CS412/413

Introduction to Compilers and Translators
Spring 2002

Lecture 1: Overview
20 Jan 03

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
HO 110

Where

Instructor
Radu Rugina
Veselin Stoyanov, Ben Viglietta
Andy Goodell
Juanita Heyerman

Teaching Assistants

Admin Assistant

Course staff email
cs412@cs.cornell.edu

Web page
www.cs.cornell.edu/courses/cs412

Newsgroup
cornell.class.412

Important

• CS 413 is required!

• Large implementation project

• Substantial amount of theory

Textbooks

• Optional texts
  – Dragon Book: Compilers -- Principles, Techniques and Tools, by Aho, Sethi and Ullman
  – Tiger Book: Modern Compiler Implementation in Java, by Andrew Appel

• They are on reserve in Engineering Library

Work Distribution

• Theory:
  – Homeworks = 20%
    • 4 homeworks: 5/5/5/5
  – Exams = 35%
    • 2 prelims: 17/18; no final exam

• Practice:
  – Programming Assignments = 45%
    • 6 assignments: 5/8/8/8/8/8
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
- Six 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 the class)
  - Project files: electronic turn in (CSUGLAB directory)
- Late homeworks, programming assignments increasingly penalized
  - Penalty linearly increasing
  - 1 day: 10%, 2 days: 20%, 3 days: 30%, 4 days: 40%, 5 days: 50%, 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-Silicon compiler:
    - Generates hardware circuits for C programs
  - Output is lower-level than typical compilers
- Translators:
  - f2c: Fortran-to-C translator (both high-level)
  - latex2html: \LaTeX{}-to-HTML (both documents)
  - dvips: 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

```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

Translation Efficiency

- Goal: generate machine code which describes the same computation as the source code
- Is there a unique translation? NO!
- Is there an algorithm for an "ideal translation"? (ideal = either fastest or smallest generated code) NO!
- Compiler optimizations = find better translations!

Example: Output Assembly Code

<table>
<thead>
<tr>
<th>Unoptimized Code</th>
<th>Optimized Code</th>
</tr>
</thead>
</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
Simplified Front-End Structure

- Source code (character stream)
  \[ (b \equiv 0) \Rightarrow a = b; \]
  
  \[
  \begin{aligned}
  &\text{Lexical Analysis} \\
  &\text{Syntax Analysis} \\
  &\text{Semantic Analysis}
  \end{aligned}
  \]

  \[
  \begin{aligned}
  &\text{Token stream} \\
  &\text{Abstract syntax tree} \\
  &\text{Abstract syntax tree}
  \end{aligned}
  \]

  \[
  \begin{aligned}
  &\text{Lexical errors} \\
  &\text{Syntax errors} \\
  &\text{Semantic errors}
  \end{aligned}
  \]

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 \equiv 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
    \[
    if (b \equiv 0) a = \text{foo}
    \]
    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: noun verb article object
    - Sentence
  - Programming language
    \[
    if (b \equiv 0) a = b
    \]
    test assignment

Big Picture

- Compiler
  - Source code
    \[
    \begin{aligned}
    &\text{Lexical Analysis} \\
    &\text{Syntax Analysis} \\
    &\text{Semantic Analysis} \\
    &\text{Optimization}
    \end{aligned}
    \]
  - Assembly code
    \[
    \begin{aligned}
    &\text{Assembler}
    \end{aligned}
    \]
  - Object code (machine code)
    \[
    \begin{aligned}
    &\text{Linker}
    \end{aligned}
    \]
  - Fully-resolved object code (machine code)
    \[
    \begin{aligned}
    &\text{Loader}
    \end{aligned}
    \]
  - Executable image

Tentative Schedule

- Lexical analysis 3 lectures
- Syntax analysis 5 lectures
- Semantic analysis 4 lectures
- Objects 3 lectures
- Prelim #1 3 lectures
- Intermediate code 9 lectures
- Analysis/optimizations 5 lectures
- Code generation 6 lectures
- Prelim #2 6 lectures
- Advanced topics 6 lectures