Introduction to Compiler Design

Lesson 1: Course Presentation

Course Description

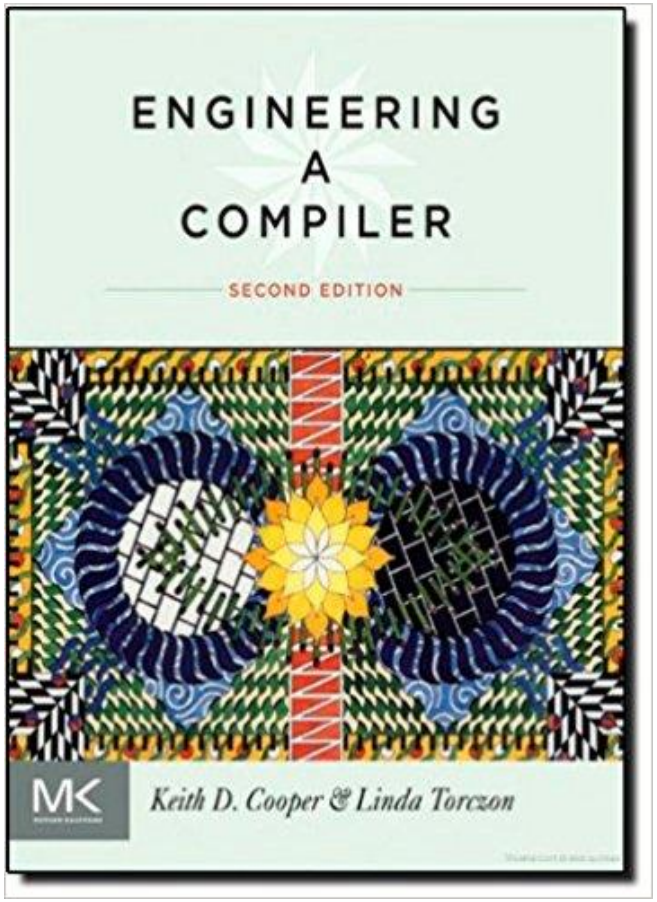
Intensive course about theory and implementation of compiler for high-level programming languages. You will design and implement parts of a compiler for a small high level language.

Prerequisites: Formal Languages and Automata, Algorithms and Data Structures, Java Programming, Computer Architecture

Textbook

Engineering a compiler
Second Edition
Keith D. Cooper
& Linda Torczon

A PDF copy is available on the web page of the course



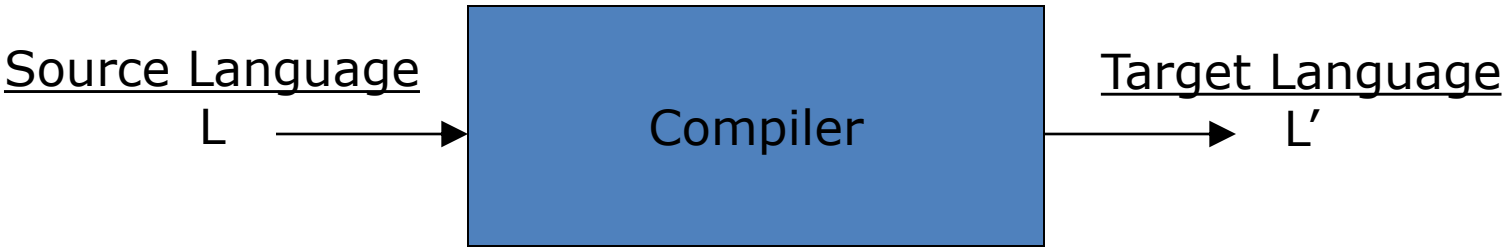
Grading

- Homeworks: 20%
  - written assignments
- Programming labs: 30%
  - 6 labs for building a simple compiler
- Final Exam: 30%
  - Paper based, closed everything
- Project: 20%
  - Completion of the programming labs

Tools

- OS
  - **Linux** preferred but possible with other OS
- Programming language
  - **Java** 1.8, plus **JLex** and **Java CUP**
- MIPS emulator
  - **Spim**, **QtSpim**
- Misc
  - A good editor (**emacs** preferred)

What is a Compiler?

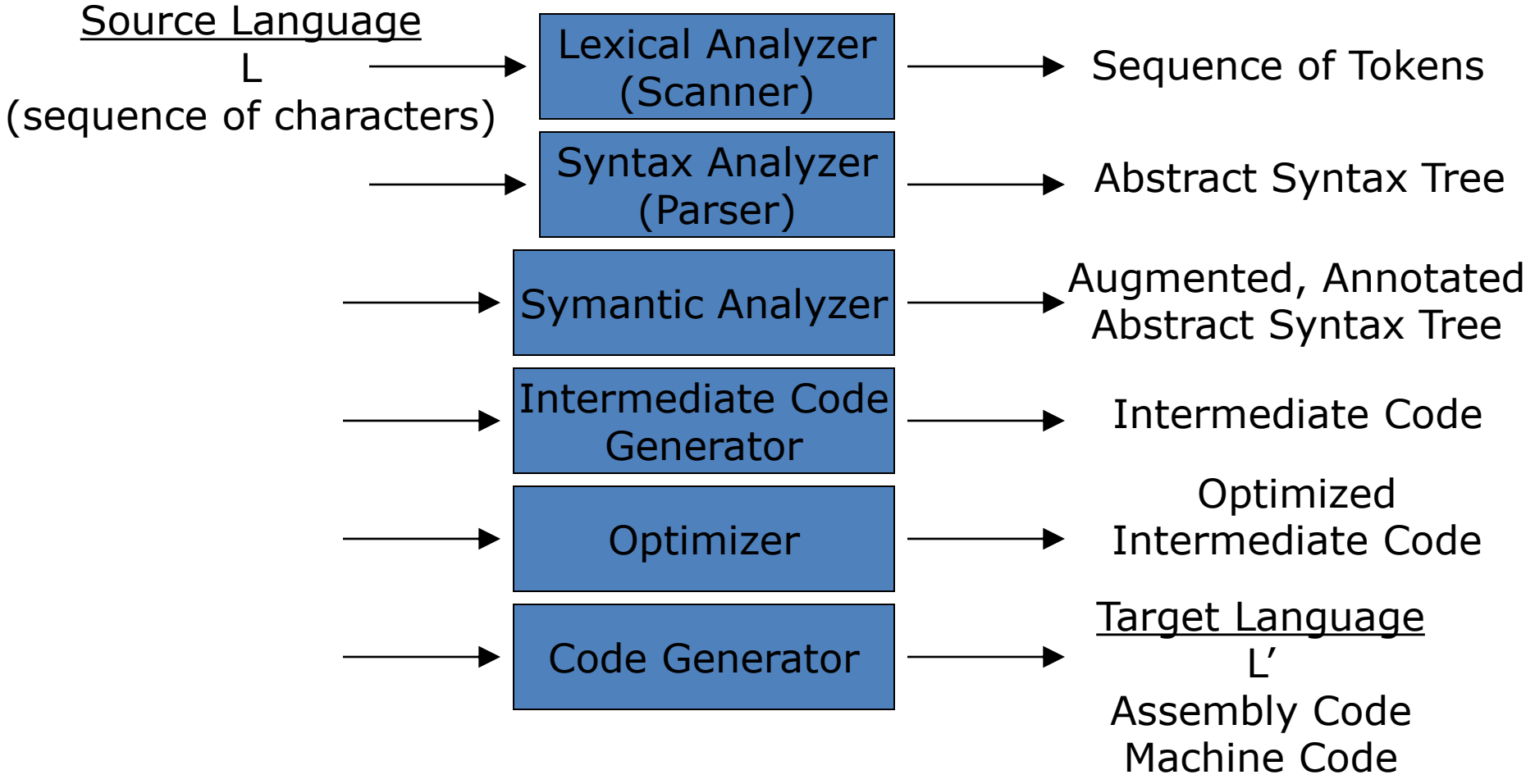


Java
C++
FORTRAN

JVM
Intel Pentium
MIPS
IBM's CELL

A compiler translates text from a source language, L, to
A target language, L'.

What is a Compiler?



The Scanner

- Reads characters from the source program.
- Groups characters into **lexemes**

  sequences of characters that "go together"
- Lexemes corresponds to a **tokens**; scanner returns next token (plus maybe some additional information) to the parser.
- Scanner also discovers lexical errors (e.g., erroneous characters such as # in java).

Example Lexemes and Tokens

Lexeme:	;	=	index	tmp	21	64.32
Token:	SEMI-COLON	ASSIGN	IDENT	IDENT	INT-LIT	INT-FLOAT

Source Code:

position = initial + rate * 60 ;

Corresponding Tokens:

IDENT ASSIGN IDENT PLUS IDENT TIMES INT-LIT SEMI-COLON

The Parser

- Groups tokens into "grammatical phrases", discovering the underlying structure of the source program.

- Finds syntax errors.

  For example, in Java the source code

  position = * 5 ;

  corresponds to the sequence of tokens:

  IDENT ASSIGN TIMES INT-LIT SEMI-COLON

  All are legal tokens, but that sequence of tokens is erroneous.

- Might find some "static semantic" errors, e.g., a use of an undeclared variable, or variables that are multiply declared.

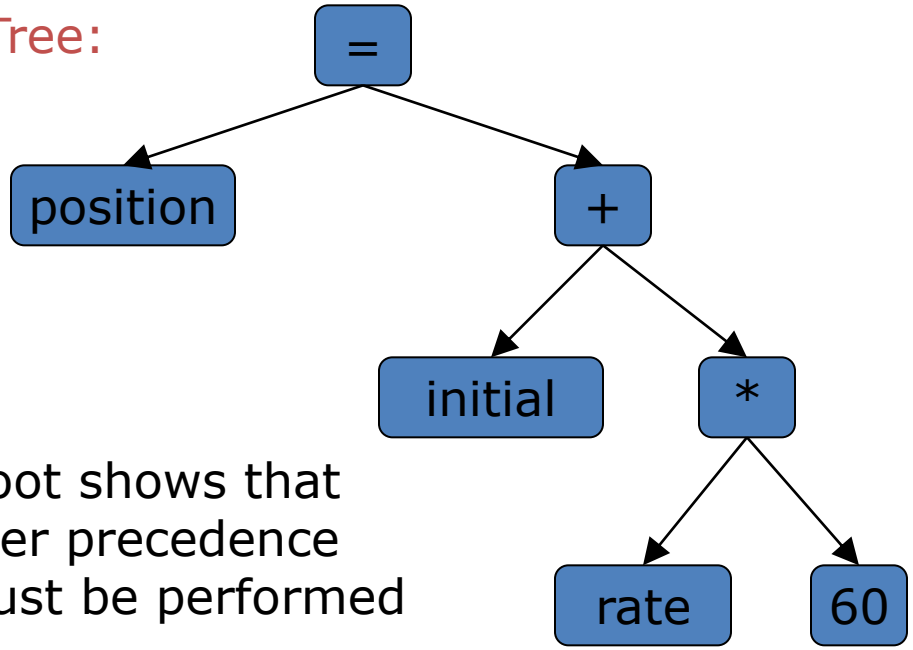
- Might generate code, or build some intermediate representation of the program such as an abstract-syntax tree.

Example of parsing

Source Code:

position = initial + rate * 60 ;

Abstract-Syntax Tree:



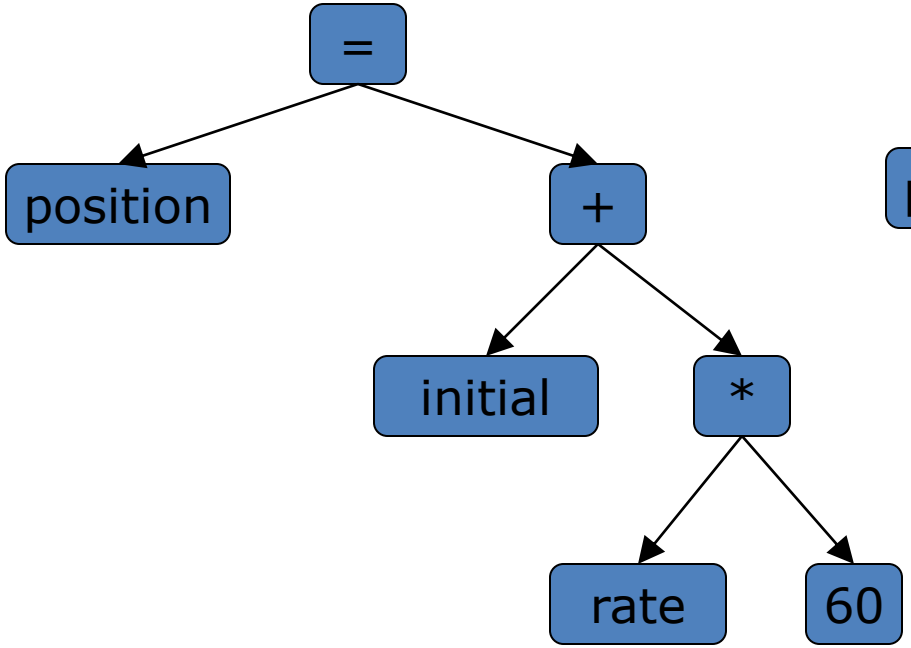
- Internal nodes are **operators**.
- A node's children are **operands**.
- Each subtree forms "logical unit"
  e.g., the subtree with * at its root shows that because multiplication has higher precedence than addition, this operation must be performed as a unit (*not* initial+rate).

Semantic Analyzis

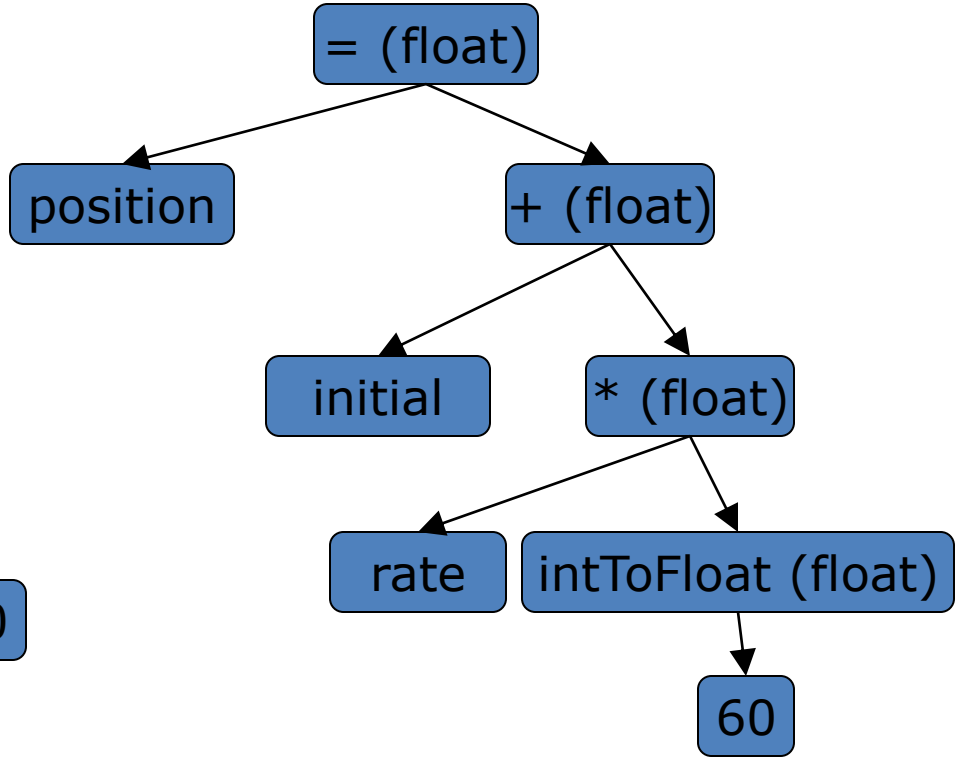
- Checks for (more) "static semantic" errors
- Annotate and/or change the abstract syntax tree

Example of Semantic Analysis

Abstract-Syntax Tree:



Annotated Abstract-Syntax Tree:



Intermediate Code Generator

- Translates from abstract-syntax tree to intermediate code
- One possibility is 3-address code

  each instruction involves at most 3 operands

  Example:

  temp1 = inttofloat(60)

  temp2 = rate * temp1

  temp3 = initial + temp2

  position = temp3

Optimizer

- Tries to improve code to
  - Run faster
  - Be smaller
  - Consume less energy

Try Optimizing This code (for speed)

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i=0; i<=N; i++) {
        x=x+(4*a/b)*i+(i+1)*(i+1);
        x=x+b*y;
    }
    return x;
}
```

Some Types of Optimization

- Constant Propagation
- Algebraic Simplification
- Copy Propagation
- Common Sub-expression Elimination
- Dead Code Elimination
- Loop Invariant Removal
- Strength Reduction

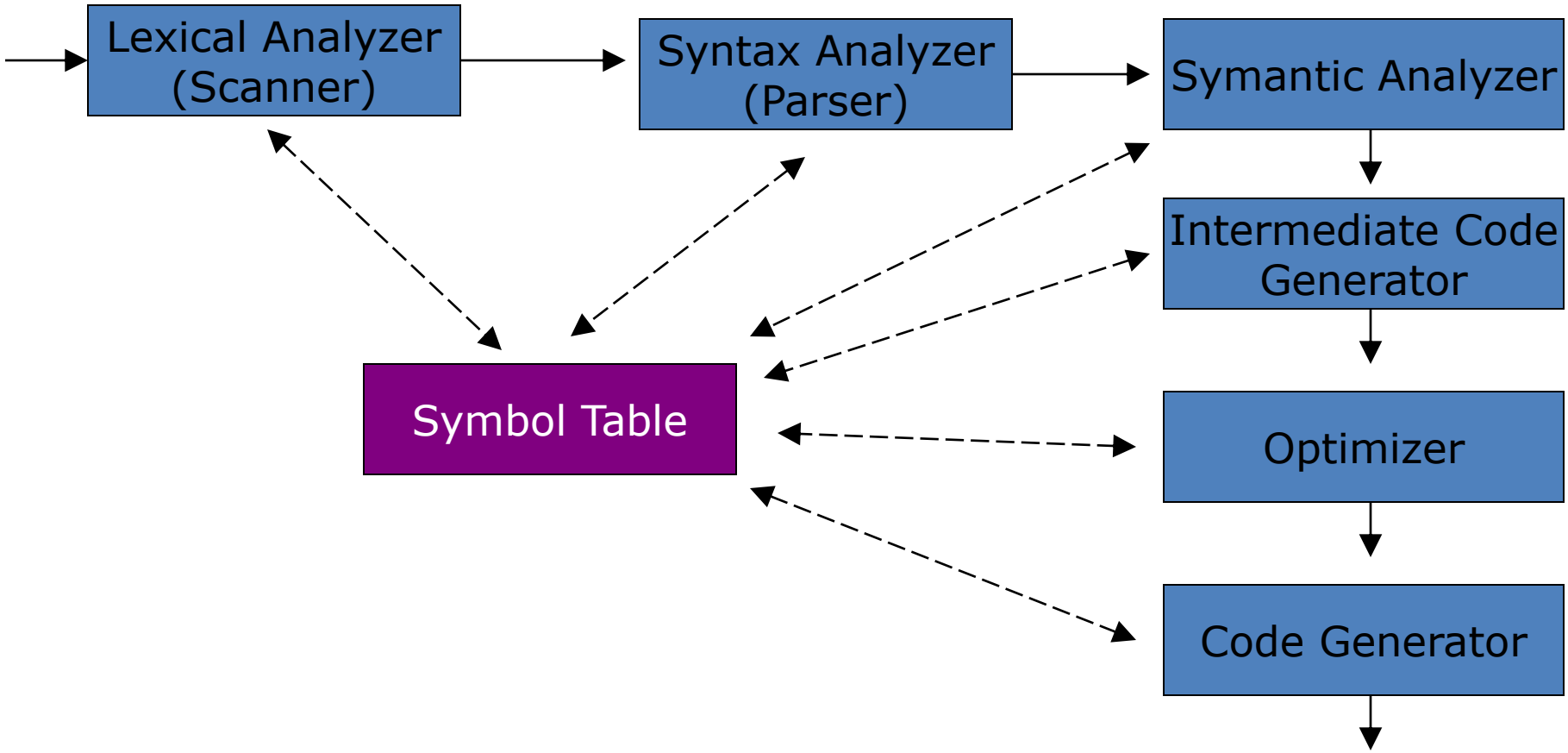
Code Generator

Generate object code from (optimized) intermediate code

Example of MIPS assembly code:

```
   .data
var1:      .word 23
   .text
main:
           lw $t0, var1
           li $t1, 5
           sw $t1, var1
           done
```

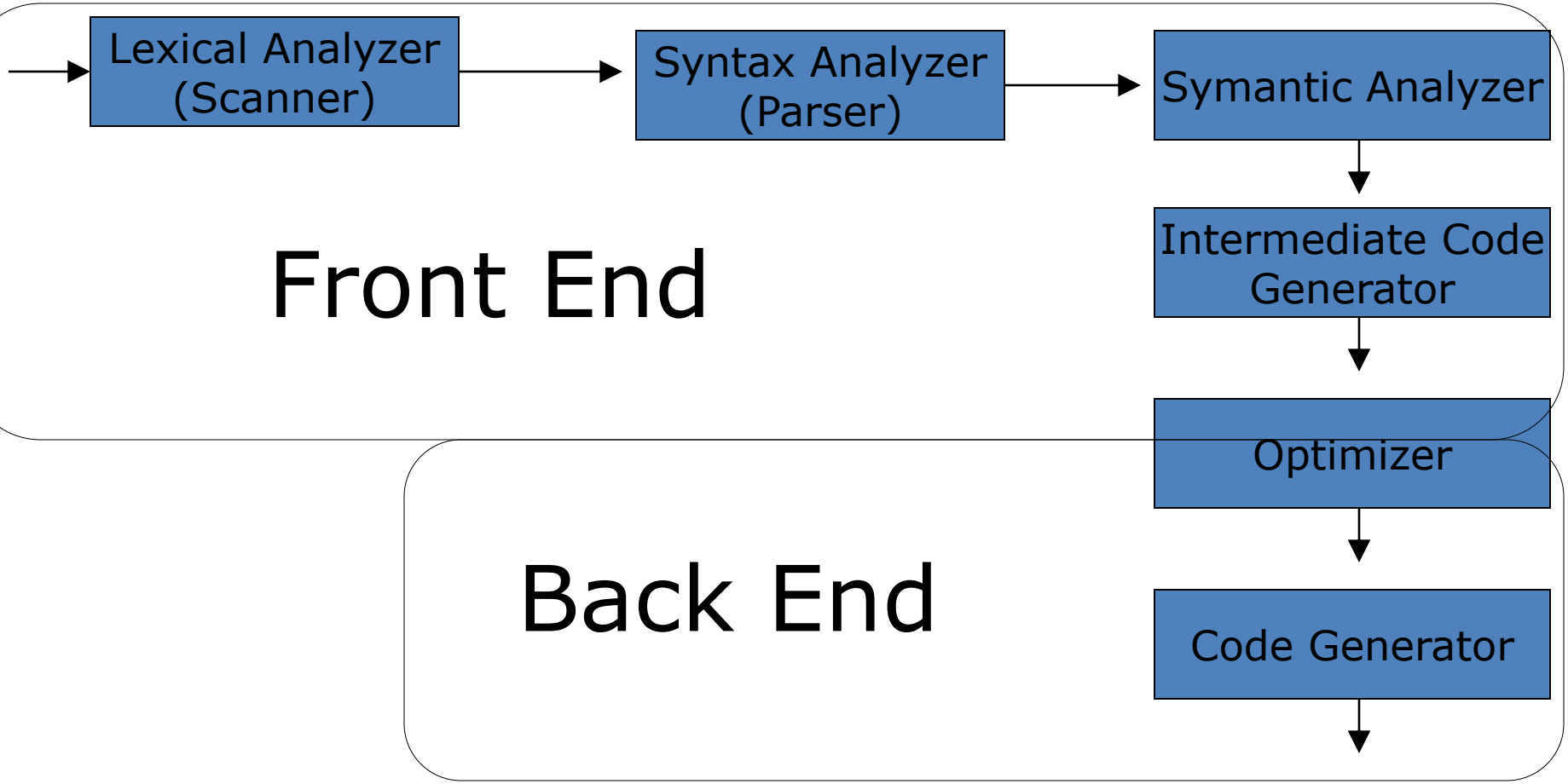
Symbol Tables



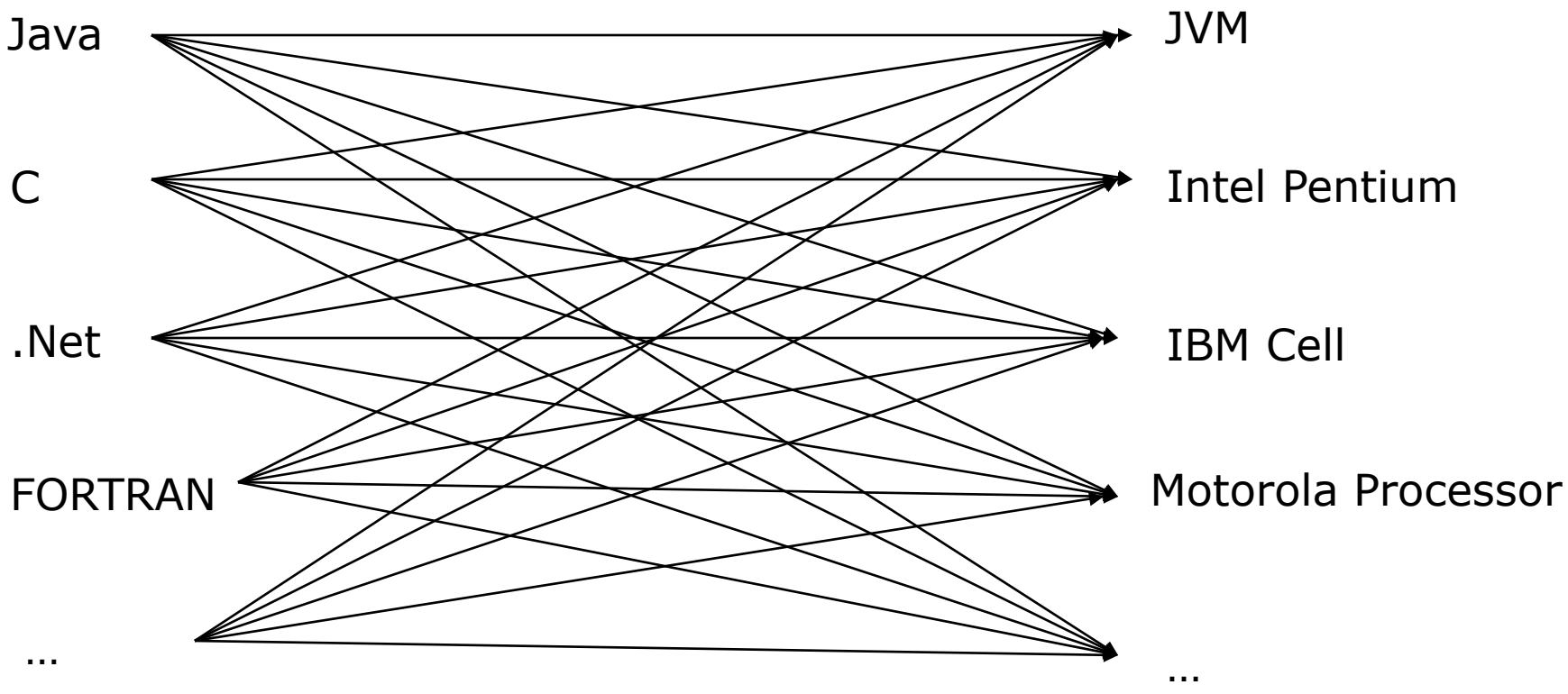
Symbol Tables

- Keep track of names declared in the program
- Separate level for each scope
- Used to analyze static symantics:
  - Variables should not be declared more than once in a scope
  - Variables should not be used before being declared
  - Parameter types for methods should match method declaration

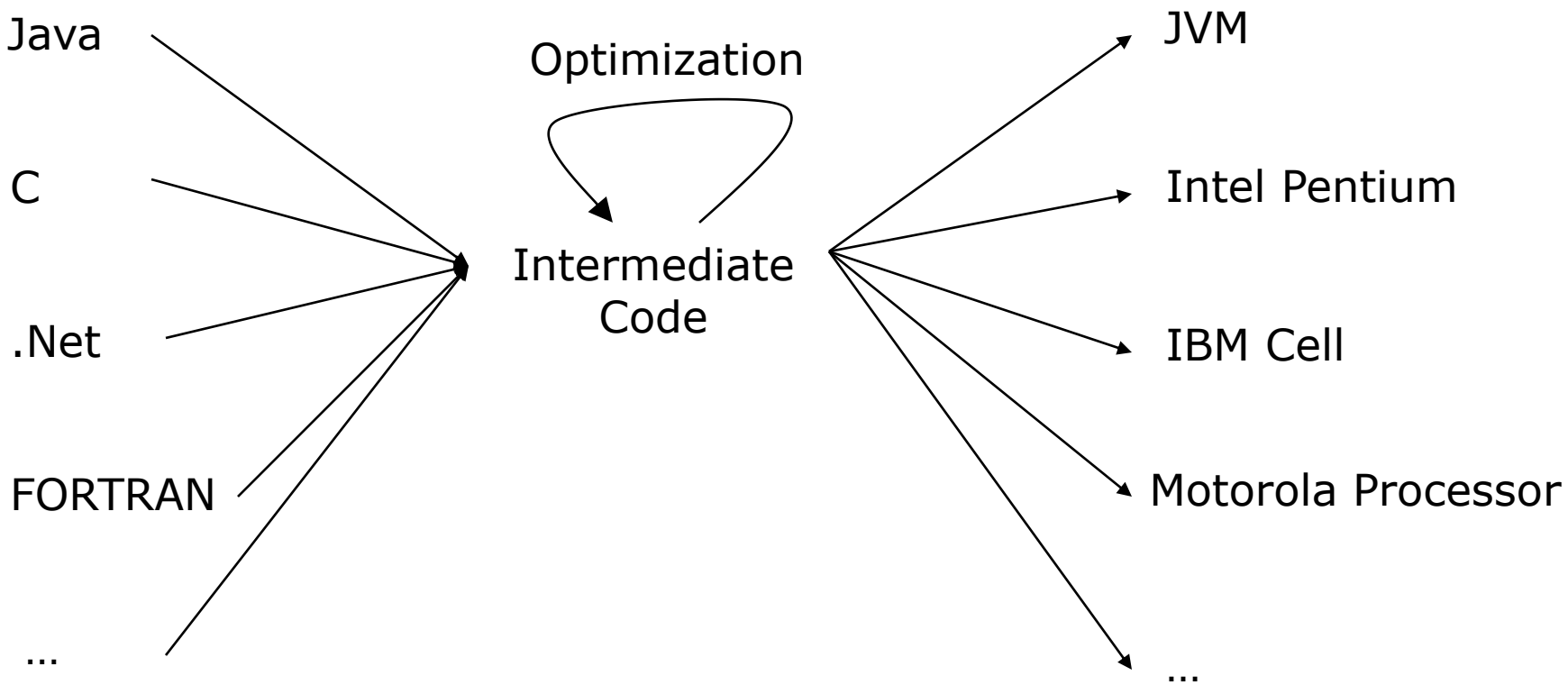
Compiler Modularity



Many Compilers



Many Compilers



Summary

- Compilers Translate Source Language to Target Language
- Compilers have several steps
  - Scanner
  - Parser
  - Semantic Analyzer
  - Intermediate Code Generator
  - Optimizer
  - Code Generator
- Symbol Table Used To Keep Track of Names Used in Program
- Front End and Back End Simplify Compiler Design
  - Introduction of new languages
  - Introduction of new hardware