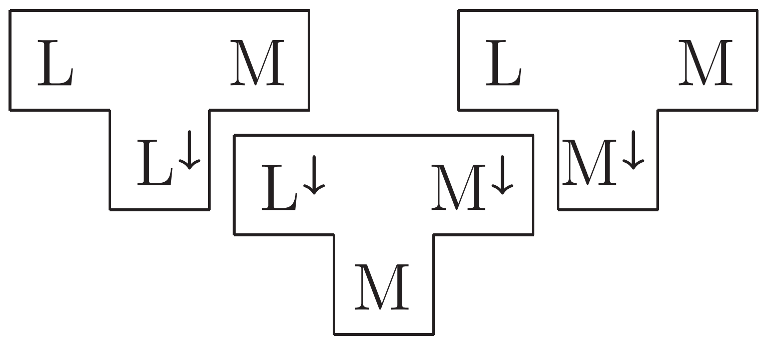


and (in parallel) implement:



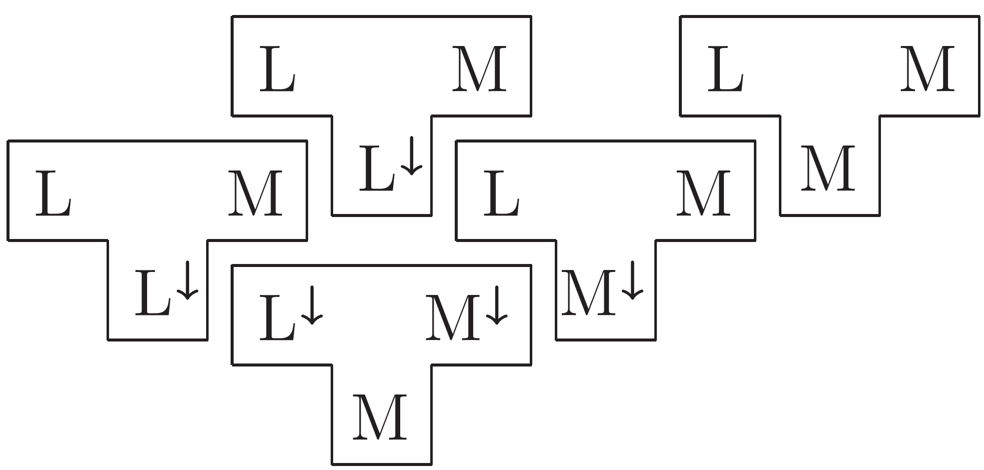
using basically our favorite language.

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.