Compilers
Principles and Practice

Dong Min

35866610@qq.com
1350706534

Compiler Construction
Principle and Pratice

Kenneth C. Louden

Compilers
Principles, Techniques,
& Tools
Alfred V. Aho

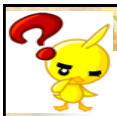
Stanford University
https://www.edx.org/course/compilers

https://space.bilibili.com/354384246?spm_id_from=333.788.b_765f7570696e666f.2

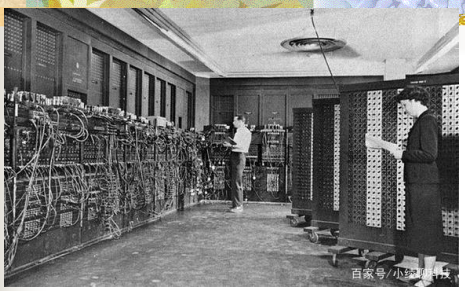
https://www.bilibili.com/video/BV1fh411C7A3

Chapter 1 Introduction

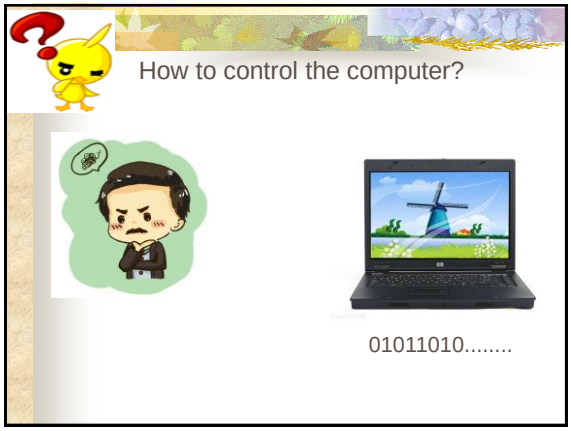
- **Study Goals:**
  - **Master: the phases of a compiler**
  - **Understand: what is a compiler**
  - **Know: interpreter,compiler structure**

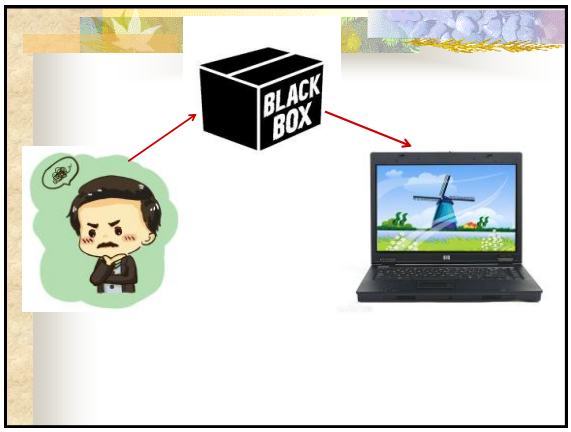


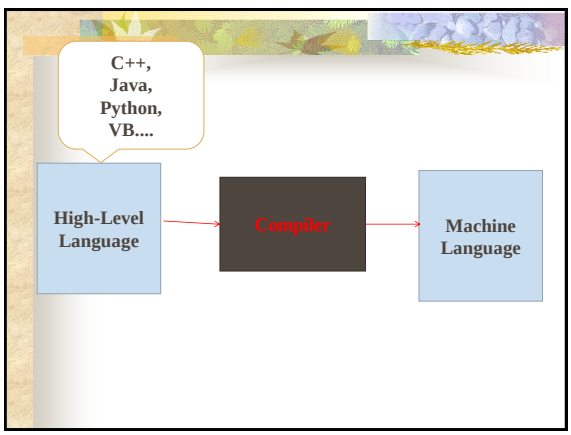
Why do we learn Compiler?



1946 : ENIAC(The Electronic Numerical Integrator And Computer)
30m * 6m * 2.4m

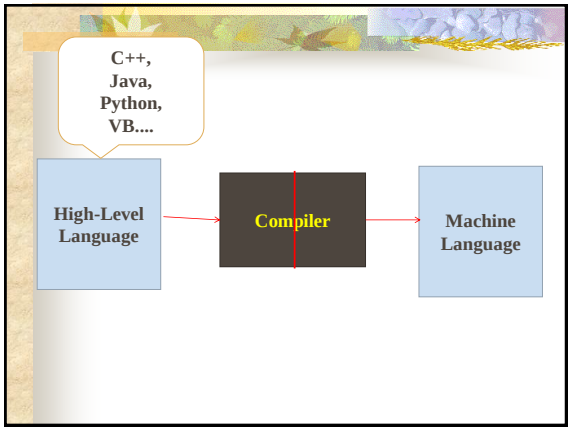


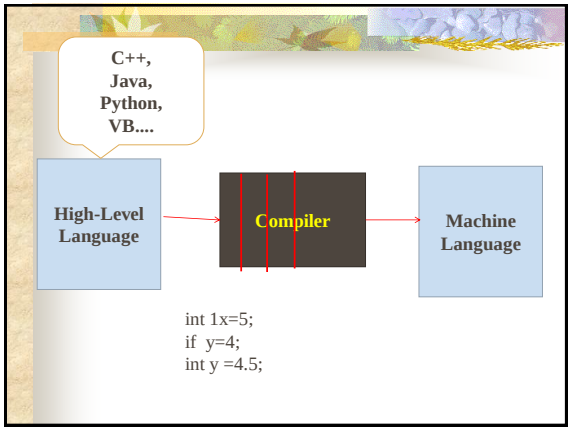




I am a teacher.
You are a studnet.
He a worker.

- Lexical error
- syntax error





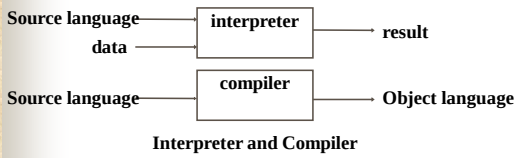
1.1 Why Compiler?A Brief History

- Why Compiler?
  - The development of programming language
    - Machine language (C7 06 0000 0002)
    - Assembly language (MOV x,2)
    - High-level language (x=2)
  - Computer can only execute the code written in the machine instructions of the computer. For high-level programs to be used, there must have a program to translate high-level programs to machine code

- What is Compiler?
  - Compilers are computer programs that translate one language to another
  - Source language:the input of a compiler,usually high-level language(such as C,C++)
  - Target language:the output of a compiler, usually object code for the target machine(such as machine code, assembly language)

Source Program → Compiler → Target Program

- Interpreter and Compiler
  - The same point:They are all language implementing system
  - Differences:
    - Interpreter executes the source program during translation
    - Compiler generates object code that is executed after translation completes



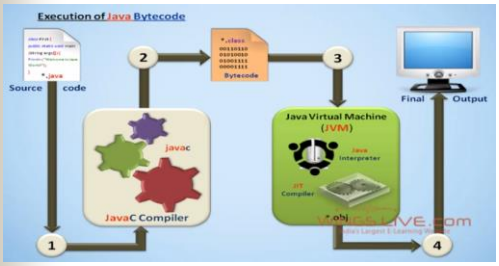
Interpreter and Compiler

- A brief history of compiler
  - The first compiler--FORTRAN compiler was developed in the 1950s
  - As grammars was found, the construction of compiler was partial automated in late 1950s
  - A fairly complete solution to parsing problem was completed in 1960s and 1970s
  - Resent work:
    - Code improvement techniques
    - IDE(Interactive development environment)
    - Parallel compiler

Questions

- Which one is more efficient, compiler or interpreter?
- Why Java is a platform independent language?

Questions



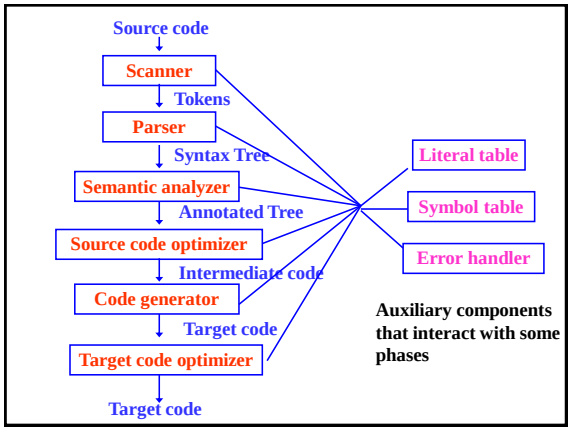
1.2 Programs Related to Compilers

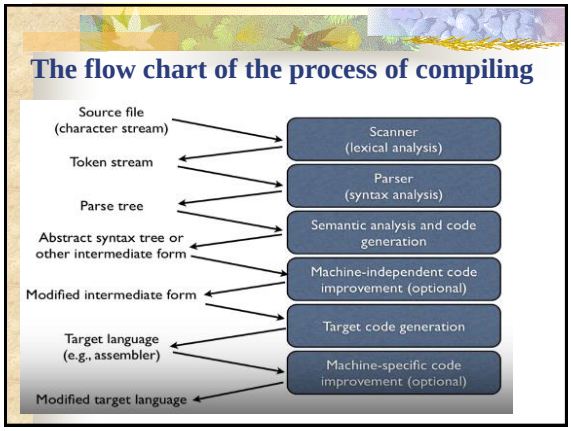
- The process of high-level programming language

Source program
↓
preprocessor
↓
Source program
↓
compiler
↓
Assembly language program
↓
assembler
↓
Relocatable machine code
↓
loader/linker ← Relocatable object files
↓
Absolute machine code

1.3 The Translation Process

- A compiler consists of a number of phases, each performs distinct logical operations
- The phases of a compiler is:
  - Scanner
  - Parser
  - Semantic Analyzer
  - Source Code Optimizer
  - Code Generator
  - Target Code Optimizer





1 The Scanner

- Scanner performs lexical analysis
- The Task of scanner:

  Reading the source program as a file of characters and dividing it up into meaningful units called tokens
  - Input:source program which is a stream of characters
  - Output:tokens

- Token
  - Each token is a sequence of characters that represents a unit of information in the source program.

- Tokens fall into several categories :
  - Identifier:user-defined strings, usually consist of letters and numbers and begin with a letter
  - Keyword:fixed strings of letters, such as “if”,“while”
  - Number
  - Operator:such as arithmetic operator ‘+’ and ‘×’, “>= ”, “<= ”
  - Special symbol:such as ‘(‘ ‘)’ ‘;’
- Token is presented as (Kind,Value)

Example
a[index]=4+2

Source program in file

a	[	i	n	d	e	x	]	=	4	+	2

Token stream after lexical analysis

(identifier,a)	(left bracket,[)	(identifier,index)	(right bracket,])
(assignment,=)	(number,4)	(plus sign,+)	(number,2)

- Other operations performed by scanner
  - Enter identifiers into the symbol table
  - Enter literals in literal table, literals include
    - Numeric constants(3.14)
    - Quoted string of text(“hello”)

2 The Parser

- Parser performs syntax analysis
- Task of parser

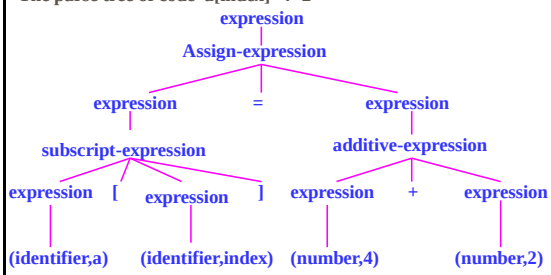
  The parser receives the source code in the form of tokens from the scanner and performs the syntax analysis ,which determines the structure of the program
  - Input:stream of tokens
  - Output:parse tree or syntax tree

- Parse tree

  Parse tree is a useful aid to visualizing the syntax of a program or program element

Example

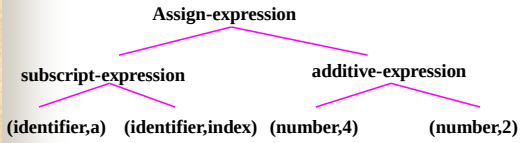
The parse tree of code a[index]=4+2



- Internal nodes are labeled by the names of the structures they represent
- Leaves represent the sequence of tokens from the input

- Abstract Syntax Tree(Syntax Tree):

  A condensation of the information contained in the parse tree



3 The Semantic Analyzer

- Semantic
  - Semantics of a program are its “meaning”
  - Static semantic: properties of a program that can be determined prior to execution
  - Dynamic semantic:properties of a program that can only be determined by execution
- Task of Semantic Analyzer

  Analyze static semantic

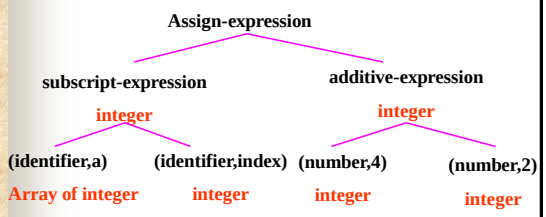
- Static Semantics Include:
  - Declarations
  - Type checking
- Semantic analysis is realized by symbol table. Attributes are entered into symbol table
- Output of Semantic Analyzer

  Semantic analyzer will produce an annotated tree. Attributes are added to the tree as annotations

- Example

  Semantic Analysis for expression a[index]=4+2 includes:
  - Whether identifiers 'a' and "index" have been declared?
  - Whether the type of 'a' is an array of integer values with subscripts from a subrange of the integers?
  - Whether the type of "index" is an integer?
  - Whether the left and right side of assignment have the same type?
  - Whether the two operands of additive operation have the same type?

Output of semantic analyzer for a[index]=4+2

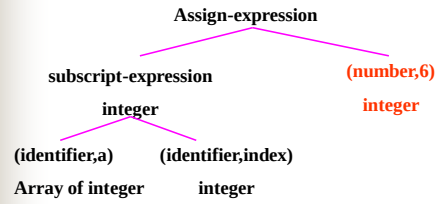


4 The Source Code Optimizer

- Optimization performed after semantic analysis that depend only on the source code
  - Some optimizations can be performed directly on the syntax tree
  - It is easier to optimize useing Intermediate Code

Example

Constant folding optimization can be performed for a[index]=4+2 on syntax tree,where 4+2 can be precomputed by 6



- Intermediate Code
  - A form of code representation intermediate between source code and object code
  - Intermediate codes have the following properties: structure is simple, meaning is clear, and it is easy to translate them to object code
  - For example

    Three-address code: it contains the addresses of three locations in memory

Example

Three-address code for a[index]=4+2 is:

t=4+2

a[index]=t

optimization performed on three-address code

three-address code after optimization is:

a[index]=6

- **IR(Intermediate Representation)**
  - **Any internal representation for the source code used by the compiler is called IR**
  - **The syntax tree and intermediate code are all IR**

5 The Code Generator

- **Task of Code Generator**
  - **The code generator takes the IR and generates code for the target machine**
  - **Usually use assembly language as target code**
  - **It is related to the properties of target machine: the number of registers, addressing mode, data representation and so on.**

Example

Target code for a[index]=6 in assembly language:

```
MOV R0,index
MUL R0,2
MOV R1,&a
ADD R1,R0
MOV *R1,6
```

These codes correspond to the following conventions on target machine:

- **An integer occupies two bytes of memory**
- **&a is the address of a**
- ***R means indirect register addressing**

6 The Target Code Optimizer

- **Task of the Target Code Optimizer**

  To improve the target code generated by the code generator, saving execution time and memory space
- **This Optimization includes**
  - **Change addressing mode to improve performance**
  - **Replace slow instructions by faster ones**
  - **Eliminate redundant or unnecessary operations**

Example

Target codes are:

```
MOV R0,index
MUL R0,2
MOV R1,&a
ADD R1,R0
MOV *R1,6
```

After optimization:

```
MOV R0,index
SHL R0
MOV &a[R0],6
```

7 Auxiliary Components of Compiler Phases

- The literal table
  - Usage of Literal Table

    Literal table stores constants and strings used in a program
  - Purpose of Literal Table

    The literal table is important in reducing the size of a program in memory by allowing the reuse of constants and strings

- The Symbol Table
  - Usage of Symbol Table

    Symbol table keeps information associated to identifiers: function, variable, constant and data type identifiers
  - Symbol Table with Compiler Phases
    - Scanner,parser or semantic analyzer may enter identifiers and information into the table
    - The optimization and code generation will use the information provide by the symbol table

- Error handler
  - Errors can be detected during almost every compiler phase
  - Error handler must generate meaningful error message and resume compilation after each error

1.4 Other Issues in Compiler Structure

- Compiler Structure

  The structure of the compiler will have a major impact on its reliability, efficiency, usefulness and maintainability

1 Analysis and Synthesis

- Analysis

  Analyze the source program to compute its properties, include:lexical analysis,syntax analysis,and semantic analysis
- Synthesis

  Operations involved in producing translated code,include:code generation

  Optimization steps may involve both analysis and synthesis

- Impact of this structure
  - Analysis and synthesis use different realization techniques
  - It is helpful to separate analysis steps from synthesis steps so each can be changed independently of the other

2 Front End and Back End

- **Front end**

  Operations that depend only on the source language,include:the scanner,parser,and semantic analyzer,source code optimizer
- **Back end**

  Operations that depend only on the target language, include :code generator,target code optimization

- **Impact of this Structure**

  This structure is especially important for compiler portability

- **Realization of compiler structure**

  Compiler may repeat one or more passes before generating target code
  - **Definition of Pass**

    A pass is to process of the entire source program or it's intermediate representation one time

- **A typical arrangement is one pass for scanning and parsing, one pass for semantic analysis and source-level optimization, and a third pass for code generation and target level optimization.**
- **How many times depend on the source language and the target machine**

Two weeks' task: watching the vedios:

- **1-Introduction**
  - **01-01: Introduction**
  - **01-02: Structure of a Compiler**
  - **01-03: optional**
- **3- Lexical Analysis**
  - **03-01: Lexical Analysis**
  - **03-02: optional**
  - **03-03: Regular Languages**
  - **03-04: optional**
  - **03-05: Lexical Specifications**