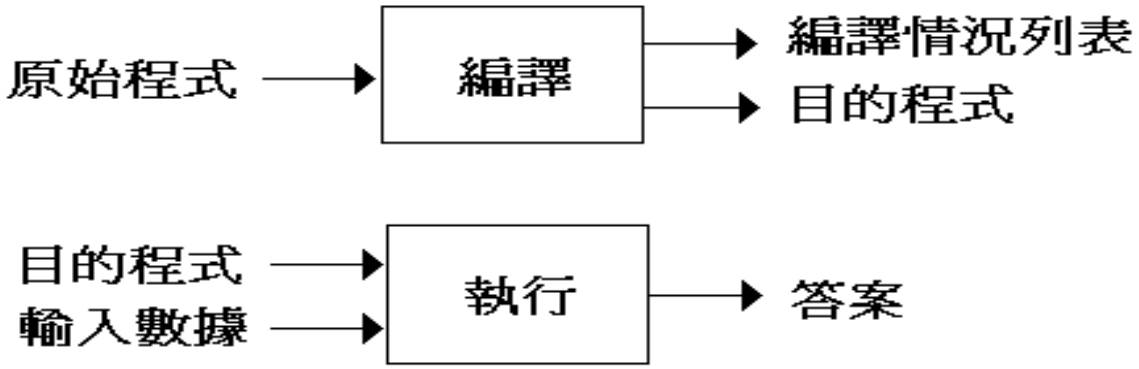
第四章 編譯器設計



- 編譯器（compiler）也稱為編譯程式，顧名思義它本身是一個程式，輸入原始程式（source program），經過編譯，輸出一個目的程式（object program），並將編譯的情況列表。然後執行目的程式，而得到答案。

編譯器設計

- 通常原始程式指使用高階語言撰寫之程式，而目的程式常為機器導向語言，也稱為低階語言。用來編譯原始程式，並產生目的程式者稱為編譯器，也稱為編譯程式。若原始程式以 PASCAL 撰寫，就稱為 PASCAL 編譯器。若原始程式以 C 語言撰寫，就稱為 C 編譯器。
- 本章所討論的原始程式為 Plone（Pascal Language ONE），意即它是一種類似 Pascal 語言的編譯器，故稱為 Plone 編譯器，唸成 ***P、Long***，或 ***「匹龍」編譯器***。

4.1 Plone 編譯器設計目標

- **4.1.1 輸入**
- 原始程式應該完全依據 Plone 的語言規則撰寫出來的，編譯器則按照該語言的語法（syntax）及語意（semantic）而產生目的程式及編譯情況列表。事實上編譯器不能期望所有的原始程式均合乎語法，因此常要診斷出原始程式的錯誤，程式語言的語法通常採用 EBNF 表示法加以定義，所以很容易了解，至於語意則較難處理，到目前為止並沒有公認的方法。
- 本章所討論的 Plone 語言採用 EBNF 表示語法，語意則在語法分析的過程中，適時加以表示。

4.1.2 輸出

- 編譯器有兩個輸出，即目的程式及編譯情況列表。通常編譯器處理不正確的原始程式比正確的為多，故編譯情況列表以及錯誤訊息與產生目的程式同等重要。編譯情況列表包含下列三項。
- ***(a)．原始程式。***
- ***(b)．偵測到的語言錯誤位置。***
- ***(c)．錯誤訊息代號。***
- （b）和（c）只有偵測到錯誤時才顯出來。站在使用者立場，希望編譯器程式能做到下列幾點：
- ***(1)．嚴格執行每一語法之審核。***
- ***(2)．一次編譯過程中，找出所有錯誤。***
- ***(3)．由上一錯誤所引起的錯誤，要清楚標出。***

4.1.3 目標

- 像其他的程式一樣，編譯器程式在設計及安裝時要受到一些限制，這些限制事實上就是設計軟體系統的一般目標，也就是
- ***可信度***、
- ***效率***、
- ***適應性***。

可信度（reliability）

- 編譯器的可信度是由簡單而結構良好的設計來決定的。所謂簡單是人人 易懂，所謂結構良好是指每一邏輯單元，設計成一個獨立的結構，稱為 模組（module），每個模組均可分別設計與測試，最後再將有關的模組 結合起來，即可得到完整的程式。

效率（efficiency）

- 影響編譯效率的因素有很多，請考慮下列幾點：
- ***（1）．編譯速率。***
- ***（2）．編譯時所用主記憶體之多寡。***
- ***（3）．主記憶體重複使用的頻率。***
-

適應性（flexibility）

- 編譯器的適應性是一特殊的優點，可以稍為修改而適用於不同的電腦系 統，產生不同的機器碼。這種設計方式稱為無關機器設計法，也就是說 要將有關機器和無關機器的部份劃分清楚。另一種較常用的設計方式稱 為無關設備設計法，就是要將有關設備與無關設備部份分開。雖然說無 關機器、無關設備，但編譯器程式設計完成後總要安裝，安裝在那一部 機器（電腦）？該機器（電腦）又有那些設備？雖說無關，還是有關， 只是程度的問題。無關的程度高表示適應性強。

有效率的編譯器必須符合下列各點

- **嚴謹的定義。**
- **必須具有某種程度的錯誤處理能力。**
- **對於目標機器能**產生有效的機器碼。
- **可信度要高。**
- **執行、儲存、及主記憶體重新使用的效率要高。**
- **要能適應目標機器的各種輸入、輸出設備。**
- **對不同的機器能**產生不同的機器碼。

4.2 Plone 定義

- Plone 是從 PASCAL 及 PL/0 中選出的一個很小子集，並做適當的增減，儘量保持編譯器程式合理、簡單，以配合本書的架構，同時又希望能解釋所有編譯高階語言的最基本的理論和觀念。當然我們也可以選擇比 Plone 更簡單或更複雜的語言為對象，但是 Plone 可以說是經過適當折衷後所產生的一種語言，因其保持了相當的簡單性，使得解說能夠透徹清晰，同時又保持了充分的複雜度，使得 Plone 成為值得一做的計劃。

Plone 敘述

- 無論是在語言結構或在語言功能上，Plone 均是相當完備的。除了包含最基本的四則運算敘述外，採用大家都熟悉的形式來表達結構化程式設計之三大觀念：（1）.循序、（2）.選擇、（3）.重複，其相對應的程式敘述為「BEGIN ... END」，IF，WHILE。也吸取副常式（subroutine）的觀念，因此也具有程序 PROCEDURE 宣告及呼叫 CALL 的敘述。同時也納入簡單的輸入敘述 READ 及輸出敘述 WRITE 等。

Plone 資料型態

- 在資料型態（type）方面，則僅守簡單的原則，數字內定為整數，文字說明內定為字串，也就是說只有字串及整數型態可供使用，因此常數為字串型態而變數為整數型態，其運算則為簡單的算術及邏輯運算。
- 在設計 Plone 編譯器程式時，也加入功能，讓每一個程序（procedure）均可定義該程序所使用的常數、變數、程序等，該變數就稱為區域（local）變數，因為它只屬於該程序，故稱為區域，若變數屬於整個程式，則稱為整體（global）變數。有了區域變數，加上動態配置，就具備遞迴呼叫的功能了。

Plone 語法規則之一

1. <Program> ::= <ProgramHead><Block>.
2. <ProgramHead> ::= PROGRAM <Identifier>;
3. <Block> ::= [<ConstDeclaration>]
   [<VarDeclaration>]
   [<ProcDeclaration>]
   <CompoundStatement>
4. <ConstDeclaration> ::=
   CONST <Identifier>=<String>
   {,<Identifier>=<String>};
5. <VarDeclaration> ::=
   VAR <IdentifierList> ;
6. <ProcDeclaration> ::=
   {PROCEDURE <Identifier>;<Block>;}

Plone 語法規則之二

- 7. <Statement> ::= [<AssignmentStatement> |
- <CallStatement> | <CompoundStatement> |
- <IfStatement> | <WhileStatement> |
- <ReadStatement> | <WriteStatement>
- 8. <AssignmentStatement> ::=
- <Identifier> := <Expression>
- 9. <CallStatement> ::= CALL <Identifier>
- 10. <CompoundStatement> ::=
- BEGIN <Statement>{;<Statement>} END
- 11. <IfStatement> ::= IF <Condition> THEN <Statement>
- 12. <WhileStatement> ::=WHILE <Condition>
- DO <Statement>
- 13. <ReadStatement> ::= READ (<IdentifierList>)
- 14. <WriteStatement> ::= WRITE (<IdentifierList>)

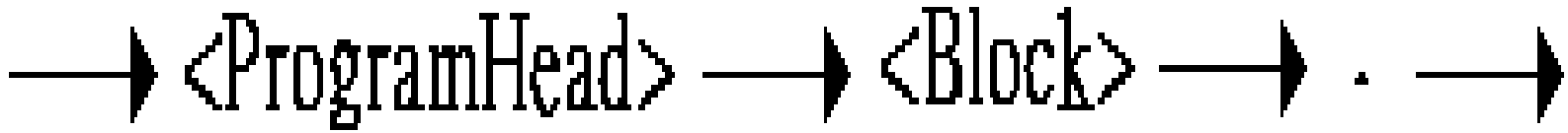
Plone 語法規則之三

- 15 <IdentifierList> ::= <Identifier> { , <Identifier> }
- 16. <Condition> ::= <Expression> \<|<=|=|<>|>|>=\
- <Expression>
- 17. <Expression> ::= [+|-] <Term> { \+|-\ <Term> }
- 18. <Term> :: = <Factor> { \ *|/ \ <Factor> }
- 19. <Factor> ::= <Identifier>|<Number>| (<Expression>)
- 20. <Identifier> ::= <Alpha> { <Alpha> | <Digit> }
- 21. <Number> ::= <Digit> { <Digit> }
- 22. <Alpha> ::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|
- S|T|U|V|W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|
- m|n|o|p|q|r|s|t|u|v|w|x|y|z
- 23. <Digit> ::= 0|1|2|3|4|5|6|7|8|9
- 24. <String> ::= " 任何非雙引號的字元集合 "

語法規則 1.

1. **<Program> ::= <ProgramHead><Block>.**

其語法圖如下：



語法規則 2.

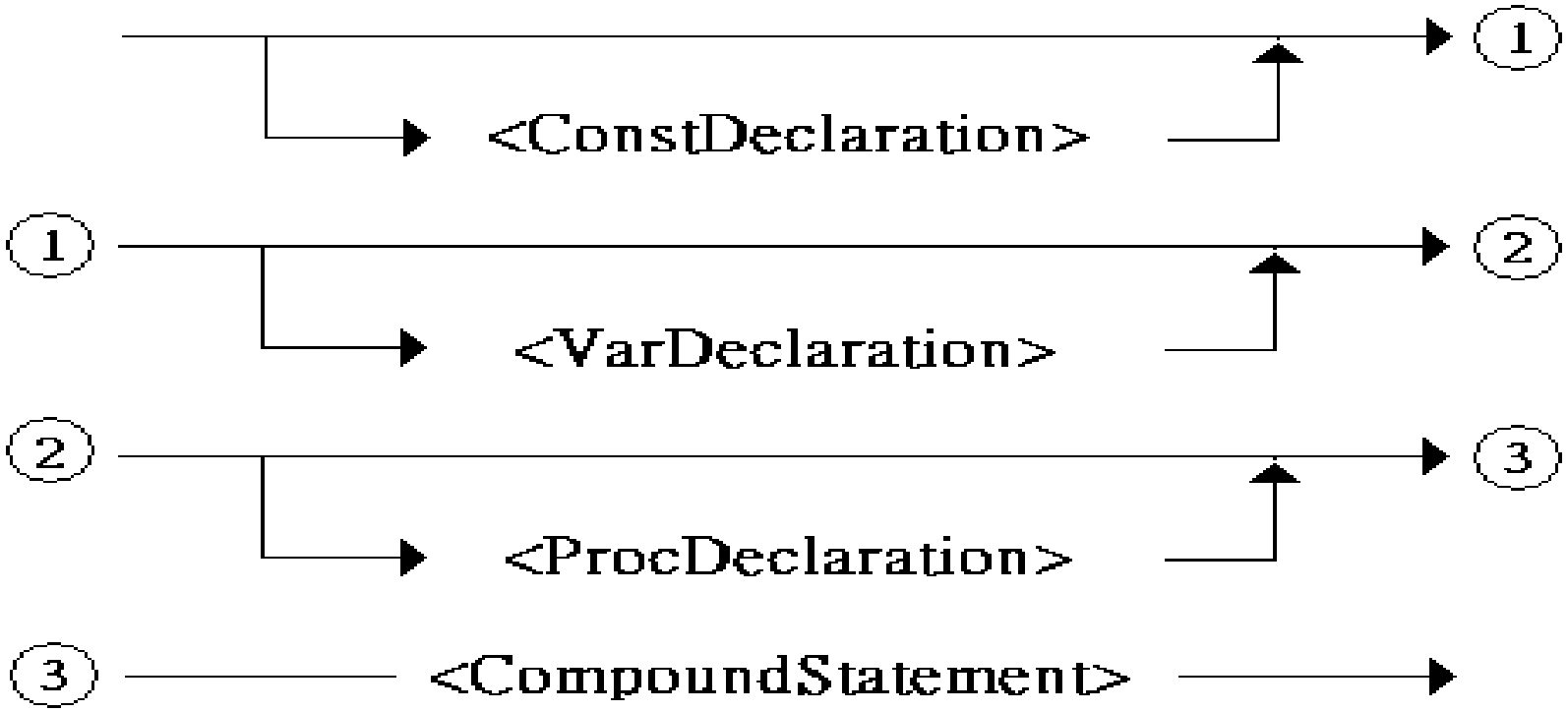
2. <ProgramHead> ::= PROGRAM <Identifier> ;

其語法圖如下：

語法規則 3.

3. <Block> ::= [<ConstDeclaration>]
[<VarDeclaration>]
[<ProcDeclaration>]
<CompoundStatement>

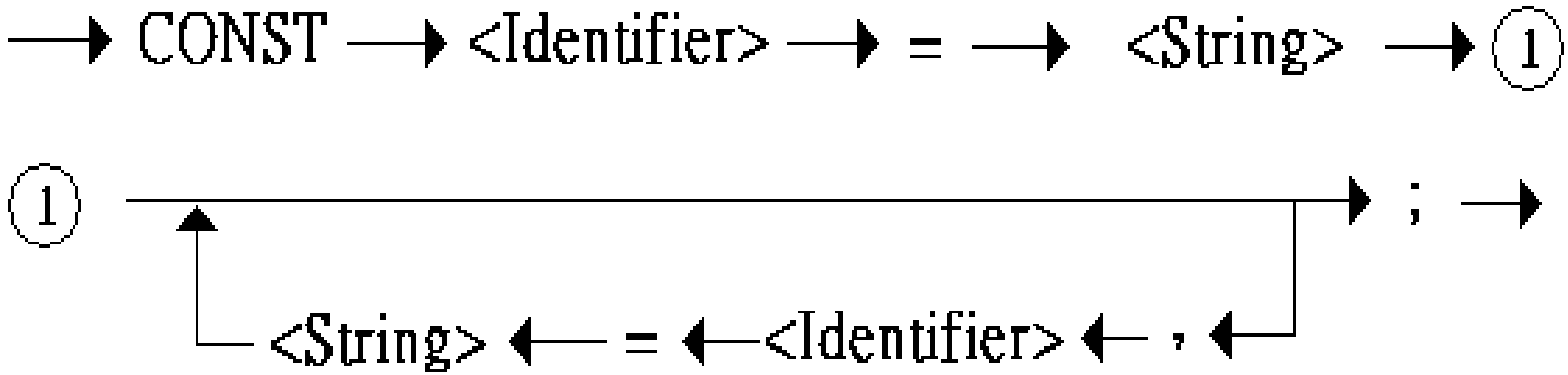
其語法圖如下：



語法規則 4.

4. <ConstDeclaration> ::=
   CONST <Identifier>=<String>
   { , <Identifier>=<String> } ;

其語法圖如下：



語法規則 5.

5. <VarDeclaration> ::= VAR <IdentifierList> ;

其語法圖如下：

語法規則 6.

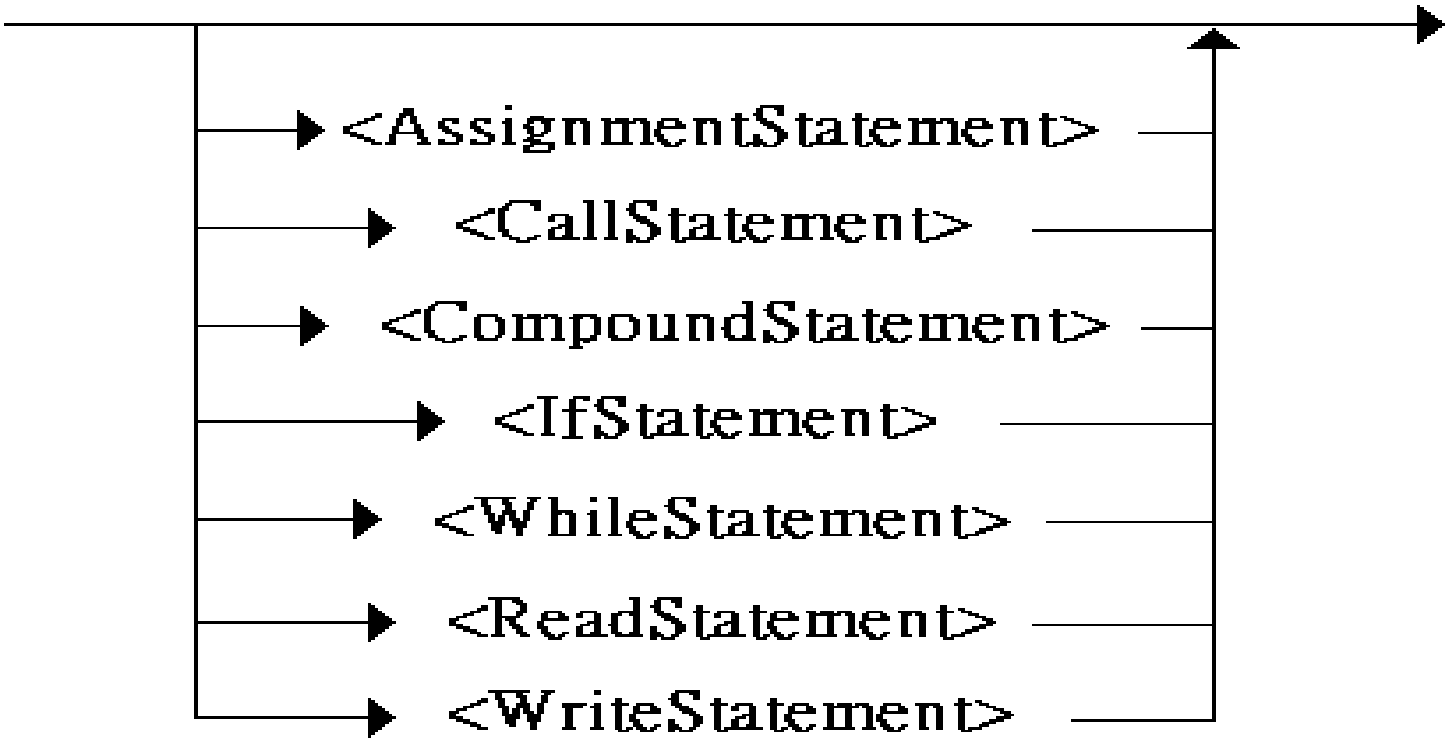
6. <ProcDeclaration> ::=
   {PROCEDURE <Identifier>;<Block>;}

其語法圖如下：

語法規則 7.

7. <Statement> ::= [<AssignmentStatement> | <CallStatement> | <CompoundStatement> | <IfStatement> | <WhileStatement> | <ReadStatement> | <WriteStatement>]

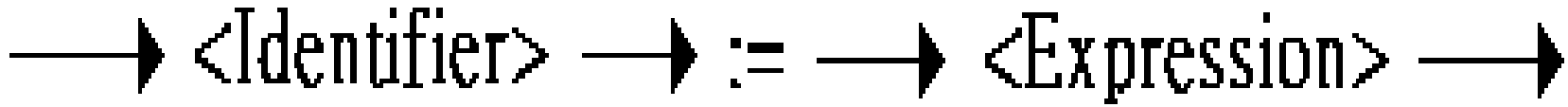
其語法圖如下：



語法規則 8.

8. <AssignmentStatement> ::=
<Identifier> := <Expression>

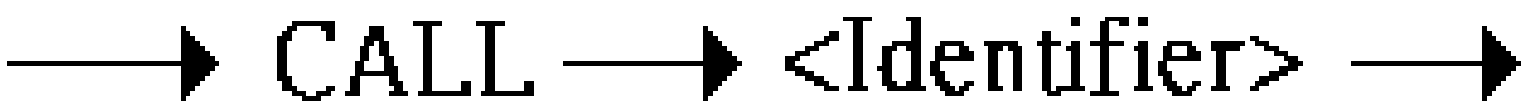
其語法圖如下：



語法規則 9.

9. <CallStatement> ::= CALL <Identifier>

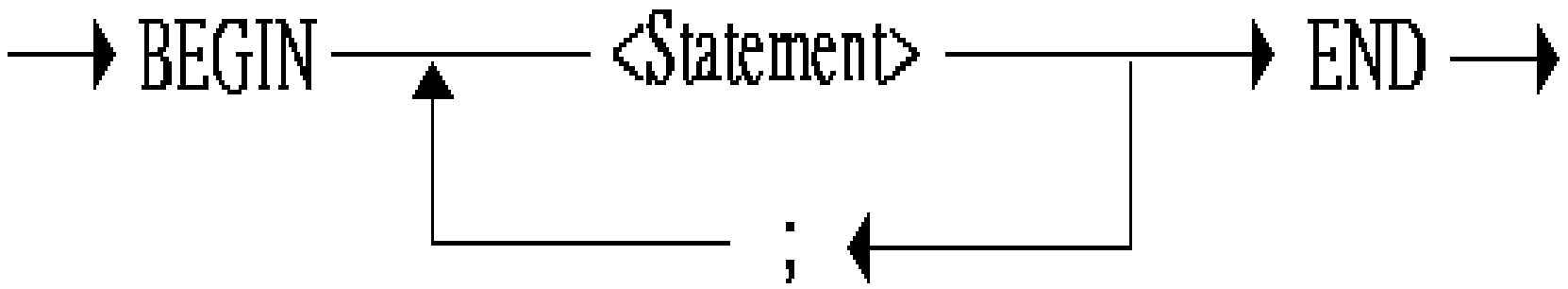
其語法圖如下：



語法規則 10.

10. <CompoundStatement> ::=
BEGIN <Statement> {; <Statement>} END

其語法圖如下：



語法規則 11.

11. <IfStatement> ::= IF <Condition>
THEN <Statement>

其語法圖如下：

語法規則 12.

12. <WhileStatement> ::= WHILE <Condition>
DO <Statement>

其語法圖如下：

語法規則 13.

13. <ReadStatement>::=READ(<IdentifierList>)

其語法圖如下：

語法規則 14.

14. <WriteStatement>::=WRITE(<IdentifierList>)

其語法圖如下：

語法規則 15.

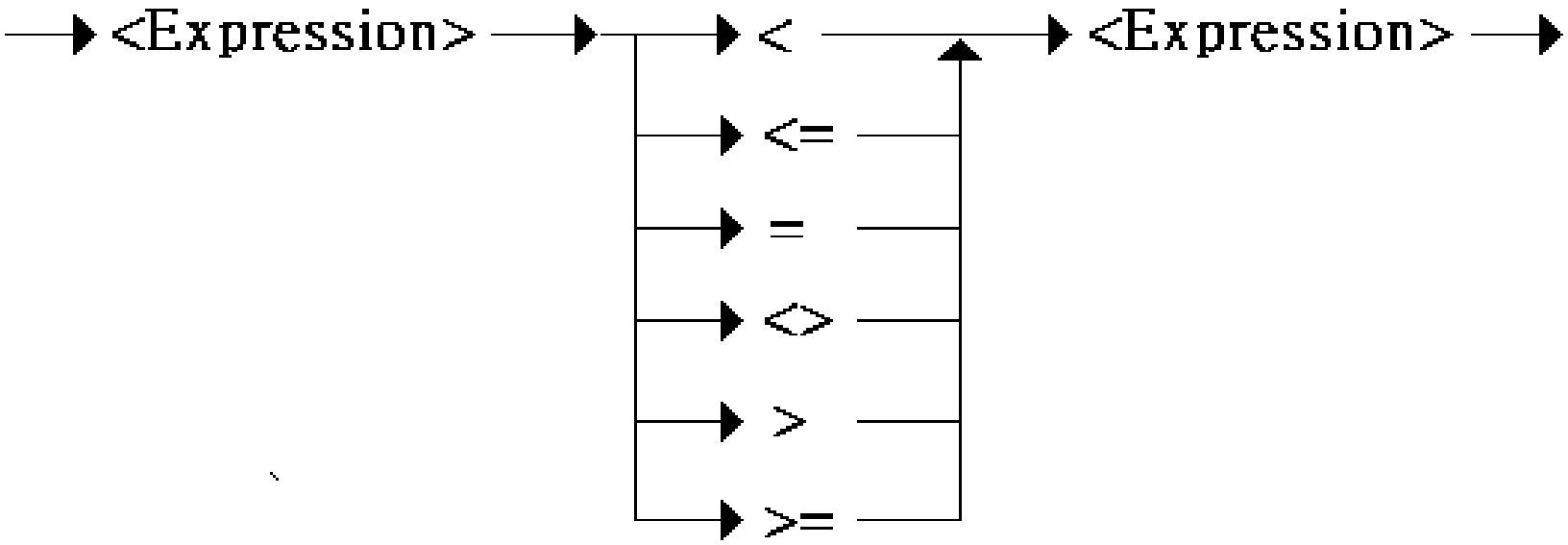
15. <IdentifierList> ::= <Identifier>{,<Identifier>}

其語法圖如下：

語法規則 16.

16. <Condition> ::= <Expression>
\<|<=|=|<>|>|>=\ <Expression>

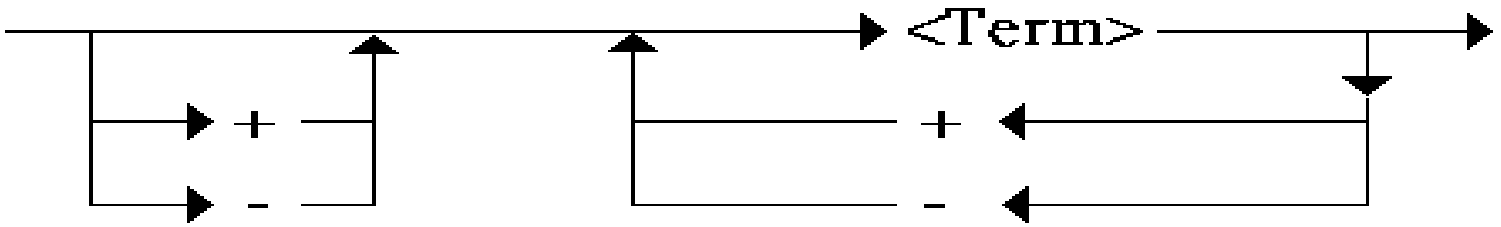
其語法圖如下：



語法規則 17.

17. <Expression> ::= [+|-] <Term> { \+|-\ <Term> }

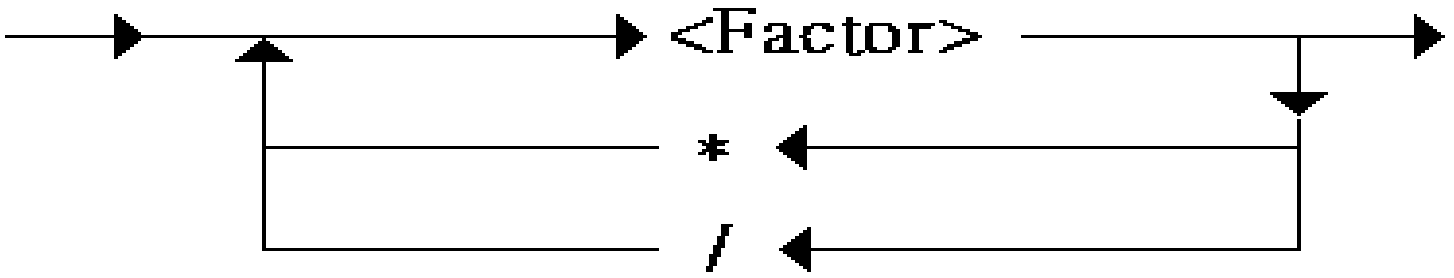
其語法圖如下：



語法規則 18.

18. <Term> :: = <Factor> { \ * | / \ <Factor> }

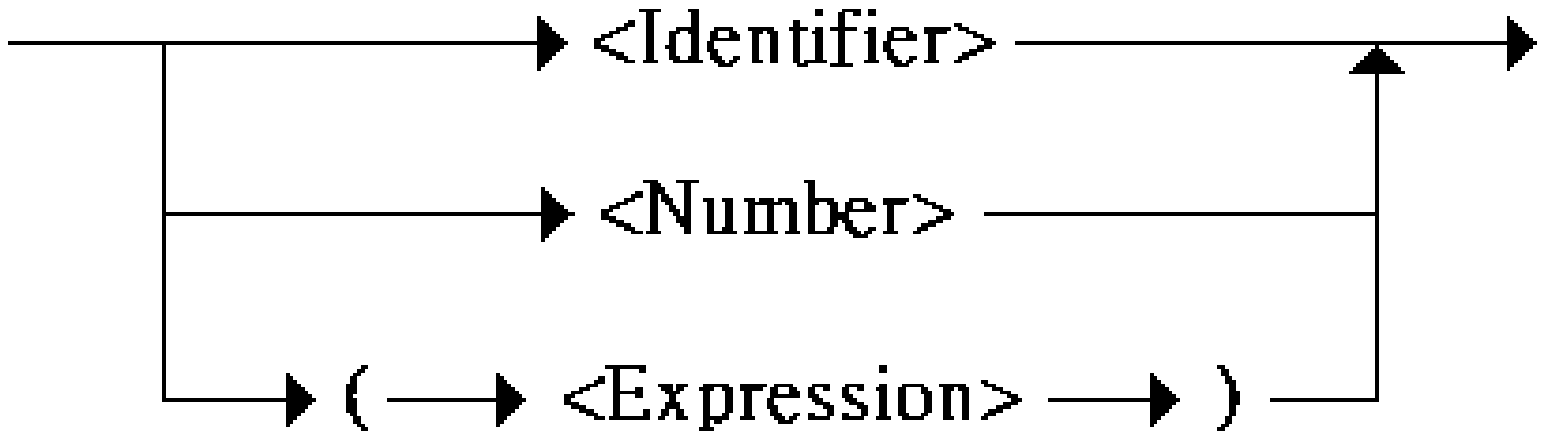
其語法圖如下：



語法規則 19.

19. <Factor> ::= <Identifier>|<Number>|(<Expression>)

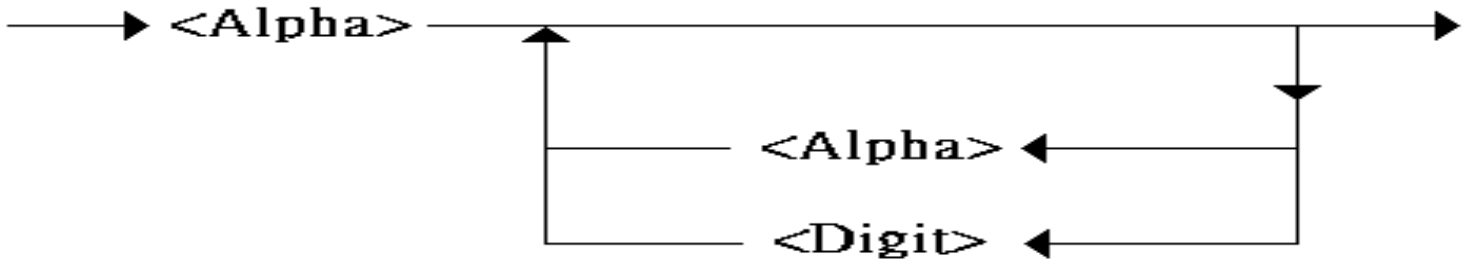
其語法圖如下：



語法規則 20.

20. <Identifier> ::= <Alpha> { <Alpha> | <Digit> }

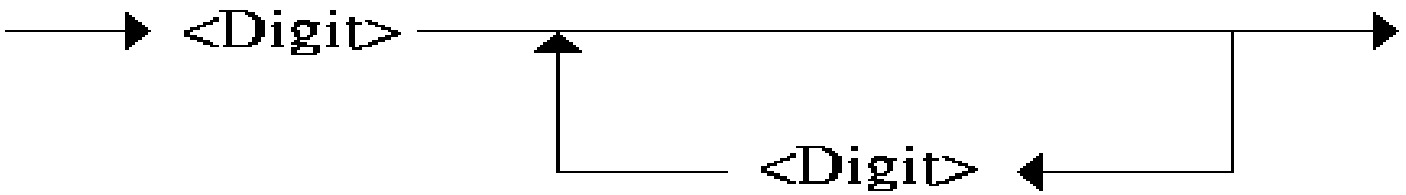
其語法圖如下：



語法規則 21.

21. <Number> ::= <Digit> { <Digit> }

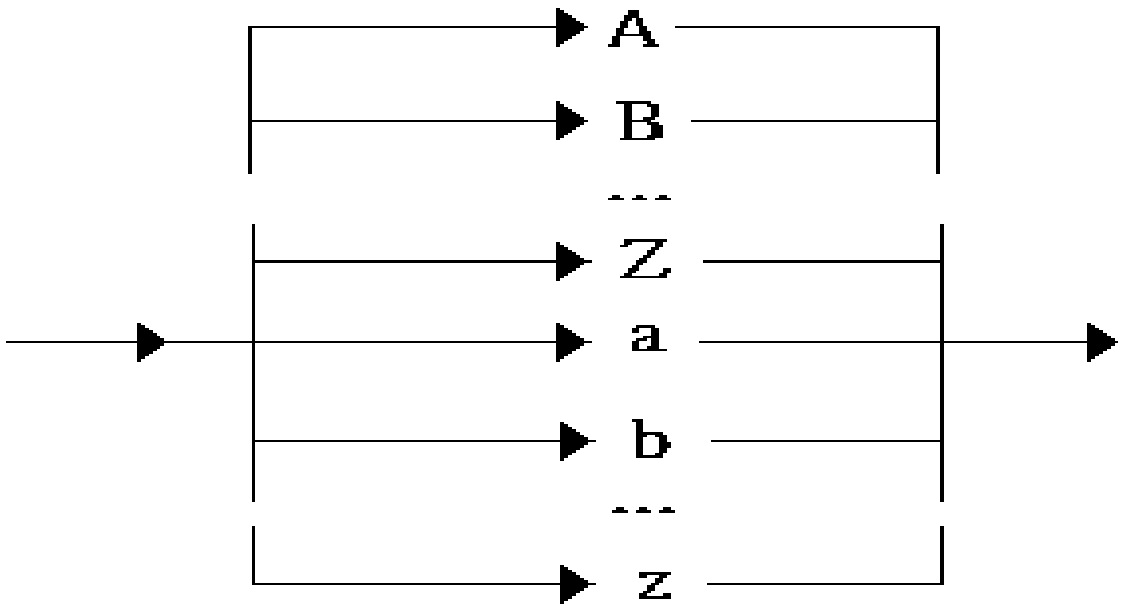
其語法圖如下：



語法規則 22.

22. <Alpha> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|
O|P|Q|R|S|T|U|V|W|X|Y|Z|
a|b|c|d|e|f|g|h|i|j|k|l|m|n|
o|p|q|r|s|t|u|v|w|x|y|z

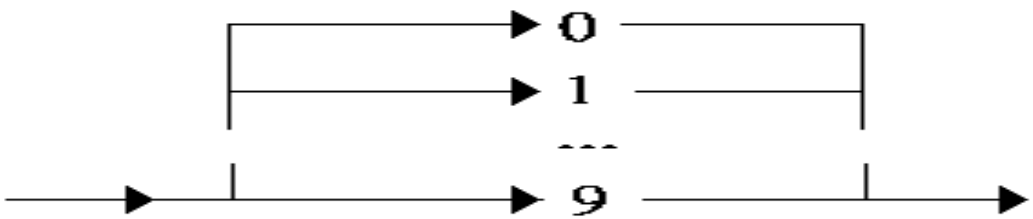
其語法圖如下：



語法規則 23.

23. <Digit> ::= 0|1|2|3|4|5|6|7|8|9

其語法圖如下：



語法規則 24.

24. <String> ::= " 任何非雙引號的字元集合 "

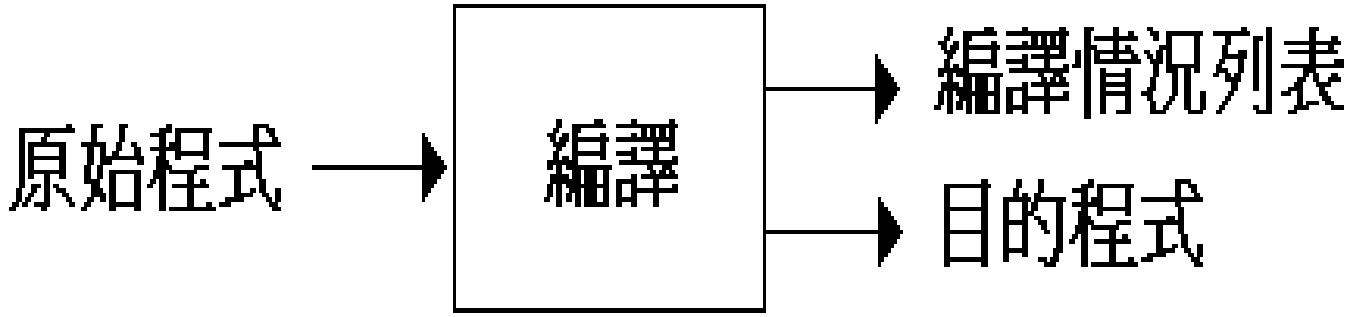
其語法圖如下：

由上而下（top-down）剖析

- 由以上語法之說明可看出是由大而小排列，最大之語法結構為非終端符號程式 <Program>，最小之語法結構為非終端符號 <String> 字串，就像一棵樹一樣，由樹根而樹枝而樹葉，故又稱為語法樹（syntax tree）。其語法之剖析由上而下（top-down），由根而葉，由非終端符號的樹根至終端符號的樹葉。

4.3 隔離有關設備部份

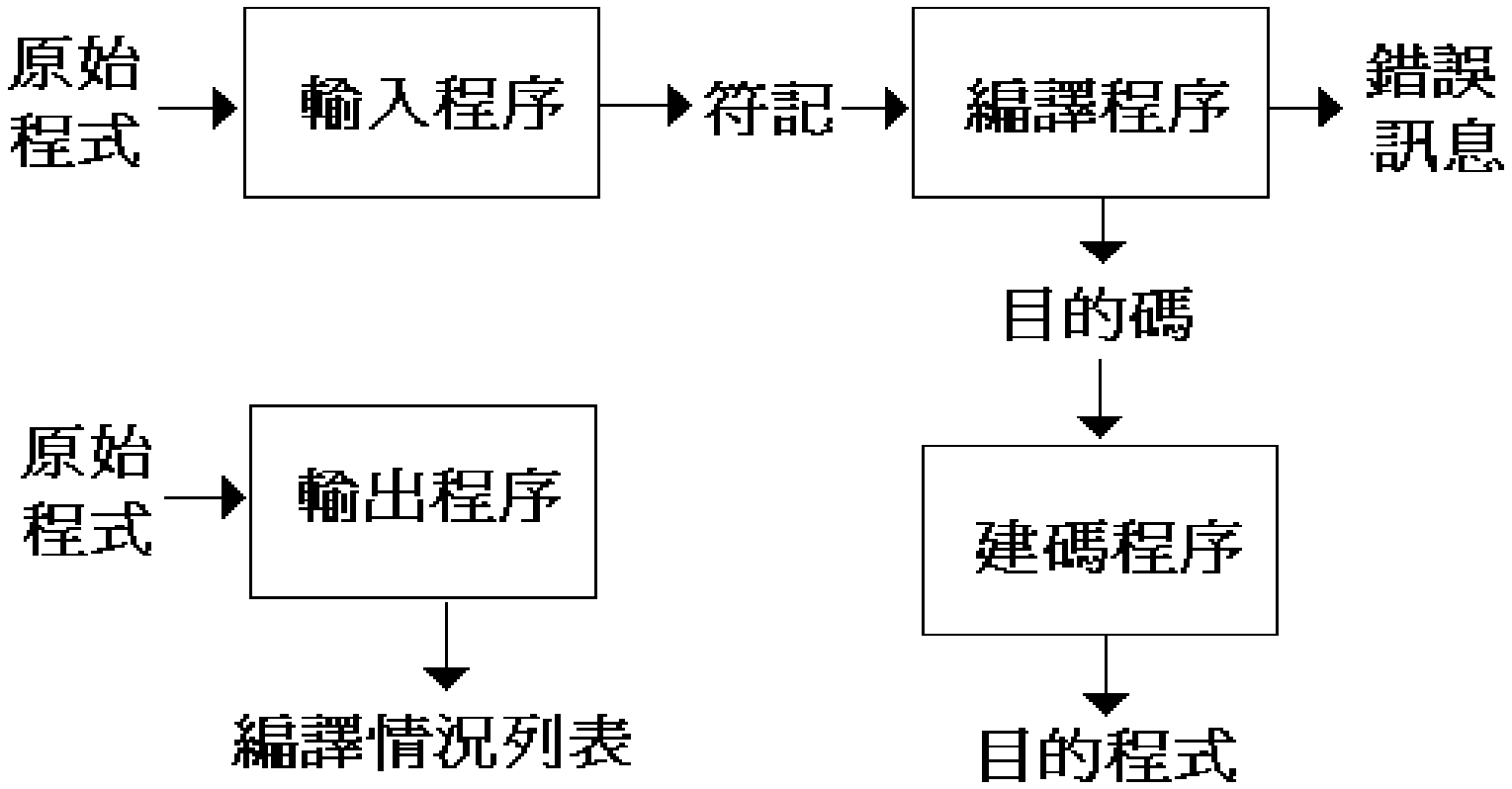
- 設計編譯器的第一個概念就是輸入原始程式，然後產生目的程式及編譯情況列表，如下圖所示。

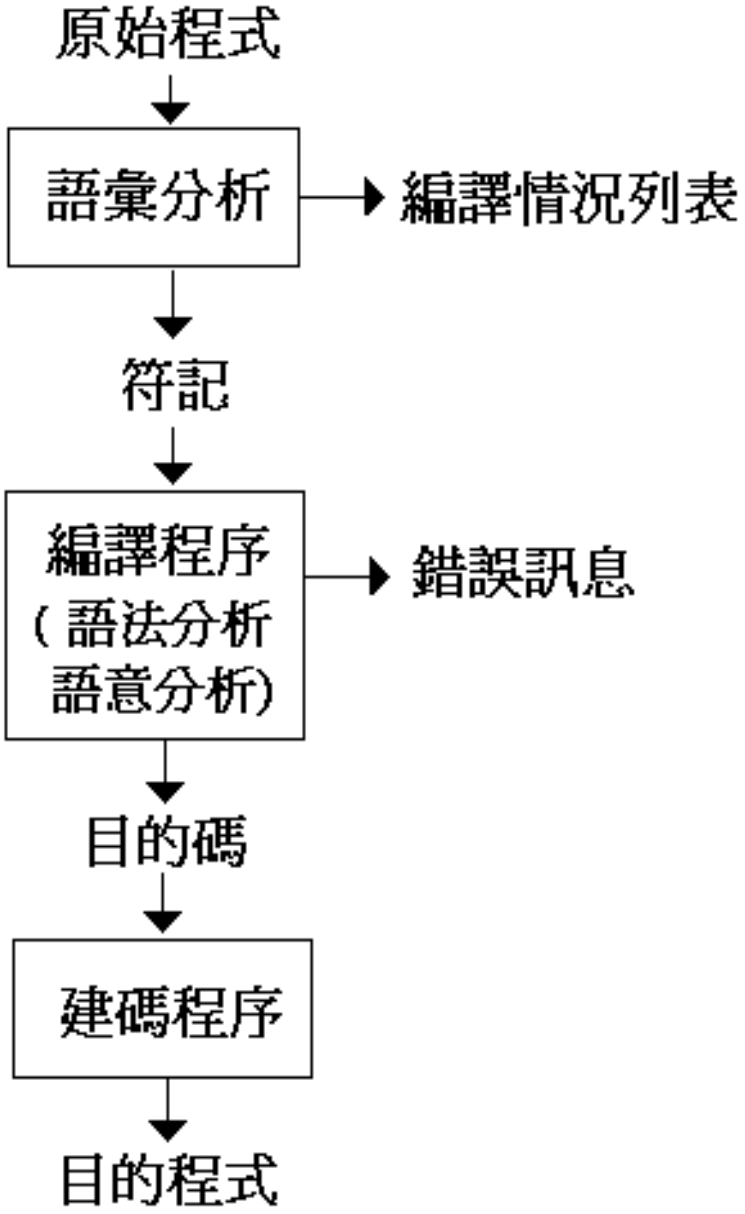


將與設備有關的部份分離出來

- 原始程式、編譯情況列表、目的程式等三個單元均涉及輸入、輸出媒體，而與設備無關是我們設計編譯器的目標，因此必須將與設備有關的部份從編譯器程式中分離出來，試考慮下列三個程序。
- ***(1)．輸入程序：***
- **將與設備有關的原始程式轉換成無關的形式。**
- ***(2)．輸出程序：***
- **將與設備有關的原始程式和由編譯器所**產生
- **的錯誤訊息一起輸出 。**
- ***(3)．建碼程序：***
- **將由編譯器所**產生與設備無關之目的碼轉換
- **成與設備有關的形式 。**
- 如此可以將與設備無關的功能分開，編譯器的結構如下圖所示。

分開與設備無關功能





上圖中之四個程序並不意味著依序執行，可以幾個程序同時進行，或一個程序進行當中呼叫另一個程序。若這種結構可行，則可達到隔離有關設備部份的目的。輸入程序和輸出程序均要處理原始程式，為了提高效率，原始程式必須只輸入一次，從頭到尾處理一次，因此將這兩個程序合併成一個程序，稱為語彙分析（lexical analysis），負責產生並輸出符記（token），因此編譯程式的結構修正如左圖所示。