第六章 語法分析

- 在語法分析的過程當中，程式設計者所撰寫的原始程式敘述被辨認為所使用程式語言的語法結構，這個結構常以 ***剖析樹*** 方式呈現。根據剖析樹建立方式的不同可分為兩個大類，由上而下或由下而上的剖析方式。
- 由上而下的方式開始於樹根，建立一棵剖析樹，企圖使終端節點匹配到被分析的敘述。由下而上的方式開始於樹葉（終端節點），企圖將它們合併成較上一層的樹，一直到樹跟為止。
- 本章說明***由上而下的剖析方式***。

6.1 Plone 程式語言語法

- 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>;}

- 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>)

- 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 "

6.2 sym 符號常數

- 對於 plone 程式語言語法裡所有的終端符號（terminal symbol）、非終端符號（non-terminal symbol）、以及相關的資料均以整數常數表示，定義於 sym.h 表頭檔裡，為了程式設計的方便其整數常數以 0 至 symSYMMAX-1 的連續整數表示，本 plone 程式語言語法裡最大值的整數常數為 symSTRING，其值為整數 34，因此整數常數symSYMMAX 的值為 34+1=35。

```
/*************** symtest.c *************/
#include <stdio.h>
#include "sym.h"
#include "symname.h"
int main()
{
  int i;
  symnameinit();
  for (i=0; i<5; i++)
    printf("sym%s = %d\n", names[i],i);
  return 0;
}
```

6.3 err 錯誤訊息

代號	錯誤訊息意義
0	必須跟著句點.
1	遺漏PROGRAM
2	PROGRAM後必須跟著識別字
3	PROGRAM後識別字必須跟著;
4	遺漏CONST
5	遺漏等號=
6	遺漏逗號,或分號;
7	遺漏VAR
8	遺漏:=
9	遺漏CALL
10	遺漏BEGIN
11	遺漏END
12	遺漏IF
13	遺漏THEN

代號	錯誤訊息意義
14	遺漏WHILE
15	WHILE敘述錯誤,遺漏DO
16	遺漏READ
17	遺漏(
18	遺漏)
19	遺漏WRITE
20	關係運算子錯誤
21	遺漏識別字
22	遺漏數字
23	飛越至下一個敘述
24	CONST宣告常數重複
25	VAR宣告變數重複
26	識別字沒有宣告

```
/******************** errtest.c *****************/
#include <stdio.h>
#include "err.h"
int main()
{
  int i;
  for (i=0; i<5; i++)
    printf("errmsgs[%d]=\"%s\"\n", i, errmsgs[i]);
  return 0;
}
```

6.4 保留字管裡

- 從 plone 程式語言的語法裡我們可以將所有的保留字存放在一個字元陣列 reswords[ ] 裡頭，並提供一個函式 isResword(char *s) 檢查參數字串 s 是否為保留字，若是保留字則傳回在陣列裡的位置，若不是保留字則傳回 -1。

```
/********************** resword.h ******************/
#include <stdlib.h>
#define RESWORDMAX 13
char reswords[RESWORDMAX][10] =
  {
  "BEGIN","CALL","CONST","DO","END","IF","PROCEDURE",
  "PROGRAM","READ","THEN","VAR","WHILE","WRITE"
  };
int isResword(char *s)
{
  int i;
  for (i=0; i<RESWORDMAX; i++)
  {
    if (strcmp(s, reswords[i])==0) return i;
  }
  return -1;
}
```

```
/********************* reswordtest.c ********************/
#include <stdio.h>
#include "resword.h"
int main()
{
  printf("isResword(\"DO\")=%d\n", isResword("DO"));
  printf("isResword(\"PROG\")=%d\n", isResword("PROG"));
  printf("isResword(\"BEGIN\")=%d\n", isResword("BEGIN"));
  printf("isResword(\"WRITE\")=%d\n", isResword("WRITE"));
  return 0;
}
```

6.6 遞迴式下降剖析

- 由上而下的剖析方法也稱為 ***遞迴式下降剖析***（recursive descent），它是由許多程序（procedure）所組成，語法中的每一個非終端符號均由一個指定的程序處理。當一個程序被呼叫時，它嘗試找出以目前符記（token）開頭，可以將它分析為與該程序有關的次字串（substring）。在找次字串的過程當中可能呼叫其他的程序，甚或遞迴式的呼叫本程序，以找尋其他的非終端符號。如果一個程序找到它所需要的非終端符號，就會傳回一個成功的訊息給呼叫程式，否則會傳回一個失敗的訊息。

- 我們先來設計一個 <IdentifierList> 非終端符號程序。這個程序的邏輯非常直接，首先預設為無法匹配，因此布林變數 found 的初值設為 false。然後判斷下一個符記是否為識別字：
- ```
  if（token->sym == symIDENTIFIER）
  {
    //略
  }
  ```
- token 是 symbol 結構變數的指標，其 sym 常數值記錄該符記的整數編號，symIDENTIFIER 是 sym.h 表頭檔裡定義的整數常數，表示它屬於識別字，兩者若相等表示此符記確實為識別字，就執行識別字的語法分析，這時才將變數 found 的值設為 1，表示已經發現所要找的符記了。
- 這裡有一點要注意的是 token->sym 的 sym 是 token 指標所指結構裡的一個欄位，而表示整數常數的 symIDENTIFIER, 其 sym 指的是表頭檔 sym.h 裡所定義的整數常數。
  ```

6.7 報告錯誤

- 若決定在剖析過程當中一發現錯誤馬上就報告，則 ReadStatement() 就不需要有傳回值，只須將「found=false;」敘述改為報告錯誤就可以了。

6.8 plone 語法剖析程序

- 您若能夠將 plone 程式語言的每一個非終端符號寫出相對應的處理方法，將來撰寫程式時將它轉換為程式碼就輕而易舉了。
-
- 下列是每一條語法規則其相對應的處理程序。

- 語法規則 #1
-
- **<Program> ::= <ProgramHead><Block>.**
-
- 其剖析程序如下：
-
- **void Program()**
- **{**
- **ProgramHead();**
- **Block();**
- **if (符記不是句點) Error(0);**
- **}**

- 語法規則 #2
- <ProgramHead> ::= PROGRAM <Identifier> ;

- 其剖析程序如下：

```
void ProgramHead()
{
  if (符記為PROGRAM)
    取得下一符記;
  else
    Error(1);
  if (符記為識別字)
    取得下一符記;
  else
    Error(2);
  if (符記為;分號)
    取得下一符記;
  else
    Error(3);
}
```

- 語法規則 #3
-
- <Block> ::= [<ConstDeclaration>]
- [<VarDeclaration>]
- [<ProcDeclaration>]
- <CompoundStatement>
-
- 其剖析程序如下：
-
- void Block()
- {
- if (符記為CONST) ConstDeclaration();
- if (符記為VAR) VarDeclaration();
- if (符記為PROCEDURE) ProcDeclaration();
- CompoundStatement();
- }

語法規則 #4

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

- 其剖析程序如下：
-
- void ConstDeclaration()
- {
- if (符記為CONST)
- 取得下一符記;
- else
- Error(4);
- Identifier();
- if (符記為=)
- 取得下一符記;
- else
- Error(5);
- if (符記為字串)
- 取得下一符記;
- else
- Error(27);
-
- while (符記為,逗號)
- {
- 取得下一符記;
- Identifier();
- if (符記為=等號)
- 取得下一符記;
- else
- Error(5);
- if (符記為字串)
- 取得下一符記;
- else
- Error(27);
- }
- if (符記為;分號)
- 取得下一符記;
- else
- Error(6);
- }

語法規則 #5

<VarDeclaration> ::= VAR <IdentifierList> ;

- 其剖析程序如下：
-

```
void VarDeclaration()
{
  if (符記為VAR)
    取得下一符記;
  else
    Error(7);
  IdentifierList();
  if (符記為;分號)
    取得下一符記;
  else
    Error(6);
}
```

語法規則 #6

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

- 其剖析程序如下：
-
- ```
  void ProcDeclaration()
  {
    while (符記為PROCEDURE)
    {
      取得下一符記;
      Identifier();
      if  (符記為;分號)
        取得下一符記;
      else
        Error(6);
      Block();
      if  (符記為;分號)
        取得下一符記;
      else
        Error(6);
    }
  }
  ```
  ```

語法規則 #7

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

- 其剖析程序如下：
-

```
void Statement()
{
  if  (符記為保留字)
  {
    if  (符記為CALL)          CallStatement();
    else if  (符記為BEGIN)  CompoundStatement();
    else if  (符記為IF)          IfStatement();
    else if  (符記為WHILE)  WhileStatement();
    else if  (符記為READ)   ReadStatement();
    else if  (符記為WRITE)  WriteStatement();
  }
  else if (符記為識別字)     AssignmentStatement();
}
```

語法規則 #8

<AssignmentStatement> ::= <Identifier> := <Expression>

- 其剖析程序如下：
-
- void AssignmentStatement()
- {
- Idntifier();
- if (符記為:=)
- 取得下一符記;
- else
- Error(8);
- Expression();
- }

語法規則 #9

<CallStatement> ::= CALL <Identifier>

- 其剖析程序如下：
-
-
```
    void CallStatement()
    {
      if  (符記為CALL)
        取得下一符記;
      else
        Error(9);
      Identifier();
    }
```

語法規則 #10

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

- 其剖析程序如下：

```
void CompoundStatement()
{
  if (符記為BEGIN)
    取得下一符記;
  else
    Error(10);
  Statement();
  while (符記為;分號)
  {
    取得下一符記;
    Statement();
  }
  if (符記為END)
    取得下一符記;
  else
    Error(11);
}
```

語法規則 #11

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

- 其剖析程序如下：

```
void IfStatement()
{
  if (符記為IF)
    取得下一符記;
  else
    Error(12);
  Condition();
  if (符記為THEN)
    取得下一符記;
  else
    Error(13);
  Statement();
}
```

語法規則 #12

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

- 其剖析程序如下：

```
void WhileStatement()
{
  if (符記為WHILE)
    取得下一符記;
  else
    Error(14);
  Condition();
  if (符記為DO)
    取得下一符記;
  else
    Error(15);
  Statement();
}
```

語法規則 #13

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

- 其剖析程序如下：

```
void ReadStatement()
{
  if (符記為READ)
    取得下一符記;
  else
    Error(16);
  if (符記為左括號)
    取得下一符記;
  else
    Error(17);
  IdentifierList();
  if (符記為右括號)
    取得下一符記;
  else
    Error(18);
}
```

語法規則 #14

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

- 其剖析程序如下：

```
void WriteStatement()
{
  if (符記為WRITE)
    取得下一符記;
  else
    Error(19);
  if (符記為左括號)
    取得下一符記;
  else
    Error(17);
  IdentifierList();
  if (符記為右括號)
    取得下一符記;
  else
    Error(18);
}
```

語法規則 #15

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

- 其剖析程序如下：
-
- void IdentifierList()
- {
- Identifier();
- while (符記為,逗號)
- {
- 取得下一符記;
- Identifier();
- }
- }

語法規則 #16

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

- 其剖析程序如下：
-
- void Condition()
- {
- Expression();
- if (符記為<或<=或=或<>或>或>=)
- 取得下一符記;
- else
- Error(20);
- Expression();
- }

語法規則 #17

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

- 其剖析程序如下：
-
- void Expression()
- {
- if (符記為+或-)
- 取得下一符記;
- Term();
- while (符記為+或-)
- {
- 取得下一符記;
- Term();
- }
- }

語法規則 #18

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

- 其剖析程序如下：
-
- void Term()
- {
- Factor();
- while (符記為*或/)
- {
- 取得下一符記;
- Factor();
- }
- }

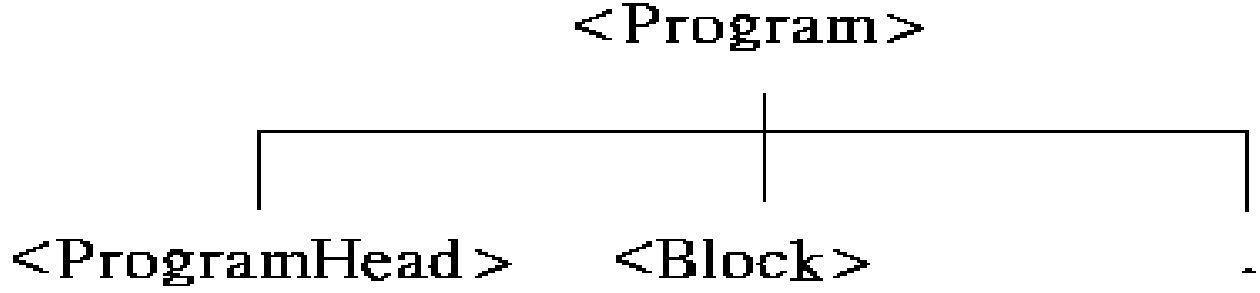
- 其剖析程序如下：
-
- ```
  void Factor()
  {
    if (符記為識別字)
      Identifier();
    else if (符記為數字)
      Number();
    else if (符記為'('左括號)
    {
      取得下一符記;
      Expression();
      if (符記為')'右括號)
        取得下一符記;
      else
        Error(18);
    }
    else
      Error(17);
  }
  ```

語法規則 #19

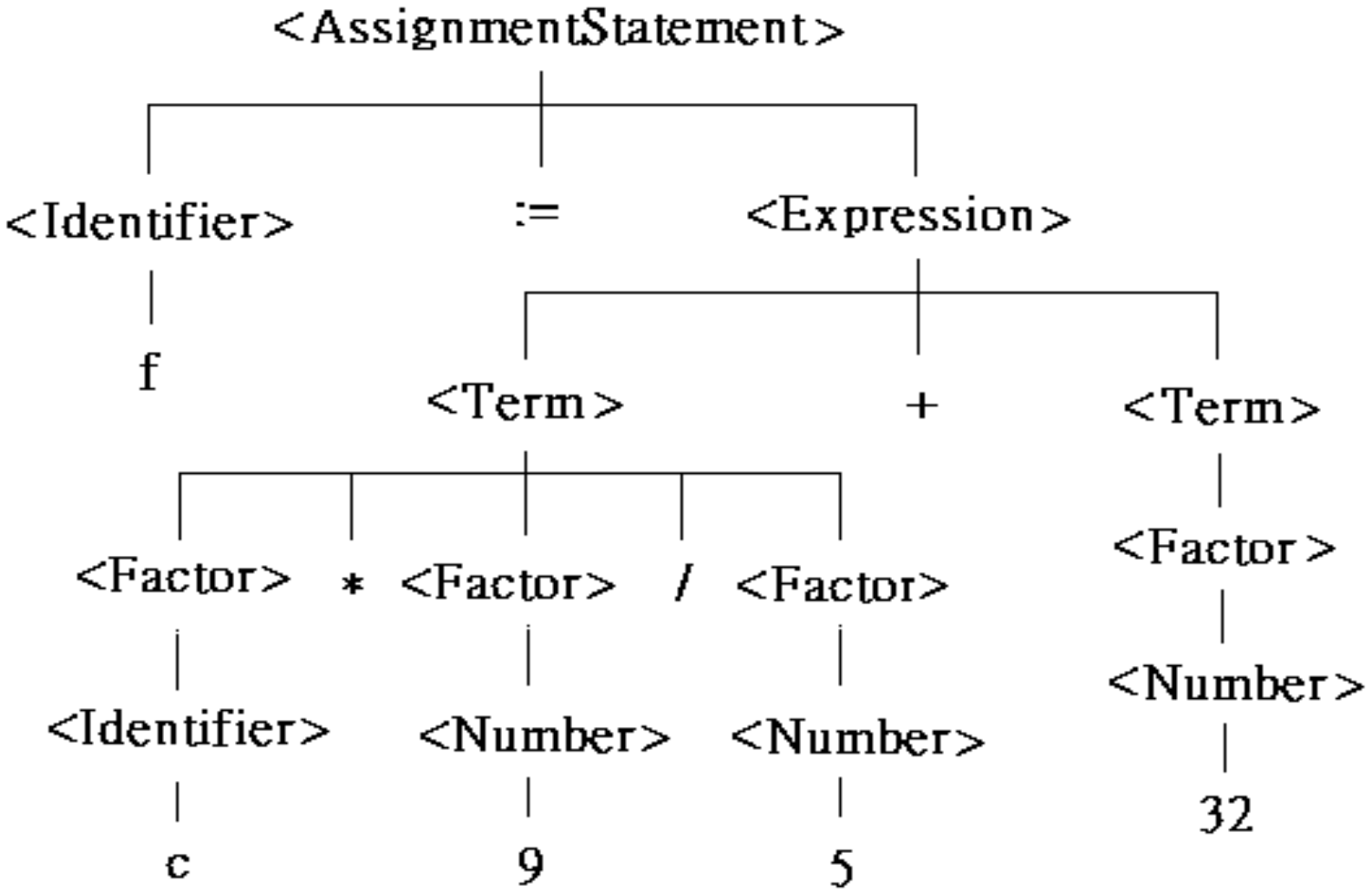
<Factor> ::=
Identifier>
|
<Number>
|
(<Expression>)
  ```

語法規則 #20

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

- 其剖析程序如下：
-
- void Identifier()
- {
- if (符記為識別字)
- 取得下一符記;
- else
- Error(21);
- }

語法規則 #21

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

- 其剖析程序如下：
-
- ```
  void Number()
  {
    if (符記為數字)
      取得下一符記;
    else
      Error(22);
  }
  ```
  ```

6.9 剖析樹

- 上一個小節的每一條語法規則都可建立一棵剖析樹（parsing tree），例如以最初的節點（node）樹根（root）來說，它相當的方法為 Program()，相當第一條語法規則，它的剖析樹如下圖所示。



f := c*9/5+32; 它的剖析樹如下圖所示。



6.10 plone 語法分析程式

- plone 語法分析程式 parser.c 是以 C 語言撰寫的一個剖析程式，完全依據上一小節的剖析程序撰寫成的程式。
- struct symbolTag *token;
- int errorCount = 0;
- 符記 token 是一個 symbolTag 結構的指標，errorCount 是語法錯誤計數。

```
void Error(int  n)
{
  int  j;
  printf("****");
  for (j=0; j<token->right; j++) printf(" ");
  printf("^%d  %s\n     %s\n",
    n, errmsgs[n], symbolToString(token));
  errorCount++;
}
```

- Error() 函式負責報告第 n 編號的語法錯誤，輸出錯誤編號、錯誤訊息、符記內容，每報告一次錯誤就將 errorCount 值增一。

- 剖析程式的主體架構如下：
- ```
  int main(int argc, char *argv[])
  {
    FILE *f=fopen(argv[1], "r");
    scanner(f);
    token = nextToken();
    Program();
    fclose(f);
    printf("\n  Plone compile completed. "
      "\n  Error count : %d\n", errorCount);
    return 0;
  }
  ```
- 執行這個程式時要從命令列輸入一個 plone 的原始程式檔名，如下例：
- ```
  parser.exe test61.pl <Enter>
  ```

- 執行這個程式時要從命令列輸入一個 plone 的原始程式檔名，如下例：
- ***parser.exe test61.pl <Enter>***
- 剖析程式名稱「parser.exe」儲存於 main() 主函式的 argv[] 字串型態的參數陣列裡的第 0 個元素，即 argv[0]。
- 參數「test61.pl」存於 argv[1] 參數陣列裡的第 1 個元素。
- ***FILE *f=fopen(argv[1], "r");***
- 開啟 argv[1] 輸入檔，例如 test61.pl，它是以 plone 語法撰寫的原始程式檔，作為本剖析程式 parser.exe 的輸入檔，檔案指標名為 f。
- ***scanner(f);***
- 將輸入檔案指標 f 傳入掃描函式 scanner() 裡，當掃描函式的輸入檔。

- token = nextToken();
- Program();
- 從掃描函式 nextToken() 取得下一個符記 token 後就從 plone 語法的樹根 <Program> 非終端符號開始剖析。
- fclose(f);
- printf("\n Plone compile completed. "
- "\n Error count : %d\n", errorCount);
- return 0;
- 剖析完畢，關閉輸入檔，輸出剖析的結果，結束程式。
- 首先建立一個從命令列鍵入 plone 原始程式檔名 argv[1] 的檔案指標 f，提供給掃描程式當它的輸入檔，取得第一個符記 token，然後呼叫剖析樹的樹根節點 <Program> 相對應的處理函式 Program()，逐步下降分析，一直到終端符號的樹葉節點為止，或遇到無法解決的錯誤才停止。

6.11 測試程式

- 依 plone 語法撰寫的原始程式 test61.pl, 其語法正確，剖析結果沒有錯誤。
- 【編譯】
- **C:\plone\ch06> gcc parser.c -o parser.exe <Enter>**
- 【執行】
- C:\plone\ch06>parser test61.pl <Enter>

- 下列的 test62.pl 則刻意撰寫些錯誤的敘述以測試語法分析程式的剖析能力，從執行的結果可以看出它能夠偵測出錯誤的敘述。
- 　　WHILE x != 0 DO
- 但偵測出錯誤的符記「!」之後就不再剖析，它罷工了，後面還有語法錯誤的敘述。
- 　　y := x^3+6;
- 其中「^」是一個錯誤的符記，但語法分析程式卻看不見，如此的剖析程式您滿意嗎？當然不滿意，但應該如何處理呢？請看下一章語法錯誤之復原處理。

```
C:\plone\ch06>parser test62.pl <Enter>
   1 PROGRAM test62;
   2 CONST
   3   msg1=" x=",
   4   msg2=" y=";
   5 VAR
   6   x, y;
   7 BEGIN
   8   x := 3;
   9   WHILE x!=0 DO
****          ^20  關係運算子錯誤
     sym=1 left=9 right=10 value="!"
****          ^17  遺漏(
     sym=1 left=9 right=10 value="!"
****          ^15  WHILE敘述錯誤,遺漏DO
     sym=1 left=9 right=10 value="!"
****          ^11  遺漏END
     sym=1 left=9 right=10 value="!"
****          ^0  必須跟著句點.
     sym=1 left=9 right=10 value="!"

  Plone compile completed.
  Error count : 5
```