第四章 菜单生成语言

本章内容：

- MCL 的概述
- 开发 MCL
- 构建 MGL
- 屏幕处理
- 结束
- MGL 代码示例
- 练习

前一章提供了解释程序的简单例子——桌面计算器。本章重点集中在编译程序设计上；开发菜单生成语言（*menu generation laguage,MGL*）和它的编译程序。我们从对要创建的语言的描述开始。然后查看几个开发 lex 和 yacc 规范的迭代过程。最后，创建与语法有关的动作，它们实现 MGL 的特征。

MGL 的概述

我们将开发一种用于生成定制菜单界面的语言。它读取输入描述文件、产生能被编译的 C 程序，该程序能在用户终端上创建输出并使用标准的 *curses* 库在屏幕上绘制菜单[1]。

在许多情况下，当应用程序要求大量冗长的、重复的代码时，设计特殊目的语言和编写将语言翻译成 C 或你的计算机能处理的其他语言的编译程序是比较容易和快速的。curses 程序设计是冗长的，因为必须亲自定位屏幕上所有的数据。MGL 自动进行大部分布局设计，这大大地减轻了工作量。

菜单描述由以下部分组成：

1．菜单屏幕的名字

2．标题

3．菜单项目列表，每个项目又包括以下内容：

项目

[1]要得到有关 curses 的更多的信息，参见 John Strang 编写的《Programming with Curses》，O ’Reilly&Associates 出版。

[命令]

动作

[属性]

项目（item）是出现在菜单上的文本串；命令（command）是对菜单系统函数的记忆码，用于提供命令行访问；动作（action）是当菜单项目被选择时将执行的过程；属性（attribute）指示项目应该如何处理。括号中的项目是可选的。

4．一个终结符

因为有用的应用程序通常有几个菜单，一个描述文件可以包含几个不同的命名菜单。

菜单描述文件的例子：

```
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 编译程序读取这个描述文件并产生 C 代码，C 代码本身必须被编译。当作为结果的程序被执行时，它创建了下面的菜单：

```
   My First Menu
   by Tony Mason

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

当用户按下“ 1 ”或输入命令“ to-do ”时，执行过程“ list-things-todo ”。

这种格式的更一般的描述是：

screen <name>
title <string>

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

当我们开发这种语言时，从这种功能性的一个子集开始，并向这个子集添加特征直到实现完整的规范。这种方式表明了在我们改变语言时修改 lex 产生的词法分析程序和 yacc 产生的语法分析程序是多么容易。

开发 MGL

查看生成前面的语法的设计过程。菜单为缺乏经验的用户提供了简单、清晰的界面。对于这些用户，由菜单系统提供使用的稳定性和简易性是理想的。菜单的主要缺点是，对经验丰富的用户而言，这种方式不能直接进入想要的应用。对于这些用户，命令驱动的界面更合意．然而，几乎大部分经验丰富的用户有时也想运行菜单来访问一些不常用的函数。

很难制定合用的语言。不过，可以为它草拟词法规范：

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

以及相应的 yacc 语法：

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

%token COMMAND
%%
start:    COMMAND
    ;
```

词法分析程序仅仅寻找关键字并且当它识别出一个关键字时返回适当的标记。如果语法分析程序看到标记 **COMMAND**，那么 **start** 规则将被匹配，并且 **yyparse()** 将成功地返回。

菜单上的每个项目都有一个与它相关的动作。我们可以引进关键字 **action**，通过添加关键字 **ignore** 和 **execute**，一个动作可以忽略项目（忽略不能实现的或不可用的命令），另一个动作可以执行程序。

因此，使用这种修改后的词汇表的示例项目如下：

```
command action execute
```

我们必须告诉它执行什么，所以添加第一个非命令参数，即一个字符串。因为程序名可以包含标点符号，所以假定程序名是引用字符串。现在示例项目成为：

```
command action execute “/bin/sh”
```

例 4-l 表明我们可以修改 lex 规范来支持新的关键字，以及新的标记类型。

例 4-1：MGL 词法分析程序的第一种版本

```
ws        [ \t]+
qstring   \”[^\”\n]*[\”\n]\”
nl        \n
%%
{ws}      ;
{qstring} {
          yylval.string = strdup(yytext+1); /* 跳过开引号 */
          if(yylval.string[yyleng-2] != '"')
             warning("Unterminated character string",(char *)0);
          else
             yylval.string[yyleng-2] = '\0'; /* 删除闭引号 */
          return QSTRING;
          }
action    { return ACTION; }
execute   { return EXECUTE; }
command   { return COMMAND; }
ignore    { return IGNORE; }
{nl}      { lineno++;}
.         { return yytext[0]; }
```

qstring 的复杂的定义对于阻止 lex 匹配类似下面的行是有必要的：

```
“apples” and “oranges”
```

模式的“ [^\”\n]* ”部分表明，要匹配不是引号或者换行符的每个字符。我们不想匹配一行以外的字符，因为丢失的闭引号会导致词法分析程序费力通查文件的其他部分（这种结果当然不是用户想要的），而且也许会溢出内部 lex 缓冲区从而使程序失败。使用这种方法，我们可以通过更加“ 礼貌 ”的方式向用户报告错误条件。当我们拷贝字符串时，删除打开的和关闭的引号，因为在代码的其他部分处理没有引号的字符串更容易。

我们还需要修改 yacc 语法（见例 4-2）。

例 4-2：MGL 语法分析程序的第一种版本

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

%union {
   char *string;
}

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

定义一个包括“ 字符串 ”类型和这个新标记 **QSTRING** 的%**union**，这个标记代表一个引用字符串。

需要将单个命令和选项组合的信息归组为一个菜单项（menu item）。使用关键字 **item**，将每个新的项目都作为一个项目来介绍。如果有相关的命令，就使用关键字 **command**。向词法分析程序添加新的关键字：

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

虽然改变了语言的基本结构，但是对词法分析程序的改变很少。这些改变显示在 yacc 语法中，如例 4-3 所示。

例 4-3：具有项目和动作的语法

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

%union {
   char *string;
}

%token COMMAND ACTION IGNORE EXECUTE ITEM
%token <string> QSTRING
%%
item:     ITEM command action
       ;
command:  /* 空值 */
       |  COMMAND
       ;
action:   ACTION IGNORE
       |  ACTION EXECUTE QSTRING
       ;
%%
```

因为每个菜单项不都需要相应的命令，所以 **command** 规则的右侧可以为空。令人惊讶的是，yacc 处理这样的规则不会有麻烦，只要其余的语法使它有可能明确地表明可选的元素不在那里就行。

我们仍然没有给关键字 **command** 任何含义。实际上尝试独自编写 yacc 语法通常是个好主意，因为它可以指示出语言设计中的“ 缺陷 ”。幸运的是，缺陷被很快地补救了。我们把命令限制为字母数字字符串。为词法分析程序的“ 标识符 ”标记添加 **ID**：

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

ID 的值是标识符名字的指针。通常，将指向 **yytext** 的指针作为符号值返回不是个好主意，因为一旦词法分析程序读取下一个标记，它就重写 **yytext**。（没有拷贝 **yytext** 是常见的词法分析程序错误，通常会产生奇怪的故障，字符串和标识符看上去会莫名其妙地改变它们的名字。）使用 **strdup()**复制标记并返回副本的指针。使用 ID 的规则必须仔细，在处理完拷贝时必须释放它。

在例 4-4 中向 yacc 语法添加 **ID** 标记。

例 4-4：具有命令标识符的语法

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

%union {
    char *string;
}

%token COMMAND ACTION IGNORE EXECUTE ITEM
%token <string> QSTRING ID
%%
item:     ITEM command action
       ;
command:  /* 空值 */
       |  COMMAND ID
       ;
action:   ACTION IGNORE
       |  ACTION EXECUTE QSTRING
       ;
%%
```

在一个菜单中，语法不提供多行。为支持多个 **items** 的 **item**，添加一些规则。

```
. . .
%%
items:    /* empty */
       |  items item
       ;
item:     ITEM command action
       ;
. . .
```

与前面所有的规则不同，这些规则依赖于递归。因为 yacc 喜欢左递归的语法，所以编写为“ items item ”的形式而不是右递归的形式“ item items ”（参见第七章的“ 递归规则 ”一节来了解为什么左递归更好）。

items 的一个规则有空的右侧。任何递归规则都必须有终结条件，即非终结符以非递归形式匹配的规则。如果一条规则是没有非递归替代物的完全递归式，那么 yacc 的大部分版本都会由于致命的错误而停止，因为在那些情况下建立一个有效的语法分析程序是不可能的。我们在许多语法中多次使用左侧递归式。

除了能指定菜单中的项目以外，在菜单的顶部还有一个标题。下面是描述一个标题的语法规则：

```
title:       TITLE     QSTRING
       ;
```

关键字 **title** 引进了一个标题。要求标题是一个引用字符串。给 lex 规范添加新的标记 TITLE：

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

要想有多个标题行。附加的语法是：

```
titles:   /* empty */
       |  titles title
       ;
title:    TITLE     QSTRING
       ;
```

递归定义允许多个标题行。

标题行的增加意味着必须添加一个由项目或标题组成的新的、高级的规则。标题在项目的前面，所以例 4-5 在语法中添加了一个新的规则 **start**。

例 4-5：具有标题的语法

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

%union {
   char *string;
}

%token COMMAND ACTION IGNORE EXECUTE ITEM TITLE
%token <string> QSTRING ID
%%
start:    titles items
       ;
titles:   /* 空值 */
       |  titles title
       ;
```

```
items:    /* 空值 */
       |  items item
       ;
item:     ITEM command action
       ;
command:  /* 空值 */
       |  COMMAND ID
       ;
action:   ACTION IGNORE
       |  ACTION EXECUTE QSTRING
       ;
%%
```

在少量使用 MGL 之后，我们发现一个菜单屏幕不够用。想要多个屏慕并能从一个屏幕中引用另一个屏幕，即允许多级菜单。

定义一条新的屏幕（screen）规则来包含具有标题和项目的完整的菜单。为了添加多个屏幕的操作，可以使用递归规则构建一个新的 **screens** 规则。要想允许空屏幕，添加 5 条新规则集：

```
screens:  /* 空值 */
       | 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
       ;
```

每个屏幕都有一个惟一的名字。当希望引用一个特殊的菜单屏幕时，比方说“ first ”，我们可以使用下面的形式：

```
item “first” command first action menu first
```

命名屏幕时，还必须指示屏幕何时结束，所以需要 **screen_terminator** 规则。因此，示例的屏幕规范也许如下所示：

```
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”
iten “main” command main action menu main
end grains
```

规则能支持没给出名字的情况；因此，**screen_name** 和 **screen_terminator** 具有两种情况，实际上当编写特定规则的动作时，我们将检测名字的一致性，来检查菜单描述缓冲区中不一致的编辑错误。

进一步考虑之后，决定向菜单生成语言添加多个特征。对于每个项目，希望指示它是否可见。因为它是单个项目的一个属性，用新的关键 **attribute** 来处理可见或不可见的选项。下面是描述一个属性的新语法的一部分：

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

允许空的属性字段接受一个默认的、也许可见的属性。例 4-6 是可工作的语法。

例 4-6：完整的 MGL 语法

```
screens:  /* 空值 */
       | 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
       ;

titles: /* 空值 */
       | titles title
       ;

title: TITLE QSTRING
       ;

lines: line
       | lines line
       ;

line: ITEM QSTRING command ACTION action attribute
       ;

command: /* 空值 */
       | COMMAND ID
       ;

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

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

我们已经用 **screes** 规则取代前面示例的 **start** 规则作为最高层的规则。如果在描述部分没有%**start** 行，那么它就使用第一条规则。

构建 MGL

现在，有了一个基本语法，然后开始构建编译程序的工作。首先，必须完成词法分析程序的修改，以处理上一次语法修改中引进的新关键字。修改的 lex 规范如例 4-7 所示。

例 4-7：MGL lex 规范

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

%%

{ws}      ;
{comment} ;
{qstring} { yylval.string = strdup(yytext+1); /* 跳过开括号 */
          if(yylval.string[yyleng-2] != '"')
             warning("Unterminated character string",(char *)0);
          else
             yylval.string[yyleng-2] = '\0'; /* 删除闭括号*/
          return QSTRING;
          }
screen    { return SCREEN; }
title     { return TITLE; }
item      { return ITEM; }
command   { return COMMAND; }
action    { return ACTION; }
execute   { return EXECUTE; }
menu      { return MENU; }
quit      { return QUIT; }
ignore    { return IGNORE; }
attribute { return ATTRIBUTE; }
visible   { return VISIBLE; }
invisible { return INVISIBLE; }
end       { return END; }
{id}      { yylval.string = strdup(yytext);
             return ID;
          }
{nl}      { lineno++; }
.         { return yytext[0]; }
%%
```

处理关键字的另一种方法如例 4-8 所示。

例 4-8：另一种 lex 规范

```
. . .
id     [a-zA-Z][a-zA-Z0-9]*
%%
. . .
{id}   {   if(yylval.cmd = keywork(yytext))
               return yylval.cmd;
           yylval.string = strdup(yytext);
           return ID;
       }
%%
/*
 * keyword: 检查一个文本字符串是否是一个合法的关键字，
 * 如果是，返回关键字 keyword 的值
 * 否则返回 0。注意：标记值必须非零
 */

static struct keyword {
   char *name;   /* 文本字符串 */
   int value;    /* 标记 */
} keyword[] =
{
"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(char *string)
{
   struct keyword *ptr = keywords;
```

```
   while(ptr->name != NULL)
      if(strcmp(ptr->name, string) == 0)
         return ptr->value;
      else
         ptr++;
   return 0; /* 不匹配 */
}
```

例 4-8 中的实现使用静态表来标识关键字。这种形式总是比较慢，因为 lex 词法分析程序的速度不取决于模式的数量或复杂性。在这里介绍它只是因为它展示了一种有用的技术，可以在需要扩展语言词汇时使用。那样的话，可以为所有的关键字应用一个查找机制，并且在必要时向表中添加新的关键字。

逻辑上，将处理 MGL 规范文件的工作分成几个部分：

初始化	初始化所有的内部数据表，给出生成的代码需要的任何前同步码。
屏幕开始处理	建立新的屏幕表条目，向名字列表添加屏幕名，并给出初始的屏幕代码。
屏幕处理	当遇到每个独立的项目时，处理它；当看到标题行时，向标题列表中添加它们，并且当看到新的菜单项目时，向项目列表添加它们。
屏幕结束处理	当看到 **end** 语句时，处理读取屏幕描述时构建的数据结构并给出屏幕的代码。
结束	“清除”内部语句，给出任何最终的代码，并确保正常结束；如果有问题，就向用户报告问题

初始化

当任何编译程序开始工作时必须执行一些工作。例如，内部数据结构必须被初始化，回忆一下例 4-8 中的使用关键字查找模式而不是例 4-7 中使用的硬编码的关键字识别模式的情况。在具有作为初始化的部分的符号表的更复杂的应用程序中，应该像例 3-10 那样将关键字插入符号表中。

主函数 **main()**例程简单地这样开始：

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

```
}
```

还必须能通过给出一个文件名来调用编译程序。因为 lex 读取 **yyin** 并向 **yyout** 写入，而它们默认地被赋值为标准输入和标准输出，我们可以将它们重新绑定(attach)输入和输出文件以获取适当的动作。为了改变输入或输出，使用 *fopen()* 从标准 I/O 库中打开想要的文件并将结果赋给 **yyin** 或 **yyout**。

如果用户调用没有参数的程序，那么我们写出到一个默认文件 *screen.out*，并从标准输入 *stdin* 中读取。如果用户调用具有一个参数的程序，那么我们仍然写入 *srceen.out* 并将命名的文件用做输入。如果用户调用具有两个参数的程序，那么用第一个参数作为输入文件，第二参数作为输出文件。

从 **yyparse()**返回后，执行后处理过程，然后检查以确保在没有错误的条件下结束。然后清除并退出。

例 4-9 展示最终结果 **main()**例程.

例 4-9：MGL **main()**例程

```
char *progname = "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;

   progname = argv[0];

   if(argc > 3)
   {
      fprintf(stderr,usage, progname);
      exit(1);
   }
   if(argc > 1)
   {
      infile = argv[1];
      /* open for read */
      yyin = fopen(infile,"r");
      if(yyin == NULL) /* 打开失败 */
```

```
      {
         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) /* 打开失败 */
   {
      fprintf(stderr,"%s: cannot open %s\n",
         progname, outfile);
      exit(1);
   }

   /* 通常与 yyin 和 yyout 的交互从这里开始 */

   yyparse();

   end_file(); /* 写出最终信息 */

   /* 现在检查 EOF 条件 */
   if(!screen_done) /* 位于屏幕中心 */
   {
      warning("Premature EOF",(char *)0);
      unlink(outfile); /* 删除坏文件 */
      exit(1);
   }
   exit(0); /* 没有错误 */
}

warning(char *s, char *t) /* 显示警告信息 */
{
   fprintf(stderr, "%s: %s", progname, s);
   if (t)
```

```
       fprintf(stderr, " %s", t);
   fprintf(stderr, " line %d\n", lineno);
}
```

屏幕处理

一旦初始化编译程序并打开文件，就进入菜单生成程序的真正工作——处理菜单描述。第一个规则 **screens** 要求没有动作。**screen** 规则分解成 **screen_uame**、**screen_contents** 和 **screen_terminator**。**screen_name** 首先引起我们的注意：

```
screen_name: SCREEN ID
          |  SCREEN
          ;
```

在名字副本中插入特定的名字；在没有指定名字的情况下，使用名字“ default ”。规则如下：

```
screen_name: SCREEN ID { start_screen($2); }
          |  SCREEN    { start_screen(strdup(“default”)); }
          ;
```

（我们需要调用 **strdup()**以与第一条规则相一致．第一条规则传递由词法分析程序动态分配的字符串。）**start_screen** 例程将名字输入屏幕列表并开始产生代码。

例如，如果输入文件出现“ screen first ”，那么 **start_screen** 例程将产生下列代码：

```
/* screen first */
menu_first()
{
   extern struct item menu_first_items[];

   if(!init) menu_init();

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

当处理菜单规范时，下一个对象是标题行：

```
title: TITLE QSTRING
       ;
```

调用 **add_title()**，它计算标题行的位置：

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

标题行的示例输出如下：

```
move(0, 37);
addstr(“First”);
refresh();
```

通过定位光标并打印出指定的引用字符串（使用基本的居中对齐）来添加标题行。这个代码能重复用于遇到的每个标题行，惟一的改变是用于产生 **move()**调用的行数。

为了论证这件事，生成一个具有额外标题行的菜单描述：

```
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
```

输出的标题行如下；

```
move(0, 37);
addstr(“First”);
refresh();
move(1, 32);
addstr(“Copyright 1992”);
refresh();
```

一旦看到项目行列表，就获取独立的条目并构建相关动作的内部表。一直继续直到看到语句 **end first**，执行后处理并结束构建屏幕。为了构建菜单项目表，向 **item**

规则中添加下列动作：

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

command、**action** 和 **attribute** 的规则主要存储后面使用的标记值：

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

command 可以为空或一个特殊的命令。在第一种情况下，保存了命令名（采用 **strdup()**与下一条规则保持一致）的空字符串，而在第二种情况下，在 **cmd_str** 中保存命令的标识符。

action 规则和相关的动作更复杂，部分是由于大量可能的变化：

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

最后，**attribute** 规则比较简单，因为惟一的语义值是由标记呈现的：

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

action 规则和 **attribute** 规则的返回值将传递给 **add_line** 例程；这个调用以不同静态字符串指针的内容，加上两个返回值作为参数，在内部状态表中创建一个条目。

根据看到的 **end first** 语句，我们必须进行屏幕的最终处理。从示例输出中，我们完成 **menu_first** 例程：

```
   menu_runtime(menu_first_items);
```

实际的菜单项目写进 **menu_first_items** 数组中：

```
struct item menu_first_items[] = {
   {“first”, “first”, 271, “”, 273},
   {“second”, “second”, 267, “/bin/sh”, 0, 273},
   {(char *)0, (char *)0, 0, (char *)0, 0, 0}
};
```

运行时例程 **menu_runtime** 将显示独立的项目，它包含在生成的文件中，作为结尾的部分代码。

结束

处理单个屏幕的最后阶段是看到屏幕的结束。回忆 **screen** 规则：

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

这个语法期待看到 **screen_terminator** 规则：

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

添加对后处理程序的调用进行屏幕结束的后处理（不是本节后面讨论的文件结束的后处理）。作为结束的规则如下：

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

它使用屏幕名字作为参数调用 **end_screen** 例程，或者如果没有名字就用“default”。这个例程验证屏幕名。例 4-10 展示了实现它的代码。

例 4-10：屏幕结束代码

```
/*
 *
/*
 * end_screen:
 * 结束屏幕. 打印出后同步信号
 */

end_screen(char *name)
{

   fprintf(yyout, "\tmenu_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();

   /* 写出文件的初始化代码 */
   if(!done_end_init)
   {
      done_end_init = 1;
      dump_data(menu_init);
   }

   current_screen[0] = '\0';  /* 没有当前屏幕 */

   screen_done = 1;

   return 0;
}
```

这个例程将屏幕名不匹配作为非致命错误处理。因为错误匹配不会导致编译程序内部的问题，所以可以报告给用户而不必结束。

这个例程通过使用 **process_items()**处理独立的项目条目来处理由 **add_item**()调用产生的数据。然后，它调用 **dump_data** 来写出一些初始化例程，这些初始化例程是真正的构成 MGL 编译程序的字符串的静态数组。在几个不同位置调用 **dump_data()**以转储不同的代码段到输出文件。另一种可替代的途径是从包含样

板（boiler-plate）代码的“ 框架 ”文件中拷贝这些代码段，正如 lex 和 yacc 的一些版本所做的那样。

后处理发生在读取和分析所有的输入之后。在成功地完成 **yyparse()**之后，通过调用 **end_file()**由 **main()**例程来进行后处理。实现为：

```
/*
 * 这个例程写出运行时的支持
 */

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

这个例程包含对 **dump_data()**的单个调用来写出运行时例程，它像初始化代码那样存储在编译程序中的静态数组中。处理样板代码的所有例程在设计中充分地模块化，这些模块可以重新编写为使用框架文件。

一旦这个例程被调用，**main** 例程通过检查以决定输入的结尾是否位于有效点上来结束，即在屏幕的边界，并且，没有产生错误消息。

MGL 代码示例

我们已经构建了一个简单的编译程序，现在来论证一下它的基本工作。MGL 的实现由三部分组成：yacc 语法、lex 规范和支持代码例程。yacc 语法的最终形式、lex 词法分析程序和支持代码展示在附录九“ MGL 编译程序代码 ”中。

我们不是要设计真正使用中的非常健壮的示例代码.而主要集中在开发第一阶段的实现。然而，作为结果的编译程序将产生一个完全的功能性菜单编译程序。下面是示例输入文件：

```
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
```

当描述文件被编译程序处理时，会得到下面的输出文件：

```
/*
 * Generated by MGL: Sat Jan  3 18:18:42 2009
 */

/* initialization information */
static int init;

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

/* structure used to store menu items */
struct item {
   char *desc;
   char *cmd;
   int  action;
   char   *act_str;    /* execute string */
   int (*act_menu)();   /* call appropriate function */
   int  attribute;
};

/* screen first */
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);
```

```
}
/* end first */
struct item menu_first_items[]={
{"dummy line","dummy",270,"",0,272},
{"run shell","shell",266,"/bin/sh",0,272},
{(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();
}

/* screen second */
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);
}
/* end second */
struct item menu_second_items[]={
{"exit program","exit",269,"",0,273},
{"other menu","first",268,"",menu_first,272},
{(char *)0, (char *)0, 0, (char *)0, 0, 0},
};
```

```
/* runtime */

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'); /* skip a line */
      if(ptr->attribute == VISIBLE) {
         visible++;
         printw("\t%d) %s",visible,ptr->desc);
      }
   }

   addstr("\n\n\t"); /* tab out so it looks nice */
   refresh();

   for(;;)
   {
      int i, nval;

      getstr(buf);

      /* numeric choice? */
      nval = atoi(buf);

      /* command choice ? */
      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; /* no match */

        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;
}
```

依次，编译由编译程序产生的代码并写到 *first.c*，使用下列命令：

```
$ cat >> first.c
main()
```

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

必须向产生的代码添加 **main()**例程；在 MGL 的修订本中，它也许包括在主循环中，用于提供调用的例程名的命令行选项或规范选项。因为使用 yacc 编写语法，所以修订很容易。例如，可以修订 **screens** 规则为：

```
screens:  /* nothing */
       | preamble screens screen
       | screens screen
       ;
preamble: START ID
       | START DEFAULT
       ;
```

为 **START** 和 **DEFAULT** 添加适当的关键字。

编译运行 MGL 产生的屏幕代码，我们看到下列菜单屏幕：

```
                Second
1) other menu
```

我们看到一个可见的菜单条目，并且当输入“ l ”或“ f 定 rs、”时移到第一个菜单：

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

练习

1. 添加命令标识主屏幕并产生一个主例程，如前所述。

2. 改善屏幕处理：用 CBREAK 模式读取字符而不是每次一行，允许更灵活的命令处理，例如，接受惟一的命令前缀而不要求整个命令，允许应用代码设置和清除不可见的属性，等等。

3. 扩展屏幕定义语言，让标题和命令名从主程序中的变量产生，例如：

```
screen sample
title $titlevar

item $label1 command $cmd1 action ignore
   attribute visible

end sample
```

其中，**titlevar** 和 **label1** 是主程序中的字符数组或指针。

4. （术语工程）设计一种指定下拉式或弹出式菜单的语言。实现几种基于同一语法分析程序的翻译程序，这样可以使用同一菜单规范创建在不同环境中运行的菜单，例如具有 curses、Motif 和 openLook 的终端。

5. yacc 通常用于实现对于一个应用领域特定的“ 小语言 ”，并将它翻译成低级的、更一般的语言。MGL 将菜单规范转换成 C 语言，*eqn* 将方程式语言转换成原始的编辑和格式化程序。可以从小语言中获得好处的其他应用领域有哪些？用 lex 和 yacc 设计和实现几个。