Compilers and Interpreters

Introduction

Computer Architecture

- The design of a computer's CPU architecture, instruction set, addressing modes
- Architecture is often defined as the set of machine attributes that a programmer should understand in order to successfully program the specific computer
- So, in general, computer architecture refers to attributes of the system visible to a programmer, that have a direct impact on the execution of a program.

Brief history of computer architecture

- First Generation (1945-1958)
  - Vacuum tubes
  - Machine code, Assembly language
  - Two types of models for a computing machine: Harvard architecture and Von Neumann architecture
  - Computers contained a central processor that was unique to that machine
  - Different types of supported instructions, few machines could be considered "general purpose"
  - Use of drum memory or magnetic core memory, programs and data are loaded using paper tape or punch cards
  - 2 Kb memory, 10 KIPS

Brief history of computer architecture (cont.)

- Second Generation (1958-1964)
  - Transistors – small, low-power, low-cost, more reliable than vacuum tubes,
  - Magnetic core memory
  - floating point arithmetic
  - Reduced the computational time from milliseconds to microseconds
  - High level languages
  - First operating Systems: handled one program at a time
  - 1959 - IBM´s 7000 series mainframes were the company’s first transistorized computers.

Brief history of computer architecture (cont.)

- Third Generation (1964-1974)
  - Introduction of integrated circuits combining thousands of transistors on a single chip
  - Semiconductor memory
  - Timesharing, graphics, structured programming
  - 2 Mb memory, 5 MIPS
  - Use of cache memory
  - IBM’s System 360 - the first family of computers making a clear distinction between architecture and implementation

Brief history of computer architecture (cont.)

- Fourth Generation (1974-present)
  - Introduction of Very Large-Scale Integration (VLSI)/Ultra Large Scale Integration (ULSI) - combines millions of transistors
  - Single-chip processor and the single-board computer emerged
  - Smallest in size because of the high component density
  - Creation of the Personal Computer (PC)
  - Wide spread use of data communications
  - Object-Oriented programming: Objects & operations on objects
  - Artificial intelligence: Functions & logic predicates

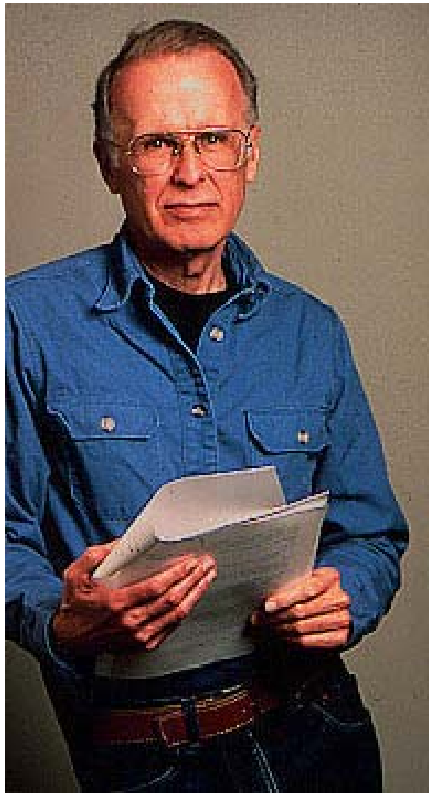
How many hardware men can claim that their machines are still in use after sixty years - software is so much more long lived.

Grace Hopper (1906-1992)

American computer scientist and United States Navy rear admiral. A pioneer in the field, she was one of the first programmers of the Harvard Mark I computer, and developed the first compiler for a computer programming language compiler.

The inventor of A-0, COBOL, and the term “compiler”.

John Backus (1924 - 2007)



"I' m a terribly unscholarly person, and lazy. That was my motivating force in most of what I did, was how to avoid work."

- John Backus (December 3, 1924 – March 17, 2007), the developer of FORTRAN, for years one of the best known and most used programming systems in the world),
- interviewed in 2006

Avram Noam Chomsky (1928 -)

American linguist, philosopher, cognitive scientist, historian, and activist. He is an Institute Professor and Professor (Emeritus) in the Department of Linguistics & Philosophy at MIT, where he has worked for over 50 years. Chomsky has been described as the "father of modern linguistics and a major figure of analytic philosophy His work has influenced fields such as computer science, mathematics, and psychology.

Chomsky is credited as the creator or co-creator of the Chomsky hierarchy, the universal grammar theory, and the Chomsky–Schützenberger theorem.

A short history of compilers

- First, there was nothing.
- Then, there was machine code.
- Then, there were assembly languages.
- Programming expensive; 50% of costs for machines went into programming.

Programming

- What’ s the problem?
  - Assembly programming very slow and error-prone
  - Software costs exceeded hardware costs!
- John Backus: “Speedcoding”
  - Simulate a more convenient machine
  - Idea: translate high-level code to assembly
- 1954-7 FORTRAN I project
  - By 1958, more than 50% of codes are in FORTRAN
  - Cut development time dramatically — from weeks to hours

FORTRAN I

- The first compiler
  - Huge impact on computer science
  - Produced codes almost as good as hand-written
- Led to an enormous body of work
  - Theoretical work on languages, compilers
  - Program semantics
  - Thousands of new languages
- Modern compilers preserve the outlines of FORTRAN I

Programming Languages: Domains of Application

- Scientific (Fortran)
- Business (COBOL)
- Artificial intelligence (LISP)
- Systems (C)
- Web (Java)
- General purpose (C++)

Kinds of Languages - 1

- Imperative
  - Specifies how a computation is to be done.
  - Examples: C, C++, C#, Fortran, Java
- Declarative
  - Specifies what computation is to be done.
  - Examples: Haskell, ML, Prolog
- von Neumann
  - One whose computational model is based on the von Neumann architecture.
  - Basic means of computation is through the modification of variables (computing via side effects).
  - Statements influence subsequent computations by changing the value of memory.
  - Examples: C, C++, C#, Fortran, Java

Kinds of Languages - 3

- Parallel
  - One that allows a computation to run concurrently on multiple processors.
  - Examples
    - Libraries: POSIX threads, MPI
    - Languages: Ada, Cilk, OpenCL, Chapel, X10
    - Architecture: CUDA (parallel programming architecture for GPUs)

Kinds of Languages - 3

- Domain specific
  - Many areas have special-purpose languages to facilitate the creation of applications.
  - Examples
    - YACC for creating parsers
    - LEX for creating lexical analyzers
    - MATLAB for numerical computations
    - SQL for database applications
- Markup
  - Not programming languages in the sense of being Turing complete, but widely used for document preparation.
  - Examples: HTML, XHTML, XML

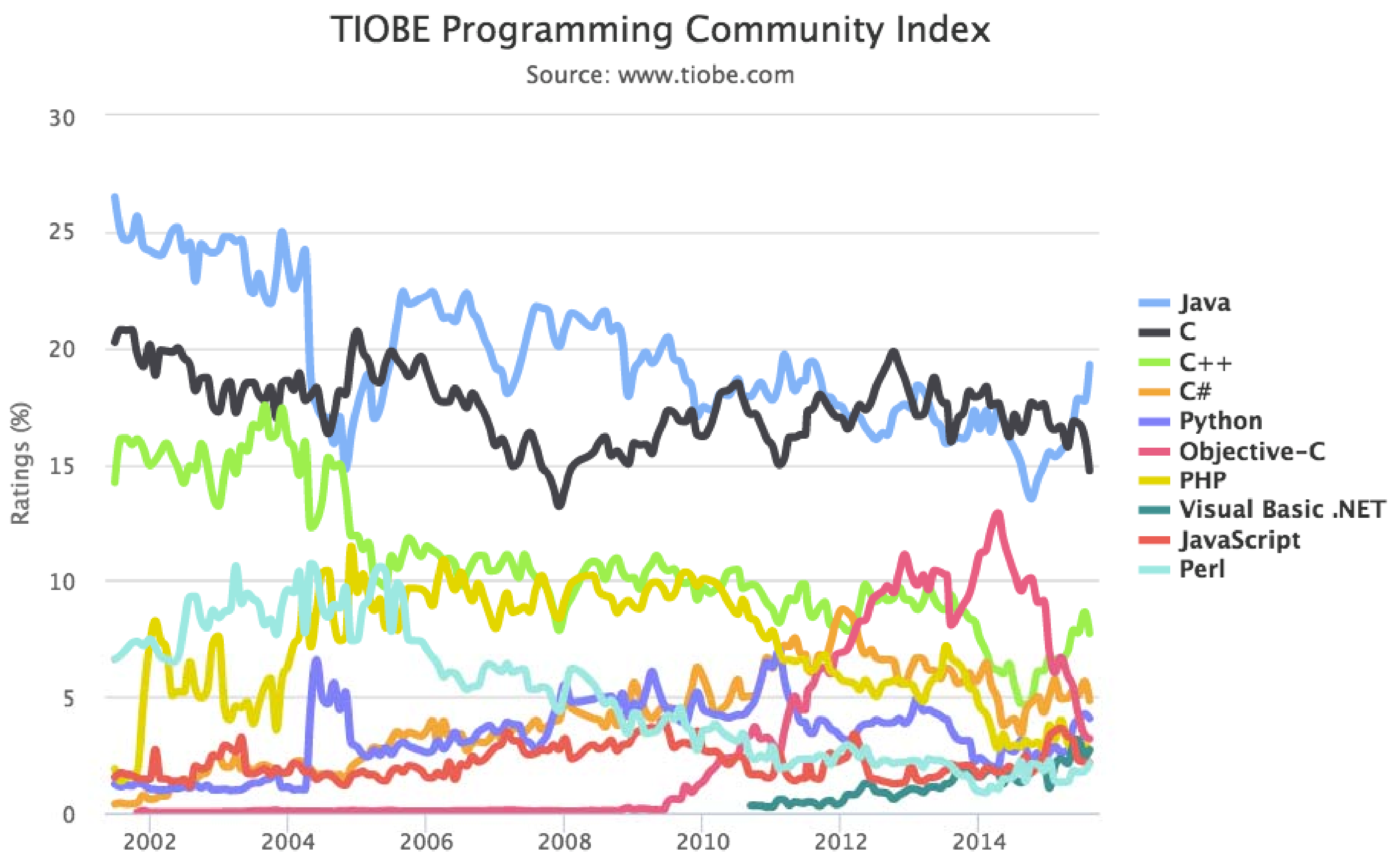
Evolution of Programming Languages

- There are thousands of programming languages.
  - First-generation: machine language
  - Second-generation: assembly languages
  - Third-generation: higher-level languages like Fortran, Cobol, Lisp, C, C++, C# and Java
  - Fourth-generation language: designed for specific applications like NOMAD, SQL, Postscript.
  - Fifth-generation: applied to logic- and constraint-based languages like Prolog and OPS5.

Evolution of Programming Languages

1970

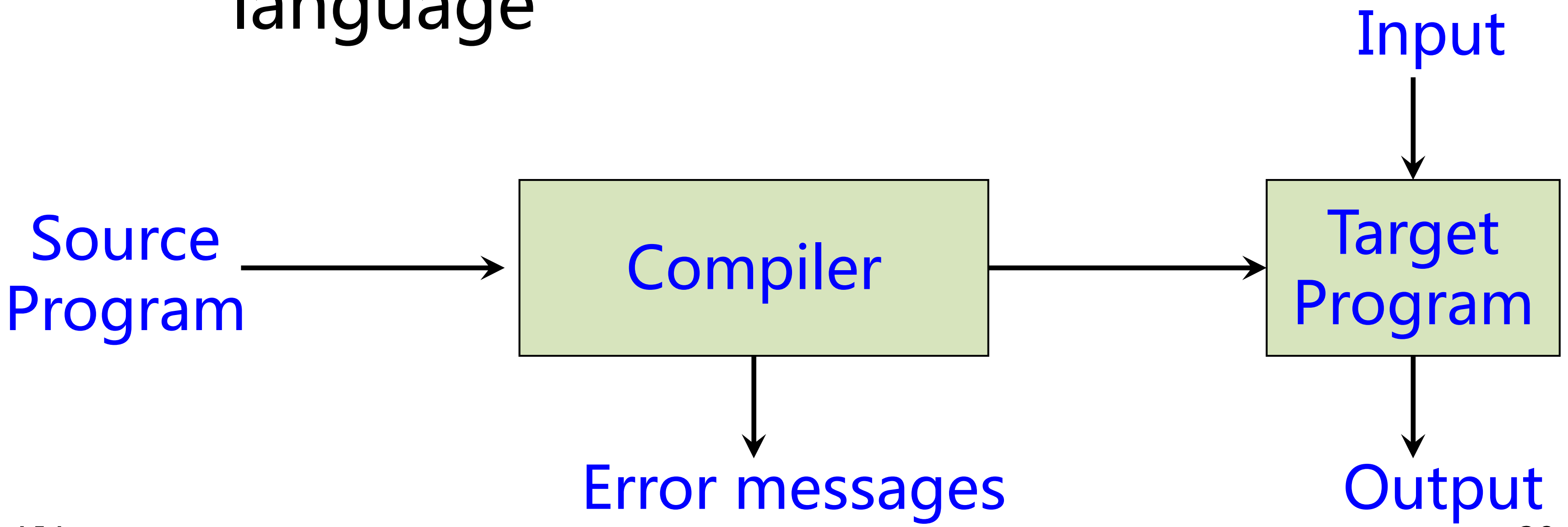
Fortran
Lisp
Cobol
Algol 60
APL
Snobol 4
Simula 67
Basic
PL/1
Pascal



[http://www.tiobe.com, Auguest 2015

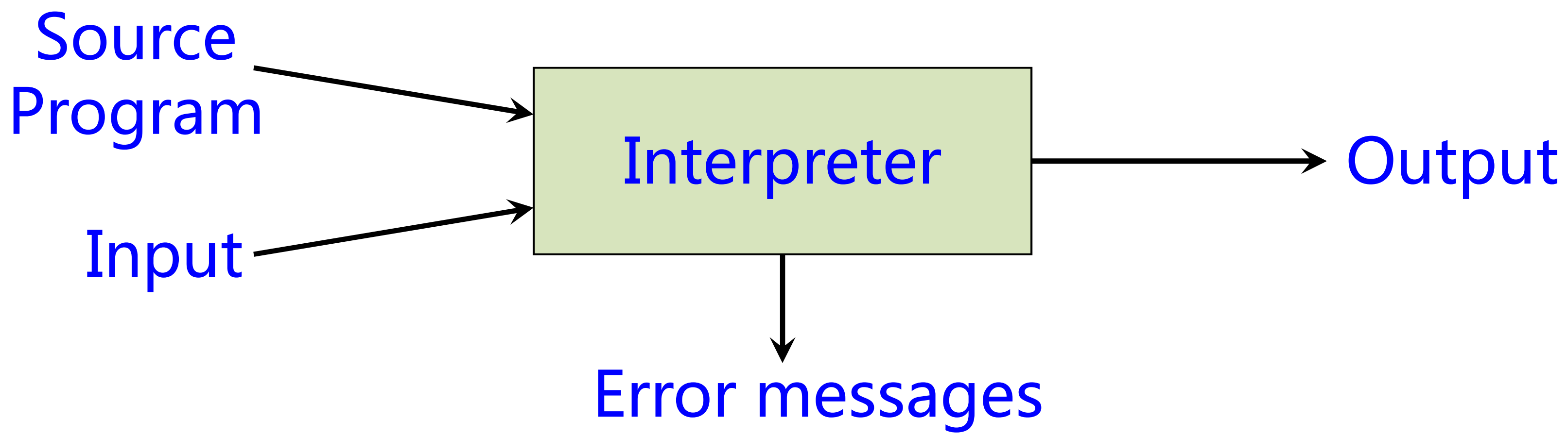
Compilers and Interpreters

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



Compilers and Interpreters

- “*Interpretation*”
  - Performing the operations implied by the source program



Language implementations

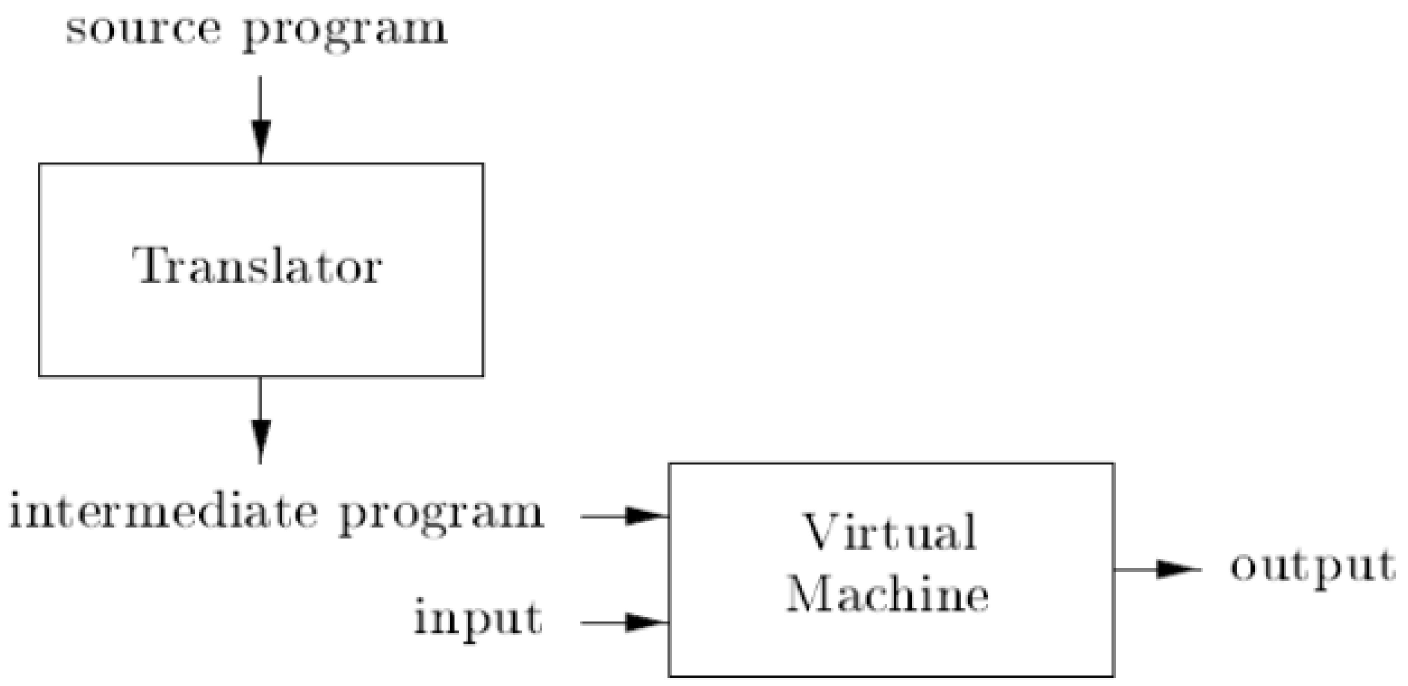
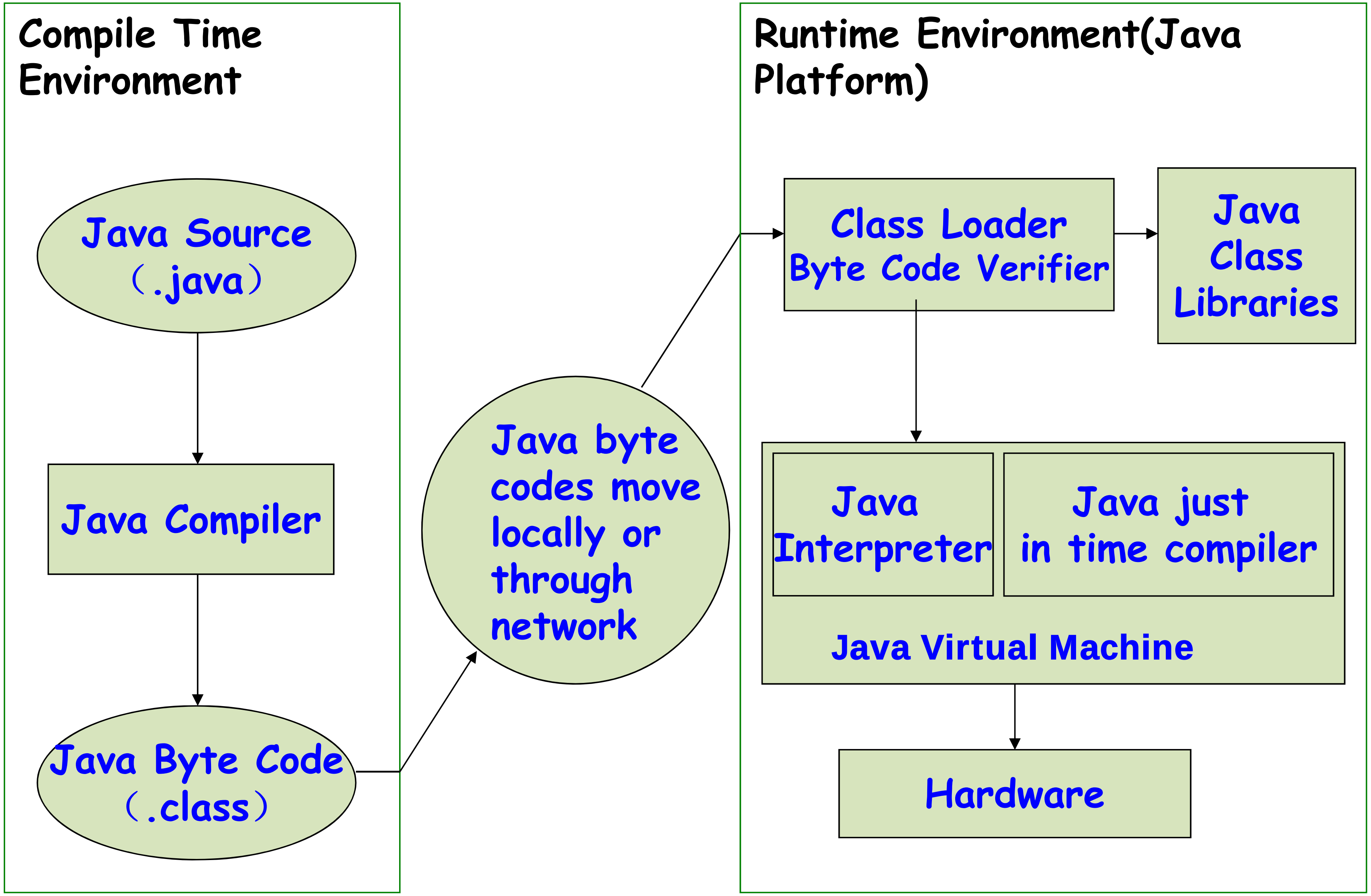


Figure 1.4: A hybrid compiler


Compile Time Environment
Java Source (.java)
Java Compiler
Java Byte Code (.class)
Java byte codes move locally or through network
Runtime Environment(Java Platform)
Class Loader Byte Code Verifier
Java Class Libraries
Java Interpreter
Java just in time compiler
Java Virtual Machine
Hardware

Language Involved

- Compilation is a language translation problem

```
int i = 10;
while (i > 0) {
    x = x * 2;
    i = i - 1;
}
```

Source

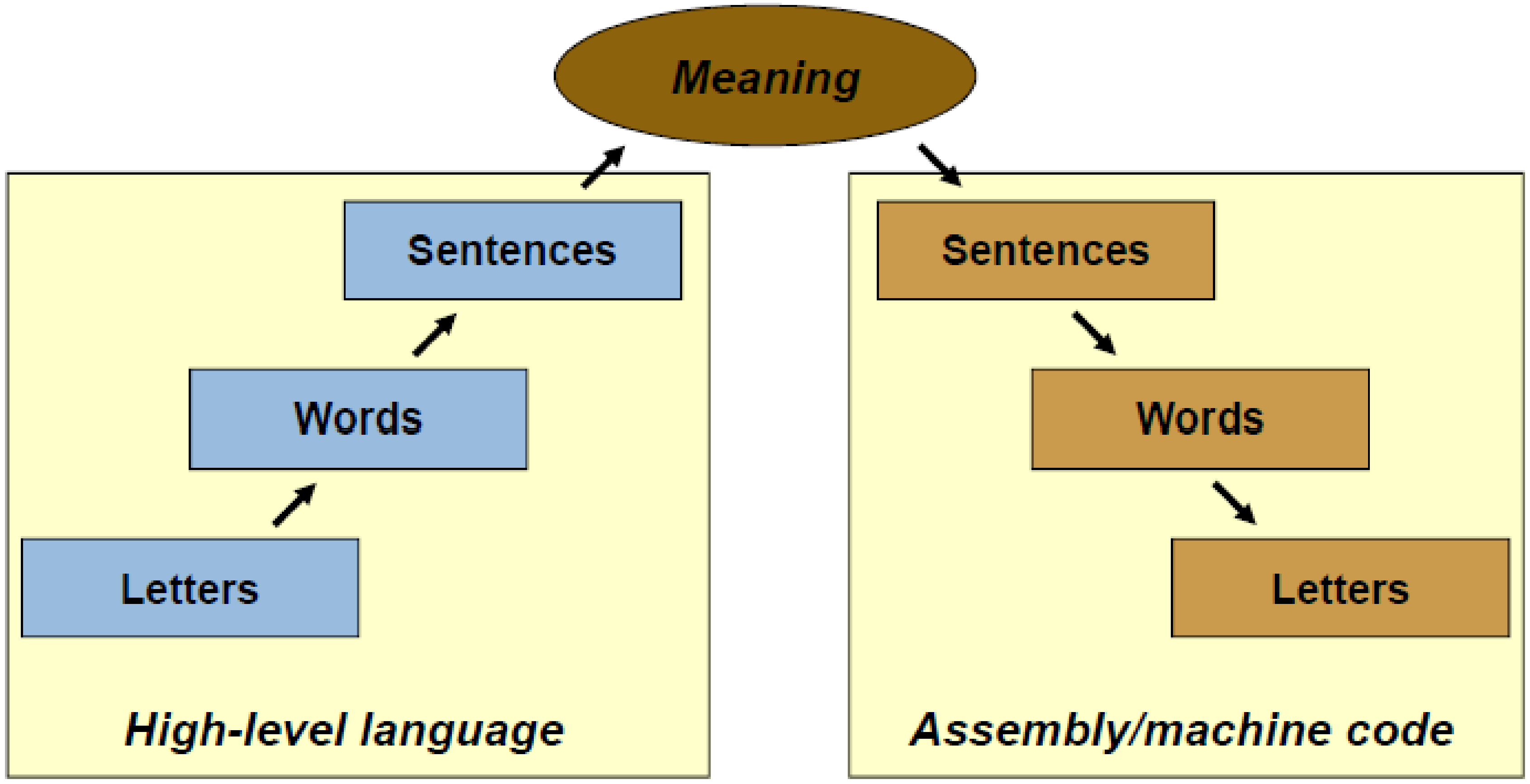
```
    movl      %esp, %ebp
    subl      $4, %esp
    movl      $10, -4(%ebp)
.L2:
    cmpl      $0, -4(%ebp)
    jle       .L3
    movl      8(%ebp), %eax
    sall      %eax
    movl      %eax, 8(%ebp)
    leal      -4(%ebp), %eax
    decl      (%eax)
    jmp       .L2
.L3:
    movl      8(%ebp), %eax
```

Target

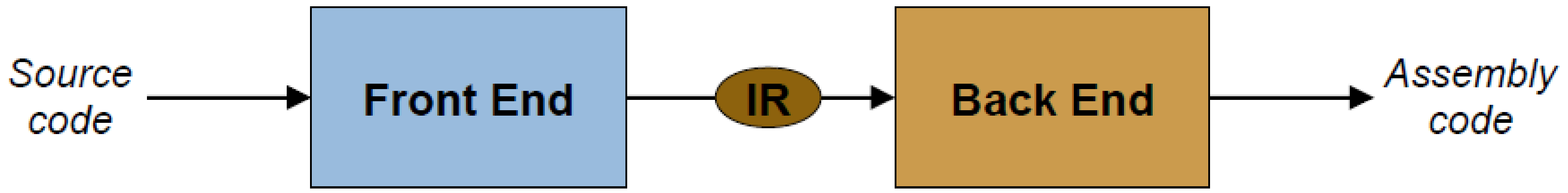
The compilation problem

- Assembly language
  - Converts trivially into machine codes
  - No abstraction: load, store, add, jump, etc.
  - Extremely painful to program
  - What are other problems with assembly programming?
- High-level language
  - Easy to understand and maintain
  - Abstraction: control (loops, branches); data (variables, records, arrays); procedures, functions
  - ***Problem**: how do we get from one to the other?*

Translation process



Basic compiler structure

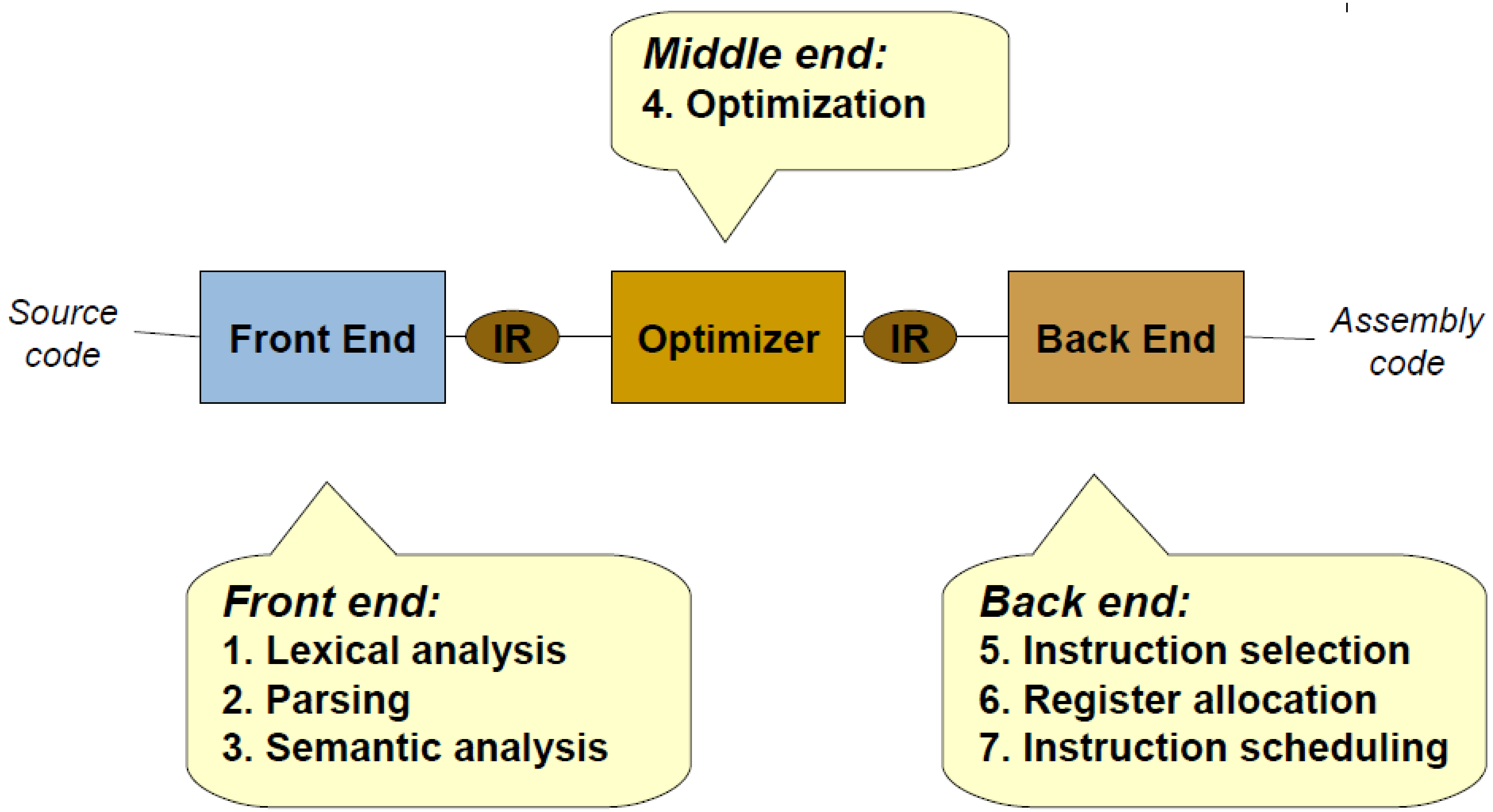


- Traditional two-pass compiler
  - ***Front-end*** reads in source code
  - ***Internal representation*** captures meaning
  - ***Back-end*** generates assembly

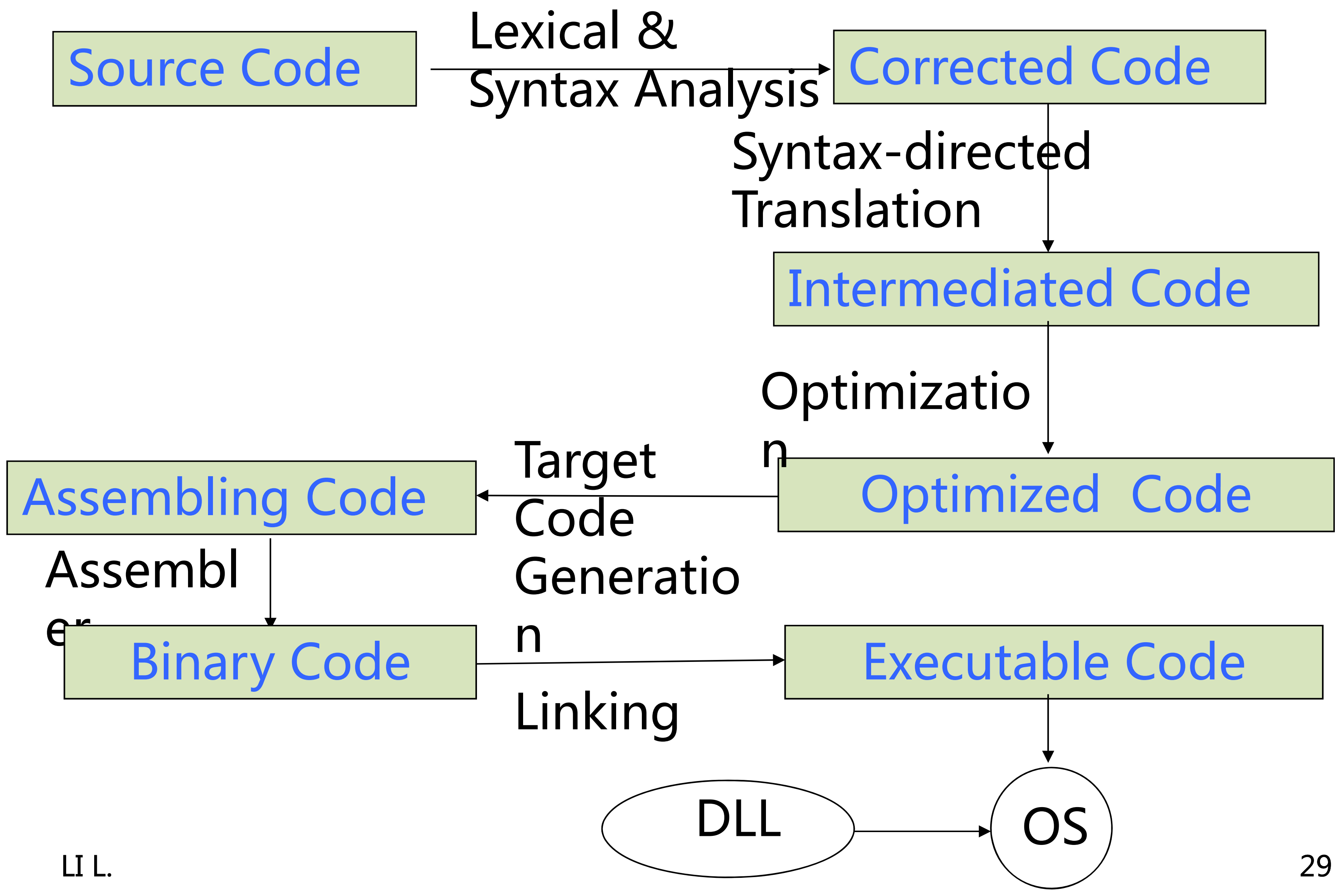
- Advantages?

  Decouples input language from target machine

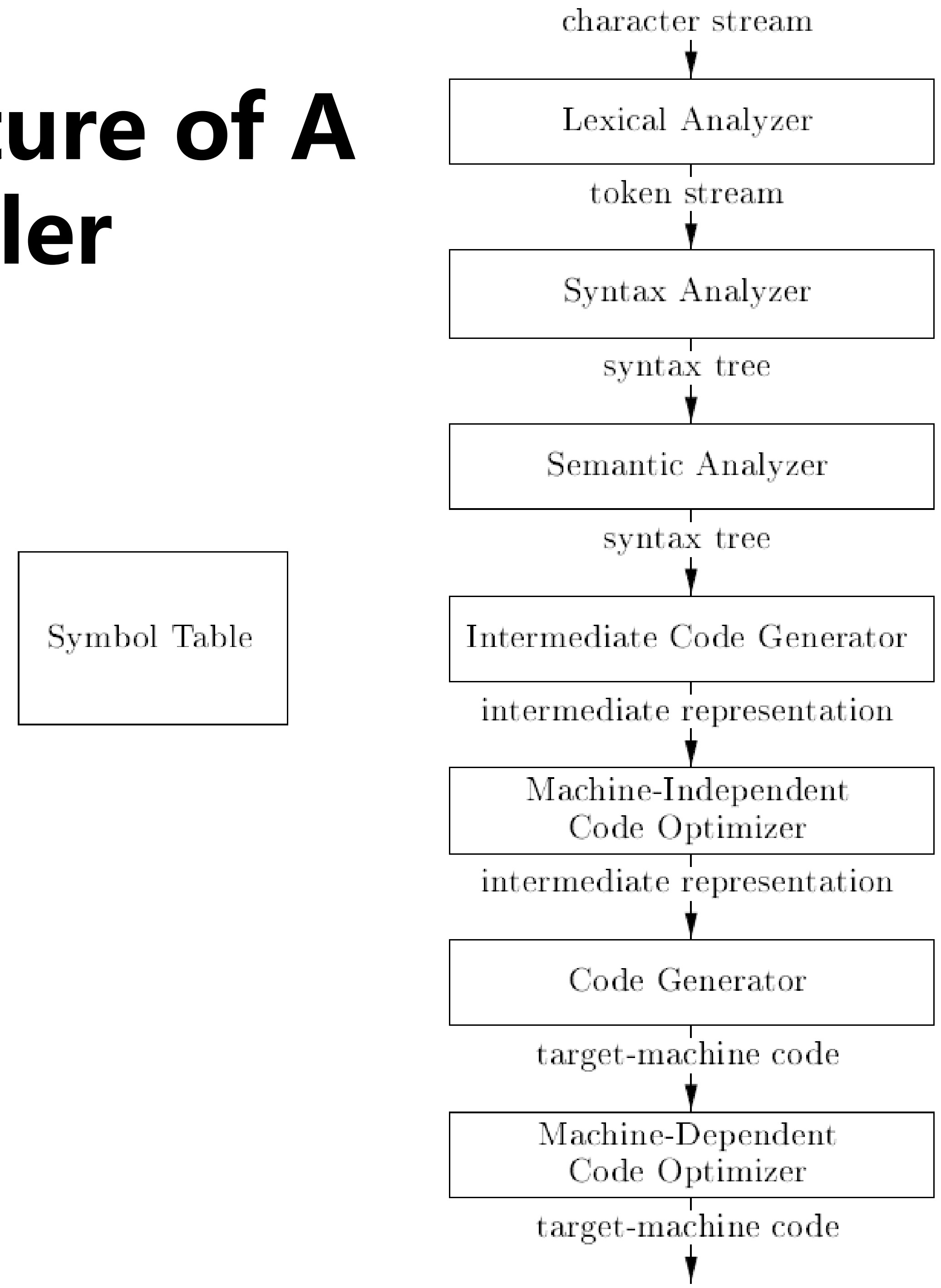
Modern optimizing compiler



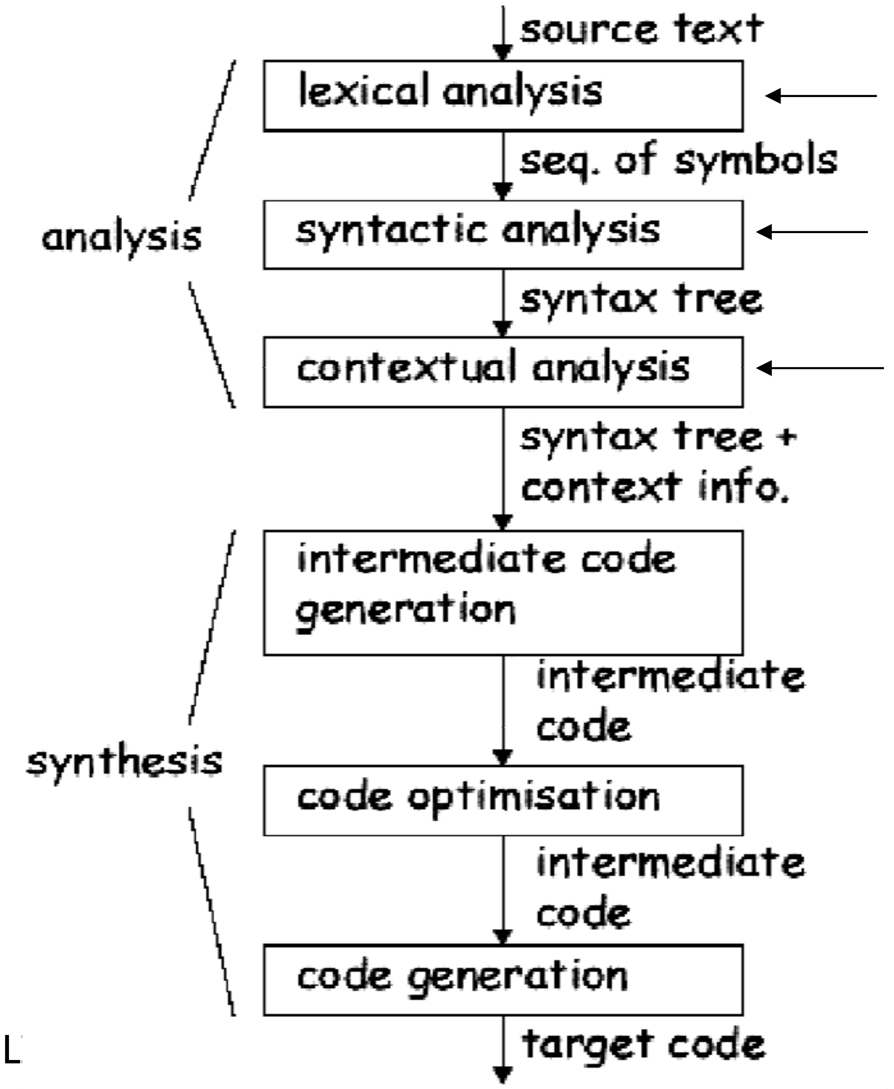
How is a program to be processed and run?



More Structure of A compiler



The Analysis-Synthesis Model of Compilation



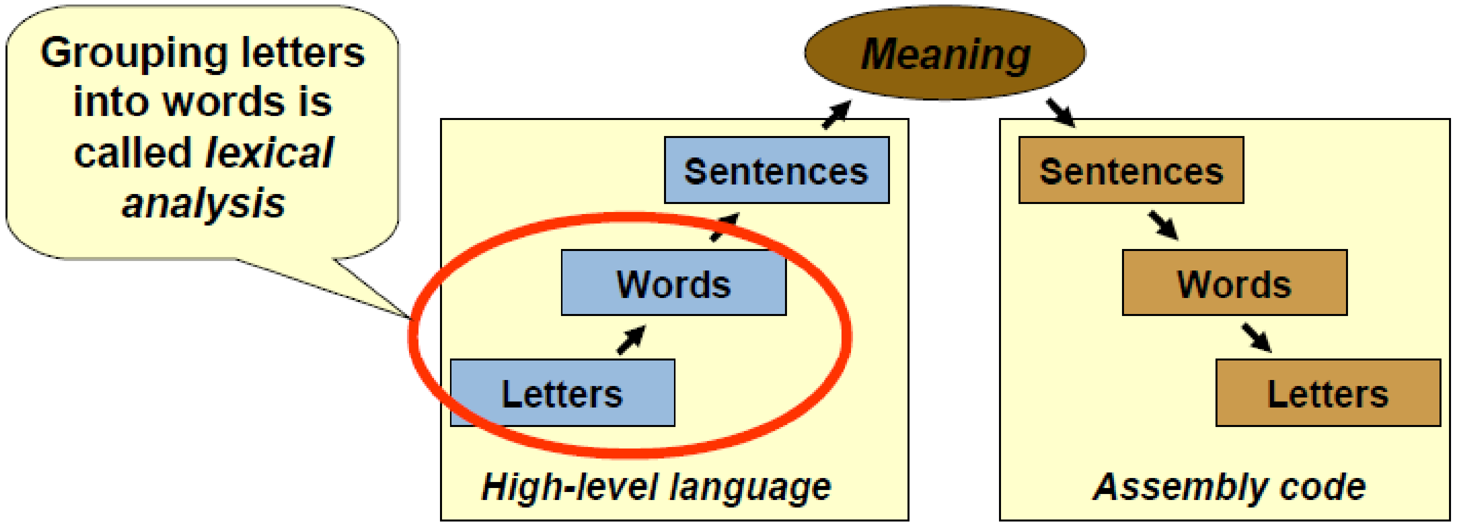
Linear analysis

Hierarchical analysis

Semantic analysis (type checking)

Lexical Analyzer

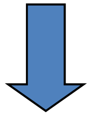
- The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes
  - A token: < token- name, attribute-value >
  - Tokens have value and type: <if, keyword>, <x, identifier>, <+=, operator>, etc….



Lexical Analyzer (cont.)

- Lexical analyzer divides program text into "words" or tokens

position=initial+rate*60



<id$_1$, 1>, <=>, <id$_2$, 2>, <+>, <id$_3$, 2>, <*>, <60>

- Some lexical rules
  - Capital "T" (start of sentence symbol)
  - Blank " " (word separator)
  - Period "." (end of sentence symbol)
  - Comment delimiter " /* "

Lexical Analyzer (cont.)

- How do we specify tokens?
- Keyword – an exact string
- What about identifier? floating point number?

- **Regular expressions**
- Just like Unix tools grep, awk, sed, etc.
- Identifier: [a-zA-Z_][a-zA-Z_0-9]*

Syntax Analyzer

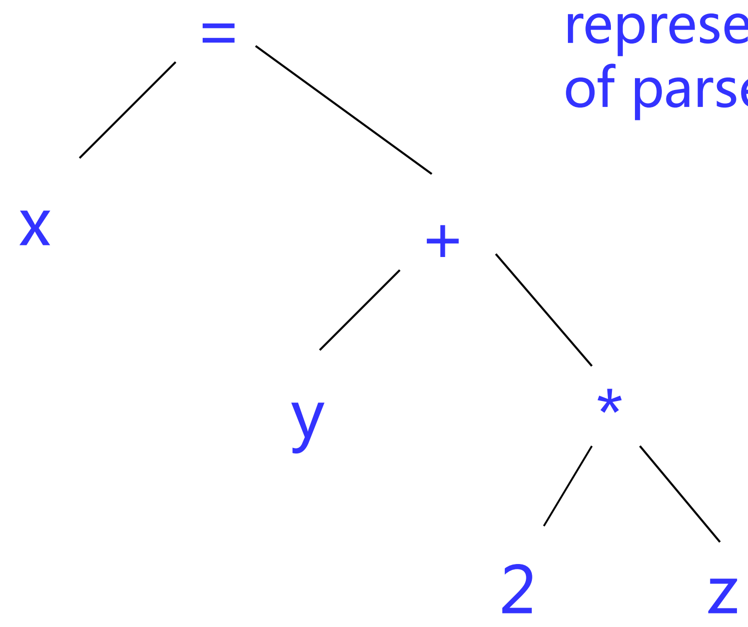
- Parsing = Diagramming Sentences
  - The diagram is a tree…

- 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.

Syntax Analyzer (cont.)

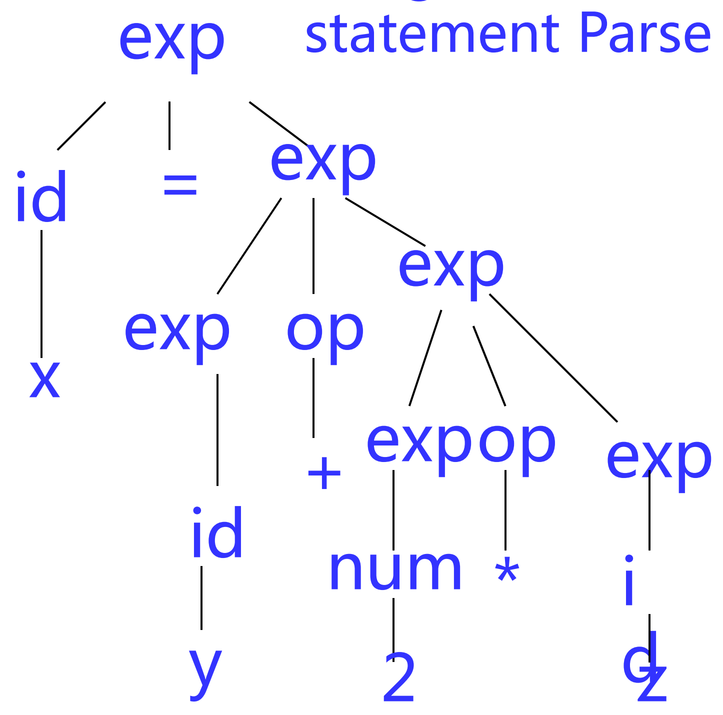
- Diagramming a Sentence
- Syntax Tree/Parse tree: A hierarchical structure

x = y + 2 * z



Syntax Tree (is a compressed representation of parse tree)

Assignment statement Parse Tre



Syntax Analyzer (cont.)

- How do we describe the language?
  - Same as English: using grammar rules

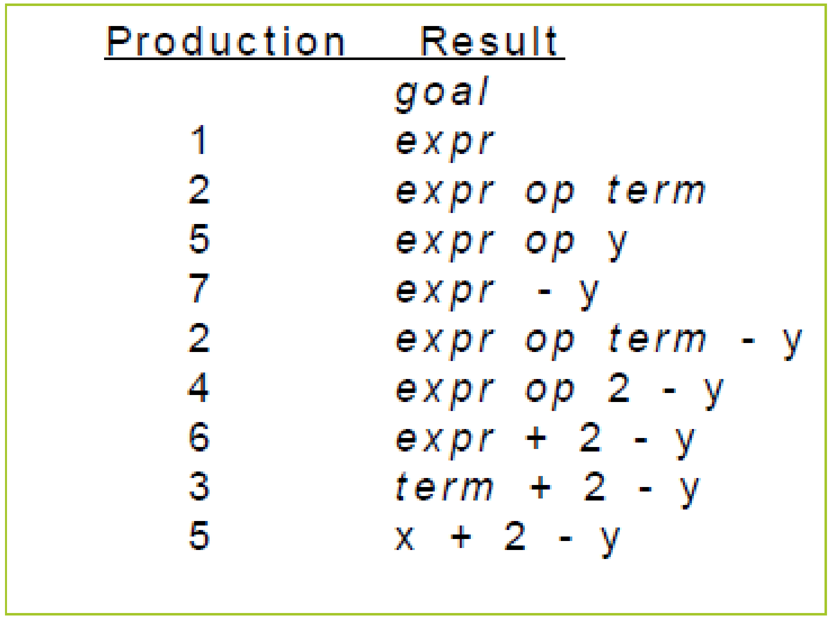
```
goal → expr
expr → expr op term |
term
term → number | id
op → + | -
…
```

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

Syntax Analyzer (cont.)

- Given a grammar, we can derive sentences by repeated substitution – Using grammars.
  - Parsing is the reverse process – given a sentence, find a derivation ((same as diagramming)

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

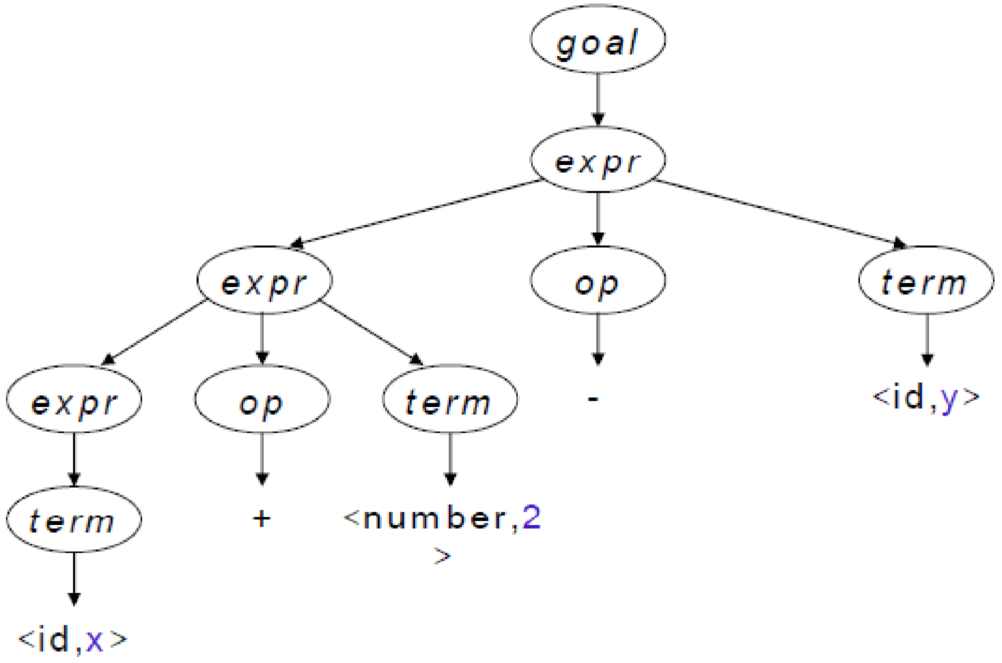


Production	Result
	goal
1	*expr*
2	*expr op term*
5	*expr op* y
7	*expr* - y
2	*expr op term* - y
4	*expr op* 2 - y
6	*expr* + 2 - y
3	*term* + 2 - y
5	x + 2 - y

Syntax Analyzer (cont.)

- Diagram is called a **parse tree** or **syntax tree**

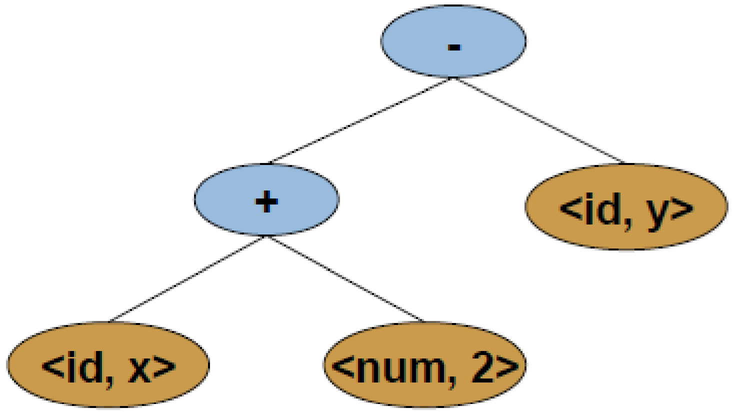
Production	Result
	goal
1	*expr*
2	*expr op term*
5	*expr op* y
7	*expr* - y
2	*expr op term* - y
4	*expr op* 2 - y
6	*expr* + 2 - y
3	*term* + 2 - y
5	x + 2 - y



Syntax Analyzer (cont.)

- Compilers often use an **abstract syntax tree**
- More concise and convenient:
  - Summarizes grammatical structure without including all the details of the derivation
  - ASTs are one kind of intermediate representation (IR)

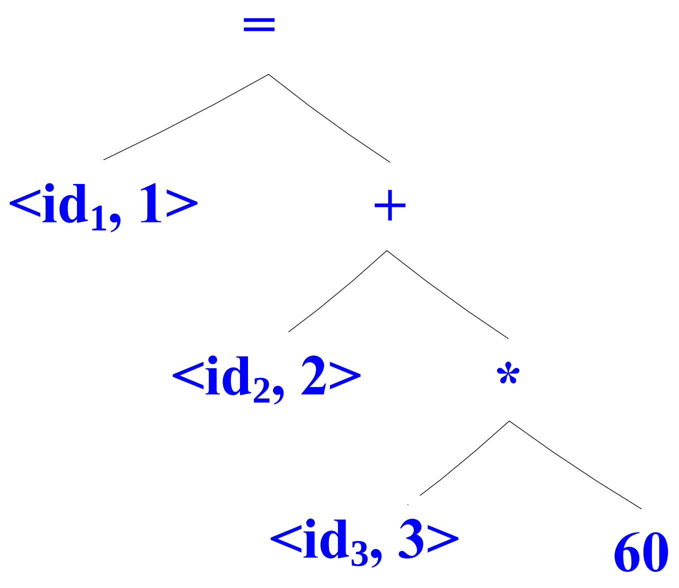
```
x  +  2  -  y
```



Syntax Analyzer (cont.)

position=initial+rate
*60

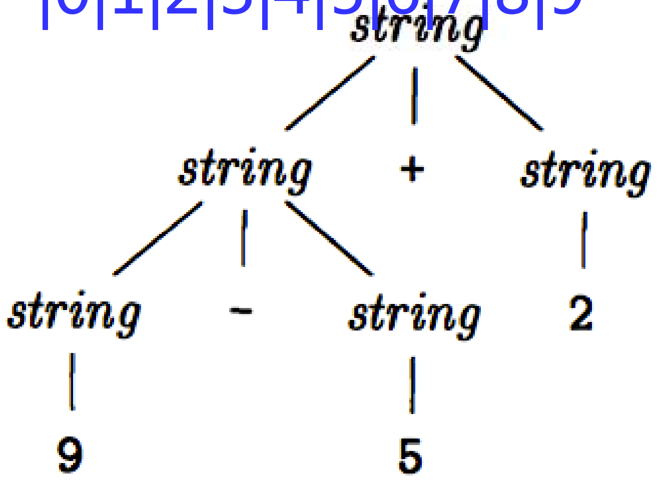
$<id_1, 1>, <=, >, <id_2, 2>, <+, >, <id_3, 3>, <*, >, <60, >$

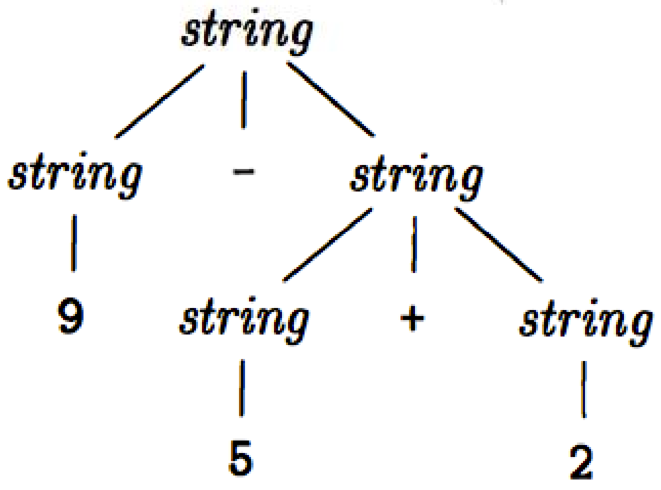


Syntax Analyzer (cont.)

- Ambiguity
  - A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be ambiguous.

string → string + string | string – string |0|1|2|3|4|5|6|7|8|9





Syntax Analyzer (cont.)

- Once sentence structure is understood, we can try to understand “meaning”
  - What would the ideal situation be?
  - Formally check the program against a specification

- Compilers perform limited analysis to catch inconsistencies

- Some do more analysis to improve the performance of the program

Syntax Analyzer (cont.)

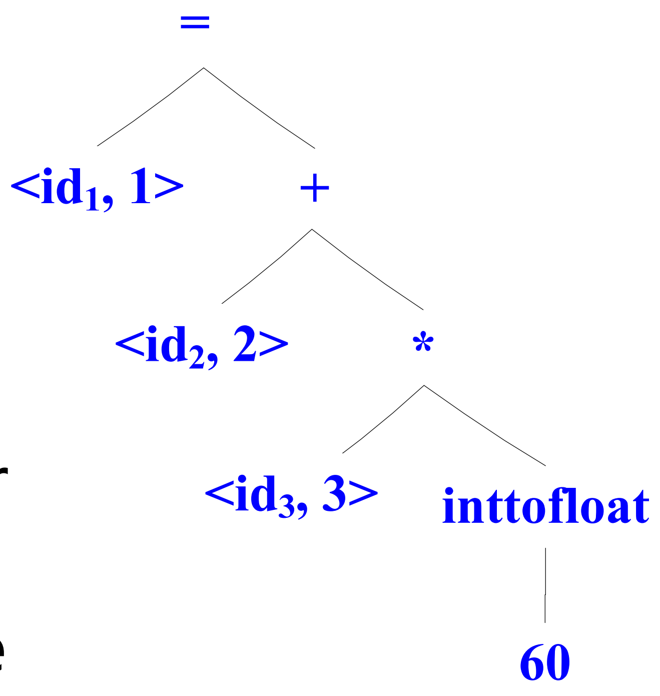
- Programming languages define strict rules to avoid ambiguities

- What does this code print? Why?

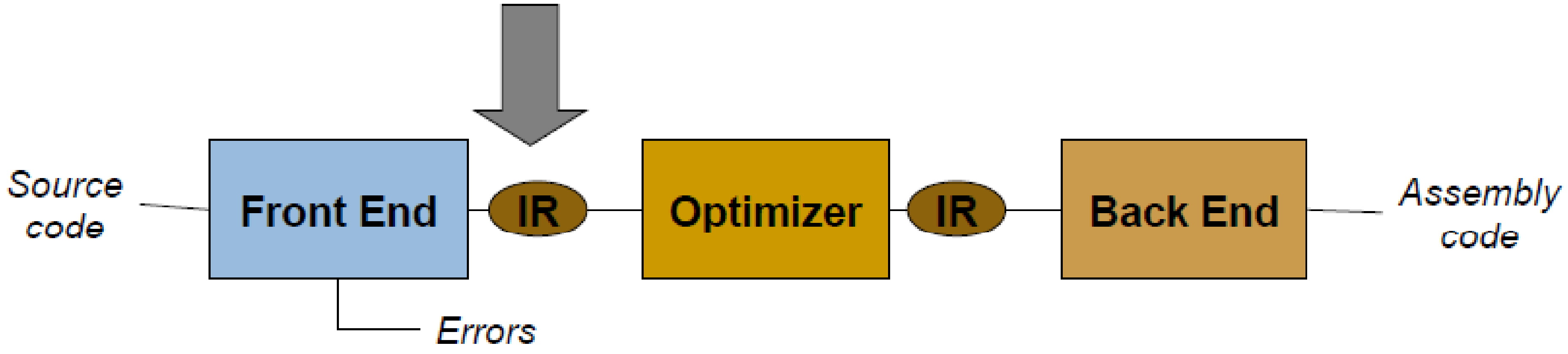
```
{
      int i = 3;
     {
       int i = 4;
       System.out.print(i);
      }
}
```

Syntax Analyzer (cont.)

- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree.
  - Static Checking, compile-time type checking, field checking, coercions, variable bindings



Where are we now?



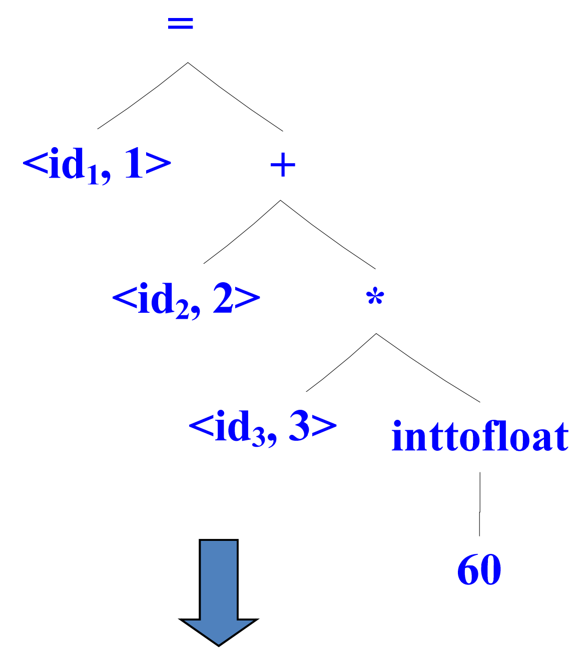
- Front end
  - Produces fully-checked AST
  - Problem: AST still represents source-level semantics

Intermediate Representations

- Many different kinds of IRs
  - High-level IR (e.g. AST)
  - Closer to source code
  - Hides implementation details
- Low-level IR
  - 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

Intermediate Code Generation

- This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.
- Dismantle complex structures into simple ones
  - Result is an IR called ***three-address code***



T_1 =inttoreal(60)
T_2= id_3*T_1
T_3= id_2+T_2
id_1= T_3

Code Optimization

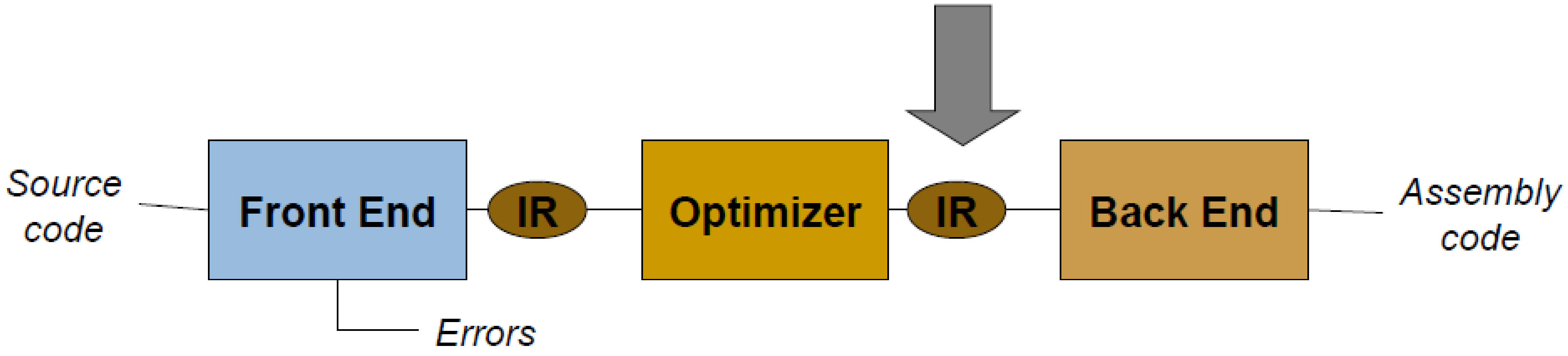
- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.
- Series of passes – often repeated
  -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

Code Optimization (cont.)

- Classic optimizations
  - Dead-code elimination, common sub-expression elimination, loop-invariant code motion, Strength reduction

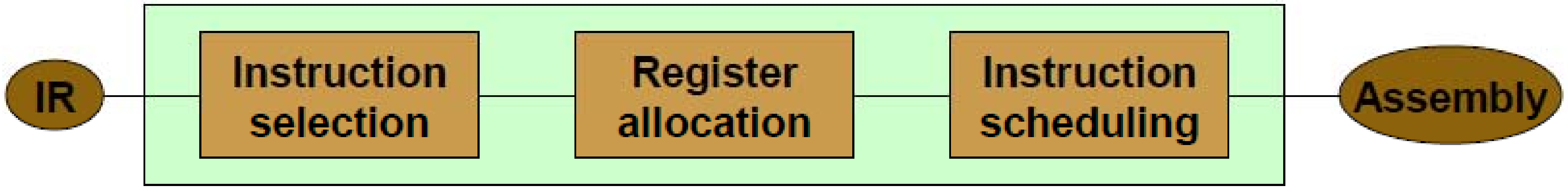
- Often contain assumptions about performance tradeoffs of the underlying machine
  - Relative speed of arithmetic operations – plus versus times
  - Possible parallelism in CPU
  - Cost of memory versus computation
  - Size of various caches

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)

Code Generation

- The code generator takes as input an intermediate representation of the source program and maps it into the target language.

$T_1 := id_3 * 60$
$id_1 = id_2 + T_1$

$\Longrightarrow$

LDF R_2 , id3
MULF R_1, R_2 , #60.0
LDF R_1 , id_2
ADDF R_1, R_1, R_2
STF id_1 , R_1

Symbol-Table Management

- 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

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

Error Detection and Reporting

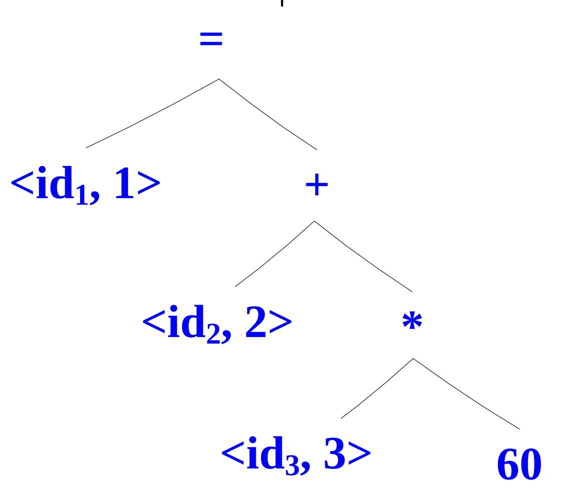
- The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler
- Exception handling

position=initial+rate*60

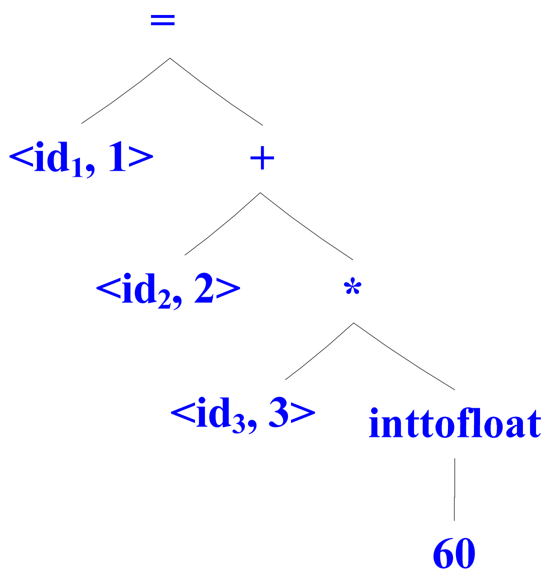
Lexical Analyzer

<id$_1$, 1> <=> <id$_2$, 2>
<+> <id$_3$, 2> <*> <60>

Syntax Analyzer



Semantic Analyzer



Intermediate Code Generator

T_1 =inttoreal(60)
T_2= id$_3$*T_1
T_3= id$_2$+T_2
id$_1$= T_3

Code Optimizer

T_1 :=id$_3$*60
id$_1$= id$_2$+T_1

Code Generation

```
LDF    R2 , id3
MULF   R1, R2 ,  #60.0
LDF    R1 , id2
ADDF   R1, R1, R2
STF    id1 , R1
```

Symbol talbe:

1	position	…
2	initial	…
3	rate	…
4	…	

The Grouping of Phases

- Compiler *passes:*
  - A collection of phases is done only once (*single pass*) or multiple times (*multi pass*)
    - **Single pass**: reading input, processing, and producing output by one large compiler program; usually runs faster
    - **Multi pass**: compiler split into smaller programs, each making a pass over the source; performs better code optimization

Translation Job #1 ---- Correctness

- Practical solution: automatic tools
  - Parser generators, regular expressions, rewrite systems, dataflow analysis frameworks, code generator-generators
  - Extensive testing

- Theoretical solution: a bunch of math
  - Formal description of semantics
  - A proof that the translation is correct

Translation Job #2 ---- Good Translation

- Produce a “good” translation

- What does it mean for compilers?
  - Good performance ---- optimization
  - Reduce the amount of work (“be concise”)
  - Utilize the hardware effectively

Finished program

- What else does the code need to run?
- Programs need support at run-time
  - Start-up code
  - Interface to OS
  - Libraries
- Varies significantly between languages
  - C – fairly minimal
  - Java – Java virtual machine

Runtime System

- Memory management services
  - Manage heap allocation
  - Garbage collection
- Run-time type checking
- Error processing (exception handling)
- Interface to the operating system
- Support of parallelism
  - Parallel thread initiation
  - Communication and synchronization

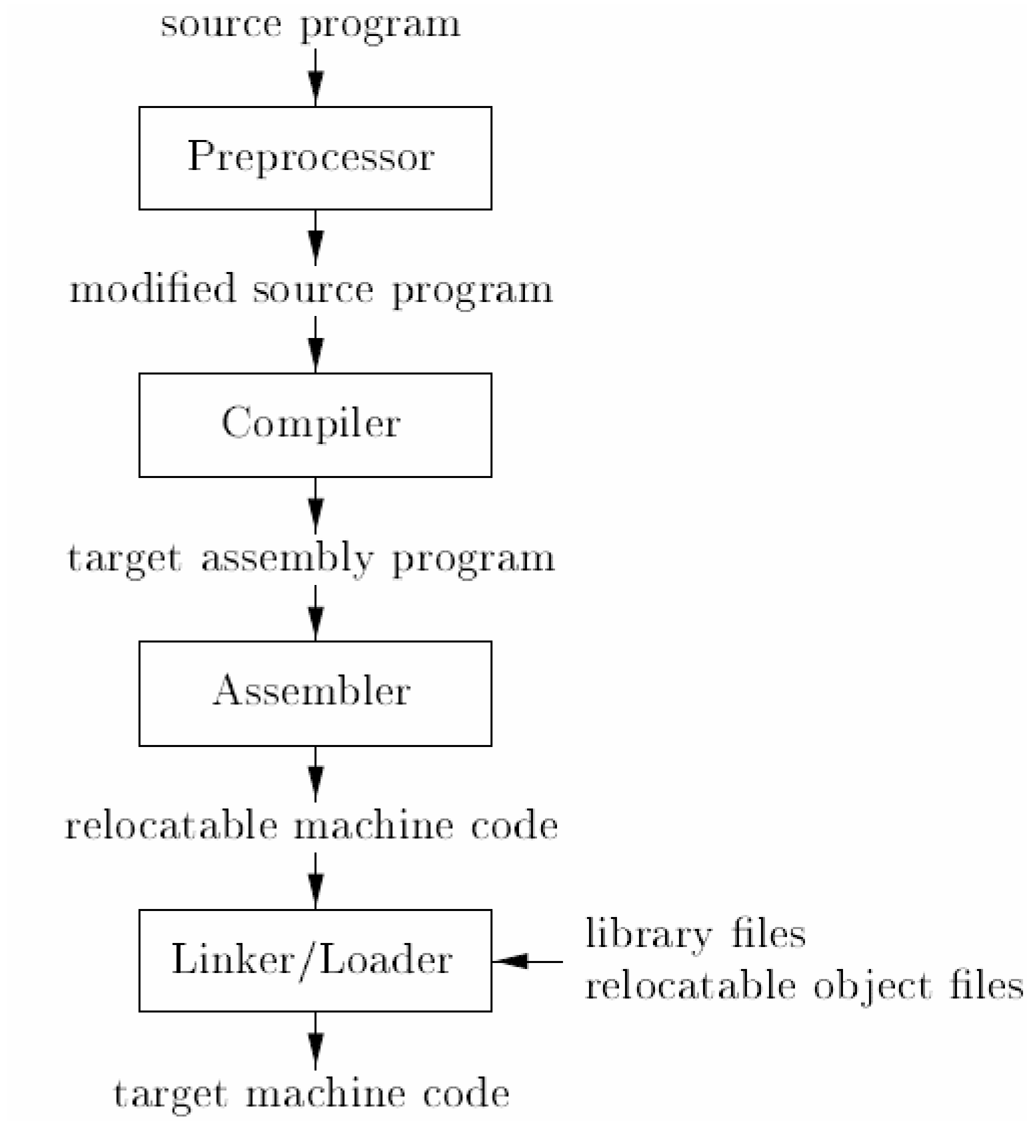
Compiler-Construction Tools

- 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*

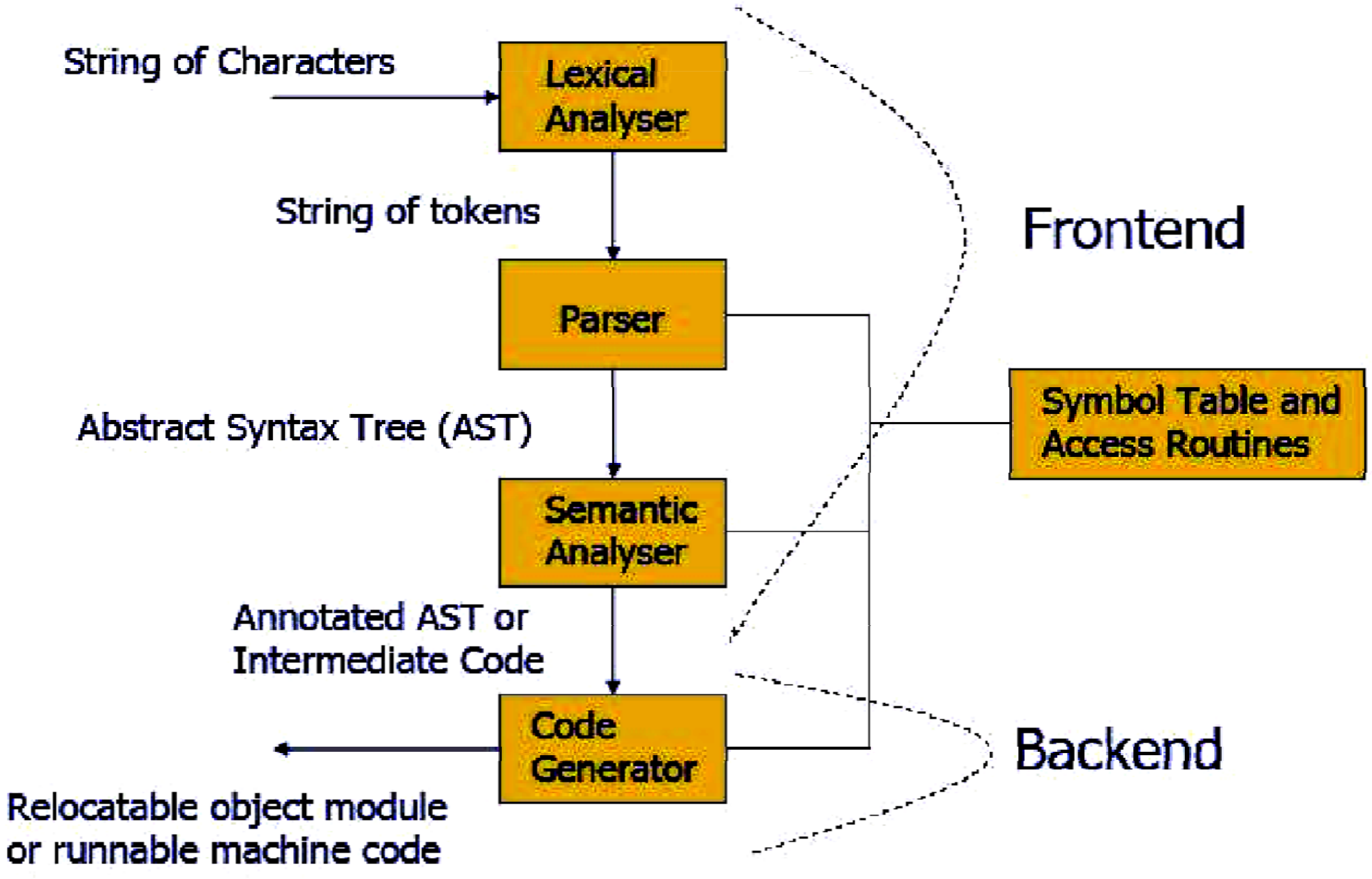
Other Tools that Use the Analysis-Synthesis Model

- *Editors* (syntax highlighting)
- *Pretty printers* (e.g. Doxygen)
- *Static checkers* (e.g. Lint and Splint)
- *Interpreters*
- *Text formatters* (e.g. TeX and LaTeX)
- *Silicon compilers* (e.g. VHDL)
- *Query interpreters/compilers* (Databases)

A language-processing system



Summary - Structure of a compiler


String of Characters
Lexical Analyser
String of tokens
Parser
Abstract Syntax Tree (AST)
Semantic Analyser
Annotated AST or Intermediate Code
Code Generator
Relocatable object module or runnable machine code
Frontend
Symbol Table and Access Routines
Backend

How to learn the course

- Focus on understand the principles deeply
- Notice the relations among the chapters
- Do more exercises , more practices and combine the theories with the labs

Discussion

- What dose a compiler do?
- Why do you need that?
- Name some compilers you have used.

Reference

- Compilers Principles, Techniques & Tools (Second Edition) Alfred Aho., Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Addison-Wesley, 2007
- Coursera Course – Compiler, http://www. Coursera.org
- Stanford Course CS143 by Keith Schwarz, http://cs143.stanford.edu