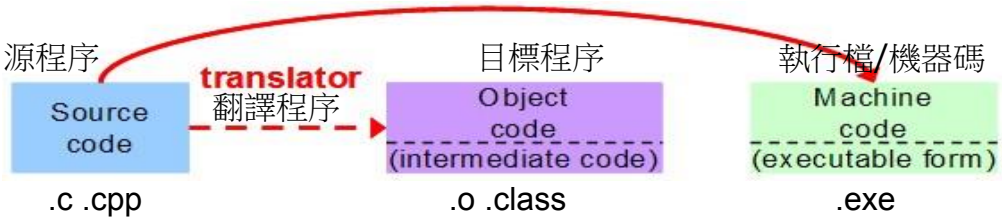
Translators 翻譯程序 (Assembler 匯編, Compiler 編譯, Interpreter 解釋程序)

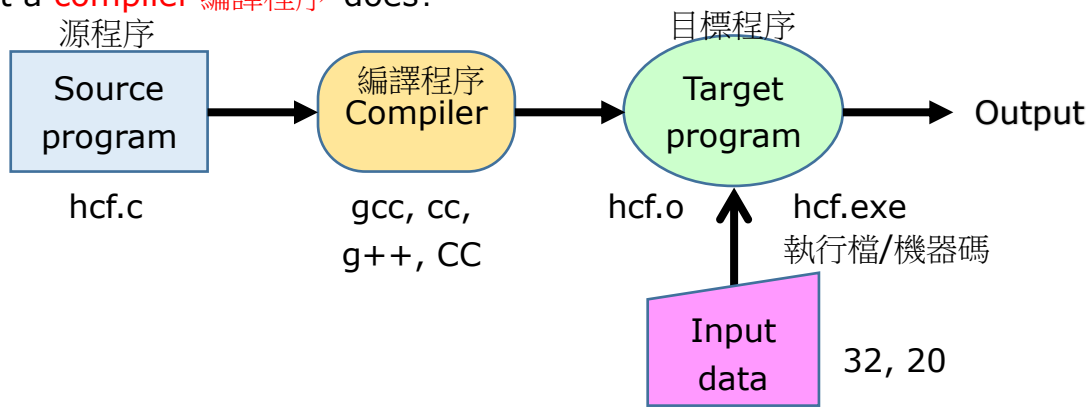
Why translator is needed?

Source code 源程序, intermediate code 目標程序, executable code 執行檔/機器碼

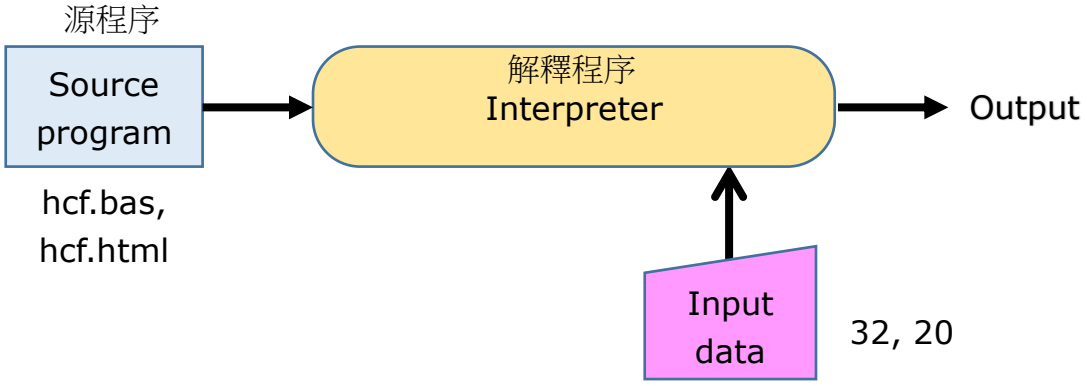


翻譯程序 translator

(1)What a compiler 編譯程序 does?



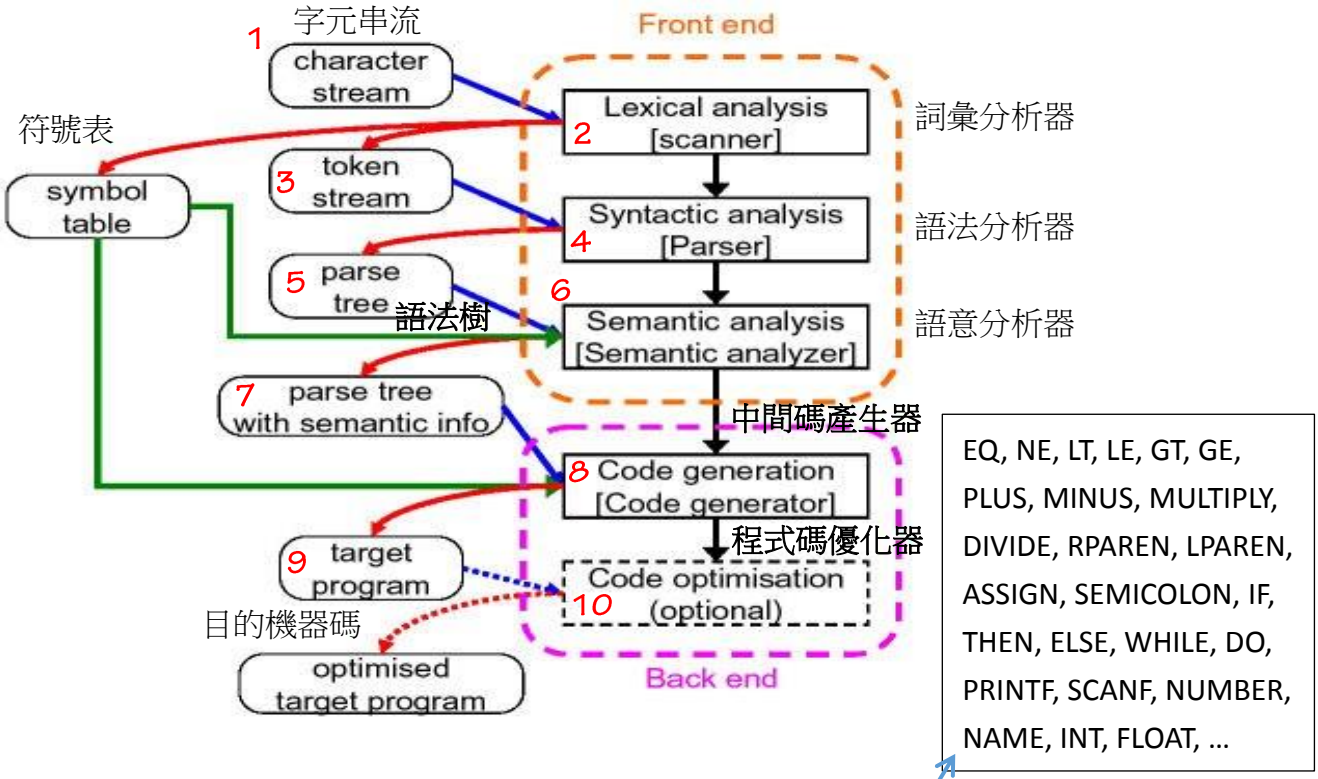
(2)What an interpreter 解釋(直譯)程序 does?



(3)Compare compilers 編譯程序 and interpreters 解釋程序

Phases 階段 of compilation 編譯過程

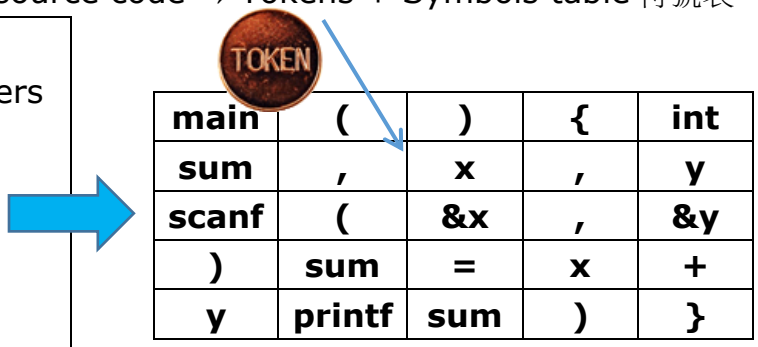
HKDSE sample paper 2D Q4



Lexical analysis 字詞/詞彙分析

- What does a lexical analyzer do?
- Character stream 串流(FILE) from source code → Tokens + Symbols table 符號表

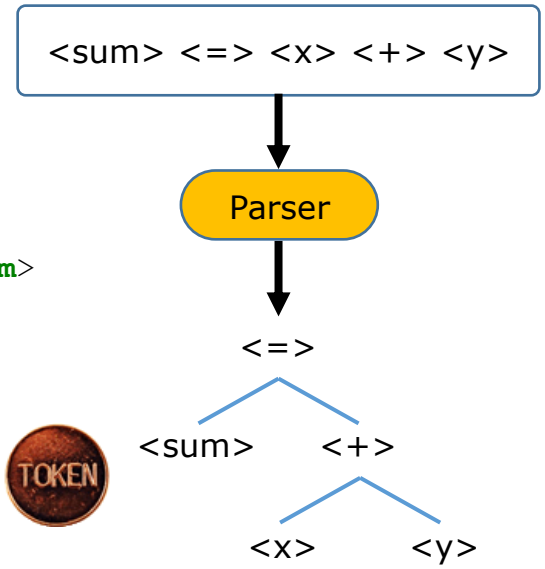
```
#include <stdio.h>
// Calculate the sum of 2 integers
main(){
    int sum,x,y;
    scanf("%i%i", &x, &y);
    sum = x+y;
    printf("%i\n", sum);
}
```

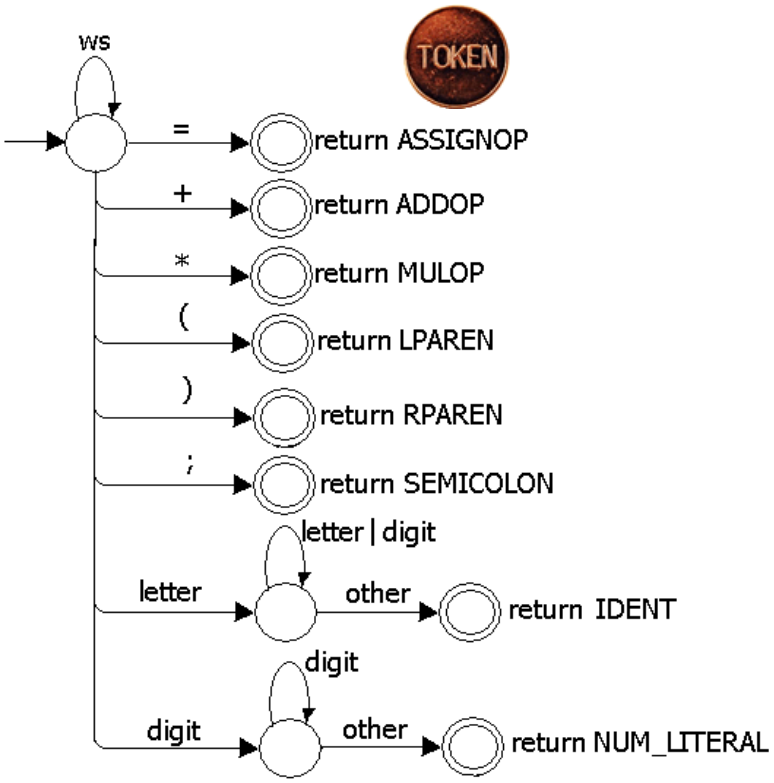


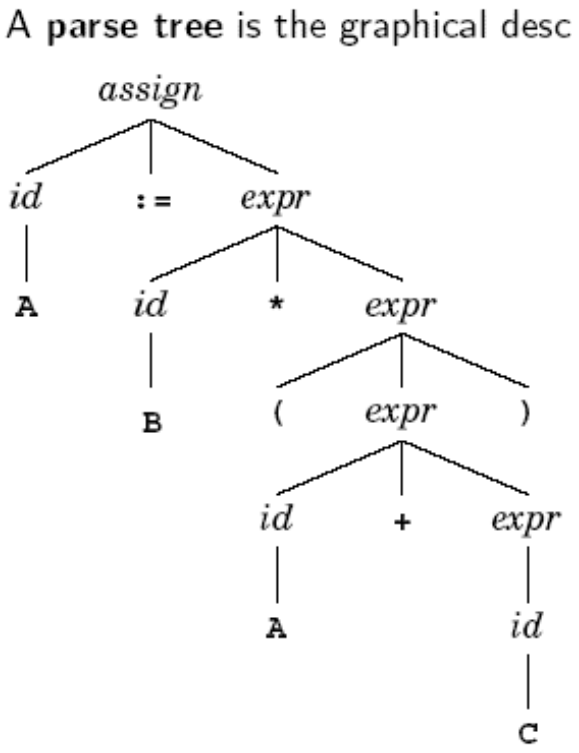
Parser 語法分析

- What is syntactic analysis?
- Tokens → Parse tree 語法樹

BNF <expr>::= <term>|<expr><addop><term>







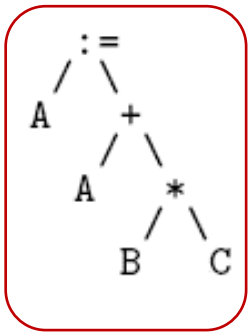
A = B * (A + C)

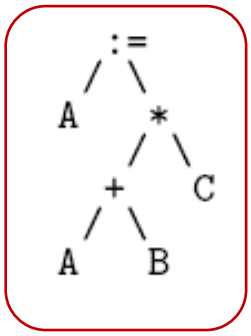
Ambiguous Grammars 語法含糊

A grammar is ambiguous if a sentence exists for which more than one parse tree can be constructed. 一個語句，有多個語法樹(意義)

assign → id:= expr
id → A | B | C
expr → expr + expr | expr * expr | (expr) | id

Consider derivation of A := A + B * C





語法 Syntax	the program's form 形式
語意 Semantics	the program's meaning 意義

1. Static Semantics

- Semantics that can be determined at compile time. 編譯時
- Usually related to type constraints e.g.
  - Declare variables before use
  - Type compatibility
  - Re-declaration illegal

2. Dynamic Semantics

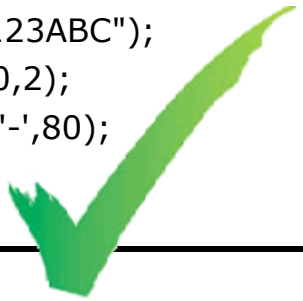
- What happens at run time 執行時
- e.g. what does `while (…) do { … }` mean?

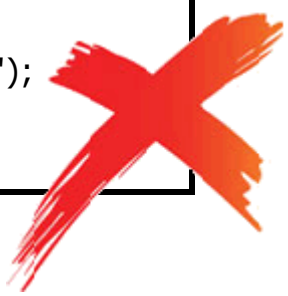
The problem

- A syntactically correct program may contain a semantic error. e.g.
  - A variable is not defined before use
  - A function is called with incorrect number of arguments.

語法 syntax 正確，不一定語意 semantic 正確

變量未宣告。	
調用 call 子程式時，參數數目不符。 itoa(255,s,16);	itoa(10,s);
變數類型不符。 n=atoi("123ABC");	int n = "abc"; n=atoi('1');
fseek(fp,0,2);	fseek(0,fp,2);
drawline('-',80);	drawline(80,'-');
重複宣告。	int n; float n;





Semantic analyzer 語意分析

•What a semantic analyzer does?

•Parse tree + Symbols → Parse tree with semantic information

•Front end

Java source code compiled to PC/mobile phone/PDA executable code

Back end of a compiler

•Code generation + Code optimization

•Intermediate code (e.g. Java bytecode)

•Java Virtual Machine (JVM)

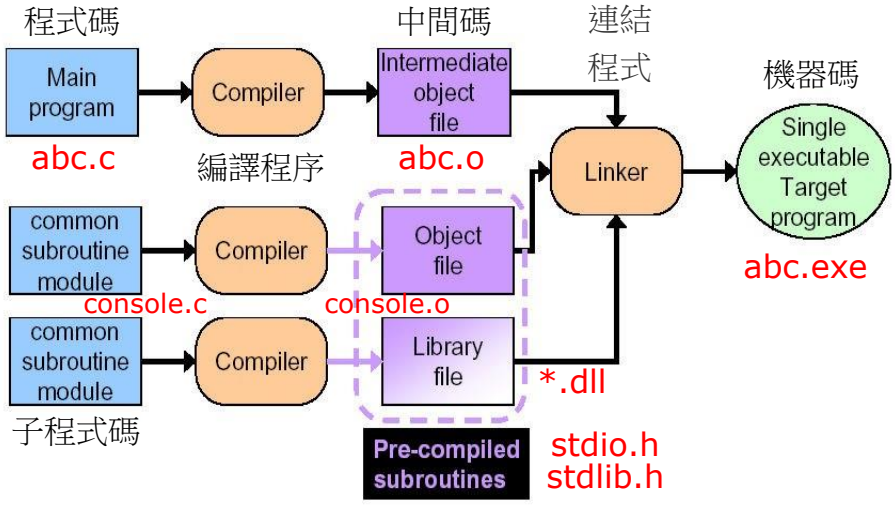
Compile once only to an intermediate object code

Bytecode runs directly on JVM of PC/mobile phone/PDA.

Linker 連結程式 and loader 載入程式

•Functions of a linker and a loader

•Dynamic linking (DLL)



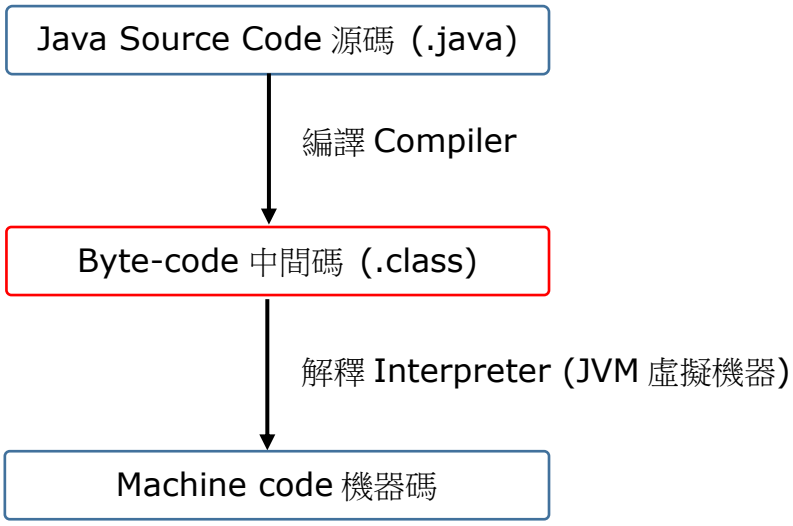
Assessment Task B #2

1(a) The source code (源程式碼) of a program written in a high level language must be translated to an object code (目的程序) before the program can be executed on a computer. What is meant by **source-code** and **object-code**?

1(b) What are the purposes of a compiler?

1(c) Suggest any TWO reasons why almost all programs are usually compiled before they are sold to customers?

Different Abstraction Levels of Programming…



跨平台

Java Compiler + Interpreter (JVM) = Portability (or Platform Independence) !!

- Code generation + Code optimization done by the Java compiler
- [javac.exe in the JDK]
- Intermediate code (e.g. Java bytecode) to be interpreted and
- executed by the Java Virtual Machine [java.exe]

Demo. of Compilation + Interpretation…(Using Commands)

Java-Compiler (javac)	**Java-Interpreter (java virtual machine)**
C:\> javac Main.java → Main.class (byte-code)	C:\>java Main.class Hello World!! (output)

http://www.c4learn.com/c-programming/lexical-analysis-phases/

Lexical Analysis – Compiler

Table of content

1 Compiler:
2 Different phases of compilers:
3 1.Analysis Phase:
4 Lexical Analysis Phase:

Compiler:

Compiler takes high level human readable program as input and convert it into the lower level code. This conversion takes place using different phases. First phase of compiler is lexical analysis.

Must Read: [What is Compiler ?]

Different phases of compilers:

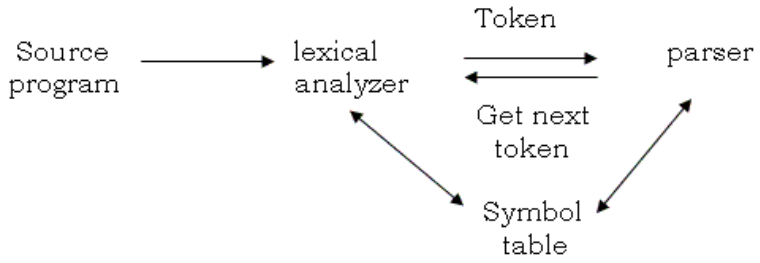
1. Analysis Phase
2. Synthesis Phase

1.Analysis Phase:

- Lexical analysis
- Syntax analysis
- Semantic analysis

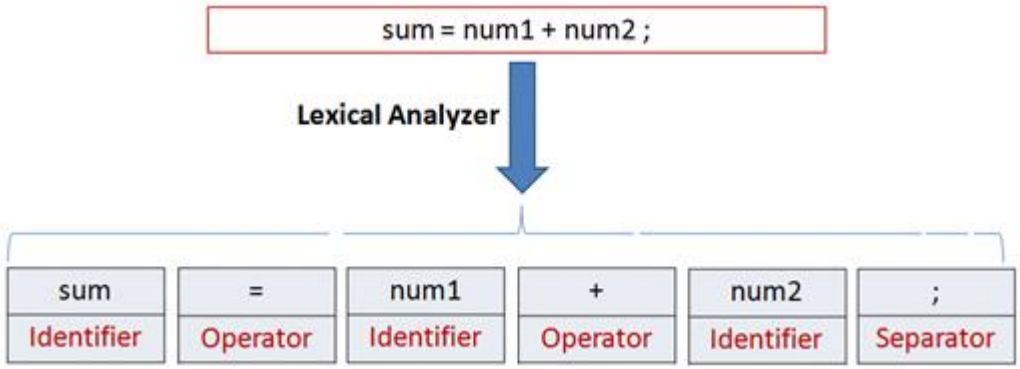
Lexical Analysis Phase:

Task of Lexical Analysis is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.



1. Lexical Analyzer is First Phase Of Compiler.
2. Input to Lexical Analyzer is "**Source Code**"
3. Lexical Analysis Identifies Different Lexical Units in a **Source Code**.
4. Different **Lexical Classes or Tokens or Lexemes**
   - Identifiers 識別字
   - Constants 常數
   - Keywords 關鍵字
   - Operators 運算子

5. Example: sum = num1 + num2 ;



So Lexical Analyzer will produce the following **Symbol Table** –

Tokens	Type
sum num1 num2	Identifier 識別字
= +	Operator 操作/運算子
;	Separator 分隔符

6. Lexical Analyzer is also called "**Linear Phase**" or "**Linear Analysis**" or "**Scanning**"
7. Individual Token is also Called **Lexeme**
8. Lexical Analyzer's Output is given to **Syntax Analysis**.

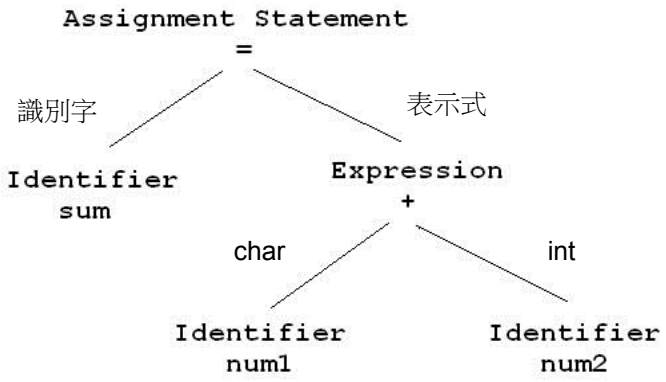
Syntax Analysis – Compiler

1 Analysis Phase: 2nd Phase of Compiler (Syntax Analysis)
2 Syntax Analysis:
3 Parse Tree Generation:
4 Explanation: Syntax Analysis

Parse Tree Generation:

```
sum = num1 + num2
```

Now Consider above C Programming statement. In this statement we Syntax Analyzer will create a parse tree from the tokens.



Syntax Analyzer will check only Syntax not the meaning of Statement

Explanation: Syntax Analysis

- We know , Addition operator plus ('+') operates on two Operands
- Syntax analyzer will just check whether plus operator has two operands or not . It does not checks the type of operands.
- Suppose 1 of the Operand is String and other is Integer then it does not throw error as it only checks whether there are 2 operands associated with '+' or not.
- So this Phase is also called Hierarchical Analysis as it generates **Parse Tree Representation** of the Tokens generated by Lexical Analyzer

Semantic Analysis – Compiler

Syntax analyzer will just create parse tree. Semantic Analyzer will **check actual meaning** of the statement parsed in parse tree. Semantic analysis can compare information in one part of a parse tree to that in another part (e.g., compare reference to variable agrees with its **declaration**, or that **parameters** to a function call match the function definition).

Semantic Analysis is used for the following –

1. **Maintaining the Symbol Table** for each block.
2. Check **Source Program for Semantic Errors**.
3. **Collect Type Information** for Code Generation.
4. **Reporting compile-time errors** in the code
   (except syntactic errors, which are caught by syntactic analysis)
5. **Generating the object code** (e.g., assembler or intermediate code)

Now In the Semantic Analysis Compiler Will Check –

1. Data Type of First Operand
2. Data Type of Second Operand
3. Check Whether + is Binary or Unary.
4. Check for Number of Operands Supplied to Operator Depending on Type of Operator (Unary | Binary | Ternary)

What is Interpreter ?

1 What is Interpreter ?
2 Examples of Programming Languages Using Interpreter:
2.1 Lisp
2.2 BASIC

What is Interpreter ?

1. Interpreter Takes **Single instruction** as input .
2. No Intermediate **Object Code is Generated**
3. Conditional Control Statements are Executes **slower**
4. **Memory Requirement** is **Less**
5. Every time higher level program is converted into lower level program
6. **Errors are displayed for every instruction** interpreted (if any)
7. The interpreter can **immediately execute high-level programs**, thus interpreters are sometimes used during the development of a program, when a programmer wants to add small sections at a time and test them quickly.
8. In addition, interpreters are **often used in education** because they **allow students to program interactively**.

Examples of Programming Languages Using Interpreter:

Lisp

```
(defun convert ()
  (format t "Enter Fahrenheit ")
    (LET (fahr)
      (SETQ fahr (read fahr))
      (APPEND '(celsisus is) (*(- fahr 32)(/ 5 9)) )
    )
)
```

BASIC

```
CLS
INPUT "Enter your name: ", Name$
IF Name$="Mike" THEN
    PRINT "Go Away!"
ELSE
    PRINT "Hello, "; Name$; ".  How are you today?"
END IF
```

Difference between Compiler 編譯程序 and Interpreter 解釋程序

No	Compiler	Interpreter
1	Takes **Entire** program as input	Takes **Single** instruction as input.
2	Intermediate Object Code is **Generated**	**No** Intermediate Object Code is **Generated**
3	Conditional Control Statements are Executes **faster**	Conditional Control Statements are Executes **slower**
4	**Memory Requirement**: **More** (Since Object Code is Generated)	**Memory Requirement** is **Less**
5	Program need **NOT** be **compiled** every time	Every time higher level program is converted into lower level program
6	**Errors** are displayed after **entire program** is checked	**Errors** are displayed for **every instruction** interpreted (if any)
7	**Example**: C Compiler	**Example**: BASIC

Explanation: Compiler Vs Interpreter

Just understand the concept of the compiler and interpreter –

1. We give complete program as input to the compiler. Our program is in the human readable format.
2. Human readable format undergoes many passes and phases of compiler and finally it is converted into the machine readable format.
3. However interpreter takes single line of code as input at a time and execute that line. It will terminate the execution of the code as soon as it finds the error.
4. Memory requirement is less in Case of interpreter because no object code is created in case of interpreter.

https://youtu.be/o-xSgi9o5vk

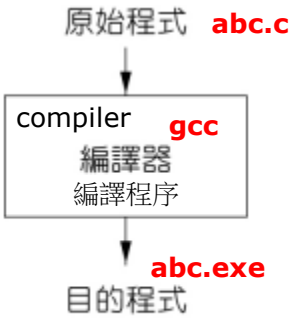


圖 1.1 編譯器

圖 1.2 執行目的程式

圖 1.3 直譯器

Source Code 源碼 is:

1. in form of Text.文本
2. Human Readable. 人類可讀
3. generated by Human. 由人編寫
4. input to Compiler. 編譯器的輸入

Object Code 目標程序 is:

1. in form of Binary.二進制碼
2. Machine Readable. 只有電腦可讀
3. generated by Compiler. 由編譯器產生
4. output of Compiler. 編譯器的輸出

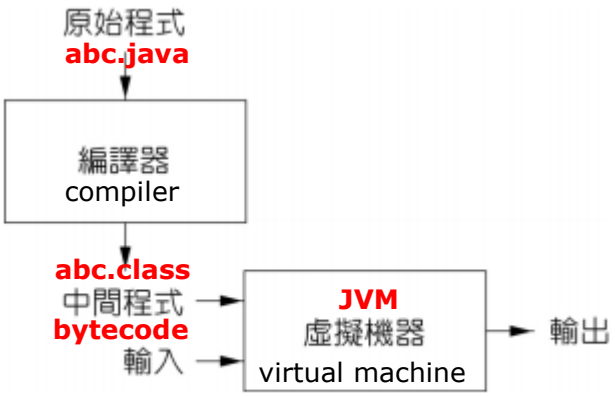


圖 1.4 混合編譯器

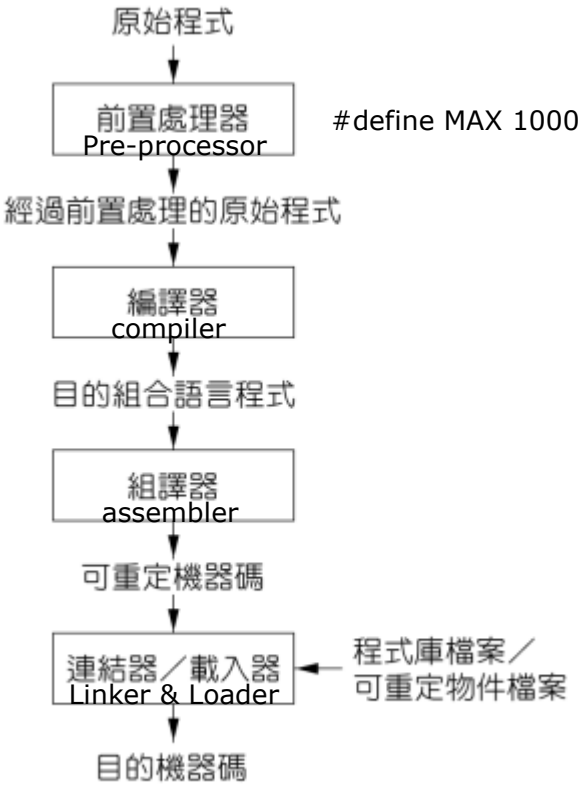


圖 1.5 語言處理系統

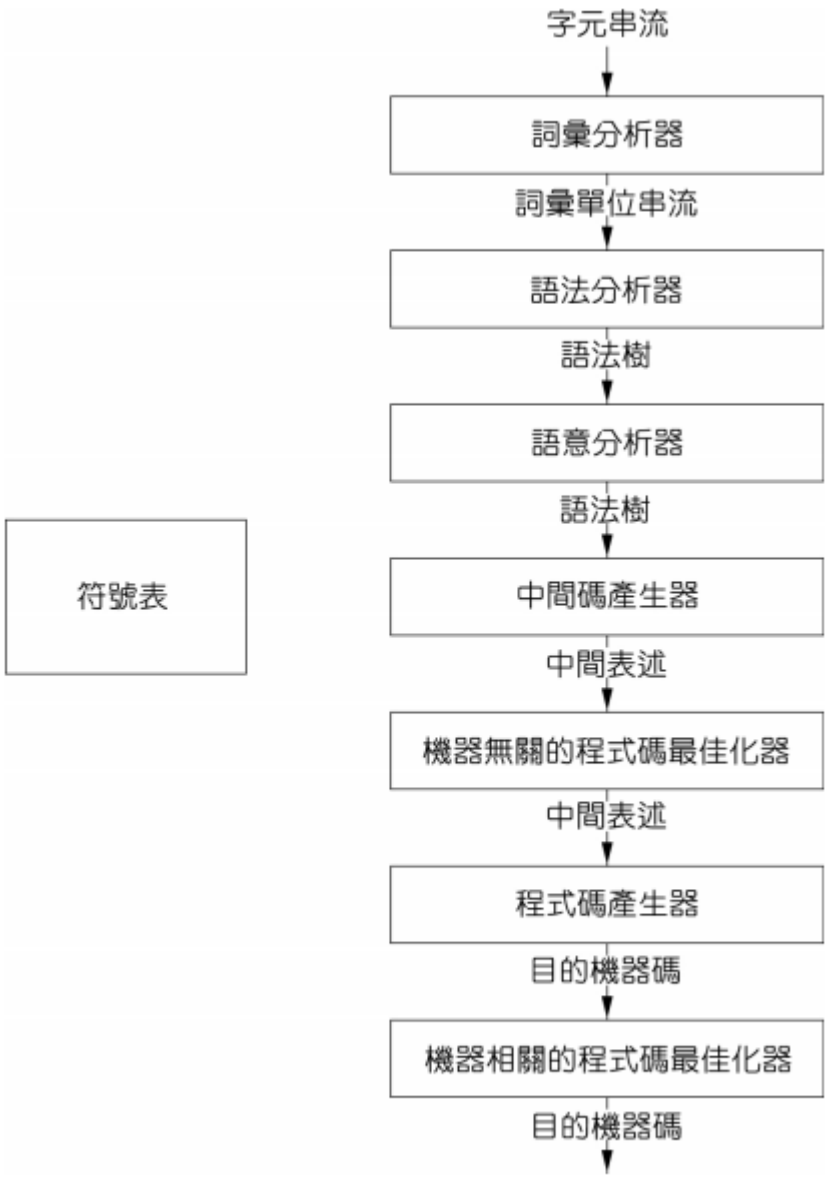
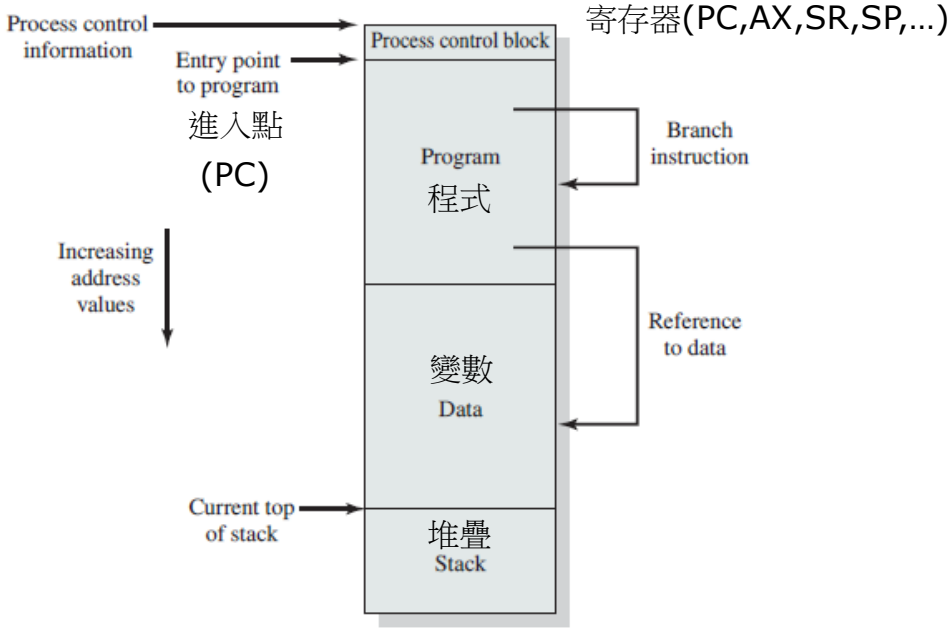


圖 1.6　編譯器的各個步驟

http://www.prudentman.idv.tw/2008/05/compile.html



編譯 Compile 流程

一個完整的編譯過程通常需要包含以下的 4 個程序：

1.Preprocess 預處理器。
2.Compiler 編譯器。會處理副檔名.c 的檔案，產生副檔名.as 的組譯檔。
3.Assembler 組譯器。會處理副檔名.as 的組譯檔，產生副檔名.o 的中間檔。
4.Linker 連結器。會處理副檔名.o 或.obj 的中間檔。

預處理器（Preprocess）

預處理器會在進行編譯前先處理原始碼內像#ifdef、#define 相對簡單的詞句替換、和一些巨集代換的功能。

編譯器（Compiler）

編譯器是將高階語言所寫的原始程式，翻譯成機器語言組成的目的程式。在編譯過程中，會執行下列的步驟：

- **語彙分析（Lexical analysis）**。分析程式中每一個字詞/權標（token）：註解（comment, 在編譯過程會被忽略）、關鍵字（keyword, 如 int、for、while 等）、常數（constant, 如 1、12、"embedded"等）、運算子（operator, 如+、-、*、/等）。
- **語法分析（Syntax analysis）**。主要是將程式符號，轉換成階層式的語法樹（Syntax tree），檢查程式的語法結構是否正確。在語法樹中，正常情況下，階層最高的節點（node）為 assign 的符號，其餘的節點為其他的運算元符號，而葉子（leaf）就都是變數的標記（token）。
- **語意分析（Semantic Analysis）**。是由語法樹（Syntax tree）來分析程式的邏輯與語法是否符合規定。分析程式的「文法」是否正確，已經從文字符號的階段進入了程式語意的判別。例如：變量未宣告。調用 call 子程式時，參數數目不符。變數類型不符。**中間碼的產生（Intermediate code generation）**。是從語法樹（Syntax tree）中，以一個節點（node）為基本單位，從最底層的節點依序往上，拆解成一個個最基本的運算式，而每一個節點也會賦予一個暫時性的符號。
- **程式碼的最佳化（code optimization）**。基本上就是減少一些不必要的暫時性節點符號。當然，另外還有一些特別的最佳化演算法也會在這個階段使用，例如針對迴圈邏輯的最佳化有三種知名的演算法： code motion、induction variable、strength reduction；因為迴圈邏輯在語法上是最沒有執行效率的語法之一，因此需要特別的最佳化。或者，有時候編譯器會調整程式的前後順序，為了在下一階段程式碼的產生過程中，暫存器的使用數目降低。
- **程式碼的產生（code generation）**。以 C 語言為例，這裡就是將最佳化後的中間碼，搭配微處理器的暫存器，逐一轉換成組合語言。

組譯器（Assembler）

組譯器會將組合語言的原始程式，翻譯成機器語言組成的中間檔 ".obj"、".o"。

連結器（Linker）

連結器會將中間檔 obj files 連結起來，找到 symbol（函式，變數名）與程式庫（shared obj）中的副程式，產生可執行的 exe 檔(executable)。

C compilation:

1) Lexical Analyzer:

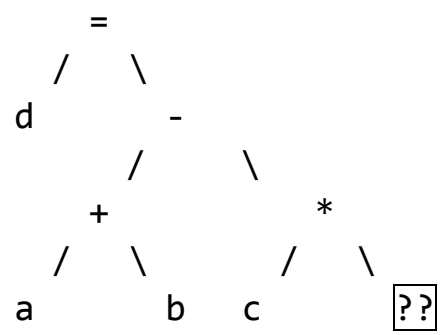
It combines characters in the source file, to form a "TOKEN". A token is a set of characters that does not have 'space', 'tab' and 'new line'. Therefore this unit of compilation is also called "TOKENIZER". It also removes the comments, generates symbol table and relocation table entries.

2) Syntactic Analyzer:

This unit check for the syntax in the code. For example:

```
int a,b,c,d;
d = a + b - c * __;
```

The above code will generate the parse error because the equation is not balanced. This unit checks this internally by generating the parser tree as follows:



Therefore this unit is also called PARSER.

3) Semantic Analyzer:

This unit checks the meaning in the statements. For example:

```
int i;
int *p;
p = i;      // p = &i;
-----
```

The above code generates the error "Assignment of incompatible type".

4) Pre-Optimization:

This unit is independent of the CPU, i.e., there are two types of optimization

1).Pre-optimization (CPU independent) 2).Post-optimization (CPU dependent)

This unit optimizes the code in following forms:

- I) Dead code elimination
- II) Sub code elimination
- III) Loop optimization

I) Dead code elimination: For example:

```
int a = 10;
if ( a > 5 ) {
      /*          ...          */
} else {
      /*     never be executed ...       */
}
```

Here, the compiler knows the value of 'a' at compile time, therefore it also knows that the if condition is always true. Hence it eliminates the else part in the code.

II) Sub code elimination: For example:

```
int a, b, c, x, y;
x = a + b;
y = a + b + c;
```

can be optimized as follows:

```
y = x + c;          // a + b is replaced by x
```

III) Loop optimization: For example:

```
int a;
for (i = 0; i < 1000; i++ ) {
        ...
        a = 10;
        ...
}
```

In the above code, if 'a' is local and not used in the loop, then it can be optimized as follows:

```
a = 10;
for (i = 0; i < 1000; i++ ) {
        /*              ...             */
}
```

5) Code generation:

Here, the compiler generates the assembly code so that the more frequently used variables are stored in the registers.

6) Post-Optimization:

Here the optimization is CPU dependent. Suppose if there are more than one jumps in the code then they are converted to one as:

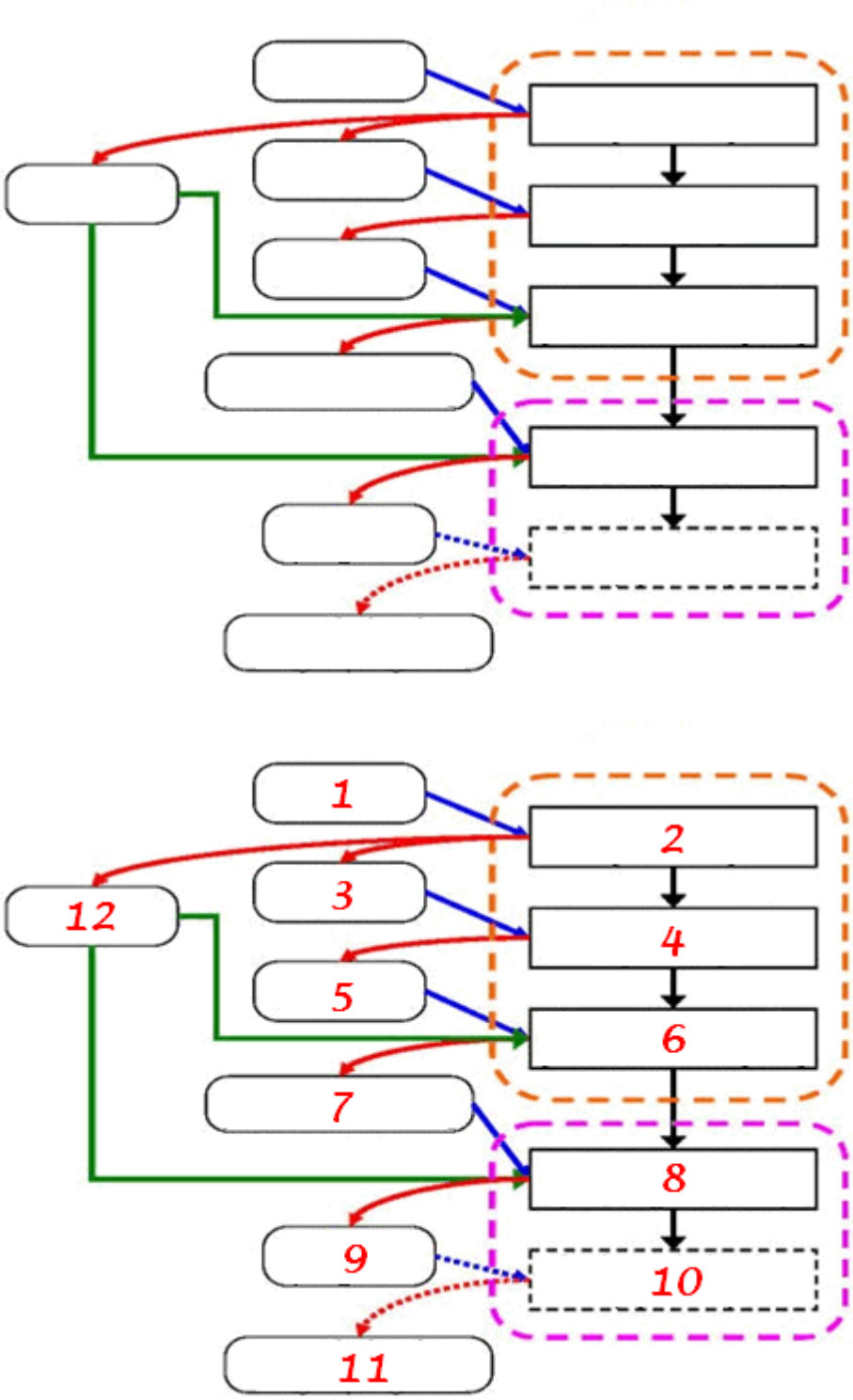
```
        -----
jmp:<addr1><addr1> jmp:<addr2>
        -----
```

The control jumps to the `<addr2>` directly.

```
jmp:<addr2>
```

Then the last phase is Linking (which creates executable or library).

When the executable is run, the libraries (stdio.h, *.dll) it requires are Loaded.



上圖為程式的編譯過程：請為以下空格配對適當的工作或物件。[8 分]

	中間碼產生器		語法分析器	3	tokens
	詞彙分析器		目的機器碼		語意分析器
5	語法樹 1		字元串流(源程式)		符號表
11	已優化的機器碼	7	語法樹 2		程式碼優化器

並簡述以下階段的作用? (i) 語法分析、(ii) 語意分析、(iii) 詞彙分析 [6 分]