南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Chapter 1: Course Introduction

Yepang Liu

liuyp1@sustech.edu.cn

Outline

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

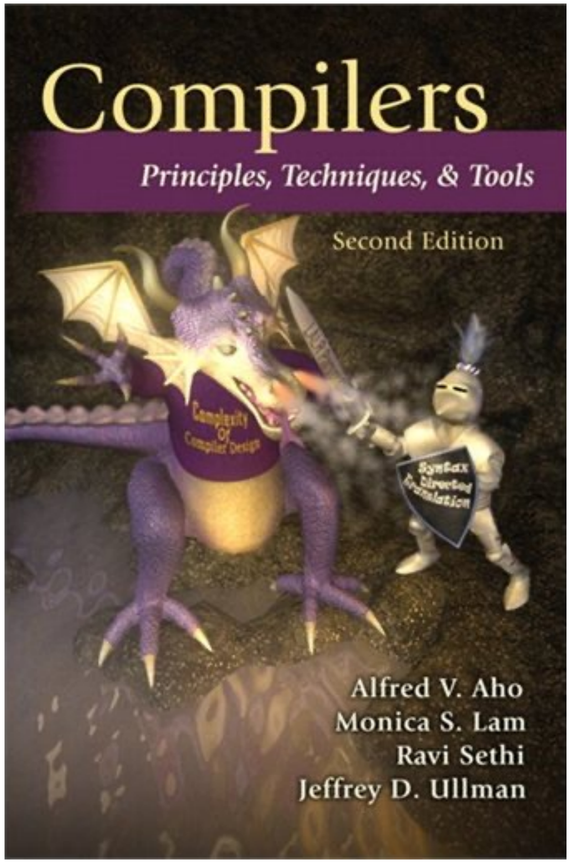
The Teaching Team

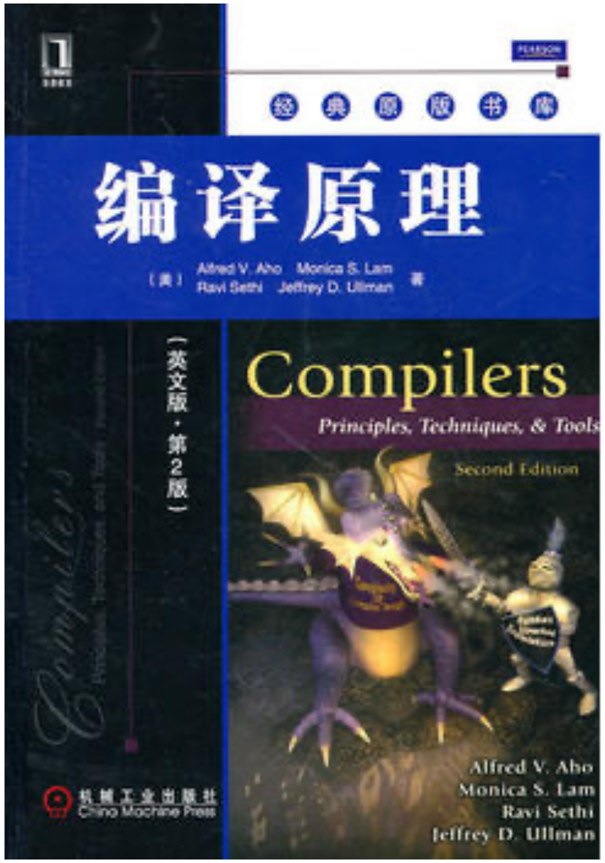
- **Instructor:** Yepang Liu (刘烨庞)
  - Email: liuyp1@sustech.edu.cn
  - Office: Room 609, CoE Building (South)

- **TA:** Yujia Fan (樊宇佳), Yige Chen (陈一戈), Ziming Rui (芮梓铭)
  - Email: 12431253, 12432744, 12432659@mail.sustech.edu.cn
  - Office: Room 650A, CoE Building (South)

- **Communication:**
  - Emails: typically replied within 24 hours
  - Office hour (Yepang Liu): 2:30pm – 4:30pm, every Monday
  - Office hour (Yige Chen): 9:00am – 10:40am, every Friday

Textbook: The "Dragon Book"




经典原版书库
编译原理
（美）Alfred V. Aho Monica S. Lam Ravi Sethi Jeffrey D. Ullman 著
（英文版・第2版）
Compilers
Principles, Techniques, & Tools
Second Edition
机械工业出版社 China Machine Press
Alfred V. Aho
Monica S. Lam
Ravi Sethi
Jeffrey D. Ullman

Available at: Yidan Library 2nd Floor ;
TP314 /E7.v1 第3排B面

40 times cheaper than the original edition
~80 ¥ on 京东 (Buy one! It's worth the money ☺.)

Textbook: Chinese Version

计算机科学丛书
第2版
编译原理
(美) Alfred V. Aho Monica S. Lam Ravi Sethi Jeffrey D. Ullman 著
赵建华 郑滔 戴新宇 译
Compilers
Principles, Techniques, & Tools
Second Edition
Compilers
Principles, Techniques and Tools
Second Edition
机械工业出版社
China Machine Press

南京大学赵建华、郑滔、戴新宇译

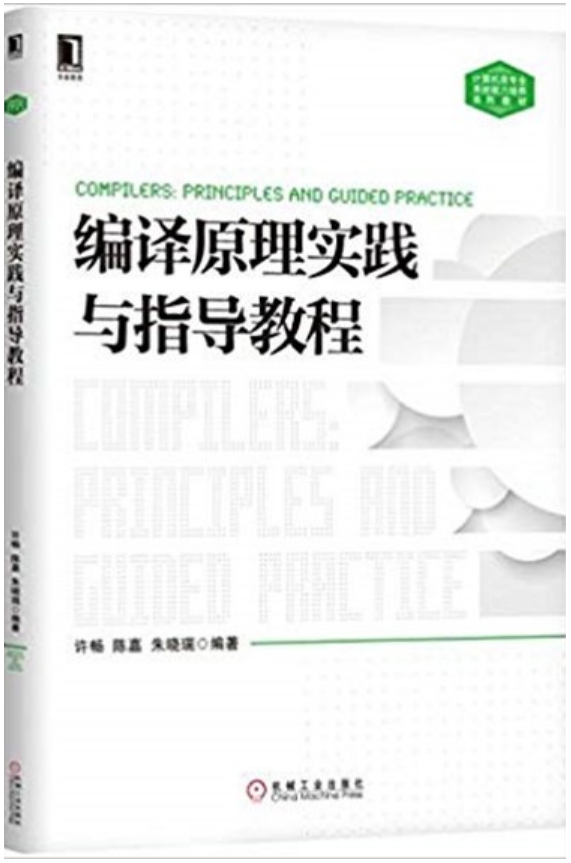
Available at:

Yidan Library 2nd Floor Reading Space ;

TP314 /4.v2 第8排A面

~60 ¥ on 京东

Reference Book for Labs



南京大学许畅、陈嘉、朱晓瑞编著

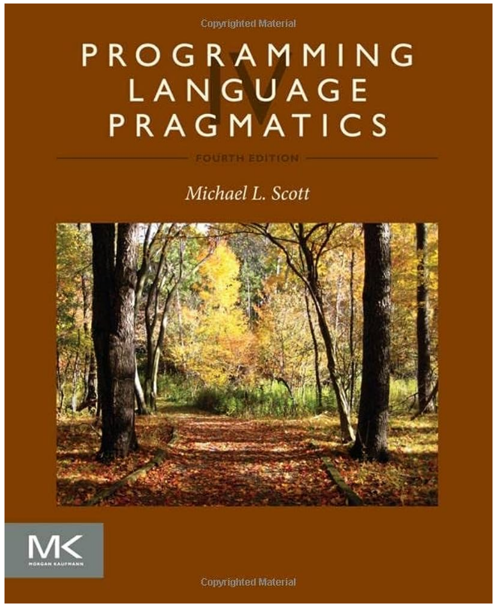
Available at:

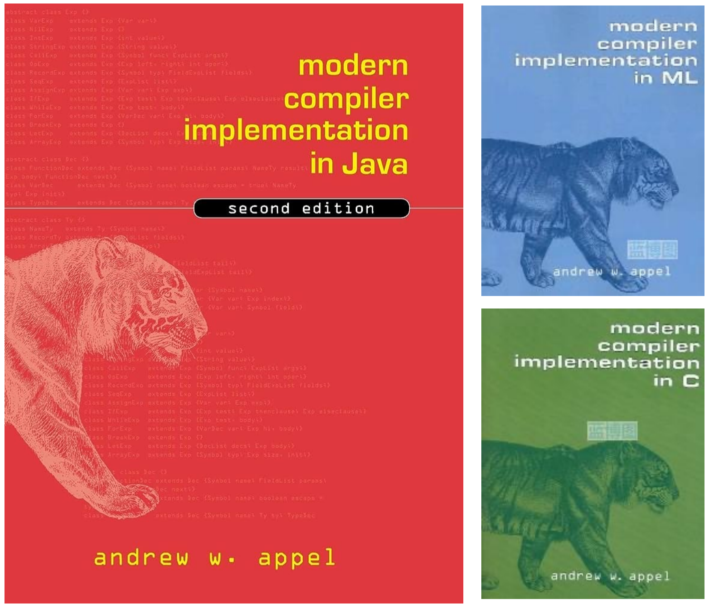
Yidan Library 2nd Floor Reading Space ;

TP314 /18 第8排A面

~100 ¥ on 京东

Other Reference Books




modern compiler implementation in Java
second edition
andrew w. appel
modern compiler implementation in ML
andrew w. appel
modern compiler implementation in C
andrew w. appel

Available at:

Yidan Library 2nd Floor

TP312 /E 5.v1 第3排B面

Available at:

Yidan Library 1st Floor

TP312JA/E207 第29排A面

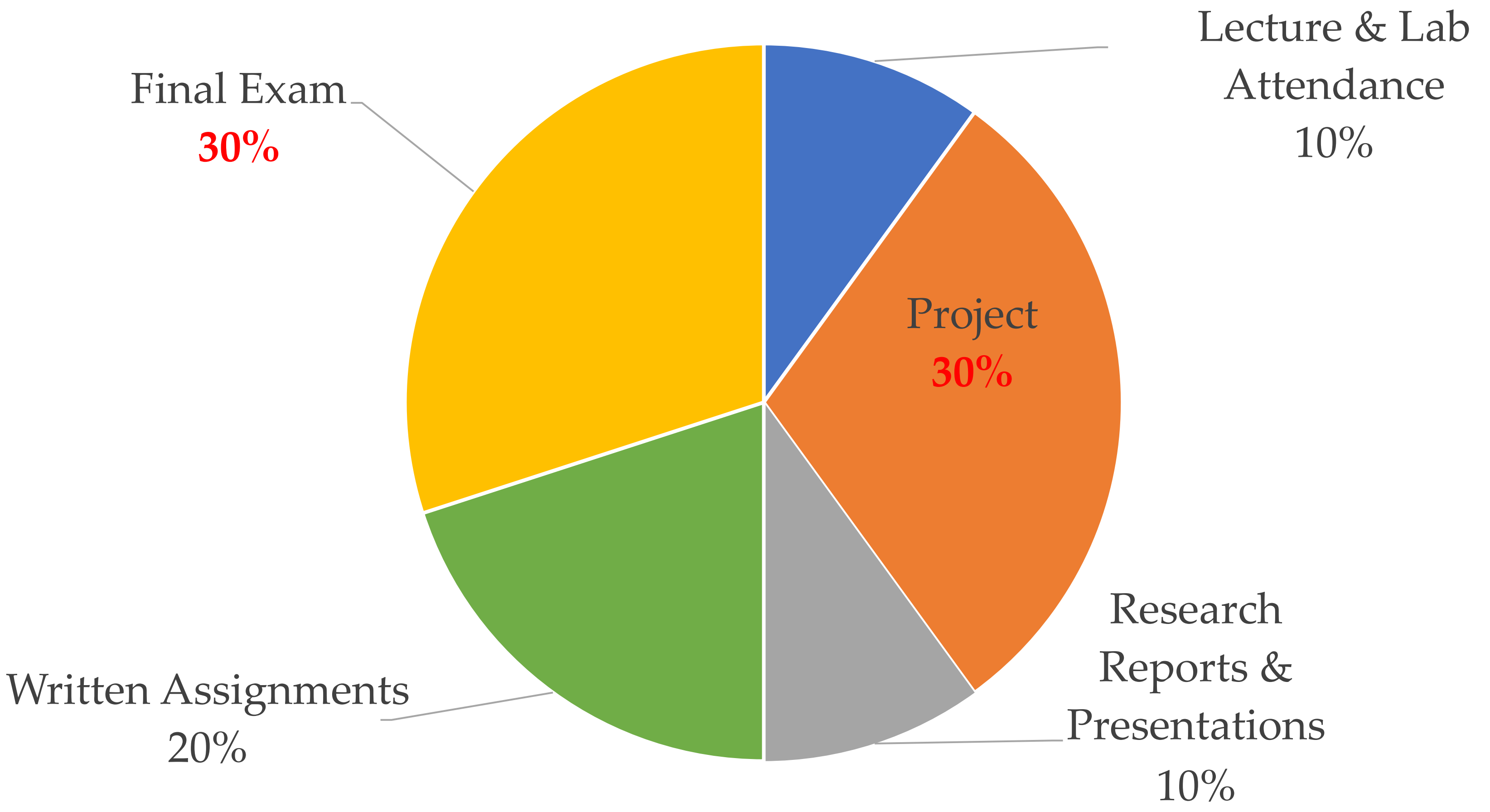
Other versions: C, ML

Course Websites

- Announcements, lecture/lab notes, written assignments, sample answers, etc. are available on Blackboard
  - Registered students will be automatically added to the site

- Other lab/project materials (e.g., project notes, code, test cases) are available on GitHub
  - https://github.com/sqlab-sustech/CS323-2024F

Marking Scheme (Tentative)



Course Content

★ indicates difficulty level, the more the harder

Introduction to Compilers (引论)	★☆☆
Regular Expressions & Context-Free Grammars (正则表达式与上下文无关文法)	★★☆
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)	★★☆

Why Study This 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:** Lab exercises and projects
- If you want to become a professional programmer

Outline

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

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

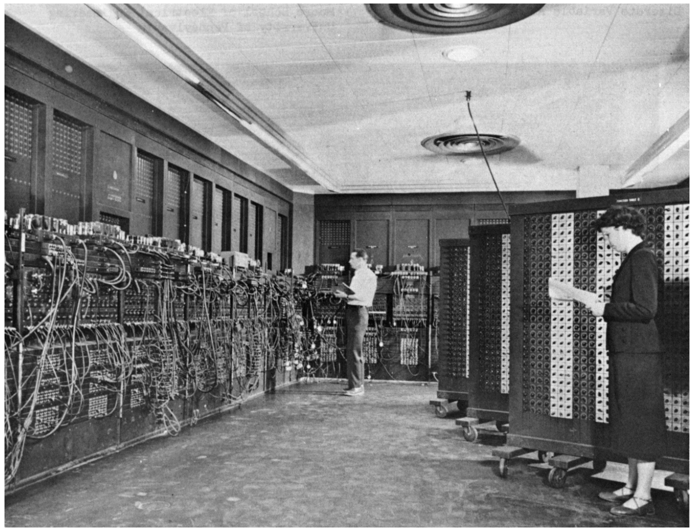
Top Programming Languages

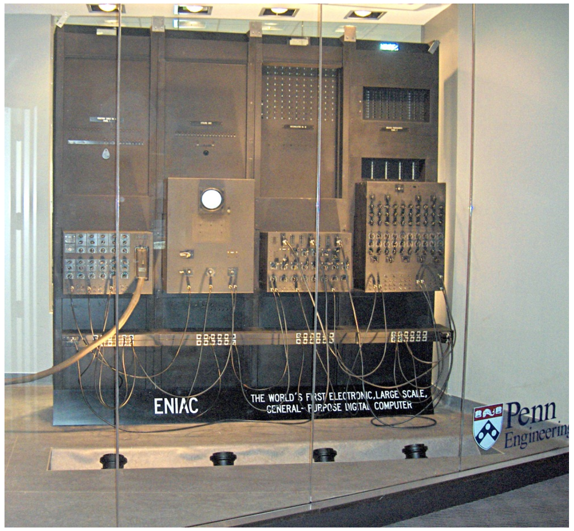
- Winners in the past 20 years according to TIOBE index[1]:
  - Python: 2021, 2020, 2018, 2010, 2007
  - C: 2019, 2017, 2008
  - C#: 2023, 2023
  - Go: 2016, 2009
  - Java: 2015, 2005
  - Objective-C: 2012, 2011
  - C++: 2022; JavaScript: 2014; Transact-SQL: 2013; Ruby: 2006; PHP: 2004

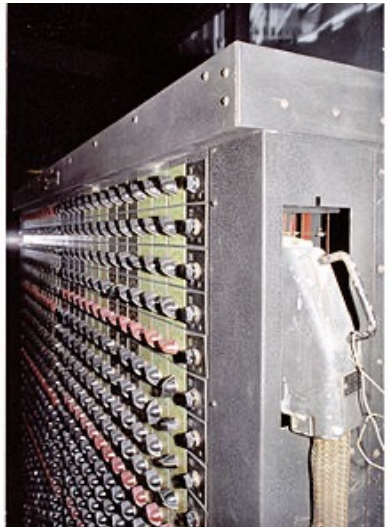
[1] A measure of popularity of programming languages calculated from the number of search engine results for queries containing the name of the language. See the index at https://www.tiobe.com/tiobe-index/.

When It All Started …

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




ENIAC
THE WORLD'S FIRST ELECTRONIC, LARGE SCALE, GENERAL-PURPOSE DIGITAL COMPUTER
Penn Engineering



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.

* The **Atanasoff–Berry computer** (**ABC**) was the first automatic electronic digital computer. It appeared a few years earlier than ENIAC, but it was neither programmable nor Turing-complete. It was designed only to solve systems of linear equations, not for general purposes.

Can You Understand This?

```
0000100100101110011001100110100101101100011001010000100
1001000100110110001100101011000110111010001110101011100
1001100101001100010010111001100011001000100000101001100
1110110001101100011001100100101111101100011011011110110
1101011100000110100101101100011001010110010000101110001
11010000001010001011100111001101100101011000110111010001
1010010110111101101110000010010010001000101110011101000
1100101011110000111010000100010000010100000100100101110
0110000101101100011010010110011101101110001000000011010
0000010100000100100101110011001110110110001101111011000
1001100001011011000010000001101101011000010110100101101
1100000101000001001001011100111010001111001011100000110
0101000010010010000000110110101100001011010010110111 0...
```

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)

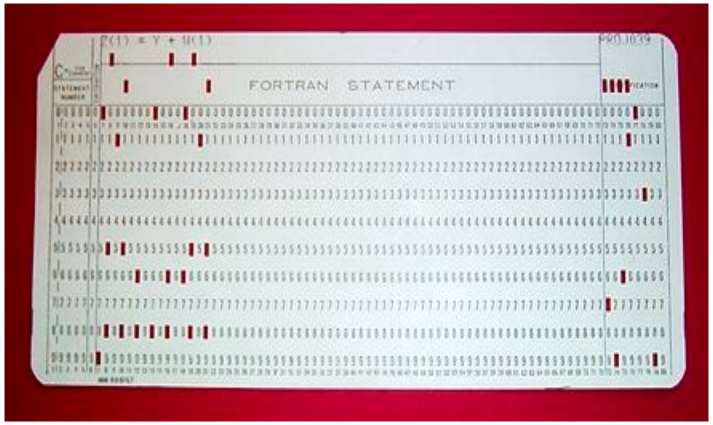
IBM 704 mainframe

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 used today (No. 10, TIOBE Index August 2024)

```
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



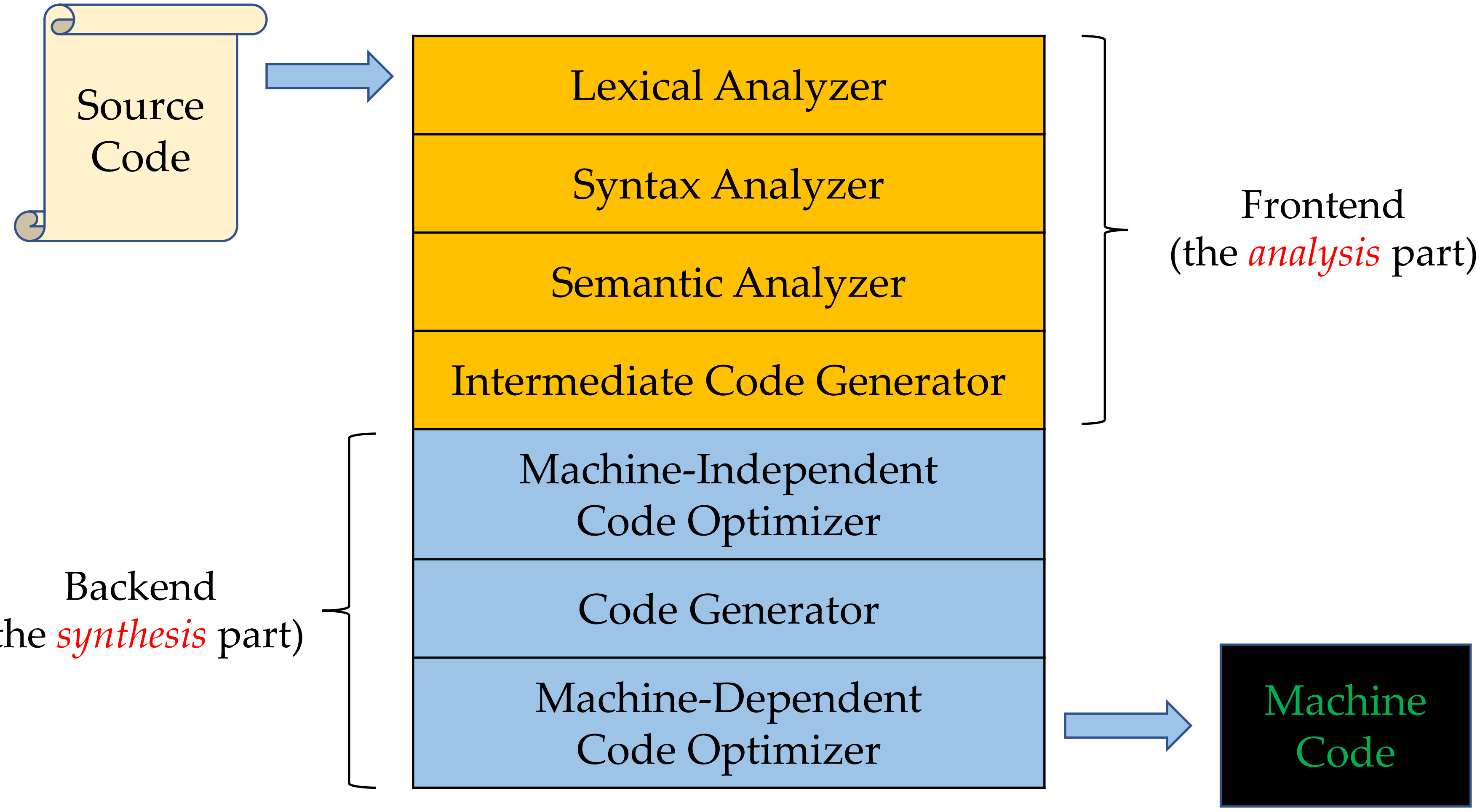
Fortran code on a punch card

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

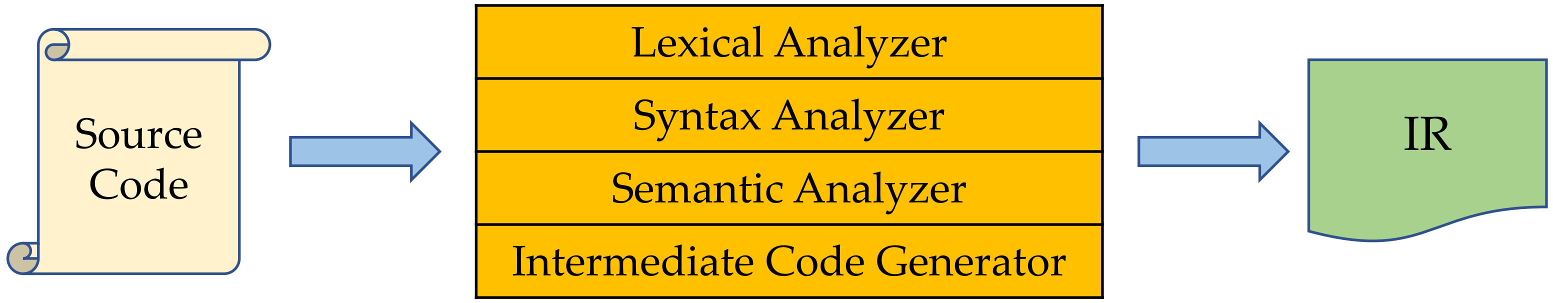
Outline

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

The Structure of a Compiler

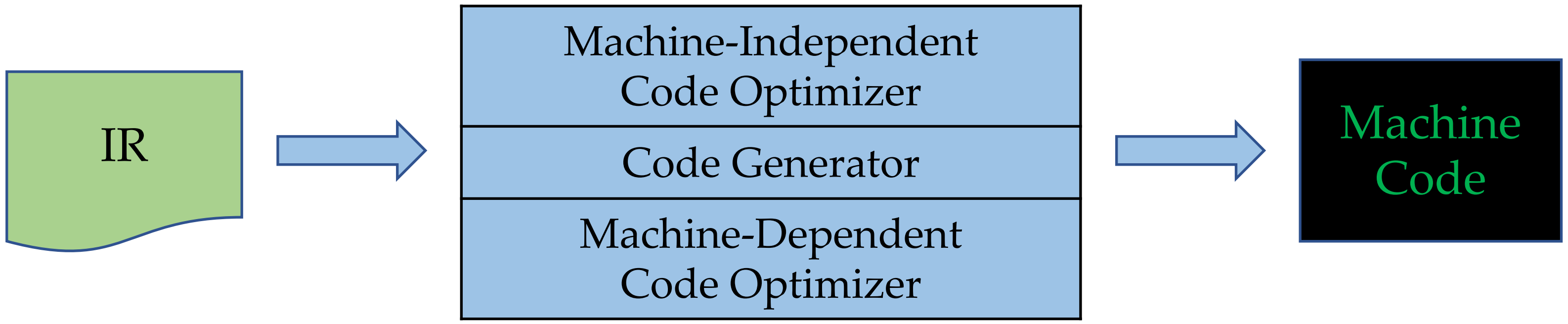

Source Code
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generator
Machine-Independent Code Optimizer
Code Generator
Machine-Dependent Code Optimizer
Machine Code
Frontend (the analysis part)
Backend (the synthesis part)

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


IR
Machine-Independent Code Optimizer
Code Generator
Machine-Dependent Code Optimizer
Machine Code

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

* Lexing and parsing are most complex and expensive in the early days, while in today, optimization dominates all other phases and lexing and parsing are very cheap.

Lexical Analysis (Scanning, 词法分析)

$$\begin{tikzcd}
\text{Character stream} \arrow[r] & \boxed{\text{Lexical Analyzer}} \arrow[r] & \text{Token stream}
\end{tikzcd}$$

- 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>
```

$$\begin{tikzcd}
\texttt{token-name} \arrow[d] & \texttt{attribute-value} \arrow[d] \\
\boxed{\text{An abstract symbol that is used during syntax analysis}} & \boxed{\text{Points to an entry in the symbol table.}}
\end{tikzcd}$$

An abstract symbol that is used during syntax analysis

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

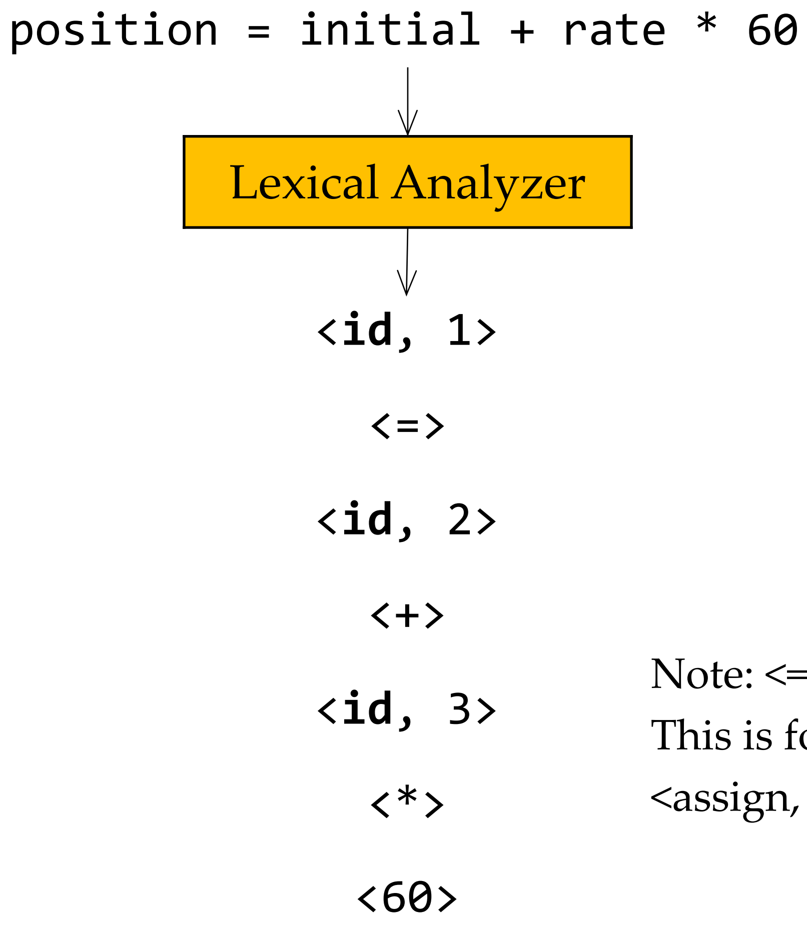
Lexemes vs. Tokens

- 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
```

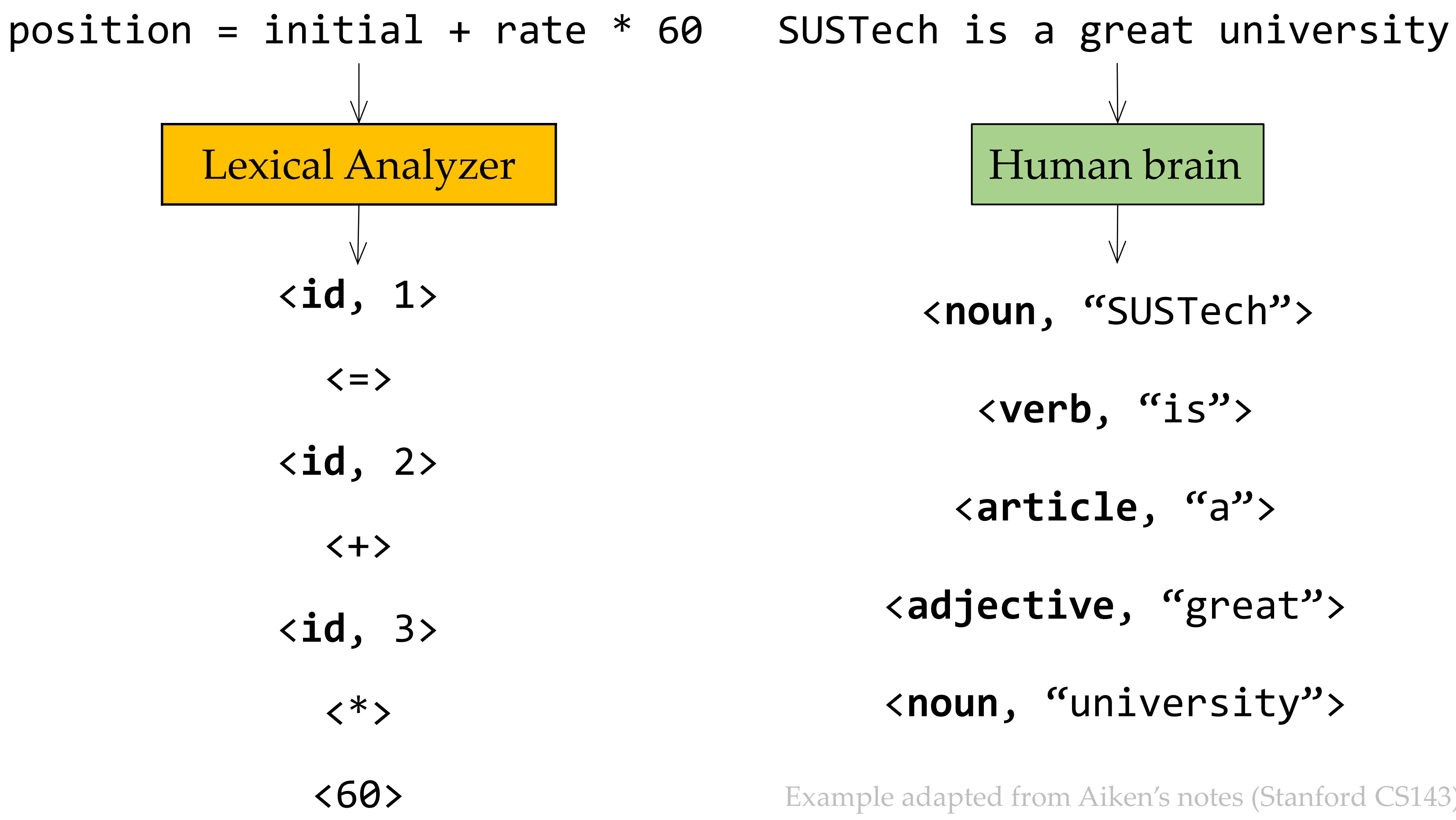
```
<id, 1>
<=>
<id, 2>
<+>
<id, 3>
<*>
<60>
```

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
```

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

```
SUSTech is a great university
```

```
<noun, “SUSTech”>
<verb, “is”>
<article, “a”>
<adjective, “great”>
<noun, “university”>
```

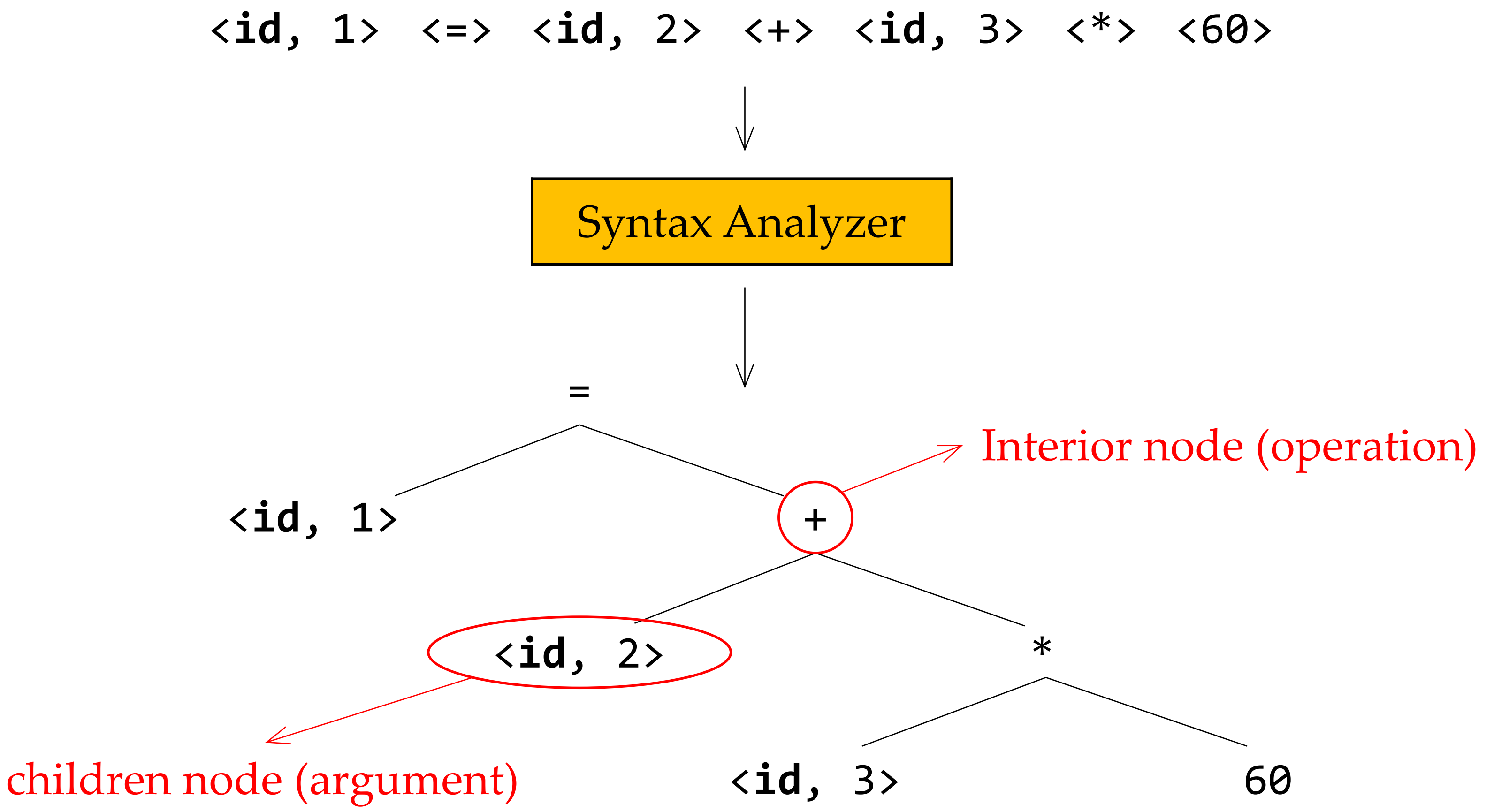
Example adapted from Aiken’s notes (Stanford CS143)

Syntax Analysis (Parsing, 语法分析)

$$\begin{tikzcd}\text{Token stream} \arrow[r] & \boxed{\text{Syntax Analyzer}} \arrow[r] & \text{Syntax tree}\end{tikzcd}$$

- 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

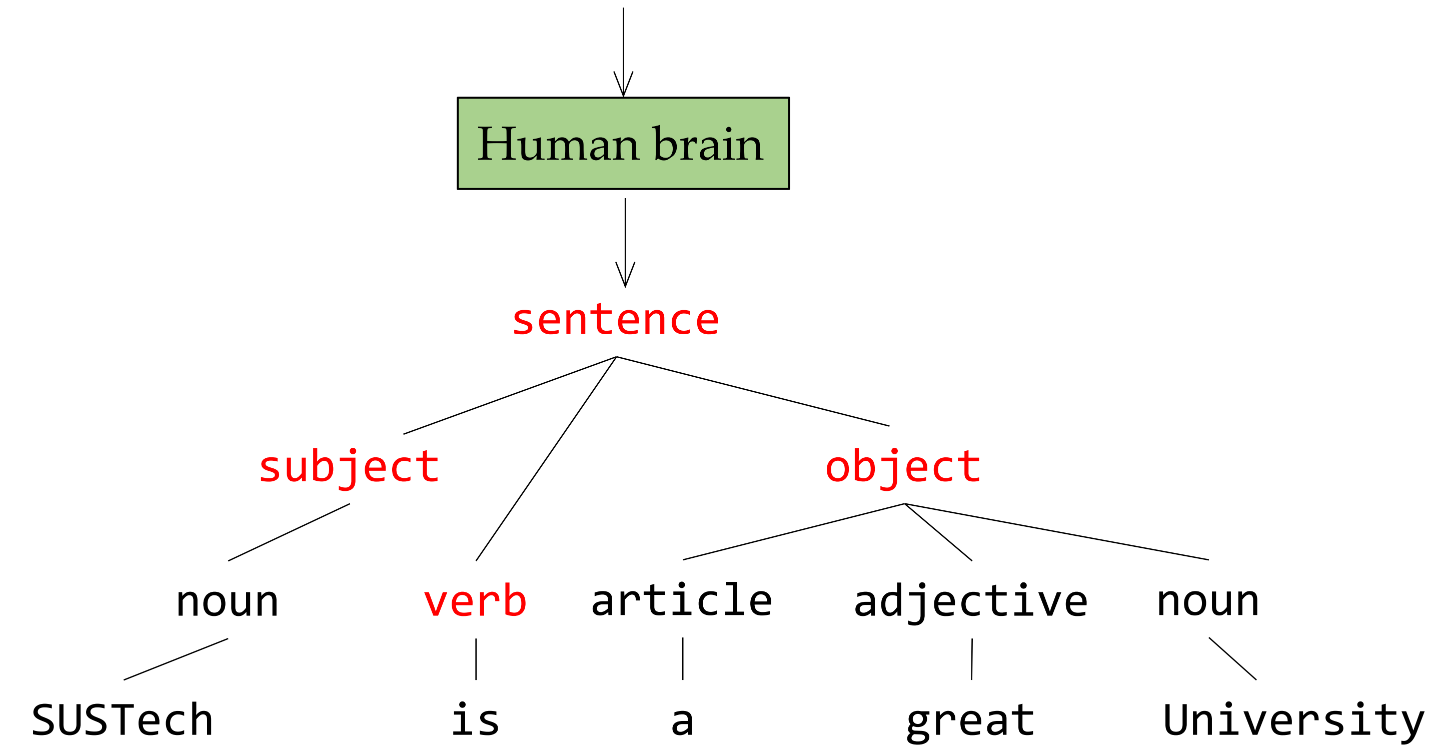
Syntax Analysis (Example)


<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>
Syntax Analyzer
=
<id, 1>
+
Interior node (operation)
<id, 2>
*
children node (argument)
<id, 3>
60

Syntax Analysis in English

<**article**, “SUSTech”> <**verb**, “is”> <**article**, “a”>

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



Semantic Analysis (语义分析)

$$\begin{tikzcd}\text{Syntax tree} \arrow[r] & \boxed{\text{Semantic Analyzer}} \arrow[r] & \text{Syntax tree}\end{tikzcd}$$

- 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)

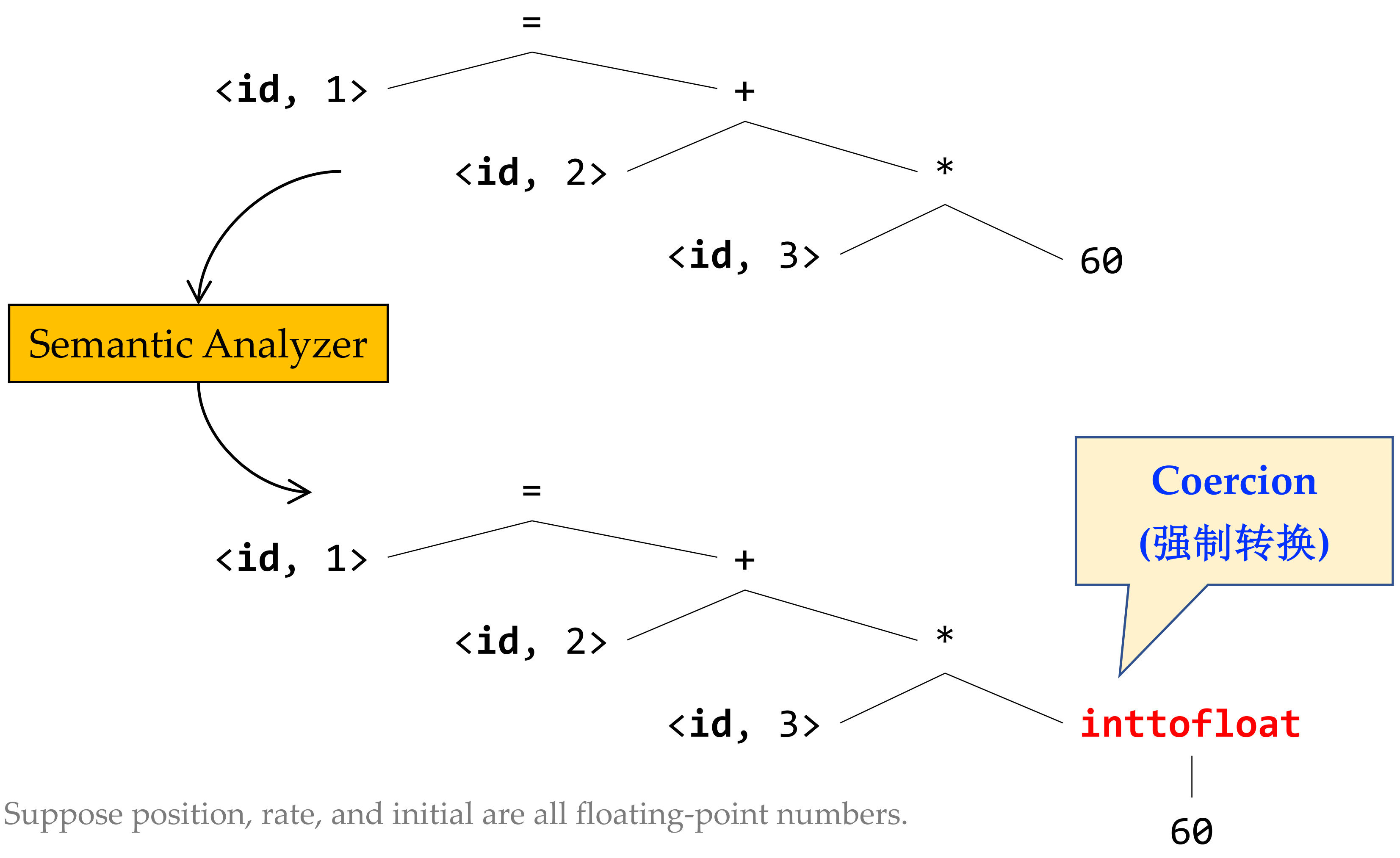
- An important part of semantic analysis is type checking
- Compilers check that each operator has matching operands (of correct types)

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

```
double x = 3.2;
int[] nums = new int[5];
nums[x] = 6;
```

Compilers should report an error!

Semantic Analysis (Example)



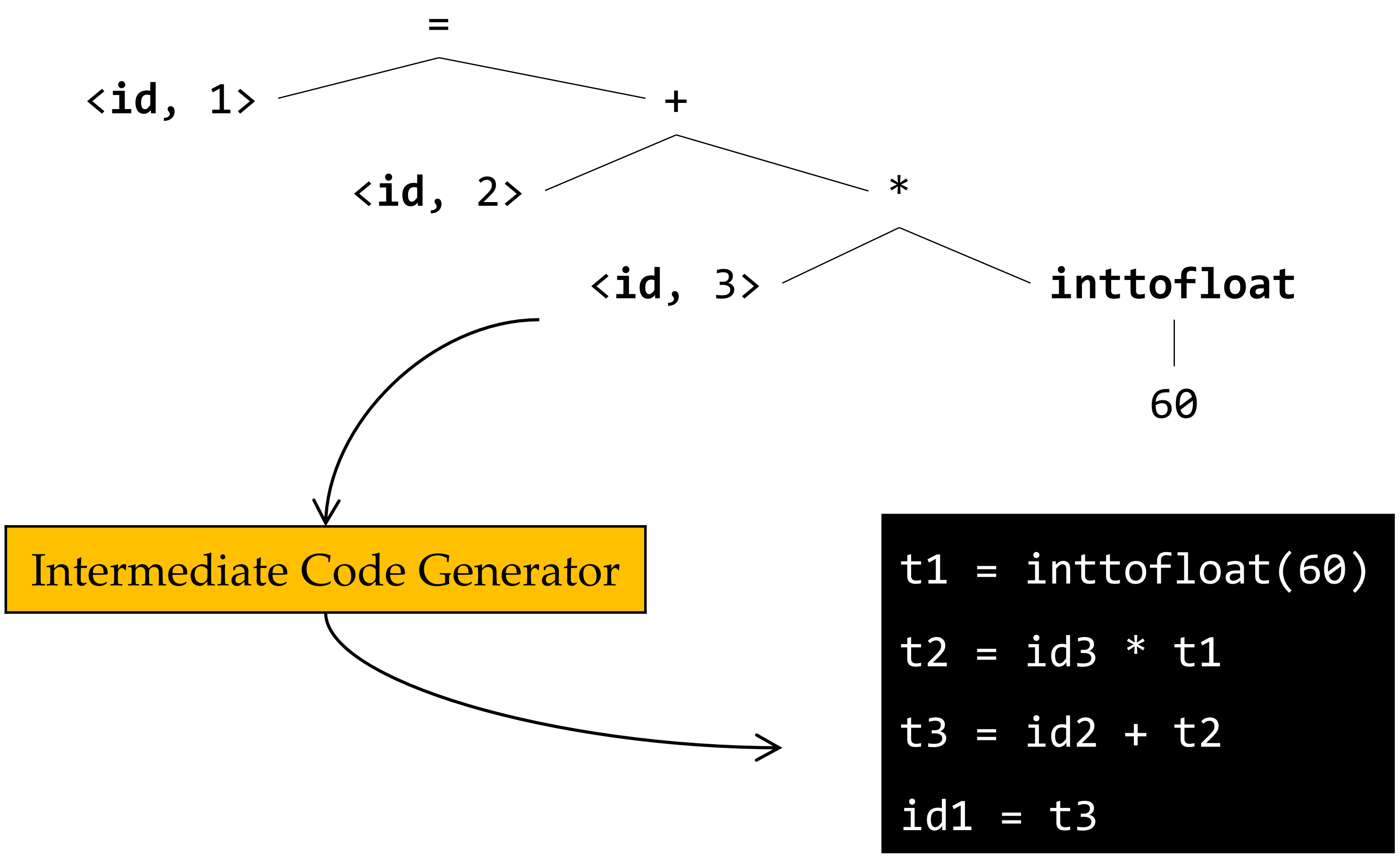
Suppose position, rate, and initial are all floating-point numbers.

Intermediate Code Generation (中间代码生成)

$$\begin{tikzcd}\text{Syntax tree} \arrow[r] & \boxed{\text{Intermediate Code Generator}} \arrow[r] & \begin{array}{c}\text{IR}\\ \text{(in three-address code)}\end{array}\end{tikzcd}$$

- 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



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

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

$$\begin{tikzcd}\text{IR} \arrow[r] & \boxed{\text{Machine-Independent Code Optimizer}} \arrow[r] & \text{IR}\end{tikzcd}$$

- 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. t2 and t3 are only used for value transmitting

Optimization

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

Code Generation (代码生成)

$$\begin{tikzcd} \text{IR} \arrow[r] & \boxed{\text{Code Generator}} \arrow[r] & \text{Target-machine code} \end{tikzcd}$$

- 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
```

$$\xrightarrow{\text{Code Generation}}$$

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

Symbol Table Management

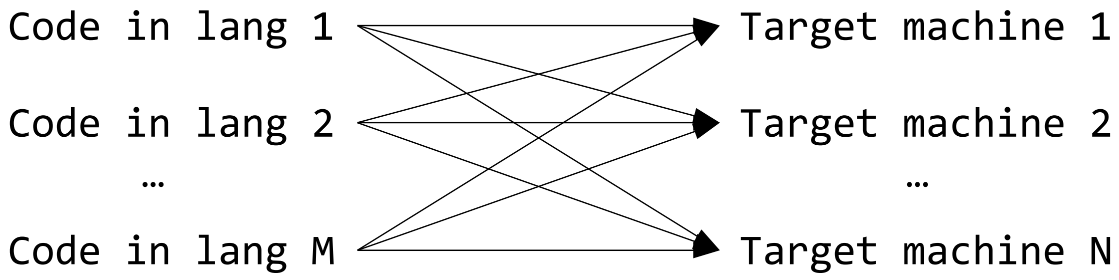
- Performed by the frontend, symbol table is passed along with the intermediate code to the backend

- Record the variable names and various attributes
  - storage allocated, type, scope

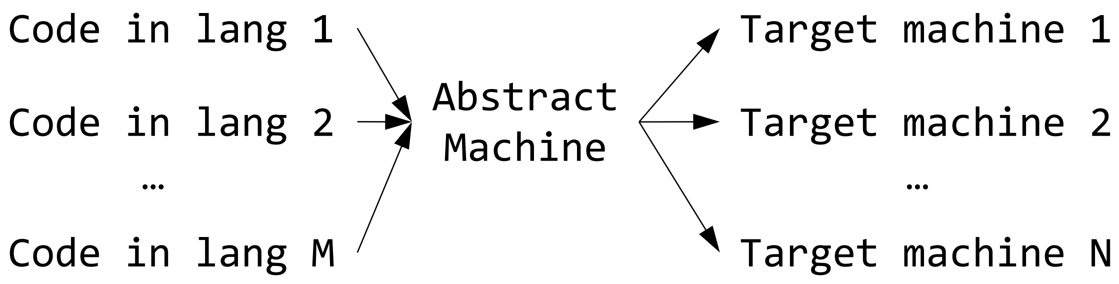
- Record the procedure names and various attributes
  - the number and type of arguments
  - the way of passing arguments (by value or by reference)
  - the return type

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



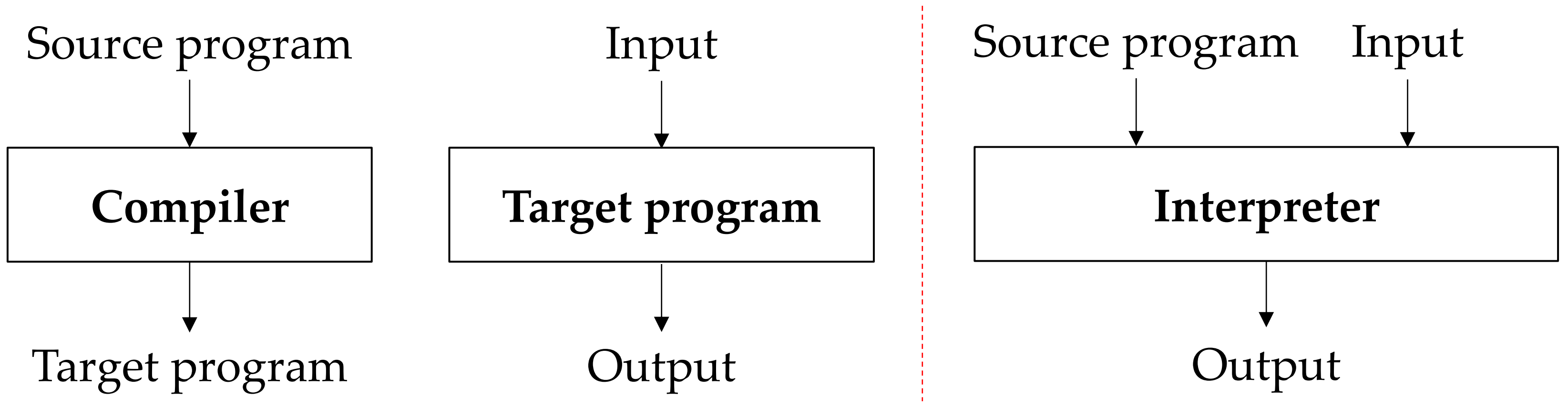
M + N compilers with a good IL

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.


$$\begin{tikzcd}
\text{Source program} \arrow[d] & \text{Input} \arrow[d] & \text{Source program} \arrow[dr] & & \text{Input} \arrow[dl] \\
\boxed{\textbf{Compiler}} \arrow[d] & \boxed{\textbf{Target program}} \arrow[d] & & \boxed{\textbf{Interpreter}} \arrow[d] & \\
\text{Target program} & \text{Output} & & \text{Output} &
\end{tikzcd}$$

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.

Reading Tasks

- Chapter 1 of the Dragon book
    - 1.1 Language Processors
    - 1.2 The Structure of a Compiler
    - 1.3 The Evolution of Programming Languages