CS42/413

Introduction to Compilers and Translators

Lecture 1: Overview
26 Jan 04

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

<table>
<thead>
<tr>
<th>When</th>
<th>MWF 10:10 - 11:00AM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where</td>
<td>HO 110</td>
</tr>
<tr>
<td>Instructor</td>
<td>Radu Rugina</td>
</tr>
<tr>
<td></td>
<td>Cristian Bucila, Richard Chung</td>
</tr>
<tr>
<td>Teaching Assistants</td>
<td>courses.cs.cornell.edu/cs412</td>
</tr>
<tr>
<td>Course staff email</td>
<td><a href="mailto:cs412-l@cs.cornell.edu">cs412-l@cs.cornell.edu</a></td>
</tr>
<tr>
<td>Web page</td>
<td>courses.cs.cornell.edu/cs412</td>
</tr>
<tr>
<td>Newsgroup</td>
<td>cornell.class.412</td>
</tr>
</tbody>
</table>

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:
  – Homworks = 20%
    • 4 homeworks: 5% each
  – Exams = 35%
    • 2 prelims: 17% and 18%; no final exam
    • Prelims on: March 9, April 27

• Practice:
  – Programming Assignments = 45%
    • 6 assignments: 5/9/9/9/9
    • 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

- Implementation:
  - Designed language = a subset of Java
  - Generated code = assembly x86
  - Implementation language = Java
- 5 programming assignments
- Groups of 3–4 students
  - Usually same grade for all
  - Group information due Friday
  - We will respect consistent preferences

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, programming assignments increasingly penalized
  - Penalty linearly increasing: 10% per day
  - 1 day: 10%, 2 days: 20%, 3 days: 30%, etc.

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:
    - Lots of mathematical models: regular expressions, automata, grammars, graphs, lattices
    - Lots of algorithms which use these models
  - Practice:
    - Apply theoretical notions to build a real compiler
    - Better understand why “theory and practice are the same in theory; in practice they are different”

Why Take This Course? (ctd.)

- Reason #3: Programming experience
  - Write a large program which manipulates complex data structures
  - Learn how to be a better programmer in groups
  - Learn more about Java and Intel x86 architecture and assembly language
What Are Compilers?

- Compilers = translate information from one representation to another
- Usually information = program
- So compilers=translators, but 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:
  - latex (document compiler):
    - Transforms a LaTeX document into DVI printing commands
  - C-to-Hardware compiler:
    - Generates hardware circuits for C programs
    - Output is lower-level than typical compilers
- Translators:
  - f90: Fortran-to-C translator (both high-level)
  - latex2html: LaTeX-to-HTML (both documents)
  - dvi2ps: DVI-to-PostScript (both low-level)

In This Class

- We will study typical compilation: from programs written in high-level languages to low-level object code and machine code
- Most of the principles and techniques in this course apply to non-typical compilers and translators

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 few cases when it is better to manually write assembly code
  - E.g. to access low-level resources of the machine (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

```
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

Translation Efficiency

- Goal: generate machine code which 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

<table>
<thead>
<tr>
<th>Unoptimized Code</th>
<th>Optimized Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
- Need to 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

Intermediate code

Optimizer

Generate

assembly code

Assembly code

cmp $0,ax
cmp $0, edx, ecx

Front end
(machine-independent)

Back end
(machine-dependent)
Simplified Front-End Structure

Source code (character stream)

If \((b == 0) \) \(a = b;\)

Lexical Analysis

Lexical errors

Token stream

Syntax Analysis

Syntax errors

Abstract syntax tree

Semantic Analysis

Semantic errors

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" "="

- Semantic analysis
  - Natural language: He wrote the computer
    noun verb article noun
    Syntax is correct; semantics is wrong!
  - Programming language
    if \((b == 0) \) \(a = foo\)
    test assignment
    if \(a\) is an integer variable and \(foo\) is a procedure, then the semantic analysis will report an error

Analogy (ctd)

- Syntactic analysis
  - Natural language:
    He wrote the program
    subject predicate object
    sentence
  - Programming language
    if \((b == 0) \) \(a = b;\)
    assignment
    if-statement

Analogy (ctd)

Big Picture

Compiler

Source code

Lexical Analysis

Syntax Analysis

Semantic Analysis

Optimization

Code Generation

Assembler

Object code

(machine code)

Linker

Fully-resolved object code

(machine code)

Loader

Executable image

Tentative Schedule

Lexical analysis 3 lectures
Syntax analysis 6 lectures
Semantic analysis 5 lectures
Prelim #1
Simple code generation 5 lectures
Analysis 6 lectures
Optimizations 6 lectures
Prelim #2
Advanced topics 6 lectures