Compilers

Advanced Systems Elective
Discussion question

- What is a compiler?

Error: obscure syntax mistake [main.cpp:375] !!!

"An angry translator."
-- previous CS 432 student
Automated translation

- A **compiler** is a computer program that **automatically translates** other programs from one language to another
  - (usually from a *human-readable* language to a *machine-executable* language, but not necessarily)
Automated translation

- Compilation vs. interpretation:

```
Program in Source Language
         v
Compiler
         ^
Program in Target Language

Program in Source Language
         v
Interpreter
         ^
Results
```
Discussion question

• Why should we study compilers?
  – (besides getting systems elective credit...)
Compilers: a convergent topic

- Data structures
  - CS 240
- Architectures, machine languages, and operating systems
  - CS 261, CS 450
- Automata and language theory
  - CS 327, CS 430
- Graph algorithms
  - CS 327
- Software and systems engineering
  - CS 345, CS 361
- Greedy, heuristic, and fixed-point algorithms
  - CS 452
Reasons to study compilers

• Shows how many areas of CS can be combined to solve a truly "hard" problem (automated language translation)
• Bridges theory vs. implementation gap
  – Theory informs implementation
  – Applicable in many other domains
• Practical experience with large(er) software systems
  – My master copy is over 7K LOC
  – Of this, you will re-write over 1K LOC this semester
• Exposure to open problems in CS
  – Many optimization issues are subject to ongoing research
Course goal

• Fundamental question
  – "How do compilers translate from a human-readable language to a machine-executable language?"

• After this course, your response should be:
  – "It's really cool! Let me tell you..."
Course design theory

• First, a bit of background ...
Course design theory

- Big ideas
  - E.g., "A compiler is a large software system consisting of a sequence of phases"
- "Enduring understandings" (stuff you should remember in five years)
  - E.g, "Large problems can sometimes be solved by composing existing solutions to smaller problems."
- Learning objectives (stuff you should remember at the end of the course)
  - E.g., "Identify the technical challenges of building a large software system such as a compiler."

- Activities and assignments flow from learning objectives
  - E.g., "Draw a diagram illustrating the major phases of a modern compiler."
- Exams reflect activities and assignments
- Goal: “engaged” and effective learning
Learning objectives

- Identify and discuss the technical and social challenges of building a large software system such as a compiler.
- Develop and analyze formal descriptions of computer languages.
- Apply finite automata theory to build recognizers (lexers) for regular languages.
- Apply pushdown automata theory to build recognizers (parsers) for context-free languages.
- Evaluate the role of static analysis in automated program translation.
- Apply tree traversals to convert a syntax tree to low-level code.
- Discuss the limitations that an architecture or execution environment places on the generation of machine code.
- Describe common optimizations and evaluate the tradeoffs associated with good optimization.
Course format

- **Website:** [https://w3.cs.jmu.edu/lam2mo/cs432/](https://w3.cs.jmu.edu/lam2mo/cs432/)
  - Make sure you’re using the right year’s website!

- **Weekly schedule (roughly)**

<table>
<thead>
<tr>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In-class</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recap &amp; new</td>
<td>Initial reading</td>
<td>Mini-lecture and</td>
<td>In-class lab</td>
<td></td>
</tr>
<tr>
<td>topic intro</td>
<td>&amp; quiz</td>
<td>discussion</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Out-of-class</strong></td>
<td>Project work</td>
<td>Project work</td>
<td>Project work</td>
<td>Project work</td>
</tr>
<tr>
<td>Project work</td>
<td>Project work</td>
<td>Project work</td>
<td>Project work</td>
<td>Project work</td>
</tr>
</tbody>
</table>

- **Formative vs. summative assessment**
  - Formative: quizzes and labs (20% of final grade)
  - Summative: projects and exams (80% of final grade)
Course textbook(s)

- **Engineering a Compiler, 2\textsuperscript{nd} Edition**
  - Keith Cooper and Linda Torczon
  - 1\textsuperscript{st} chapter scanned; posted under “Files” on Canvas
  - Reserve copy at Rose library

- **Compilers: Principles, Techniques, & Tools, 2\textsuperscript{nd} Edition**
  - Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman
  - “The Dragon Book” (definitive text on compilers)
  - Two chapters scanned; posted under “Files” on Canvas

- **Decaf/ILOC references**
  - PDFs on website
Semester-long project

- Compiler for "Decaf" language
  - Implementation in Java
  - Maven build system w/ fully-integrated test suite
  - Compiles to ILOC (new machine language from textbook)
  - Five major projects: "pieces" of the full system

- Submission: code + review + response
  - Code can be written in teams of two
    - Benefits vs. costs of working in a team
  - Individual graded code reviews due a week later
  - Review responses (how did your reviewer do?)
Evolution of CS 432

- Fall 2015 - special topics (CS 480)
- Fall 2016 - first time taught as CS 432
  - First time teaching CS 261 as well
- Fall 2017
  - Expanded test suite significantly
  - Added type systems and lambda calculus
- Fall 2018
  - Finished pure visitor implementation for P5
  - Added Y86 translator
  - Removed lambda calculus
  - Added exam study guides
- Fall 2019
  - Removed reflection paper
  - Switch to Dragon book for LR parsing
  - Streamlined calling conventions
Course policies

• Questions?
Compiler rule #1

- "The compiler must preserve the meaning of the program being compiled."
  - What is a program's meaning?
Compilers encode a program's meaning using an intermediate representation (IR)

- Tree- or graph-based: abstract syntax tree (AST), control flow graph (CFG)
- Linear: register transfer language (RTL), Java bytecode, intermediate language for an optimizing compiler (ILOC)

\[
\begin{align*}
    &= \text{a} \\
    = &\text{c} \times \text{d} \\
    \text{+} &\text{b} \\
    \text{load } b &\rightarrow r1 \\
    \text{load } c &\rightarrow r2 \\
    \text{add } r1, r2 &\rightarrow r3 \\
    \text{load } d &\rightarrow r4 \\
    \text{mult } r3, r4 &\rightarrow r5 \\
    \text{store } r5 &\rightarrow a
\end{align*}
\]
Standard compiler framework

- **Front end**: understand the program (src → IR)
- **Back end**: encode in target language (IR → targ)
- **Primary benefit**: easier *re-targeting* to different languages or architectures
Modern compiler framework

- Front-end passes
  - Scanning (lexical analysis)
  - Parsing (syntactic analysis)

- Middle-end passes
  - Static/semantic analysis
  - IR code generation
  - IR optimization

- Back-end passes
  - Instruction selection
  - Machine code optimization
  - Register allocation
  - Instruction scheduling
  - Assembling/linking

- Modern approach: nanopasses
  - Dozens or hundreds of passes (https://llvm.org/docs/Passes.html)
Our Decaf compiler

Source code

```c
int main() {
    int x = 4 + 5;
    return x;
}
```

Tokens

```
x + 4 5
```

Syntax tree

```
=  
 / 
 x + 
   / 
  4 5
```

Checked AST + Symtables

```
✓
✓
✓
✓
✓
```

Lexing (P2)

Parsing (P3)

Analysis (P4)

Checked AST + Symtables

```
✓
✓
✓
✓
✓
```

IR Code Gen (P5)

Register Allocation (P6)

Machine Code Gen

Run via ILOC interpreter

```
main:
    loadI 4 => r1
    loadI 5 => r2
    add r1, r2 => r3
    i2i r3 => RET
```

Optimized Linear IR

```
main:
    loadI 4 => r0
    loadI 5 => r1
    add r0, r1 => r0
    i2i r0 => RET
```

Run via yas + 261 P4

```
irmovq $4, %rcx
irmovq $5, %rdx
addq %rcx, %rdx
rrmovq %rdx, %rax
ret
```
The compiler should *help* the programmer in some way

- What does *help* mean?
Discussion question

• What would be your design goals for a compiler?
  – E.g., what functionality or properties would you like it to have?
  – (Besides rule #1 – correct translation)
Compiler design goals

- Optimize for fast execution
- Minimize memory/energy use
- Catch software defects early
- Provide helpful error messages
- Run quickly
- Be easily extendable
Differing design goals

What differences might you expect in compilers designed for the following applications?

- A just-in-time compiler for running server-side user scripts
- A compiler used in an introductory programming course
- A compiler used to build scientific computing codes to run on a massively-parallel supercomputer
- A compiler that targets a number of diverse systems
- A compiler that targets an embedded sensor network platform
Decaf language

- Simple imperative language similar to C or Java
- Example:

```java
// add.decaf - simple addition example

def int add(int x, int y)
{
    return x + y;
}

def int main()
{
    int a;
    a = 3;
    return add(a, 2);
}
```

$ java -jar decaf-1.0.jar -i add.decaf

5
Before Wednesday

- **Readings**
  - "Engineering a Compiler" (EAC) Ch. 1 (23 pages)
  - Decaf reference ("Resources" page on website)

- **Tasks**
  - Complete first reading quiz on Canvas
  - Complete course intro survey on Canvas
  - Download the reference compiler from Canvas ("Files")
    - Bring your laptop on Wednesday!
  - Write some code in Decaf
  - Install the JDK and Maven on your system
Upcoming events

- CS senior night
  - Wednesday, Sept. 4 (5-6:30pm ISAT/CS 259)
  - Graduation info, job fairs, photos, etc.
  - Senior students only

- CISE career fair
  - Wednesday, Sept. 18 (Festival Ballroom)
  - Companies looking for technology majors
  - Internships and full-time positions
See you Wednesday

• Have a great semester!