轉譯程式

轉譯程式爲一種系統軟體，其功能是將輸入的原始程式轉換成另一種相對應的程式語言

轉譯程式大致上可包含下列四種：

(1) 組合程式 (Assembler)

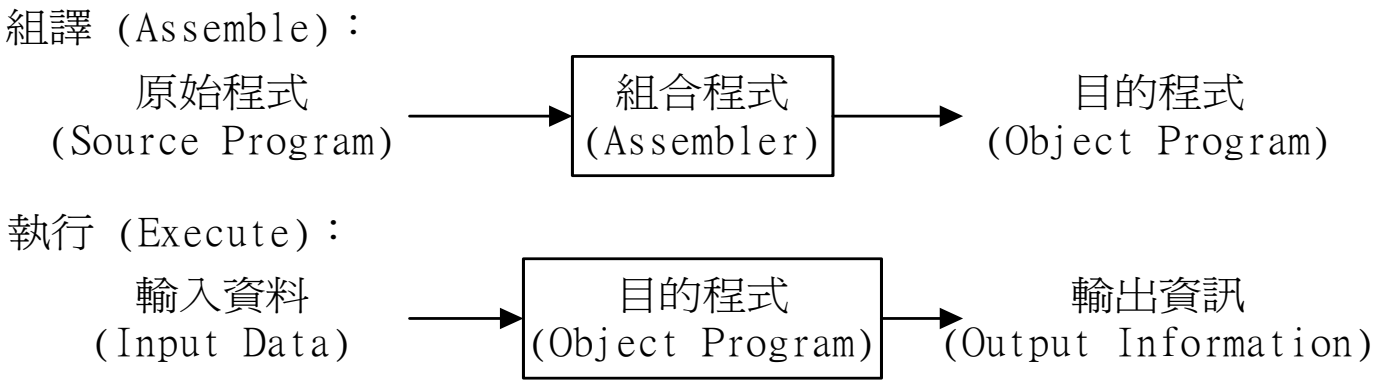
(2) 編譯程式 (Compiler)

(3) 前置處理程式 (Preprocessor)

(4) 直譯程式 (Interpreter)

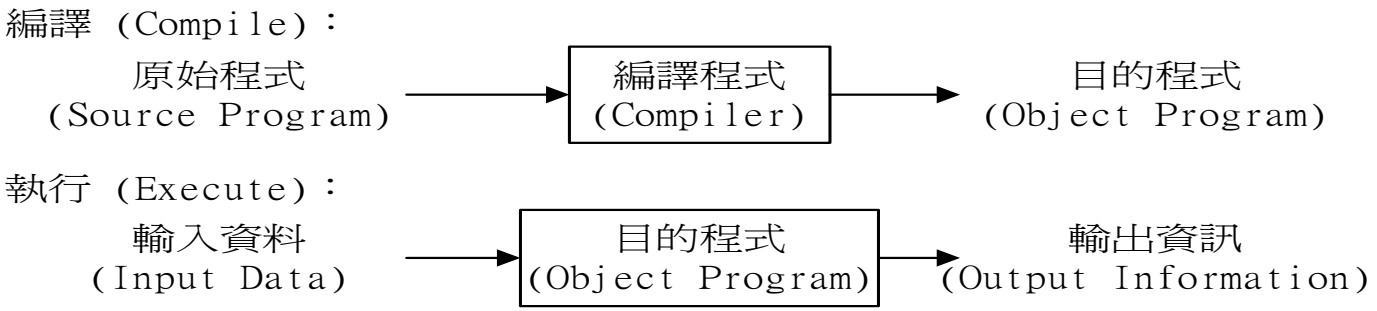
組合程式

- 組合程式是一種系統軟體，其功能是將用組合語言所撰寫的原始程式翻譯成相對應的目的碼



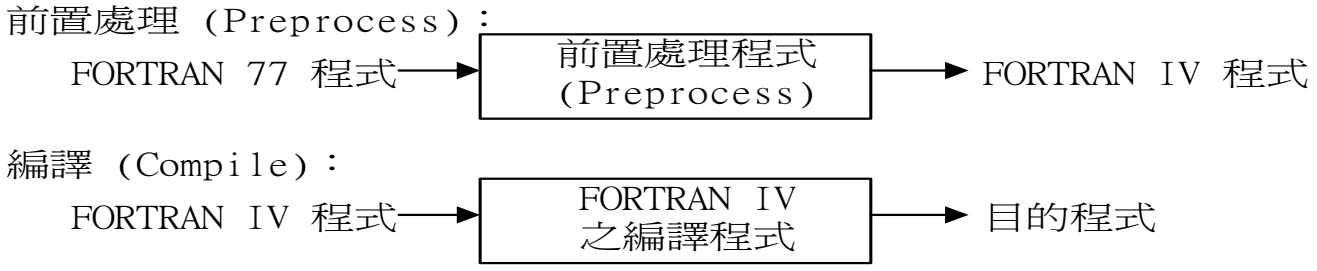
編譯程式

- 編譯程式爲一種系統軟體，其功能是將以高階語言所撰寫的原始程式翻譯成機器導向語言 (如組合語言，機器語言)



前置處理程式

- 可將某一種高階語言所撰寫之程式先轉譯成另一種高階語言程式，然後再予以編譯，此種負責高階語言間轉譯工作的程式稱之爲前置處理程式

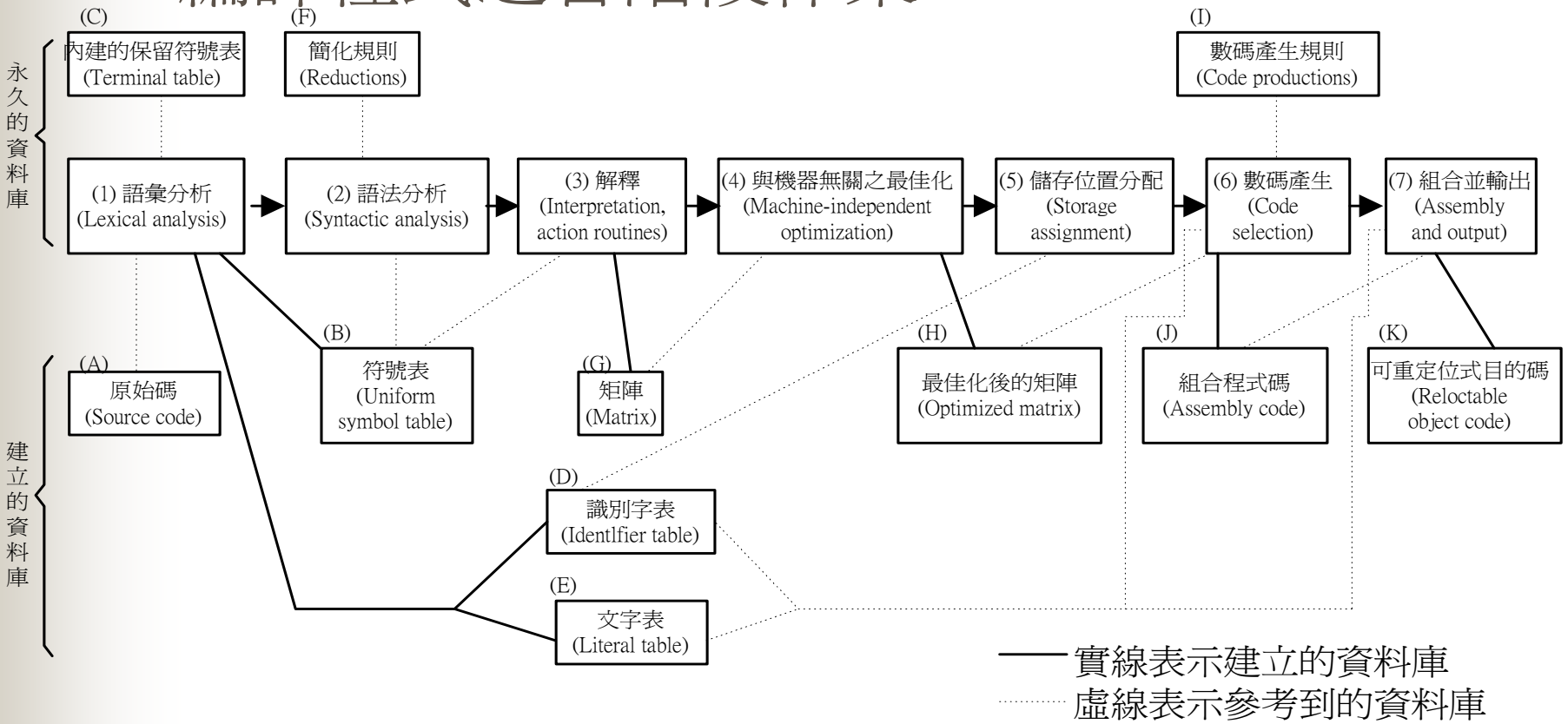


直譯程式

- 直譯程式功能與編譯程式相似，但並不產生目的碼。直譯程式與編譯程式這二種語言處理程式的最大差異在於直譯程式直接執行原始程式，而編譯程式則將原始程式轉換成另一種較低階的語言

編譯程式

編譯程式之各階段作業



- (1) 語彙分析階段 (Lexical Analysis Phase) —
- 其主要的工作是以一個字元一個字元的方式將原始程式讀入，把敘述句分解成單語 (Token)，如：變數 (Variable)、常數 (Constant)、區分符號 (Delimiter)、關鍵字 (Keyword)、運算元 (Operator) 等基本單元。並建立文字表 (Literal Table)、識別字表 (Identifier Table) 與符號表 (Symbol Table)。負責此一工作者稱爲敘述句分解程式 (Lexical Analyzer 或 Scanner)。

- (2) 語法分析階段 (Syntactic Analysis Phase) —
- 其工作爲將單語組合成語法架構 (Syntactic Structure) 如將A、+、B三個單語組合成A+B。如將程式中的宣告敘述、指定敘述依文法規則進行剖析的工作。此過程亦稱爲構文剖析 (Parsing)，而負責執行此項工作者稱之爲構文剖析程式 (Syntactic Analyzer 或稱 Parser)。

- (3) 解釋階段 (Interpretation Phase) —
- 本階段主要是由動作常式 (Active Routine) 所組成，當語法分析階段已辨認出語句的結構時，便呼叫相對應之動作常式。這些動作常式的功能乃是將原始程式轉換成中間形式碼，並且在識別字表中加入必要的資訊。一般而言，本階段可合併於構文剖析程式 (Parser) 中處理。

- (4) 與機器無關之最佳化階段 (Machine Independent Optimization Phase) —
- 將構文剖析程式輸出的矩陣或構文樹進行最佳化的工作，輸出最佳化的矩陣或已簡化的構文樹 (Reduced Syntax Tree)，以節省儲存的空間與執行的時間。

與機器無關之最佳化處理之四種技巧：

- a. 刪除共同的副式子 (Elimination of Common Subexpression)：

如：
$$\begin{cases} X := \omega + (a*b); \\ Y := Z + (a*b); \end{cases} \quad \text{改成} \Rightarrow \quad \begin{cases} Temp := (a*b); \\ X := \omega + Temp; \\ Y := Z + Temp; \end{cases}$$

- b. 編譯時做必要之計算 (Compile Time Computation)：

如：
$$A = (2*3) + B; \quad \overset{\text{改成}}{\Rightarrow} \quad A = 6 + B;$$

- c. 布林表示式最佳化 (Boolean Expression Optimization)：

如：

If C_1 or C_2 ThenS_1 改成 ⇒

If C_1 thenS_1

elseIf C_2 thenS_1

- d. 將不變的計算移至迴路外面 (Loop Optimization)：

如

```
Bound : = 10 ;
While (i ≤ Bound – 2) do
```

改成 ⇒

```
Bound : = 10;
t : = Bound - 2;
While (i ≤ t) do
```

```
While (i ≤ 10) do
  Begin
    X:= 1;
    Y:= X + Z;
      ⋮
  End;
```

⇒

```
  X:= 1;
While (i ≤ 10) do
  Begin
    Y:= X + Z;
      ⋮
  End;
```

- (5) 儲存位置分配階段 (Storage Assignment Phase) —

  事先預留記憶體空間，以便儲存產生的目的碼。一般可將此階段併入數碼產生 (Code Generation) 階段。本階段主要目的爲

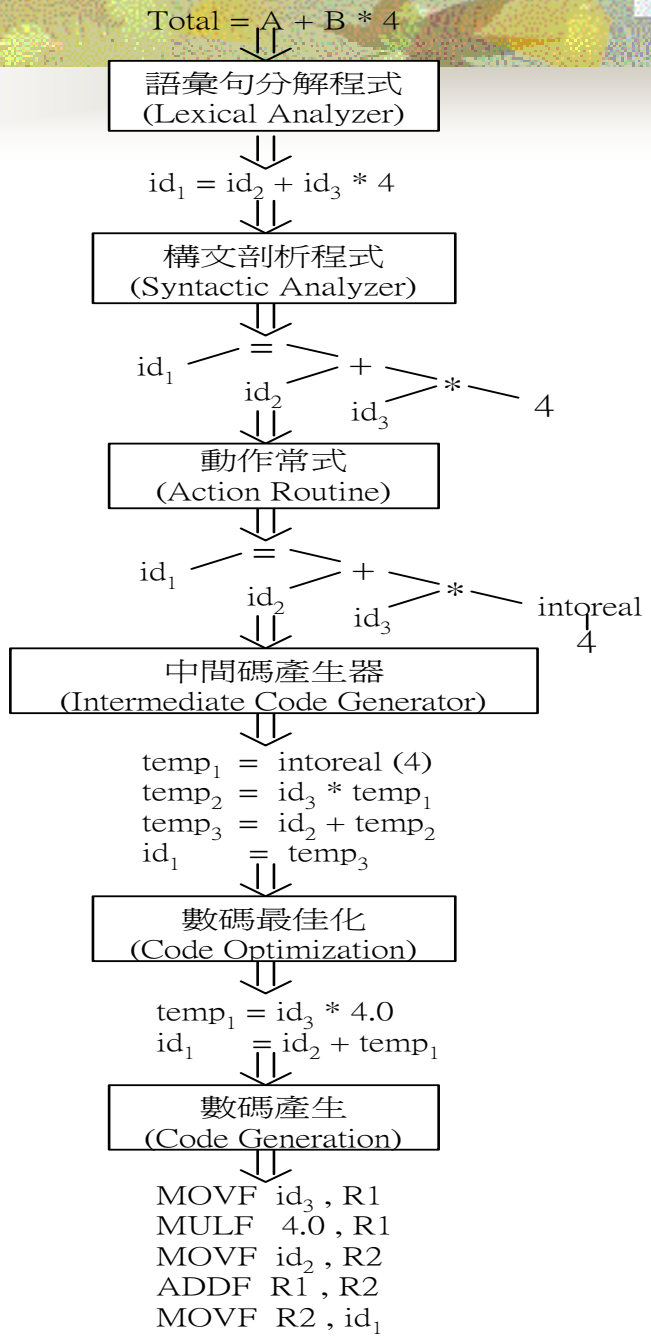
- a. 指定位置給予程式中使用到的變數。
- b. 預留位置以便儲存某些運算的中間結果。
- c. 設定位置給程式中所有的文字 (Litertal)。
- d. 給定起始值。

- (6) 數碼產生階段 (Code Generation Phase) —
  本階段主要是產生目的碼 (組合語言或機器語言)，並進行與機器有關之最佳化。其方法一般可有下列三種：
- a. 刪除多餘的儲存 (Store) 與載入 (Load) 指令。
- b. 儘量利用未被使用的暫存器 (Register)。
- c. 以執行速度較快之指令取代執行速度較慢之指令。 (如：以暫存器存取之指令取代記憶體存取之指令)。

- (7) 組合並輸出階段 (Assembly and Output Phase) —

  解決目的碼之間的位址變數，並輸出可重定位之目的碼 (Rrelocatable Object Code)。

❑ 範 例 ：


Total = A + B * 4
語彙句分解程式
(Lexical Analyzer)
id1 = id2 + id3 * 4
構文剖析程式
(Syntactic Analyzer)
id1 = + id2 * id3 4
動作常式
(Action Routine)
id1 = + id2 * id3 intoreal 4
中間碼產生器
(Intermediate Code Generator)
temp1 = intoreal (4)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
數碼最佳化
(Code Optimization)
temp1 = id3 * 4.0
id1 = id2 + temp1
數碼產生
(Code Generation)
MOVF id3 , R1
MULF 4.0 , R1
MOVF id2 , R2
ADDF R1 , R2
MOVF R2 , id1

不明確文法

一個文法對於某一句子會產生一個以上的剖析樹，則稱爲不明確文法

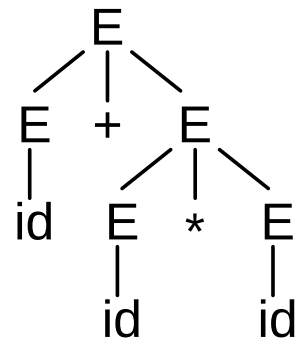
- 範例：有一文法 G 如下：

  E→E+E | E*E | (E) | -E | id

  則 id+id*id 有下列兩種最左邊的推演法 (Leftmost Derivation)

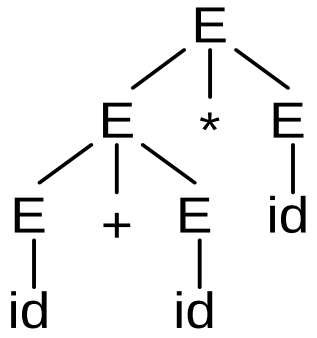
$\Rightarrow$E + E
$\Rightarrow$id + E
$\Rightarrow$id + E * E
$\Rightarrow$id + id * E
$\Rightarrow$id + id * id

剖析樹：



E $\Rightarrow$E * E
$\Rightarrow$E + E * E
$\Rightarrow$id + E * E
$\Rightarrow$id + id * E
$\Rightarrow$id + id * id

剖析樹：



不明確文法之解決方法 —

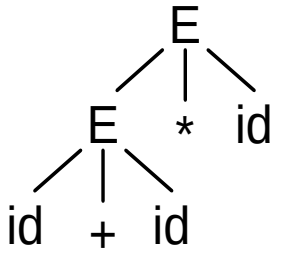
對於不明確文法之解決方式一般可分成兩種：

(1) 指明運算子 (Operator) 是採用左邊結合或右邊結合的方式。

- ❑ 範例：有一文法 G, E→E+E | E*E | id

  則 id+id*id 會產生兩種剖析樹。

  若對運算子採用左邊結合，則上述 id+id*id 中的 id+id 會先運算，運算之後的結果再乘 id。所以，只會產生一種剖析樹。即



(2) 定義運算子的順位

- ❑ 範例：(同上)

  若規定運算子之優先順序依次爲 * 與 +，則上述 id+id*id 中的 id*id 會先運算，運算之後的結果再與 id 相加。如此也只會產生一種剖析樹。即