Compiler Principles

Instructor: Zhen Gao (高珍)

Email: gaozhen@tongji.edu.cn

Office: Room 514, Jishi Building

Course Overview

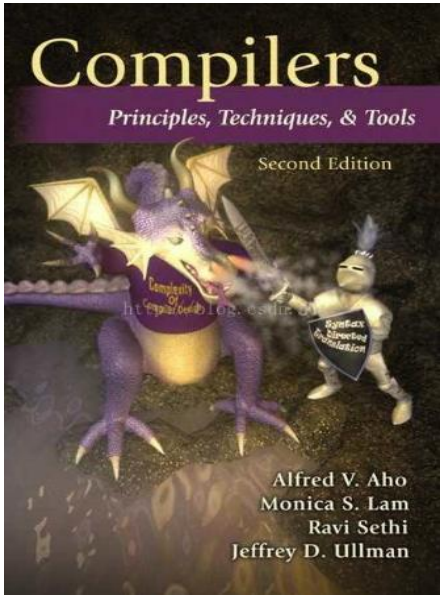
This course introduces the general principles and fundamental implementation methods of compiler construction.

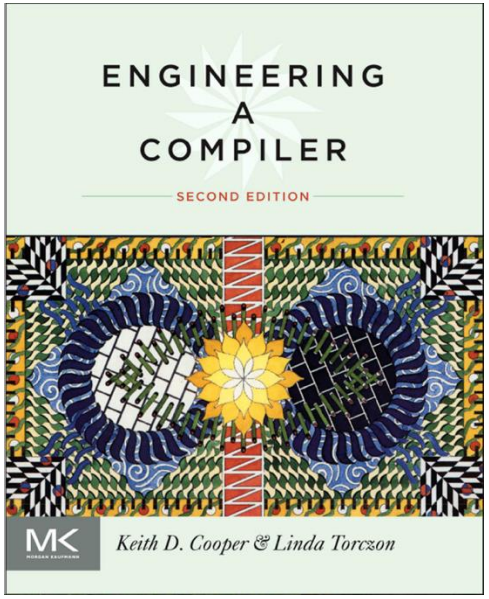
- **Covered Theories:**
  - **Formal languages and automata theory**
  - **Syntax-directed translation and attribute grammars**
  - **Program analysis principles**
- **Key Emphases:**
  - **Emphasis on formal description techniques and automatic generation methods**
  - **Focus on a broad understanding of compiler principles and technologies**
  - **Avoiding distraction by minor algorithms**
  - **No preference for any specific source language or target machine**

Course Overview

Textbooks and References

- *程序设计语言　编译原理（第3版）陈火旺等编著 (387 pages)，国防工业出版社*
- ***Compilers: Principles, Techniques, and Tools (2nd Edition)*** **by Alfred V. Aho et al. (also known as the "Dragon Book") (1038 pages)**
- ***Engineering a Compiler*****, Keith D Cooper, Linda Torczon, Morgan Kaufmann Publisher (825 pages)**





Course Assessment

- Your final grade will be composed of the following parts:
  - Attendance: 10%
  - In-class Quizzes: 20%
  - Final Project: 30% (16th week)
  - Theory Exam: 40% (15th week)

Content Outline

- **What is a compiler**
- **Overview of the compilation process**
- **Structure of a compiler**
- **How a compiler is constructed**
- **Summary**

The Evolution of Programming Languages

Binary or hexadecimal instructions that directly control the hardware

Machine Language

C7 06 0000 0002

moves the number 2 into address 0000

Uses symbols to represent machine instructions and memory addresses.

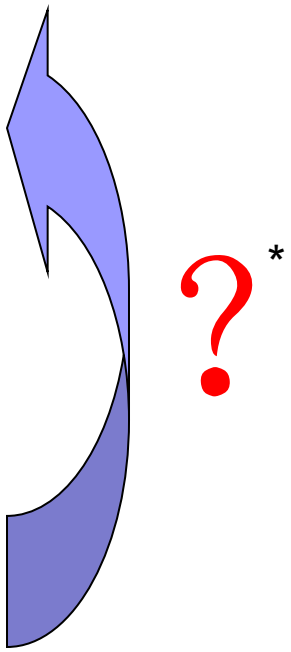
Assembly Language

MOV X, 2

Uses a more natural, human-readable syntax.

High-Level Language

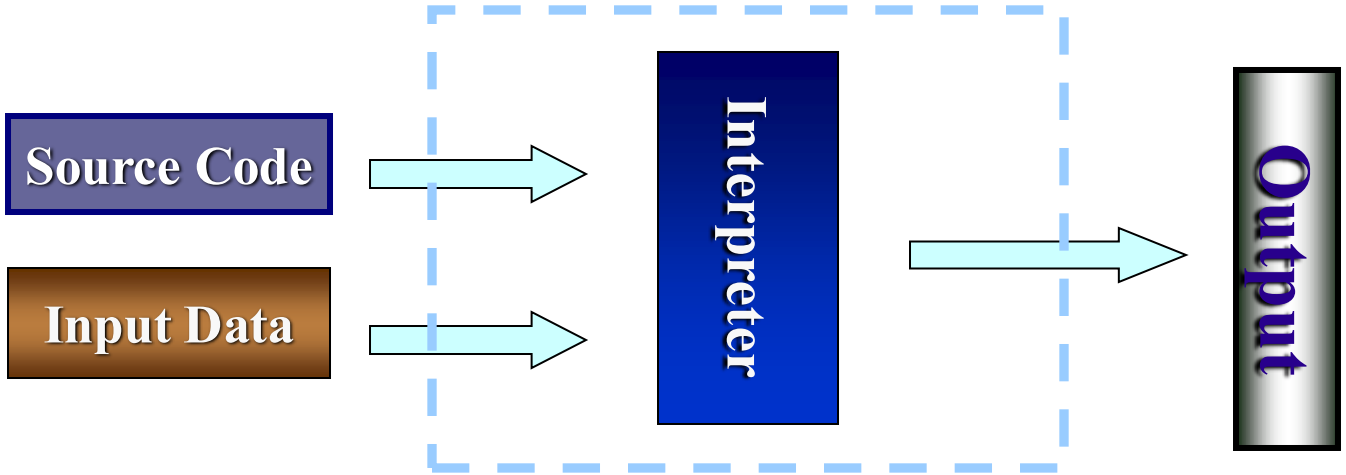
X=2



Two Execution Methods for Programs

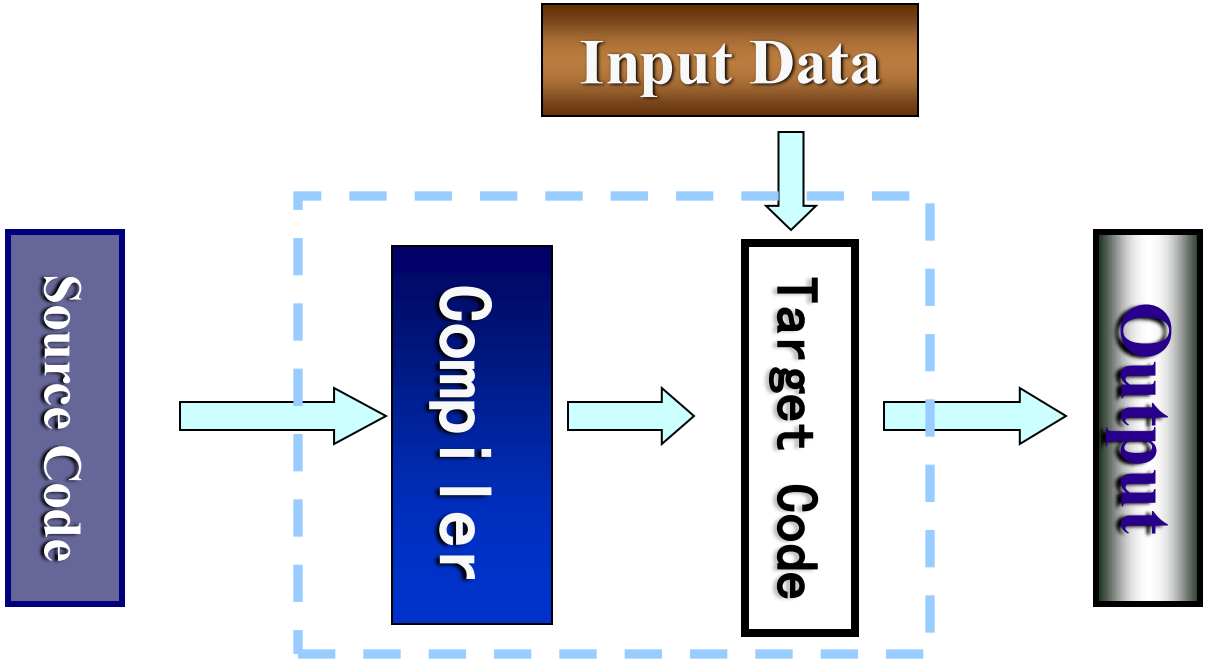
（1）Interpreter

- Executes source code directly, line by line.
- No separate target code is generated.



（2）Compiler

- Translates source code into target code (usually machine code), which can be run independently.
- The target code behaves the same as the original source logically.



Two Execution Methods for Programs

- **Two Ways to Execute High-Level Programs**

Interpretation	Compilation
Executes statements one by one ■ **Examples**: BASIC, Prolog, Shell, JavaScript. ■ **Pros**: Easy to debug. ■ **Cons**: Low efficiency and slow execution speed.	Analyzes and translates the whole program into equivalent machine code before execution. ■ **Examples**: Pascal, Fortran, COBOL, C, C++. ■ **Pros**: Translation happens only once; faster execution. ■ **Cons**: Runtime errors must be located by scanning the entire program.

What type of language is **Java**?

Difference Between Interpreter and Compiler

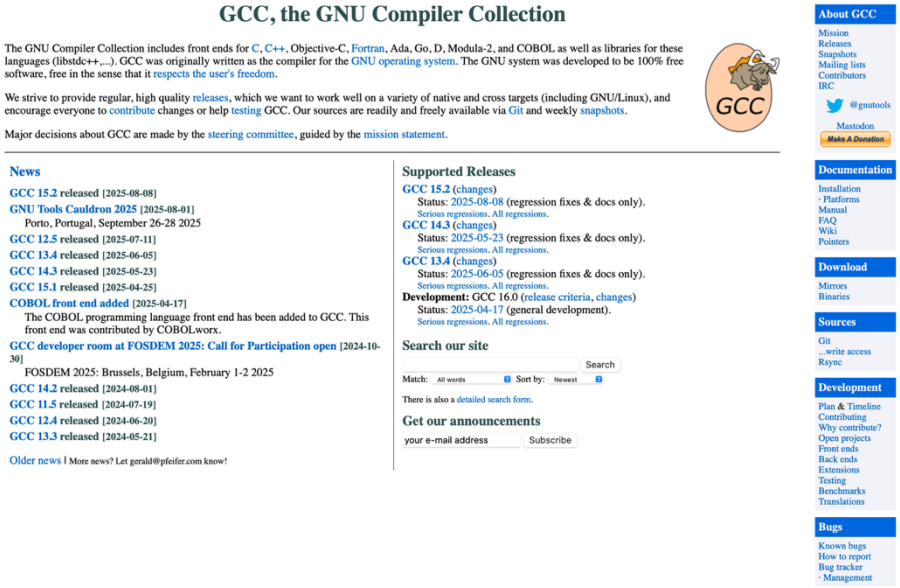
	Function	Output
Compiler	A **translation system** for source programs	Machine Code
Interpreter	An **execution system** for source programs	Execution Result

The Development of Compiler Technology

- **1954–1957**
  John Backus and his team at IBM developed the **FORTRAN** language and its compiler, which could translate arithmetic expressions into machine code.
- **1956**
  Noam Chomsky began research on natural language structure. His findings - **Chomsky Hierarchy** - laid the theoretical foundation that made compiler design much more structured and simplified.
- **1970s**
  Research in **Finite Automata** and formal languages significantly boosted compiler development.
- **1975**
  - Steve Johnson created **YACC** (Yet Another Compiler Compiler) for the Unix system — a tool to automatically generate syntax analyzers.
  - Around the same time, Mike Lesk developed **LEX**, another Unix tool for generating lexical analyzers.

Open-source Compiler

- **https://gcc.gnu.org**
- **https://llvm.org**



GCC, the GNU Compiler Collection

The GNU Compiler Collection includes front ends for C, C++, Objective-C, Fortran, Ada, Go, D, Modula-2, and COBOL as well as libraries for these languages (libstdc++,...). GCC was originally written as the compiler for the GNU operating system. The GNU system was developed to be 100% free software, free in the sense that it respects the user's freedom.

We strive to provide regular, high quality releases, which we want to work well on a variety of native and cross targets (including GNU/Linux), and encourage everyone to contribute changes or help testing GCC. Our sources are readily and freely available via Git and weekly snapshots.

Major decisions about GCC are made by the steering committee, guided by the mission statement.

News

GCC 15.2 released [2025-08-08]
GNU Tools Cauldron 2025 [2025-08-01]
Porto, Portugal, September 26-28 2025
GCC 12.5 released [2025-07-11]
GCC 13.4 released [2025-06-05]
GCC 14.3 released [2025-05-23]
GCC 15.1 released [2025-04-25]
COBOL front end added [2025-04-17]
The COBOL programming language front end has been added to GCC. This front end was contributed by COBOLworx.
GCC developer room at FOSDEM 2025: Call for Participation open [2024-10-30]
FOSDEM 2025: Brussels, Belgium, February 1-2 2025
GCC 14.2 released [2024-08-01]
GCC 11.5 released [2024-07-19]
GCC 12.4 released [2024-06-20]
GCC 13.3 released [2024-05-21]

Older news ! More news? Let gerald@pfeifer.com know!

Supported Releases

GCC 15.2 (changes)
Status: 2025-08-08 (regression fixes & docs only).
Serious regressions. All regressions.
GCC 14.3 (changes)
Status: 2025-05-23 (regression fixes & docs only).
Serious regressions. All regressions.
GCC 13.4 (changes)
Status: 2025-06-05 (regression fixes & docs only).
Serious regressions. All regressions.
Development: GCC 16.0 (release criteria, changes)
Status: 2025-04-17 (general development).
Serious regressions. All regressions.

Search our site

Search
Match: All words Sort by: Newest
There is also a detailed search form.

Get our announcements

your e-mail address Subscribe

About GCC
Mission
Releases
Snapshots
Mailing lists
Contributors
IRC
@gnutools
Mastodon
Make A Donation

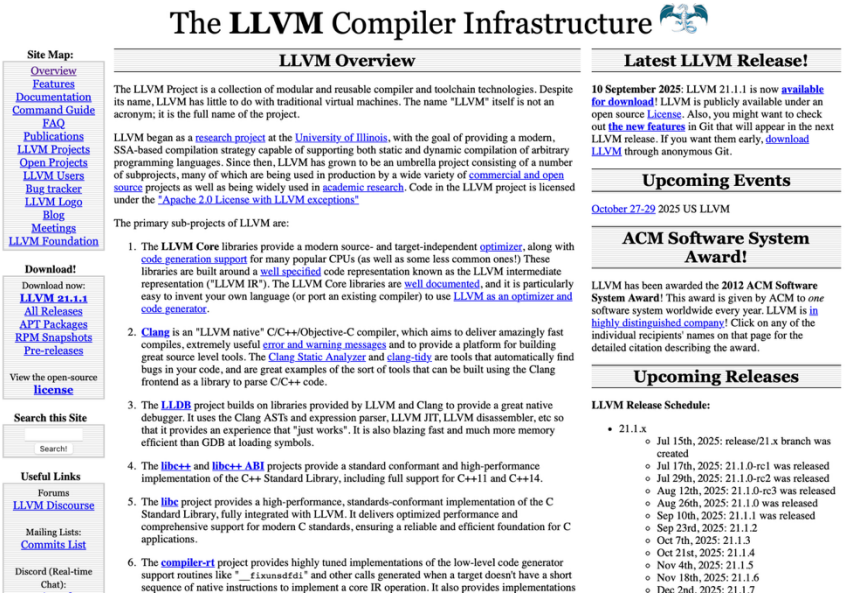
Documentation
Installation
· Platforms
Manual
FAQ
Wiki
Pointers

Download
Mirrors
Binaries

Sources
Git
...write access
Rsync

Development
Plan & Timeline
Contributing
Why contribute?
Open projects
Front ends
Back ends
Extensions
Testing
Benchmarks
Translations

Bugs
Known bugs
How to report
Bug tracker
· Management



The LLVM Compiler Infrastructure

Site Map:
Overview
Features
Documentation
Command Guide
FAQ
Publications
LLVM Projects
Open Projects
LLVM Users
Bug tracker
LLVM Logo
Blog
Meetings
LLVM Foundation

Download!
Download now:
LLVM 21.1.1
All Releases
APT Packages
RPM Snapshots
Pre-releases
View the open-source license

Search this Site
Search!

Useful Links
Forums
LLVM Discourse
Mailing Lists:
Commits List
Discord (Real-time Chat):

LLVM Overview

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project.

LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of subprojects, many of which are being used in production by a wide variety of commercial and open source projects as well as being widely used in academic research. Code in the LLVM project is licensed under the "Apache 2.0 License with LLVM exceptions"

The primary sub-projects of LLVM are:

1. The **LLVM Core** libraries provide a modern source- and target-independent optimizer, along with code generation support for many popular CPUs (as well as some less common ones!) These libraries are built around a well specified code representation known as the LLVM intermediate representation ("LLVM IR"). The LLVM Core libraries are well documented, and it is particularly easy to invent your own language (or port an existing compiler) to use LLVM as an optimizer and code generator.
2. **Clang** is an "LLVM native" C/C++/Objective-C compiler, which aims to deliver amazingly fast compiles, extremely useful error and warning messages and to provide a platform for building great source level tools. The Clang Static Analyzer and clang-tidy are tools that automatically find bugs in your code, and are great examples of the sort of tools that can be built using the Clang frontend as a library to parse C/C++ code.
3. The **LLDB** project builds on libraries provided by LLVM and Clang to provide a great native debugger. It uses the Clang ASTs and expression parser, LLVM JIT, LLVM disassembler, etc so that it provides an experience that "just works". It is also blazing fast and much more memory efficient than GDB at loading symbols.
4. The **libc++** and **libc++ ABI** projects provide a standard conformant and high-performance implementation of the C++ Standard Library, including full support for C++11 and C++14.
5. The **libc** project provides a high-performance, standards-conformant implementation of the C Standard Library, fully integrated with LLVM. It delivers optimized performance and comprehensive support for modern C standards, ensuring a reliable and efficient foundation for C applications.
6. The **compiler-rt** project provides highly tuned implementations of the low-level code generator support routines like "`__fixunsdfdi`" and other calls generated when a target doesn't have a short sequence of native instructions to implement a core IR operation. It also provides implementations

Latest LLVM Release!

10 September 2025: LLVM 21.1.1 is now **available for download**! LLVM is publicly available under an open source License. Also, you might want to check out **the new features** in Git that will appear in the next LLVM release. If you want them early, download LLVM through anonymous Git.

Upcoming Events

October 27-29 2025 US LLVM

ACM Software System Award!

LLVM has been awarded the **2012 ACM Software System Award**! This award is given by ACM to *one* software system worldwide every year. LLVM is in highly distinguished company! Click on any of the individual recipients' names on that page for the detailed citation describing the award.

Upcoming Releases

LLVM Release Schedule:

- 21.1.x
  - Jul 15th, 2025: release/21.x branch was created
  - Jul 17th, 2025: 21.1.0-rc1 was released
  - Jul 29th, 2025: 21.1.0-rc2 was released
  - Aug 12th, 2025: 21.1.0-rc3 was released
  - Aug 26th, 2025: 21.1.0 was released
  - Sep 10th, 2025: 21.1.1 was released
  - Sep 23rd, 2025: 21.1.2
  - Oct 7th, 2025: 21.1.3
  - Oct 21st, 2025: 21.1.4
  - Nov 4th, 2025: 21.1.5
  - Nov 18th, 2025: 21.1.6
  - Dec 2nd, 2025: 21.1.7

The Development of Compilation Technology

Huawei Ark Compiler (OpenArkCompiler): A Decade of Dedicated Development

- In 2009, Huawei launched research into foundational 5G technologies and simultaneously began forming its compiler team. The first batch of domestic and international researchers joined.
- In 2013, Huawei introduced its self-developed compiler HCC for the base station domain and officially proposed the concept of a compiler framework.
- In 2014, numerous well-known experts from around the world joined Huawei, and the Ark Project was officially initiated.
- In 2016, the Compiler and Programming Language Lab was established.
- In 2019, the Huawei Ark Compiler (OpenArkCompiler) was officially released!

Outline

- ✓ What is a Compiler
- ✓ **Overview of the Compilation Process**
- ✓ Structure of a Compiler
- ✓ Compiler Construction Techniques
- ✓ Summary

Overview of the Compilation Process

- Understanding the five main stages of compilation is fundamental. These stages are compared to English-to-Chinese translation for better understanding:

Language Translation	Compilation
1. Recognize words	1. Lexical Analysis
2. Parse grammar	2. Syntax Analysis
3. Interpret meaning	3. Semantic Analysis & Intermediate Code
4. Refine expression	4. Optimization
5. Final translation	5. Target Code Generation

1, Lexical Analysis

- The lexical analyzer is also known as a **scanner**.
- **Task**:
  To read the character stream of the source program and identify tokens (also called lexical symbols or simply symbols), such as identifiers, keywords, constants, and delimiters, and convert them into internal representations.
  - **Input**: Character stream from the source program
  - **Output**: Tokens in internal format, i.e., attribute pairs
    - **(Token-name, Attribute-value)**
      - Token-name: abstract name
      - Attribute-value: pointer to the symbol table
- **Effective tools for describing lexical rules**:
  - **Regular grammer**
  - Regular expressions
  - Finite automata
- **Methods**:
  - State diagrams
  - **DFA** (Deterministic Finite Automaton)
  - **NFA** (Nondeterministic Finite Automaton)

1, Lexical Analysis Example

A fragment of a C source program

```
int a,b,c;
a=a+2;
```

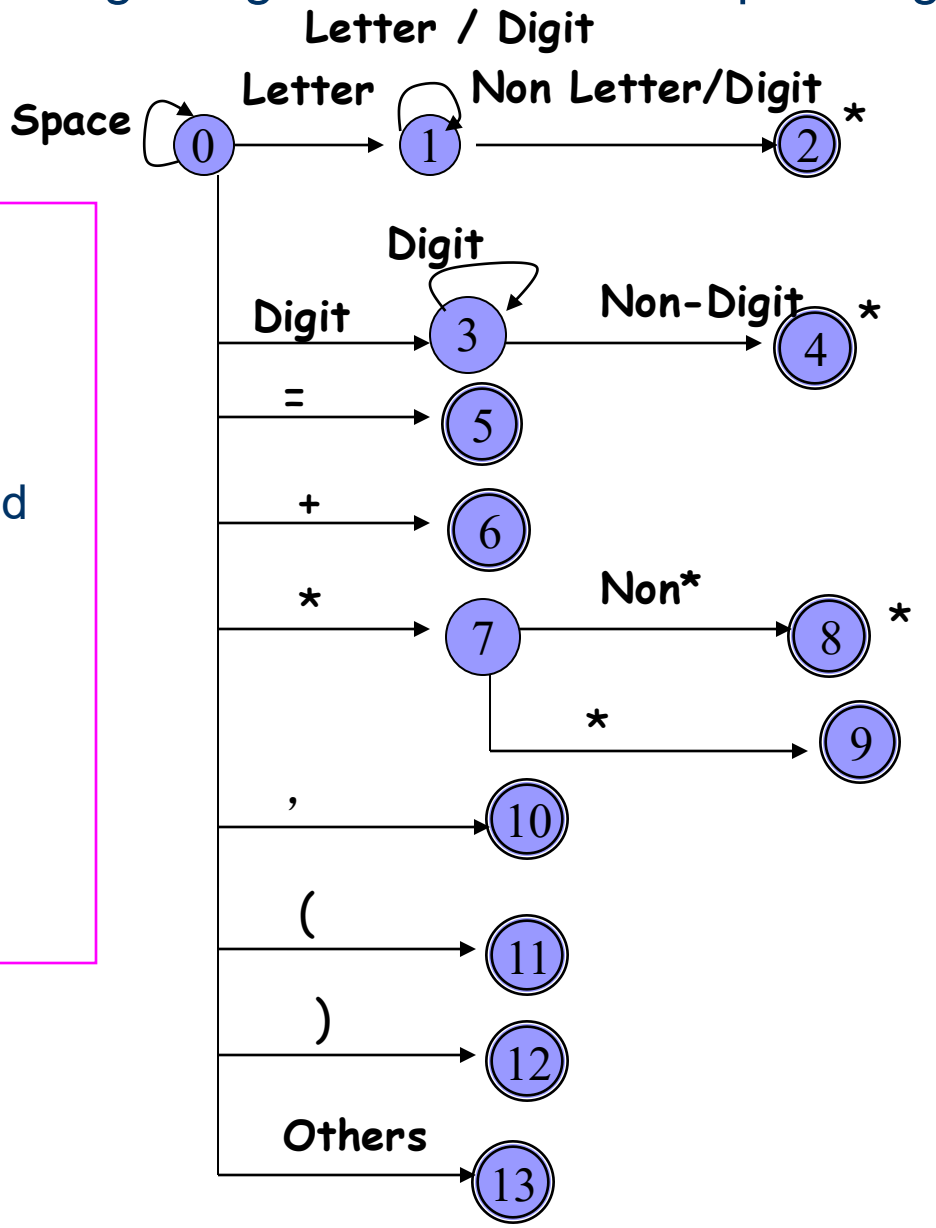
Symbolic Table

1	a	. . .
2	b	. . .
3	c	. . .

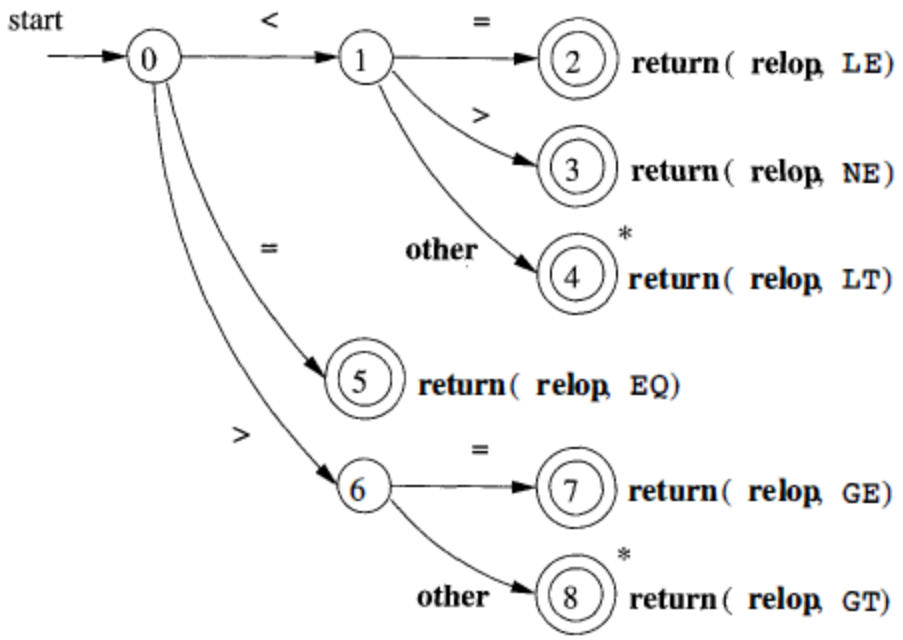
Token Type	Token Value	
Keyword	int	<int>
Identifier	a	<id,1>
Delimiter	,	<,>
Identifier	b	<id,2>
Delimiter	,	<,>
Identifier	c	<id,3>
Delimiter	;	<;>
Identifier	a	<id,1>
Operator	=	<op,EQ>
Identifier	a	<id,1>
Operator	+	<+>
Integer	2	<2>
Delimiter	;	<;>

State Transition Diagram for Recognizing All Tokens in a Simple Language

- **Assumptions**
  - Keywords are treated as **reserved words**.
  - Reserved words are handled as identifiers and distinguished using a **reserved word table**.
  - If there is no operator or delimiter between keywords, identifiers, or constants, a **space** is added.



Example




```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character proce
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

```python
class DFA:
    def __init__(self):
        # 状态集合
        self.states = ['S0', 'S1', 'S2']
        # 初始状态
        self.initial_state = 'S0'
        # 接受状态集合
        self.accepting_states = ['S2']
        # 状态转换表
        self.transition_table = {
            'S0': {'a': 'S1', 'b': 'S0'},
            'S1': {'a': 'S1', 'b': 'S2'},
            'S2': {'a': 'S1', 'b': 'S0'}
        }
        self.current_state = self.initial_state

    def process_input(self, input_string):
        for symbol in input_string:
            if symbol in self.transition_table[self.current_state]:
                self.current_state = self.transition_table[self.current_state][symbol]
            else:
                print(f"Invalid transition from {self.current_state} on symbol {symbol}
                return False
        return self.current_state in self.accepting_states

# 测试
dfa = DFA()
input_string = "ab"
if dfa.process_input(input_string):
    print("String accepted")
else:
    print("String rejected")
```

2, Syntax Analysis

- The syntax analyzer is also known as a **parser**
- **Task**:
  To read the tokens recognized by the lexical analyzer and, based on given grammar rules, identify grammatical units (such as phrases, clauses, statements, program segments, and programs), and generate another form of internal representation.
  - **Input**: Tokens identified and transformed by the lexical analyzer
  - **Output**: Another internal representation, such as **a syntax tree** or other intermediate forms
- Grammar rules are usually described using context-free grammars.
- **Methods**:
  - **LL(1)** parsing
  - **LR** parsing
  - Operator precedence parsing

Example: sum:= first + count * 10

- Grammar Rules:

<assignment> → <identifier> “:=” <expression>

<expression> → <expression> “+” <expression>

<expression> → <expression> “*” <expression>

<expression> → “(” <expression> “)”

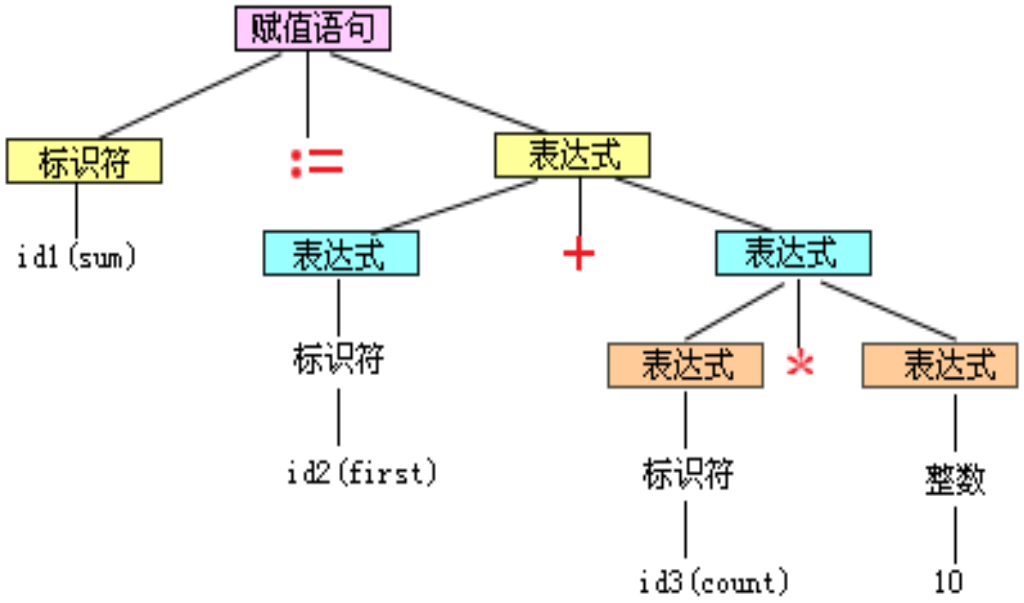
<expression> → <identifier>

<expression> → <integer>

<expression> → <real number>

Syntax Tree for the Example

sum:= first + count * 10



3-1, Semantic Analysis

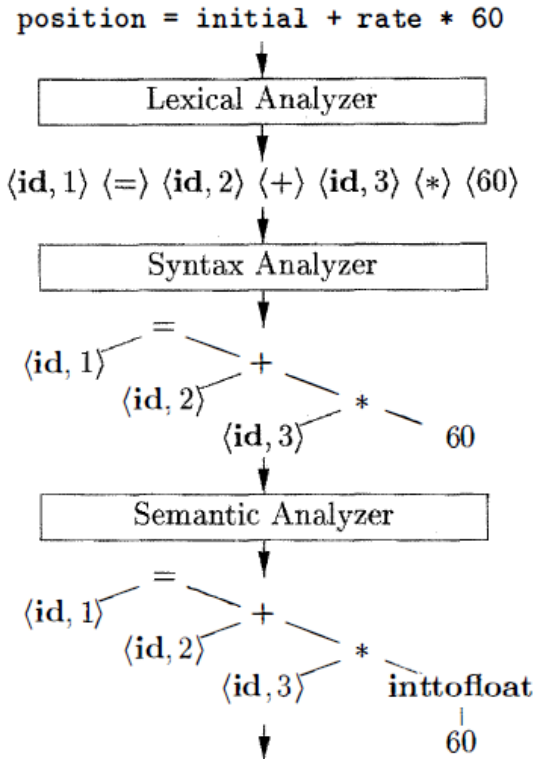
- Performs **static semantic checks** on the syntax tree or other intermediate representations. If the check passes, it proceeds to intermediate code translation.
- Semantic analysis processes statements one by one, following the hierarchical structure and order of the syntax tree.
- **Main Tasks**:
  - **Type checking**, Verifying whether each **operator** conforms to language rules, Reporting errors if not conforming
    - Ensuring variables are declared
    - Ensuring types are correct

Semantic Analysis Example

A syntax tree is shown with semantic processing nodes inserted.

Symbolic Table

1	**position**	float . . .
2	**initial**	float . . .
3	**rate**	float . . .



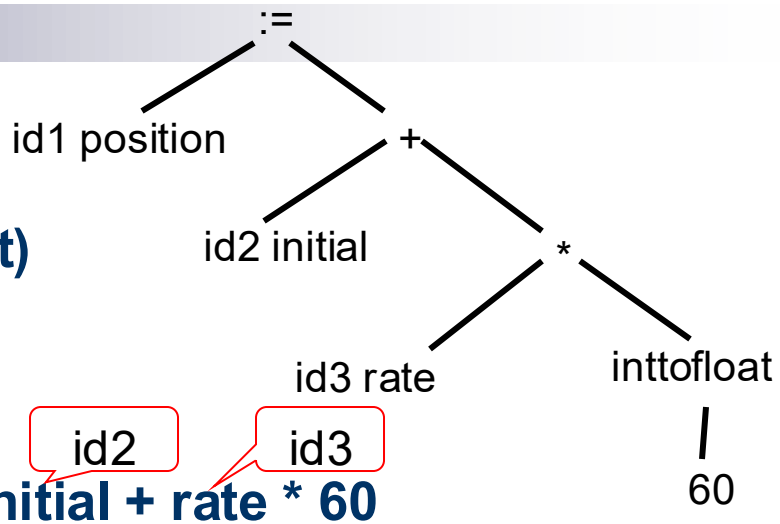
From: Compilers Principles Techniques and Tools(2nd Edition)

3-2, Intermediate Code Generation

- Intermediate code is a symbolic system **independent of specific hardware**. It may resemble machine instructions or can be easily translated into machine code.
- **Task**:
  Translate grammatical units such as “expressions,” “statements,” and “programs” into intermediate code sequences.
  - **Input**: Sentences/Grammer tree
  - **Output**: Intermediate code sequences
- **Forms**:
  - Quadruples (four-address code)
  - Triples (three-address code)
  - Reverse Polish notation
- **Methods**:
  - Semantic subroutines
  - DAGs (Directed Acyclic Graphs)
  - **Syntax-directed translation**

Format of a quadruple:

(operator, operand1, operand2, result)



Example: Source program position := initial + rate * 60

Generated quadruples:

(1) (inttofloat, 60 , — , t1)

(2) (* , id3 , t1 , t2)

(3) (+ , id2 , t2 , t3)

(4) (:= , t3 , — , id1)

Where:

- id1, id2, id3 represent the internal representations of position, initial, rate
- t1, t2, t3 are temporary variables representing intermediate results

- **Example 2**: C source code a = b * c + b * d
- Three-address sequence (quadruples in assignment form):

(1) t1 := b*c	(*,b,c,t1)
(2) t2 := b*d	(*,b,d,t2)
(3) t3 := t1 + t2	(+,t1,t2,t3)
(4) a := t3	(:=,t3-,a)

Source Code：

```
if (a<=b)
   a=a-c;
else c=b*c;
```


Quadruples：

```
100  (j<=,a,b,102)
101  (j, _, _, 104)
102  (-, a, c, t1)
103  (=, t1, _, a)
104  (*, b, c, t2)
105  (=, t2, _, c)
```

Error?

4，Optimization

- The task of optimization is to refine the intermediate code generated in previous stages into an equivalent form that is more efficient — aiming for faster execution or reduced memory usage.
- Optimization follows the principle of **program equivalence transformation**.
- **Methods include**:
  - Removing dead (useless) code
  - Eliminating common subexpressions
  - **Loop** optimization
  - ……

Example

id1 id2 id3

position = initial + rate*60

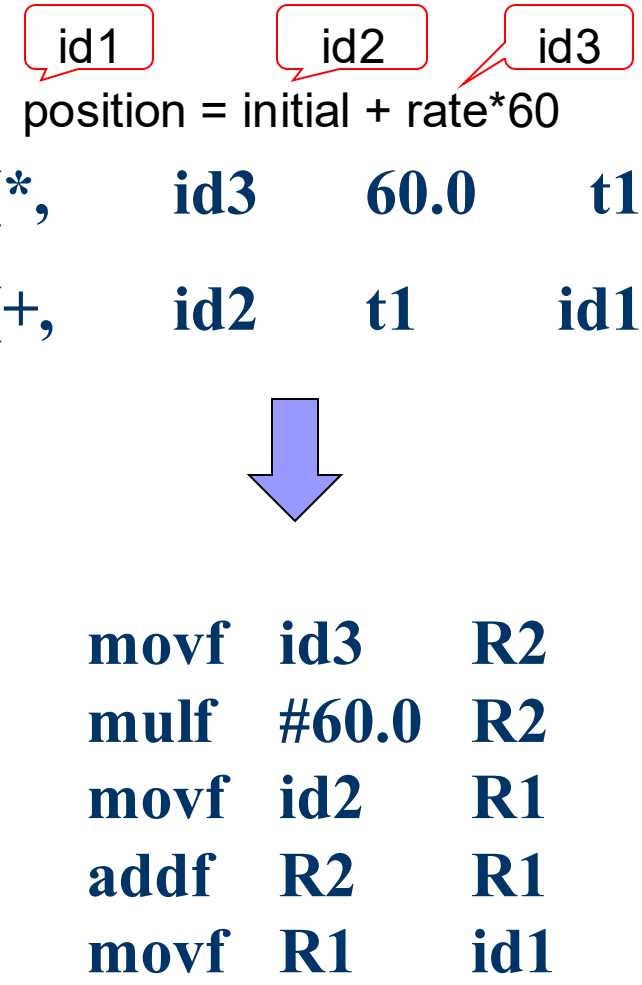
(1)	(inttofloat	60	-	t1	)
(2)	(*	id3	t1	t2	)
(3)	(+	id2	t2	t3	)
(4)	(:=	t3	-	id1	)

⇒

（1）	(*	id3	60.0	t1	)
（2）	(+	id2	t1	id1	)

5, Target Code Generation

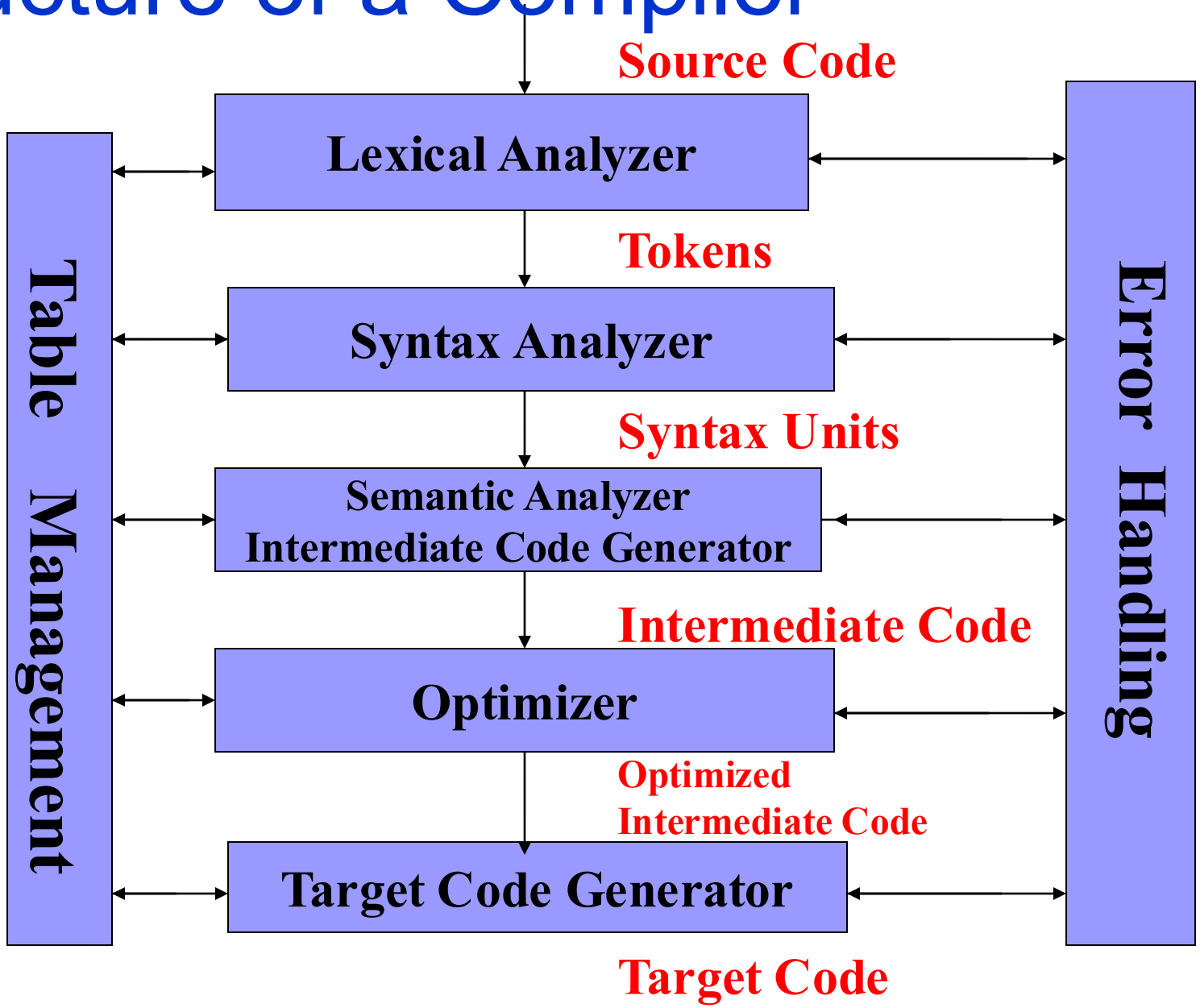
- This stage generates low-level code for **a specific machine** based on the intermediate code (possibly optimized). It depends on
  - the **hardware architecture**
  - the **machine instructions**



Outline

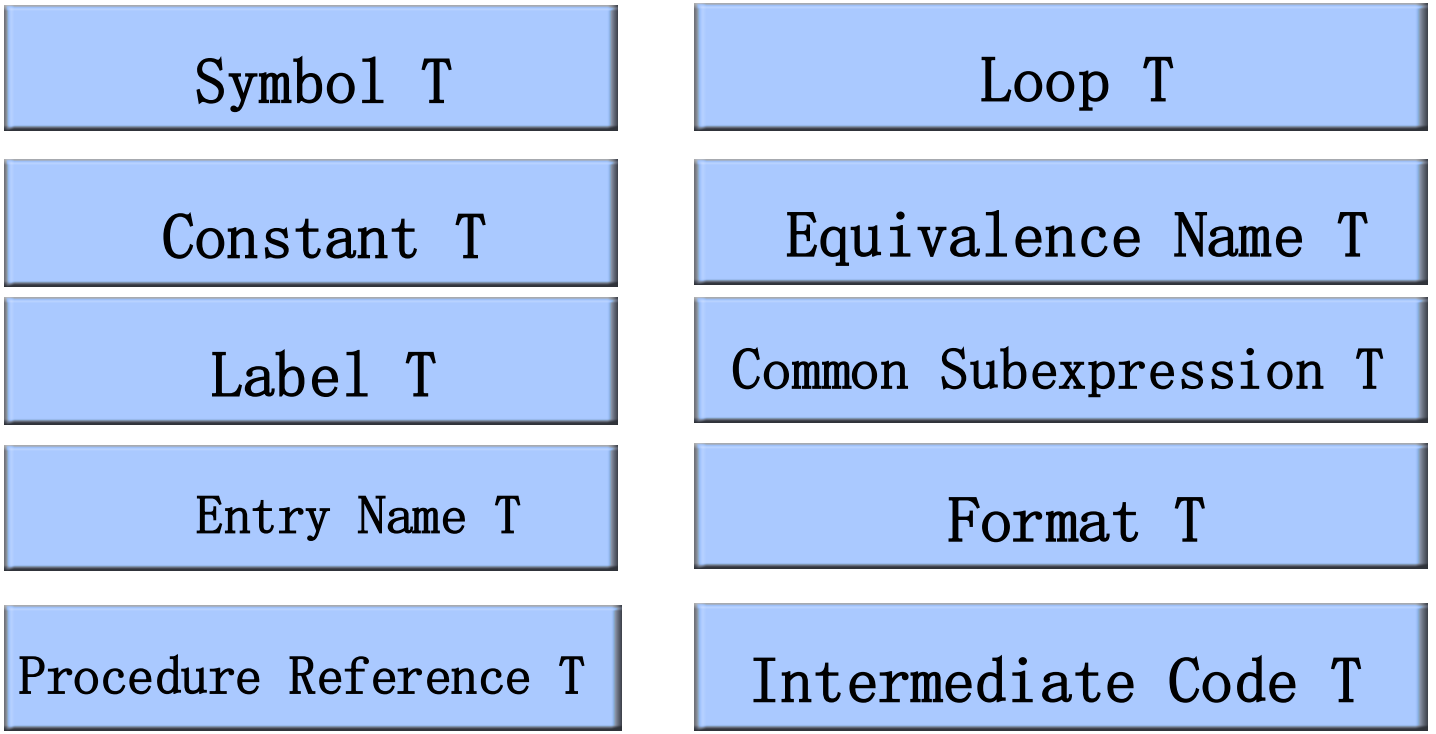
- ✓ What is a Compiler
- ✓ Overview of the Compilation Process
- ✓ **Structure of a Compiler**
- ✓ Compiler Construction
- ✓ Summary

Structure of a Compiler



Tables and Table Management

- A compiler uses several tables



Symbol Table

- The most important is the **symbol table**, which:
- Records names (identifiers) used in the source program
- Collects attribute information for each name
  - Type
  - Scope
  - Storage allocation details

Name	Type	Data Type	Level	Offset
m	Procedure		0	
id1	Variable	real	1	d
id2	Variable	real	1	d+4
id3	Variable	real	1	d+8

Error Handling

- Program errors fall into two main categories:

(1) Syntax Errors

Detected during lexical or syntax analysis

Examples:

- Misspelled reserved words:
  ~~*DIMENTOIN*~~ A(10) (instead of DIMENSION)
- Mismatched parentheses:

  F: WRITE (6,10)(A(1,J), J=1,10), I=1,10)

  T: WRITE (6,10)(A(1,J), J=1,10), **(**I=1,10)

Error Handling

(2) Semantic Errors

- Some can be detected at compile time, others only at runtime.
- Examples of **compile-time** semantic errors:
  - Undeclared **identifiers** used
  - Referenced **labels** not defined
  - Mismatches in number, type, or position of **formal & actual parameters**
- Examples of **runtime** semantic errors:
  - Array index out of bounds
  - Arithmetic overflow
  - Invalid arguments for standard functions

Error Handling

A good compiler should:

- **Comprehensive**: Detect as many errors as possible.
- **Accurate**: Precisely indicate the nature and location of the error.
- **Localized**: Limit the impact of an error to the smallest possible scope.

It is even better if the compiler can **automatically correct** errors, but this comes at a very high cost.

Combining Compilation Stages

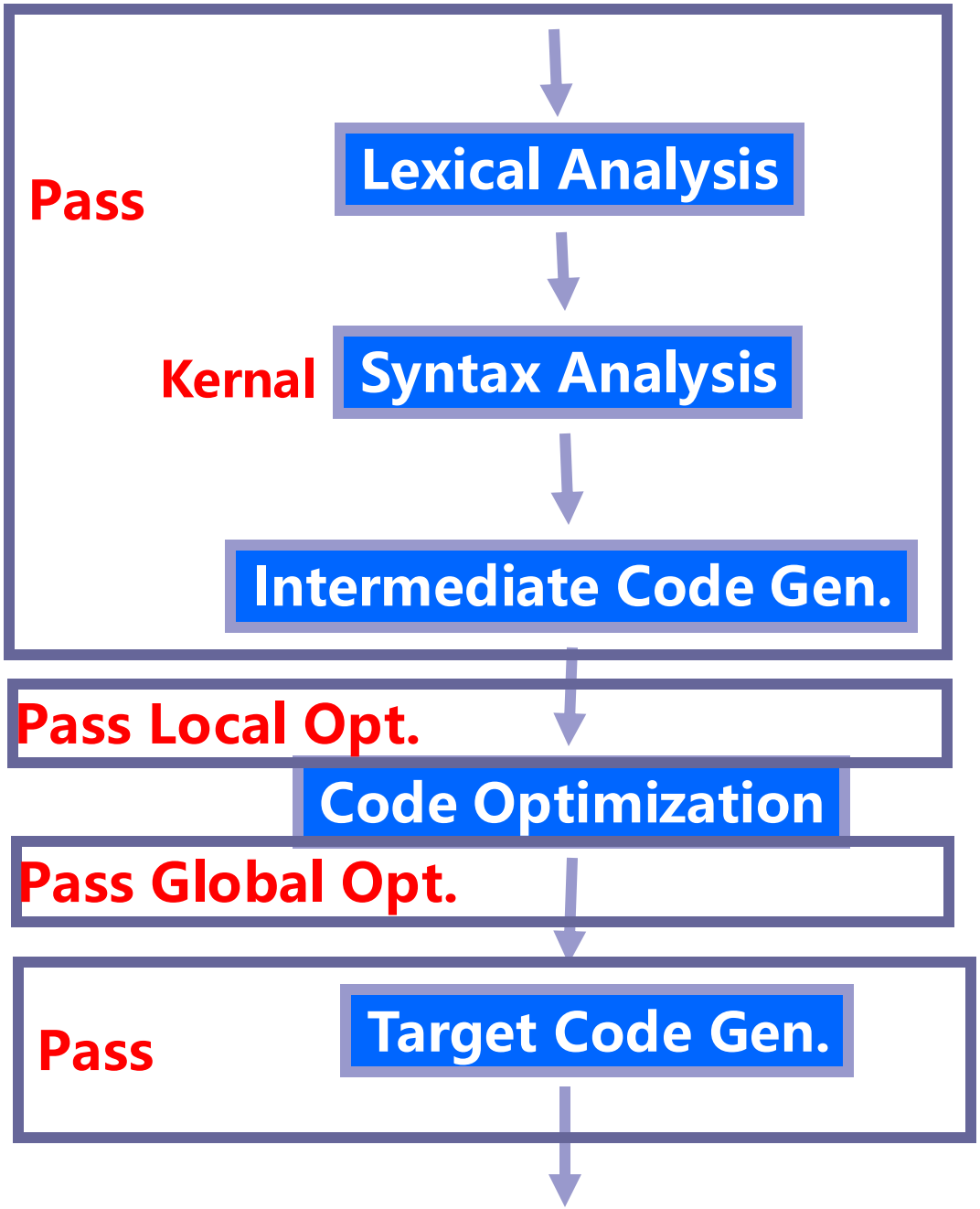
- From a logical perspective, a compiler can be divided into several stages. However, in implementation, multiple stages are often combined.
- For example:
  - **Front-End**:
    Lexical analysis, syntax analysis, semantic analysis, intermediate code generation, and optimization (machine-independent)
  - **Back-End**:
    Target-machine-specific optimization and target code generation

Source code related

Target code related

Pass

- A **pass** is a complete cycle of **reading input** (e.g., source code or intermediate result) and **producing output.**
  - The input is read from external storage
  - After processing, results are written back to external storage
- Passes allow logical separation of stages.



Number of Passes vs. Performance

- A compiler may complete its task in one pass, two passes, or multiple passes.
- **Multi-pass compilation**:
  - Less memory usage
  - More time required
- **Single-pass compilation**:
  - More memory usage
  - Less time required

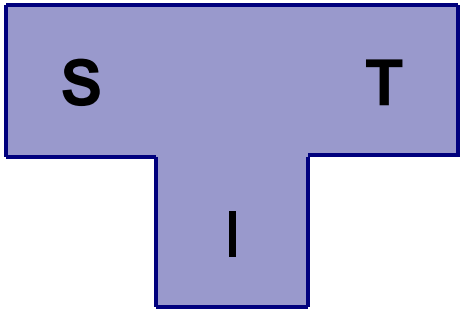
Outline

- ✓ What is a Compiler
- ✓ Overview of the Compilation Process
- ✓ Structure of a Compiler
- ✓ **Compiler Construction**
- ✓ Summary

Compiler Implementation Languages

- **Machine Language**
- **Assembly Language**
  - **Fully utilizes specific hardware capabilities**
  - **Meets platform-specific requirements**
- **High-Level Languages**
  - **Greatly reduce programming effort**
  - **Easier to read, maintain, and port the compiler**

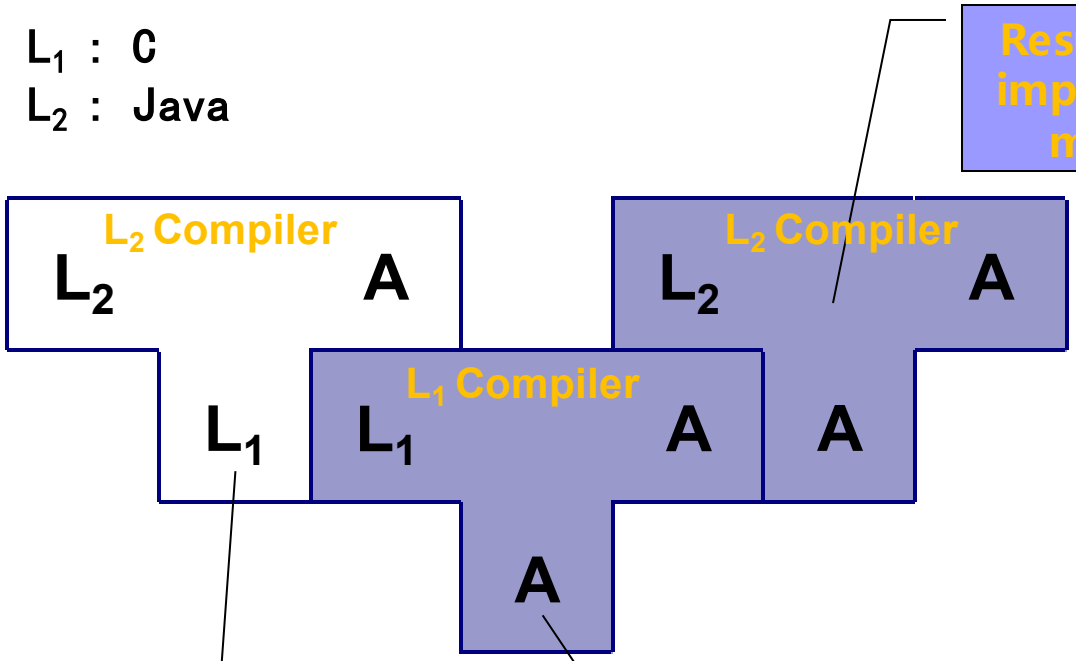
T-Shape Diagram



- Definitions:
  - S: Source language (program)
  - T: Target language (program)
  - I: Implementation language

Example: Developing a Compiler for a New Language

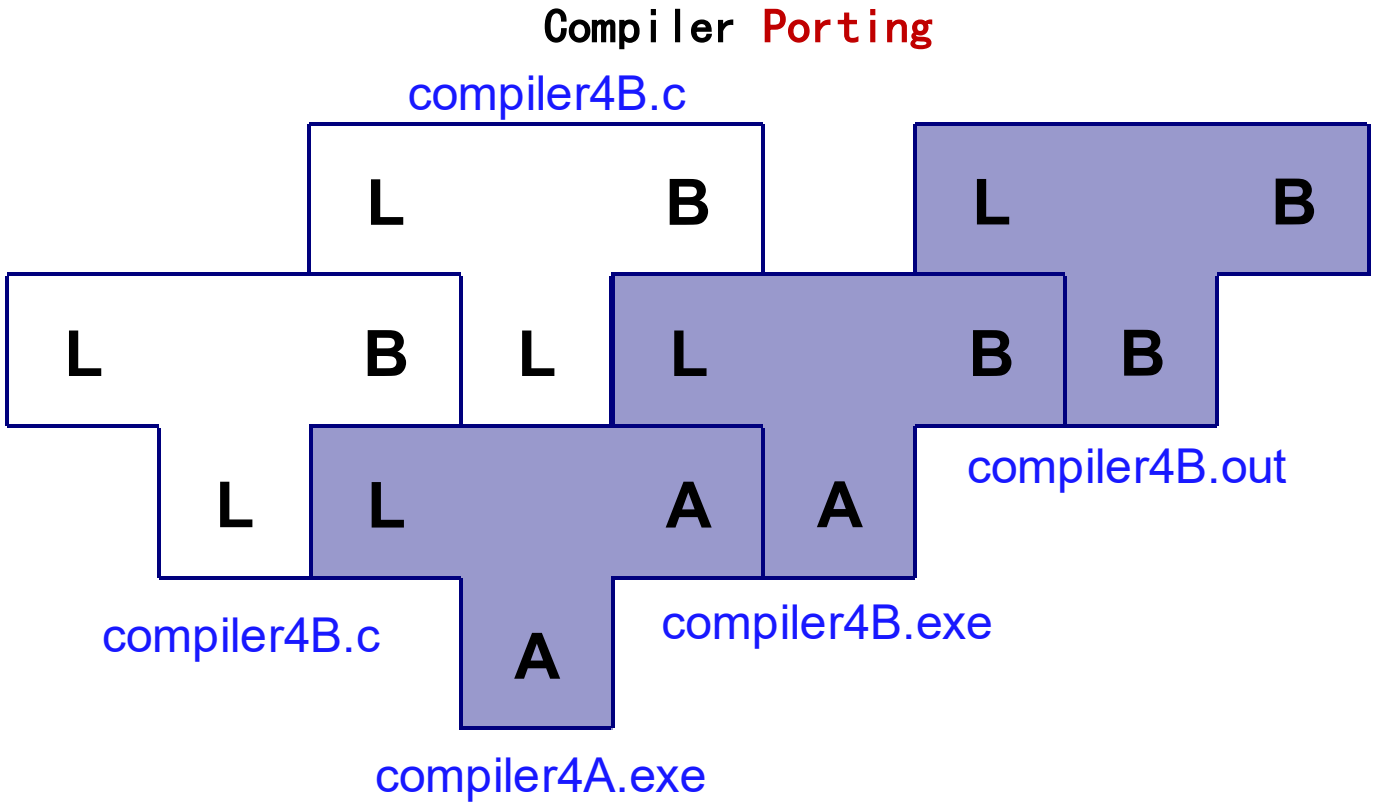
L_1 : C
L_2 : Java



教材原文：如果A机器上已有一个用A机器码实现的某高级语言L1的编译程序，则我们可以用L1语言编写另一种高级语言L2的编译程序，把写好的L2编译程序经过L1编译程序编译后就可得到A机器代码实现的L2编译程序

Example (Based on P9-10, Second Last Paragraph)

If Machine A already has a compiler, we can develop a compiler for Machine B.



L：C
A：Windows
B：MacOS

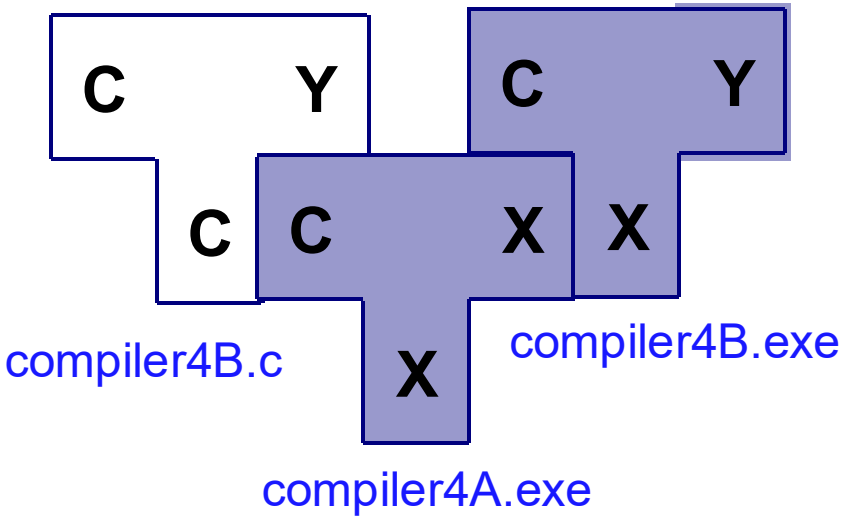
Assume we have:： compiler4A: .c → .exe
Our task： compiler4B: .c → .out

Self-Compilation

- Writing a compiler for a language using that **same language** is called **self-compilation**.
- Example:
  - If Machine A already has a C compiler
  - Write a more powerful C compiler in C
  - Use the existing compiler to compile the new compiler
  - Result: A more powerful C compiler that runs on Machine A

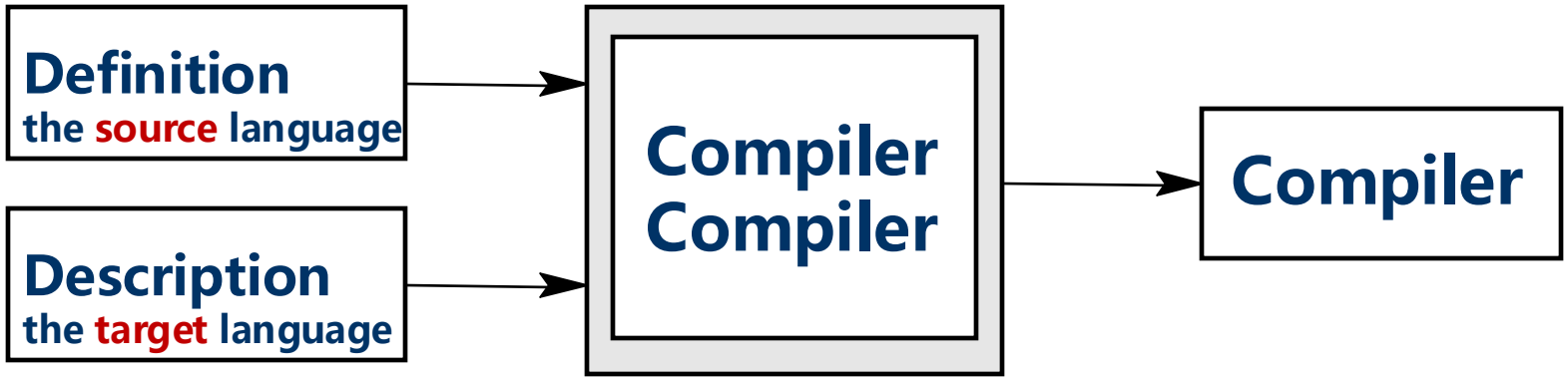
Cross Compilation

- **Cross compilation** means generating target code for Machine Y using a compiler running on Machine X.
- **Example**:
  - Machine X has a C compiler
  - Write a compiler in C on Machine X
  - That compiler outputs code for Machine Y
  - Result: Low-level code that runs on Machine Y
- **Applications**:
  - Embedded systems
  - Mobile operating systems



Automatic Compilation

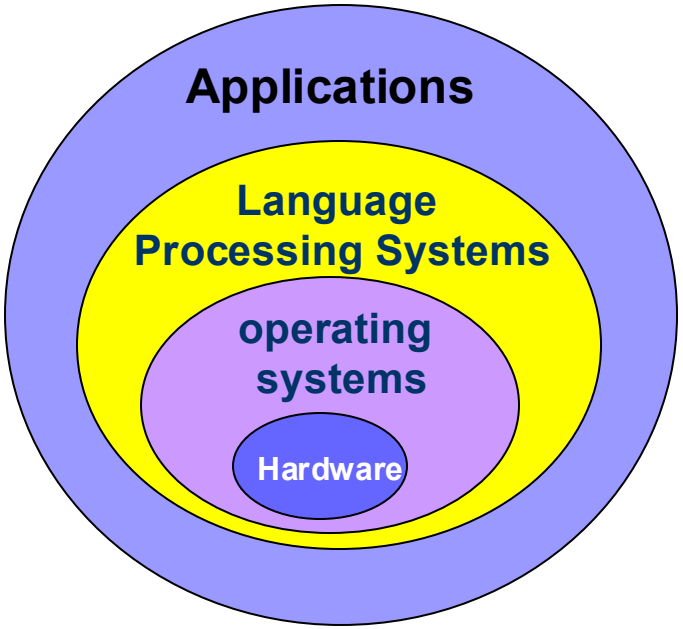
- Using specialized software tools, compilers can be **automatically generated** by providing:
  - A definition of the source language
  - A description of the target language



- **Examples of compiler generation tools**:
  - LEX / FLEX
  - BISON / YACC
  - ANTLR (Another Tool for Language Recognition)

The Role of Compilers in Computer Systems

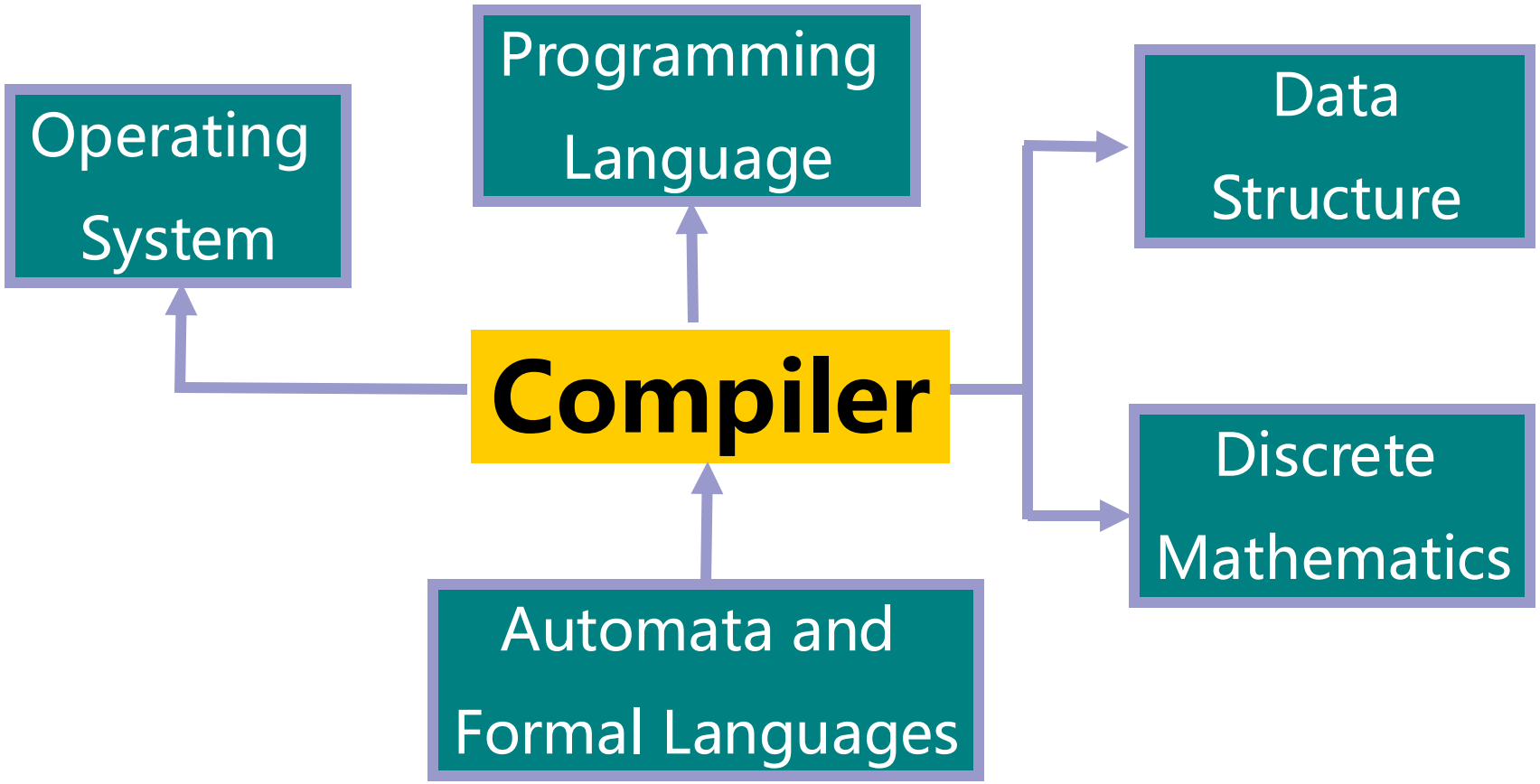
- The compiler system is a type of **system software**
- **System software:**
  - Lies closest to hardware
  - Enables other software to function
  - Independent of specific application areas
  - Includes compiler systems and operating systems
- **Language Processing Systems**: Convert programs written in high-level languages into executable code — i.e., compilers.



Outline

- ✓ What is a Compiler
- ✓ Overview of the Compilation Process
- ✓ Structure of a Compiler
- ✓ Compiler Construction
- ✓ **Summary**

The relationship



Dank u
Dutch

Merci
French

Спасибо
Russian

Gracias
Spanish

شكراً
Arabic

감사합니다
Korean

Tack så mycket
Swedish

धन्यवाद
Hindi

תודה רבה
Hebrew

Obrigado
Brazilian Portuguese

谢谢
Chinese

Dankon
Esperanto

Thank You !

ありがとうございます
Japanese

Trugarez
Breton

Tak
Danish

Danke
German

Grazie
Italian

நன்றி
Tamil

děkuji
Czech

ขอบคุณ
Thai

go raibh maith agat
Gaelic