

中国矿业大学计算机学院

系统软件开发实践报告

课程名称 系统软件开发实践

报告时间 2022.3.12

学生姓名 王杰永

学 号 03190886

专 业 计算机科学与技术

任课教师 张博

成绩考核

编号	课程教学目标	占比	得分
1	目标 1: 针对编译器中词法分析器软件要求，能够分析系统需求，并采用 FLEX 脚本语言描述单词结构。	15%	
2	目标 2: 针对编译器中语法分析器软件要求，能够分析系统需求，并采用 Bison 脚本语言描述语法结构。	15%	
3	目标 3: 针对计算器需求描述，采用 Flex/Bison 设计实现高级解释器，进行系统设计，形成结构化设计方案。	30%	
4	目标 4: 针对编译器软件前端与后端的需求描述，采用软件工程进行系统分析、设计和实现，形成工程方案。	30%	
5	目标 5: 培养独立解决问题的能力，理解并遵守计算机职业道德和规范，具有良好的法律意识、社会公德和社会责任感。	10%	
总成绩			
指导教师		评阅日期	

目录

二、 综合实验 2.....	1
1 实验目的.....	1
2 实验内容.....	1
3 实验步骤.....	1
3.1 Windows 系统下进行实验.....	1
3.2 Linux 系统下进行试验	2
4 源码分析.....	2
4.1 fb3-2.y 源码分析	3
4.2 fb3-2.l 源码分析	5
4.3 fb3-2.h 源码分析	6
4.4 fb3-2.funcs.c 源码分析	8
5 抽象语法树的建立、遍历与计算.....	9
5.1 基本节点 ast.....	9
5.2 数字节点 numval.....	10
5.3 比较运算节点 ast.....	10
5.4 标识符节点 symref.....	11
5.5 赋值节点 symasgn.....	11
5.6 内置函数节点 fncall.....	12
5.7 自定义函数节点 ufncall.....	12
5.8 创建自定义函数.....	13
5.9 表达式求值	14
6 Debug 模式	15
7 增加函数 pow 及其重载函数的详细过程分析.....	15
7.1 二操作数 pow 函数代码改动.....	15
7.2 二操作数 pow 的计算逻辑.....	16
7.3 二操作数 pow 函数抽象语法树的建立过程.....	17
7.4 二操作数 pow 的计算结果.....	18
7.5 三操作数 pow 函数.....	18
8 实验总结.....	20

二、综合实验 2

1 实验目的

阅读《flex&Bison》第三章。使用 flex 和 Bison 开发一个具有全部功能的计算器，包括如下功能：

- 1) 支持变量；
- 2) 实现复制功能；
- 3) 实现比较表达式（大于小于等）；
- 4) 实现 if/then/else 和 do/while 流程控制；
- 5) 用户可以自定义函数；
- 6) 简单的错误恢复机制。

2 实验内容

- 5)阅读《Flex&Bison》第三章 P60~P79，学习抽象语法树；
- 6)阅读 fb3-2.y、fb3-2.l、 fb3-2funcs.c、fb3-2.h；
- 7)使用内置函数 sqrt(n)、 exp(n) ， log(n)
- 8)定义函数 sq(n)、 avg(a, b)，用于计算平方根；
- 9)撰写实验报告，结合实验结果，给出抽象语法树的构建过程。。

3 实验步骤

有了上一次对于 atof 函数的错误处理策略，本次实验在 Windows 与 linux 系统下都进行的较为顺利。

3.1 Windows 系统下进行实验

编译 Flex 与 Bison 文件后，使用 gcc 编译生成的全部 c 文件，得到可执行程序。实验顺利，结果截图如下。

```

D:\Here\recently\系统软件开发实践\综合实验\3-2\code>fb3-2.tab.exe
> sqrt(10)
= 3.162
> exp(2)
= 7.389
> log(7.389)
= 2
> let sq(n) = e = 1; while | ((t = n / e) - e) > .001 do e = avg(e, t);;
Defined sq
> let avg(a, b) = (a + b) / 2;
Defined avg
> sq(10)
= 3.162
> sq(10) - sqrt(10)
= 0.000178
>

```

图 2-1 Windows 系统下的运行结果

3.2 Linux 系统下进行试验

Linux 系统下同理，使用 gcc 编译 flex 与 bison 生成的 c 文件，计算器成功运行。结果如下图。

```

chen@chen:~/桌面/chen/e6$ ^C
chen@chen:~/桌面/chen/e6$ bison -d fb3-2.y
chen@chen:~/桌面/chen/e6$ flex -offb3-2.lex.c fb3-2.l
chen@chen:~/桌面/chen/e6$ gcc fb3-2.tab.c ffb3-2.lex.c fb3-2funcs.c -o fb3-2 -lmfb3-2.tab.c: In function 'yyparse':
fb3-2.tab.c:1184:16: warning: implicit declaration of function 'yylex' [-Wimplicit-function-declaration]
    yychar = yylex ();
                  ^~~~~
fb3-2funcs.c: In function 'main':
fb3-2funcs.c:438:10: warning: implicit declaration of function 'yyparse' [-Wimplicit-function-declaration]
    return yyparse();
           ^~~~~~
chen@chen:~/桌面/chen/e6$ ./fb3-2
> sqrt(10)
= 3.162
> exp(2)
= 7.389
> log(7.389)
= 2
> let sq(n) = e = 1; while | ((t = n / e) - e) > .001 do e = avg(e, t);;
Defined sq
> let avg(a, b) = (a + b) / 2;
Defined avg
> sq(10)
= 3.162
> sq(10) - sqrt(10)
= 0.000178
>

```

图 2-2 Linux 系统下的运行结果

4 源码分析

本次实验相较于上次，源码在代码量上增加了很多，同时计算器的功能也更为复杂。以下对源码的分析主要围绕如何在已有内置函数 `sqrt`、`exp`、`log`、`print` 的基础上，增加一个新的内置函数 `pow` 及其重载形式展开。

4.1 fb3-2.y 源码分析

首先分析 Yacc 文件。分析器使用 `%union` 来声明语法分析器中符号值的类型。查阅参考书得知，在 `bison` 语法分析器中，每一个语法符号（不论是终结符和非终结符），都可以有一个相应的值。默认情况下这个值是整型，但对于有着更多功能的语法分析器而言，一个整型数字显然不够，因此定义了 `%union` 联合体来记录语法符号的值。

```
1. %union {
2.     struct ast *a;
3.     double d;
4.     struct symbol *s;          /* which symbol */
5.     struct symlist *sl;
6.     int fn;                   /* which function */
7. }
```

当定义了 `union` 类型后，一个很直接的问题是，我们需要告知 `bison` 每个语法符号使用的是 `union` 中的哪种类型的值。查阅参考书得知，声明 `token` 时，在 `token` 名前使用 `<>`，尖括号中写入对应的类型即可。

```
1. /* declare tokens */
2. %token <d> NUMBER
3. %token <s> NAME
4. %token <fn> FUNC
5. %token EOL
```

例如，计算器的 `NUMBER` 语法符号使用 `union` 中的类型 `d`，即 `double` 类型；而 `FUNC` 语法符号使用 `union` 中的 `fn`，即表明内置函数类型的整型数。这里也启发我，如果要新增一个自定义函数的话，需要将为 `fn` 引入新的取值。

最后，计算器的语法规则及相应的语法制导翻译模式如下：

```
1. stmt: IF exp THEN list          { $$ = newflow('I', $2, $4, NULL); }
2.     | IF exp THEN list ELSE list { $$ = newflow('I', $2, $4, $6); }
3.     | WHILE exp DO list         { $$ = newflow('W', $2, $4, NULL); }
4.     | exp
5. ;
6.
7. list: /* nothing */ { $$ = NULL; }
8.     | stmt ';' list { if ($3 == NULL)
9.                         $$ = $1;
10.                    else
```

```

11.          $$ = newast('L', $1, $3);
12.          }
13.      ;
14.
15. exp: exp CMP exp      { $$ = newcmp($2, $1, $3); }
16.   | exp '+' exp      { $$ = newast('+', $1,$3); }
17.   | exp '-' exp      { $$ = newast('-', $1,$3); }
18.   | exp '*' exp      { $$ = newast('*', $1,$3); }
19.   | exp '/' exp      { $$ = newast('/', $1,$3); }
20.   | '|' exp          { $$ = newast('|', $2, NULL); }
21.   | '(' exp ')'      { $$ = $2; }
22.   | '-' exp %prec UMINUS { $$ = newast('M', $2, NULL); }
23.   | NUMBER          { $$ = newnum($1); }
24.   | FUNC '(' explist ')' { $$ = newfunc($1, $3); }
25.   | NAME            { $$ = newref($1); }
26.   | NAME '=' exp     { $$ = newasgn($1, $3); }
27.   | NAME '(' explist ')' { $$ = newcall($1, $3); }
28. ;
29.
30. explist: exp
31.   | exp ',' explist { $$ = newast('L', $1, $3); }
32. ;
33. symlist: NAME      { $$ = newsymlist($1, NULL); }
34.   | NAME ',' symlist { $$ = newsymlist($1, $3); }
35. ;
36.
37. calclist: /* nothing */
38.   | calclist stmt EOL {
39.       if(debug) dumpast($2, 0);
40.       printf("= %4.4g\n> ", eval($2));
41.       treefree($2);
42.   }
43.   | calclist LET NAME '(' symlist ')' '=' list EOL {
44.       dodef($3, $5, $8);
45.       printf("Defined %s\n> ", $3->name); }
46.
47.   | calclist error EOL { yyerrok; printf("> "); }
48. ;

```

可以看出，语法分析的产生式规则较为简单。值得注意的是，在行 39 出现了关键字"debug"，由此可以看出该计算器具有 debug 功能。

4.2 fb3-2.l 源码分析

对于词法分析的 `flex` 文件，通过前面若干次实验，已经比较熟悉了。

值得注意的一些规则如下：

```
1. /* comparison ops */
2. ">"      { yylval.fn = 1; return CMP; }
3. "<"      { yylval.fn = 2; return CMP; }
4. "<>"     { yylval.fn = 3; return CMP; }
5. "=="     { yylval.fn = 4; return CMP; }
6. ">="     { yylval.fn = 5; return CMP; }
7. "<="     { yylval.fn = 6; return CMP; }
8.
9. /* built in functions */
10. "sqrt"   { yylval.fn = B_sqrt; return FUNC; }
11. "exp"    { yylval.fn = B_exp; return FUNC; }
12. "log"    { yylval.fn = B_log; return FUNC; }
13. "print"  { yylval.fn = B_print; return FUNC; }
```

从这里可以看出，无论匹配到的 `token` 是 `CMP` 还是 `FUNC`，都使用 `union` 中的 `fn` 作为区别。可以推断出，`fn` 的值不能唯一确定当前的语法符号，语法制导翻译会通过 `fn` 与 `token` 类型共同确定。

在上述代码段中还可以看到，内置函数 `sqrt`、`exp` 等的词法规则为 `union` 的 `fn` 赋值 `B_sqrt`、`B_exp`。在 `fb3-2.h` 文件中可以找到这些变量的定义。

```
1. enum bifs {          /* built-in functions */
2.   B_sqrt = 1,
3.   B_exp,
4.   B_log,
5.   B_print
6. };
```

这些变量是枚举变量的常数值。这样的编码风格使得程序的可读性强，容易维护。同时，如果我们想增加内置函数的话，需要在 `bifs` 枚举类型下增加新的枚举值。

还有这样几个词法匹配模式值得我们去注意：

```
1. "//".*
2. [\t] /* ignore white space */
```

这两条词法模式匹配规则，由于其后没有任何动作，因此计算器允许我们输入 `"//"` 作

为注释或是任意空格、制表符来“美化”输入。

```
1. \\n    printf("c> "); /* ignore line continuation */
2. "\n"   { return EOL; }
```

最后一条规则很简单，当我们输入换行符时，返回一个语法分析的结束符号 EOL 以此结束当前表达式，从而求值。

"\\n"规则则允许我们"换行继续输入"，这里的反斜杠是转义符的含意，当匹配到反斜杠且其后紧跟换行符时，当前表达式的计算不会停止，词法分析器仅输出一个"c>"符号以此表示该行输入紧跟上一行。即，该计算器具有分多行输入的功能。

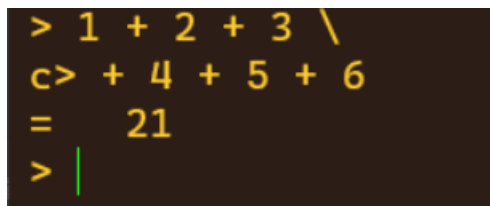


图 2-3 计算器具有多行输入功能

最后，还有着 debug 模式的词法匹配规则。规则如下。

```
1. /* debug hack */
2. "debug"[0-9]+ { debug = atoi(&yytext[5]); printf("debug set to %d\n",
    debug); }
```

从这里也能看出，当输入 debug+ 数字的时候，会进入到 debug 模式。将在 xxxxxxxx 部分演示。

4.3 fb3-2.h 源码分析

与上一次实验一样，由于词法/语法规则足够多，因此将全部的辅助 c 代码放于另外一组文件中——fb3-2.h 与 fb3-2funcs.c。

在 fb3-2.h 头文件中，定义了各类全局变量与不同语法符号的结构体（抽象语法树节点）信息。

◆ 符号表

```
1. /* symbol table */
2. struct symbol {          /* a variable name */
3.     char *name;
4.     double value;
5.     struct ast *func;    /* stmt for the function */
```

```

6.  struct symlist *syms; /* list of dummy args */
7.  };
8.
9.  /* simple symtab of fixed size */
10. #define NHASH 9997
11. struct symbol symtab[NHASH];

```

符号表使用结构体定义。每一个符号都可以有一个变量和一个用户自定义函数。
value 域用来保存符号值，**func** 域指向用抽象语法树表示的该函数用户代码。

◆ 各种不同类型的节点

```

1. struct ast {
2.     int nodetype;
3.     struct ast *l;
4.     struct ast *r;
5. };
6. struct fncall {          /* built-in function */
7.     int nodetype;        /* type F */
8.     struct ast *l;
9.     enum bifs functype;
10. };
11. struct ufncall {        /* user function */
12.     int nodetype;        /* type C */
13.     struct ast *l;       /* list of arguments */
14.     struct symbol *s;
15. };
16. struct flow {
17.     int nodetype;        /* type I or W */
18.     struct ast *cond;    /* condition */
19.     struct ast *tl;      /* then or do list */
20.     struct ast *el;      /* optional else list */
21. };
22. struct numval {
23.     int nodetype;        /* type K */
24.     double number;
25. };
26. struct symref {
27.     int nodetype;        /* type N */
28.     struct symbol *s;
29. };
30. struct symasgn {
31.     int nodetype;        /* type = */
32.     struct symbol *s;

```

```

33.  struct ast *v;      /* value */
34. };

```

计算器的功能相较于上一次实验增多了很多，抽象语法树节点类型的增多是最直接的体现。

以上是各类节点的定义。每一个节点都有一个 **nodetype**，遍历树的代码使用这个变量来判断当前访问的节点类型——**nodetype** 的取值及含意在源码的注释中已经全部表述。

```

1.  /* nodes in the Abstract Syntax Tree */
2.  /* all have common initial nodetype */
3.
4.  /* node types
5.   * + - * / |
6.   * 0-7 comparison ops, bit coded 04 equal, 02 less, 01 greater
7.   * M unary minus
8.   * L statement list
9.   * I IF statement
10.  * W WHILE statement
11.  * N symbol ref
12.  * = assignment
13.  * S list of symbols
14.  * F built in function call
15.  * C user function call
16.  */

```

通过分析源码可以发现，内置函数使用 **fncall** 节点，流程控制表达式使用 **flow** 节点，常量使用 **numval** 节点，符号引用使用 **symref** 节点。因此，若想要增加一个自定义的内置函数，在语法制导翻译时，需要使用 **ufncall** 节点类型。

4.4 fb3-2.funcs.c 源码分析

该文件中是对 **fb3-2.h** 头文件中各个已经声明的函数的实现。包括抽象语法树的构造函数——**newast()**, **newcmp()**, **newfunc()**等，抽象语法树的求值函数 **eval()**，定义新的内置函数的函数 **dodef()**，释放语法树所占空间的函数 **treefree()**，以及 **debug** 模式下起作用的 **dumpast()**函数。

该部分源码定义了抽象语法树的建立逻辑，代码量较大。在**部分 5 抽象语法树的建立**中详细描述。

5 抽象语法树的建立、遍历与计算

在 4.3 中已经提到，该桌面计算器的源码中定义了多种类型的节点结构，根据归约出的不同语法符号，会创建不同类型的节点。

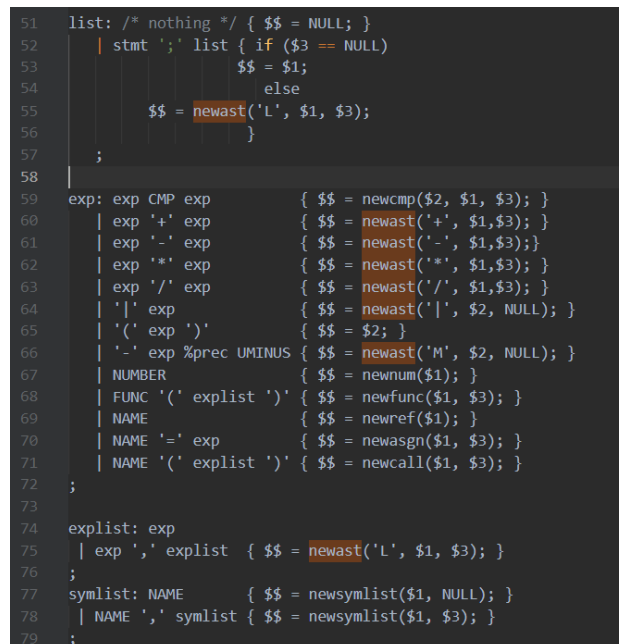
5.1 基本节点 ast

创建基本节点 ast 的函数 `newast()` 的源码如下。

```

1. struct ast *
2. newast(int nodetype, struct ast *l, struct ast *r)
3. {
4.     struct ast *a = malloc(sizeof(struct ast));
5.
6.     if(!a) {
7.         yyerror("out of space");
8.         exit(0);
9.     }
10.    a->nodetype = nodetype;
11.    a->l = l;
12.    a->r = r;
13.    return a;
14. }
```

可以看出，ast 节点有左右两棵子树，因此推断二元运算符才会创建 ast 节点。我们在语法分析的.y 文件中查找 `newast` 函数，得到如下结果。



```

51 list: /* nothing */ { $$ = NULL; }
52 | stmt ';' list { if ($3 == NULL)
53     $$ = $1;
54     else
55     $$ = newast('L', $1, $3);
56 }
57 ;
58
59 exp: exp CMP exp { $$ = newcmp($2, $1, $3); }
60 | exp '+' exp { $$ = newast('+', $1,$3); }
61 | exp '-' exp { $$ = newast('-', $1,$3); }
62 | exp '*' exp { $$ = newast('*', $1,$3); }
63 | exp '/' exp { $$ = newast('/', $1,$3); }
64 | '|' exp { $$ = newast('|', $2, NULL); }
65 | '(' exp ')' { $$ = $2; }
66 | '-' exp %prec UMINUS { $$ = newast('M', $2, NULL); }
67 | NUMBER { $$ = newnum($1); }
68 | FUNC '(' explist ')' { $$ = newfunc($1, $3); }
69 | NAME { $$ = newref($1); }
70 | NAME '=' exp { $$ = newasgn($1, $3); }
71 | NAME '(' explist ')' { $$ = newcall($1, $3); }
72 ;
73
74 explist: exp
75 | exp ',' explist { $$ = newast('L', $1, $3); }
76 ;
77 symlist: NAME { $$ = newsymlist($1, NULL); }
78 | NAME ',' symlist { $$ = newsymlist($1, $3); }
79 ;
```

图 2-4 语法分析中查找 `newast` 函数

对于基本运算符"+-*/"以及"|"运算符，均以左右两表达式的值\$1、\$3 来创建 ast 节点。同时，在图 2-4 的行 55 可以看出，整棵抽象语法树的根节点类型也为 ast。

5.2 数字节点 numval

```
1. struct ast *
2. newnum(double d)
3. {
4.     struct numval *a = malloc(sizeof(struct numval));
5.
6.     if(!a) {
7.         yyerror("out of space");
8.         exit(0);
9.     }
10.    a->nodetype = 'K';
11.    a->number = d;
12.    return (struct ast *)a;
13. }
```

数字类型节点的 **nodetype** 为 'K'。由于数字类型节点一定是抽象语法树的叶子节点，因此 **numval** 结构不含左右子树指针。创建完成后，将数字类型强转为类型 **ast** 返回。

5.3 比较运算节点 ast

```
1. struct ast *
2. newcmp(int cmptype, struct ast *l, struct ast *r)
3. {
4.     struct ast *a = malloc(sizeof(struct ast));
5.
6.     if(!a) {
7.         yyerror("out of space");
8.         exit(0);
9.     }
10.    a->nodetype = '0' + cmptype;
11.    a->l = l;
12.    a->r = r;
13.    return a;
14. }
```

可以看出，比较运算创建的节点类型仍是 **ast**，之所以不能和普通的加减乘除写在一个函数内的原因是比较运算的 **nodetype** 确定方法与加减乘除不同。

5.4 标识符节点 symref

```
1. struct ast *
2. newref(struct symbol *s)
3. {
4.     struct symref *a = malloc(sizeof(struct symref));
5.
6.     if(!a) {
7.         yyerror("out of space");
8.         exit(0);
9.     }
10.    a->nodetype = 'N';
11.    a->s = s;
12.    return (struct ast *)a;
13. }
```

结合 symref 的结构定义，对函数 newref 可以更好地理解。

```
1. struct symref {
2.     int nodetype;           /* type N */
3.     struct symbol *s;
4. };
```

S 是指向符号表的指针。标识符节点会通过该指针连接到符号表中的某一符号，使用目标符号的值代替该标识符的值。

5.5 赋值节点 symasgn

```
1. struct ast *
2. newasgn(struct symbol *s, struct ast *v)
3. {
4.     struct symasgn *a = malloc(sizeof(struct symasgn));
5.
6.     if(!a) {
7.         yyerror("out of space");
8.         exit(0);
9.     }
10.    a->nodetype = '=';
11.    a->s = s;
12.    a->v = v;
13.    return (struct ast *)a;
14. }
```

赋值符号的规则为标识符=表达式，因此在 `symasgn` 节点中，使用指针 `s` 指向符号表中的标识符，使用指针 `v` 指向一颗抽象语法树 `ast`，使用 `ast` 的值赋值给标识符。

5.6 内置函数节点 `fncall`

```
1. struct ast *
2. newfunc(int functype, struct ast *l)
3. {
4.     struct fncall *a = malloc(sizeof(struct fncall));
5.
6.     if(!a) {
7.         yyerror("out of space");
8.         exit(0);
9.     }
10.    a->nodetype = 'F';
11.    a->l = l;
12.    a->functype = functype;
13.    return (struct ast *)a;
14. }
```

`nodetype` 为 'F' 的内置函数节点仅有一棵子树，但增加了 `functype` 域，值为枚举变量 `bifs`。从此处再次看出，如果要增加内置函数的话，需要为枚举类型 `bifs` 增加新值。

5.7 自定义函数节点 `ufncall`

```
1. struct ast *
2. newcall(struct symbol *s, struct ast *l)
3. {
4.     struct ufncall *a = malloc(sizeof(struct ufncall));
5.
6.     if(!a) {
7.         yyerror("out of space");
8.         exit(0);
9.     }
10.    a->nodetype = 'C';
11.    a->l = l;
12.    a->s = s;
13.    return (struct ast *)a;
14. }
```

当我们使用 `let` 保留字创建好了自定义函数后，再次使用函数时会创建 `ufncall` 节点。

创建自定义函数的方法在 5.8 中介绍。

5.8 创建自定义函数

在计算器的语法分析部分中，我们找到了如图所示的两条文法规则。symlist 定义了函数参数列表的匹配规则，调用 newsymlist 方法为 symlist 中添加参数。calclist 定义了标识符 Let 的使用规则：

let 函数名(参数列表) = 表达式

```

74  √ explist: exp
75      | exp ',' explist { $$ = newast('L', $1, $3); }
76      ;
77  √ symlist: NAME      { $$ = newsymlist($1, NULL); }
78      | NAME ',' symlist { $$ = newsymlist($1, $3); }
79      ;
80
81  √ calclist: /* nothing */
82  √      | calclist stmt EOL {
83  √          if(debug) dumpast($2, 0);
84              printf("= %4.4g\n> ", eval($2));
85              treefree($2);
86          }
87  √      | calclist LET NAME '(' symlist ')' '=' list EOL {
88              dodef($3, $5, $8);
89              printf("Defined %s\n> ", $3->name); }
90
91      | calclist error EOL { yyerrok; printf("> "); }
92      ;
93      %%
94

```

图 2-5 创建自定义函数的文法规则

创建自定义函数时，调用了 dodef 函数，参数为 \$3,\$5,\$8 即（标识符，参数列表，表达式）。

```

1.  /* define a function */
2.  void
3.  dodef(struct symbol *name, struct symlist *syms, struct ast *func)
4.  {
5.      if(name->syms) symlistfree(name->syms);
6.      if(name->func) treefree(name->func);
7.      name->syms = syms;
8.      name->func = func;
9.  }

```


结合源码与实验参考书可以得知，当函数被定义时，参数列表和抽象语法树将被简单的保存到符号表中函数名对应的条目中，同时替换了任意可能的旧版本。

5.9 表达式求值

对于抽象语法树的求值方法 `eval`，根据节点 `nodetype` 的不同，选用不同的方法递归的调用 `eval` 求值。

部分选择与递归计算的源码如下。

```

1.  /* expressions */
2.  case '+': v = eval(a->l) + eval(a->r); break;
3.  case '-': v = eval(a->l) - eval(a->r); break;
4.  case '*': v = eval(a->l) * eval(a->r); break;
5.  case '/': v = eval(a->l) / eval(a->r); break;
6.  case '|': v = fabs(eval(a->l)); break;
7.  case 'M': v = -eval(a->l); break;
8.
9.  /* comparisons */
10. case '1': v = (eval(a->l) > eval(a->r)) ? 1 : 0; break;
11. case '2': v = (eval(a->l) < eval(a->r)) ? 1 : 0; break;
12. case '3': v = (eval(a->l) != eval(a->r)) ? 1 : 0; break;
13. case '4': v = (eval(a->l) == eval(a->r)) ? 1 : 0; break;
14. case '5': v = (eval(a->l) >= eval(a->r)) ? 1 : 0; break;
15. case '6': v = (eval(a->l) <= eval(a->r)) ? 1 : 0; break;

```

当 `nodetype` 为内置函数时，调用 `callbuiltin` 方法，根据 `functype` 枚举值的不同，进而得知当前需要按照哪一个内置函数去计算。由此，如果要新增一个内置函数的话，需要在 `callbuiltin` 方法中增加一个对应枚举值的 `case`。

```

1. static double
2. callbuiltin(struct fncall *f)
3. {
4.     enum bifs functype = f->functype;
5.     double v = eval(f->l);
6.
7.     switch(functype) {
8.     case B_sqrt:
9.         return sqrt(v);
10.    case B_exp:
11.        return exp(v);
12.    case B_log:
13.        return log(v);

```

```

14. case B_print:
15.     printf("= %4.4g\n", v);
16.     return v;
17. default:
18.     yyerror("Unknown built-in function %d", functype);
19.     return 0.0;
20. }
21. }

```

6 Debug 模式

当我们输入 debug1 的时候，计算器会进入 debug 的模式 1。

此时，输入任何表达式，会在求值前打印其所建立的抽象语法树的信息。

```

> debug1 a = 1 * 2
debug set to 1
= a
  binop *
    number 1
    number 2
= 2
> 5 + (3 - 4)
binop +
  number 5
  binop -
    number 3
    number 4
= 4
>

```

图 2-6 debug 模式

7 增加函数 pow 及其重载函数的详细过程分析

7.1 二操作数 pow 函数代码改动

通过前面的分析，增加内置函数需要改动源码中的三处。

- 1) 词法分析 fb3-2.1 中，新增保留字匹配模式 pow.
- 2) fb3-2.h 头文件中，为枚举类型 bifs 增加取值 B_pow.

<pre> 48 /* built in functions */ 49 "sqrt" { yylval.fn = B_sqrt; return FUNC; } 50 "exp" { yylval.fn = B_exp; return FUNC; } 51 "log" { yylval.fn = B_log; return FUNC; } 52 "print" { yylval.fn = B_print; return FUNC; } 53 "pow" { yylval.fn = B_pow; return FUNC; } 54 </pre>	<pre> 48 enum bifs { /* built-in functions */ 49 B_sqrt = 1, 50 B_exp, 51 B_log, 52 B_print, 53 B_pow 54 }; 55 </pre>
--	--

图 2-7 前两处代码改动

3) fb3-2funcs.c 中，为函数 callbuiltin 中增加 case B_pow，并设置相应的计算方法

```

287 static double
288 callbuiltin(struct fncall *f)
289 {
290     enum bifs functype = f->functype;
291     double v = eval(f->l);
292
293     switch(functype) {
294     case B_sqrt:
295         return sqrt(v);
296     case B_exp:
297         return exp(v);
298     case B_log:
299         return log(v);
300     case B_print:
301         printf("= %4.4g\n", v);
302         return v;
303     case B_pow:
304         return pow(eval(f->l->l), v);
305     default:
306         yyerror("Unknown built-in function %d", functype);
307         return 0.0;
308     }
309 }
310

```

图 2-8 增加计算逻辑

其中，前两处改动较为容易理解，但第三处的改动则需要一些特殊的处理。在 7.2 部分中详细叙述。

7.2 二操作数 pow 的计算逻辑

pow 函数与其它内置函数如 sqrt、log 的不同之处在于，pow 的有两个操作数，这使得其计算逻辑与其余单操作数函数有很大的变化。

在图 2-8 中可以看出，callbuiltin 方法先调用 eval 计算 fncall 节点的子树，得到子树的值后，根据 functype 的不同或对结果开方，或对结果取对数等。

由于 sqrt 等方法是单操作数的，因此才可以使用 eval 的返回值去直接计算。

在双操作数的方法中，我们使用 eval 计算子树的值并返回赋值给变量 v，那么，计算后 v 的值是什么就成了很关键的一个问题。

下式是内置函数计算的语法规则，explist 是归约出的参数列表。

$$exp \rightarrow FUNC (explist)$$

观察 explist 的语法规则，发现对应的语法制导翻译动作调用 newast 方法，创建了 nodetype 为 L 的 ast 节点。

查看 eval 方法对 nodetype 为 L 的节点的计算逻辑，问题得以解决。

```

1. double v;
2. switch(a->nodetype){
3. ...
4. case 'L': eval(a->l); v = eval(a->r); break;
5. ...
6. }
7. return v;

```

可以看出，分别计算左右子树的值后，**返回了右子树的值**。返回右子树的值的原因也就很清晰了：函数的参数列表是**逗号表达式**，逗号表达式的值在一般的程序设计语言中都被定义为最右侧表达式的值。

因此，在图 2-8 中，首先计算了 `pow(x, y)` 的 `y` 的值赋值给 `v`，我们需要重新遍历一遍语法树，计算 `x` 的值（即调用 `eval(f->l->l)`），由此可以得到 `pow` 函数的计算逻辑。

7.3 二操作数 `pow` 函数抽象语法树的建立过程

- 词法分析匹配到保留字"`pow`"，为 `yyval` 赋值 `B_pow`，返回语法分析标识符 `FUNC`
- 标识 `FUNC` 压入语法分析符号栈，当前句型的句柄未出现，继续读入下一个单词 `token`
- 词法分析匹配到`'(`，返回语法分析标识符`'(`
- 标识`'(`压入语法分析符号栈，当前句型的句柄未出现，继续读入下一个单词 `token`
- 词法分析分别匹配到 `pow` 函数的两个参数`(x,y)`，交给语法分析
- 语法分析将参数 `x, y` 规约为 `explist`，执行语法制导翻译动作

$$\{\$ \$ = \text{newast}('L', \$1, \$3); \}$$

- 调用 `newast` 方法，创建新的抽象语法树节点，左孩子为 `x`，右孩子为 `y`。

- 将归约出的标识 **explist** 压入语法分析符号栈，当前句型句柄未出现，继续读入下一个单词 **token**
- 词法分析匹配到')'，返回语法分析标识符')

标识')'压入语法分析符号栈，句柄出现，使用下表达式归约。

$$exp \rightarrow FUNC (explist)$$

- 规约后，执行该产生式对应语法制导翻译的动作{ \$\$ = newfunc(\$1, \$3); }, 进而为抽象语法树创建 **pow** 函数的节点。

7.4 二操作数 pow 的计算结果

计算结果如下图所示：

```
D:\Here\recently\系统软件开发实践\综合实验\3-2\code>fb3-2.exe
> pow(2, 3)
= 8
> pow(3, 2)
= 9
> pow(1 + 2, 2 * 1)
= 9
> |
```

图 2-9 pow 计算结果

7.5 三操作数 pow 函数

实现了二操作数的 **pow** 函数，对于其重载函数——三操作数的 **pow** 函数的添加就很简单了。

三操作数的 **pow** 函数的表达式如下：

$$pow(x, y, c) = x^y \% c$$

由于是重载函数，对于 6.1 中的前两处修改不再需要，只需要将第三处即 `pow` 的计算逻辑处增加额外的修改即可。

```

303 case B_pow:
304     if(f->l->r->nodetype == 'L'){ //说明是带取余的pow，三操作数
305         /*
306             v: 取余
307             eval(f->l->l): x
308             eval(f->l->r->l): y
309         */
310         double x = eval(f->l->l);
311         double y = eval(f->l->r->l);
312         double ans = pow(x, y);
313         ans = ans - (int) (ans / v) * v;
314         return ans;
315     } else {
316         return pow(eval(f->l->l), v);
317     }
318 default:
319     yyerror("Unknown built-in function %d", functype);
320     return 0.0;
321 }
322 }

```

图 2-10 三操作数 `pow` 的代码修改

我们使用函数节点的右孩子节点的 `nodetype` 来判断当前是二操作数还是三操作数的 `pow` 函数。其抽象语法树的节点类型如图 2-11 所示。节点上的字母为 `nodetype` 类型。

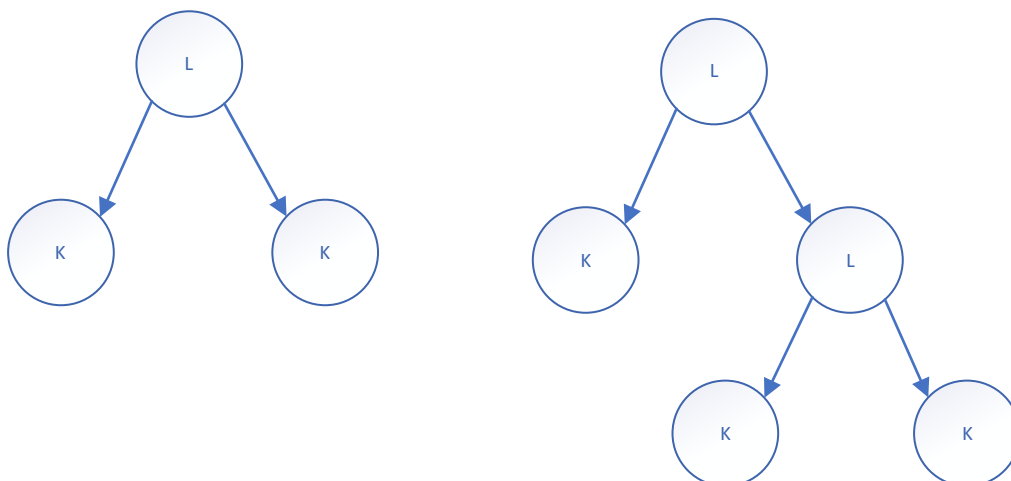
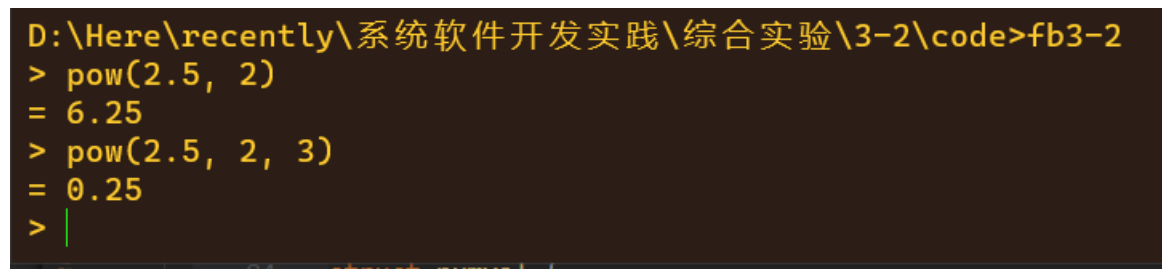


图 2-11 二操作数（左）与三操作数（右）`pow` 函数的抽象语法树

可见，三操作数的 `pow` 函数与二操作数 `pow` 函数的区别在于根节点的右孩子的节点类型——三操作数为逗号表达式（L），而二操作数为数字节点（K）。因此，在图 2-10

的代码中，行 304 的 if 语句可以实现对两个重载函数的区分。

运行结果如图所示。



```
D:\Here\recently\系统软件开发实践\综合实验\3-2\code>fb3-2
> pow(2.5, 2)
= 6.25
> pow(2.5, 2, 3)
= 0.25
> |
```

图 2-12 pow 重载的运行结果

8 实验总结

本次实验收获颇丰。

结合参考书，理解了相当一部分的源代码，并在源码中发掘出了很多实验要求中没有涉及到的计算器模式——多行输入（图 2-3）、debug 模式（图 2-6）等。在仔细理解源码的基础上，修改源码，增加了内置函数 `pow(x,y)` 及其重载的三操作数形式 `pow(x,y,c)`（图 2-12），再次体会到了代码阅读能力与实践编码能力的重要性。

最后，对整个桌面计算器的源码架构与工作逻辑有了更深刻的理解。