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

---

**Abstract view of compiler**

```
source code  ─── compiler ─── machine code
       \                    \errors
```

### 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 ⇒ better code

*Front end is O(n) or O(n log n)*
*Back end is NP-Complete*

**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

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

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

\[ \text{source code} \rightarrow \text{front end} \rightarrow \text{middle end} \rightarrow \text{back end} \rightarrow \text{machine code} \]

Code Improvement

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

\[ \text{il} \rightarrow \text{opt}_1 \rightarrow \ldots \rightarrow \text{opt}_n \rightarrow \text{il} \]

*Modern optimizers are usually built as a set of passes.*