第七章 語法錯誤復原處理

- 上一章在做 plone 程式語言語法分析，所使用的是由上而下的剖析，從樹根的非終端符號一直剖析到樹葉的終端符號，只是很單純的剖析語法，且假設程式設計者所撰寫出來的原始程式都是正確的，對於語法錯誤時著墨不多，本章要補充在這方面的問題。

7.1 起始符號與跟隨符號

- plone 程式語言語法，均合乎起始符號及跟隨符號規定。起始符號規定是說對於每一個語法結構分叉點上的起始符號必須完全不同。跟隨符號規定要求所有可能為該語法結構之跟隨者的起始符號集合，必須和該語法結構的起始符號集合互斥（不一樣）。由此我們知道整個語法圖中的起始符號集合和跟隨符號集合，對於以後的工作是十分有用的，所以決定先求得 plone 語法中非終端符號的起始符號集合及跟隨符號集合。

7.2 <Block>的起始符號與跟隨符號

- plone 程式語言語法中有關 <Block> 非終端符號的語法有三條相關的規則，如下：
- **1. <Program> ::= <ProgramHead><Block>.**
- **3. <Block> ::= [<ConstDeclaration>]**
- **[<VarDeclaration>]**
- **[<ProcDeclaration>]**
- **<CompoundStatement>**
- **6. <ProcDeclaration> ::= {PROCEDURE**
- **<Identifier>;<Block>;}**
- 依據跟隨符號規定，<Block> 非終端符號的跟隨符號為句點「.」及分號「;」，句點在我們的 sym.h 整數常數定義為 symPERIOD，簡稱為PERIOD。分號整數常數定義為 symSEMI。簡稱為SEMI。非終端符號 <Block> 的跟隨符號集合如下：
- **{ PERIOD SEMI }**

<Block> 非終端符號的起始符號

- 依據起始符號規定，<Block> 非終端符號的起始符號為各非終端符號的起始符號，非終端符號 <ConstDeclaration> 的起始符號為 CONST，在 sym.h 整數常數定義為 ***symCONST***，<VarDeclaration> 非終端符號的起始符號為 VAR，其整數常數定義為 ***symVAR***，<ProcDeclaration> 非終端符號的起始符號為 PROCEDURE，其整數常數定義為 ***symPROCEDURE***，非終端符號 <CompoundStatement> 的起始符號為 BEGIN，其整數常數定義為 ***symBEGIN***。
- 因此 <Block> 非終端符號的起始符號集合如下：
- **{ CONST VAR PROCEDURE BEGIN }**

- **非終端符號 <Block> 的起始符號集合的 *程式碼* 如下：**
- char block[symSYMMAX];
- block[symCONST] = 1;
- block[symVAR] = 1;
- block[symPROCEDURE] = 1;
- block[symBEGIN] = 1;
- **跟隨符號集合的 *程式碼* 如下：**
- char block[symSYMMAX];
- block[symPERIOD] = 1;
- block[symSEMI] = 1;

7.3 <CompoundStatement>的起始符號與跟隨符號

- **plone 程式語言語法中有關 <CompoundStatement> 非終端符號的語法有下列三條：**
-
- 3. <Block> ::= [<ConstDeclaration>] [<VarDeclaration>]
- [<ProcDeclaration>] <CompoundStatement>
- 7. <Statement> ::= [<AssignmentStatement> |
- <CallStatement> | <CompoundStatement> |
- <IfStatement> | <WhileStatement> |
- <ReadStatement> | <WriteStatement>]
- 10. <CompoundStatement> ::=
- BEGIN <Statement> { ; <Statement> } END

起始符號與跟隨符號

- <CompoundStatement> 的起始符號如下：
- 　　{ BEGIN }
- <CompoundStatement> 的跟隨符號為 <Block> 的跟隨符號集合以及 <Statement> 的跟隨符號集合的聯集，也就是 { PERIOD SEMI END } 集合了，與 <Statement> 相同。
- <CompoundStatement> 的跟隨符號如下：
- 　　{ PERIOD SEMI END }

7.4 <Statement>的起始符號與跟隨符號

- <Statement> 的起始符號集合為各非終端符號起始符號集合的聯集，如下：
- { CALL BEGIN IF WHILE
- READ WRITE IDENTIFIER }
- <Statement> 的跟隨符號集合為各非終端符號跟隨符號集合的聯集，如下：
- { SEMI PERIOD END }

7.5 起始符號與跟隨符號表

非終端符號	起始符號集合	跟隨符號集合
<Block>	{CONST VAR PROCEDURE BEGIN}	{PERIOD SEMI}
<CompoundStatement>	{BEGIN}	{PERIOD SEMI END}
<Statement>	{IDENTIFER CALL BEGIN IF WHILE READ WRITE}	{PERIOD SEMI END}
<Condition>	{PLUS MINUS LPAREN IDENTIFIER NUMBER}	{THEN DO}
<Expression>	{PLUS MINUS LPAREN IDENTIFER NUMBER}	{PERIOD SEMI RPAREN END THEN DO}
<Term>	{IDENTIFER NUMBER LPAREN}	{PERIOD SEMI LPAREN PLUS MINUS END THEN DO}
<Factor>	{IDENTIFER NUMBER LPAREN}	{PERIOD SEMI LPAREN PLUS MINUS MUL DIV END THEN DO}

7.6 起始與跟隨符號程式碼

- <Statement> 非終端符號的起始符號集合程式碼如下：
- char statement[symSYMMAX];
- statement[symIDENTIFIER] = 1;
- statement[symCALL] = 1;
- statement[symBEGIN] = 1;
- statement[symIF] = 1;
- statement[symWHILE] = 1;
- statement[symREAD] = 1;
- statement[symWRITE] = 1;
- <Statement> 非終端符號的跟隨符號集合程式碼如下：
- char statement[symSYMMAX];
- statement[symPERIOD] = 1;
- statement[symSEMI] = 1;
- statement[symEND] = 1;

- <Condition> 非終端符號的起始符號集合程式碼如下：
-
- char condition[symSYMMAX];
- condition[symPLUS] = 1;
- condition[symMINUS] = 1;
- condition[symLPAREN] = 1;
- condition[symIDENTIFIER] = 1;
- condition[symNUMBER] = 1;
-
- <Condition> 非終端符號的跟隨符號集合程式碼如下：
-
- char condition[symSYMMAX];
- condition[symTHEN] = 1;
- condition[symDO] = 1;

- <Expression> 非終端符號的起始符號集合程式碼如下：
-
- char expression[symSYMMAX];
- expression[symPLUS] = 1;
- expression[symMINUS] = 1;
- expression[symLPAREN] = 1;
- expression[symIDENTIFIER] = 1;
- expression[symNUMBER] = 1;
-
- <Expression> 非終端符號的跟隨符號集合程式碼如下：
-
- char expression[symSYMMAX];
- expression[symPERIOD] = 1;
- expression[symSEMI] = 1;
- expression[symRPAREN] = 1;
- expression[symEND] = 1;
- expression[symTHEN] = 1;
- expression[symDO] = 1;

- <Term> 非終端符號的起始符號集合程式碼如下：
-
- char term[symSYMMAX];
- term[symIDENTIFIER] = 1;
- term[symNUMBER] = 1;
- term[symLPAREN] = 1;
-
- <Term> 非終端符號的跟隨符號集合程式碼如下：
-
- char term[symSYMMAX];
- term[symPERIOD] = 1;
- term[symSEMI] = 1;
- term[symRPAREN] = 1;
- term[symPLUS] = 1;
- term[symMINUS] = 1;
- term[symEND] = 1;
- term[symTHEN] = 1;
- term[symDO] = 1;

- <Factor> 非終端符號的起始符號集合程式碼如下：
- char factor[symSYMMAX];
- factor[symIDENTIFIER] = 1;
- factor[symNUMBER] = 1;
- factor[symLPAREN] = 1;
- <Factor> 非終端符號的跟隨符號集合程式碼如下：
- char factor[symSYMMAX];
- factor[symPERIOD] = 1;
- factor[symSEMI] = 1;
- factor[symRPAREN] = 1;
- factor[symPLUS] = 1;
- factor[symMINUS] = 1;
- factor[symMUL] = 1;
- factor[symDIV] = 1;
- factor[symEND] = 1;
- factor[symTHEN] = 1;
- factor[symDO] = 1;

7.7 測試起始符號與跟隨符號

```
/*************** followsym.h ***************/
#include "sym.h"
char factor[symSYMMAX];
char term[symSYMMAX];
char expression[symSYMMAX];
char condition[symSYMMAX];
char statement[symSYMMAX];
char block[symSYMMAX];
void followsyminit()
{
  factor[symPERIOD] = 1;
  factor[symSEMI] = 1;
  //略
  block[symSEMI] = 1;
}
```

- 程式 followsymtest.c 用於測試 followsym.h 表頭檔是否正確。
- 表頭檔 synmane.h 定義符記整數常數的名稱，例如 symEND 符記整數常數，其相對應名稱字串為 "END"。表頭檔 followsym.h 定義非終端符號的跟隨符號集合，集合以 C 語言字元陣列表示。symnameinit() 函式初始化符記整數常數的名稱，followsyminit() 函式初始化非終端符號的跟隨符號集合。
- 測試程式以非終端符號 <Statement> 為例，輸出該非終端符號的跟隨符號集合。

```
/******************** followsymtest.c ******************/
#include <stdio.h>
#include "symname.h"
#include "followsym.h"
int main()
{
  int i;
  symnameinit();
  followsyminit();
  for (i=0; i<symSYMMAX; i++)
  {
    if (statement[i] != 0)
      printf("statement[%s]=%d\n",names[i],statement[i]);
  }
  return 0;
}
```

7.8 語法錯誤之復原處理

- 原始程式常常含有語法錯誤的敘述，若剖析工作因而停止，對於實用的編譯器程式而言，這不是理想的處理方式，此時編譯器程式應該能產生一個合適的錯誤診斷報告，並能繼續進行剖析工作。常用的方法有兩種:
-
- ***1）. 列舉各種錯誤，並將診斷到的錯誤報告***
- ***使用者。如錯誤訊息表。***
- ***2）. 跳過輸入符記，直到能夠繼續剖析為止。***

關鍵字與飛越規則

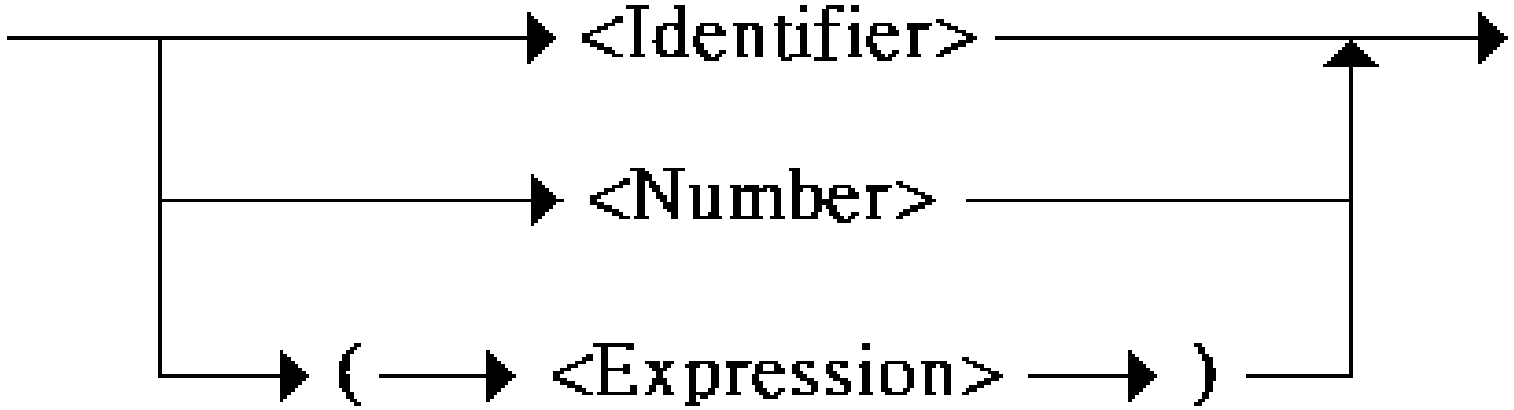
- ***關鍵字規則（keyword rule）***
- 　每一敘述均以關鍵字起頭。
- ***飛越規則（skip rule）***
- 　當偵測到語法錯誤時，則飛越緊跟著的符記，
- 　直到跟隨符號出現為止。
- **依關鍵字規則，每一敘述均以關鍵字起頭，這些關鍵字若設定為跟隨符號，則依飛越規則，當語法錯誤時，會停在關鍵字處而繼續剖析。**

語法錯誤之復原處理程序

```
void  skip (跟隨符號，錯誤訊息代號n)
{
    if  (符記不是跟隨符號)
    {
        Error (n);
        while  (符記不是跟隨符號)
            取得下一符記;
    }
}
```

- 程序 skip() 審查正在剖析之符記，若在「跟隨符號」中則為正確，否則為代號「n」之語法錯誤，若該符記也不屬於跟隨符號，則越過該符記，然後審查下一符記，一直到目前符記屬於跟隨符號時才停止。

7.8.1 <Factor> 之復原處理



```
void Factor()
{
  if (token->sym == symIDENTIFIER)        /*識別字*/
    Identifier();
  else if (token->sym == symNUMBER)   /*阿拉伯數字*/
    Number();
  else if (token->sym == symLPAREN)       /*左括號*/
  {
    token = nextToken();              /*取得下一符記*/
    Expression();                     /*運算式*/
    if (token->sym == symRPAREN)          /*右括號*/
      token = nextToken();            /*取得下一符記*/
    else
    {
      Error(18);                        /*報告錯誤*/
      skip(factor, 23);           /*飛越至跟隨符號*/
    }
  }
}
```

- 以上圖中之 <Factor> 語法剖析為例，首先檢驗目前之符記是否為起始符號集合中之一個，<Factor> 的起始符號集合如下：
- ***{識別字 數字 左括號}***
- 首先檢驗目前之符記是否為「***識別字***」，若是則呼叫識別字處理方法 Identifier()。
- 接著檢驗符記是否為「***數字***」，是則呼叫數字處理方法 Number()。
- 最後檢驗符記是否為「***左括號***」，是則取得下一符記後呼叫運算式處理方法 Expresion()，接著檢驗目前符記是否為「右括號」，若是則取得下一符記，完全合乎語法。

- 但若一連串的辨認工作下來，其間發生錯誤，則最後辨認是否為右括號時一定產生錯誤，馬上報告第 18 號的錯誤，然後呢？這時候就要執行復原處理，讓語法分析不致中斷。
- ***skip (factor, 23);***
- 這時要呼叫 skip() 方法，設定跟隨符號集合為 <Factor> 非終端符號的跟隨符號集合 factor，該字元陣列定義於 followsym.h 表頭檔裡頭。若目前符記不屬於 factor 集合中的一個元素，則產生一個第 23 號的錯誤，再繼續剖析。
- <Factor> 之跟隨符號集合 factor 定義於followsym.h 表頭檔裡，其值如下：
- **{ PERIOD SEMI RPAREN PLUS MINUS MUL DIV END THEN DO }**

7.8.2 <IfStatement> 之復原處理

再以 <IfStatement> 非終端符號語法錯誤時的復原處理說明跟隨符號的使用。**<IfStatement> 跟隨符號與 <Statement> 跟隨符號相同，其集合如下：**

{ PERIOD SEMI END }

非終端符號 <IfStatement> 的語法如下圖所示。

非終端符號 <IfStatement> 語法轉換為程式碼如下：

```
void IfStatement()
{
  if (strcmp(token->value,"IF")==0)
  {
    token = nextToken();              /*下一符記*/
    Condition();                      /*條件*/
    if (strcmp(token->value, "THEN")==0)
    {
      token = nextToken();            /*下一符記*/
      Statement();                    /*敘述*/
    }
    else
    {
      Error(13);                      /*報告錯誤*/
      skip(statement, 23);            /*飛越至跟隨符號*/
    }
  }
  else
  {
    Error(12);                        /*報告錯誤*/
    skip(statement, 23);              /*飛越至跟隨符號*/
  }
}
```

- 首先檢驗目前取得的符記是否為保留字 IF，若是則繼續剖析，否則馬上報告第 12 號的語法錯誤，並跳至下一個敘述繼續剖析。辨認 IF 保留字後，取得下一個符記，呼叫 Condition() 方法辨認並處理條件，不管有沒有問題，接下來辨認是否為 THEN 保留字，若是則再取得下一個符記，呼叫 Statement() 敘述，若都沒有問題就可判斷 IF 敘述語法正確，繼續剖析下一個敘述。
- 若一連串的辨認，其中某一個環節發生問題則執行下列的復原處理：
- ***skip (statement, 23) ;***
- 這時要呼叫 skip() 方法，設定跟隨符號集合為 <Statement> 非終端符號的跟隨符號集合 statement，該集合定義於 followsym.h 表頭檔裡。若目前符記不屬於 statement 集合中的一個元素，則產生第 23 號的錯誤，再繼續剖析。

7.9 剖析程式

- 剖析程式是由語法中非終端符號轉換成程式碼的方法所組成，方法名稱採用與非終端符號相同的名稱有助於對剖析程式的了解，也讓程式的撰寫更為簡潔。您若對程式的結構有充分的了解，要修改它就容易了。

- 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);
    followsyminit();
    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 test71.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 test71.pl <Enter>***
- 程式名稱「parser.exe」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「test71.pl」存於 argv[1]。
- ***FILE *f=fopen(argv[1], "r");***
- 開啟 argv[1] 原始程式檔名為本程式的輸入檔，檔案指標名為 f。
- ***scanner(f);***
- 將檔案指標 f 傳入掃描函式 scanner() 裡，當掃描函式的輸入檔。

- ***followsyminit();***
- 將非終端符號的跟隨符號初始化。
- ***token = nextToken();***
- ***Program();***
- 從掃描函式取得下一個符記 token 後就從樹根 <Program> 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。

7.10 測試程式

- 下列兩個測試程式 test71.pl 及 test72.pl 都很簡單，並不包含程序的呼叫。程式 test71.pl 是正確的，而 test72.pl 則刻意製造一些錯誤，以便測試剖析程式是否可偵測出來。

```
C:\plone\ch07> parser test71.pl <Enter>
   1 PROGRAM test71;
   2 CONST
   3   msg1=" x=",
   4   msg2=" y=";
   5 VAR
   6   x, y;
   7 BEGIN
   8   x := 3;
   9   WHILE x>0 DO
  10     BEGIN
  11       y := x*3+6;
  12       WRITE(msg1,x);
  13       WRITE(msg2,y);
  14       x := x-1;
  15     END;
  16 END.

  Plone compile completed.
  Error count : 0
```

```
C:\plone\ch07> parser test72.pl <Enter>
  1 PROGRAM test72;
  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
****          ^23  飛越至下一個敘述
 10    BEGIN
 11      y := x^3+6;
****               ^15  WHILE敘述錯誤,遺漏DO
 12      WRITE(msg1,x);
 13      WRITE(msg2,y);
 14      x := x-1;
 15    END;
****       ^0  必須跟著句點.

  Plone compile completed.
  Error count : 4
```

- 上列兩個測試程式 test71.pl 及 test72.pl 都很簡單，並不包含程序的呼叫。
- 現在我們要設計一個含有程序呼叫的 test73.pl 及 test74.pl 測試程式。程式 test73.pl 是正確的，而 test74.pl 則刻意製造一些錯誤，以便測試剖析程式是否可偵測出來。
-
- 測試程式 test73.pl 裡，從主程序呼叫 gcd 程序，然後從 gcd 程序分別呼叫 xDIVy 及 yDIVx 程序。x 設定為常數值 12，y 設定為常數值 9，呼叫 gcd 求出 x 及 y 的最大公約數 x=3 後輸出。