CMSC 430 — “Theory of Language Translation”

Topics in the design of programming language translators, including scanning, parsing, error recovery, code generation, and code improvement.

Prerequisite: CMSC 330

Important facts:

Prof: Chau-Wen Tseng  
Email: tseng@cs.umd.edu  
Office: A.V. Williams 4135  
Hours: Tue & Thu 3:15–4:15pm

TA: Jik-Soo Kim  
Email: jiksoo@cs.umd.edu  
Office: A.V. Williams 1151  
Hours: Mon & Wed 9:30–10:30am

Class Email: cmsc430@cs.umd.edu  
Class URL: http://www.cs.umd.edu/class/fall2002/cmsc430  
Newsgroup: csd.cmsc430 news.umd.edu

Textbook is Modern Compiler Implementation by Andrew Appel
Course Overview

Basis for grades:

- 30% midterms, 20% final exam, 50% 5 programming projects

Programming Projects

- scanner construction (REs to minimal DFAs)
- scanner/parser using JLex and CUP
- simple type checker
- Java byte code generation
- advanced code generation, optimizations

Policies

- no collaboration (code sharing) allowed
- 1-week late policy, no incompletes

Lecture notes

- all lectures are on the Web, you should still take notes & read textbook
Compiler Overview

What is a compiler?

- a program that translates an *executable* program in one language into an *executable* program in another language
- the compiler typically *lowers* the level of abstraction of the program
- for “optimizing” compilers, we also expect the program produced to be *better*, in some way, than the original

Compilers are large, complex pieces of software. By working on compilers, you’ll learn to use

- programming tools (compilers, debuggers)
- program-generation tools (JLex, CUP)
- software libraries (Java class libraries)

*Hopefully you will also improve your programming and software engineering skills.*
Abstract view of compiler

source code → compiler → machine code

Implications:

- recognize legal (and illegal) programs
- generate correct code
- manage storage of all variables and code
- need format for object (or assembly) code

Big step up from assembler – higher level notations
Traditional two pass compiler

Implications:

- intermediate language ($il$)
- front end maps legal code into $il$
- back end maps $il$ onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes $\Rightarrow$ better code

$Front\ end\ is\ O(n)\ or\ O(n\ \log\ n)$

$Back\ end\ is\ NP-Complete$
Front end

Responsibilities:

• recognize legal procedure
• report errors
• produce \( il \)
• preliminary storage map
• shape the code for the back end

Much of front end construction can be automated
Scanner

- maps characters into tokens – the basic unit of syntax
  \[ x = x + y; \]
  becomes
  \[ <id, x> = <id, x> + <id, y> ; \]
- character string for a token is a lexeme
- typical tokens: number, id, +, −, *, /, do, end
- eliminates white space (tabs, blanks, comments)
- a key issue is speed
  ⇒ use specialized recognizer (lex)
Parser:

- recognize context-free syntax
- guide context-sensitive analysis
- construct \( il(s) \)
- produce meaningful error messages
- attempt error correction

*Parser generators mechanize much of the work*
Back end

\[ il \rightarrow \text{instruction selection} \rightarrow \text{register allocation} \rightarrow \text{machine code} \]

Responsibilities

- translate \( il \) into target machine code
- choose instructions for each \( il \) operation
- decide what to keep in registers at each point
- ensure conformance with system interfaces

*Automation has been less successful here*
Optimizing compilers

Modern optimizers are usually built as a set of passes.

Code Improvement

- analyzes and changes *il*
- goal is to reduce runtime
- must preserve values