
本章內容：

- * MGL 簡介
- * MGL 的設計
- * 螢幕的處理
- * 結束部分的處理
- * MGL 的範例程式碼
- * 習題

第四章

選單產生器的語言

上一章，我們設計了一個計算機當作直譯器 (interpreter) 的例子。在本章裡，我們藉由設計一個選單產生器的語言 (Menu Generation Language, 以下簡稱為 MGL) 以及相關的編譯器，來研究編譯器的設計。我們先描述這個語言的功能與外觀，然後開始設計的過程。在設計過程中，你將會不斷的修改 lex 與 yacc 規格。最後，我們會開發出 MGL 語法的動作程式碼，來達到 MGL 的功能。

MGL 簡介

我們的目的是設計出一個語言，讓它能產生不同的選單介面。我們利用這個語言寫出一個描述檔，再經由編譯器編譯後產生 C 語言程式碼。這些程式碼是利用 curses 函式庫 (註) 在螢幕上畫出描述檔所描述的選單。

註 關於 curses 的設計，請參考由 John Stang 所著的「Programming with Curses」。

在很多情況下，一個發展中的程式會有許多重複且十分繁雜的程式碼。這個時候，如果先設計一個簡單的語言，寫一個配合的編譯器來產生 C 語言程式碼或是其他的結果，通常速度會比較快，過程也比較容易。例如，要使用 `curses` 函式庫，它的程式碼就很繁雜，因為你必須自己一步一步決定資料要擺在螢幕上的位置。透過 MGL，可用來產生版面設計的程式碼，大幅減輕我們的工作。

要描述一個選單，包括下面幾項：

1. 選單的名稱
2. 標題
3. 一連串的選單選項。每一個選項包括：

選項

[命令]

動作

[屬性]

「選項」就是要顯示在選單上的字串，「命令」是一段助憶碼 (mnemonic)，讓使用者在命令列中，可以直接利用命令來執行選單對映的功能；「動作」則是當使用者選擇選項之後要執行的步驟；最後，「屬性」用來代表這個選項要不要被處理。方括號中的項目是可有可無的。

4. 結束部分

對一個真正有用的程式來說，通常都包含很多個選單。所以在選單的描述檔中，也要能夠描述各種名稱的不同選單。

下面是一個選單的描述範例：

```
screen myMenu
title "My First Menu"
title "by Tony Mason"

item "List Things to Do"
command "to-do"
```

```
action execute list-things-todo
attribute command

item "Quit"
command "quit"
action quit

end myMenu
```

MGL 編譯器讀進這個描述檔，產生對映的 C 語言程式碼。如果我們編譯產生的程式碼，執行後就會得到描述檔所描述的選單。例如：

```
My First Menu
by Tony Mason

1) List Things to Do
2) Quit
```

當使用者按下「1」鍵，或者輸入「to-do」這個命令時，「list-things-todo」這個程式就會被執行。

比較一般化的選單描述格式應該是：

```
screen <name>
title <string>

item <string>
[ command <string> ]
action {execute | menu | quit | ignore} <name>
[ attribute {visible | invisible} ]

end <name>
```

當我們在發展 MGL 時，一開始只會發展一小部份，然後慢慢加進其他的，最後完成整個功能。藉由這種方式你將會發現：當我們在修改語言的設計時，要修改 lex 產生的字彙分析器和 yacc 產生的解析器，事情會好辦多了。

MGL 的設計

讓我們看看要如何設計才能建立上面的語法。選單提供初學者一個簡單的程式介面，因為選單的環境很穩定而且容易操作。

不過選單有一個主要的缺點：它使得有經驗的老手不能直接執行想要的功能，這些人比較喜歡能直接下命令的介面。但每個人，包括高手在內，偶爾還是需要選單，幫助他們執行一些不常用的功能。

所以，在我們設計 MGL 時，希望同時達到這兩個目標。一開始，我們用「command」關鍵字開頭，表示使用者想要執行選單中某個選項，或是某個命令。

光是一個關鍵字，看起來實在沒什麼作用。不過，利用這個關鍵字，我們可以寫出大概的 lex 規格，

```
ws  [ \t]+
nl  \n
%%
{ws} ;
command      { return COMMAND; }
{nl} { lineno++; }
. { return yytext[0]; }
```

以及相對的 yacc 語法：

```
%{
#include <stdio.h>
}%

%token COMMAND
%%

start:      COMMAND
          ;
```

到目前為止，我們的字彙分析器就是尋找有沒有關鍵字。當它找到的時候，就傳回適當的 token。然後，一旦解析器看到 COMMAND 這個 token 時，就發現符合 start 規則。最後，yyparse() 函式就執行成功並返回。

選單中的每個選項，會對映某個動作指令，所以我們需要「action」關鍵字。選項的動作可能是設定成忽略（因為可能是我們還沒有完成的功能，或是目前還不能執行的功能），或是執行另外的程式，所以我們需要另外兩個關鍵字：「ignore」、「execute」。

因此，綜合上面提到的字彙，一個選項的範例像是：

```
command action execute
```

不過我們在此必須說明：到底要執行什麼，所以還要加上要執行程式的名稱。由於程式名稱可能包含標點符號，所以我們預設程式名稱是用引號括起來的字串。之前的範例可以改成：

```
choice action execute "/bin/sh"
```

我們將修改後的 lex 規格寫在範例 4-1，主要是針對新的關鍵字，以及新的 token 來做修改。

範例 4-1：MGL 字彙分析器的第一個版本

```
ws      [ \t]+
qstring "[^\"\\n]*[\"\\n]
nl      \n
%%
{ws}    ;
{qstring} { yylval.string = strdup(yytext+1); /* 跳過開頭的引號 */
            if(yyval.string[yylen-2] != '"')
                warning("Unterminated character string", (char *)0);
            else
                yyval.string[yylen-2] = '\0'; /* 將結尾引號刪除 */
            return QSTRING;
        }
action   { return ACTION; }
execute  { return EXECUTE; }
command  { return COMMAND; }
ignore   { return IGNORE; }
{nl}     { lineno++; }
.        { return yytext[0]; }
```

範例中的 `qstring` 的定義相當複雜。這是為了避免 `lex` 將下面的輸入比對成單一字串。

```
"apples" and "oranges"
```

表示法中的「`[^\n]*`」是用來描述任何一個字元，除了引號跟換行字元之外。我們不希望字串超過一行；假如字串的結尾引號根本不存在，字彙分析器會一直搜尋下去，甚至找到檔案結束，這並不是我們想要的，而且這樣無止境的找下去，會使得字彙分析器的暫存區爆掉（`overflow`），甚至造成整個程式當掉。若使用範例中的表示法，可以用比較優雅的方式來警告使用者。在複製字串的時候，我們將開頭與結尾的引號刪除，因為沒有引號的字串處理起來比較容易。

同樣的，`yacc` 語法也要修正，如範例 4-2 所示。

範例 4-2：MGL 解析器的第一個版本

```
%{
#include <stdio.h>
}%

%union {
    char *string;          /* 字串暫存區 */
}

%token COMMAND ACTION IGNORE EXECUTE
%token <string> QSTRING
%%
start:      COMMAND action
          ;

action:     ACTION IGNORE
          | ACTION EXECUTE QSTRING
          ;
%%
```

我們用 `%union` 加入 `string` 來儲存字串，以及新增 `QSTRING` 這個 `token`，用來代表被引號括著的字串。

接下來，我們需要將命令以及相關的動作合起來，當作一個選單選項，所以我們新增一個「item」關鍵字來代表選項。假如這個選項有相關的指令，我們就加上command 這個關鍵字。以下是我們將新的關鍵字加進來字彙分析器：

```
    . . .
%%
    . . .
    item          { return ITEM; }
```

雖然我們已經加進來新功能將語言的結構改變很多，可是對字彙分析器來說，只要修改一點地方。其他需要修改的大多是解析器的規格，如範例 4-3 所示。

範例 4-3：具有選項跟動作的語法

```
%{
#include <stdio.h>
}%

%union {
    char *string;          /* 指向字串的指標 */
}

%token COMMAND ACTION IGNORE EXECUTE ITEM
%token <string> QSTRING
%%
item:    ITEM command action
        ;

command: /* 空的 */
        | COMMAND
        ;

action:  ACTION IGNORE
        | ACTION EXECUTE QSTRING
        ;

%%
```

並不是每個選項都有對映的命令，所以 `command` 的其中一條規則是空的 RHS。雖然是空的，`yacc` 也能夠處理這種情況，只要其他語法不會因為命令的存在與否，而有分歧性就行了。

到這裡為止，我們還沒有給 `command` 關鍵字下真正的「定義」，只是給一個關鍵字。事實上，通常我們會先單獨寫一條規則，看起來就像整個語言中的一個洞，等一下再回頭來補齊。不過，我們很快就要補上了。

接下來，我們限定命令只能是字母或數字組成的字串。於是，我們在字彙分析器中加入 `ID` 這個 `token` 來代表識別字 (`identifier`)：

```

. . .
id      [a-zA-Z][a-zA-Z0-9]*
%%
. . .
{id}    { yyval.string = strdup(yytext);
          return ID;
        }

```

`ID` 的符號值是一個指向識別字名稱的指標。一般來說，直接將 `yytext` 的指標當作某個符號值，都是不適當的，因為當字彙分析器讀到下一個 `token` 之後，`yytext` 的內容就會改變（所以，沒有將 `yytext` 複製一份，通常會造成字彙分析器執行錯誤，像字串的內容會莫名其妙的被改變）。我們在這裡利用 `strdup()` 函式，將 `token` 複製一份，然後傳回複製後字串的指標。使用 `ID` 的規則必須記得在處理完的時候，釋放複製的字串空間。

在範例 4-4 中，我們加入了新的 `ID token` 到 `yacc` 解析器中。

範例 4-4：可以處理命令列識別字的語法

```

%{
#include <stdio.h>
%}

%union {
    char *string;          /* 字串暫存區 */
}

%token COMMAND ACTION IGNORE EXECUTE ITEM

```

範例 4-4 (續)

```

%token <string> QSTRING ID
%%
item:    ITEM command action
        ;

command: /* 空的 */
        | COMMAND ID
        ;

action:  ACTION IGNORE
        | ACTION EXECUTE QSTRING
        ;
%%

```

目前的語法並不允許一個選單中有多個選項，所以我們加入了 `items` 的規則，來達到多個 `item` 的功能：

```

. . .
%%
items: /* 空的 */
      | items item
      ;

item:  ITEM command action
      ;
. . .

```

跟之前規則不同的是：這一次的新增的規則是遞迴形式的。由於 `yacc` 對左遞迴的執行效果比較好，我們在規則裡是寫「`items item`」而不是右遞迴的形式的「`item items`」(請參考第七章中的「遞迴規則」，以瞭解為什麼左遞迴執行效果比較好)。

`items` 的其中一條規則有空的 RHS。對大部分的遞迴規則來說，這就是所謂的終結條件 (terminating condition)。終結條件必須符合非終端符號，而且要是非遞迴的形式。假如一條規則完全都是遞迴而沒有非遞迴的部分，則大部分的 `yacc` 都會因此當掉，因為在這樣子的情況下，`yacc` 無法產生一個可以用的解析器。在之後的例子，我們還會用到許多左遞迴的規則。

選單中除了有選項之外，你可能還希望在選單上能顯示標題。下面就是用來描述標題的語法：

```
title:      TITLE      QSTRING
          ;
```

我們用 `title` 關鍵字來開始描述標題。此外，我們希望標題是一個被引號括起來的字串。在 `lex` 規格中，我們加入一個 `TITLE` 這個新的 `token`：

```
. . .
%%
title      { return TITLE; }
. . .
```

我們可能希望標題能夠超過一行，於是將語法修改成：

```
titles:    /* 空的 */
          | titles title
          ;

title:     TITLE      QSTRING
          ;
```

同樣的，有了遞迴的定義之後，就允許多行的標題。

加入標題的功能之後，我們就需要加入一個更高階的規則來同時包含標題與選項，而且標題在選項之前。所以在範例 4-5 中，我們加入了 `start` 這條新的規則。

範例 4-5：可以處理標題的語法

```
%{
#include <stdio.h>
}%

%union {
    char *string;          /* 字串暫存區 */
}

%token COMMAND ACTION IGNORE EXECUTE ITEM TITLE
%token <string> QSTRING ID
%%
```

範例 4-5 : (續)

```
start: titles items
      ;

titles: /* 空的 */
       | titles title
       ;

title:  TITLE      QSTRING
      ;

items: /* 空的 */
      | items item
      ;

item:  ITEM command action
      ;

command: /* 空的 */
        | COMMAND ID
        ;

action: ACTION IGNORE
       | ACTION EXECUTE QSTRING
       ;

%%
```

用了 MGL 幾次之後，你可能覺得只有一個選單不太方便，所以我們希望一次定義多個選單，而且能夠從一個選單跳到另一個選單。因此，我們定義了新的 screen (螢幕) 【譯註】規則來包含標題和選項。此外，為了使用多重選單，我們利用遞迴規則來寫下 screens 規則。為了包括空的螢幕定義，我們總共加進五條新規則。

譯註 我將 screen 翻成螢幕，因為選單可以看做是出現在螢幕上的。不要和顯示器的硬體螢幕混為一談。

```
screens: /* 空的 */
        | screens screen
        ;

screen: screen_name screen_contents screen_terminator
       | screen_name screen_terminator
       ;

screen_name: SCREEN ID
           | SCREEN
           ;

screen_terminator: END ID
                | END
                ;

screen_contents: titles lines
```

每一個螢幕都有自己的名稱。當我們需要描述某一個螢幕選單，例如像「first」，可以這樣子寫：

```
item "first" command first action menu first
```

由於每個選單螢幕都有名稱，我們還需要一個命令來指出選單螢幕的結束，所以需要一條 `screen_terminator` 規則。幾個選單螢幕的範例如下：

```
screen main
title "Main screen"
item "fruits" command fruits action menu fruits
item "grains" command grains action menu grains
item "quit" command quit action quit
end main

screen fruits
title "Fruits"
item "grape" command grape action execute "/fruit/grape"
item "melon" command melon action execute "/fruit/melon"
item "main" command main action menu main
```

```
end fruits
screen grains
title "Grains"
item "wheat" command wheat action execute "/grain/wheat"
item "barley" command barley action execute "/grain/barley"
item "main" command main action menu main
end grains
```

不過，我們的規則允許沒有名稱的選單螢幕，所以，`screen_name` 跟 `screen_terminator` 各自有兩條規則。但是，當我們真正要開始撰寫規則的動作碼時，必須要檢查選單螢幕的名稱是否對稱等等。

我們還可以加入更多的功能到 MGL 中。對每一個選項來說，我們可以指定「可見的 (visible)」或是「不可見的 (invisible)」。由於這個屬性是單獨與每一個選項有關，我們在語法定義裡加進新的 `attribute` 關鍵字。以下是用來描述這個屬性的新語法：

```
attribute: /* 空的 */
          | ATTRIBUTE VISIBLE
          | ATTRIBUTE INVISIBLE
          ;
```

我們允許 `attribute` 的欄位是空的，然後預設為「visible」。範例 4-6 是修改後的語法。

範例 4-6：完整的 MGL 語法

```
screens: /* empty */
        | screens screen
        ;

screen:  screen_name screen_contents screen_terminator
        | screen_name screen_terminator
        ;

screen_name:  SCREEN ID
             | SCREEN
             ;
```

範例 4-6 : (續)

```
screen_terminator:  END ID
                    |  END
                    ;

screen_contents: titles lines
                ;

titles: /* empty */
       | titles title
       ;

title: TITLE QSTRING
      ;

lines: line
      | lines line
      ;

line: ITEM QSTRING command ACTION action attribute
     ;

command: /* empty */
        | COMMAND ID
        ;

action:  EXECUTE QSTRING
        | MENU ID
        | QUIT
        | IGNORE
        ;

attribute: /* 空的 */
          | ATTRIBUTE VISIBLE
          | ATTRIBUTE INVISIBLE
          ;
```

我們將之前範例使用的 `start` 規則，改為 `screens` 規則來當作最上層的規則。因為當沒有在定義段落中用 `%start` 來特別指定時，預設就是以第一條規則為最上層規則。

建立 MGL

既然已經有了基本的 MGL 語法，該是建立編譯器的時候了。不過，我們必須根據之前的討論，在字彙分析器裡加入新的關鍵字。範例 4-7 是修改後的 `lex` 規格。

範例 4-7：MGL 用的 `lex` 規格

```
ws      [ \t]+
comment #.*
qstring \"^[^\"\\n]*[\\\"\\n]
id      [a-zA-Z][a-zA-Z0-9]*
nl      \\n

%%

{ws}    ;
{comment} ;
{qstring} { yylval.string = strdup(yytext+1);
            if(yylval.string[yyleng-2] != '\\')
                warning("Unterminated character string", (char *)0);
            else
                yylval.string[yyleng-2] = '\\0'; /* 移除結尾的引號 */
            return QSTRING;
        }
screen  { return SCREEN; }
title   { return TITLE; }
item    { return ITEM; }
command { return COMMAND; }
action  { return ACTION; }
execute { return EXECUTE; }
menu    { return MENU; }
```

範例 4-7 (續)

```
quit      { return QUIT; }
ignore    { return IGNORE; }
attribute { return ATTRIBUTE; }
visible   { return VISIBLE; }
invisible { return INVISIBLE; }
end        { return END; }
{id}      { yyval.string = strdup(yytext);
           return ID;
           }
{nl}      { lineno++; }
.         { return yytext[0]; }
%%
```

範例 4-8 是另一種加進新關鍵字的方法。

範例 4-8：另一種方法的 lex 規格

```
. . .
id        [a-zA-Z][a-zA-Z0-9]*
%%
. . .
{id}      { if(yyval.cmd = keyword(yytext)) return yyval.cmd;
           yyval.string = yytext;
           return ID;
           }
%%
/*
 * keyword() 函式：傳進一個字串，先判斷是不是關鍵字。
 * 若是，則傳回關鍵字的符號值；若不是，則傳回零。
 * 注意：關鍵字的符號值不能為零。
 */

static struct keyword {
char *name;      /* text string */
int value;      /* token */
} keywords[] =
{
```

範例 4-8 (續)

```
"screen",    SCREEN,
"title",     TITLE,
"item",      ITEM,
"command",   COMMAND,
"action",    ACTION,
"execute",   EXECUTE,
"menu",      MENU,
"quit",      QUIT,
"ignore",    IGNORE,
"attribute", ATTRIBUTE,
"visible",   VISIBLE,
"invisible", INVISIBLE,
"end",       END,
NULL, 0,
};

int keyword(string)
    char *string;
    {
        struct keyword *ptr = keywords;
        while(ptr->name != NULL)
            if(strcmp(ptr->name, string) == 0)
                {
                    return ptr->value;
                }
            else
                ptr++;
        return 0; /* 沒有關鍵字 */
    }
```

範例 4-8 的方式是利用一個靜態的表格來記錄關鍵字。在執行的時候，這種方式一定比較慢；當 lex 比對關鍵字時，比對的速度不會受到關鍵字的個數或是樣式複雜度而影響，但範例 4-8 就會。這裡舉出範例 4-8 只是為了告訴大家：假如你打算要增加關鍵字到一個語言中，這也是一種可行的方式。在這個例子中，所有的關鍵字我們都是用相同的機制來比對，而且可在必要的時候加入新的關鍵字。

我們在邏輯上可以將建立 MGL 規格的過程，拆成下列幾個部分：

初始化的處理	將內部用到的表格初始化。先產生一些前置程式碼，供後來產生的程式碼使用。
螢幕開始部分的處理	在內部新增一個螢幕表格元素、將螢幕的名稱加到螢幕串列裡，以及產生螢幕處理需要的初始程式碼。
螢幕本身的處理	看到一個選項就處理一個；看到標題時，就加進標題串列中；當看到新的選單時，就加進新的選單串列中。
螢幕結束部分的處理	看到一個 end 指令時，就處理之前產生與螢幕相關的資料結構。
結束的處理	清除內部的狀態，產生最後的程式碼，以及確保整個編譯過程順利；若編譯過程有錯，就產生錯誤訊息給使用者。

初始化的處理

任何編譯器在開始真正的編譯之前，必須完成一些初始化的動作。例如：將內部要使用的資料結構初始化。還記得嗎？範例 4-8 使用查表的方法來比對關鍵字，而不像範例 4-7 是將關鍵字直接寫死在規格裡。在某些較複雜的情況時，符號表的初始化就是將關鍵字輸入符號表內，像範例 3-10 一樣。

一開始，我們的 main() 函式只有：

```
main()
{
    yyparse();
}
```

我們在開始編譯之前，必須要給編譯器一個檔名來進行編譯。由於 lex 從 yyin 讀進資料，將輸入寫到 yyout，而 yyin 預設是標準輸入，yyout 預設是標準輸出，所以我們只要重新設定 yyin 與 yyout 的內容，就可以達到想要的功能。要改變 yyin 或 yyout 的內容，我們使用標準 I/O 函式庫的 fopen() 函式來開啟要處理的檔案，將傳回值設給 yyin 或是 yyout。

假如使用者並沒有輸入任何檔名當作參數，我們就將結果輸出到 `screen.out` 這個預設檔案，以及從 `stdin` 標準輸入讀進資料。假如使用者輸入一個檔名當作參數，我們仍然以 `screen.out` 檔案當作輸出，但是將參數的檔案當作輸入。最後，假如使用者一次給兩個檔名當作參數，我們就把第一個參數當作輸入檔案，第二個參數當作輸出檔案。

當程式從 `yyparse()` 函式返回時，還必須做一些後置處理，以及檢查是否有錯誤發生。最後，清除資料後離開。

範例 4-9 是我們討論後的 `main()` 函式。

範例 4-9 : MGL 用的 `main()` 函式

```
char *programe = "mgl";
int lineno = 1;

#define DEFAULT_OUTFILE "screen.out"

char *usage = "%s: usage [infile] [outfile] \n";

main(int argc, char **argv)
{
    char *outfile;
    char *infile;
    extern FILE *yyin, *yyout;

    programe = argv[0];

    if(argc > 3)
    {
        fprintf(stderr, usage, programe);
        exit(1);
    }
    if(argc > 1)
    {
        infile = argv[1];
        /* 開啟輸入檔案 */
    }
}
```

範例 4-9 (續)

```
yyin = fopen(infile,"r");
if(yyin == NULL) /* 開啟失敗 */
{
    fprintf(stderr,"%s: cannot open %s \n",
            progname, infile);
    exit(1);
}

if(argc > 2)
{
    outfile = argv[2];
}
else
{
    outfile = DEFAULT_OUTFILE;
}

yyout = fopen(outfile,"w");
if(yyout == NULL) /* open failed */
{
    fprintf(stderr,"%s: cannot open %s \n", progname, outfile);
    exit(1);
}

/* 以下還是從 yyin 讀進輸入，從 yyout 輸出 */

yyparse( );

end_file( ); /* 輸出最後的部分 */

/* now check EOF condition */
if(!screen_done) /* 假如我們處理到螢幕定義的中間就結束了 */
{
    warning("Premature EOF", (char *)0);
    unlink(outfile); /* 移除錯誤的輸出 */
}
```

範例 4-9 (續)

```

        exit(1);
    }
    exit(0); /* 沒有錯誤發生 */
}

warning(char *s, char *t) /* 印出錯誤訊息 */
{
    fprintf(stderr, "%s: %s", progname, s);
    if (t)
        fprintf(stderr, " %s", t);
    fprintf(stderr, " line %d \n", lineno);
}

```

螢幕處理

一旦我們處理完初始化動作，以及開啟輸出輸入檔案之後，接下來是有關選單產生的工作，也就是處理選單的描述檔。首先，screens 規則並不需要動作程式碼。screen 規則包括screen_name (螢幕名稱)、screen_contents (螢幕內容)、以及screen_terminator (螢幕結束) 三個部分。首先來看 screen_name：

```

screen_name:  SCREEN ID
              | SCREEN
              ;

```

我們必須將螢幕的名稱加進螢幕串列中。若沒有指定名稱，我們就用「default」當作預設檔名。新增動作程式碼後，規則變成：

```

screen_name:  SCREEN ID  { start_screen($2); }
              | SCREEN   { start_screen(strdup("default")); }
              ;

```

(為了和第一條規則一樣傳進一個參數，在第二規則中，我們使用 strdup() 函式來複製字串，傳給 start_screen() 函式)。start_screen() 函式將傳進的名稱加進螢幕名稱的串列，以及產生必要的程式碼。

舉例來說，假如你在輸入檔案中有「screen first」，則 `start_screen()` 函式則可能要產生下列的程式碼：

```
/* 螢幕的第一個部分 */
menu_first()
{
    extern struct item menu_first_items[];

    if(!init) menu_init();

    clear();
    refresh();
}
```

接下來處理選單的定義。首先是標題：

```
title: TITLE QSTRING
;
```

我們呼叫 `add_title()` 函式，來決定要將標題擺在哪個位置：

```
title: TITLE QSTRING { add_title($2); }
;
```

所以，有關標題所產生的程式碼就像：

```
move(0,37);
addstr("First");
refresh();
```

對於每一個標題行，我們先移動游標，然後印出標題字串（就像基本的置中動作一樣）。當我們再遇到標題的定義時，就可以重複產生同樣的程式碼，唯一要改變的地方就是 `move()` 函式中的位置。

為了看看到底要怎麼做，我們讓選單有多一點的標題行：

```
screen first
title "First"
title "Copyright 1992"

item "first" command first action ignore
attribute visible
```

```

item "second" command second action execute "/bin/sh"
    attribute visible
end first

screen second
title "Second"
item "second" command second action menu first
    attribute visible
item "first" command first action quit
    attribute invisible
end second

```

根據上面的描述，標題行產生的程式碼是：

```

move(0,37);
addstr("First");
refresh();
move(1,32);
addstr("Copyright 1992");
refresh();

```

接下來看到的是一連串的選項。我們替每一個選項，在內部的表格裡建立相關要執行的動作，直到我們看到 `end first` 的描述。這個時候，我們就開始進行後置處理，最後完成一個螢幕描述的處理。為了要建立一個選項的表格，我們在 `item` 的規則裡要加進相關的動作程式碼：

```

line: ITEM qstring command ACTION action attribute
    {
        item_str = $2;
        add_line($5, $6);
        $$ = ITEM;
    }
;

```

其他如 `command`、`action` 以及 `attribute` 規則，主要是將符號值儲存起來供以後使用。

```

command: /* 空的 */ { cmd_str = strdup(""); }
    | COMMAND id { cmd_str = $2; }
;

```

command 可以是空的或是特別寫明的命令名稱。如果是前者，我們將指令字串設為空字串（利用 `strdup()` 函式達到和下一條規則同樣的效果）：如果是後者，我們將識別字存進 `cmd_str`。

action 規則的動作程式碼就比較複雜一些，因為有許多種情形要處理：

```

action: EXECUTE qstring
    { act_str = $2;
      $$ = EXECUTE;
    }
| MENU id
    { /* 要產生 "menu_" $2 */
      act_str = malloc(strlen($2) + 6);
      strcpy(act_str, "menu_");
      strcat(act_str, $2);
      free($2);
      $$ = MENU;
    }
| QUIT { $$ = QUIT; }
| IGNORE { $$ = IGNORE; }
;

```

attribute 規則比較簡單，只要適當的設定符號值就好了：

```

attribute: /* 空的 */          { $$ = VISIBLE; }
          | ATTRIBUTE VISIBLE  { $$ = VISIBLE; }
          | ATTRIBUTE INVISIBLE { $$ = INVISIBLE; }
;

```

action 和 attribute 規則的傳回值會傳給 `add_line()` 函式，這個函式透過兩個不定長度的字串指標以及兩個傳回值，在內部的狀態表格中產生一筆新的紀錄。

看到 `end first` 敘述之後，我們必須要結束整個螢幕部分的處理。我們結束 `menu_first` 函式：

```

    menu_runtime(menu_first_items);
}

```

實際上，每一個選單選項被儲存在 menu_first_items 這個陣列之中：

```
/* first 螢幕定義結束 */
struct item menu_first_items[]={
    {"first","first",271,"",0,273},
    {"second","second",267,"/bin/sh",0,273},
    {(char *)0, (char *)0, 0, (char *)0, 0, 0},
};
```

產生的程式碼還包括 menu_runtime() 函式，我們利用它來動態的顯示選項。

結束部分的處理

在處理結束部分的時候，我們呼叫一個後置處理函式來完成（請不要跟等一下提到的檔案結束後置處理函式混淆）。所以螢幕結束的規則是：

```
screen:  screen_name screen_contents screen_terminator
        | screen_name screen_terminator
        ;
```

所以螢幕結束的規則是：

```
screen_terminator:  END ID
                  | END
                  ;
```

所以螢幕結束的規則是：

```
screen_terminator:  END id { end_screen($2); }
                  | END { end_screen(strdup("default")); }
                  ;
```

利用呼叫 end_screen() 函式來當作動作程式碼，並且傳進螢幕的名稱，假如沒有名稱的話，就把「default」當作螢幕名稱傳進去。在 end_screen() 函式中，我們會檢查螢幕名稱前後有沒有一致。範例 4-10 是 end_screen() 函式的程式碼。

範例 4-10：螢幕結束處理函式

```
/*
 * end_screen:
 * 結束處理螢幕，並且輸出後置處理程式碼。
 */

end_screen(char *name)
{

    fprintf(yyout, " menu_runtime(menu_%s_items); \n",name);
    if(strcmp(current_screen,name) != 0)
    {
        warning("name mismatch at end of screen",
                current_screen);
    }
    fprintf(yyout, "}\n");
    fprintf(yyout, "/* end %s */\n",current_screen);
    process_items( );
    /* write initialization code out to file */
    if(!done_end_init)
    {
        done_end_init = 1;
        dump_data(menu_init);
    }

    current_screen[0] = '\0'; /* 重設目前處理的螢幕 */

    screen_done = 1;

    return 0;
}
```

這個函式並不會將螢幕名稱不對稱的錯誤當成很嚴重，因為螢幕名稱不對稱，對編譯器的執行並不會造成問題，所以我們只是輸出警告訊息給使用者。

這個函式呼叫 `process_items()` 來處理由 `add_item()` 所產生的資料。然後，它再呼叫 `dump_data()` 來產生一些初始化函式。這些初始化函式其實是一個靜態的字串陣列，是原本就建在 MGL 編譯器裡的。我們還會在其他的其他地方呼叫 `dump_data()`，來輸出不同的程式碼到輸出檔案中。另外一種方法，是將範本檔案中的程式碼段落直接複製到輸出檔案，某些版本的 `lex` 與 `yacc` 就是採取這種做法。

最後的後置處理，是在所有的輸入都已經讀過且"完了才進行。在 `yyparse()` 函式之後，我們在 `main()` 函式中呼叫 `end_file()` 來進行後置處理。我們的 `end_file()` 函式如下：

```
/*
 * 這個函式將會輸出動態執行的函式
 */

end_file()
{
    dump_data(menu_runtime);
}
```

這個函式只有呼叫 `dump_data()` 函式，來將執行期 (run-time) 的函式輸出。這些輸出的函式是被儲存在編譯器中的靜態陣列中，和那些初始化的程式碼一樣。我們將用來當作範本的函式設計得相當模組化，以便可以隨時改寫後輸出。

當 `main()` 呼叫這個函式之後，會檢查是否已經讀到輸入的結尾，也就是螢幕敘述的結束。如果是，就不會產生錯誤訊息。

MGL 的範例程式碼

既然已經完成編譯器的大略設計，讓我們來看看如何使用。我們的 MGL 包括三個部分：`yacc` 語法、`lex` 規格、以及支援函式。最後的版本在附錄 1「MGL 編譯器程式碼」。目前還不必將這個範例程式設計的十分完整，我們主要的目標是先弄一個基本的樣子出來。

最後真正完成的編譯器，才是真正的選單產生程式編譯器。以下是一個輸入範例：

```
screen first
title "First"

item "dummy line" command dummy action ignore
    attribute visible

item "run shell" command shell action execute "/bin/sh"
    attribute visible

end first

screen second

title "Second"

item "exit program" command exit action quit
    attribute invisible

item "other menu" command first action menu first
    attribute visible

end second
```

當編譯器處理完如上的描述檔，我們會得到下列的輸出結果：

```
/*
 * Generated by MGL: Thu Aug 27 18:03:33 1992
 */

/* 初始化相關資訊 */
static int init;

#include <curses.h>
#include <sys/signal.h>
#include <ctype.h>
#include "mglyac.h"

/* 用來儲存選項的結構 */
struct item {
```

```
char *desc;
char *cmd;
int action;
char *act_str; /* 要執行的程式名稱 */
int (*act_menu)(); /* 呼叫適當的函式 */
int attribute;
};

/* first 螢幕 */
menu_first()
{
    extern struct item menu_first_items[];
    if(!init) menu_init();

    clear();
    refresh();
    move(0,37);
    addstr("First");
    refresh();
    menu_runtime(menu_first_items);
}

/* first 螢幕結束 */
struct item menu_first_items[]={
{"dummy line","dummy",269,"",0,271},
{"run shell","shell",265,"/bin/sh",0,271},
{(char *)0, (char *)0, 0, (char *)0, 0, 0},
};

menu_init()
{
    void menu_cleanup();
    signal(SIGINT, menu_cleanup);
    initscr();
    crmode();
}

menu_cleanup()
```

```
{
    mvcur(0, COLS - 1, LINES - 1, 0);
    endwin();
}

/* second 螢幕 */
menu_second()
{
    extern struct item menu_second_items[];
    if(!init) menu_init();
    clear();
    refresh();
    move(0,37);
    addstr("Second");
    refresh();
    menu_runtime(menu_second_items);
}

/* second 螢幕結束 */
struct item menu_second_items[]={
{"exit program","exit",268,"",0,272},
{"other menu","first",267,"",menu_first,271},
{(char *)0, (char *)0, 0, (char *)0, 0, 0},
};

/* 執行時的函式 */
menu_runtime(items)
struct item *items;
{
    int visible = 0;
    int choice = 0;
    struct item *ptr;
    char buf[BUFSIZ];

    for(ptr = items; ptr->desc != 0; ptr++) {
        addch('\n');          /* 跳過一行 */
        if(ptr->attribute == VISIBLE) {
            visible++;
        }
    }
}
```

```
        printw("\t%d) %s",visible,ptr->desc);
    }
}

addstr("\n\n\t");          /* 利用移位來使排列更美觀 */
refresh();

for(;;)
{
    int i, nval;
    getstr(buf);
    /* 數字按鍵的輸入 */
    nval = atoi(buf);
    /* 命令列字串的輸入 */
    i = 0;
    for(ptr = items; ptr->desc != 0; ptr++) {
        if(ptr->attribute != VISIBLE)
            continue;

        i++;
        if(nval == i)
            break;

        if(!casecmp(buf, ptr->cmd))
            break;
    }
    if(!ptr->desc)
        continue;          /* 沒有找到對應的選項 */

    switch(ptr->action)
    {
    case QUIT:
        return 0;
    case IGNORE:
        refresh();
        break;
    case EXECUTE:
        refresh();
        system(ptr->act_str);
    }
}
```

```
        break;
    case MENU:
        refresh();
        (*ptr->act_menu)();
        break;
    default:
        printw("default case, no action \n");
        refresh();
        break;
    }
    refresh();
}

casecmp(char *p, char *q)
{
    int pc, qc;
    for(; *p != 0; p++, q++) {
        pc = tolower(*p);
        qc = tolower(*q);
        if(pc != qc)
            break;
    }
    return pc-qc;
}
```

然後，我們利用編譯器產生的 `first.c`，用以下的命令加上 `main()` 函式：

```
$ cat >> first.c
main()
{
    menu_second();
    menu_cleanup();
}
^D
$ cc -o first first.c -lcurses -ltermcap
$
```

如果我們要增加 MGL 的功能，可以將 MGL 修改成：提供一個命令列的的選項，或是在描述檔中提供在 `main()` 函式中呼叫某個函式的功能。不過由於我們是用 `yacc` 來寫語法規則，整個修改相當容易。例如，我們可以將 `screens` 規則改成：

```
screens: /* 空的 */
        | preamble screens screen
        | screens screen
        ;
preamble: START ID
         | START DEFAULT
         ;
```

在這裡，我們加入適當的關鍵字，`START` 與 `DEFAULT`。

當執行 MGL 產生的程式碼，我們在螢幕上可以看到：

```
Second
1) other menu
```

在上面的執行結果，我們看到一個可見的選項。然後，當我們輸入「1」或「first」時，就會跳到 `first` 選單。結果如下：

```
First
1) dummy line
2) run shell
```

習題

1. 在 MGL 加入一個新的指令，指定第一個要顯示的螢幕，以及產生適當的 `main()` 函式，如之前最後一段所描述的功能。
2. 增加螢幕處理的功能，如：在 `CBREAK` 模式下讀取字元，而不是一次一行；允許更簡單的命令輸入方式；例如只要輸入明確的命令開頭字串，就能夠執行命令，不必輸入完整的命令名稱；允許設定選項可見與不可見等等的功能。
3. 使螢幕的描述可以讓標題及命令名稱來自原本程式中的變數。例如：

```
screen sample
title $titlevar
item $label1 command $cmd1 action ignore
    attribute visible
end sample
```

`titlevar` 和 `label1` 就是原本程式中的字元陣列或是指標。

4. (專題) 設計一個語言來處理選單的下拉與縮回。寫一個基本的解析器，根據這個解析器寫出不同版本的轉譯器，可將相同的選單描述在不同的環境下執行。例如：使用 `curses` 的終端機、`Motif` 以及 `Open Look` 等。
5. `yacc` 通常應用在小型與特殊的案例，然後轉譯出較低階、更一般化的語言。在 MGL 的例子中，MGL 將選單描述轉成 C 語言，`eqn` 將數學式語言轉成 `troff`。還有哪些場合可利用這種語言轉換的方式？請利用 `lex` 與 `yacc` 寫出幾種實際的應用。