南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Chapter 0: Course Introduction

Yepang Liu

liuyp1@sustech.edu.cn

Outline

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

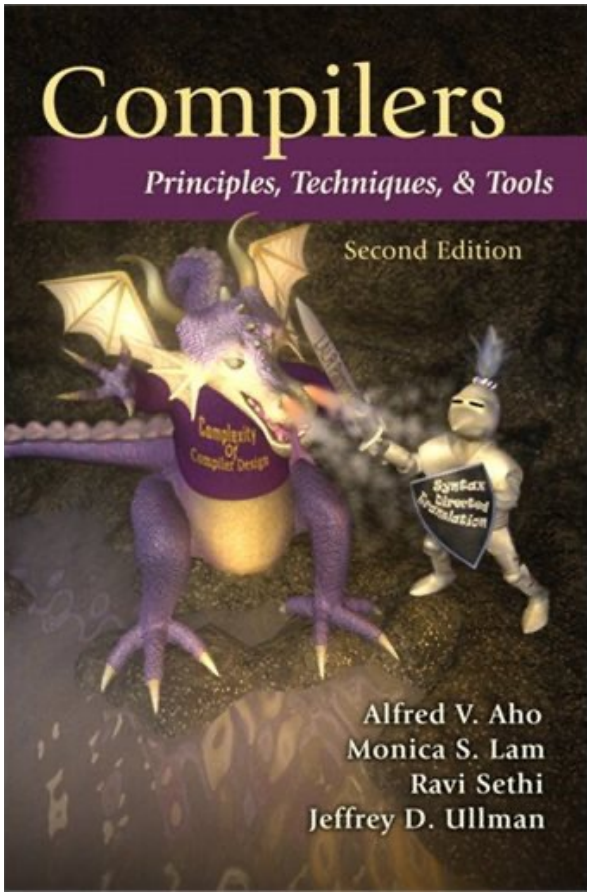
The Teaching Team

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

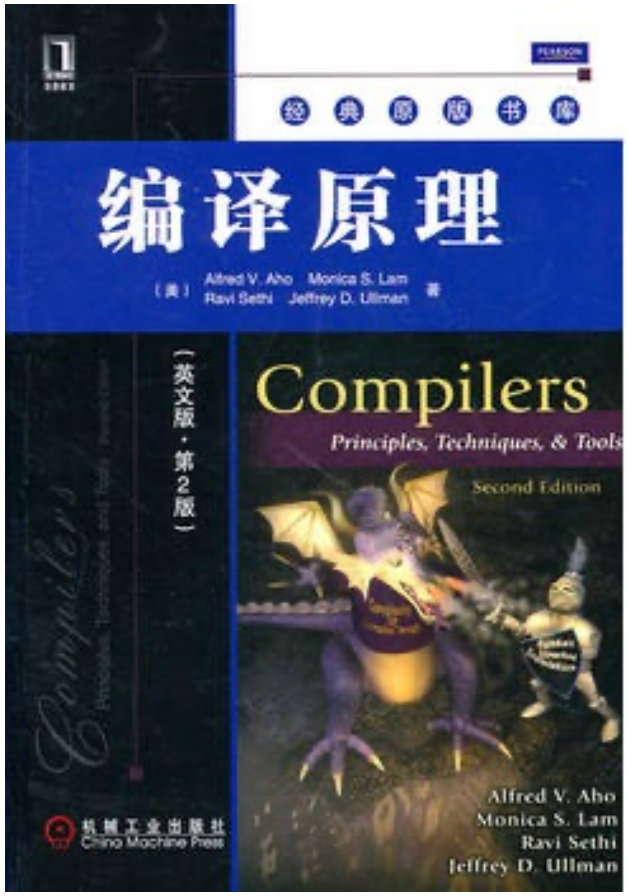
- **TA:** 陈俊峰（RA）, 陈一戈（MSc Student）, 王海龙（MSc Student）
  - Email: chenjf2020, 12432659, 12532569@mail.sustech.edu.cn
  - Office: Room 650A, CoE Building (South)

- **Communication:**
  - Emails: typically replied within 24 hours
  - WeChat: 请扫右侧企业微信群二维码入群
  - Office hour: 10:00 am – 12:00 pm, every Monday

Textbook: The "Dragon Book"



Available at library



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

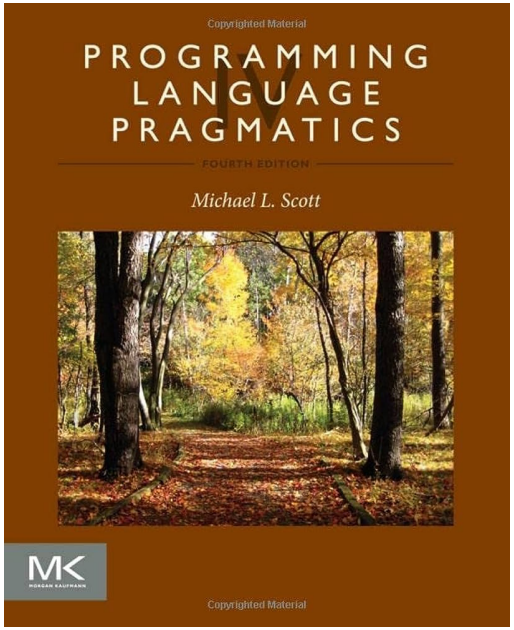
Textbook: Chinese Version

南京大学赵建华、郑滔、戴新宇译

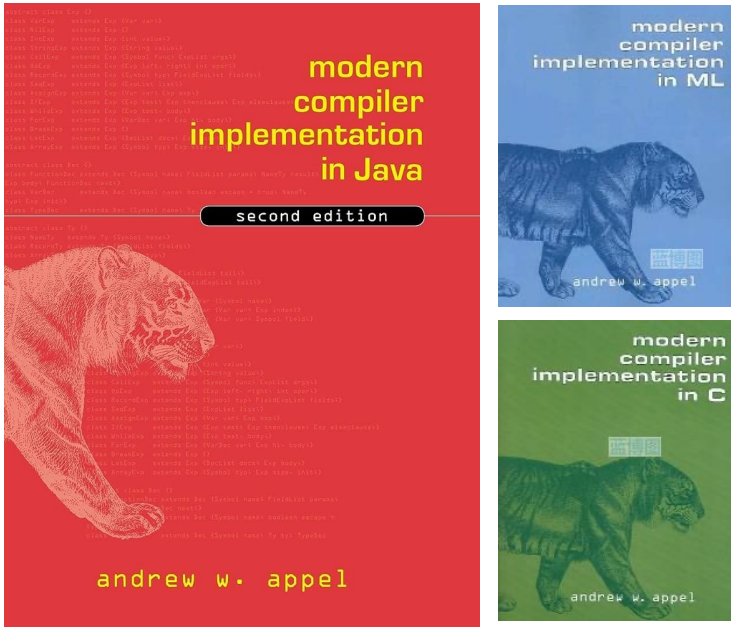
Available at library

~60￥on 京东

Other Reference Books



Available at library



Available at library

Different versions: Java, C, ML

Lecture Materials

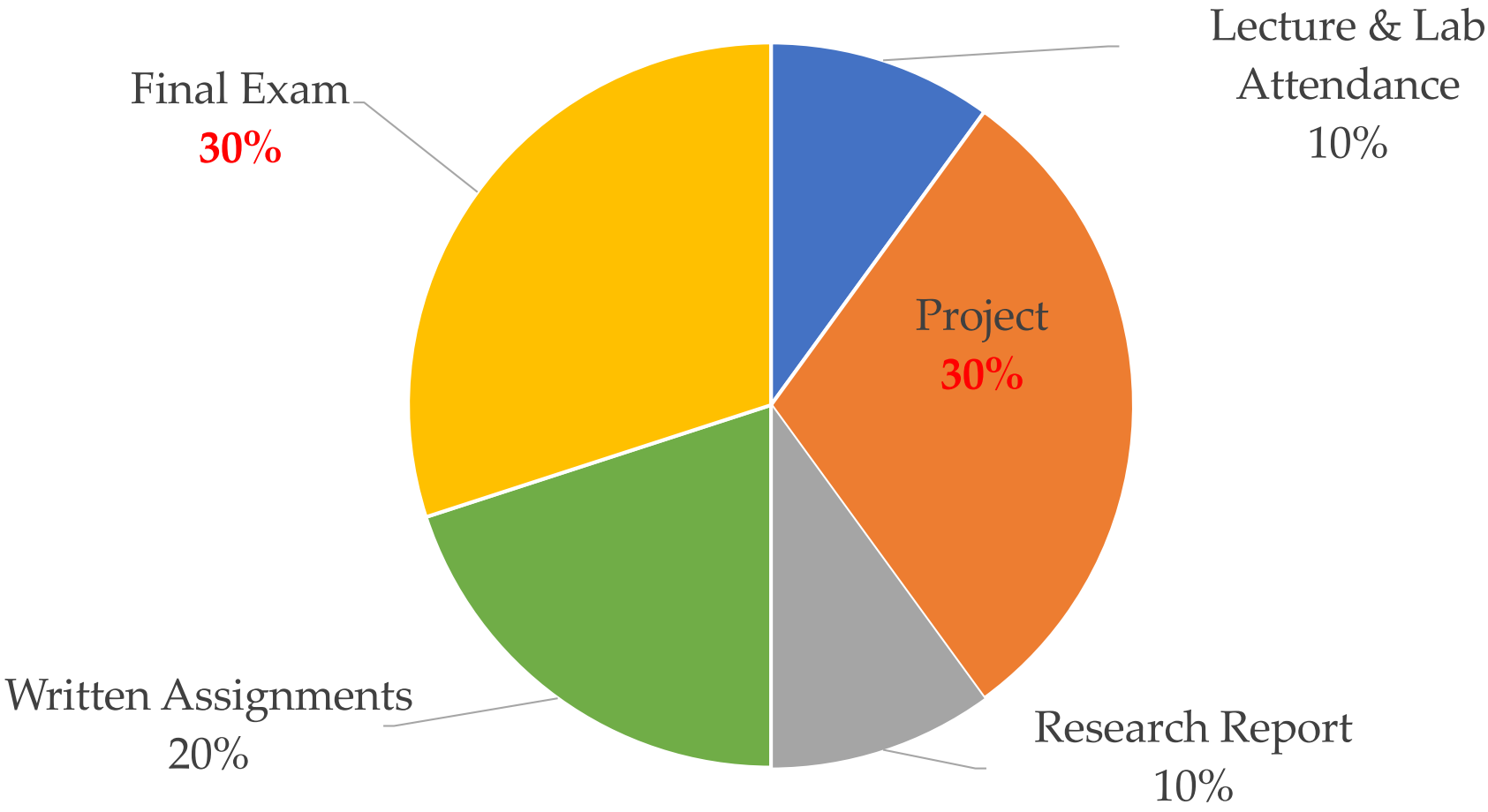
- Announcements, lecture/lab notes, written assignments, sample answers, etc. are available on Blackboard
  - Registered students will be automatically added to the site

Lab Resources

- Lab tutorials: https://sqlab-sustech.github.io/CS323-Compilers-2025F-docs/

- Lab project template repo: https://github.com/sqlab-sustech/CS323-Compilers-2025F-Projects/

Marking Scheme



Note: The marking scheme may be subject to minor changes.

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?

- **Gain a deep understanding of computer programs**
  - The transformation from high-level abstractions to machine operations
  - The relationship between code and hardware
- **Learn program analysis and code optimization techniques**
  - Essential for writing high-performance code and systems
- **Learn how to design and implement languages for computing**
  - Domain-specific languages (DSLs) have wide applications
- **The core technologies are fundamental to modern tools**
  - Deep learning compilers (e.g., Apache TVM) take DL framework models as input and generate optimized executables for the target hardware (e.g., specialized accelerators)

Ultimate Training in Computational Thinking and Problem-Solving:
Compilers involve formal language theory, algorithms, data structures, computer architecture, and software engineering.

Your Compiler Survival Guide

- This course is a marathon:
  - 15 lecture and lab sessions, some of which are quite difficult
  - 5 written assignments + 6 projects + a research report
  - Several quizzes + a final exam

- **Tip 1:** Focus on the concepts, principles, algorithms.
- **Tip 2:** Start projects early. Complexity unfolds over time.
- **Tip 3:** Find good teammates. Collaboration is important.
- **Tip 4:** Communicate with us proactively. Don't stay stuck for days.

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
- There are over 700 programming languages[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

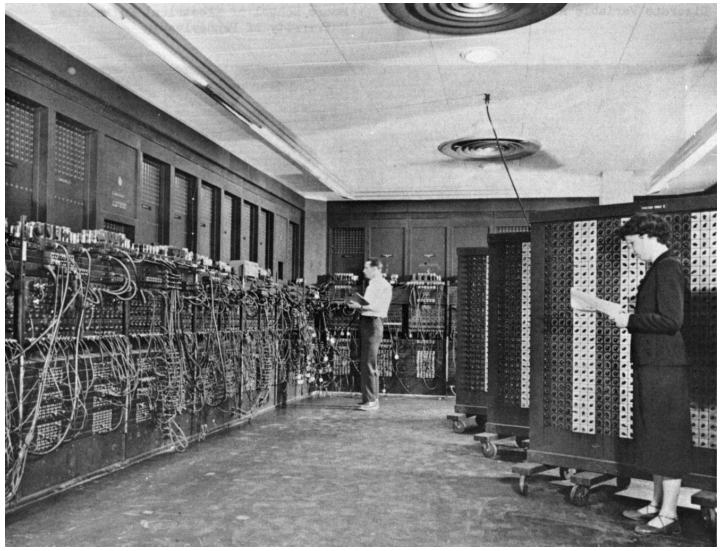
Top Programming Languages

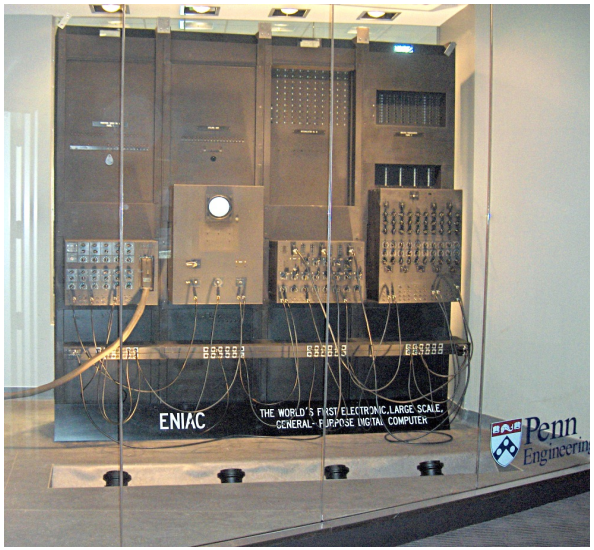
- Winners in the past 10 years according to TIOBE index[1]:
  - Python: 2024, 2021, 2020, 2018
  - C: 2019, 2017
  - C#: 2023
  - C++: 2022
  - Go: 2016
  - Java: 2015

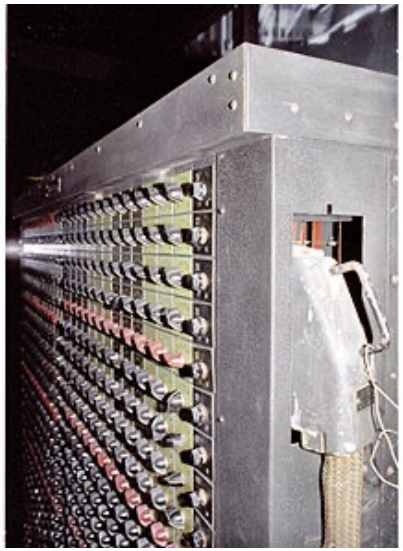
[1] A measure of popularity of programming languages calculated from the number of search engine results for queries containing the name of the language. See the index at https://www.tiobe.com/tiobe-index/.

When It All Started …

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







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?

```
00001001001011100110011001101001011011000110010100001000
10010001001101100011001010110001101111010001110101011100
10011001010011000100101110011000110010001000001010011000
11101100011011000110011001001011111011000110110111101100
11010111000001101001011011000110010101100100001011100011
11010000010100010111001110011011001010110001101110100011
10100101101111011011100000100100100010001011100111010000
11001010111100001110100001000100000101000001001001011100
01100001011011000110100101100111011011100010000000110100
00000101000001001001011100110011101101100011011110110000
10011000010110110000100000011011010110000101101001011010
11000001010000010010010111001110100011110010111000001100
01010000100100100000011011010110000101101001011011100...
```

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 (宏指令) 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)

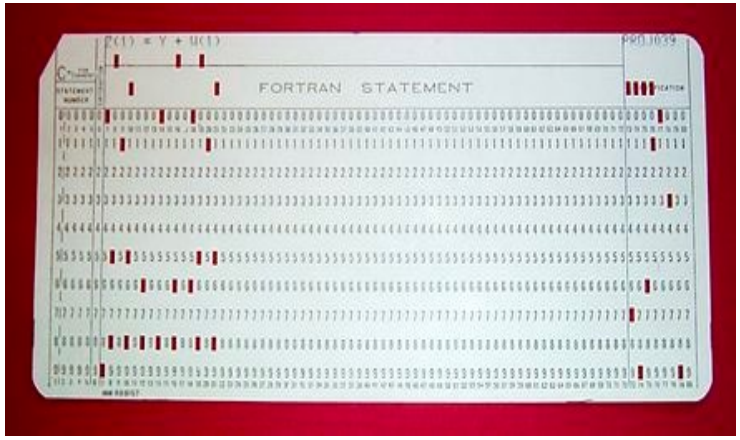
IBM 704 mainframe

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. 11, TIOBE Index August 2025)

```
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



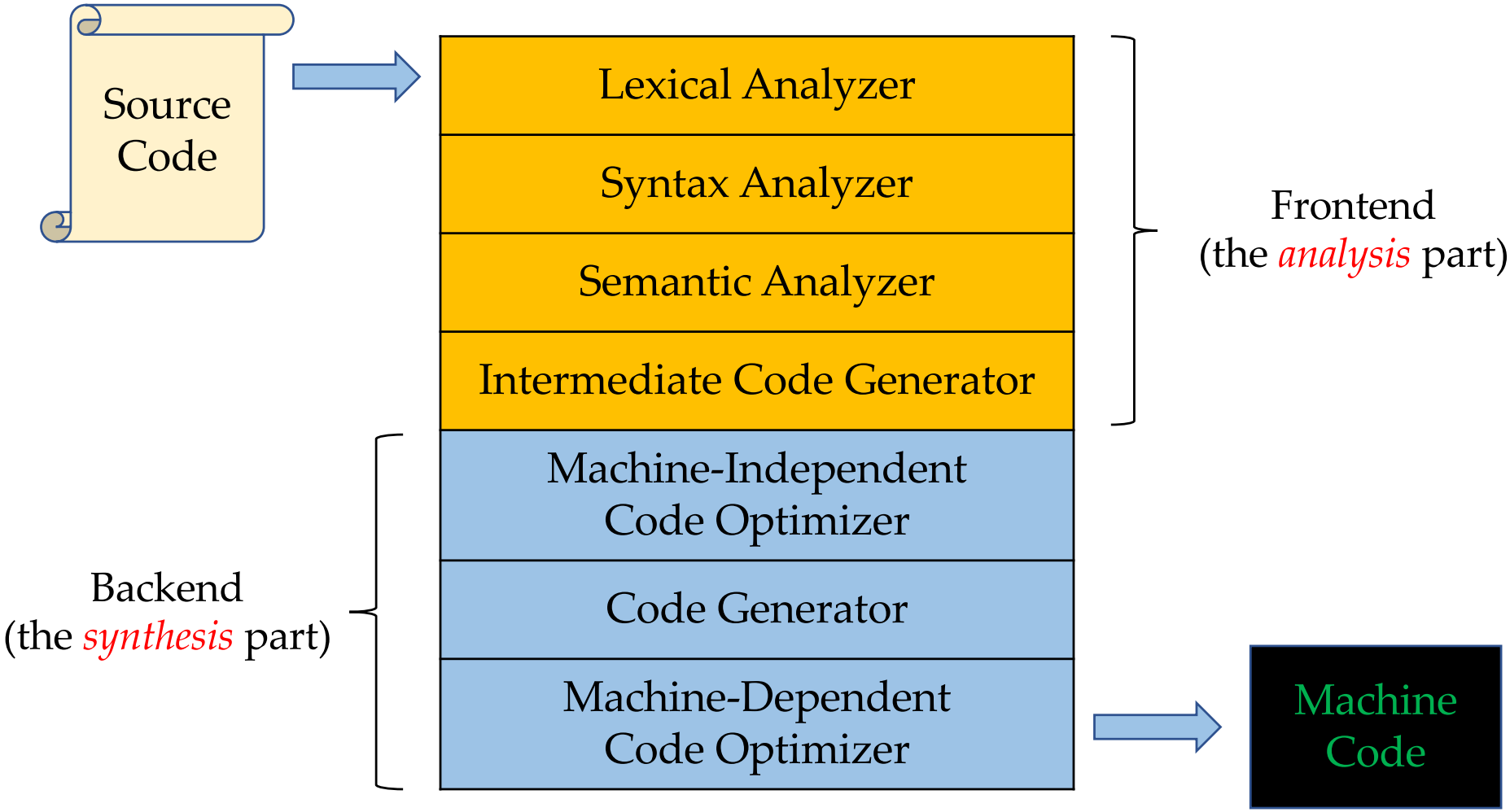
Fortran code on a punch card

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

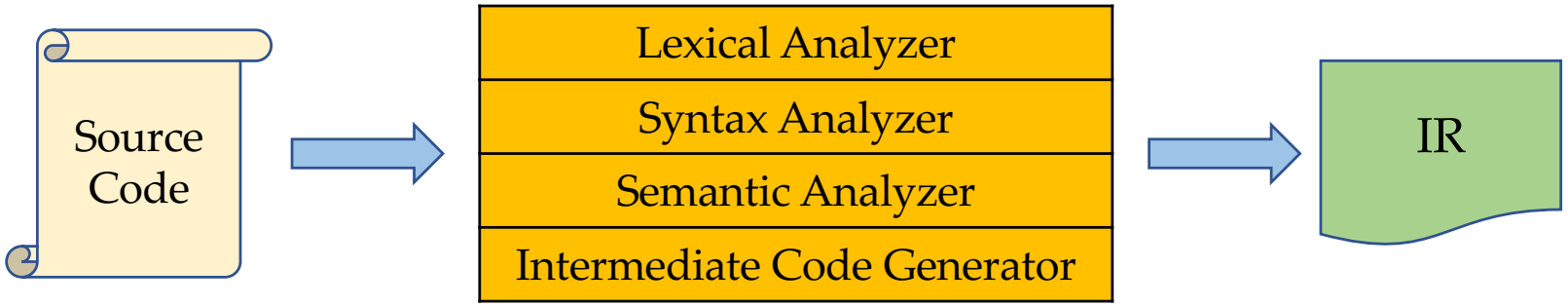
Outline

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

The Structure of a Compiler

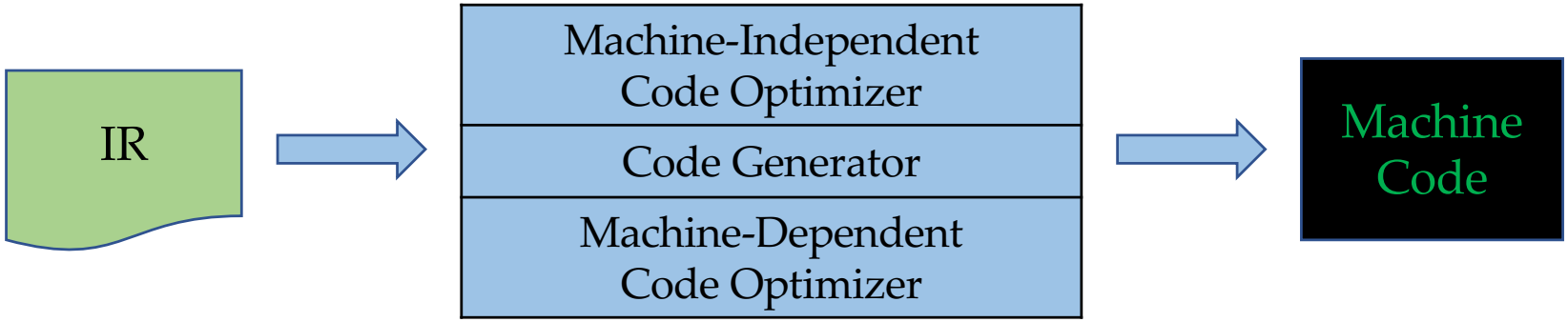


The Frontend (前端) of a Compiler



- 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*

* 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)

- 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>
```

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

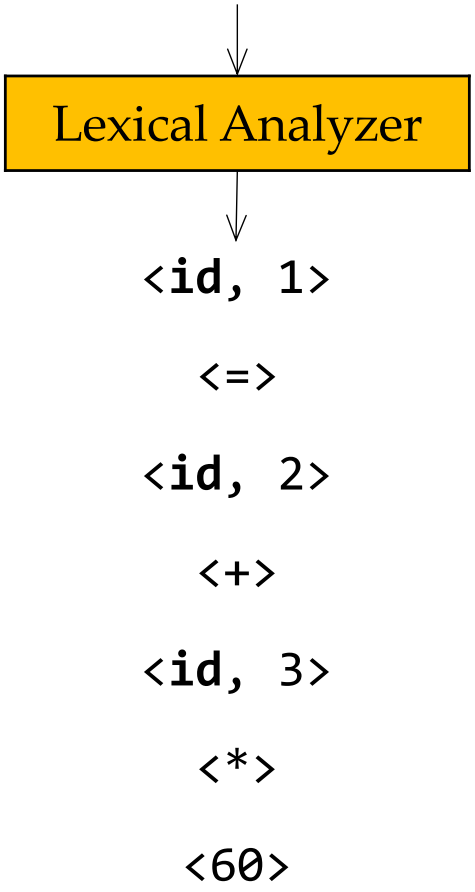
- A **lexeme** is a string of characters that is a lowest-level syntactic unit in the programming language
  - "words" and punctuation of the programming language (**instance**)

- 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
```

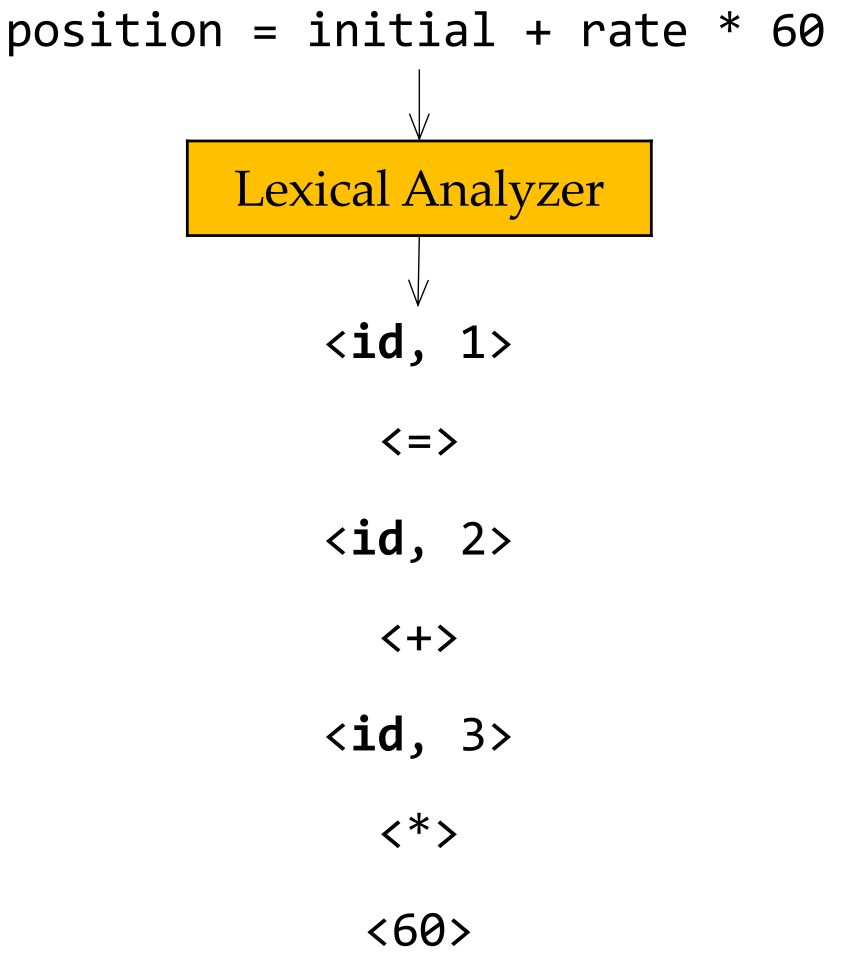


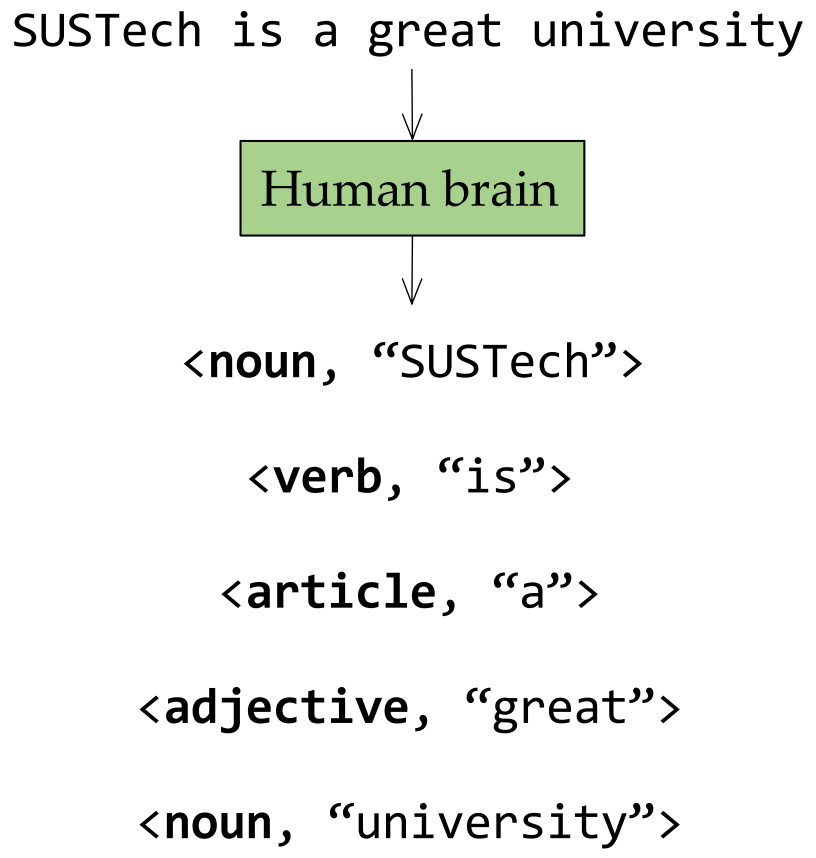
SYMBOL TABLE

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)



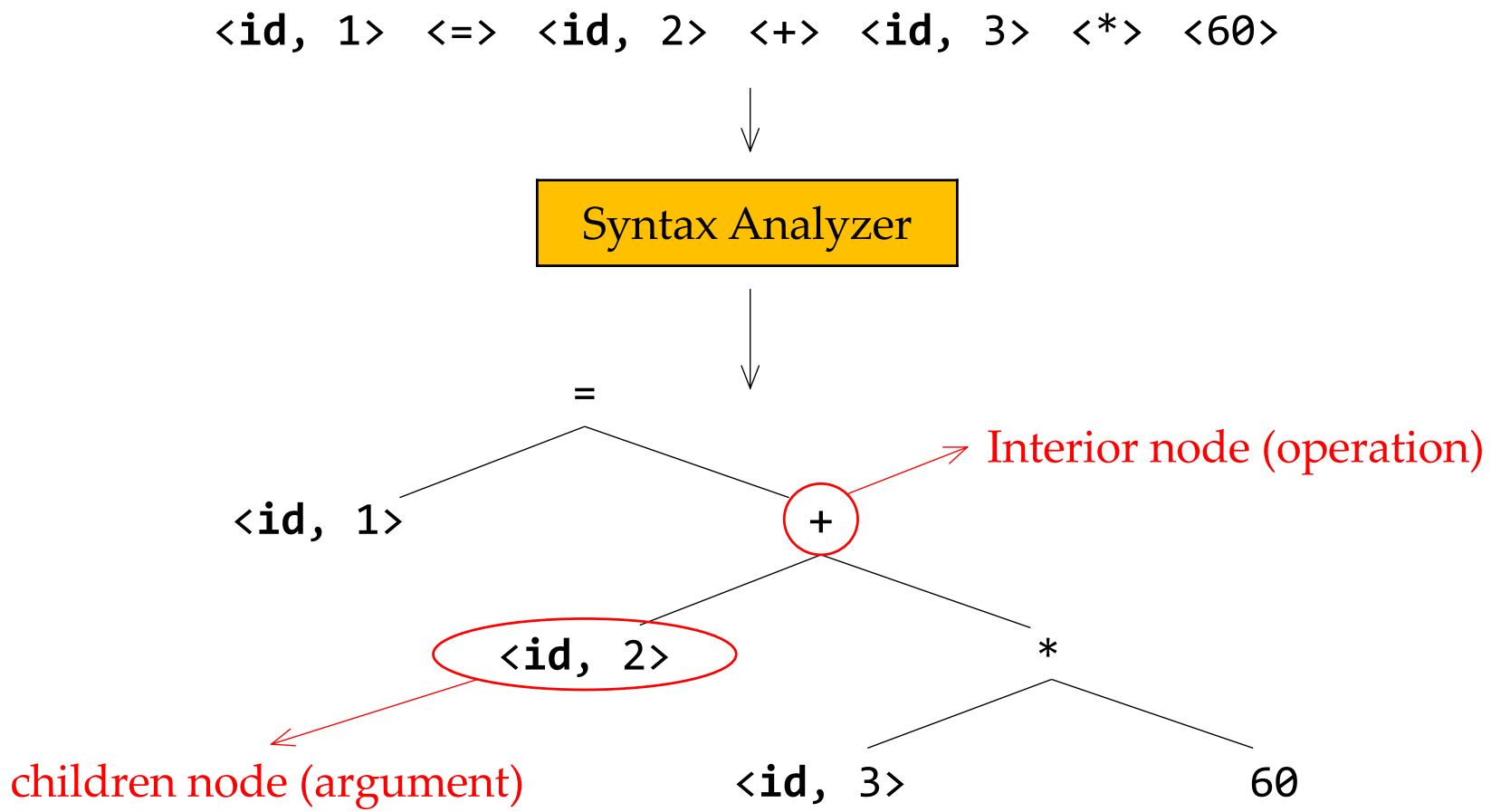


Example adapted from Aiken's notes (Stanford CS143)

Syntax Analysis (Parsing, 语法分析)

- 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

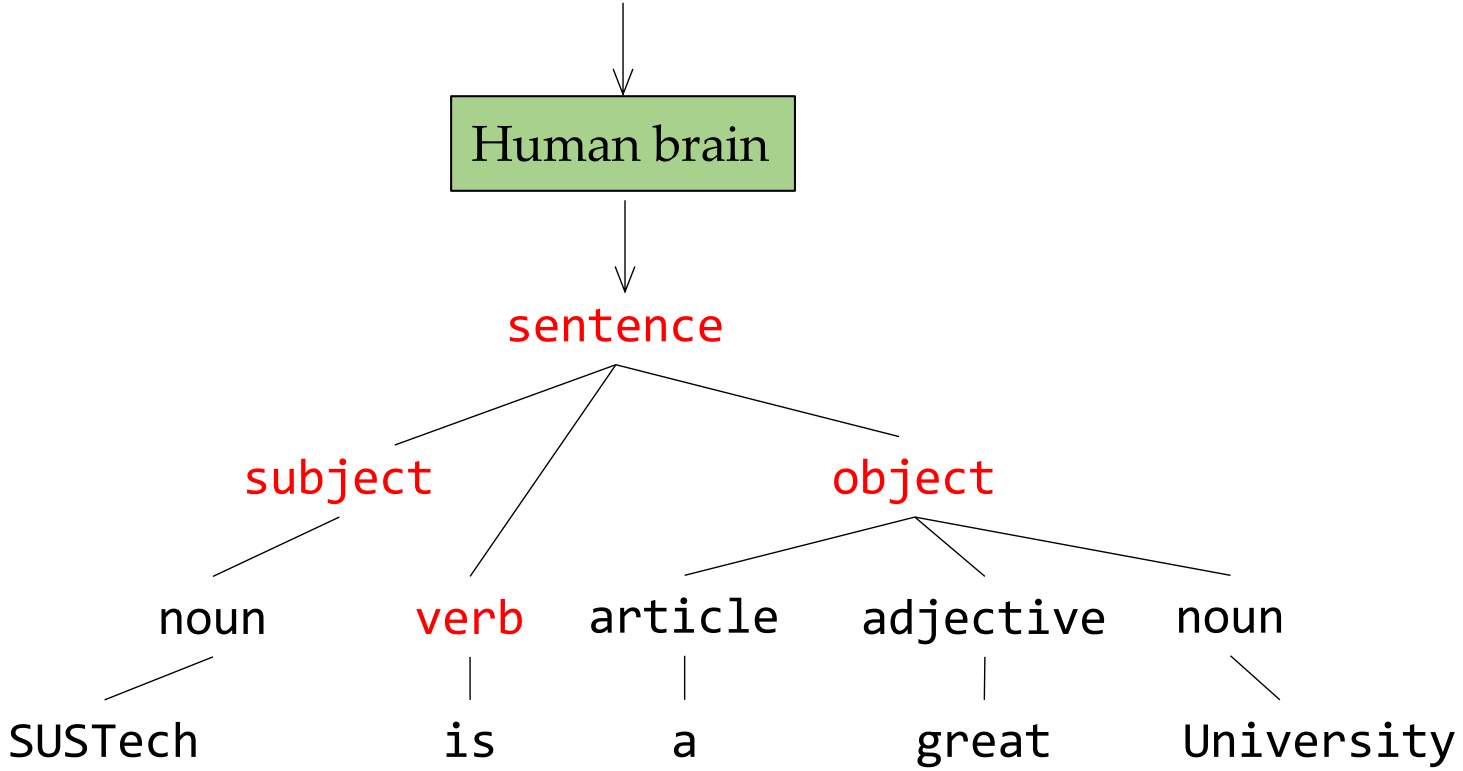
Syntax Analysis (Example)



Syntax Analysis in English

<article, "SUSTech"> <verb, "is"> <article, "a">

<adjective, "great"> <noun, "university">



Semantic Analysis (语义分析)

- 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 (such as type checking) 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 languages require an array index to be an integer.

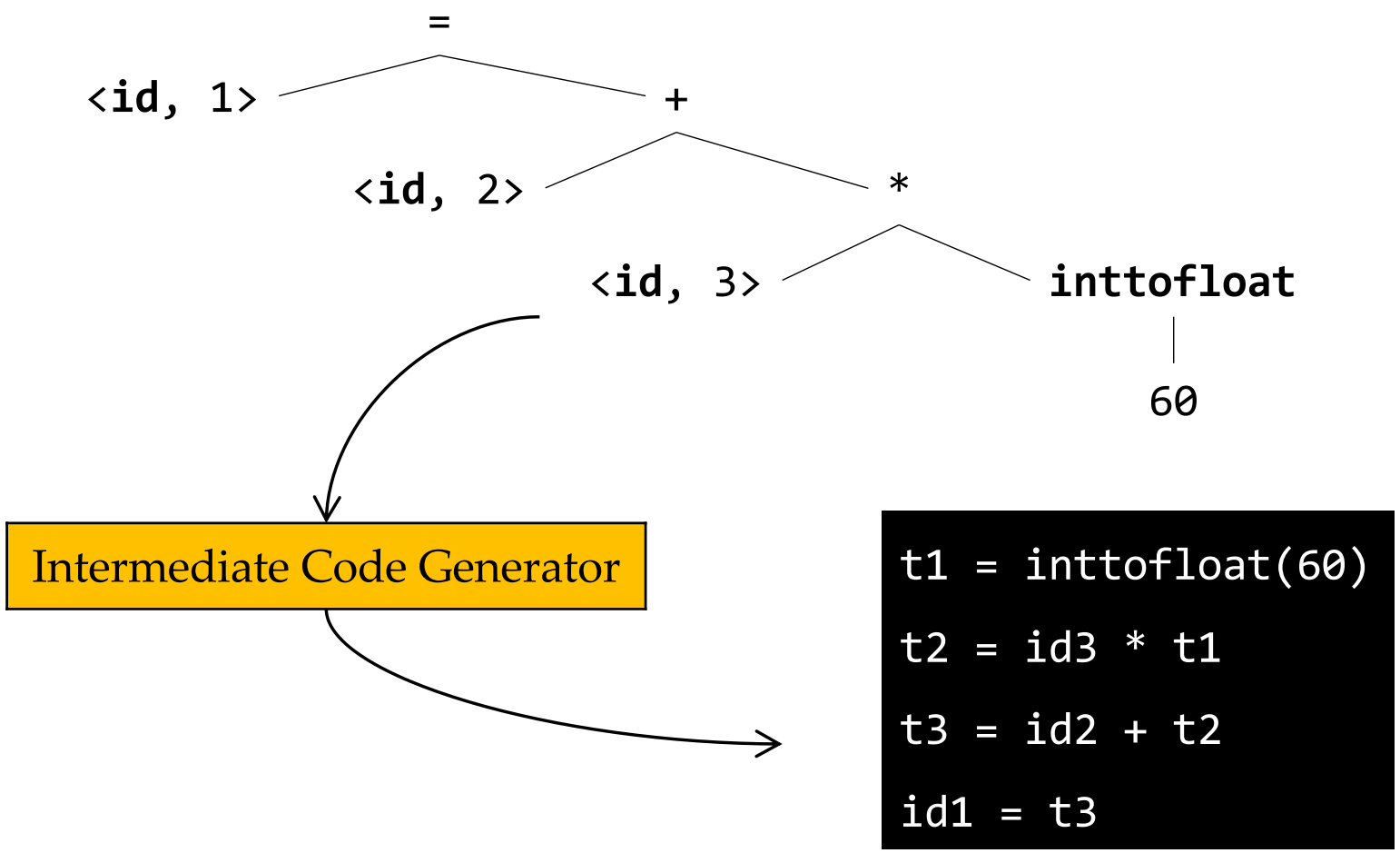
```
double x = 3.2;
int[] nums = new int[5];
nums[x] = 6;
```

Compilers should report an error!

Intermediate Code Generation (中间代码生成)

- 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)

- 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
```

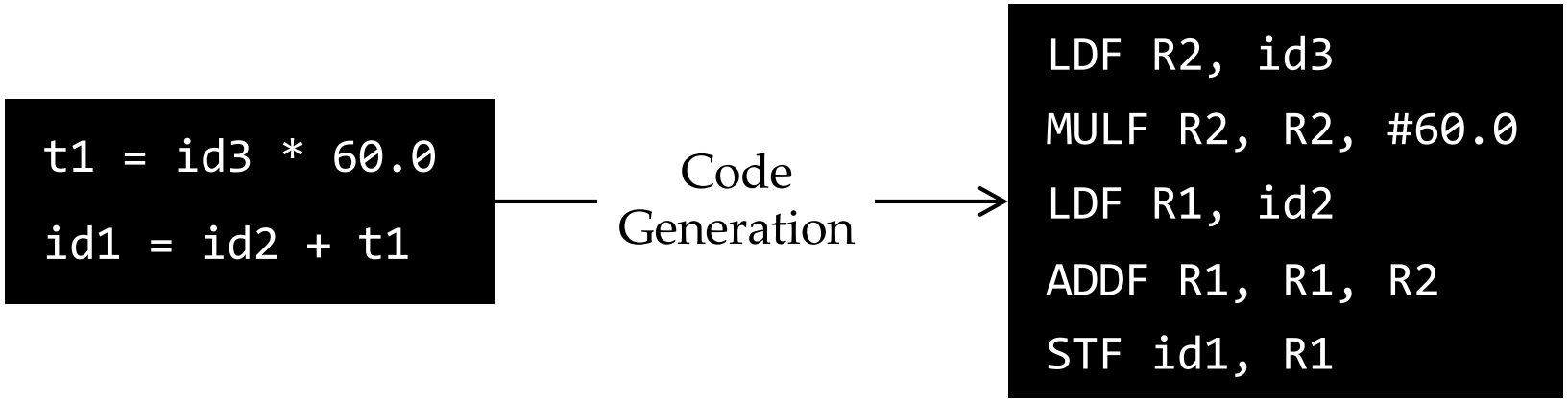
1. 60 is a constant integer value. Its conversion to floating-point can be done once and for all at compile time
2. t2 and t3 are only used for value transmitting

Optimization

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

Code Generation (代码生成)

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

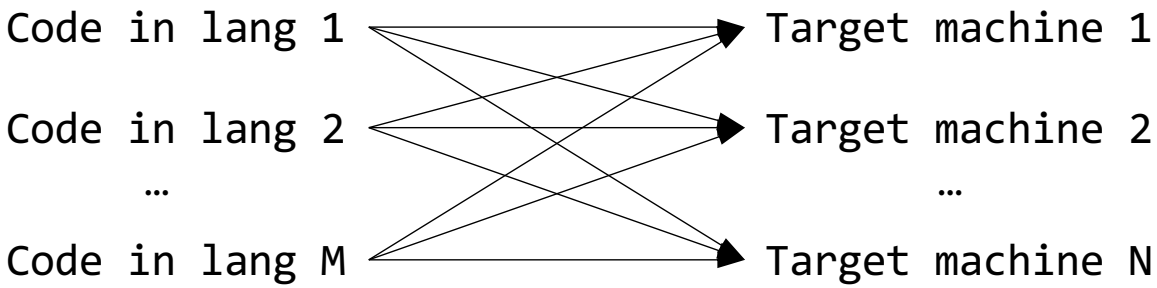


Symbol Table Management

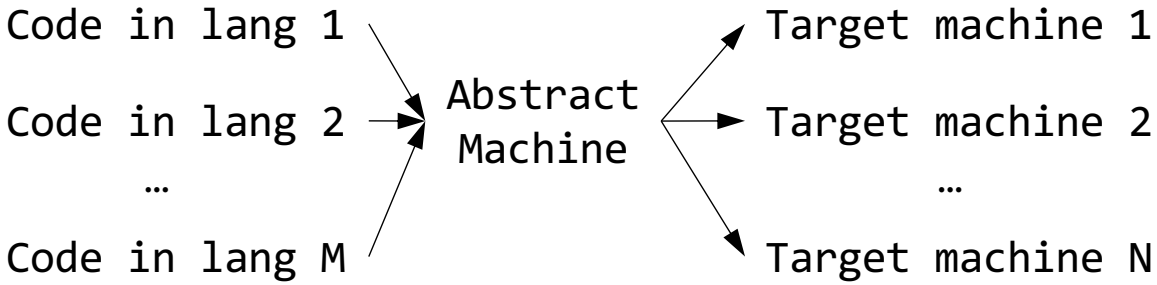
- Performed by the frontend, symbol table is passed along with the intermediate code to the backend
- Record the variable names and various attributes
  - storage allocated, type, scope
- Record the procedure names and various attributes
  - the number and type of arguments
  - the way of passing arguments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

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