Crafting a Compiler with C (II)

資科系
林偉川

Compiler V. S. Interpreter

- **Compilation - Translate high-level program to machine code**
  Lexical Analyzer, Syntax Analyzer, Intermediate code generator(Semantics Analyzer) , Optimization , Code Generation ➔ use symbol table
  - **Slow translation + Linker**
  - **Fast execution**
  - **Von-Neumann bottleneck**
    connection between computer's memory and CPU

2

Compiler V. S. Interpreter

- **Pure interpretation for source-code debugger**
  - **No translation**
  - **Fetch/decode/execution cycle**
  - **Slow execution (every-time statement decoding)**

  LISP, shell of UNIX, Script language

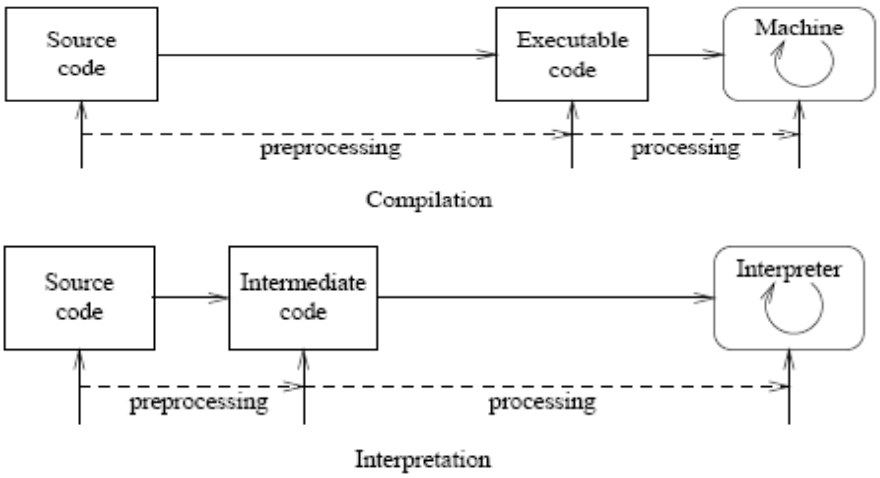
3

Compiler V. S. Interpreter

- Interpreter directly interpret the source programs they process or syntactically transformed of them
- A compiler has distinct translation and execution phase. The translation phase may generate a virtual machine language that is interpreted by a particular computer, either in Firmware or HW

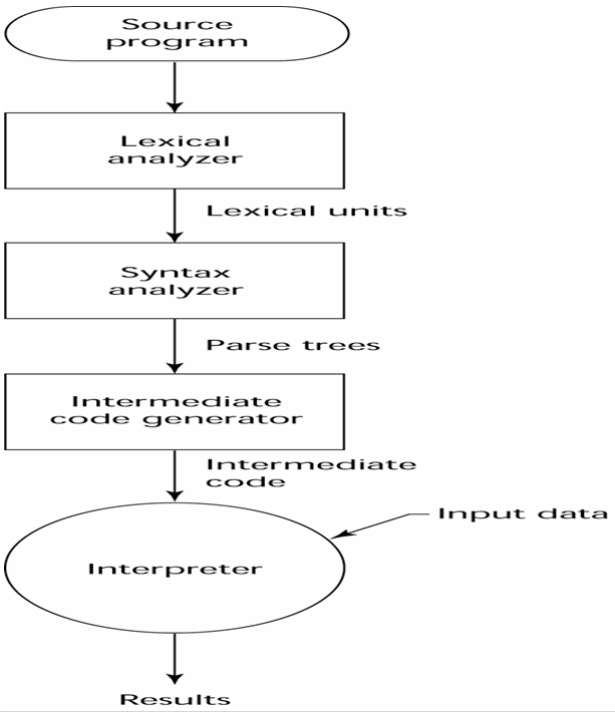
4

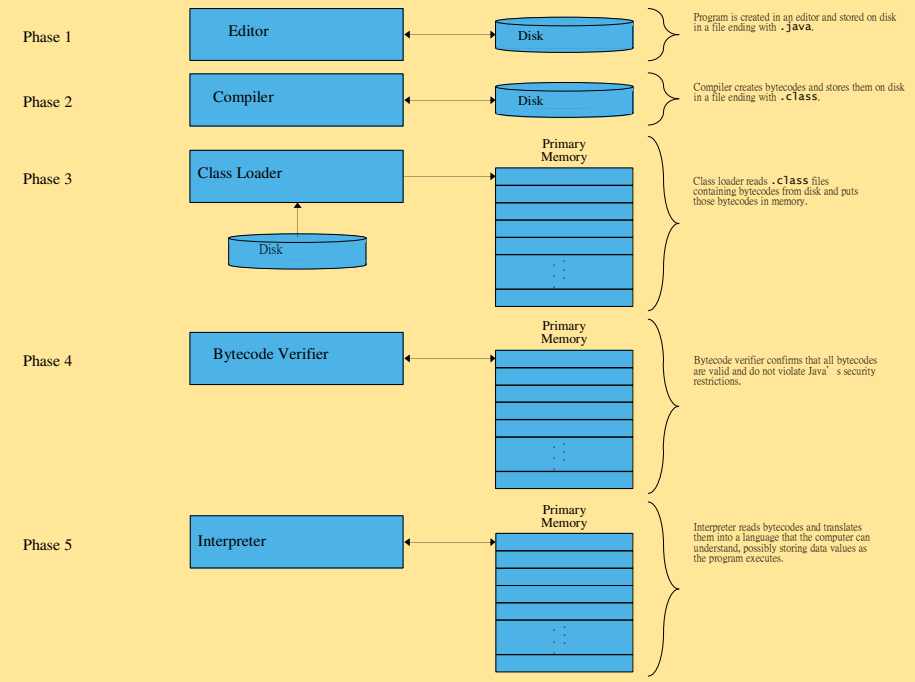
Compiler V. S. Interpreter



5

Hybrid Implementation system





Compiler Structure

- Scanner → Scan source program and build tokens such as identifier, integer, real, reserved words, delimiters (tokens are encoded as integer)
  - **Put program into a compact and uniform formats (token)**
  - **Eliminates unneeded information (comments)**
  - **Enter preliminary information into symbol and attribute tables**
  - **Formats and lists the source program**

  →use **regular expression** to describe tokens and can be converted into **Finite Automata** [scanner generator→Lex]

  Hand-coding scanner V.S. Table-driven scanner (generator produce)

8

Compiler Structure

- Parser → Build the syntactic structure defined by a programming language syntax from tokes
  - Parser reads tokens and group them into units as specified by the productions of **CFG**
  - Parsers are **driven by tables** created from CFG by a parser generator (Yacc)
  - The parser verifies correct syntax, if a syntax error is found, it can **repair the error or error recovery** by consulting **error repair table** created by parser/repair generator

9

Compiler Structure

- Semantics routine → supply the meaning of the program based on the syntactic structure to generate intermediate or target code
  - Check the **static semantics** of each construct to be legal, if the construct is correct, semantic routines also do the **actual translation** (intermediate code generation) → **attribute grammar**
  - Static semantic checking is **machine independent**, **intermediate code generation** is **machine dependent**
  - If two components are separated, **retargeting** a compiler to generate code for a **different target machine** is simplified

10

Compiler Structure

- Optimizer → intermediate code generation is analyzed and transformed into improved code by optimizer
  - **Very complex and slow**
  - **Maybe turn off to speed up translation**
  - **Usually it was skipped**
  - **Peephole optimization**
    - **Attempt to do simple, often machine dependent, code improvements**

    →subset optimizer to reduce the instructions:
    - **Elimination the instructions such as V * 1 or V + 0**
    - **Eliminate Store Reg_a into memory(A) , Load memory(A) to Reg_a**

11

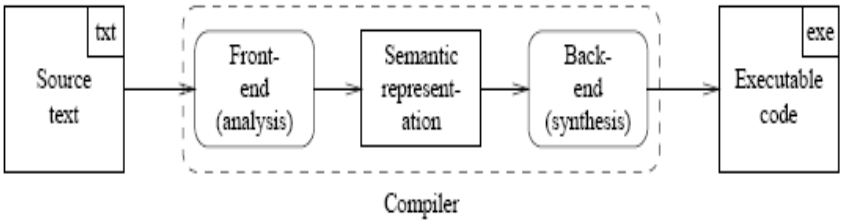
Compiler Structure

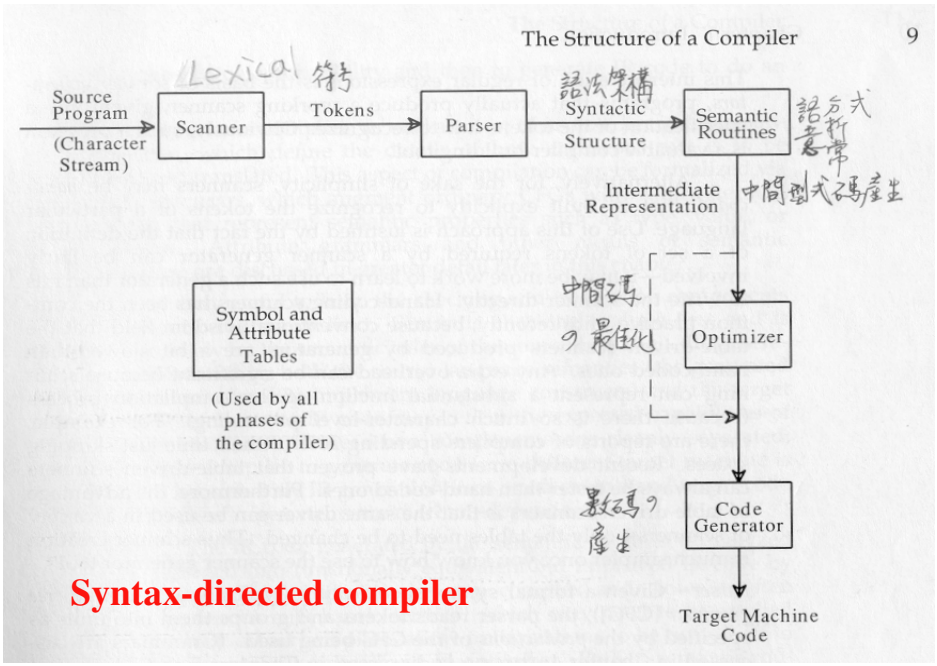
- Code generator → intermediate code is mapped into target code by code generator
  - **Need detailed information about the target machine**
    - **Register allocation**
    - **Addressing mode (direct/indirect)**
    - **Choice of instruction formats (RR, RM, MM)**
  - **Generation of good target machine code requires consideration of many special cases**

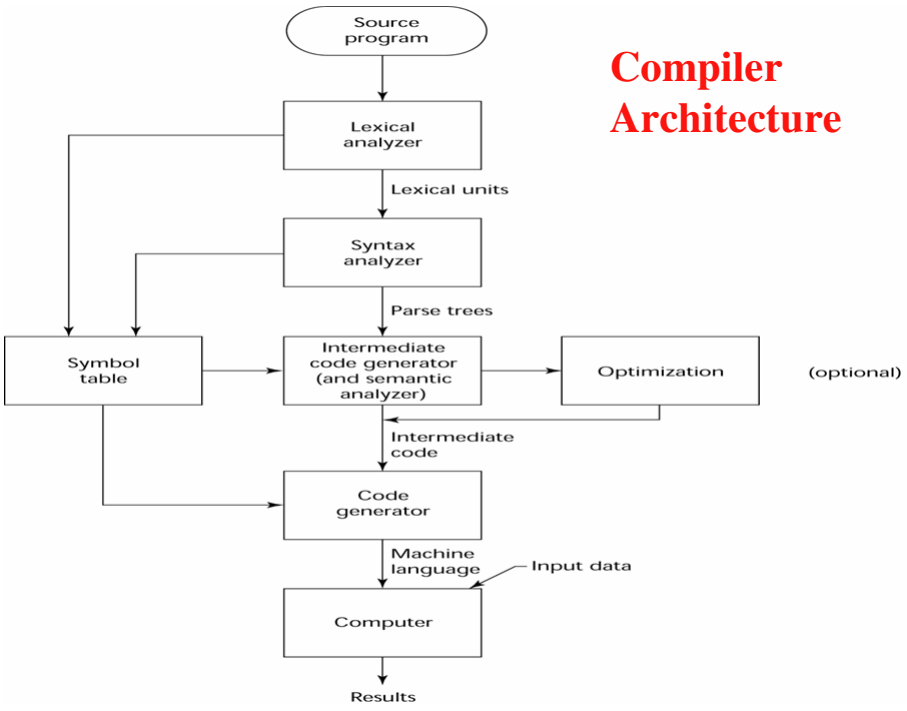
12

Compiler Structure

- Compilers analyze their input, construct a semantic representation, and synthesize their output from it. (analysis-synthesis paradigm)
- A program consists of a front-end which analyzes the text and constructs internally a table of (length, frequency) pairs







Compiler method

- One-pass compiler→no optimization is needed, just merge code generation with semantic routine and intermediate code
- Automatic generation of code generators has been widely studied ➔ automatically match a low-level IR to target instruction templates
- Localizes the target machine dependencies of a compiler and makes it possible to retarget a compiler to a new target machine (Assembly into another machine's)

16

Compiler method

- Symbol and attribute tables
  - A symbol table is a mechanism that allow information (attributes) to be associated with identifiers
  - Each time an identifier is used, a symbol table provides access to the information collected about the identifier when its declaration was processed
  - Symbol tables can be used by any of the compiler components to enter, share, and later retrieve information about variables, procedures name, labels, … etc.

17

Compiler method

- Compiler writing tools
  - Compiler-compilers or compiler generator includes scanner and parser generator and some also include symbol table routines and code-generation capability
  - These generators greatly aid in building piece of compilers, but the great bulk of the effort in building a compiler lies in writing and debugging semantic routines
  - Describe what a program should do and generate a program from it by a program generator

18

Advantage of using compiler generator

- Input to compiler generator is much higher level of abstraction than handwritten program. This increases the chance that the program will be correct
- Allows increasing flexibility and modifiability. If a small change in syntax of a language, a handwritten parser would be a major block to any such change. With a generated parser, just changes the syntax description and generates a new parser

19

Advantage of using compiler generator

- Pre-canned or tailored code can be added to the generated program, enhancing its power at hardly any cost. Input error handling is usually a difficult affair in handwritten parser. A generated parser can include tailored error correction code with no effort on the part of the part of the programmer

20

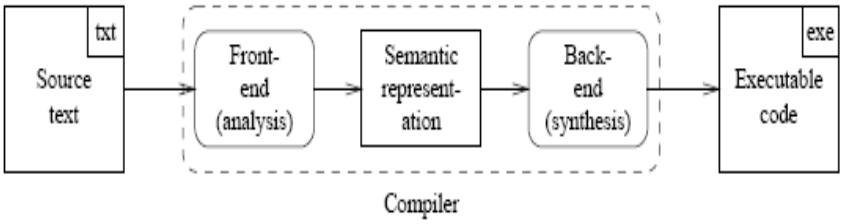
Advantage of using compiler generator

- A formal description can be used to generate more than one type of program
- Generated compiler program may be more or less efficient than handwritten one whenever the possibility exists. Generating a compiler program is always to be preferred
- Using a parser generator, a parser can be generated automatically. Especially for the file conversion system profited from compiler construction technique such as converting text file to .ps or .pdf

21

Compiler Structure

- Compilers analyze their input, construct a semantic representation, and synthesize their output from it. (analysis-synthesis paradigm)
- A program consists of a front-end which analyzes the text and constructs internally a table of (length, frequency) pairs



Semantic representation

- Semantic representation takes the "intermediate code" as data structure such as annotated abstract syntax tree (parse tree) ➔ AST
- Syntax tree (parse tree) of program text is a data structure which shows how the various segments of the program text are to be viewed in term of grammar
- Parsing is the process of structuring a text according to a given grammar

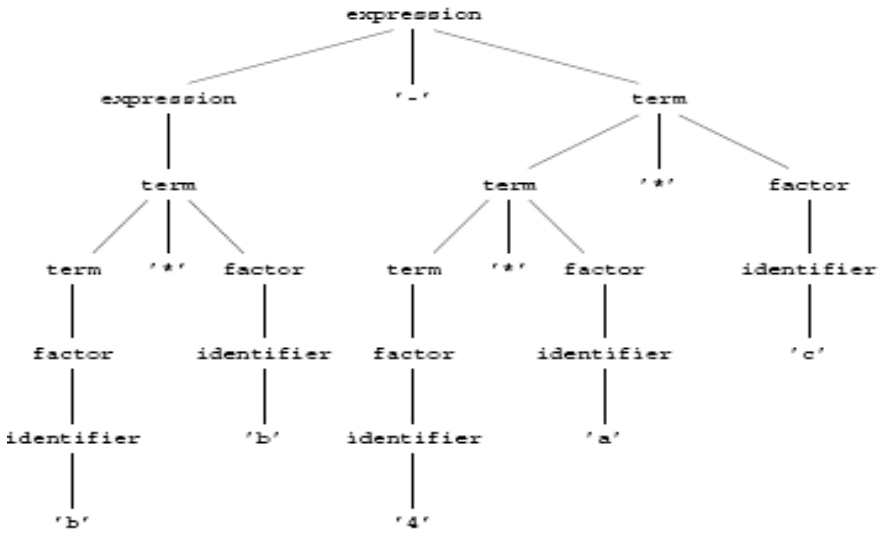
23

Semantic representation

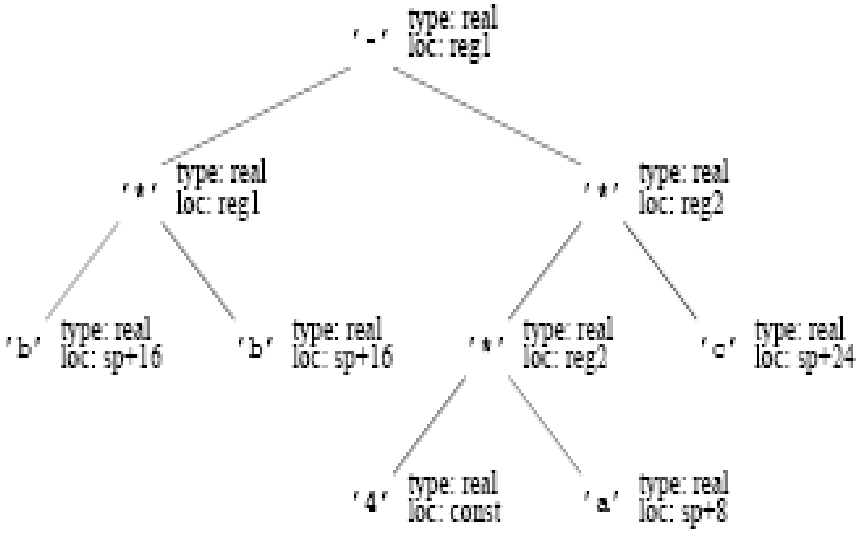
- The parser can be written by hand if the grammar is very small and simple. For larger and/or more complicated grammars it can be generated by a parser generator
- The expression b*b-4*a*c, the annotated parse tree according to the grammar shown as followed:
  expression → expression '+' term | expression '-' term | term
  term → term '*' factor | term '/' factor | factor
  factor → identifier | constant | '(' expression ')'

24

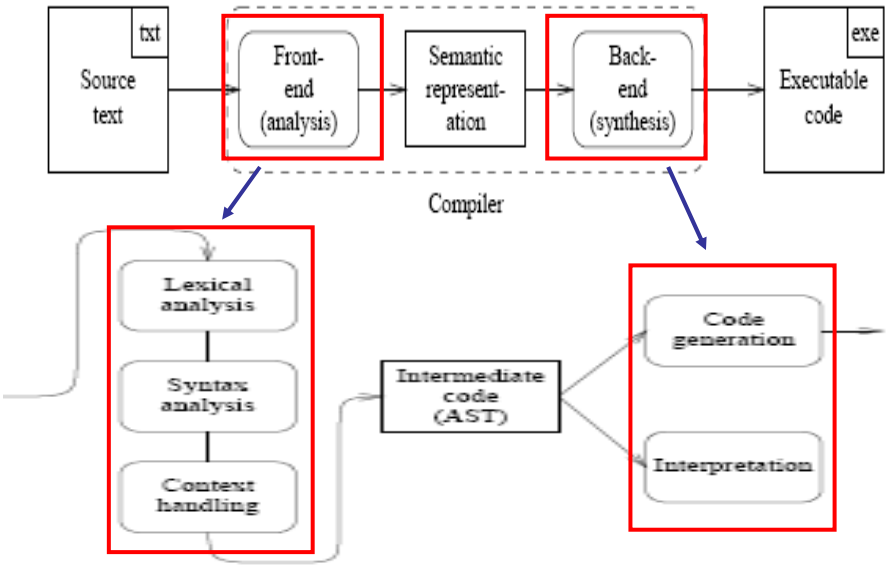
Parse tree of an expression b*b-4*a*c



Annotated AST of an expression b*b-4*a*c



Structure of the compiler



Syntax of PL ➔ CFG

- Context-free syntax defines legal sequences of symbols independent of any notion of what the symbols mean
- Context-free cannot describe type compatibility (type checking) and scoping rules (block structure)➔ (CFGs limitation)

  ➔ context-sensitive

  a=b+c is illegal if b or c is undeclared

28

Another semantic of PL - Context-sensitive

- Context-sensitive restriction can be handled by semantic process which can be divided into static and run-time semantics
- Static semantics rules define that all identifiers should be declared, operators and operands should be type-compatible ➔ **attribute grammar**

  Ex. $E_1 \rightarrow E_2+T$

  Should be augmented with a type attribute for E and T, a predicate requiring type compatibility

  (E_2.type = numeric) and (T.type = numeric)

29

Attribute grammar

Attribute Grammar

Rule 1. Syntax rule: <assign> → <var> = <expr>
Semantic rules:
<expr>.expected_type ← <var>.actual_type 由LHS type決定

Rule 2. Syntax rule: <expr> → <var>[2] + <var>[3]
Semantic rules:
<expr>.actual_type ←
if (<var>[2].actual_type = int) and
(<var>[3].actual_type = int) then int else real
Predicate:
<expr>.actual_type = <expr>.expected_type

30

Attribute grammar

3. Syntax rule: <expr> → <var>
Semantic rule:
<expr>.actual_type ← <var>.actual_type
Predicate: <expr>.actual_type = <expr>.expected_type

4. Syntax rule: <var> → A | B | C
Semantic rule:
<var>.actual_type ← Look-up(<var>.string)

Look-up function looks up a given variable name in the symbol table and returns the variable's type

31

Example of attribute grammar

Ex. A=A+B <assign> → <var> = <expr>

1. <var>.actual_type ← Look-up(A) [Rule 4]
2. <expr>.expect_type ← <var>.actual_type [Rule 1]
3. <var>[2].actual_type ← Look-up(A) [Rule 4]
<var>[3].actual_type ← Look-up(B) [Rule 4]
4. <expr>.actual_type ← either int or real [Rule 2]
5. <expr>.expected_type =<expr>.actual_type is either TRUE or FALSE [Rule 2]

32

Attribute grammar

Characteristics:

Attribute grammar is used to describe the syntax and the static semantics of a PL.

Advantage:

Attribute grammar can be used as the formal definition of a language that can be input to a compiler generation System.

33

Attribute grammar

Disadvantage:

- **Hard to describe all of the syntax and static semantics of a real PL because its size and complexity**
- **The number of attributes and semantics rules make such grammars difficult to read and write**
- **The attribute values on a large parse tree are expensive**

34

Runtime semantic

- Axiomatic semantics based on specified relation or predicates that relate program variables

35

Runtime semantic

Ex. The axiom definition var = exp usually states that a predicate involving **var** is true after statement execution iff the predicate obtained by replacing all occurrences of var by **exp** is true

➔ for y > 3 to be true after executing the statement y=x+1, the predicate x+1>3 should have to be true before the statement is executed

- Aliasing makes axiomatic definitions much more complex

36

Axiomatic semantics

- Advantage
  - Good for deriving proofs of program correctness
  - Concentrate on how relations among variables are changed by statement execution
- Disadvantage
  - Difficult to be used to completely define most PL
  - Cannot model implementation consideration such as run out of memory

37

Denotational model

- Rely on notation from mathematics and often fairly compact
- A syntax-directed definition in term of the meaning of its immediate constituents

Ex. Define **integer addition (cannot overflow)**

$E[T_1+T_2] = E[T_1]$ is integer and $E[T_2]$ is integer

➔range($E[T_1]+E[T_2]$) else error

The first Ada compiler basis

38