第五章 語彙分析

- 語彙分析包含掃描欲被編譯的原始程式本文文字，以及辨認出組成原始程式的符記。掃描程式負責辨認出組成原始程式裡的關鍵字（keyword）、運算子（operator）、識別字、數字等等的語法最小單元，稱為符記。這些符記的組成規則依被編譯程式語言的語法而定。
- 本章要討論的是 Plone 程式語言的語法，依該語法辨認出組成原始程式本文文字裡的符記。

5.1 Plone 程式語言語法

- 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 "

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

- 就以語法規則編號 20 的識別字 <Identifier> 非終端符號來講，它規定第一個字元必須是英文字母，不管大小寫都可以，第二個字元（含）以後可以跟隨著英文字母或數字，其他的字元則不合規定，這是 Plone 程式語言的規定，其他的程式語言不見得做這樣的規定，像 Java 程式語言的識別字可含底線（under score），因此每一種程式語言辨認的符記會不同的。

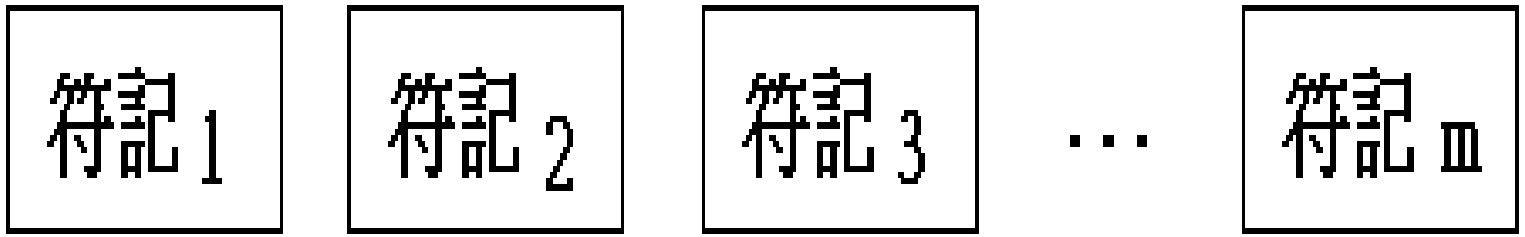
<Number> ::= <Digit>{<Digit>}

- 語法規則編號 21，它規定數字 <Number> 非終端符號第一個字元必須是阿拉伯數字，若有第二個字元（含）也必須是阿拉伯數字，第三、四、等等字元都必須是阿拉伯數字，其他的字元不符合數字 <Number> 的語法。

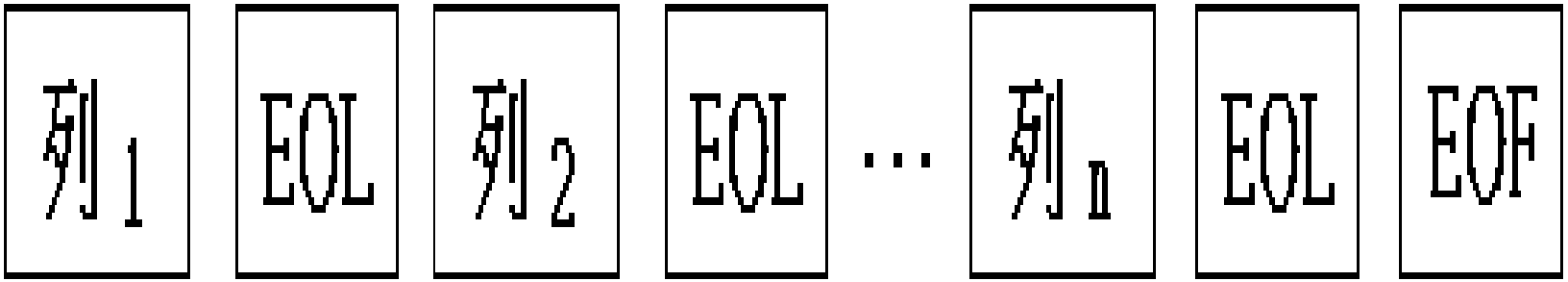
5.2 符記編號

- 我們的掃描程式屬於特殊用途，例如針對 Plone 程式語言語法的掃描，通常可以更為有效的執行辨認的工作，因為原始程式大部份是由這種多字元的識別字所組成的，若能有效率的處理，將可節省許多剖析的時間。
- 當然掃描程式通常會直接辨認單一字元和多字字元的符記，例如 ***VAR*** 會被辨認成一個符記，而非三個英文字母 ***V***、***A***、***R***。「**:=**」會被辨認成一個符記，而非兩個特殊字元「**:**」及「**=**」。雖然可一次一個字元的方式來處理多字字元的符記，卻增加工作量，不過還好現代的電腦其輸入均透過緩衝器執行，一次一個字元的方式來處理多字字元的符記，還是可行的方式。

掃描程式的輸入是原始程式本文文字，輸出是一序列（sequence）的符記。如下圖所示。



原始程式檔（file）是由原始程式列（line）所組成，而原始程式列又由字元所組成，列與列間存在一個列結束字元（End Of Line），以 EOL 表示，EOL 在 Unix 或 Linux 是以新列字元 '\n' 表示，但在 Windows 視窗作業系統卻以兩個字元表示，即 '\r' 及 '\n'。原始程式檔屬本文檔，在檔尾處有一個檔案結束符號（EOF），在 Windows 系統以 Ctrl-Z 表示。因此整個原始程式檔可看成一長字串，如下圖所示。



sym.h

- 為了往後的使用方便，每一個符記以一個整數表示，plone 程式語言的終端符號（terminal symbol）符記以及非終端符號（non-terminal symbol）符記的整數表示定義於 sym.h 表頭檔裡頭。這些符記的整數表示，其值及前後位置並無關係。
- 從表頭檔裡您可看到一個有（sym***SYMMAX***） 35 個整數常數值，從 0 編至 34，常數名稱就代表相當的符記。例如 ***symCONST*** 就是代表 "CONST" 符記，***symBEGIN*** 就是代表 "BEGIN" 符記，***symerror*** 表示錯誤的符記，***symEOF*** 表示已至檔尾（End Of File）。

5.3 符記結構

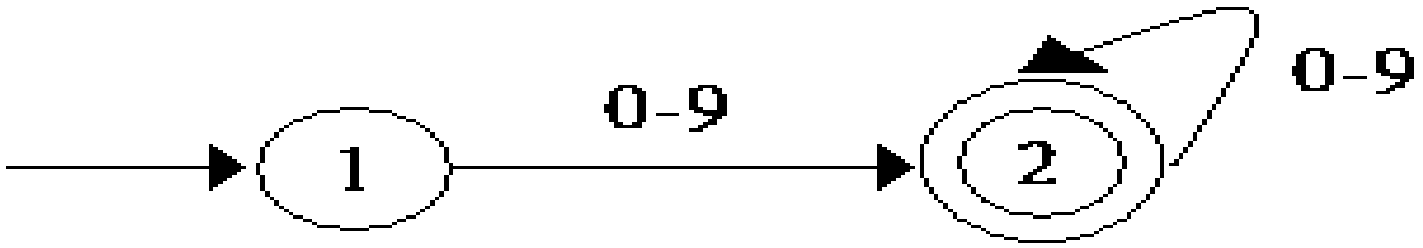
- 當掃描程式掃描到的符記是關鍵字（也稱為保留字）或運算子時，這種符記的整數已提供完全的資訊，但對於像識別字符記而言卻不夠，還必須提供識別字的名稱，對於數字、浮點數、字串等等的符記亦復如此。除了符記名稱之外，有時需要報告錯誤的訊息，還必須曉得該符記名稱的位
- 置，例如在原始程式檔裡的第幾列，在該列裡第幾行，因此有必要將符記設計成一個結構，透過下列的資料您可輕易建立一個 symbol.h 表頭檔的符記結構。

Symbol.java

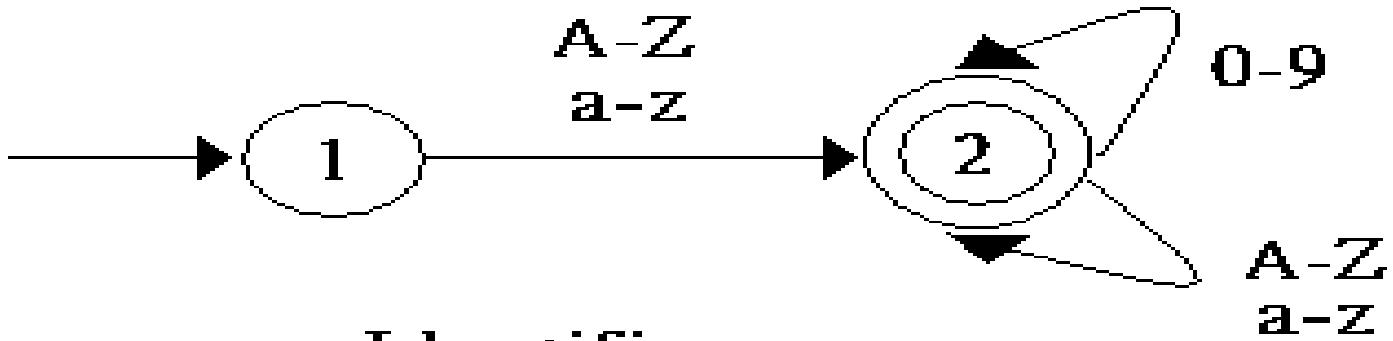
- 在下列的 symbol.h 表頭檔裡：
- int sym;
- int left, right;
- char value[36];
- 宣告了四個***變數成員***。
- ***sym***
- 符記整數編號，也就是 sym.h 裡頭所定義的整數常數。
- ***left***
- 用來記錄符記字串在原始程式裡的列數（lines）。
- ***right***
- 記錄符記字串在原始程式裡該列的開始行數（columns）。
- ***value***
- 是一個字元陣列，用來記錄符記字串等資料。

5.4 有限狀態自動機

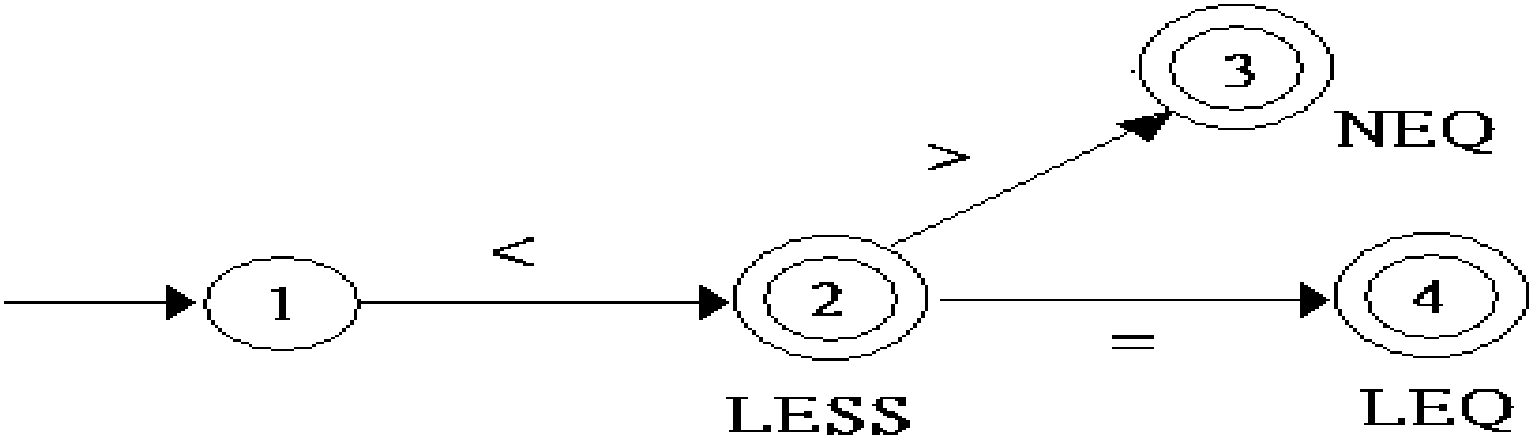
- 大部份程式語言的符記都可以使用有限狀態自動機（finite automata）來辨認。在數學上定義一組有限的狀態（state）和一組從一個狀態至另一個狀態的轉移（transition）所組成。
- 開始的狀態稱為起始狀態（start state），最後一個狀態稱為最終狀態（final state）。有限狀態自動機常以圖形表示。



Number



Identifier


1
<
2
LESS
>
3
NEQ
=
4
LEQ

辨認成功或失敗

- 有限狀態自動機停止在最終狀態時表示辨認***成功***，否則辨認***失敗***。有限狀態自動機提供一種簡單的方式模擬掃描程式的運作情形，這種表示方法其優點在於實作容易。

5.5 設計掃描程式

- 掃描程式首先要考慮如何取得下一字元。原始程式檔是由原始程式列所組成，而原始程式列又由字元所組成，列與列間存在著一個***列結束字元***（EOL），在檔尾處存在著一個***檔案結束符號***（EOF）。
- 設 cp 為列中字元之位置，nextChar 為***所取得的下一字元***。line 為輸入一列的位元組陣列，若 cp 為一列之最後字元位置則換另一列。直到整個程式檔結束為止。

取得下一個符記 nextToken()

- 首先檢查由 advance() 所取得的下一字元 nextChar 是否為空白字元，若是則繼續取得下一字元，直到不是空白字元為止。不是空白字元，那是什麼字元呢？若屬於 A-Z 或 a-z 則是識別字 Identifier 符記。若屬於 0-9 則為 Number 數字符記。若是某符號則為某符號符記。其他，則屬其他符記，須特殊處理。

傳回一個 symbolTag 的結構指標

- 取得下一個符記 nextToken() 方法所傳回的是一個 struct symbolTag * 結構指標，識別字 Identifier 符記辨認成功時以下列敘述傳回：
- **return newSymbol(symIDENTIFIER, linenum, cp, s);**
- 傳回一個結構指標，第一個引數（argument）為 Identifier符記的整數代碼，即 sym.h 表頭檔裡所定義的 symIDENTIFIER 整數常數，其值為 2，這個值傳給第一個參數。同樣的引數列編號 linenum 傳給 left 參數，cp 傳給 right 參數，s 為符記名稱字串傳給 value 參數。

傳回一個 struct symbolTag * 的指標

- 辨認數字符記 Number 成功時以下列敘述傳回：
- **return newSymbol(symNUMBER, linenum, cp, s);**
- 格式與 Identifier 符記一樣，所不同的是符記編號為 symNUMBER 而已，其值多少並不重要。
- 辨認 '+' 符記成功時以下列敘述傳回：
- **return newSymbol(symPLUS, linenum, cp, s);**
- 格式與 Number 符記一樣，所不同的是符記編號為 symPLUS 而已。

- 在掃描程式 scanner.h 表頭檔裡宣告一個 FILE *sourcefile 檔案指標，名為 sourcefile 為輸入原始程式檔，它是透過 scanner() 函式的參數傳進來，並儲存起來備用。
- ```
  FILE *sourcefile;
  void scanner(FILE *f)
  {
    sourcefile = f;
  }
  ```
- 掃描程式 scanner.h 表頭檔裡的 nextToken() 函式用到符記整數編號的表頭檔 sym.h 以及符記表頭檔 symbol.h，因此必須將它們先行引入。
- ```
  #include "sym.h"
  #include "symbol.h"
  ```

5.6 語彙分析程式

- 語彙分析程式 lexer.c 使用前面提過的三個表頭檔程式。
- 1. 符記整數編號的 sym.h 表頭檔程式。
- 2. 符記結構的 symbol.h 表頭檔程式。
- 3. 掃描程式 scanner.h 表頭檔程式。
- 引入 sym.h 及 symbol.h。
- 因為 sym.h 及 symbol.h 已在 scanner.h 表頭檔中先行引入了，因此在語彙分析程式 lexer.c 中只須引入 scanner.h 就可以了。
- #include "scanner.h"

- 下列的敘述宣告一個符記結構指標 token。
- struct symbolTag *token;
- 請注意它擺在 main() 函式的前面，屬於整體變數，宣告在其後的每一個函式都可以存取得到。
- FILE *f=fopen(argv[1], "r");
- scanner(f);
- 從命令列取得輸入原始程式檔名 argv[1]，建立相當的檔案指標 f，傳給掃描函式 scanner(f) 當掃描輸入檔。

```
while ((token=nextToken()) != NULL)
{
  if (token->sym == symerror || argc==3)
  {
    if (token->sym==symerror)
      printf("***錯誤***");
    printf("\t符記編號=%d\t列號=%d\t行號=%d"
      "\t符記名稱=\"%s\"\n", token->sym,
      token->left, token->right, token->value);
  }
}
```

- 重複取得下一個符記，一直到沒有符記為止。對於每一個符記判斷是否為錯誤的符記或是否從命令列輸入三個引數（argument），只要一個條件成立就輸出符記內容。

- **C:\plone\ch05> lexer test52.pl <Enter>　[註]兩個引數**
- **C:\plone\ch05> lexer test52.pl 1 <Enter> [註]三個引數**
-
- 上列的前一個命令只輸出錯誤的符記，後一個命令輸出所有符記，包括錯誤訊息 "***錯誤***"。
- 　　fclose(f);
- 　　printf("\n 語彙分析完成。");
- 最後關閉輸入檔，輸出語彙分析完成訊息。

5.7測試程式

- 測試程式 test51.pl 完全依據 plone 語法撰寫而成，因此語彙分析沒有錯誤，執行 lexer.exe 時從命令列輸入第三個引數（1），強迫輸出每一個符記的內容。

- C:\plone\ch05> lexer test51.pl 1 <Enter>
- 1 PROGRAM test51;
- 符記編號=2 列號=1 行號=7 符記名稱="PROGRAM"
- 符記編號=2 列號=1 行號=14 符記名稱="test51"
- 符記編號=17 列號=1 行號=15 符記名稱=";"
- 2 CONST
- 符記編號=2 列號=2 行號=5 符記名稱="CONST"
- 3 msg1=" x=",
- 符記編號=2 列號=3 行號=6 符記名稱="msg1"
- 符記編號=8 列號=3 行號=7 符記名稱="="
- 符記編號=34 列號=3 行號=12 符記名稱=" x="
- 符記編號=16 列號=3 行號=13 符記名稱=","
- 4 msg2=" y=";
- 符記編號=2 列號=4 行號=6 符記名稱="msg2"
- 符記編號=8 列號=4 行號=7 符記名稱="="
- 符記編號=34 列號=4 行號=12 符記名稱=" y="
- 符記編號=17 列號=4 行號=13 符記名稱=";"
- 5 VAR
- 符記編號=2 列號=5 行號=3 符記名稱="VAR"

- 6 x, y;
- 符記編號=2 列號=6 行號=3 符記名稱="x"
- 符記編號=16 列號=6 行號=4 符記名稱=","
- 符記編號=2 列號=6 行號=6 符記名稱="y"
- 符記編號=17 列號=6 行號=7 符記名稱=";"
- 7 BEGIN
- 符記編號=2 列號=7 行號=5 符記名稱="BEGIN"
- 8 x := 3;
- 符記編號=2 列號=8 行號=3 符記名稱="x"
- 符記編號=19 列號=8 行號=6 符記名稱=":="
- 符記編號=3 列號=8 行號=8 符記名稱="3"
- 符記編號=17 列號=8 行號=9 符記名稱=";"
- 9 WHILE x>0 DO
- 符記編號=2 列號=9 行號=7 符記名稱="WHILE"
- 符記編號=2 列號=9 行號=9 符記名稱="x"
- 符記編號=12 列號=9 行號=10 符記名稱=">"
- 符記編號=3 列號=9 行號=11 符記名稱="0"
- 符記編號=2 列號=9 行號=14 符記名稱="DO"

- 10 BEGIN
- 符記編號=2 列號=10 行號=9 符記名稱="BEGIN"
- 11 y := x*3+6;
- 符記編號=2 列號=11 行號=7 符記名稱="y"
- 符記編號=19 列號=11 行號=10 符記名稱=":="
- 符記編號=2 列號=11 行號=12 符記名稱="x"
- 符記編號=6 列號=11 行號=13 符記名稱="*"
- 符記編號=3 列號=11 行號=14 符記名稱="3"
- 符記編號=4 列號=11 行號=15 符記名稱="+"
- 符記編號=3 列號=11 行號=16 符記名稱="6"
- 符記編號=17 列號=11 行號=17 符記名稱=";"
- 12 WRITE(msg1,x);
- 符記編號=2 列號=12 行號=11 符記名稱="WRITE"
- 符記編號=14 列號=12 行號=12 符記名稱="("
- 符記編號=2 列號=12 行號=16 符記名稱="msg1"
- 符記編號=16 列號=12 行號=17 符記名稱=","
- 符記編號=2 列號=12 行號=18 符記名稱="x"
- 符記編號=15 列號=12 行號=19 符記名稱=")"
- 符記編號=17 列號=12 行號=20 符記名稱=";"

- 13 WRITE(msg2,y);
- 符記編號=2 列號=13 行號=11 符記名稱="WRITE"
- 符記編號=14 列號=13 行號=12 符記名稱="("
- 符記編號=2 列號=13 行號=16 符記名稱="msg2"
- 符記編號=16 列號=13 行號=17 符記名稱=","
- 符記編號=2 列號=13 行號=18 符記名稱="y"
- 符記編號=15 列號=13 行號=19 符記名稱=")"
- 符記編號=17 列號=13 行號=20 符記名稱=";"
- 14 x := x-1;
- 符記編號=2 列號=14 行號=7 符記名稱="x"
- 符記編號=19 列號=14 行號=10 符記名稱=":="
- 符記編號=2 列號=14 行號=12 符記名稱="x"
- 符記編號=5 列號=14 行號=13 符記名稱="-"
- 符記編號=3 列號=14 行號=14 符記名稱="1"
- 符記編號=17 列號=14 行號=15 符記名稱=";"
- 15 END;
- 符記編號=2 列號=15 行號=7 符記名稱="END"
- 符記編號=17 列號=15 行號=8 符記名稱=";"
- 16 END.
- 符記編號=2 列號=16 行號=3 符記名稱="END"
- 符記編號=18 列號=16 行號=4 符記名稱="."

測試程式 test52.pl

- 測試程式 test52.pl 依據 plone 語法撰寫，但第 9 列敘述如下：
- 　WHILE x!=0 DO
- 其中字元「!」並非 plone 語言合法的字元，因此顯示錯誤。
-
- 但第 11 列敘述如下：
- 　y := x^3+6;
- 其中字元「^」並非 plone 語言合法的字元，因此顯示錯誤。

```
C:\plone\ch05> lexer test52.pl <Enter>
   1 PROGRAM test52;
   2 CONST
   3   msg1=" x=",
   4   msg2=" y=";
   5 VAR
   6   x, y;
   7 BEGIN
   8   x := 3;
   9   WHILE x!=0 DO
***錯誤***  符記編號=1  列號=9  行號=10 符記名稱="!"
  10    BEGIN
  11      y := x^3+6;
***錯誤***  符記編號=1  列號=11 行號=13 符記名稱="^"
  12      WRITE(msg1,x);
  13      WRITE(msg2,y);
  14      x := x-1;
  15    END;
  16 END.

  語彙分析完成。
```