

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](mailto:tseng@cs.umd.edu)  
*Office:* A.V. Williams 4135  
*Office Hours:* Tue & Thu 3:14–4:15pm

*Class URL:* <http://www.cs.umd.edu/class/spring2006/cmsc430/>

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

## Course Overview

---

### Basis for grades:

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

### Programming Projects (tentative)

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

### Policies

- no collaboration (code sharing) allowed
- 1-week late policy (20% 1st day, 10% additional days)

### 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

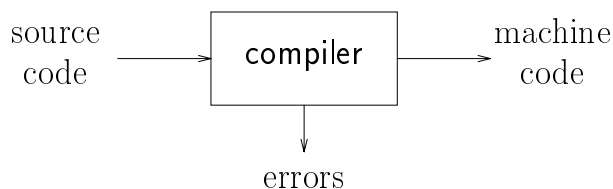
Compilers are large, complex pieces of software. By working on compilers, you’ll learn to use

- 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

---



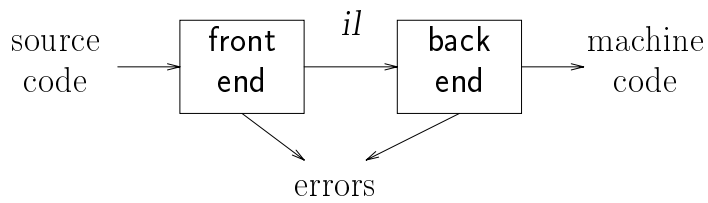
#### Implications:

- 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:

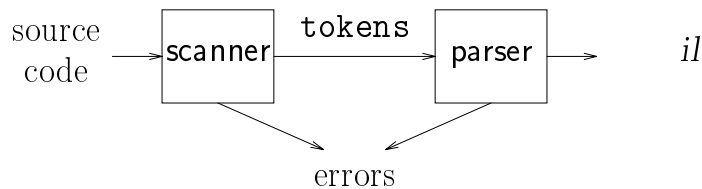
- intermediate language (*il*)
- front end maps legal code into *il*
- back end maps *il* onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes  $\Rightarrow$  better code

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

*Back end is NP-Complete*

## Front end

---



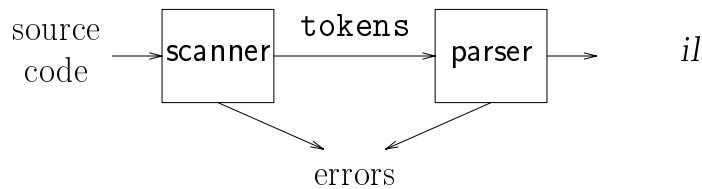
### Responsibilities:

- 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

---

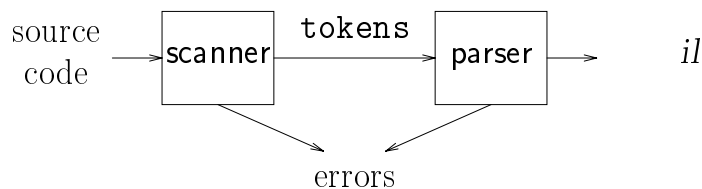


### Scanner

- maps characters into *tokens* – the basic unit of syntax  
 $x = x + y;$   
becomes  
 $\langle \text{id}, x \rangle = \langle \text{id}, x \rangle + \langle \text{id}, y \rangle ;$
- 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

---



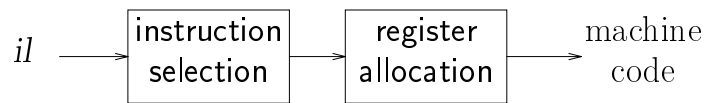
### Parser:

- 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

---



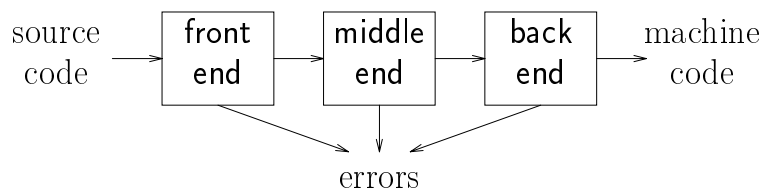
### Responsibilities

- 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*

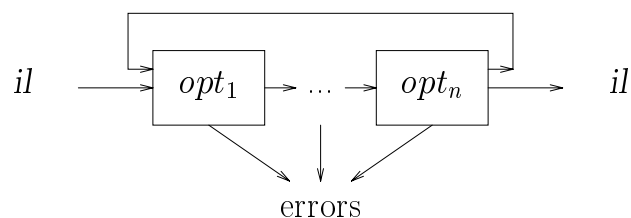
## Optimizing compilers

---



### Code Improvement

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



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