北京邮电大学
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Chapter 1: Introduction

Yanhui Guo

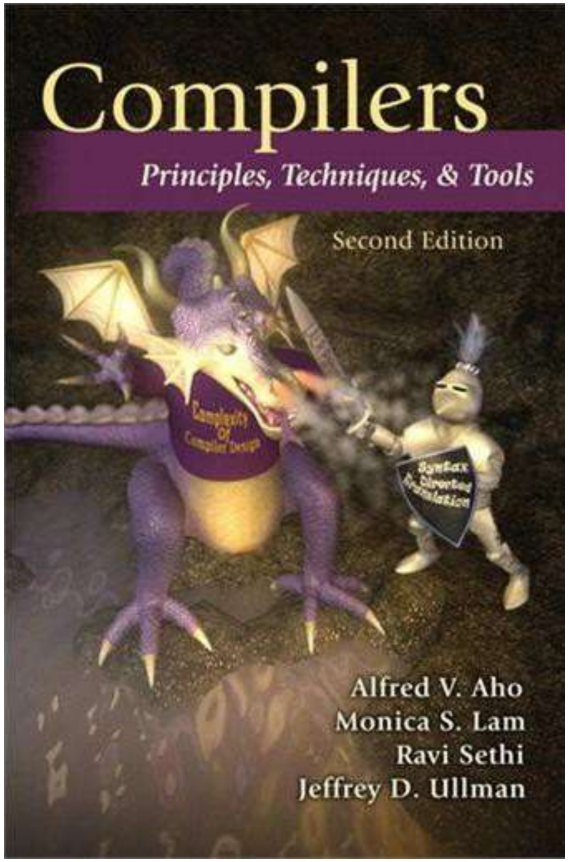
yhguo@bupt.edu.cn

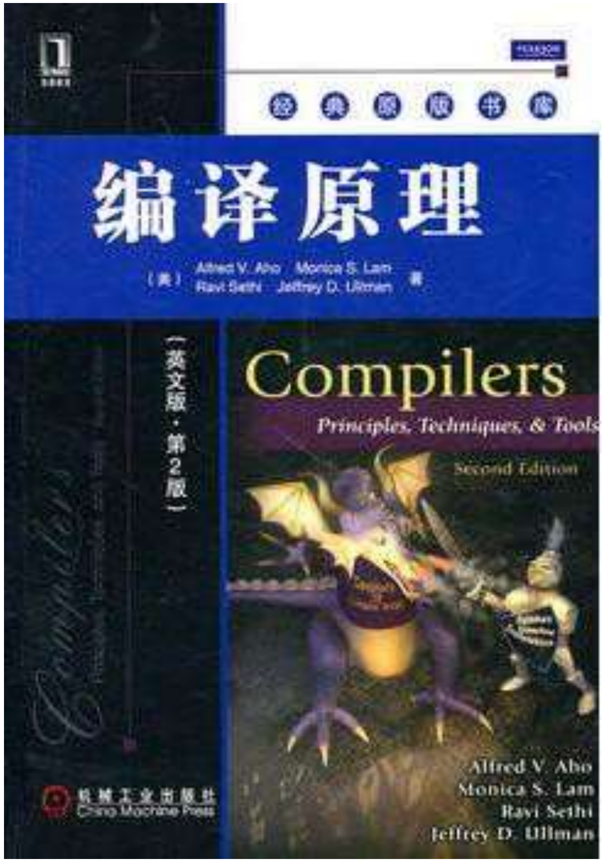
Outline

- Course Information
- Why Study Compilers?
- The Evolution of Programming Languages
- Compiler Structure and Phases
- How to Learn Compilers?

Textbook: The “Dragon Book”

(Second Edition)

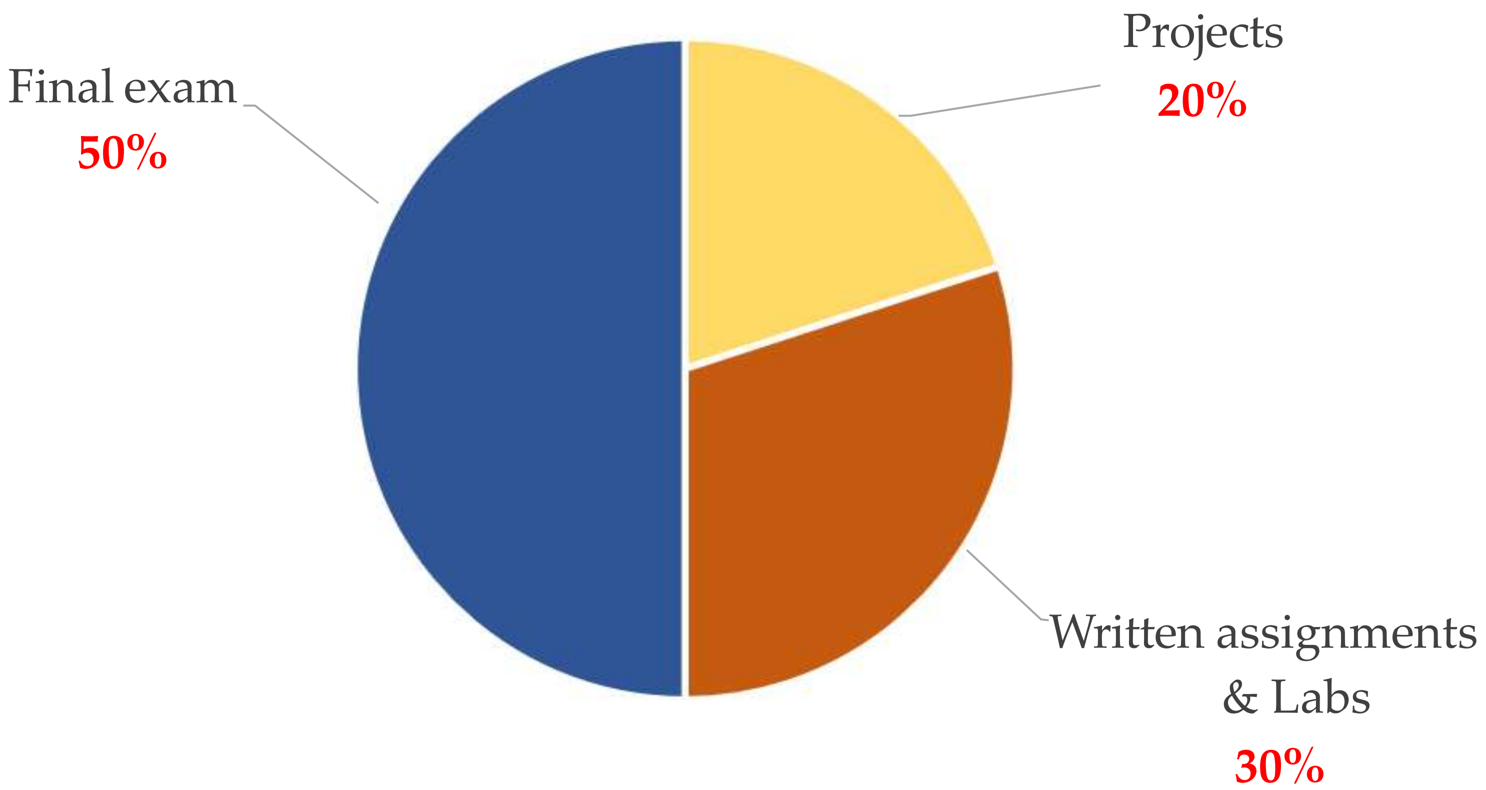




Reference Book for Labs

- 《程序设计语言编译原理》，陈火旺等编著、国防工业出版社，2012.7
- 《编译原理》（第3版）， 王生原等编著， 清华大学出版社，2015.12
- 《编译原理》，蒋宗礼著，高等教育出版社，2010.2
- 《编译原理与技术》，李文生编著，清华大学出版社，2018.6

Marking Scheme



Course Content

Introduction to Compilers (引论)	★☆☆
Lexical Analysis (词法分析)	★★★
Syntax Analysis (语法分析)	★★★
Syntax-Directed Translation (语法制导的翻译)	★★☆
Intermediate-Code Generation (中间代码生成)	★★★
Run-Time Environments (运行时刻环境)	★☆☆
Code Generation (代码生成)	★★☆
Machine-Independent Optimizations (机器无关优化)	★★☆

★ indicates difficulty level, the more the harder

Outline

- Course Information
- Why Study Compilers?
- The Evolution of Programming Languages
- Compiler Structure and Phases
- How to Learn Compilers?

Why Study This "Difficult" Course?

- A fundamental computer science course
- Learn compilation and program analysis techniques
- Learn how to build programming languages
- Course covers both theoretical and practical aspects
  - **Theory:** Lectures and written assignments
  - **Practice:** Labs and programming assignments
- If you want to become a hardcore programmer

Outline

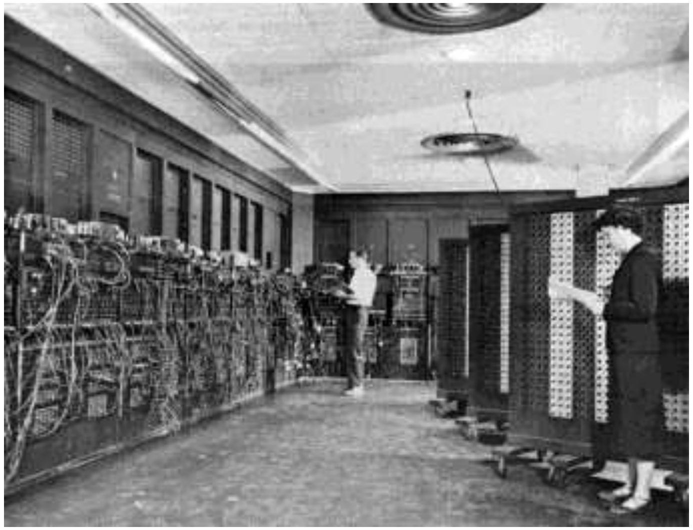
- Course Information
- Why Study Compilers?
- The Evolution of Programming Languages
- Compiler Structure and Phases
- How to Learn Compilers?

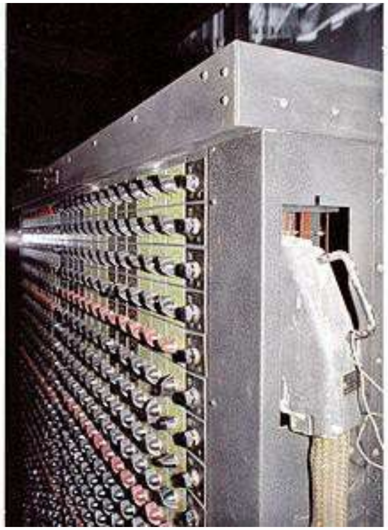
Programming Languages

- Notations for describing computations
- All software is written in some programming language
- Nowadays, there are over 700 programming languages (~250 popular ones)[1]
  - Low-level (低级语言): directly understandable by a computer
  - High-level (高级语言): understandable by human beings, need a translator to be understood by a computer

[1] https://en.wikipedia.org/wiki/List_of_programming_languages

When It All Started …



The first electronic computer ENIAC appeared in 1946. It was programmed in machine language (sequences of 0's and 1's) by setting switches and cables.

https://en.wikipedia.org/wiki/ENIAC

Can You Understand This?

```
00001001001011100110011001101001011011000110010100000100
10010001001101100011001010110001101110100011101010111100
10011001010011000100101110011000110010001000000101001100
11101100011011000110011001001011111011000110110111110110
11010111000001101001011011000110010101100100000101110001
11010000010100010111001110011011001010110001101111010001
10100101101111011011100000010010010001000101110011101000
11001010111100001110100001000100000010100000100100101110
01100001011011000110100101100111011011100010000000011010
00000101000000100100101110011001110110110001101111011000
10011000010110110000100000011011010110000101101001011101
11000001010000001001001011100111010001111001011100000110
01010000100100100000011011010110000101101001011011 10...
```

Assembly Language (Early 1950s)

```
save %sp,-128,%sp
mov 1,%o0
st  %o0,[%fp-20]
mov 2,%o0
st  %o0,[%fp-24]
ld  [%fp-20],%o0
ld  [%fp-24],%o1
add  %o0,%o1,%o0
st  %o0,[%fp-28]
mov 0,%i0
nop
```

- 1st step towards human-friendly languages
- Mnemonic names (助记符) for machine instructions
- Macro instructions for frequently used sequences of machine instructions
- Explicit manipulation of memory addresses and content
- Still low-level and machine dependent

The Move to High-Level Languages

- Disadvantages of assembly language
  - Programming is tedious and slow
  - Programs are not understandable by human beings
  - Programs are error-prone and hard to debug
- High-level programming languages appeared in the second half of the 1950s
  - Fortran: for scientific computation
  - Cobol: for business data processing
  - Lisp: for symbolic computation

Fortran: The 1st High-Level Language

- In 1953, John Backus proposed to develop a more practical alternative to assembly language for programming on IBM 704 mainframe computer

John Backus (1924 – 2007)
American Computer Scientist
ACM Turing Award (1997)

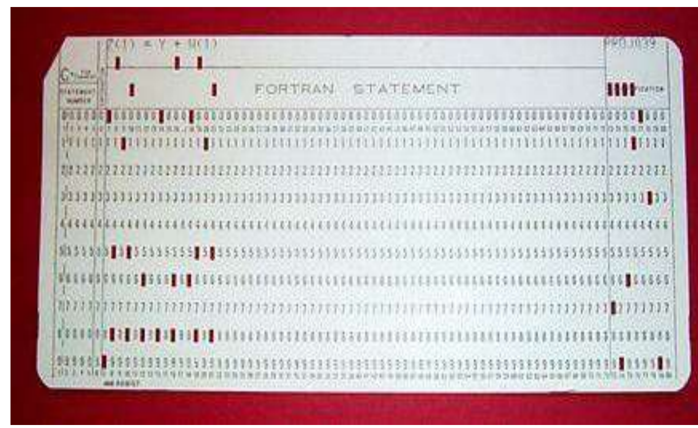
IBM 704 mainframe

Fortran: The 1st High-Level Language

- The 1st Fortran (**For**mula **Tran**slation) compiler was delivered in 1957
- Coding became much faster, 50%+ software was in Fortran in 1958
- **Huge impact**, modern compilers preserve the outline of Fortran I
- Fortran is still widely used today (No. 31, TIOBE Index 2020)

```
C---- THIS PROGRAM READS INPUT FROM THE CARD READER,
C---- 3 INTEGERS IN EACH CARD, CALCULATE AND OUTPUT
C---- THE SUM OF THEM.
  100 READ(5,10) I1, I2, I3
   10 FORMAT(3I5)
      IF (I1.EQ.0 .AND. I2.EQ.0 .AND. I3.EQ.0) GOTO 200
      ISUM = I1 + I2 + I3
      WRITE(6,20) I1, I2, I3, ISUM
   20 FORMAT(7HSUM OF , I5, 2H, , I5, 5H AND , I5,
     *   4H IS , I6)
      GOTO 100
  200 STOP
      END
```

Fortran code example



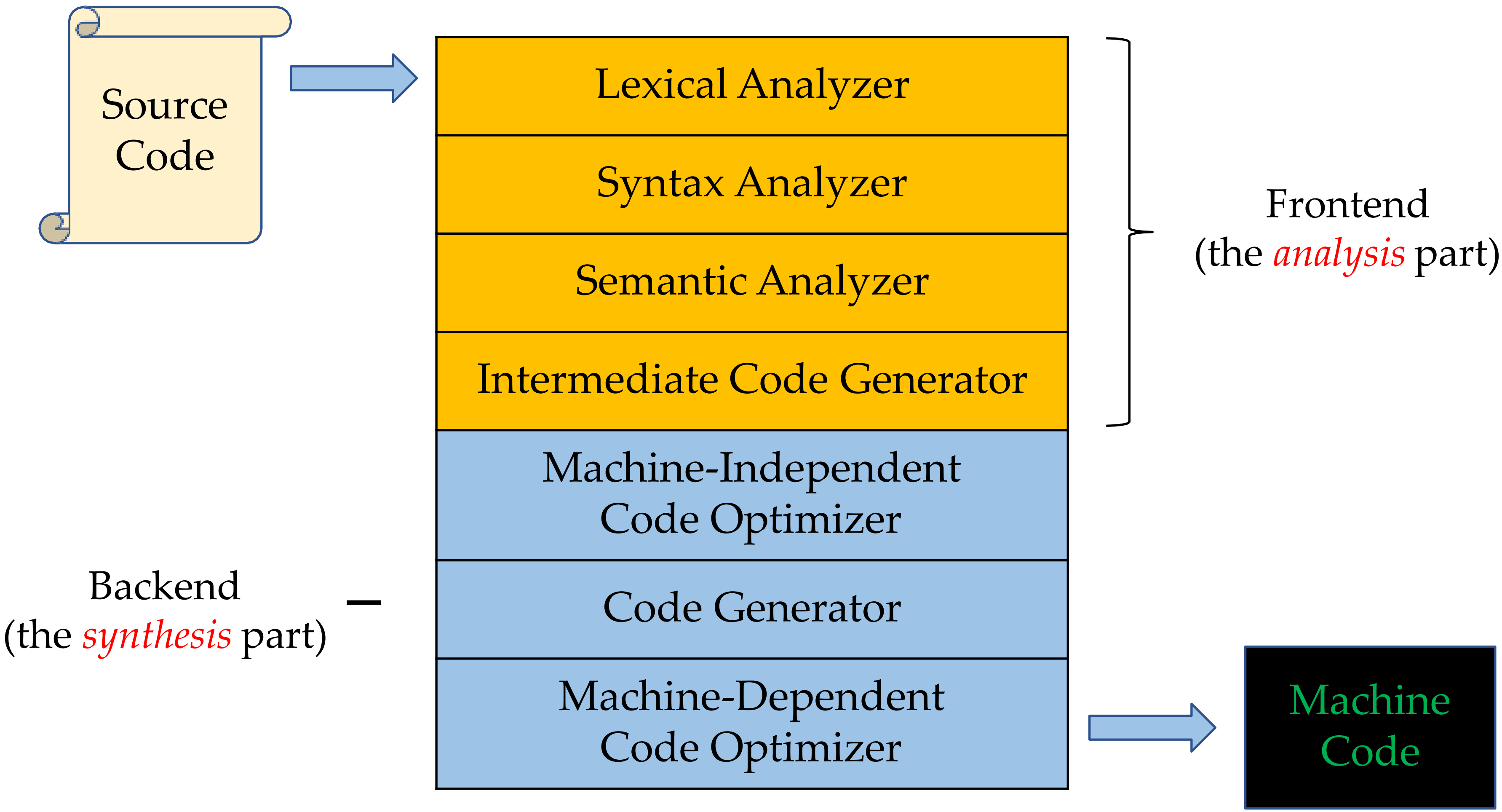
Fortran code on a punch card

http://www.herongyang.com/Computer-History/FORTRAN-Program-Store-on-Punch-Card.html

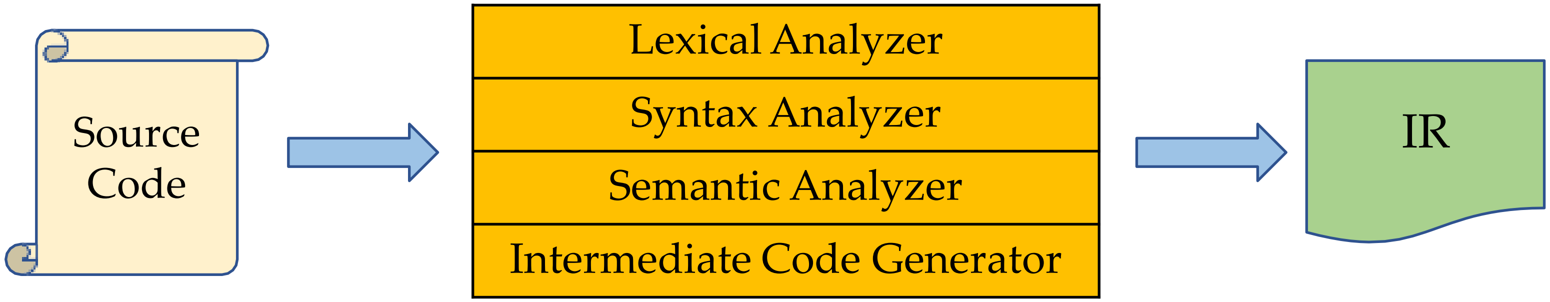
Outline

- Course Information
- Why Study Compilers?
- The Evolution of Programming Languages
- Compiler Structure and Phases
- How to Learn Compilers?

The Structure of a Compiler

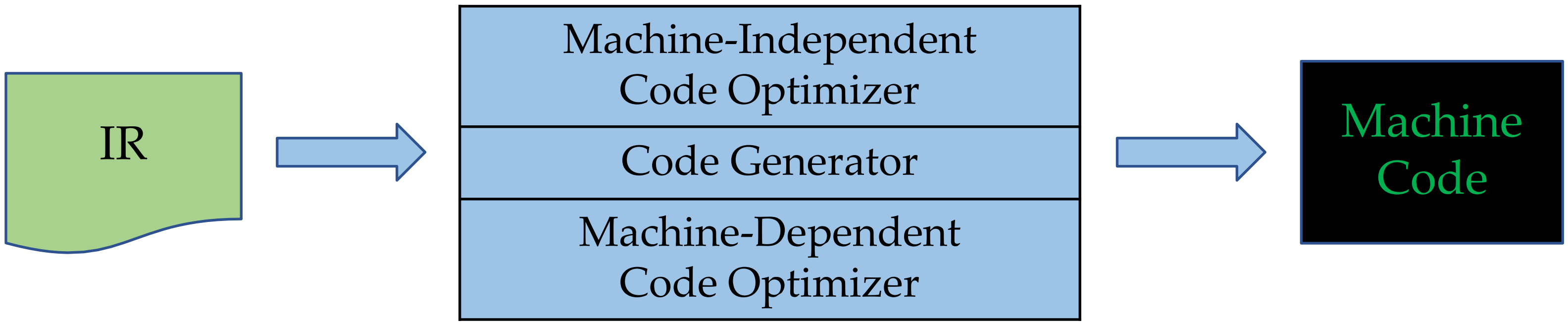


The Frontend（前端） of a Compiler



- Breaks up the source program into constituent pieces and imposes a grammatical structure on them
- Uses the grammatical structure to create an intermediate representation (IR) of the source program
- Collect the information about the source program and stores it in a data structure called *symbol table* (will be passed to backend with IR)

The Backend（后端） of a Compiler



- Constructs the target program (typically, in machine language) from the IR and the information in the symbol table
- Performs code optimizations during the process

Lexical Analysis (Scanning, 词法分析)

Character stream → **Lexical Analyzer** → Token stream

- The lexical analyzer (lexer/tokenizer/scanner) breaks down the source code into a sequence of "lexemes"（词素） or "words"
- For each lexeme, produce a "token"（词法单元） in the form:

<token-name, attribute-value>

token-name: An abstract symbol that is used during syntax analysis

attribute-value: Points to an entry in the symbol table. Info in the table entry is for semantic analysis and code generation.

Lexemes vs. Tokens

- A **lexeme** is a string of characters that is a lowest-level syntactic unit in the programming language
  - "words" and punctuation of the programming language (**instance**)

- A **token** is a syntactic category representing a class of lexemes
  - **In English:** Noun, Verb, Adjective…
  - **In programming language**: Identifier, Keyword, Whitespace… (**pattern**)

https://courses.cs.vt.edu/~cs1104/Compilers/Compilers.070.html

Lexical Analysis (Example)

```
position = initial + rate * 60
```

Lexical Analyzer

```
<id, 1>

<=>

<id, 2>

<+>

<id, 3>

<*>

<60>
```

SYMBOL TABLE

1	position	...
2	initial	...
3	rate	...

Note: <=>, <+>, <*>, <60> are not in the defined form. This is for notational convenience. <=> could have been <assign, -> and <60> could have been <number, 4>.

Lexical Analysis (Analogy)

`position = initial + rate * 60`

Lexical Analyzer

<**id**, 1>

<=>

<**id**, 2>

<+>

<**id**, 3>

<*>

<60>

`BUPT is a good university`

Human brain

<**noun**, “BUPT”>

<**verb**, “is”>

<**article**, “a”>

<**adjective**, “good”>

<**noun**, “university”>

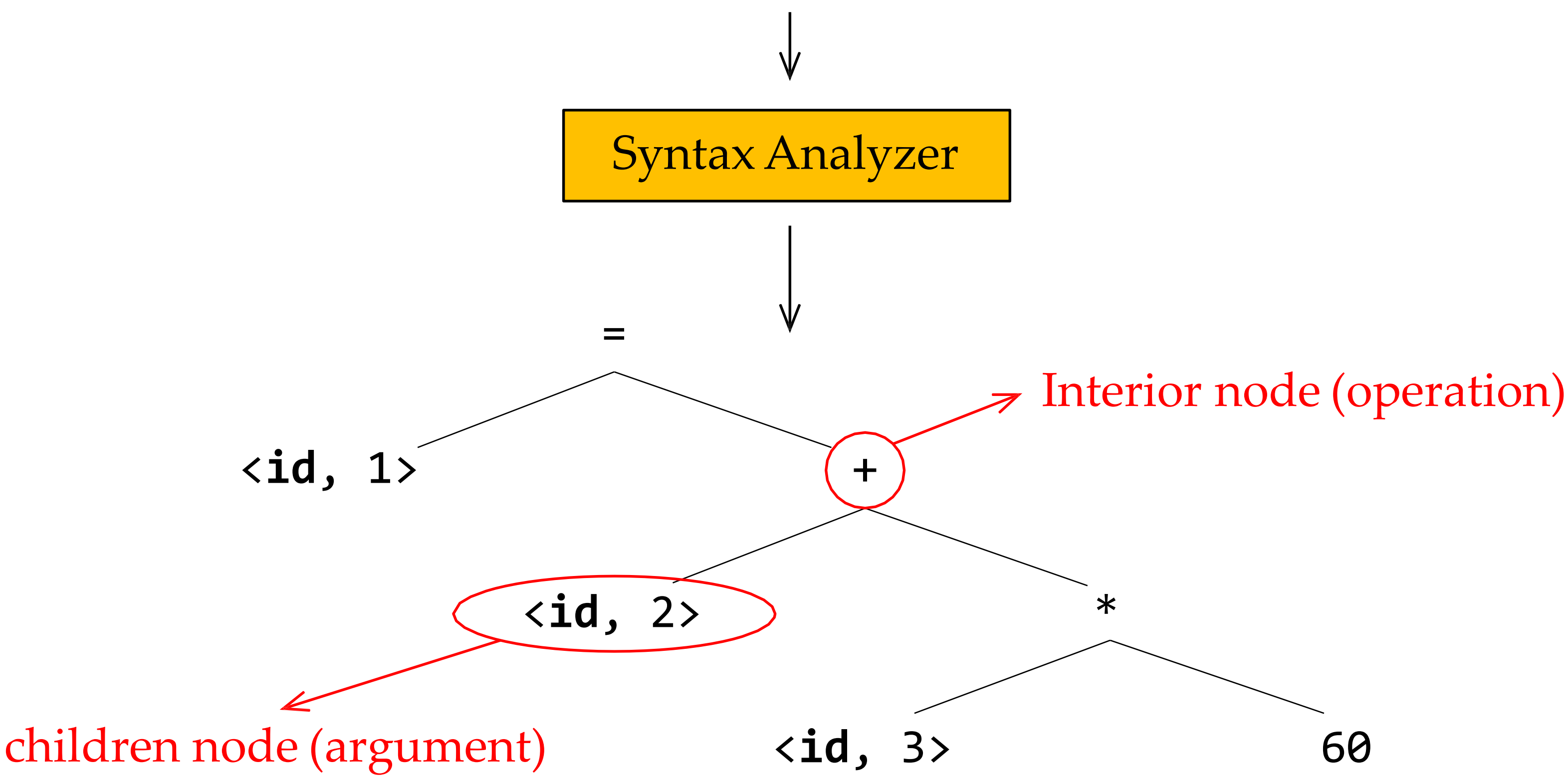
Syntax Analysis (Parsing, 语法分析)

Token stream → Syntax Analyzer → Syntax tree

- The syntax analyzer (parser) uses the token names produced by the lexer to create an intermediate representation that depicts the grammar structure of the token stream, typically a *syntax tree*
- Each interior node represents an operation and the children of the node represent the arguments of the operation

Syntax Analysis (Example)

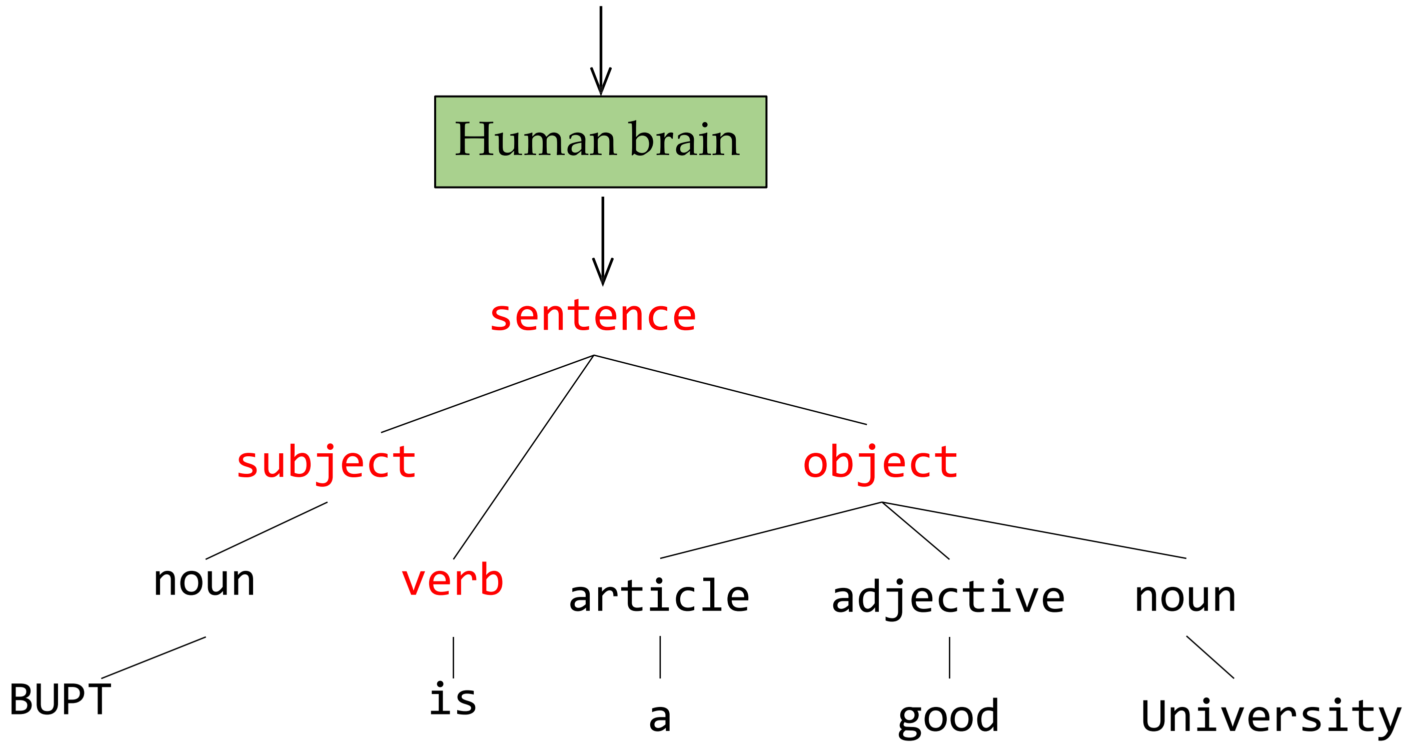
<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>



Syntax Analysis in English

<**noun**, “BUPT”> <**verb**, “is”> <**article**, “a”>

<**adjective**, “good”> <**noun**, “university”>



Semantic Analysis (语义分析)

Syntax tree → **Semantic Analyzer** → Syntax tree

- 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
- Also gathers type information for type checking, type conversion, and intermediate code generation

What is Semantics?

- The syntax of a programming language describes the proper form of its programs

- The semantics of a programming language describes the meaning of its programs, i.e., what each program does when it executes

Semantic Analysis in English

`Jack said Jerry left his assignment at home.`

What does "his" refer to? Jack's or Jerry's?

`Jack said Jack left his assignment at home.`

How many Jacks? Which one left the assignment?

Examples are from Aiken's notes (Stanford CS143)

Semantic Analysis in Programming

- Understanding the meaning of a program is very hard ☹
- Compilers perform only very limited analysis to catch semantic inconsistencies.

```
1. {
2.   int Jack = 3;
3.   {
4.     int Jack = 4;
5.     print Jack;
6.   }
7. }
```

Which value will be printed?

Programming languages define strict rules to avoid ambiguities.

Compiler will bind Jack at line 5 to its inner definition at line 4.

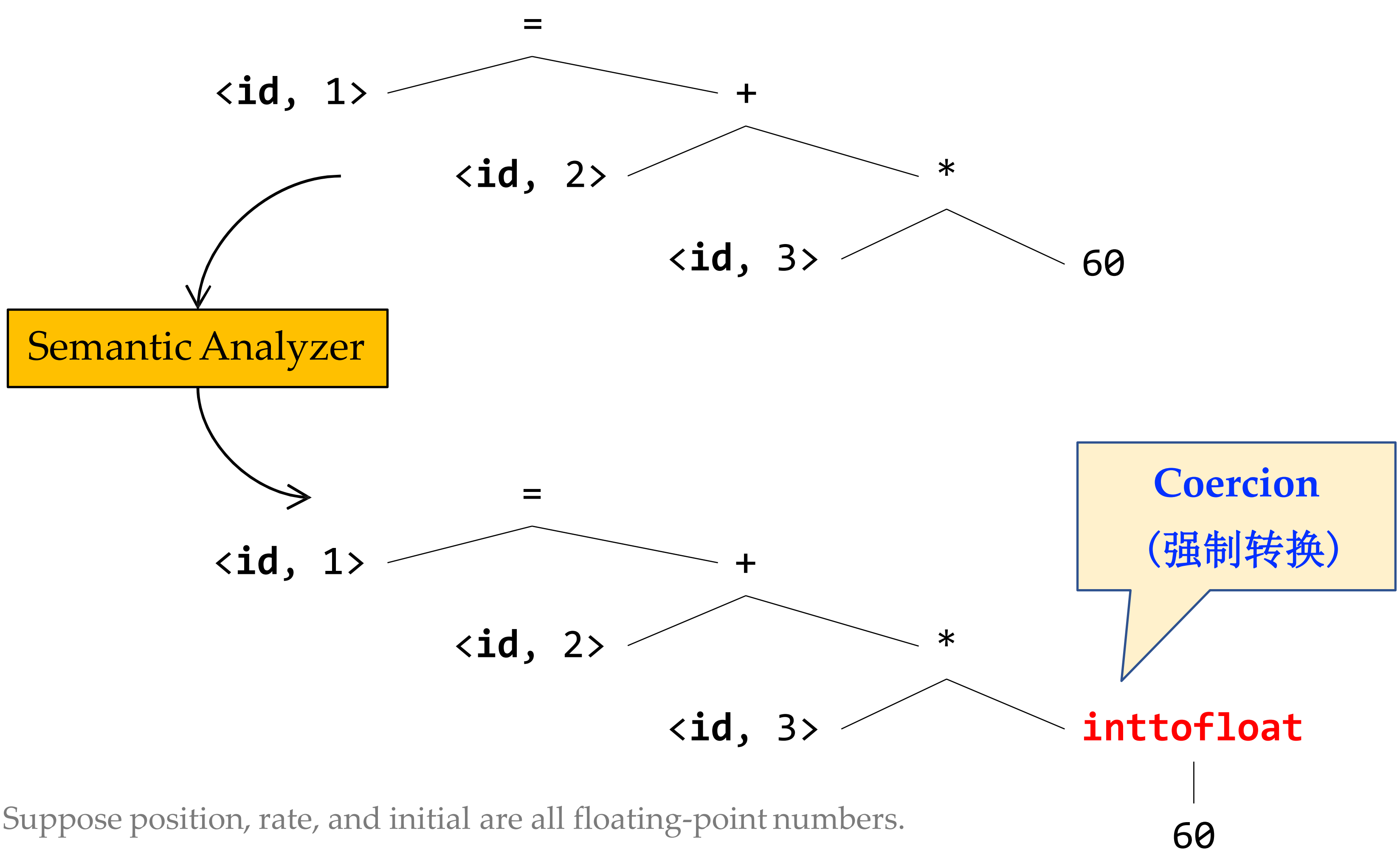
Type Checking (类型检查)

- An important part of semantic analysis is type checking
- Compilers check that each operator has matching operands (of correct types)

Example: Many language definitions require an array index to be an integer.

Compilers should report an error if the argument of an array element access operation is a floating-point number.

Semantic Analysis (Example)



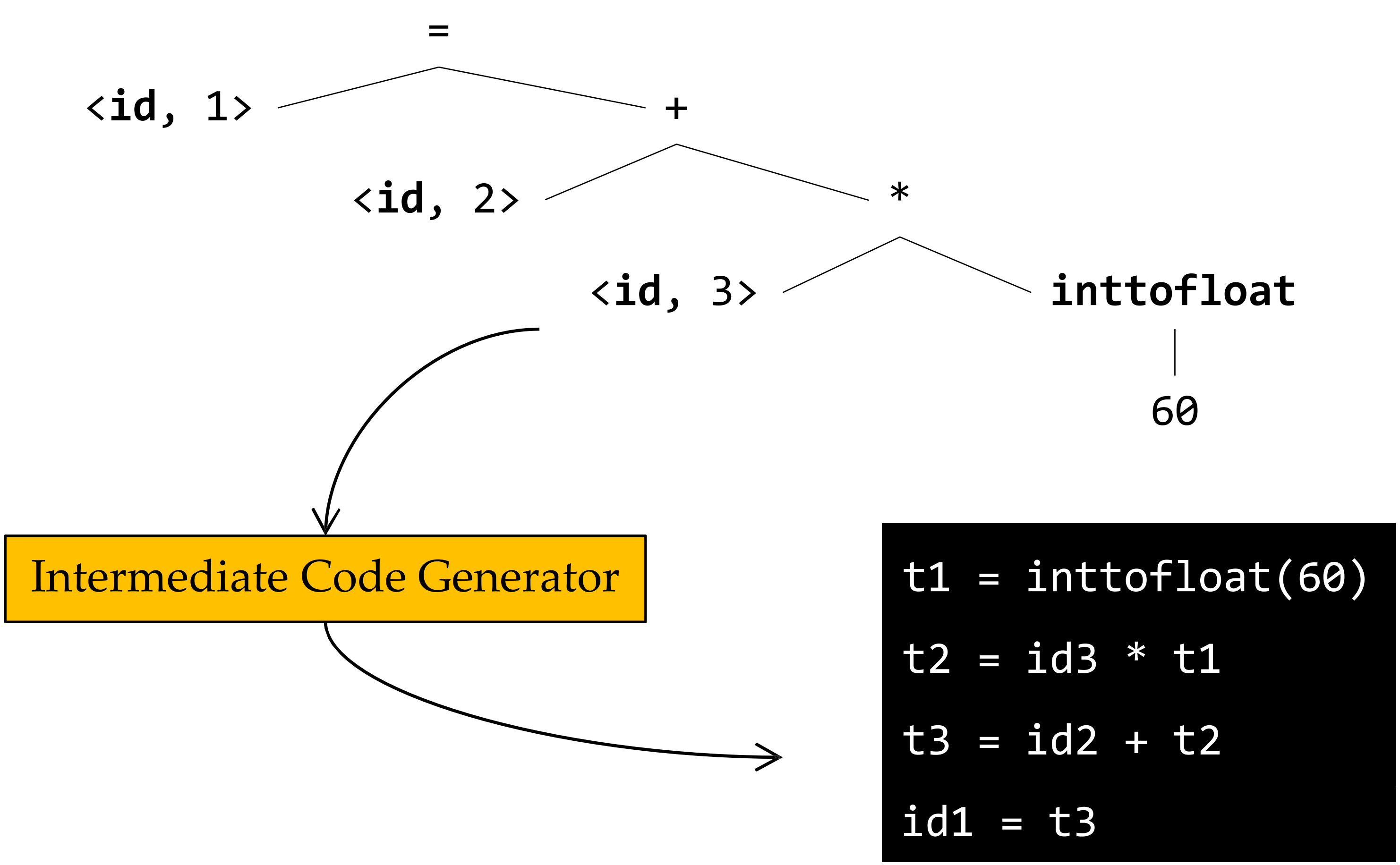
Suppose position, rate, and initial are all floating-point numbers.

Intermediate Code Generation (中间代码生成)

Syntax tree → **Intermediate Code Generator** → IR (in three-address code)

- After semantic analysis, compilers generate an intermediate representation, typically *three-address code (三地址码)*
  - Assembly-like instructions with three operands per instruction
  - Each operand acts like a register
  - Each assignment instruction has at most one operator on the RHS
  - Easy to translate into machine instructions of the target machine

Three-Address Code Example



Machine-Independent Code Optimization (机器无关的代码优化)

IR → Machine-Independent Code Optimizer → IR

- Akin to article editing/revising in English
- Improve the intermediate code for better target code
  - Run faster
  - Use less memory
  - Shorter code
  - Consume less power …

Code Optimization (Example)

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

1. 60 is a constant integer value. Its conversion to floating-point can be done once and for all at compile time
2. t3 is only used for value transmitting

Optimization

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generation (代码生成)

IR → Code Generator → Target-machine code

- Map IR to target language, analogous to human translation
- It is crucial to allocate register and memory to hold values

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generation →

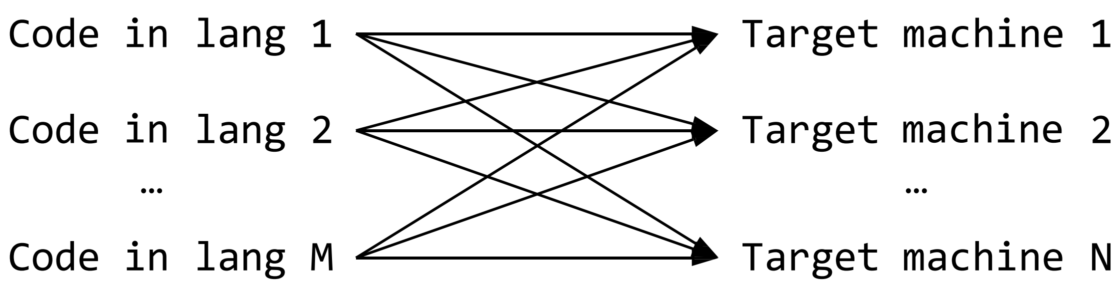
```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

Symbol Table Management

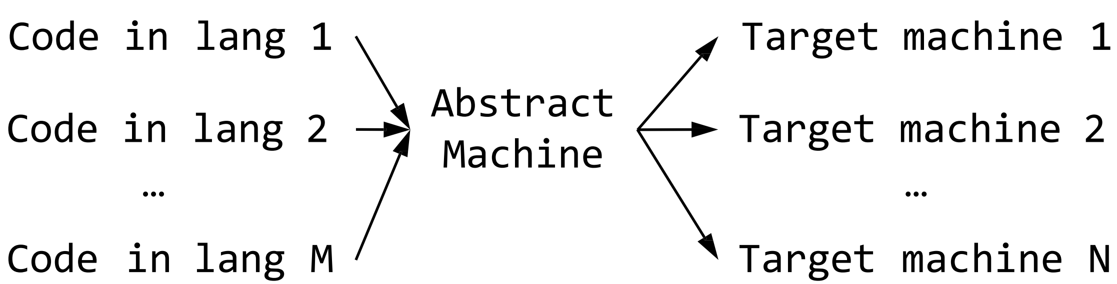
- Performed by the frontend, symbol table is passed along with the intermediate code to the backend
- Record the variable names and various attributes
  - storage allocated, type, scope
- Record the procedure names and various attributes
  - the number and type of arguments
  - the way of passing arguments (by value or by reference)
  - the return type

Intermediate Language (IL)

- Intermediate code is in IL (e.g., three-address code)
- A good IL eases compiler implementation



M * N compilers without a good IL



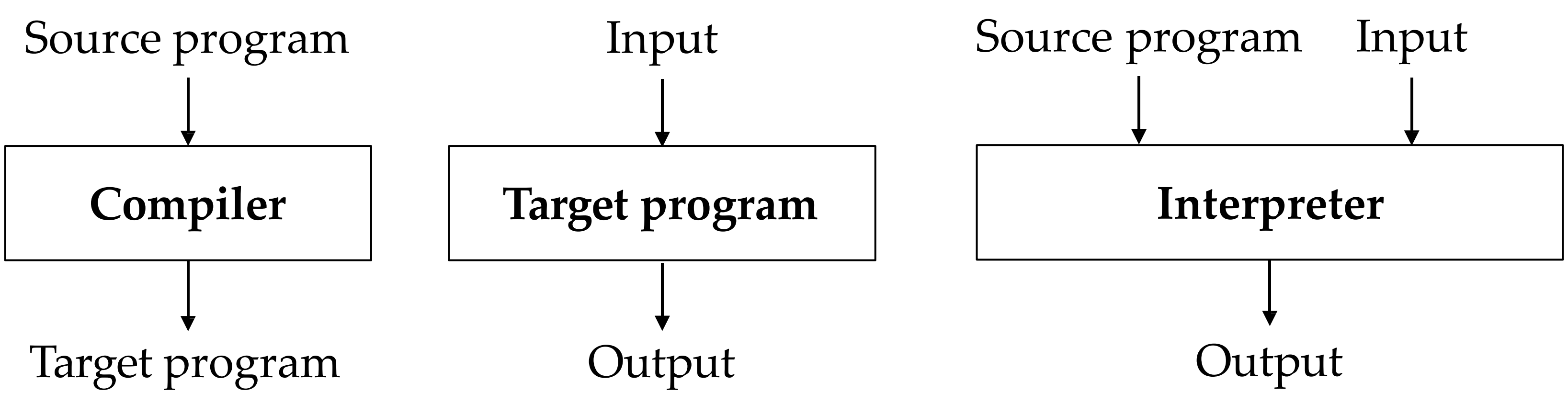
M + N compilers with a good IL

Compilers vs. Interpreters

1

A complier translates source programs written in high-level languages into machine codes that can run directly on the target computer.

An interpreter directly executes each statement in the source code, without requiring the program to have been compiled into machine codes.



Compilers vs. Interpreters

2

Interpreters often take less time to analyze the source code: they simply parse each statement and execute it (e.g., Python code).

In comparison, compilers typically analyze the relationships among statements (e.g., control and data flows) to enable optimizations.

3

Interpreters continue executing a program until the first error is met, in which case they stop.

For compiled languages, programs are executable only after they are successfully compiled.

Outline

- Course Information
- Why Study Compilers?
- The Evolution of Programming Languages
- Compiler Structure and Phases
- How to Learn Compilers?

Group Chat for Course

Feishu（飞书）

编译原理_2023_网安
北京邮电大学

编译原理_2023_信安
北京邮电大学

For
- Handling queries
- Sharing courseware
- Sending some notifications

About Assignments

Assignments types:

- **Written assignments**
- **Labs**
- **Projects**

How to view assignments ?

Feishu --- 云文档 --- 共享空间 --- “编译原理-课程资源共享”

How to submit assignments ?

Teaching cloud platform（教学云平台）

（信息门户 --- 系统直通车 --- 教学云平台 --- 云邮教学空间）

About Labs & Projects

Labs:

- **Standalone exercises involving the implementation of different concepts**
- **Providing support for projects**

Projects:

- **More comprehensive tasks**
- **Applying the knowledge gained from labs to a practical compiler development**

Programming practices

Complement each other

For building a compiler

Projects Overview

- **Goal**

Design & implement a compiler for BUPT Programming Language(BPL).

BPL is a C-like Programming language, but it removes most advanced features, while it keeps general-purpose programming language functionalities.

Projects Overview

- **Target**

The compiler reads a BPL source code, translates it into a particular intermediate representation (IR).

During this semester, you will learn how to build your own compiler from scratch. You are expected to realize it through lexical and syntax analysis, semantics checking, intermediate code generation and target code generation(optional).

Projects Overview

- **Stages (approximately 3 weeks each):**
  - ➢ Lexical analysis & syntax analysis
  - ➢ Semantic analysis
  - ➢ Intermediate code generation
  - ➢ Target code generation (optional)

BPL Specification

- **token.txt**
  Lexical specification, defines valid tokens in BPL.

- **syntax.txt**

Grammar rules, specifies the syntactical structure.

BPL Specification

- **Lexical specification(a snippet)**

```
INT     -> /* integer in 32-bits (decimal or hexadecimal) */
FLOAT   -> /* floating point number (only dot-form) */
CHAR    -> /* single character (printable or hex-form) */
ID      -> /* identifier */
TYPE    -> int | float | char
STRUCT  -> struct
IF      -> if
ELSE    -> else
WHILE   -> while
RETURN  -> return
DOT     -> .
SEMI    -> ;
```

BPL Specification

- **Grammar rule (a single production)**

```
Stmt -> Exp SEMI
      | CompSt
      | RETURN Exp SEMI
      | IF LP Exp RP Stmt
      | IF LP Exp RP Stmt ELSE Stmt
      | WHILE LP Exp RP Stmt
```

Development Environment

The necessary dependencies as follow：

- GCC version 7.5.0
- GNU Make version 4.1
- GNU Flex version 2.6.4
- GNU Bison version 3.0.4
- Python3 version 3.6.9

编译原理_2023_网安
北京邮电大学

编译原理_2023_信安
北京邮电大学

To reduce the burden, we choose docker for environmental configuration.

- Scan the Feishu QR code to enter the course group chat
- Obtain the environment configuration tutorial on the shared space.（共享空间）

Assignment

Lab 1

Note:
- The lab 1 has been published on the teaching cloud platform.
- Before conducting lab 1, it is necessary to deploy the docker experimental environment.
- Submit through teaching cloud platform.

Dealine:
- September 17, 2023, 24:00