CS415 Compilers
Overview of the Course

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Welcome to CS415 — Compilers

Topics in the design of programming language translators, including parsing, run-time storage management, error recovery, code generation, and optimization

- Instructor: Zheng Zhang (zz124@cs.rutgers.edu)
- Teaching Assistant: Jan Vesely (jan.vesely@rutgers.edu)
- My Office Hours: Wednesday, 3:00-4:00pm, CoRE310
- Required Text: Engineering a Compiler, Morgan-Kaufmann
  → Lab handouts, homework, slides …
  → I will not have handouts in class; get them from the web
- News group: sakai system sakai.rutgers.edu
  → Homework and project questions
- Get account on ilab
No recitation this week (today!)

Special permission numbers
please come and see me after class
Basis for Grading

• Exams
  → Midterm 30%
  → Final 35%
• Recitation Attendance 5%
• Projects (tentative!)
  → Instruction scheduling 10%
  → Front End 10%
  → Code generator/vectorizer 10%

Notice: This grading scheme and projects are tentative and subject to change.
## Basis for Grading

<table>
<thead>
<tr>
<th>Exams</th>
<th>Homework</th>
<th>Projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ Midterm</td>
<td></td>
<td>→ Instruction Scheduling</td>
</tr>
<tr>
<td>→ Final</td>
<td></td>
<td>→ Front End</td>
</tr>
<tr>
<td></td>
<td></td>
<td>→ Code generator</td>
</tr>
<tr>
<td><em>Closed-notes, closed-book</em></td>
<td><em>Reinforce concepts, provide practice</em></td>
<td><em>High ratio of thought to programming</em></td>
</tr>
<tr>
<td><em>final is cumulative</em></td>
<td></td>
<td><em>single student labs (note academic integrity information)</em></td>
</tr>
</tbody>
</table>
Rough Syllabus

- Overview § 1
- Local Register Allocation § 13
- Instruction Scheduling § 12
- Scanning § 2
- Parsing § 3
- Context Sensitive Analysis § 4
- Inner Workings of Compiled Code § 6, 7
- Introduction to Optimization § 8
- Code Selection § 11
- More Optimization (*time permitting*)
- Advanced topics in language design/compilation:
  automatic parallelization / vectorization
Class-taking technique for CS 415

- I will use projected material extensively
  → I will moderate my speed
- You should read the book
  → Not all material will be covered in class
  → Book complements the lectures
- You are responsible for material from class
  → The tests will cover both lecture and reading
  → I will probably hint at good test questions in class
- CS 415 is not a programming course
  → Projects are graded on functionality, documentation, and project reports more than style. However, things should be reasonable
- Use the resources provided to you
  → See me or the TA in office hours if you have questions
  → Post questions regarding homework and projects on message group
About Textbooks

• “Engineering a Compiler” by Cooper and Torczon
  → First or second edition

• Book presents modern material
  → Considers classical problems of post-1986 computers
  → Addresses modern techniques

• Other recommended textbook is the “Dragon Book”
  → Older version (Dragon book) also fine
• What is a (formal) language?
  → Words are finite sequences of symbols taken from a finite alphabet; A language is a set of words over a finite alphabet
  → Languages can be finite or infinite

• How to specify a language?
  → The more structure the language has, the more complex its specification
  → One option: use a “rewrite rule” based system to generate words in the language

• How to recognize whether a word is in a language or not?
  → The more structure the language has, the more complex its recognition
  → One option: use an abstract, “mathematical” machine to “parse” a word
• **What is a compiler?**
  → A program that translates an *executable* program in one language into an *executable* program in another language
  → A good compiler should improve the program, *in some way*

• **What is an interpreter?**
  → A program that reads an *executable* program and produces the results of executing that program

• *C* is typically compiled, *Scheme* is typically interpreted
• *Java* is compiled to bytecode (code for the Java VM)
  → which is then interpreted
  → Or a hybrid strategy is used
    ▪ Just-in-time compilation
    ▪ Dynamic optimization (hot paths)
Why Study Compilation?

• **Compilers are important system software components**
  → They are intimately interconnected with architecture, systems, programming methodology, and language design

• **Compilers include many applications of theory to practice**
  → Scanning, parsing, static analysis, instruction selection

• **Many practical applications have embedded languages**
  → Commands, macros, ...

• **Many applications have input formats that look like languages,**
  → Matlab, Mathematica

• **Writing a compiler exposes practical algorithmic & engineering issues**
  → Approximating hard problems; efficiency & scalability
Compiler construction involves ideas from many different parts of computer science.

<table>
<thead>
<tr>
<th>Artificial intelligence</th>
<th>Greedy algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Heuristic search techniques</td>
</tr>
<tr>
<td>Algorithms</td>
<td>Graph algorithms, union-find</td>
</tr>
<tr>
<td></td>
<td>Dynamic programming</td>
</tr>
<tr>
<td>Theory</td>
<td>DFAs &amp; PDAs, pattern matching</td>
</tr>
<tr>
<td></td>
<td>Fixed-point algorithms</td>
</tr>
<tr>
<td>Systems</td>
<td>Allocation &amp; naming, synchronization,</td>
</tr>
<tr>
<td></td>
<td>parallelism, data locality</td>
</tr>
<tr>
<td>Architecture</td>
<td>Pipeline &amp; hierarchy management</td>
</tr>
<tr>
<td></td>
<td>Instruction set use</td>
</tr>
</tbody>
</table>
Compiler construction poses challenging and interesting problems:

- Compilers must do a lot but also run fast
- Compilers have primary responsibility for run-time performance
- Compilers are responsible for making it acceptable to use the full power of the programming language
- Computer architects perpetually create new challenges for the compiler by building more complex machines (e.g.: many-core)
- Compilers must hide that complexity from the programmer
- Success requires mastery of complex interactions
It was our belief that if FORTRAN, during its first months, were to translate any reasonable “scientific” source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger... I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.

— John Backus
Implications

- Must recognize legal (and illegal) programs
- Must generate correct code
- Must manage storage of all variables (and code)
- Must agree with OS & linker on format for object code

*Big step up from assembly language—use higher level notations*
Implications

- Use an intermediate representation (IR)
- Front end maps legal source code into IR
- Back end maps IR into target machine code
- Extension: multiple front ends & multiple passes (better code)

Typically, front end is $O(n)$ or $O(n \log n)$, while back end is NPC
Can we build $n \times m$ compilers with $n+m$ components?

- Must encode all language specific knowledge in each front end
- Must encode all features in a single IR
- Must encode all target specific knowledge in each back end

Limited success in systems with very low-level IRs
Responsibilities

- Recognize legal (& illegal) programs
- Report errors in a useful way
- Produce IR & preliminary storage map
- Shape the code for the back end
- Much of front end construction can be automated
The Front End

Scanner

- Maps character stream into words—the basic unit of syntax
- Produces pairs — a word & its part of speech
  \[ x = x + y \] becomes \[ <\text{id},x> = <\text{id},x> + <\text{id},y> \]
  \[ \rightarrow \text{word} \equiv \text{lexeme}, \text{part of speech} \equiv \text{token type} \]
  \[ \rightarrow \text{In casual speech, we call the pair a token} \]
- Typical tokens include number, identifier, +, -, new, while, if
- Scanner eliminates white space (including comments)
- Speed is important
The Front End

- **Parser**
  - Recognizes context-free syntax & reports errors
  - Guides context-sensitive (“semantic”) analysis (type checking)
  - Builds IR for source program

*Hand-coded parsers are fairly easy to build*

*Most books advocate using automatic parser generators*
Context-free syntax is specified with a grammar:

\[
\text{SheepNoise} \rightarrow \text{SheepNoise} \ baa \\
| \ baa
\]

This grammar defines the set of noises that a sheep makes under normal circumstances.

It is written in a variant of Backus-Naur Form (BNF).

Formally, a grammar \( G = (S,N,T,P) \):

- \( S \) is the start symbol
- \( N \) is a set of non-terminal symbols
- \( T \) is a set of terminal symbols or words
- \( P \) is a set of productions or rewrite rules \( (P : N \rightarrow N \cup T) \)
The Front End

Context-free syntax can be put to better use

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><strong>goal</strong> → <strong>expr</strong></td>
</tr>
<tr>
<td>2.</td>
<td><strong>expr</strong> → <strong>expr</strong> <strong>op</strong> <strong>term</strong></td>
</tr>
<tr>
<td>3.</td>
<td><strong>term</strong> → <strong>number</strong></td>
</tr>
<tr>
<td>4.</td>
<td><strong>term</strong> → <strong>id</strong></td>
</tr>
<tr>
<td>5.</td>
<td><strong>op</strong> → <strong>+</strong></td>
</tr>
<tr>
<td>6.</td>
<td><strong>op</strong> → <strong>-</strong></td>
</tr>
</tbody>
</table>

- This grammar defines simple expressions with addition & subtraction over “number” and “id”
- This grammar, like many, falls in a class called “context-free grammars”, abbreviated CFG

\[
\begin{align*}
S &= \text{goal} \\
T &= \{ \text{number, id, +, -} \} \\
N &= \{ \text{goal, expr, term, op} \} \\
P &= \{ 1, 2, 3, 4, 5, 6, 7 \}
\end{align*}
\]
Given a CFG, we can *derive* sentences by repeated substitution.

<table>
<thead>
<tr>
<th>Production</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal</td>
<td>expr</td>
</tr>
<tr>
<td>1</td>
<td>expr op term</td>
</tr>
<tr>
<td>2</td>
<td>expr op y</td>
</tr>
<tr>
<td>5</td>
<td>expr - y</td>
</tr>
<tr>
<td>7</td>
<td>expr op term - y</td>
</tr>
<tr>
<td>2</td>
<td>expr op term - y</td>
</tr>
<tr>
<td>4</td>
<td>expr op 2 - y</td>
</tr>
<tr>
<td>6</td>
<td>expr + 2 - y</td>
</tr>
<tr>
<td>3</td>
<td>term + 2 - y</td>
</tr>
<tr>
<td>5</td>
<td>x + 2 - y</td>
</tr>
</tbody>
</table>

To recognize a valid sentence in some CFG, we reverse this process and build up a *parse*. 
A parse can be represented by a tree \((\text{parse tree or syntax tree})\)

\[ x + 2 - y \]

This contains a lot of unneeded information.
Compilers often use an abstract syntax tree

The AST summarizes grammatical structure, without including detail about the derivation

This is much more concise

ASTs are one kind of intermediate representation (IR)
Responsibilities

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces

Automation has been *less* successful in the back end
The Back End

Instruction Selection
• Produce fast, compact code
• Take advantage of target features such as addressing modes
• Usually viewed as a pattern matching problem
  → *ad hoc* methods, pattern matching, dynamic programming
Register Allocation

- Have each value in a register when it is used
- Manage a limited set of resources
- Select appropriate LOADs & STOREs
- Optimal allocation is NP-Complete

Typically, compilers approximate solutions to NP-Complete problems
Instruction Scheduling

- Avoid hardware stalls and interlocks
- Use all functional units productively
- Can increase lifetime of variables (changing the allocation)

Optimal scheduling is NP-Complete in nearly all cases

Heuristic techniques are well developed
Traditional Three-pass Compiler

Code Improvement (or Optimization)

- Analyzes IR and rewrites (or transforms) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power dissipation, energy consumption, ...
- Must preserve “meaning” of the code
  - Measured by values of named variables
Next class

Registers Allocation

Read EaC: Chapters 13.1 – 13.3

No recitation this week (today!)