CS412/413

Introduction to Compilers and Translators

Lecture 1: Overview
23 Jan 06

Outline

• Course Organization
  – General course information
  – Homework & project information

• Introduction to Compilers
  – What are compilers?
  – Why do we need compilers?
  – General compiler structure

General Information

When
MWF 10:10 - 11:00AM
Where
HO 110
Instructor
Radu Rugina
Teaching Assistant
Maksim Ōlovich
Course staff email
cs412-10cs.cornell.edu
Web page
courses.cs.cornell.edu/cs412
Newsgroup
cornell.class.412

Important

• CS 413 is required!
• Large implementation project
• Substantial amount of theory

Textbooks

• Optional texts
  – Compilers -- Principles, Techniques and Tools
    (Dragon Book), by Aho, Sethi and Ullman (1986)
  – Modern Compiler Implementation in Java
    by Andrew Appel (2002)
  – Engineering a Compiler
    by Linda Torczon and Keith Cooper (2003)
• They are on reserve in Engineering Library

Work Distribution

• Theory:
  – Homeworks = 20%
    • 4 homeworks: 5% each
  – Exams = 35%
    • 2 prelims: 17% and 18%; no final exam
    • Prelims on: March 9, April 27 (evening)
• Practice:
  – Programming Assignments = 45%
    • 6 assignments: 5/9/9/9/9/9
    • Project demo
Homeworks

• 4 homework assignments
  – Three assignments in first half of course
  – One homework in second half

• Not done in groups
  – do your own work

Project

• Build a full compiler:
  – Compile a subset of Java
  – Generate assembly x86
  – Implementation in Java 1.5

• 5 programming assignments

• Groups of 3-4 students
  – Usually same grade for all
  – Form your group in CMS!

Assignments

• Due at beginning of class
  – Homeworks: paper turn in (at beginning of class)
  – Project files: electronic turn in (day before class)
  – Assignments managed with CMS

• Late homeworks, project submissions
  – Avoid late submissions
  – Late submission penalty: 10% per day

Why Take This Course?

• CS412/413 is an elective course

• Reason #1: understand compilers/languages
  – Understand the code structure
  – Understand the language semantics
  – Understand the relation between source code and generated machine code
  – Become a better programmer

Why Take This Course? (ctd.)

• Reason #2: nice balance of theory and practice:
  – Theory:
    • Many mathematical models: regular expressions, automata, grammars, graphs, lattices
    • Lots of algorithms that use these models
  – Practice:
    • Apply theory to build a real compiler
    • Better understand why “theory and practice are the same in theory, but in practice they are different”

Why Take This Course? (ctd.)

• Reason #3: Programming experience
  – Write a large program that manipulates complex data structures
  – Software development in groups
  – Learn more about Java and Intel x86 architecture and assembly language
What Are Compilers?

- Compilers = translate information from one representation into another
- Usually information = program
- Typically:
  - Compilers refer to the translation from high-level source code to low-level code (e.g. object code)
  - Translators refer to the transformation at the same level of abstraction

Examples

- Typical compilers: gcc, javac
- Non-typical compilers:
  - \texttt{latex} (document compiler):
    - Transforms a \LaTeX document into DVI printing commands
  - \texttt{C-to-Hardware} compiler:
    - Generates hardware circuits for C programs

Related Paradigms

- Interpretation
  - Interpreter executes source program
  - You’ve seen them in CS211, CS312
- Compilation for a Virtual Machine
  - E.g., Java bytecode compilation
  - Portable compilation
- JIT (Just-in-Time) Compilation
  - Dynamic compilation

In This Class

- We will study typical compilation:
  - from programs written in high-level languages
  - to low-level machine-specific assembly code

Why Do We Need Compilers?

- It is difficult to write, debug, maintain, and understand programs written in assembly language
- Tremendous increase in productivity when first compilers appeared (∼50 years ago)
- There are still a few cases when people manually write assembly code
  - E.g. to access low-level machine resources such as device drivers
  - These code fragments are very small; the compiler handles the rest of the code in the application

Overall Compiler Structure
Source Code

- Optimized for human readability
  - Matches human notions of grammar
  - Uses named constructs such as variables and procedures

```c
int expr(int n)
{
    int d;
    d = 4 * n * n * (n + 1) * (n + 1);
    return d;
}
```

Assembly and Machine Code

- Optimized for hardware
  - Consists of machine instructions; uses registers and unnamed memory locations
  - Much harder to understand by humans

```assembly
lda $30,-32($30)
stq $26,0($30)
stq $15,8($30)
bis $30,$30,$15
bis $16,$16,$1
stl $1,16($15)
lds $f1,16($15)
sts $f1,24($15)
ldl $5,24($15)
bis $5,$5,$2
s4addq $2,0,$3
ldl $4,16($15)
mull $4,$3,$2
ldl $3,16($15)
addq $3,1,$4
mull $2,$4,$2
stl $2,20($15)
ldl $0,20($15)
br $31,$33
$33:
bis $15,$15,$30
ldq $26,0($30)
ldq $15,8($30)
addq $30,32,$30
ret $31,($26),1
```

Translation Efficiency

- **Goal**: generate machine code that describes the same computation as the source code
- Is there a unique translation?
- Is there an algorithm for an “ideal translation”? (ideal = either fastest or smallest generated code)
- Compiler optimizations = find better translations!

Example: Output Assembly Code

Unoptimized Code

```

```

Optimized Code

```

Translation Correctness

- The generated code must execute precisely the same computation as in the source code
- Correctness is very important!
  - hard to debug programs with broken compiler...
  - implications for development cost, security
  - this course: techniques proved to ensure correct translation

How To Translate?

- Translation is a complex process
  - source language and generated code are very different
- Structure the translation
  - Define intermediate steps
  - At each step use a specific program representation
  - More machine-specific, less language-specific as translation proceeds
**Simplified Compiler Structure**

- Source code
- if (b == 0) a = b;
- Understand source code
- Front end (machine-independent)
  - Intermediate code
  - Optimize
- Back end (machine-dependent)
  - Interim code
  - Generate assembly code
  - Assembly code
  - cmp $0, ecx
  - cmovz edx, ecx

---

**Simplified Front-End Structure**

- Source code (character stream)
- if (b == 0) a = b;
- Front end (machine-independent)
  - Lexical Analysis
  - Token stream
  - Syntax Analysis
  - Abstract syntax tree
  - Semantic Analysis
  - Abstract syntax tree

---

**Analogy**

- Front end can be explained by analogy to the way humans understand natural languages

  - Lexical analysis
    - Natural language: "He wrote the program"
    - words: "he" "wrote" "the" "program"
    - Programming language: "if (b == 0) a = b"
    - tokens: "if" "(" "b" "==" "0" ")" "a" "=" "b"

---

**Analogy (ctd)**

- Syntactic analysis
  - Natural language:
    - He
    - noun
    - wrote
    - verb
    - the
    - noun
    - Syntax is correct; semantics is wrong!
  - Programming language:
    - if (b == 0) a = b
    - a is an integer and foo is a method, the compiler will complain.

---

**Beyond Assembly Code**

- Source code
- Compiler
  - Lexical Analysis
  - Syntax Analysis
  - Semantic Analysis
  - Optimization
  - Code Generation
- Object code (machine code)
  - Assembler
  - Linker
  - Fully-resolved object code (machine code)
  - Loader
- Executable image
## Tentative Schedule

<table>
<thead>
<tr>
<th>Topic</th>
<th>Lectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexical analysis</td>
<td>3</td>
</tr>
<tr>
<td>Syntax analysis</td>
<td>6</td>
</tr>
<tr>
<td>Semantic analysis</td>
<td>5</td>
</tr>
<tr>
<td>Prelim #1</td>
<td></td>
</tr>
<tr>
<td>Simple code generation</td>
<td>6</td>
</tr>
<tr>
<td>Analysis</td>
<td>8</td>
</tr>
<tr>
<td>Optimizations</td>
<td>3</td>
</tr>
<tr>
<td>Prelim #2</td>
<td></td>
</tr>
<tr>
<td>Advanced topics</td>
<td>6</td>
</tr>
</tbody>
</table>