Andrew Appel

Textbook is Modern Compiler Implementation in Java (2nd edition) by

Class URL: https://www.cs.umd.edu/class/spring2006/cmsc430/

Office Hours: The class Thu 3:14-4:15pm

Office: A.V. Williams 4135

Email: Chau-Vyen Lien

Important Facts:

Prequisite: CMSC 330

Improvement: scanning, parsing, error recovery, code generation, and code

Topics in the design of programming language translators, including

"Theory of Language Translation"

CMSC 430
Lecture 1, Page 2

All lectures are on the Web, you should still take notes & read textbook

Lecture notes

1-week late policy (20% first day, 10% additional days)

Policies

Compiler optimizations

Java Byte code generation

Simple type checker

scmner/parser using lex & Yacc

Programming projects (tentative)

20% midterm, 30% final exam, 50% programming projects

Grades for:

Course Overview
engineering skills.

Hopefully you will also improve your programming and software engineering skills.

- Software libraries (Java, C++, etc.)
- Program generation tools (Jek, CUP)
- Programming tools (compilers, debuggers)

You'll learn to use

Compilers are large, complex pieces of software. By working on compilers,

better, in some way, than the original.

For “optimizing” compilers, we also expect the program produced to be

the compiler typically lowers the level of abstraction of the program,

executable program in another language

a program that translates an executable program into another language.

What is a compiler?

Compiler Overview
Big step up from assembler – higher level notations

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

Implications:
Back end is NP-Complete

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

- multiple passes better code
- allows multiple front ends
- simply rearranges
- back end maps \( II \) onto target machine
- front end maps legal code into \( II \)
- Intermediate Language (II)

Implications:

![Diagram of two pass compiler]

Traditional two pass compiler
Much of front end construction can be automated

- Shape the code for the back end
- Preliminary storage map
- Produce $I$
- Report errors
- Recognize legal procedure

Responsibilities:

- **Parser**:
  - Reads input
  - Produces tokens
  - Reports errors

- **Scanner**:
  - Analyses input
  - Creates code

Front end
use specialized recognizer

• a key issue is speed

eliminates white space (tabs, blanks, comments)

• typical tokens: number, id, +, -, *, /, do, end

character string for a token is a lexeme

• becomes

  \[ x + y = x \]

maps characters into tokens — the basic unit of syntax

---

Scanner

Errors

Parser

Source

tokens

scanner

scanner
Parser generators mechanize much of the work:

- Attempt error correction
- Produce meaningful error messages
- Construct II(s)
- Guide context-sensitive analysis
- Recognize context-free syntax

Parser:

![Diagram of parser, scanner, tokens, errors, and source code]
Automation has been less successful here.

- Ensure compatibility with system interfaces
- Decide what to keep in registers at each point
- Choose instructions for each operation
- Translate $II$ into target machine code

Responsibilities

Back end
Modern optimizers are usually built as a set of passes.

- Errors must be preserved values
- Goal is to reduce runtime
- Analyzes and changes

Code Improvement

Optimizing Compilers