第十章 簡單的建碼程式

- 在語法分析的過程當中，程式設計者所撰寫的原始程式被辨認為所使用程式語言的語法，這個語法以由上而下剖析樹方式呈現，語彙分析、語法分析、語意分析等在前面幾章已經討論過了，本章要探討***如何建碼***。
- 為了突顯建碼的一些基本技巧，特地將 Plone 語言簡化，拿掉非終端符號 <ConstDeclaration>、<ProcDeclaration> 以及 <Statement> 裡面的非終端符號 <CompoundStatement>、<CallStatement>、<ReadStatement>、<WhileStatement>，***簡化後的 Plone 程式語言稱為 simple Plone。***

10.1 simple Plone 程式語言語法

- 1. <Program> ::= <ProgramHead><Block>.
- 2. <ProgramHead> ::= PROGRAM <Identifier>;
- 3. <Block> ::= [<VarDeclaration>]
- <CompoundStatement>
- 5. <VarDeclaration> ::=
- VAR <IdentifierList> ;
- 7. <Statement> ::= [<AssignmentStatement> |
- <IfStatement> | <WriteStatement>]
- 8. <AssignmentStatement> ::=
- <Identifier> := <Expression>
- 10. <CompoundStatement> ::=
- BEGIN <Statement>{;<Statement>} END

simple Plone 程式語言語法

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

根據簡化後的 Plone 語法建碼

- 根據這個簡化過後的 Plone 語法建碼，所建的為 NASM 組合語言指令碼。
- 很多教學編譯器所建的都屬***虛擬碼***（pseudo code），針對一個假想的電腦建碼，然後對這個虛擬碼解釋並執行，這樣您要了解這個虛擬碼的指令，您還要知道如何解譯並執行，其優點就是這些虛擬碼與機器無關，但如何解譯並執行還是與機器有關。
- ***本書採用比較簡潔的方式，就是直接建成 NASM 組合語言的指令***，該組合語言是跨平台的，也就是說您可以在 Windows 系統執行，也可以在其他的作業系統執行，例如 Unix、Linux 等，其缺點是 NASM 是針對 Intel 80X86 處理器而設計，對於其他的處理器就無法執行了，這是我們選用 NASM 的代價了。有關 NASM 請參考文魁資訊出版的「組合語言程式設計」（XP6237）一書。

10.2 建碼主程式

```
int main(int argc, char *argv[ ])
{
  FILE *f=fopen(argv[1], "r");
  scanner(f);
  followsyminit();
  token = nextToken();
  Program();
  fprintf(outfile, "\tMOV\tAX, 4C00H\n"
                "\tINT\t21H\n");
  printf("\n  Plone compile completed. "
    "\n  Error count : %d\n", errorCount);
  //略
}
```

建碼主程式

- ***FILE *f = fopen(argv[1], "r");***
- ***scanner(f);***
- 從命令列取得輸入原始程式檔名 argv[1] 後建立一個檔案指標 f，傳給掃描程式 scanner(f) 當它的輸入檔。
- ***followsyminit();***
- 初始化各個非終端符號的跟隨符號集合。
- ***token = nextToken();***
- ***Program();***
- 從掃描程式取得下一個符記結構指標 token，呼叫 Program() 函式，就開始由上而下的剖析，一層一層往下剖析至樹葉為止。

- fprintf(outfile, "\tMOV\tAX, 4C00H\n"
- "\tINT\t21H\n");
-
- 建立 NASM 程式返回作業系統的指令：
-
- MOV AX, 4C00H
- INT 21H
-
- 下列的程式碼顯示編譯完成，輸出錯誤筆數。
-
- printf("\n Plone compile completed. "
- "\n Error count : %d\n", errorCount);

10.3 建碼輸出檔案

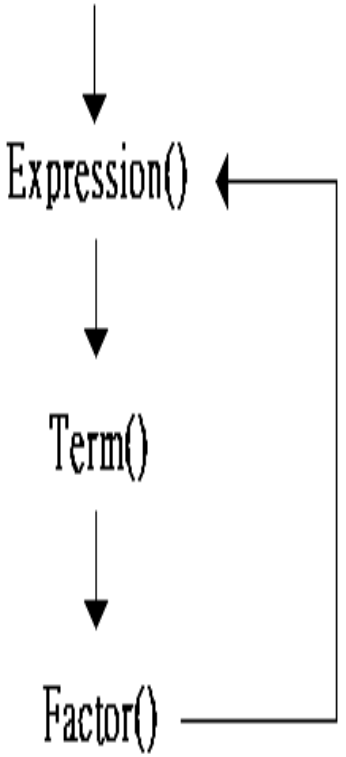
- 建碼程式的輸出是一個含有 NASM 組合語言指令的本文檔案。
- char outname[IDLEN];
- FILE *outfile;
- 宣告一個輸出檔名為 outname 的檔案指標 outfile。
- 在 ProgramHead() 函式裡等到辨認出程式識別字名稱時，立即定義此檔案，以程式名稱附加「.asm」副檔名作為輸出檔名。
- strcpy(outname, token->value); /*拷貝程式識別字*/
- strcat(outname, ".asm"); /*附加.asm副檔名*/
- outfile = fopen(outname, "w"); /*開啟為輸出檔*/

- 然後輸出表頭，樣式如下：
-
- ;********** **程式名稱**.asm **********
- ;
- ORG 100H
- JMP _start1
- _intstr DB ' ','$'
- _buf TIMES 256 DB ' '
- DB 13,10,'$'
- %include "itostr.mac"
- %include "newline.mac"

- 當辨認到分號符記時表示 ProgramHead() 已經結束了，這時要插入一些相關的指令以及標籤（label），其 NASM 程式碼如下：
- ```
  _intstr   DB       '     ','$'
  _buf      TIMES    256 DB ' '
            DB       13,10,'$'
  %include           "itostr.mac"
  %include           "newline.mac"
  ```
- 請注意，以底線開頭的識別字是 plone 編譯器所使用的，您的應用程式最好避免使用。「 intstr」表示整數轉換成字串後所儲存的記憶體位址，DB 為 NASM 組合語言指令，表示定義一個位元組，13 為返回字元的 ASCII 碼值，也可以寫成十六進位表示法 0x0D，簡稱 CR。10 為新列的 ASCII 碼值，也可以寫成十六進位表示法 0x0A，簡稱 LF。 字元 '$' 為 NASM 表示字串結束符號。
  ```

10.4 變數宣告建碼

- 在執行 VarDeclaration() 方法時，當辨認到第一個變數宣告，或往後的第二個、第三個等等變數名稱時，執行下列的建碼動作：
- sprintf(buf, "%s\tDW\t0\n", token->value);
- fprintf(outfile, buf);
- 輸出如下的樣子：
- 變數名稱1 DW 0
- 變數名稱2 DW 0
- ...
- 變數名稱n DW 0
- DW 為 NASM 組合語言指令，表示定義一個字組，該變數初值為 0。

10.5 區塊 Block 建碼

- 在區塊 Block 裡唯一的建碼就是建立一個標籤「_startX:」表示變數宣告之後，複合敘述開始之前的一個位置，其程式碼如下：
- ```
  void Block()
  {
    ++level;
    if (strcmp(token->value, "VAR")==0)
      VarDeclaration();
      sprintf(buf, "_start%d:\n", labelCount);
      fprintf(outfile, buf);
    CompoundStatement();
    --level;
  }
  ```
- 這個標籤「_startX:」是本區塊程式碼開始執行所跳至的第一個標籤，從這裡開始執行您的程式碼指令。X 為某一數字。
  ```

10.6 運算式Expression建碼

- 有關運算式建碼，首先來看一看它的語法是怎樣規定的，請看語法規則第 17-19 條，如下：
- 17. <Expression> ::= [+|-]<Term>{\+|-\<Term>}
- 18. <Term> :: = <Factor>{*|/\ <Factor>}
- 19. <Factor> ::= <Identifier>|<Number>|
- (<Expression>)
- 運算式是項次 <Term> 加或減另一個項次 <Term>，一個項次是一個因子 <Factor> 乘或除另一個因子 <Factor>，一個因子是一個識別字、數字、或小括號內的一個運算式，也就是說一個運算式裡可以包含另一個運算式，其方法呼叫的流程如下圖所示。



- 我們先來看看運算式如何建碼。若第一個符記為加號或減號，則取得下一個符記後呼叫 Term() 方法，第一個符記為加號則不理會，第一個符記為減號，那麼要將 Term() 計值的結果改變符號，正數變負數，負數變正數。輸出指令如下：
- POP AX
- NEG AX
- PUSH AX

```
    if (token->sym == symPLUS ||
        token->sym == symMINUS)       /*檢查加號或減號*/
    {
      int operator = token->sym;          /*儲存運算子*/
      token = nextToken();               /*取得下一符記*/
      Term();
      if (operator == symMINUS)       /*運算子是減號？*/
      {
        fprintf(outfile,"₩tPOP₩tAX₩n"           /*疊出*/
                        "₩tNEG₩tAX₩n"      /*改變正負號*/
                        "₩tPUSH₩tAX₩n");        /*疊入*/
      }
    }
    else
      Term();
```

- 若第一個符記不是加號，也不是減號，則直接呼叫 Term() 計值，其程式碼如上所述。

- 後面的項次計算其運算子不是加號就是減號，計算 Term() 值後判斷加法或減法。
- 若是執行加法則產生下列的指令：
-
- POP BX /*疊出第二個運算元至BX*/
- POP AX /*疊出第一個運算元至AX*/
- ADD AX, BX /*AX=AX+BX*/
- PUSH AX /*疊入AX*/
-

- 若是執行減法則產生下列的指令:
- POP BX /*疊出第二個運算元至BX*/
- POP AX /*疊出第一個運算元至AX*/
- SUB AX, BX /*AX=AX-BX*/
- PUSH AX /*疊入AX*/

10.7 項次Term建碼

- 項次 <Term> 是將兩個因子 <Factor> 相乘或相除。若是相乘則產生下列的指令：
-
- POP BX /*疊出第二個運算元至BX*/
- POP AX /*疊出第一個運算元至AX*/
- MUL BX /*DX:AX=AX*BX*/
- PUSH AX /*疊入AX*/
-

- 若是相除則產生下列的指令：
- POP BX /*疊出第二個運算元至BX*/
- MOV DX, 0 /*DX=0*/
- POP AX /*疊出第一個運算元至AX*/
- DIV BX /*DX:AX/BX，商AX餘數DX*/
- PUSH AX /*疊入AX*/

10.8 因子Factor建碼

- 若因子 <Factor> 屬於識別字則產生下列的指令：
- ***PUSH　WORD [識別字]***
- 將識別字位址的內含值疊入堆疊頂端。
-
- 若因子 <Factor> 屬於數字則產生下列的指令：
- ***PUSH　識別字字串***
- 將識別字字串疊入堆疊頂端。

- 若第一個符記為左括號，則呼叫運算式後再判斷下一個符記是否為右括號。其程式碼如上所述。

10.9 指定敘述建碼

- 指定敘述的語法如下：
- 8. <AssignmentStatement> ::=
- <Identifier> := <Expression>
- 識別字定義為指定運算式的值。產生下列的指令：
- ***POP AX***
- ***MOV [識別字] , AX***
- 從工作堆疊頂端疊出頂端元素至暫存器 AX，再搬移至識別字位址當它的內含值。

10.10 條件建碼

- 兩個運算式的關係有六種，小於「<」、小於等於「<=」、等於「=」、不等於「<>」、大於「>」、大於等於「>=」，其語法規則如下：
- 16. <Condition> ::= <Expression>
- \<|<=|=|<>|>|>=\ <Expression>
- 運算式計值後都疊入堆疊頂端，因此堆疊頂端的兩個值就是要比較的兩個運算元，頂端的為右運算元，次頂端的為左運算元，建立下列的指令：
- ***POP BX /*疊出第二個運算元至BX*/***
- ***POP AX /*疊出第一個運算元至AX*/***
- ***CMP AX, BX /*比較左右兩運算元之值，*/***
- ***/*比較結果存於旗標暫存器*/***
- 比較後產生六種結果，分別跳至指定的標籤。JE 指令表示比較結果相等時跳躍至指定標籤。JNE 表示不相等、JL 表示小於、JLE 表示小於等於、JG 表示大於、JGE 表示大於等於。

- 例如下列的 WHILE 敘述：
- ***WHILE <Condition> DO***
- ***敘述開始標籤：***
- ***<Statement>***
- ***敘述結束標籤：***
- 當條件 <Condition> 比較結果產生六種跳躍，都是條件成立時才跳躍的，以本例來說，條件成立時要躍過 DO，跳躍至「敘述開始標籤：」處執行敘述 <Statement>，條件不成立時要躍過 <Statement> 敘述，跳躍至「敘述結束標籤：」處繼續執行下一個敘述。

- 又如下列的 IF 敘述：
- ***IF <Condition> THEN***
- ***敘述開始標籤：***
- ***<Statement>***
- ***敘述結束標籤：***
- 當條件 <Condition> 比較的結果產生六種跳躍，都是條件成立時才跳躍的，本例條件成立時要躍過 THEN，跳躍至「敘述開始標籤：」處執行敘述 <Statement>，條件不成立時要躍過 <Statement> 敘述，跳躍至「敘述結束標籤：」處繼續執行下一個敘述。

10.11 IF敘述建碼

- IF 敘述的語法如下，判斷條件 <Condition> 是否成立，若成立則執行 THEN 後面的敘述。
- 11. <IfStatement> ::= IF <Condition> THEN <Statement>
- IF 敘述的邏輯如下：
- ***<Condition>***
- ***依條件跳至head標籤（這個指令在<Condition>內建立）***
- ***JMP tail***
- ***head:***
- ***<Statement>***
- ***tail:***
- 當條件成立時跳至「head:」標籤處執行 <Statement> 敘述，當條件不成立時跳至「tail:」標籤處執行下一個敘述。

10.12 WRITE敘述建碼

- WRITE 敘述的語法如下，小括號內為以逗號隔開的識別字，這些識別字的值會顯示在螢幕。
- 14. <WriteStatement> ::= WRITE (<IdentifierList>)
- 15 <IdentifierList> ::= <Identifier>{,<Identifier>}
- 當辨認到第一個識別字時，產生下列的指令：

```
itostr  識別字 , _intstr, '$' /*整數轉成字串_intstr*/
MOV     DX, _intstr          /*字串位址存入DX暫存器*/
MOV     AH, 09H              /*顯示DX位址字串功能*/
INT     21H                  /*顯示字串於螢幕*/
newline                      /*游標移至下一列*/
```

- 將識別字的整數值透過 itostr 巨集轉換成字串，存入「 intstr」位址，將該位址存入 DX 暫存器，呼叫 21H 插斷的 09H 功能將 DX 所指位址的字串顯示在螢幕上，newline 巨集執行換新列的工作。

10.13 simple plone 程式碼

- simple plone 程式語言的建碼程式 simpleplone.c 完全依據語法剖析程序撰寫成的程式，再適當的辨認符記，成功時插入 NASM 組合語言的程式碼。
- 【simpleplone.c】
- /****************** simpleplone.c ******************/
- #include <stdio.h>
- #include <stdlib.h>
- /*
- ** 自訂表頭檔
- */
- #include "scanner.h"
- #include "resword.h"
- #include "err.h"
- #include "followsym.h"
- #include "idobj.h"
- #include "idobjstack.h"
- //略

10.14 測試程式

- 測試程式 test1001.pl 如下：
- PROGRAM test1001;
- VAR a, b, c;
- BEGIN
- a := 5;
- b := 2;
- IF a>b THEN c:=a-b;
- WRITE(c);
- END.
- 編譯時只要在命令列輸入 test1001.pl 原始程式檔後按 Enter 鍵就可以了，若想了解識別字結構堆疊的狀況可在命令列再輸入一個引數就行。

- C:\plone\ch10> simpleplone test1001.pl 1 <Enter>
- 1 PROGRAM test1001;
- 2 VAR a, b, c;
- 3 BEGIN
- 4 a := 5;
- 5 b := 2;
- 6 IF a>b THEN c:=a-b;
- 7 WRITE(c);
- 8 END.
-
- Plone compile completed.
- Error count : 0
-
- 識別字結構堆疊內容如下：
- name:"a" sym:2 attr:31 level:0
- name:"b" sym:2 attr:31 level:0
- name:"c" sym:2 attr:31 level:0

- 若編譯完成，沒有錯誤，那會自動產生一個 test1001.bat 批次檔。您必須在「Command Prompt」命令提示視窗執行此批次檔。從「命令提示字元」視窗轉至命令提示視窗只需執行 COMMAND.COM 命令就可以，不過建議您各開一個視窗較為方便。
-
- **C:\PLONE\CH10> test1001.bat <Enter>**
- **C:\plone\ch10>nasmw test1001.asm -o test1001.com**
- **C:\plone\ch10>test1001.com**
- **3**

10.15 建碼指令檢驗

- 測試程式 test1001.pl 經編譯後產生 test1001.asm 組合語言指令檔，再經 NASM 組譯程式 nasmw.exe 組譯後產生 test1001.com 可執行檔，執行後可得結果。
- 現在讓我們來檢驗一下 plone 原始程式每一個敘述建碼的情形。
- ● 原始程式： PROGRAM test1001
- 建碼如下：
-
- ;*************** test1001.asm ****************
- ;
- ORG 100H
- JMP _start1

- ● 原始程式： VAR a, b, c;
- 建碼如下：
- a DW 0
- b DW 0
- c DW 0
-
- ● 原始程式： a:=5;
- 建碼如下：
- PUSH 5
- POP AX
- MOV [a], AX
-
- ● 原始程式： b:=2;
- 建碼如下：
- PUSH 2
- POP AX
- MOV [b], AX

- ● 原始程式： IF a>b THEN c:=a-b;
- 建碼如下：

```
        PUSH    WORD [a]
        PUSH    WORD [b]
        POP     BX
        POP     AX
        CMP     AX, BX
        JG      _go2
        JMP     _go3
_go2:
        PUSH    WORD [a]
        PUSH    WORD [b]
        POP     BX
        POP     AX
        SUB     AX, BX
        PUSH    AX
        POP     AX
        MOV     [c], AX
_go3:
```

- ● 原始程式： WRITE(c);
- 建碼如下：
-
- itostr c, _intstr, '$'
- MOV DX, _intstr
- MOV AH, 09H
- INT 21H
- newline
-
- ● 原始程式： .
- 建碼如下：
-
- MOV AX, 4C00H
- INT 21H