第十二章 plone 編譯器

- 我們的編譯器程式 plone.c 總算經過掃描程式測試、語彙分析測試、語法分析測試、建碼測試後最後才完成的，經過測試後發現有下列的缺陷：
- ***（1）. 所有的識別字，不管是程序之內或之外，***
- ***均不能重複，否則會產生語法錯誤。***
- ***（2）. 從程序外面可以取得程序內之區域變數。***
- 其實這兩個缺陷在我們當初訂出設計目標時就已經埋下原因，因為我們為了簡化，只對於識別字作單純的管理，並沒有分整體變數或區域變數的關係。
- 本章要對這兩個缺陷修正，修正後的編譯器程式才是完整的。

12.1 識別字結構

- 程序內外的識別字若允許重複，在建碼時必須將它們區別開來，例如在程序 A 裡宣告一個識別字 n，在程序 B 裡也宣告一個識別字 n，您若不加於區分，那麼建碼時會碰到下列的情形：
- n DW 0
- ...
- n DW 0
- 那麼在組譯程式 nasmw 組譯時就會發生語法錯誤，無法組譯當然無法產生目的程式，也就無法執行。
- 若識別字 n 與程序名稱結何在一起，如下：
- ***A_n DW 0***
- ***...***
- ***B_n DW 0***
- 將可避免重複宣告的問題。
- 因此識別字結構 struct idobjTag 要加上程序名稱 procname 一欄，其程式碼設計成表頭檔 idobj.h 如下。

```
/****************** idobj.h *****************/
struct idobjTag
{
  char name[36];
  int sym;
  int attr;
  int level;
  char procname[36];
  struct idobjTag *next;
};
char * idobjToString(struct idobjTag *p)
{
  static char str[512];
  sprintf(str,"  name:\"%s\"\tsym:%d\tattr:%d"
    "\tlevel:%d\tprocname:\"%s\"",
    p->name, p->sym, p->attr,
    p->level, p->procname);
  return str;
}
```

- 程式 idobjtest.c 用於測試 idobj.h 表頭檔是否正確。在程式裡建立一個識別字結構 struct idobjTag 的指標變數 p，並取得一塊其結構大小的記憶體，其第 0 個位元組的位址就是 p。
- ***name 欄位***
- **存入字串 "speed"，為該識別字的名稱。**
- ***sym 欄位***
- **存入整數 symIDENTIFIER，該整數常數定義於 sym.h 表頭檔裡頭，表示該識別字屬於識別字符記（token）。**
- ***attr 欄位***
- **存入整數 symVAR，表示該識別字宣告於 VAR 變數區域。**
- ***level 欄位***
- **存入整數 1，表示該識別字所屬的程序層次為第一層。**
- ***procname 欄位***
- **存入 "myproc"，表示該識別字屬於 myproc 程序。**
- ***next 欄位***
- **存入 NULL 常數，表示該識別字為 myproc 程序所宣告的最後一個識別字。**

- 識別字結構裡的每一個欄位值確定後，透過 idobjToString(p) 函式將該結構的內容以文字方式傳回並輸出。

```
/******************** idobjtest.c ********************/
#include <stdio.h>
#include <stdlib.h>
#include "sym.h"
#include "idobj.h"
int main()
{
  struct idobjTag *p=malloc(sizeof(struct idobjTag));
  strcpy(p->name, "speed");
  p->sym = symIDENTIFIER;
  p->attr = symVAR;
  p->level = 1;
  strcpy(p->procname, "myproc");
  p->next = NULL;
  printf("idobjToString(p)=\n%s\n", idobjToString(p));
  return 0;
}
```

12.2 範圍規則

- 一個程式裡若包含程序，程式裡所宣告的變數稱為整體（global）變數，程序裡所宣告的變數稱為區域（local）變數。如下例的程式 A2，宣告整體變數 m、n，程序 B 裡宣告區域變數 n、k。您在程序 B 裡不但可以存取自己的區域變數 n、k，同時也可以存取程式 A2 裡公用的整體變數 m、n。但您在程式 A2 裡卻無法存取在程序 B 裡所宣告的區域變數。
-
- 這種作法稱為範圍規則（scope rule）。

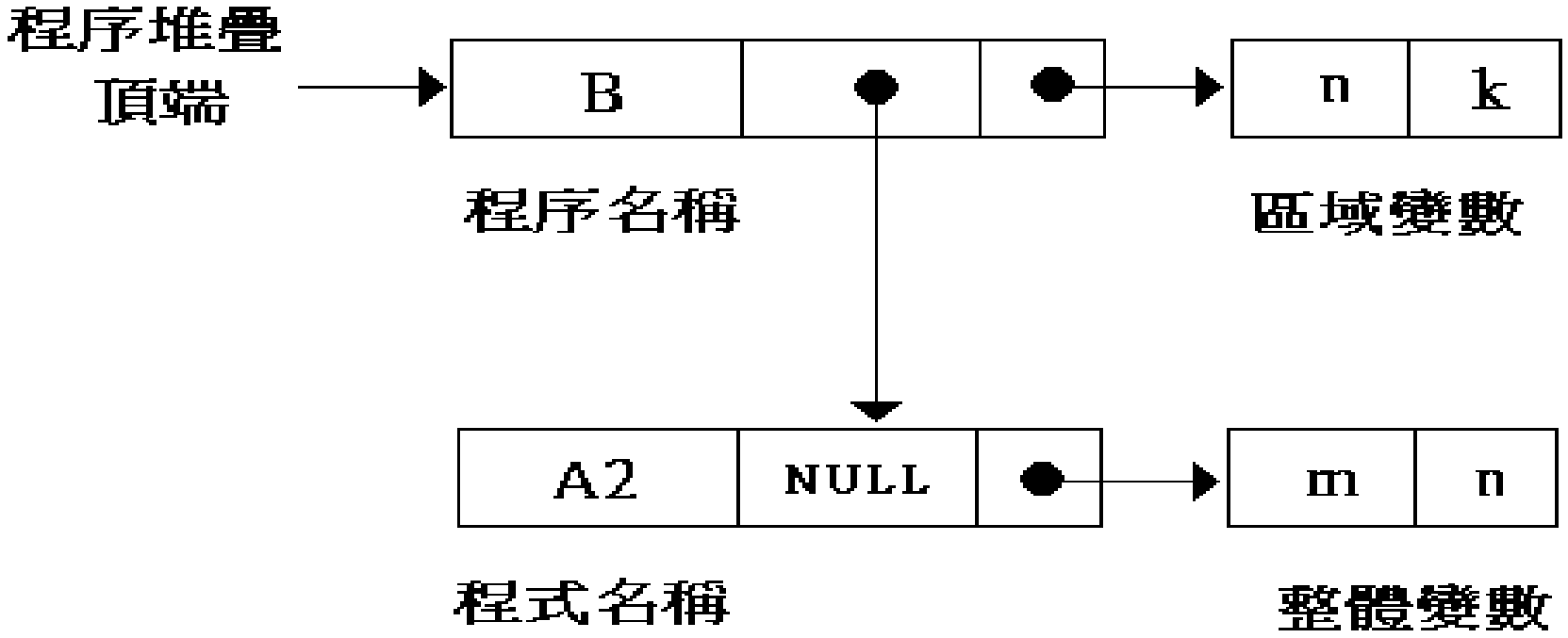
【程式a2.pl】

```
PROGRAM A2;
  VAR
    m, n;
  PROCEDURE B;
    VAR
      n, k;
    BEGIN
      n:=2;
      WRITE(n);
    END;
  BEGIN
    n:=1;
    WRITE(n);
    CALL B;
    WRITE(k);
  END.
```

- 【編譯a2.pl程式】
- C:\plone\ch12> plone a2.pl 1 <Enter>
- 1 PROGRAM A2;
- 2 VAR
- 3 m, n;
- 4 PROCEDURE B;
- 5 VAR
- 6 n, k;
- 7 BEGIN
- 8 n:=2;
- 9 WRITE(n);
- 10 END;
- 11 BEGIN
- 12 n:=1;
- 13 WRITE(n);
- 14 CALL B;
- 15 WRITE(k);
- **** ^26 識別字沒有宣告
- 16 END.
-
- Plone compile completed.
- Error count : 1

- 程序結構堆疊內容如下：
- B
- name:"n" sym:2 attr:31 level:1 procname:"B"
- name:"k" sym:2 attr:31 level:1 procname:"B"
- A2
- name:"m" sym:2 attr:31 level:0 procname:"A2"
- name:"n" sym:2 attr:31 level:0 procname:"A2"
- name:"B" sym:2 attr:32 level:0 procname:"A2"

- 程式檔 a2.pl 編譯時會發生錯誤，因為在程式 A2 裡「WRITE(k);」敘述要取得程序 B 裡所宣告的區域變數 k，違反範圍規則，會顯示 k 沒有宣告的訊息。程式 A2 裡整體變數及區域變數的關係如下圖所示。



- 程式 A2 若將錯誤的敘述「WRITE(k);」拿掉，儲存成 a.pl 再行編譯，將產生下列的組合語言 NASM 的程式碼，您有沒有注意到整體變數 m、n 在編譯後所產生的組合語言裡，變數前頭都冠上程式的名稱 A_m、A_n，區域變數 n、k 在編譯後所產生的組合語言裡，變數前頭都冠上程序的名稱 B_n、B_k，如此在程式 A 裡的整體變數 n 以及區域變數 n 就不會重複了。

```
;*************** A.asm *****************
;
            ORG     100H
            JMP     _start1
_intstr     DB      '     ','$'
_buf        TIMES   256 DB ' '
            DB 13,10,'$'
%include    "dispstr.mac"
%include    "itostr.mac"
%include    "readstr.mac"
%include    "strtoi.mac"
%include    "newline.mac"
A_m         DW      0
A_n         DW      0
```

```
_start1:
        JMP     _go2
B:
        JMP     _start2
B_n       DW      0
B_k       DW      0
_start2:
        PUSH      2
        POP     AX
        MOV     [B_n], AX
        itostr  B_n, _intstr, '$'
        MOV     DX, _intstr
        MOV     AH, 09H
        INT     21H
        newline
        RET
```

```
_go2
        PUSH    1
        POP     AX
        MOV     [A_n], AX
        itostr  A_n, _intstr, '$'
        MOV     DX, _intstr
        MOV     AH, 09H
        INT     21H
        newline
        CALL    B
        MOV     AX, 4C00H
        INT     21H
```

12.3 程序結構

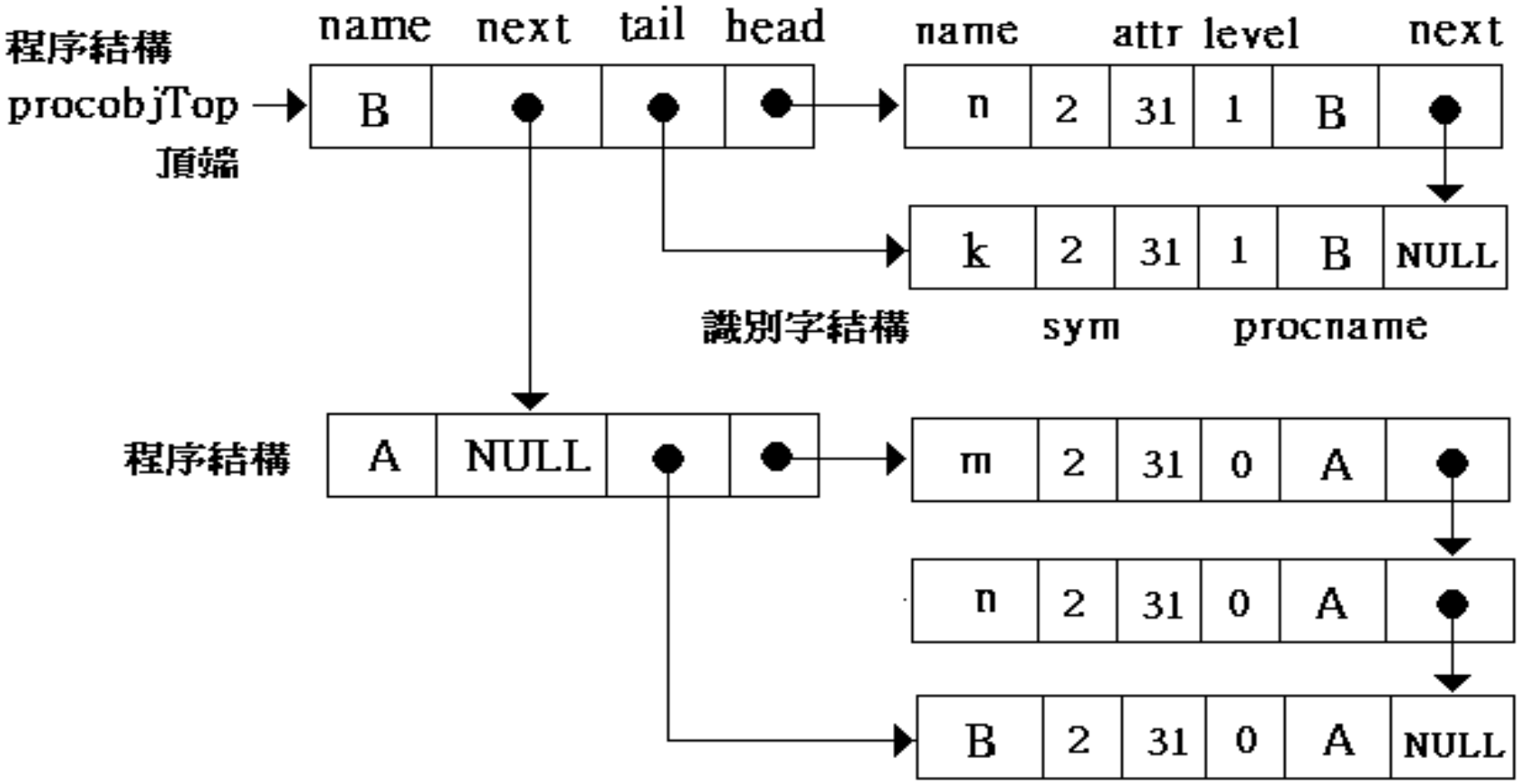
- 我們看了程式 A 裡整體變數及區域變數的關係後，就會思考在 plone 編譯器程式裡要如何使用 C 語言實作這種結構。程式或程序的結構包括四個欄位（field）：
- ***第一個欄位*** 為程式或程序的名稱 name。
- ***第二個欄位*** 指到上一層的程序或程式的指標 next，程式 A 是最上層，其上已經沒有了，因此以 NULL 表示，程序 B 的上一層為程式 A，因此指到 A。
- ***第三個欄位*** 宣告程式或程序本身的第一個變數指標 head。
- ***第四個欄位*** 宣告程式或程序本身的最後一個變數指標 tail。

程序結構 struct procobjTag

- 因此我們的程序結構 struct procobjTag 的架構如下：
- struct procobjTag
- {
- char name[36];
- struct procobjTag *next;
- struct idobjTag *head, *tail;
- } *procobjTop=NULL;
- 從上面的架構您可以看出 name 為程式或程序的名稱。next 是指到上一層的程序結構，一直指到最上層的程式結構為止。head 及 tail 識別字結構指標分別指到本程序結構所宣告的第一個及最後一個識別字，這個結構設計成佇列（queue），屬於先進先出（first in first out）的 FIFO 結構。

所有的程序結構堆疊

- 程式及程式裡所有的程序結構建立一個堆疊（stack），其頂端元素的指標為 procobjTop，其初值為 NULL 表示堆疊空白。堆疊裡的每一個元素均屬於 struct procobjTag 型態的結構。　以 a.pl 程式為例，其 procobjTag 所指的結構如下圖所示。



這種程序結構屬於靜態的（static），也就是說當 a.pl 程式撰寫完成後，其程序結構也就固定不變了。為了要建立一個程式的程序結構，您必須提供一些有用的函式。

建立一個空白的程序結構

- 首先您必須提供一個已知程序名稱，就能建立一個空白的程序結構。

```
struct procobjTag * newProcobj(char *name)
{
  struct procobjTag *p=malloc(sizeof(struct procobjTag));
  strcpy(p->name, name);
  p->head = NULL;
  p->tail = NULL;
  p->next = NULL;
  return p;
}
```

疊入或疊出

- 此新建的程序結構 p 必須可以疊入（push）或疊出（pop）程序結構堆疊的頂端，其函式如下。
- ```
  void  procpush(struct procobjTag *p)
  {
    if (procobjTop != NULL)
      p->next = procobjTop;
    procobjTop = p;
  }
  ```
- 函式 procpush() 將 p 程序結構疊入 procobjTop 頂端。從程式碼可以看出 p 的 next 值為原來的頂端元素，新的頂端指向 p 元素。
- ```
  struct  procobjTag * procpop()
  {
    struct procobjTag *p=procobjTop;
    procobjTop = procobjTop->next;
    return p;
  }
  ```
- 函式 poppush() 從堆疊 procobjTop 頂端疊出頂端元素至 p，並將原來的頂端元素指標 procobjTop 指向原來頂端的下一個元素。將剛疊出的元素指標 p 傳回。

從已知的程序名稱找出其指標

- 我們也需要從已知的程序名稱 procname 找出其指標，findProcobj() 函式的程式碼如下：

```
struct procobjTag * findProcobj(char *procname)
{
  struct procobjTag *p;
  p = procobjTop;
  while (p!=NULL)
  {
    if (strcmp(p->name, procname)==0)
      return p;
    else
      p = p->next;
  }
  return p;
}
```

建立一個識別字結構 p

```
struct idobjTag *newIdobj(char name[], int sym, int attr,
                          int level, char procname[])
{
  struct idobjTag *p=malloc(sizeof(struct idobjTag));
  strcpy(p->name, name);
  p->sym = sym;
  p->attr = attr;
  p->level = level;
  strcpy(p->procname, procname);
  p->next = NULL;
  return p;
}
```

將識別字結構加入至程序結構佇列尾端

```
void  varlistadd(struct procobjTag *p, struct idobjTag *v)
{
  if (p->head == NULL)
  {
    p->head = v;
    p->tail = v;
  }
  else
  {
    (p->tail)->next = v;
    p->tail = v;
  }
}
```

```
/********************* procobjtest.c *********************/
#include <stdio.h>
#include "idobj.h"
#include "procobj.h"
int main()
{
  struct procobjTag *A=newProcobj("A");
  struct procobjTag *B=newProcobj("B");
  struct idobjTag *id_1 = newIdobj("m",0,0,0,"A");
  struct idobjTag *id_2 = newIdobj("n",0,0,0,"A");
  struct idobjTag *id_3 = newIdobj("n",0,0,1,"B");
  struct idobjTag *id_4 = newIdobj("k",0,0,1,"B");
  varlistadd(A, id_1);
  varlistadd(A, id_2);
  varlistadd(B, id_3);
  varlistadd(B, id_4);
  procpush(A);
  procpush(B);
```

```
    printf("程序結構如下：\n%s\n", procobjToString());
    struct idobjTag *id = getIdobj(A,"n");
    printf("程序A裡的識別字n如下：\n%s\n",idobjToString(id));
    id = getIdobj(B,"n");
    printf("程序B裡的識別字n如下：\n%s\n",idobjToString(id));
    return 0;
  }
```

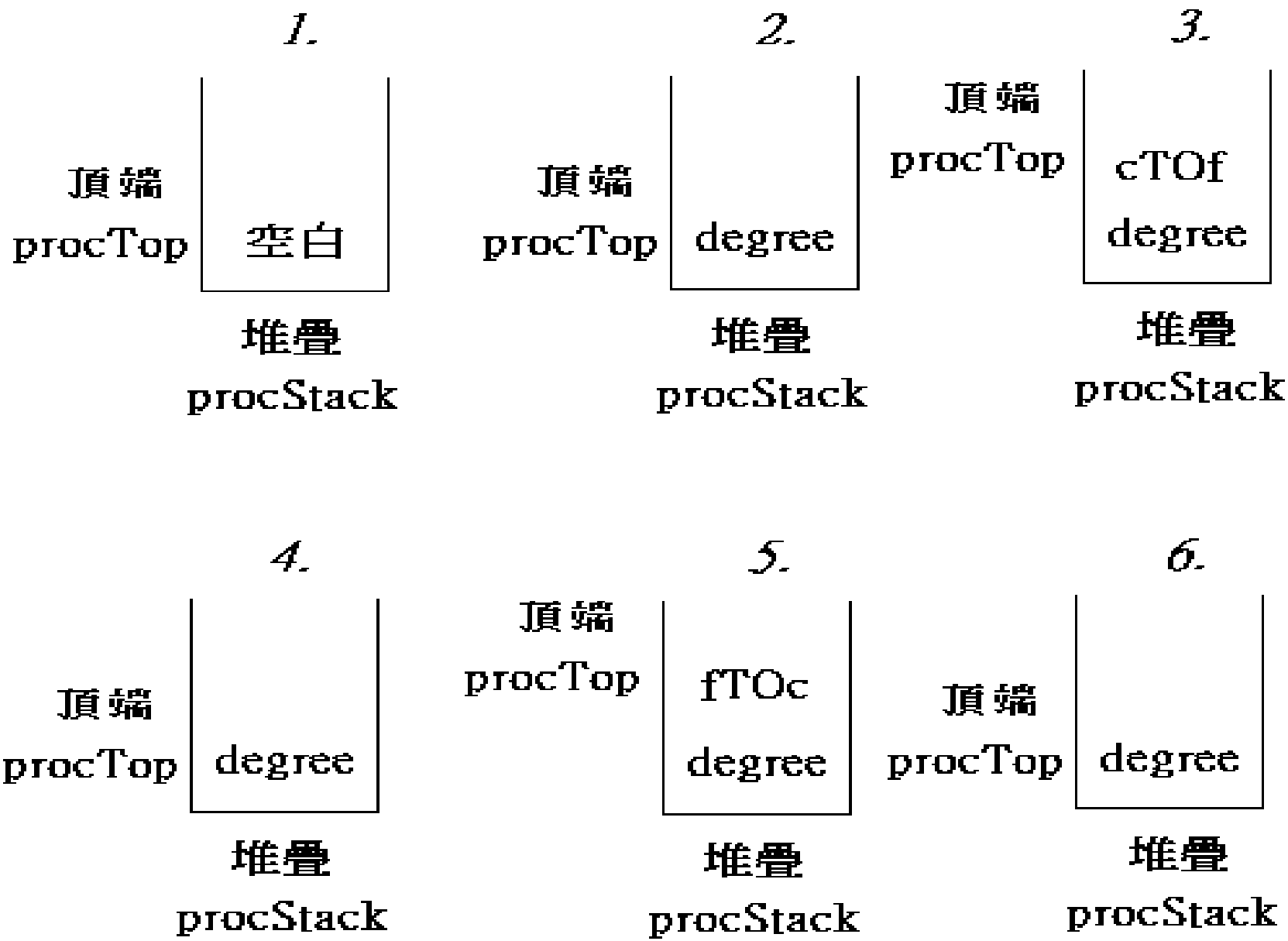
12.4 程序與區塊

- **plone 編譯器的與法裡頭，區塊 Block 包含 ProcDeclaration 程序宣告，而程序宣告裡頭又包含區塊，構成一個遞迴的（resursive）關係。**

- **3. <Block> ::= [<ConstDeclaration>]**
- **[<VarDeclaration>]**
- **[<ProcDeclaration>]**
- **<CompoundStatement>**
- **6. <ProcDeclaration> ::=**
- **{PROCEDURE <Identifier>;<Block>;}**

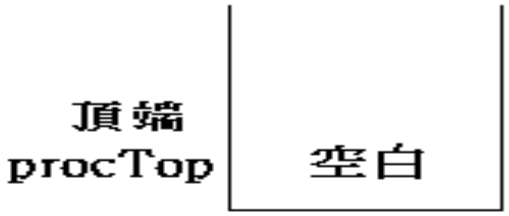
- **語法規則 3** 說明區塊可包含常數宣告、變數宣告、程序宣告，必須包含一個複合敘述。**語法規則 6** 說明程序宣告以保留字 **PROCEDURE 起頭，其後跟著一個識別字、分號、區塊、分號，且可重複無窮次的程序宣告，而每一個區塊裡可能又包含程序宣告，變化無窮，可以**說程序宣告裡又有程序宣告，程序宣告的層次與堆疊的作業類似，最深層的程序宣告最先返回，最淺層的最後返回。

- degree.pl 程式包含下列三個程序：
- 程式 degree
- 程序 cTOf
- 程序 fTOc
- 程式 degree 的層次為 0，程序 cTOf 及 fTOc 的層次均為 1，表示它們在這個程式裡的地位相同。
- 我們在處理原始程式 degree.pl 時首先碰到的是程式 degree 就將它疊入堆疊頂端，然後碰到程序 cTOf，再將它疊入堆疊頂端，此程序處理完成後再將此程序疊出，碰到程序 fTOc，再將其它疊入堆疊頂端，此程序處理完成後再將該程序疊出，程式 degree 處理完成後再將程式疊出，可見使用堆疊的結構符合我們要求。如下圖所示。


1.
頂端
procTop
空白
堆疊
procStack
2.
頂端
procTop
degree
堆疊
procStack
3.
頂端
procTop
cTOf
degree
堆疊
procStack
4.
頂端
procTop
degree
堆疊
procStack
5.
頂端
procTop
fTOc
degree
堆疊
procStack
6.
頂端
procTop
degree
堆疊
procStack

- 我們在處理原始程式 hcfmain.pl 時首先碰到的是程式 hcfmain，將它疊入堆疊頂端，然後碰到程序 hcfproc，再將它疊入堆疊頂端，此程序處理完成後再將該程序疊出，碰到程序 nextdivide 時，再將其它疊入堆疊頂端，此程序處理完成後再將它疊出，碰到程序 lcmproc，將它疊入堆疊頂端，此程序處理完成後再將該程序疊出，程式 hcfmain 處理完成後再將程式疊出，可見使用堆疊的結構符合我們要求。如下圖所示。

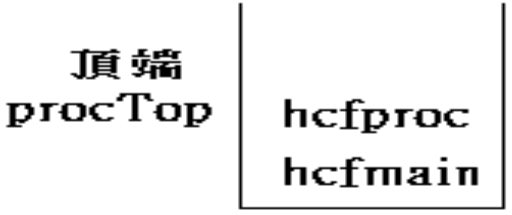
1.


頂端
procTop
空白
堆疊
procStack

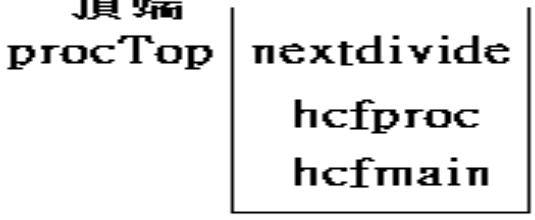
2.


頂端
procTop
hcfmain
堆疊
procStack

3.


頂端
procTop
hcfproc
hcfmain
堆疊
procStack

4.


頂端
procTop
nextdivide
hcfproc
hcfmain
堆疊
procStack

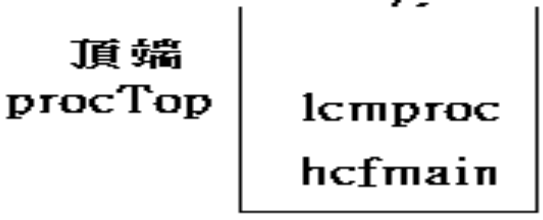
5.


頂端
procTop
hcfproc
hcfmain
堆疊
procStack

6.


頂端
procTop
hcfmain
堆疊
procStack

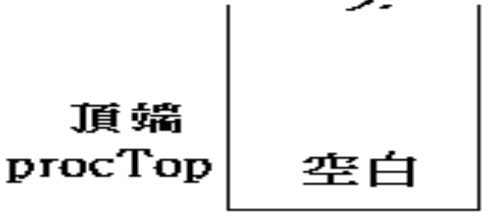
7.


頂端
procTop
lcmproc
hcfmain
堆疊
procStack

8.


頂端
procTop
hcfmain
堆疊
procStack

9.


頂端
procTop
空白
堆疊
procStack

12.5 修正後的plone編譯器程式

- /************************ plone.c ************************/
- #include <stdio.h>
- #include <stdlib.h>
- #include <string.h>
- /*
- ** 自訂表頭檔
- */
- #include "scanner.h"
- #include "resword.h"
- #include "err.h"
- #include "followsym.h"
- #include "idobj.h"
- #include "procobj.h"
- //略

12.6 測試程式

- 這個 degree.pl 測試程式包含主程式 degree 以及 cTOf、fTOc 等二個程序，這兩個程序均屬於 degree 主程式。編譯命令如下：
- 【編譯】
- C:\plone\ch12> plone degree.pl 1 <Enter>

```
 1 PROGRAM degree;
 2 CONST
 3   msg = " please keyin a degree (deg) : ",
 4   msgDeg = " deg = ",
 5   msgC = " C degree in PROGRAM degree = ",
 6   msgF = " F degree in PROGRAM degree = ",
 7   msgF2 = " F degree in PROCEDURE cTOf = ";
 8 VAR
 9   deg, f, c;
10 PROCEDURE cTOf;
11   VAR
12     f;
13   BEGIN
14     f := deg*9/5+32;
15     WRITE(msgF2, f);
16   END;
```

```
17 PROCEDURE fTOc;
18   VAR
19     g;
20   BEGIN
21     c := (deg-32)*5/9;
22   END;
23 BEGIN
24   WRITE(msg);
25   READ(deg);
26   CALL cTOf;
27   CALL fTOc;
28   WRITE(msgF, f);
29   WRITE(msgC, c);
30 END.

Plone compile completed.
Error count : 0
```

- **程序結構堆疊**內容如下：
- fTOc
- name:"g" sym:2 attr:31 level:1 procname:"fTOc"
- cTOf
- name:"f" sym:2 attr:31 level:1 procname:"cTOf"
- degree
- name:"msg" sym:2 attr:30 level:0 procname:"degree"
- name:"msgDeg" sym:2 attr:30 level:0 procname:"degree"
- name:"msgC" sym:2 attr:30 level:0 procname:"degree"
- name:"msgF" sym:2 attr:30 level:0 procname:"degree"
- name:"msgF2" sym:2 attr:30 level:0 procname:"degree"
- name:"deg" sym:2 attr:31 level:0 procname:"degree"
- name:"f" sym:2 attr:31 level:0 procname:"degree"
- name:"c" sym:2 attr:31 level:0 procname:"degree"
- name:"cTOf" sym:2 attr:32 level:0 procname:"degree"
- name:"fTOc" sym:2 attr:32 level:0 procname:"degree"

測試程式 degree.pl 其整體變數及區域變數的關係如下圖所示。