Compiler Principles

Overview

Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
School of Computer Science E301

Basic Information

Type： Compulsory
Credit： 3
Hours： 54
Prerequisites： Programming Languages、Discrete Mathematics、Data Structure、Principles of Computer Organization、etc.

Teacher

- Xiaoyuan Xie， Professor, School of Computer Science E301
- xxie@whu.edu.cn

Objectives

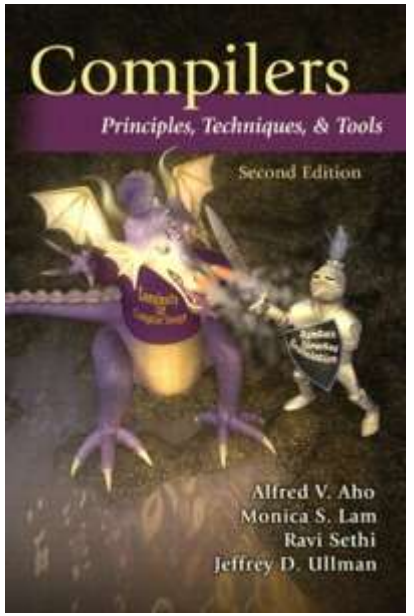
- Most important and fundamental course to CS students
- Master theoretical principles of compilers
  - Chomsky language hierarchy, Type III, Type II, etc.
  - Lexical analysis, top-down&bottom-up parsing, etc.
  - Syntax-directed Translation
  - Etc.
- Be able to build a compiler for a (simplified) language
  - Know how to use compiler construction tools, such as generators of scanners and parsers

Syllabus

Content	Hours
1. Introduction	4
2. Lexical analysis	6
3. Context-free grammar	4
4. Top-down parsing	4
5. Bottom-up parsing	8
6. Syntax-directed translation	8
7. Intermediate representation	4
8. Intermediate code generation	6
9. Runtime environment	6
10. Code generation and optimization	4

Materials

- Textbook:
  - Compilers: Principles, Techniques, and Tools (2nd Edition), Alfred V·Aho， Monica S·Lam，Ravi Sethi，Jeffrey D·Ullman, Addison Wesley; 2nd edition (September 10, 2006)
- Other references:
  - Coursera Course – Compiler, http://www. Coursera.org
  - Stanford Course CS143 by Keith Schwarz, http://cs143.stanford.edu
  - Parsing Techniques - A Practical Guide (Second Edition), Dick Grune and Ceriel J.H. Jacobs, Springer



Rules and Regulations

- **Attendance: check every week (begin, end)**
- **Homework: every week**
  - Submission due: next Monday 12:00 (pm);
  - No later than next Friday 17:00 (pm) --- with a punishment
- **Assignments: 1-2 assignments**
- **Score:**
  - Final exam 60% + Homework & assignment & attendance 40%

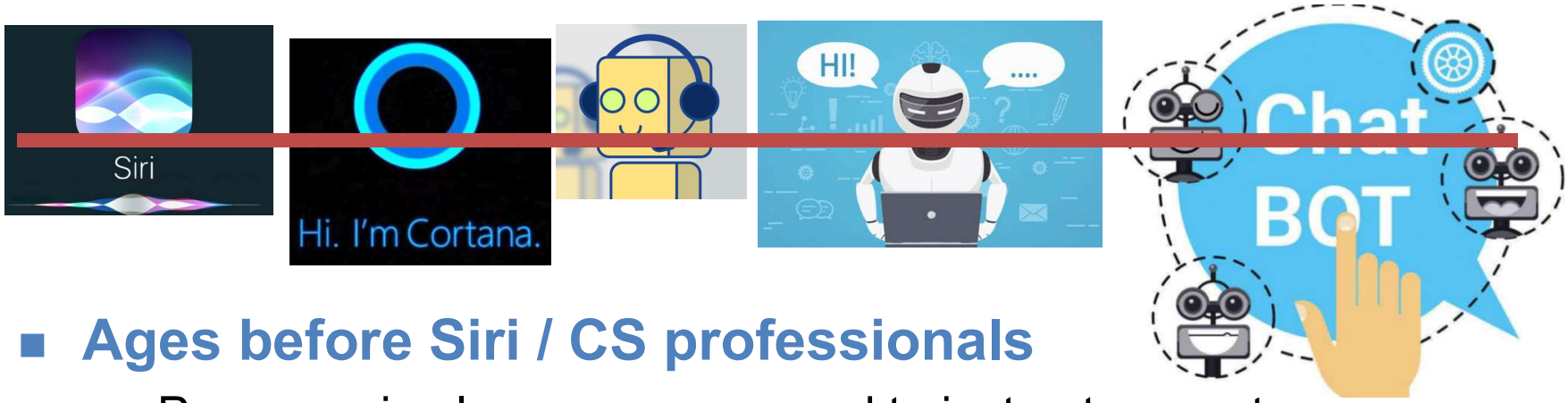
Lecture 1: Introduction

Compiler in a Nutshell

Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
School of Computer Science E301

Motivation question

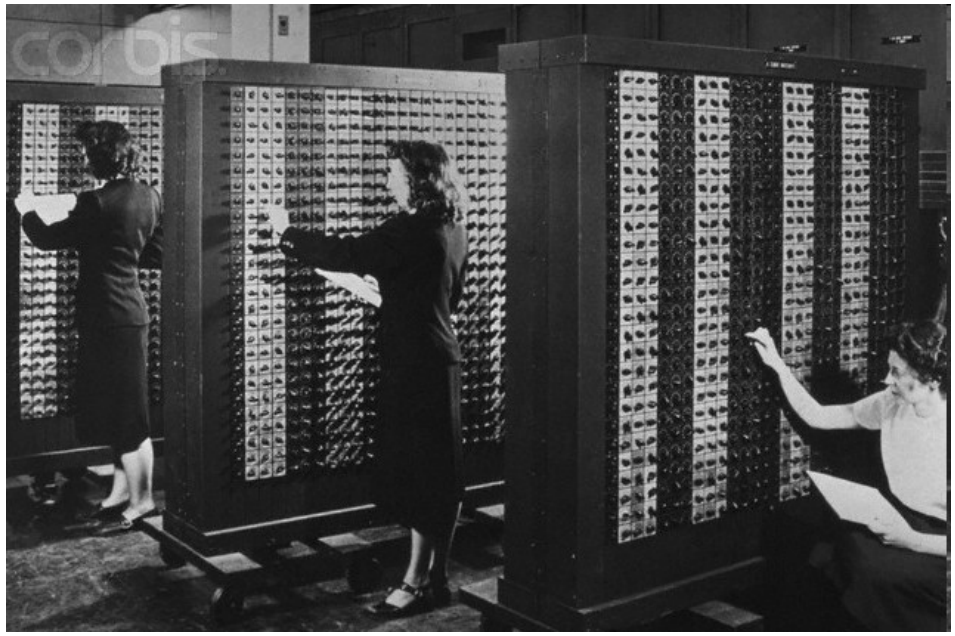
- **How do we communicate with computer?**

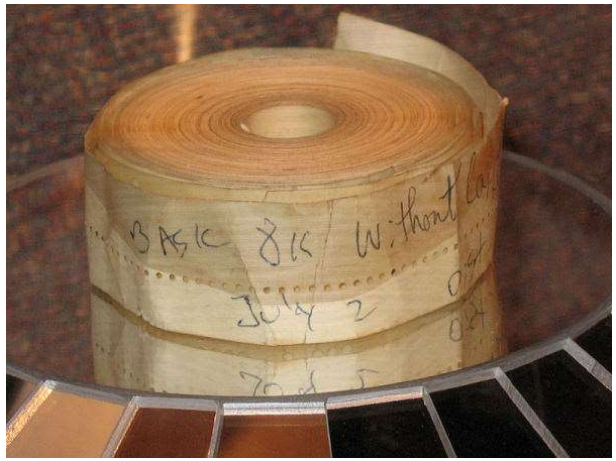


- **Ages before Siri / CS professionals**
  - Programming languages are used to instruct computers

How to Instruct a Computer?

- **Long long ago…**





How to Instruct a Computer?

- **Problem:**
  - Not readable by human
  - Programmer Productivity --- Programming expensive; 50% of costs for machines went into programming
  - Expensive to modify (e.g. insert an instruction)
  - Extremely difficult to debug
  - Programming bit-by-bit doesn't scale
- **Solution:**
  - Instruct the computer at a higher level of abstraction --- a higher-level programming language

How to Instruct a Computer?

- **Example**

President

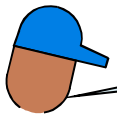
General

Sergeant

Foot Soldier

My poll ratings are low, lets invade a small nation

Cross the river and take defensive positions



Forward march, turn left Stop!, Shoot

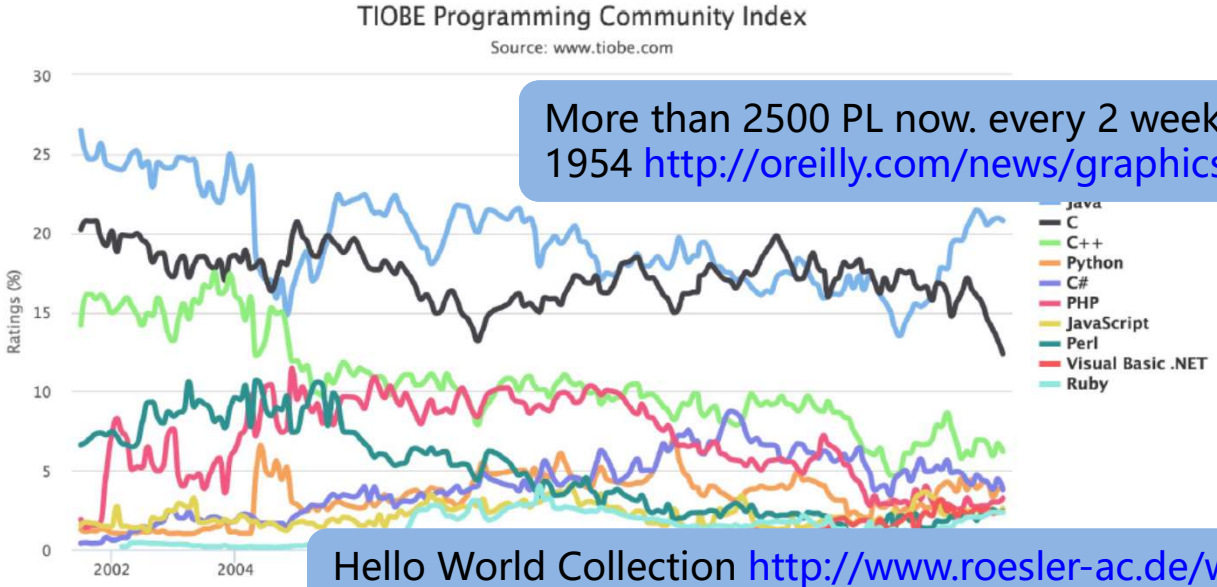
High-level languages

- **High-level Candidature --- Natural language**
  - Powerful in abstraction，but ambiguous
  - Same expression describes many possible actions
- **Programming languages**
  - high abstraction
  - precision (avoid ambiguity)
  - conciseness
  - expressiveness
  - modularity

Programming Languages

How many

- **Statistics**



More than 2500 PL now. every 2 weeks, a new PL is born since 1954 http://oreilly.com/news/graphics/prog_lang_poster.pdf.

Hello World Collection http://www.roesler-ac.de/wolfram/hello.htm
收集了428 个不同语言写的``Hello World'' 程序.

What is a Programming Languages

- **A programming language is a set of rules that provides a way of telling a computer what operations to perform.**
- **It provides a linguistic framework for describing computations**

What is a Programming Language

- **English is a natural language. It has words, symbols and grammatical rules.**
- **A programming language also has words, symbols and rules of grammar.**
  - The grammatical rules are called syntax.
  - Each programming language has a different set of syntax rules.

Levels of Programming Languages

High-level program

```
class Triangle {
  ...
  float surface()
    return b*h/2;
  }
```

Low-level program

```
LOAD r1,b
LOAD r2,h
MUL r1,r2
DIV r1,#2
RET
```

Executable Machine code

```
0001001001000101001001001110110010101101001...
```

Generations

- First Generation Languages
- Second Generation Languages
- Third Generation Languages
- Fourth Generation Languages
- Fifth Generation Languages

First Generation Languages

- **Machine language**
  - Operation code – such as addition or subtraction.
  - Operands – that identify the data to be processed.
  - Machine language is machine dependent as it is the only language the computer can understand.
  - Very efficient code but very difficult to write.

Second Generation Languages

- **Assembly languages**
  - Symbolic operation codes replaced binary operation codes.
  - Assembly language programs needed to be “assembled” for execution by the computer. Assembly language instructions will translated into machine language instructions.
  - Very efficient code and easier to write.

Third Generation Languages

- **Closer to English but included simple mathematical notation.**
  - Programs are written in source code, and must be translated into machine language programs (called object code).
  - The translation of source code to object code is accomplished by a machine language system program called a compiler.
  - FORTRAN、COBOL、C and C++、Visual Basic

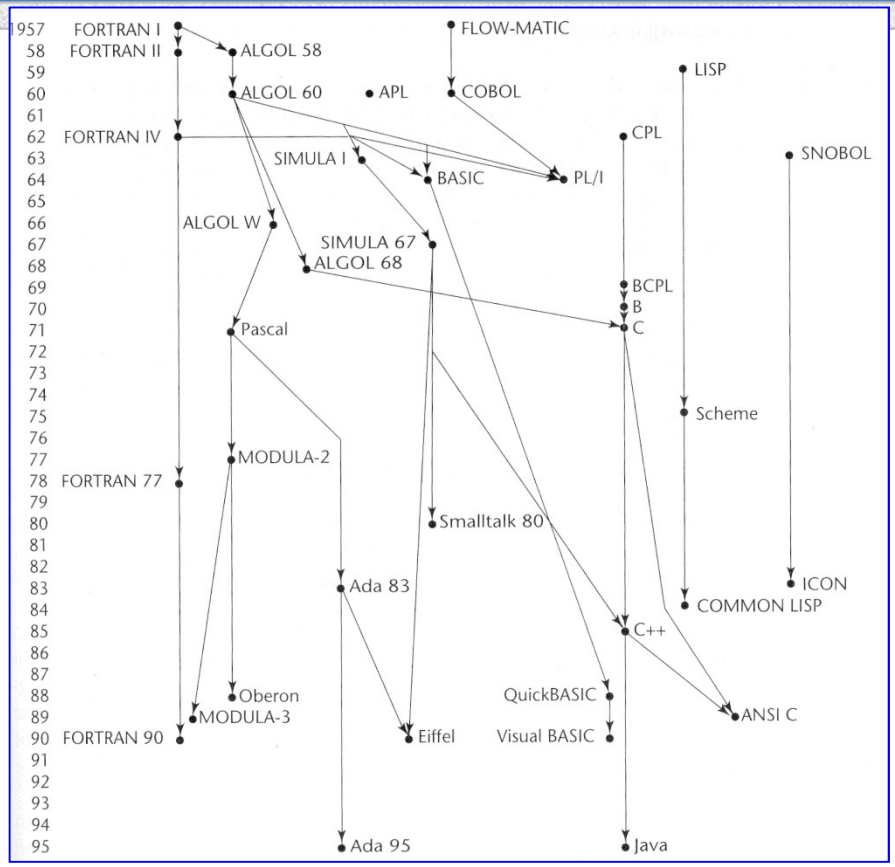
Fourth Generation Languages

- **A high level language that requires fewer instructions to accomplish a task than a third generation language**
- **Used with databases**
  - Query languages
  - Report generators
  - Forms designers
  - Application generators

Fifth Generation Languages

- **Declarative languages**
- **Functional: Lisp, Scheme, SML**
  - Also called applicative
  - Everything is a function
- **Logic: Prolog**
  - Based on mathematical logic
  - Rule- or Constraint-based

Language Family Tree



The principal paradigms

- **Imperative Programming --- compiler**
  - Procedure: Fortran, ALGOL, COBOL and BASIC, Pascal and C
  - Object-Oriented Programming (C++, Java)
- **Declarative Programming**
  - Logic: Prolog
  - Functional/Applicative Programming (Lisp)

Programming Languages

- **Procedure programming languages**
  - Sequences of instructions
  - First, second and some third generation languages
- **Object-oriented programming languages**
  - Objects are created rather than sequences of instructions
  - Some third generation, and fourth and fifth generation languages

Traditional Programming Languages

- **FORTRAN**
  - FORmula TRANslation.
  - Developed at IBM in the mid-1950s.
  - Designed for scientific and mathematical applications by scientists and engineers.

Traditional Programming Languages (cont'd.)

- **COBOL**
  - COmmon Business Oriented Language.
  - Developed in 1959.
  - Designed to be common to many different computers.
  - Typically used for business applications.

Traditional Programming Languages (cont'd.)

- **BASIC**
  - Beginner's All-purpose Symbolic Instruction Code.
  - Developed at Dartmouth College in mid 1960s.
  - Developed as a simple language for students to write programs with which they could interact through terminals.

Traditional Programming Languages (cont'd.)

- **C**
  - Developed by Bell Laboratories in the early 1970s.
  - Provides control and efficiency of assembly language while having third generation language features.
  - Often used for system programs.
  - UNIX is written in C.

Object-Oriented Programming Languages

- **Simula**
  - First object-oriented language
  - Developed by Ole Johan Dahl in the 1960s.
- **Smalltalk**
  - First purely object-oriented language.
  - Developed by Xerox in mid-1970s.
  - Still in use on some computers.

Object-Oriented Programming Languages (cont'd.)

- **C++**
  - It is C language with additional features.
  - Widely used for developing system and application software.
  - Graphical user interfaces can be developed easily with visual programming tools.

Object-Oriented Programming Languages (cont'd.)

- **JAVA**
  - An object-oriented language similar to C++ that eliminates lots of C++'s problematic features
  - Allows a web page developer to create programs for applications, called applets that can be used through a browser.
  - Objective of JAVA developers is that it be machine, platform and operating system independent.

Special Programming Languages

- **Scripting Languages**
  - JavaScript and VBScript
  - Php and ASP
  - Perl and Python
- **Command Languages**
  - sh, csh, bash
- **Text processing Languages**
  - LaTex, PostScript

Special Programming Languages

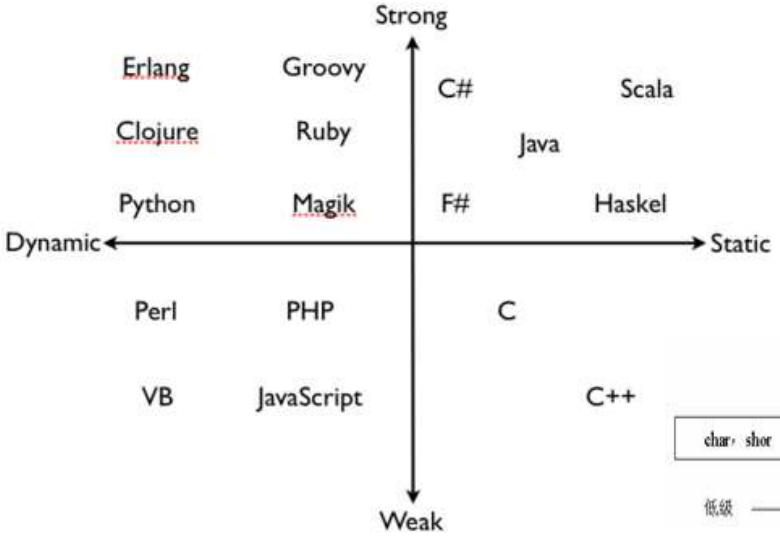
- **HTML**
  - HyperText Markup Language.
  - Used on the Internet and the World Wide Web (WWW).
  - Web page developer puts brief codes called tags in the page to indicate how the page should be formatted.

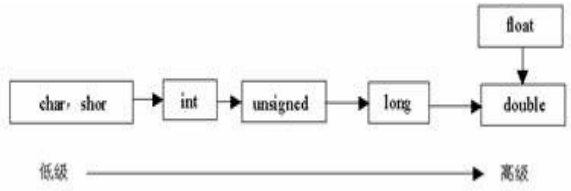
Special Programming Languages (cont'd.)

- **XML**
  - Extensible Markup Language.
  - A language for defining other languages.

(Variable) Type system

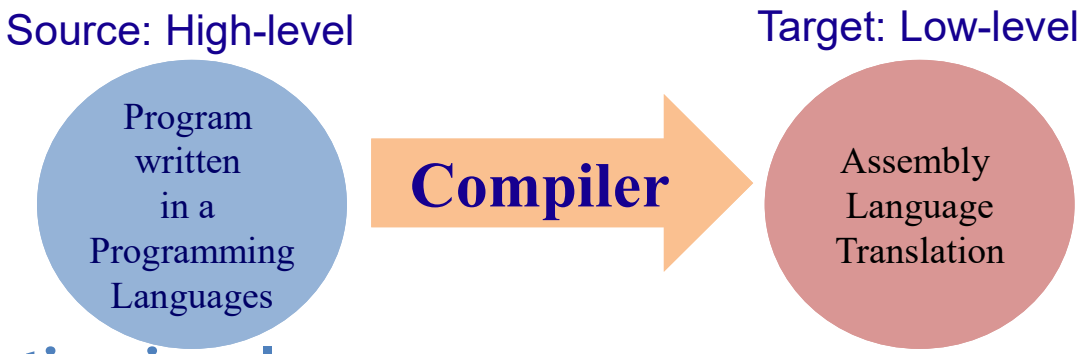
- **Dynamically typed language**
  - Types of ***val***
  - A poorly type … halt or otherwise **sig**
- **Statically typ**
  - Types of vari … **ted**, and a program might be reje
  - Type errors a
- **Strong typed**
  - There are **no I**
- **Weak typed allows Implicit type conversion**
  - Type system can be subverted (invalidating any guarantees)





Overview of Compiler

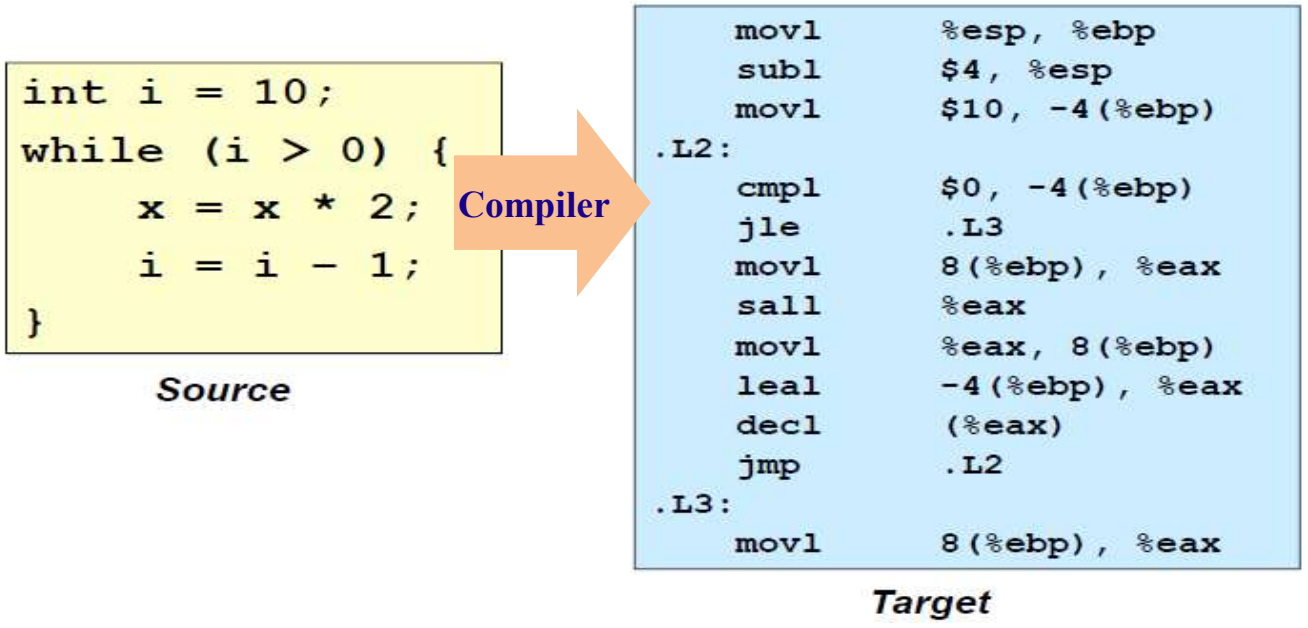
Compiler's major task



- **Translation involves:**
  - Read and understand the program
  - Precisely determine what actions it require
  - Figure-out how to faithfully carry-out those actions
  - Instruct the computer to carry out those actions

Compiler's major task

- **Example**



Compiler's major task

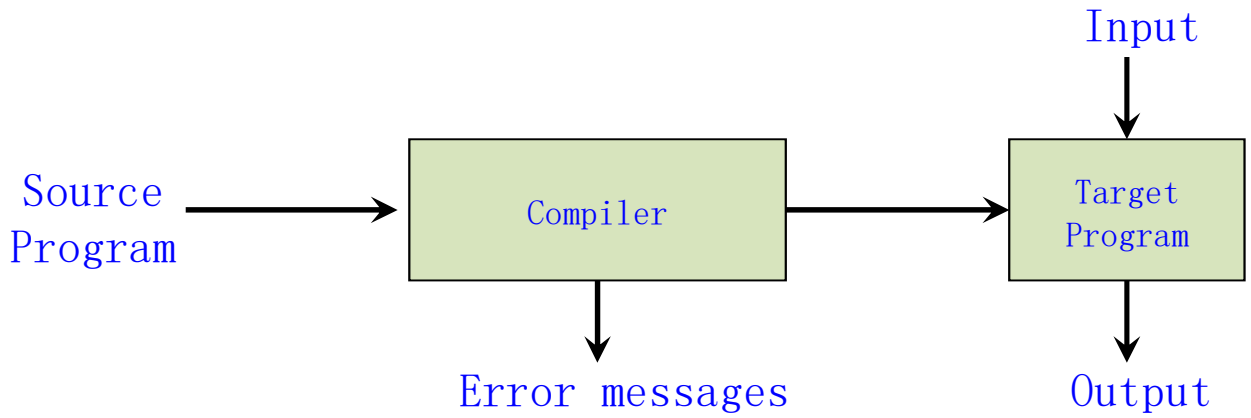
- **Input: Standard imperative language (C, C++)**
  - State
    - Variables,
    - Structures,
    - Arrays
  - Computation
    - Expressions (arithmetic, logical, etc.)
    - Assignment statements
    - Control flow (conditionals, loops)
    - Procedures

Compiler's major task

- **Output: target program**
  - State
    - Registers
    - Memory with Flat Address Space
  - Machine code – load/store architecture
    - Load, store instructions
    - Arithmetic, logical operations on registers
    - Branch instructions

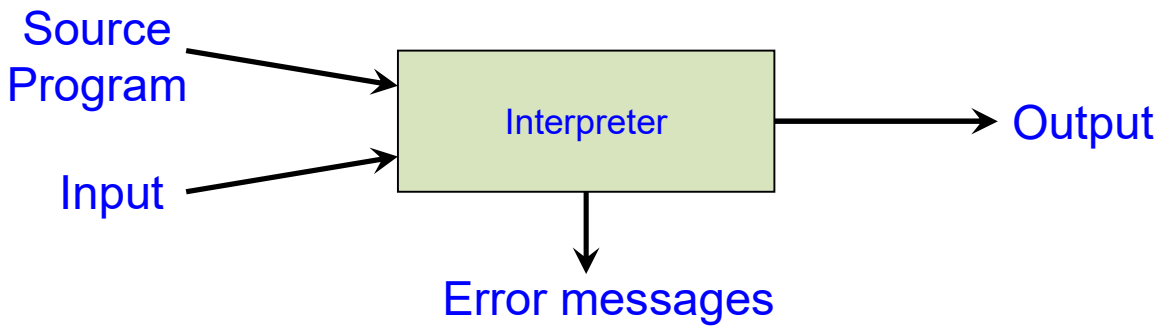
Compiler v.s. Interpreter

- *"Compilation"*
  - Translation of a program written in a source language into a semantically equivalent program written in a target language



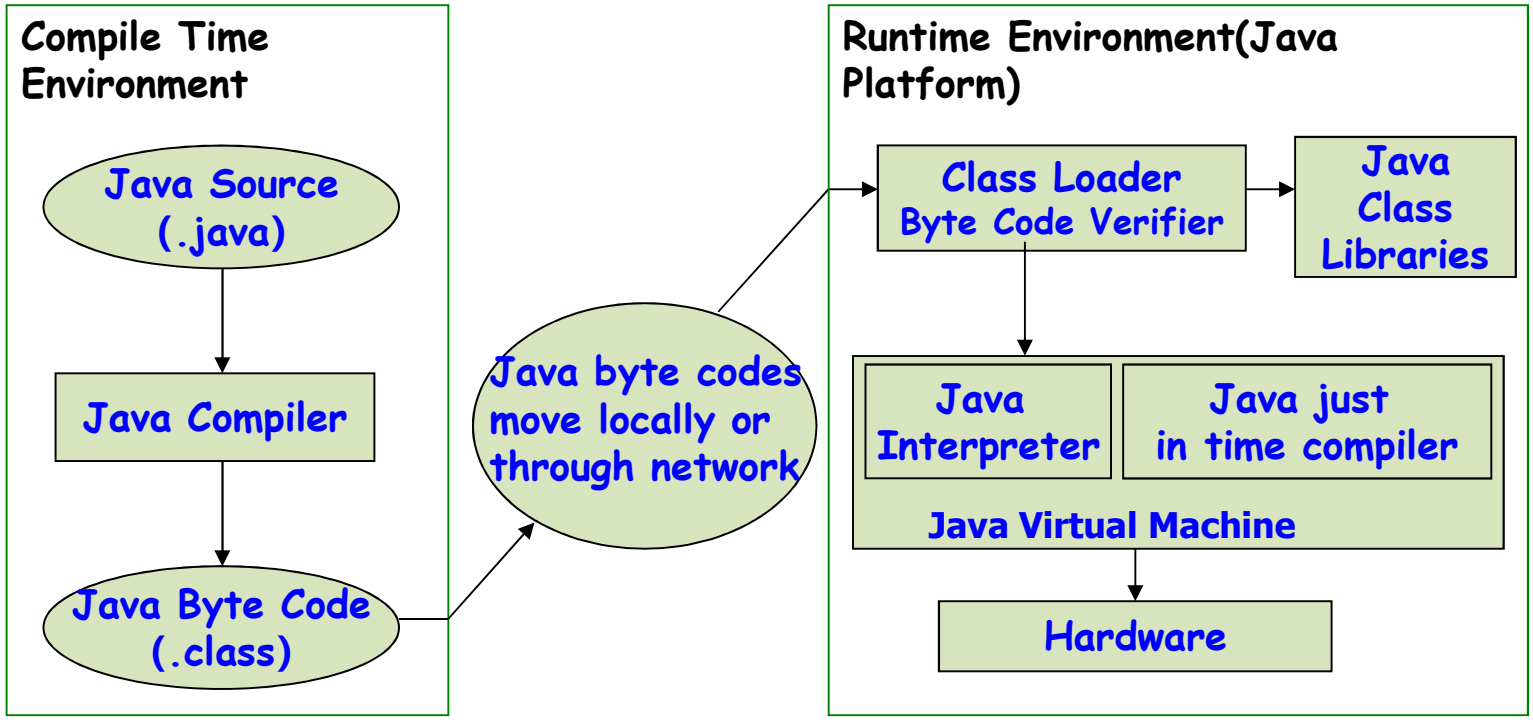
Compiler v.s. Interpreter

- "*Interpretation*"
  - Performing the operations implied by the source program



Compiler v.s. Interpreter

- Java

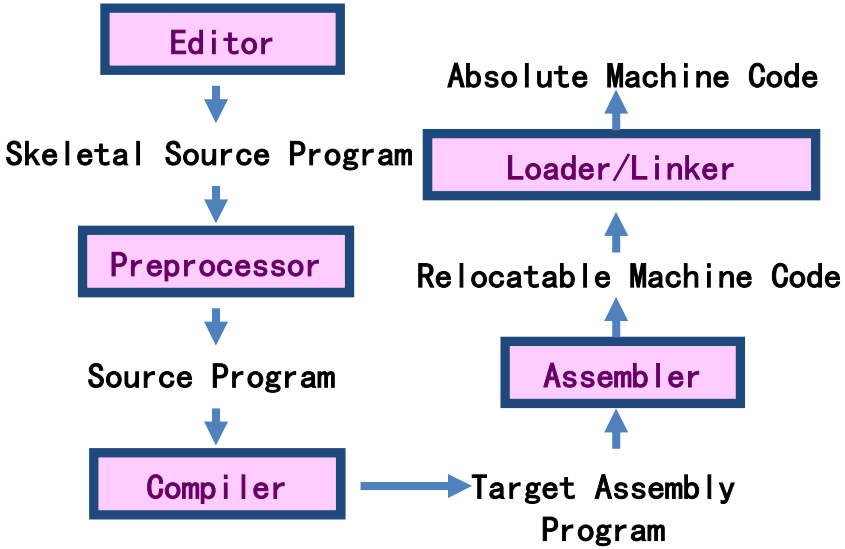


A complete compiler

Accompanies of a compiler

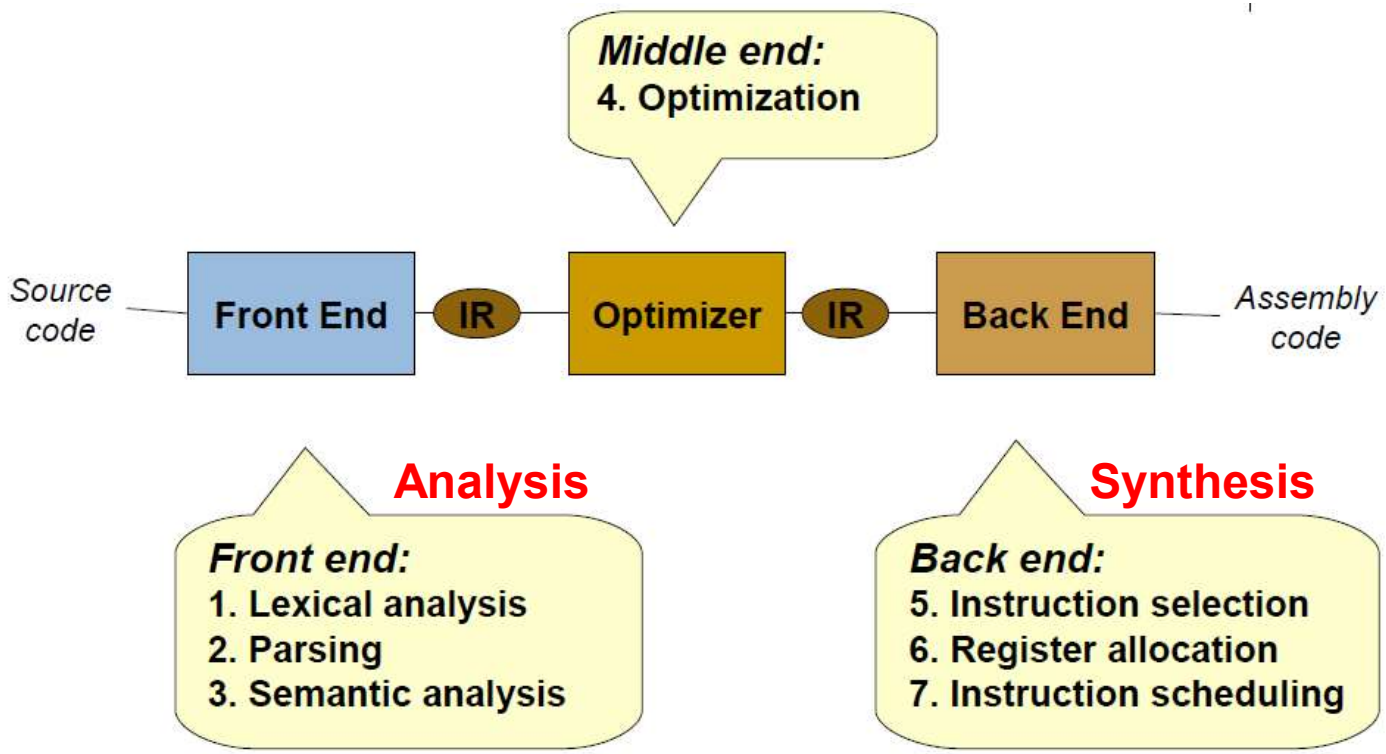
- Editor is the environment where you can type-in your source code, sometimes with highlights, automatic matching, or intelligent completion.
- Preprocessor provides the first pass of compilation. It processes include-files, conditional compilation instructions and macros, delete comments, etc.
- Assembler performs the third stage of compilation. It takes the assembly source code and produces an assembly listing with offsets. The assembler output is stored in an object file.
- Linker performs the final stage of compilation. It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file.
- Loader loads the executable code into the memory.

Overall process



Construction of Compiler

Construction



Lexical analysis

Lexical analysis / scanning

- The lexical analyzer reads the stream of characters making up the source program and separates it into meaningful sequences called lexemes
- Based on the lexemes, we construct tokens:
  - A token: < token- name, attribute-value >
  - Tokens have value and type: < keyword, if>, <identifier, x>, <operator, +=>, etc…
  - Is the atomic entity for compilation process
  - token- name will e used in the syntax analysis
  - attribute-value points to the corresponding item in symbol table， will be used in sematic analysis and code generation
- Every programming language has its own lexical regulations.

Lexical analysis

Program (character stream):
position = initial + rate * 60

Lexical Analyzer (Scanner)

Token Stream: <id,1> <=, > <id, 2> <+, > <id,3> <*, > <number, 4>

Error: 18..23 + val#ue

Variable names cannot have '#' character

Not a number

Syntax analysis

- **Syntax analysis/parsing**

  I cat sky interesting

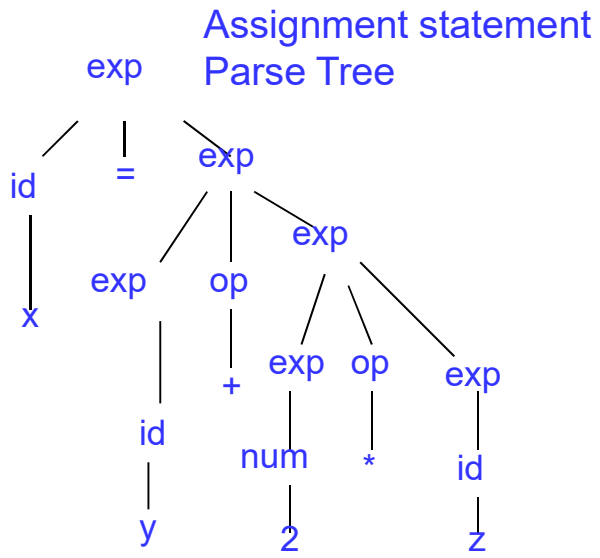
  A = a ++ b

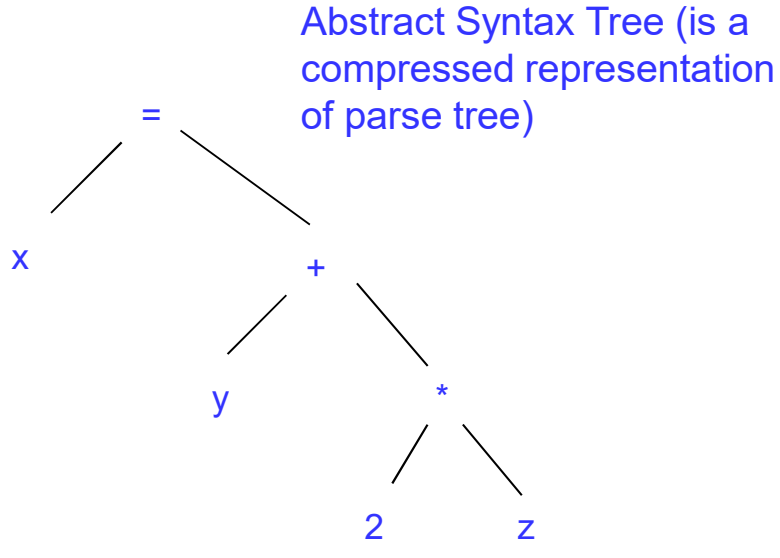
  - Parsing will check whether the sentence is legal, by building a tree
  - Parsing = Diagramming Sentences， where the diagram is a tree
  - The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream --- if we can build a tree, it is legal

Syntax analysis

Diagramming a Sentence into a Syntax Tree/Parse tree:
A hierarchical structure

x = y + 2 * z





Syntax analysis

- What is the basis for syntax analysis?
  - Rules of language --- grammar

```
goal → expr
expr → expr op term |
            term
term → number | id
op → + | -
...
```

  - Specification - Formal grammars
    - Chomsky hierarchy – context-free grammars
    - Each rule is called a production

Syntax analysis

Given a grammar, we can derive sentences by repeated substitution

e.g. subj + verb + obj + prep-phrase: I play basketball in the playground

Parsing is the reverse process – given a sentence, find out how it was derived from the grammar

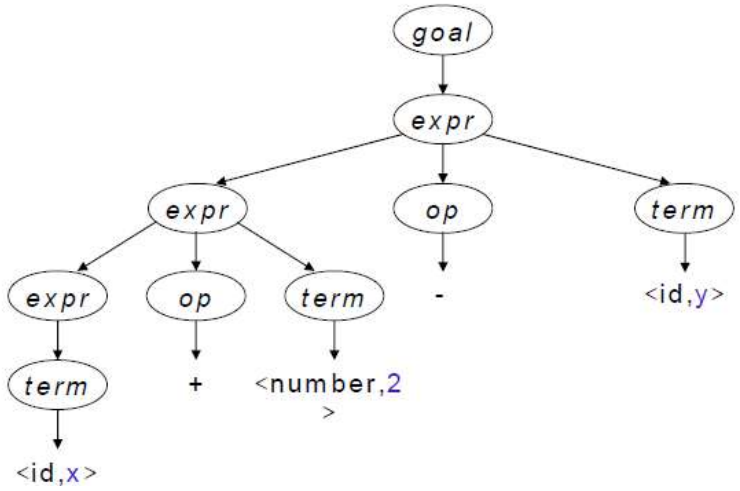
```
goal → expr
expr → expr op term
     |  term
term → number
     | id
op → +
   | -
```

Production	Result
	goal
1	*expr*
2	*expr op term*
5	*expr op* y
7	*expr* - y
2	*expr op term* - y
4	*expr op* 2 - y
6	*expr* + 2 - y
3	*term* + 2 - y
5	x + 2 - y

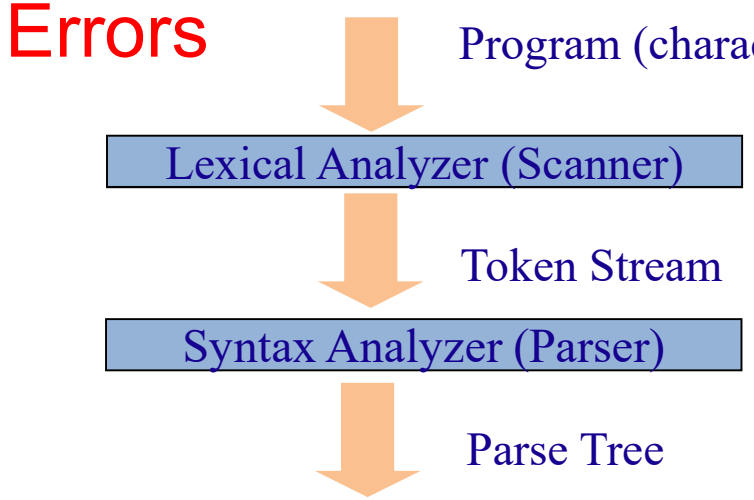
Syntax analysis

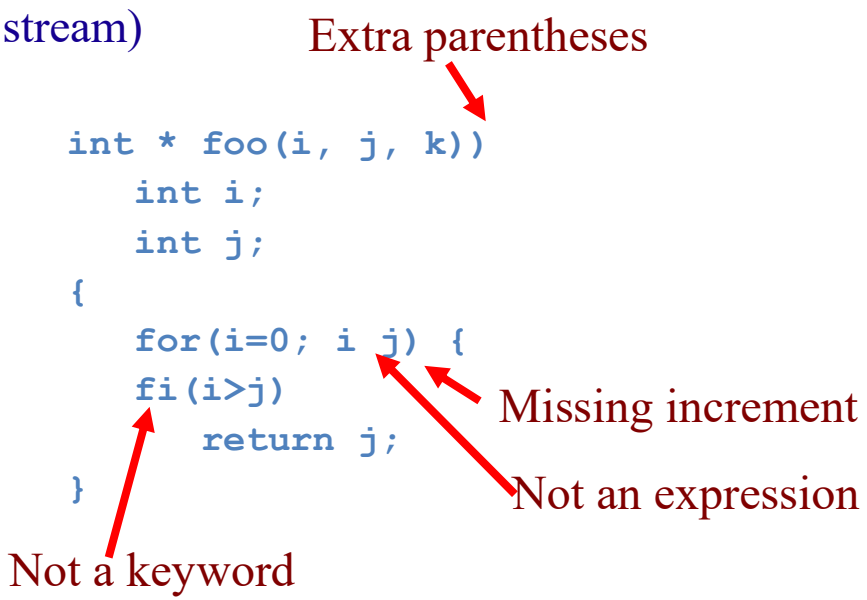
Diagram is called a parse tree or syntax tree

Production	Result
	goal
1	*expr*
2	*expr op term*
5	*expr op* y
7	*expr* - y
2	*expr op term* - y
4	*expr op* 2 - y
6	*expr* + 2 - y
3	*term* + 2 - y
5	x + 2 - y



Syntax analysis





Semantic analysis

- **Once sentence structure is understood, we can try to understand “meaning”**
  - Formally check the program against a specification, e.g. static checking, compile-time type checking, field checking, coercions, variable bindings

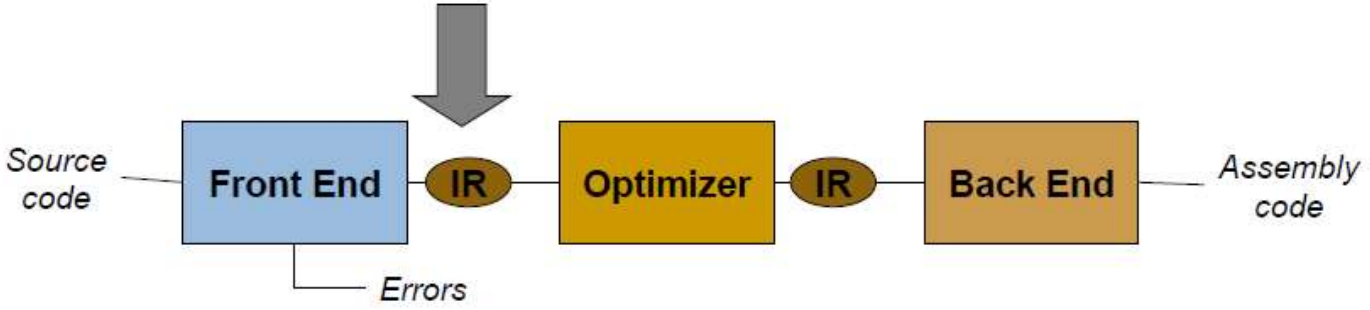
I eat cat in the sky

Semantic analysis

- Programming languages define strict rules to avoid ambiguities
- What does this code print? Why?

```
{
      int i = 3;
     {
        int i = 4;
        System.out.print(i);
      }
}
```

Where are we now?



- Front end
  - Produces fully-checked AST
  - Problem: AST still represents source-level semantics

Intermediate Representations

- **High-level IR (e.g. AST)**
  - Closer to source code
  - Hides implementation details
- **Low-level IR (e.g. three-address code)**
  - Closer to the machine
  - Exposes details (registers, instructions, etc)
- **Many tradeoffs in IR design**
- **Most compilers have 1 or maybe 2 IRs:**
  - Typically closer to low-level IR
  - Better for optimization and code generation

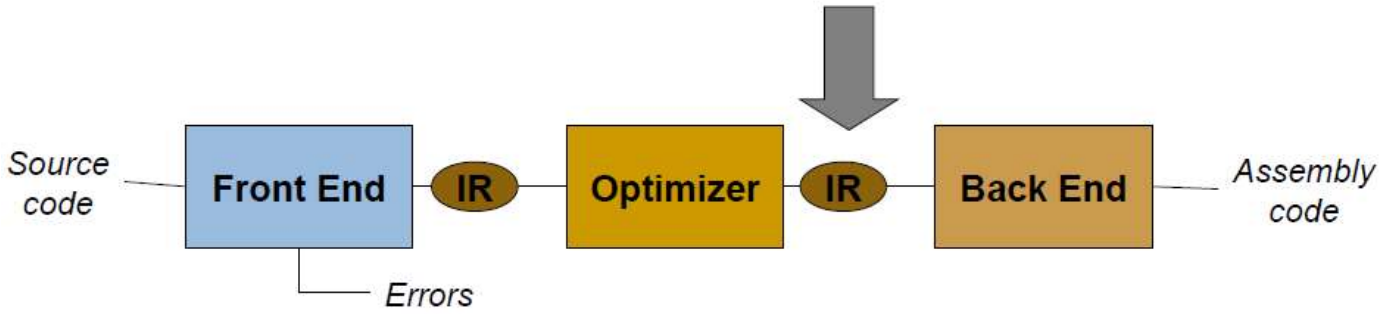
Code Optimization

- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.
- Series of passes – often repeated
  - Reduce cost
  - Run faster
  - Use less memory
  - Conserve some other resource, like power
- Must preserve program semantics

Code Optimization

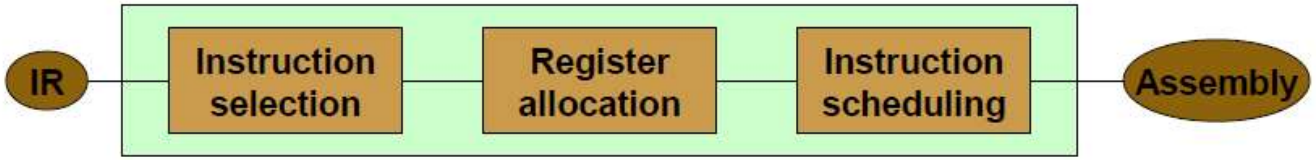
- Typical optimizations
  - Dead-code elimination, common sub-expression elimination, loop-invariant code motion, strength reduction
- Often contain assumptions about performance tradeoffs of the underlying machine
  - Relative speed of arithmetic operations – plus versus times
  - Possible parallelism in CPU
  - Cost of memory versus computation
  - Size of various caches

Where are we ?



- Optimization output
  - Transformed program
  - Typically, same level of abstraction

Back End



- Responsibilities
  - Map abstract instructions to real machine architecture
  - Allocate storage for variables in registers
  - Schedule instructions (often to exploit parallelism)
  - Finally, output the target code

Symbol-Table Management

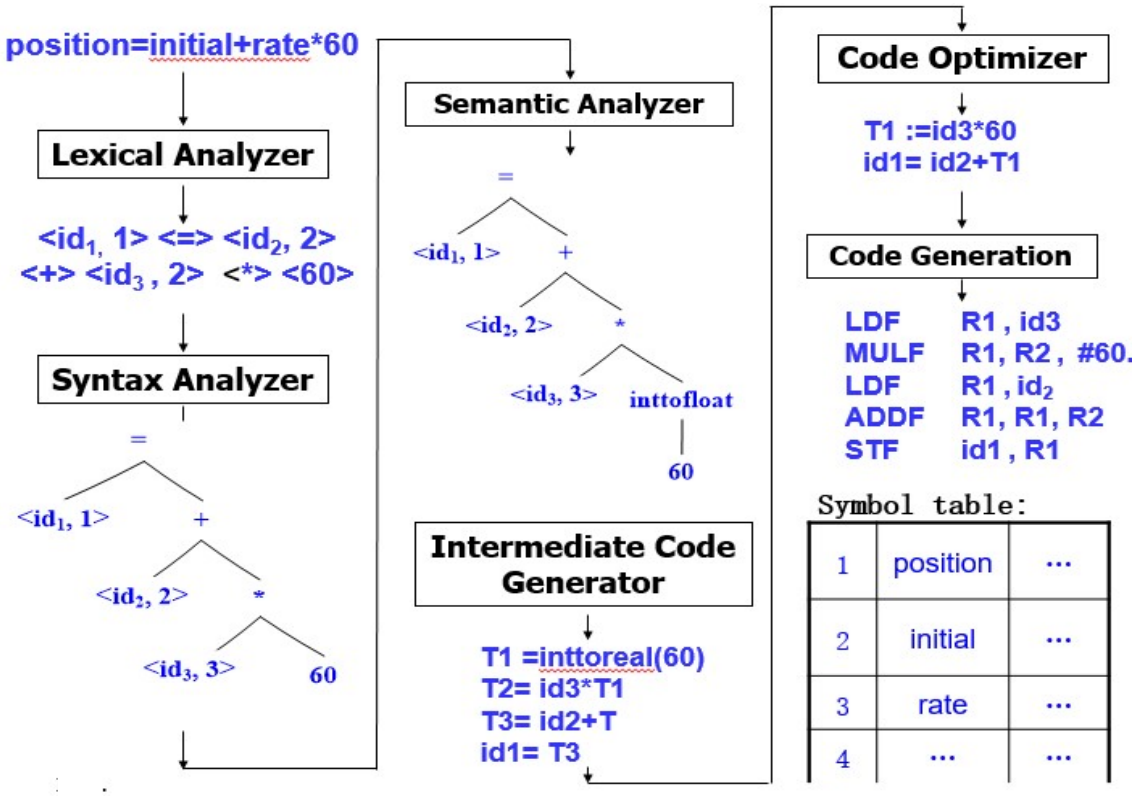
Identifier	Class	Type	Value/Address	···
rate	variable	Integer	relative at 8 hex	···
compare	Procedure	1 integer param	Absolute at 1000 hex	···

- A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- Record the identifiers used in the source program and collect information about various attributes of each identifier, such as its type, its scope
- Shared by later phases

Error Detection and Reporting

- The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler
- Exception handling

A complete example



Implementation of Compiler

Implementation

- **Requirements**
  - Correct
    - The actions requested by the program has to be faithfully executed
  - Efficient
    - Intelligently and efficiently use the available resources to carry out the requests
    - (the word optimization is used loosely in the compiler community – Optimizing compilers are never optimal)

Compiler-Construction Tools

- **Software development tools are available to implement one or more compiler phases**
  - Scanner generators
  - Parser generators
  - Syntax-directed translation engines
  - Automatic code generators
  - Data-flow engines

课后作业

- Textbook, Page 3: 1.1.2, 1.1.3, 1.1.4
- List some famous tools for constructing compilers, with brief introduction and comparison.

Thank you!