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

实验名称 实验六 Flex/Bison 综合实验二

学生姓名 胡钧耀

学　　号 06192081

专业班级 计算机科学与技术 2019-4 班

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

实验（六） Flex/Bison 综合实验二

6.1 实验要求与目标

使用 flex 和 bison 开发了一个具有全部功能的桌面计算器，能够支持变量，过程，循环和条件表达式，使它成为一个虽然短小但具有现实意义的编译器。

学习抽象语法树的用法，它具有强大而简单的数据结构来表示分析结果。

6.2 实验内容

阅读 *Flex/Bison.pdf* 第三章。使用 Flex 和 Bison 开发了一个具有全部功能的桌面计算器：支持变量；实现赋值功能；实现比较表达式（大于、小于等）；实现 if/then/else 和 do/while 的流程控制；用户可以自定义函数；简单的错误恢复机制。

6.3 不同环境的配置和使用

6.3.1 Windows

根据以往经验，为防止报警，需要修改代码，首先是给 *fb3-2funcs.c* 加入 yyparse 方法的显式声明。

```
int yyparse();
```

其次是要在 *fb3-2.y* 给加入 yylex 方法的显式声明。

```
int yylex();
```

执行如下代码，可生成计算器 *3-2.exe*。调用该计算器，验证结果如下，与真实情况一致，表明计算器无问题。

```
bison -d fb3-2.y
flex -ofb3-2.lex.c fb3-2.l
gcc -o cal3-2.exe fb3-2.tab.c fb3-2.lex.c fb3-2funcs.c -lm
```

```
D:\Docs\CUMT_StudyFiles\3-2-系统软件开发实践\flex&bison实验2>bison -d fb3-2.y

D:\Docs\CUMT_StudyFiles\3-2-系统软件开发实践\flex&bison实验2>flex -ofb3-2.lex.c fb3-2.l

D:\Docs\CUMT_StudyFiles\3-2-系统软件开发实践\flex&bison实验2>gcc -o cal3-2.exe fb3-2.tab.c fb3-
2.lex.c fb3-2funcs.c -lm

D:\Docs\CUMT_StudyFiles\3-2-系统软件开发实践\flex&bison实验2>cal3-2.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 1 Windows 环境生成计算器 *3-2.exe* 及其计算结果

6.3.2 Linux

直接把在 Windows 修改好的四个源文件复制到 Linux 系统中，输入如下代码，并进行调用，验证结果正确。

```
bison -d fb3-2.y
flex -ofb3-2.lex.c fb3-2.l
gcc -o cal3-2.out fb3-2.tab.c fb3-2.lex.c fb3-2funcs.c -lm
```

```
hujunyao@HJYUbuntu:~/系统软件开发实践/实验6$ bison -d fb3-2.y
hujunyao@HJYUbuntu:~/系统软件开发实践/实验6$ flex -ofb3-2.lex.c fb3-2.l
hujunyao@HJYUbuntu:~/系统软件开发实践/实验6$ gcc -o cal3-2.out fb3-2.ta
b.c fb3-2.lex.c fb3-2funcs.c -lm
hujunyao@HJYUbuntu:~/系统软件开发实践/实验6$ ./cal3-2.out
> 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 Linux 生成计算器 *3-2.out* 及运算结果

6.4 分析源码

6.4.1 *fb3-2.y*

首先引用一些必要的头文件，避免让 yylex 和 yyparse 等方法报错或报警。

```
%{
#  include <stdio.h>
#  include <stdlib.h>
#  include "fb3-2.h"
int yylex();
%}
```

定义了一个结构体，包括一个抽象语法树结构体指针、一个数字、一个符号结构体指针、一个符号表结构体指针和一个函数标识。

```
%union {
  struct ast *a;
  double d;
  struct symbol *s;    /* which symbol */
  struct symlist *sl;
  int fn;       /* which function */
}
```

声明了一些终结符号，一个是 NUMBER 数字，一个是名称 NAME，一个是函数 FUNC，一个是 EOL（End of Line）行结束符号，还有一些终结符号是

结构如 IF、THEN、ELSE、WHILE、DO、LET。

```
%token <d> NUMBER
%token <s> NAME
%token <fn> FUNC
%token EOL
%token IF THEN ELSE WHILE DO LET
```

下面是对定义的或者需要设置的符号手动设置优先级，这是为了防止出现移进规约冲突的发生，这样使得语法分析器性能更加优良。

可以使用%left、%right、%noassoc，和%precedence 来定义符号的优先级和结合性（它们分别代表着是左结合、右结合、没有结合性、未定义的结合性）。

这里函数和比较关系没有结合性，等号有右结合性，四则运算有左结合性，绝对值和取负数符号没有结合性。

```
%nonassoc <fn> CMP
%right '='
%left '+' '-'
%left '*' '/'
%nonassoc '|' UMINUS
```

声明非终结符 exp、stmt、list、explist、symlist。设置起始符号 symlist。

```
%type <a> exp stmt list explist
%type <sl> symlist
%start calclist
```

接着是各种模式的匹配以及对应的语法动作。

stmt→IF exp THEN list。读取 exp 和 list，加上 NULL，使用 if（I）连接起来，成为新的短语 newflow。

stmt→IF exp THEN list ELSE list。读取 exp、list1 和 list2，使用 if（I）连接起来，成为新的短语 newflow。

stmt→WHILE exp DO list。读取 exp 和 list，加上 NULL，使用 WHILE（W）连接起来，成为新的短语 newflow。

stmt→exp。直接转换，没有做任何其他操作。

```
stmt: IF exp THEN list           { $$ = newflow('I', $2, $4,
NULL); }
   | IF exp THEN list ELSE list  { $$ = newflow('I', $2, $4,
$6); }
   | WHILE exp DO list           { $$ = newflow('W', $2, $4,
NULL); }
   | exp
;
```

exp→exp CMP exp。读取 exp1 和 exp2，使用 CMP 连接起来，成为新的比较短语 newcmp。

exp→exp + exp。读取 exp1 和 exp2，使用+连接起来，成为新的抽象语法树 newast。

exp→exp - exp。同上。

exp→exp * exp。同上。

exp→exp / exp。同上。

exp→| exp。读取 exp，加上 NULL，和|连接起来，成为新的抽象语法树 newast。

exp→(exp)。直接读取 exp。抽象语法树无需括号。

exp→- exp %prec UMINUS。读取 exp，加上 NULL，和 minus（M）连接起来，成为新的抽象语法树 newast。%prec UMINUS 表示定义规则对应的符号（也就是定义此规则和哪个符号的优先级相同）。

exp→NUMBER。读取 NUMBER，生成新的数字 newnum。

exp→FUNC (explist)。读取 exp，和 function（FUNC）连接起来，成为新的函数 newfunc。

exp→NAME。读取 NAME，生成新的标识符 newref。

exp→NAME = exp。读取 exp，和 NAME 连接起来，成为新的函数 newasgn。

exp→NAME (explist)。读取 explist，和 NAME 连接起来，成为新的调用 newcall。

```
exp: exp CMP exp          { $$ = newcmp($2, $1, $3); }
   | exp '+' exp          { $$ = newast('+', $1,$3); }
   | exp '-' exp          { $$ = newast('-', $1,$3);}
   | exp '*' exp          { $$ = newast('*', $1,$3); }
   | exp '/' exp          { $$ = newast('/', $1,$3); }
   | '|' exp              { $$ = newast('|', $2, NULL); }
   | '(' exp ')'          { $$ = $2; }
   | '-' exp %prec UMINUS { $$ = newast('M', $2, NULL); }
   | NUMBER               { $$ = newnum($1); }
   | FUNC '(' explist ')' { $$ = newfunc($1, $3); }
   | NAME                 { $$ = newref($1); }
   | NAME '=' exp         { $$ = newasgn($1, $3); }
   | NAME '(' explist ')' { $$ = newcall($1, $3); }
;
```

explist→exp。直接转换为 exp。

explist→exp , explist。读取 exp 和 explist，和 list（L）连接起来，成为新的抽象语法树 newast。

```
explist: exp
 | exp ',' explist  { $$ = newast('L', $1, $3); }
;
```

symlist→NAME。读取 NAME，加上 NULL，成为新的参数列表 newsymlist。

explist→NAME , symlist。读取 symlist，和 NAME 连接起来，成为新的参数列表 newsymlist。

```
symlist: NAME       { $$ = newsymlist($1, NULL); }
 | NAME ',' symlist { $$ = newsymlist($1, $3); }
;
```

calclist→ ε 。什么也不做。

calclist→calclist stmt EOL。如果是调试是对的，就接着进行对 stmt 的 dumpast 操作，g 说明根据数值不同选择浮点数、十进制计数法%f 或浮点数、e 计数法%e。4.4 说明如果是小数，宽度 4，小数点后保留 4 位。treefree 表示对 exp 这棵抽象语法树进行释放（类似 delete）。应该是一个递归类型的操作。

calclist→calclist LET NAME (symlist) = list EOL。这里对 NAME、symlis、list 进行了 dodef 也就是定义函数操作，然后输出程序已经定义了以 name 为名的函数的提示。

calclist→calclist error EOL。这里操作含有 yerror，因为 parser 检测到语法错误的时候，默认的做法是把所有东西清空，然后返回让编译器强制停止。但是这种默认行为就只能检测出某一段代码的第一个语法错误。如果要让 parser 在检测到错误的时候不停止，而是继续往下匹配，那么就应该用 yyerrok，这样 parser 就会继续根据规则去匹配后面的代码，而不至于立即退出。在这个程序里遇到错误会输出一行只有提示符的空行，然后继续进行。

```
calclist: /* nothing */
  | calclist stmt EOL {
    if(debug) dumpast($2, 0);
     printf("= %4.4g\n> ", eval($2));
     treefree($2);
    }
  | calclist LET NAME '(' symlist ')' '=' list EOL {
                       dodef($3, $5, $8);
                       printf("Defined %s\n> ", $3->name); }

  | calclist error EOL { yyerrok; printf("> "); }
 ;
```

6.4.2 *fb3-2.l*

flex 的选项影响最终生成的词法分析器的属性和行为。这些选项可以在运行 flex 命令时在终端输入，也可以在.*l* 文件中使用%option 指定。noyywrap 表示在该.*l* 文件中不会调用 yywrap，假设生成的扫描器只扫描单个文件。nodefault 表示不使用默认规则。yylineno 表示记录符号所在行号。

```
%option noyywrap nodefault yylineno
```

引用一些必要的头文件，避免报错报警。

```
%{ # include "fb3-2.h" # include "fb3-2.tab.h" %}
```

定义了一种科学计数法的指数表达式。

```
/* float exponent */
EXP ([Ee][-+]?[0-9]+)
```

遇到符号，返回自己本身 yytext[0]。

```
"+" | "-" | "*" | "/" | "|" | "(" | ")"  { return yytext[0]; }
```

遇到六种比较运算符，定义其代表的标识，并返回 CMP 标记。

```
">"     { yylval.fn = 1; return CMP; }
"<"     { yylval.fn = 2; return CMP; }
"<>"    { yylval.fn = 3; return CMP; }
"=="    { yylval.fn = 4; return CMP; }
">="    { yylval.fn = 5; return CMP; }
"<="    { yylval.fn = 6; return CMP; }
```

定义关键词并分别返回自己。

```
"if"    { return IF; }
"then"  { return THEN; }
"else"  { return ELSE; }
"while" { return WHILE; }
"do"    { return DO; }
"let"   { return LET;}
```

定义四种内部（built-in）函数，定义其代表的标识，并都返回 FUNC 标记。

```
"sqrt"  { yylval.fn = B_sqrt; return FUNC; }
"exp"   { yylval.fn = B_exp; return FUNC; }
"log"   { yylval.fn = B_log; return FUNC; }
"print" { yylval.fn = B_print; return FUNC; }
```

遇到 debug+数字，输出其数字。

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

遇到 id 标识符，首先查表，然后返回 NAME。

```
[a-zA-Z][a-zA-Z0-9]*  { yylval.s = lookup(yytext); return
```

```
NAME; }
```

匹配到整数、小数、科学计数，把 yytext 转为浮点数，然后返回 NUMBER。

```
[0-9]+"."[0-9]*{EXP}? | "."?[0-9]+{EXP}?
{ yylval.d = atof(yytext); return NUMBER; }
```

忽略注释、忽略空格、忽略空白行。匹配到换行符，返回 EOL 行末标识。其他符号则作为未知符号进入 yyerror 处理操作。

```
"//".*
[ \t]   /* ignore white space */
\\\n    printf("c> "); /* ignore line continuation */
"\n"    { return EOL; }
.   { yyerror("Mystery character %c\n", *yytext); }
%%
```

6.4.3 *fb3-2funcs.c*

引用一些必要的头文件，避免报错报警。

```
#  include <stdio.h>
#  include <stdlib.h>
#  include <stdarg.h>
#  include <string.h>
#  include <math.h>
#  include "fb3-2.h"
int yyparse();
```

定义 symhash 函数对 sym 进行哈希，得到其哈希值。

```
static unsigned
symhash(char *sym)
{
  unsigned int hash = 0;
  unsigned c;
  while(c = *sym++) hash = hash*9 ^ c;
  return hash;
}
```

定义 lookup 函数，查找 sym 时候存在，如果不存在空位置、溢出就会报错（参考《数据结构》课程知识）。

```
struct symbol *
lookup(char* sym)
{
```

```
  struct symbol *sp = &symtab[symhash(sym)%NHASH];
  int scount = NHASH;   /* how many have we looked at */

  while(--scount >= 0) {
    if(sp->name && !strcmp(sp->name, sym)) { return sp; }

    if(!sp->name) {   /* new entry */
      sp->name = strdup(sym);
      sp->value = 0;
      sp->func = NULL;
      sp->syms = NULL;
      return sp;
    }

    if(++sp >= symtab+NHASH) sp = symtab; /* try the next
entry */
  }
  yyerror("symbol table overflow\n");
  abort(); /* tried them all, table is full */

}
```

定义了建立新的子节点的函数 newast，参数是节点类型 nodetype、左值指针 l、右值指针 r，返回值是该抽象树根节点的指针。其操作主要是把新的根节点 a 和左节点 l 和右节点 r 连接起来。

```
struct ast *
newast(int nodetype, struct ast *l, struct ast *r)
{
  struct ast *a = malloc(sizeof(struct ast));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = nodetype;
  a->l = l;
  a->r = r;
  return a;
}
```

定义了建立新的子节点的函数 newnum，参数是数字 d，返回值是该抽象树根节点的指针。其操作主要是把新的数字根节点 a 的结构体标注上对应的节点类型 K 和数字的值 d。

```
struct ast *
newnum(double d)
{
  struct numval *a = malloc(sizeof(struct numval));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = 'K';
  a->number = d;
  return (struct ast *)a;
}
```

定义了建立新的子节点的函数 newcmp，参数是比较类型 cmptype 和左右语法树指针，返回值是该抽象树根节点的指针。其操作主要是把新的比较根节点 a 的结构体标注上对应的比较类型 nodetype，并连接需要比较的左右子树。

```
struct ast *
newcmp(int cmptype, struct ast *l, struct ast *r)
{
  struct ast *a = malloc(sizeof(struct ast));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = '0' + cmptype;
  a->l = l;
  a->r = r;
  return a;
}
```

定义了建立新的子节点的函数 newfunc，参数是比较类型 functype 和语法树指针，返回值是该抽象树根节点的指针。其操作主要是把新的比较根节点 a 的结构体标注上对应的类型 function（F），函数类型为 functype。其实在初始时，已经定义了 sqrt、exp、log、print 这 4 种内置函数类型。

```
struct ast *
newfunc(int functype, struct ast *l)
{
  struct fncall *a = malloc(sizeof(struct fncall));
```

```
  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = 'F';
  a->l = l;
  a->functype = functype;
  return (struct ast *)a;
}
```

定义了建立新的子节点的函数 newcall，参数是符号 symbol 和语法树指针，返回值是该抽象树根节点的指针。其操作主要是把新的根节点 a 的结构体标注上对应的类型 call（C），符号类型为 s，表达式是 l。

```
struct ast *
newcall(struct symbol *s, struct ast *l)
{
  struct ufncall *a = malloc(sizeof(struct ufncall));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = 'C';
  a->l = l;
  a->s = s;
  return (struct ast *)a;
}
```

定义了建立新的子节点的函数 newref，参数是符号 symbol，返回值是该抽象树根节点的指针。其操作主要是把新的根节点 a 的结构体标注上对应的类型 N，符号类型为 s。

```
struct ast *
newref(struct symbol *s)
{
  struct symref *a = malloc(sizeof(struct symref));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = 'N';
  a->s = s;
```

```
  return (struct ast *)a;
}
```

定义了建立新的子节点的函数 newasgn，参数是符号 symbol 和抽象语法树指针 v，返回值是该抽象树根节点的指针。其操作主要是把新的根节点 a 的结构体标注上对应的类型等于（=），符号类型为 s。

```
struct ast *
newasgn(struct symbol *s, struct ast *v)
{
  struct symasgn *a = malloc(sizeof(struct symasgn));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = '=';
  a->s = s;
  a->v = v;
  return (struct ast *)a;
}
```

定义了建立新的子节点的函数 newflow，参数是节点类型 nodetype、条件语句抽象语法树指针 cond，else 前面的抽象语法树指针 tl 和 else 后面的抽象语法树指针 el，返回值是该抽象树根节点的指针。其操作主要是把新的根节点 a 的结构体标注上对应的分支类型 if 或者 while（I、W），符号类型为 s。

```
struct ast *
newflow(int nodetype, struct ast *cond, struct ast *tl, struct
ast *el)
{
  struct flow *a = malloc(sizeof(struct flow));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = nodetype;
  a->cond = cond;
  a->tl = tl;
  a->el = el;
  return (struct ast *)a;
}
```

定义了建立新的子节点的函数 newsymlist，参数是等待连接的符号的抽象语法树指针 sym 和已经连好的抽象语法树指针 next，返回值是该 symlist 的指针，而不是抽象语法树。仅维护自身符号与下一个节点的符号。

```
struct symlist *
newsymlist(struct symbol *sym, struct symlist *next)
{
  struct symlist *sl = malloc(sizeof(struct symlist));

  if(!sl) {
    yyerror("out of space");
    exit(0);
  }
  sl->sym = sym;
  sl->next = next;
  return sl;
}
```

symlistfree 函数用来递归地释放符号链表。

```
void
symlistfree(struct symlist *sl)
{
  struct symlist *nsl;
  while(sl) {
    nsl = sl->next;
    free(sl);
    sl = nsl;
  }
}
```

dodef 函数定义一个函数。如果该定义已有，则先删去之前的定义，接着记录函数名 syms，以及函数对应的语法树 func。

```
void
dodef(struct symbol *name, struct symlist *syms, struct ast
*func)
{
  if(name->syms) symlistfree(name->syms);
  if(name->func) treefree(name->func);
  name->syms = syms;
  name->func = func;
}
```

定义了计算表达式数值的函数 eval，这容易使人联想到 Python 的 eval 方法。参数是抽象语法树的根结点指针 a，返回的是该抽象语法树的浮点数值结果。其操作主要是根据 case 语句分析出不同的 nodetype 要进行不同的处理，例如类型 K 标识根节点是数字，就可以直接输出对应的数值 v，如果类型是+，就要把 a 指针的左节点的 eval 方法返回值和右节点的 eval 方法的返回值进行相加，把这个结果 v 返回。不是所列出的类型，就要报错匹配不到对应的节点类型。

```
double
eval(struct ast *a)
{
  double v;

  if(!a) {
    yyerror("internal error, null eval");
    return 0.0;
  }

  switch(a->nodetype) {
    /* constant */
  case 'K': v = ((struct numval *)a)->number; break;

    /* name reference */
  case 'N': v = ((struct symref *)a)->s->value; break;

    /* assignment */
  case '=': v = ((struct symasgn *)a)->s->value =
      eval(((struct symasgn *)a)->v); break;

    /* expressions */
  case '+': v = eval(a->l) + eval(a->r); break;
  case '-': v = eval(a->l) - eval(a->r); break;
  case '*': v = eval(a->l) * eval(a->r); break;
  case '/': v = eval(a->l) / eval(a->r); break;
  case '|': v = fabs(eval(a->l)); break;
  case 'M': v = -eval(a->l); break;

    /* comparisons */
  case '1': v = (eval(a->l) > eval(a->r))? 1 : 0; break;
  case '2': v = (eval(a->l) < eval(a->r))? 1 : 0; break;
  case '3': v = (eval(a->l) != eval(a->r))? 1 : 0; break;
  case '4': v = (eval(a->l) == eval(a->r))? 1 : 0; break;
  case '5': v = (eval(a->l) >= eval(a->r))? 1 : 0; break;
  case '6': v = (eval(a->l) <= eval(a->r))? 1 : 0; break;
```

```
  /* control flow */
  /* null if/else/do expressions allowed in the grammar, so
check for them */
  case 'I':
    if( eval( ((struct flow *)a)->cond) != 0) {
      if( ((struct flow *)a)->tl) {
  v = eval( ((struct flow *)a)->tl);
      } else
  v = 0.0;    /* a default value */
    } else {
      if( ((struct flow *)a)->el) {
        v = eval(((struct flow *)a)->el);
      } else
  v = 0.0;    /* a default value */
    }
    break;

  case 'W':
    v = 0.0;    /* a default value */

    if( ((struct flow *)a)->tl) {
      while( eval(((struct flow *)a)->cond) != 0)
  v = eval(((struct flow *)a)->tl);
    }
    break;      /* last value is value */

  case 'L': eval(a->l); v = eval(a->r); break;

  case 'F': v = callbuiltin((struct fncall *)a); break;

  case 'C': v = calluser((struct ufncall *)a); break;

  default: printf("internal error: bad node %c\n", a-
>nodetype);
  }
  return v;
}
```

callbuiltin 方法是识别内建函数的，本实验有四种内建函数即 B_sqrt、B_exp、B_log、B_print。

```
static double
callbuiltin(struct fncall *f)
{
```

```
  enum bifs functype = f->functype;
  double v = eval(f->l);

 switch(functype) {
 case B_sqrt:
   return sqrt(v);
 case B_exp:
   return exp(v);
 case B_log:
   return log(v);
 case B_print:
   printf("= %4.4g\n", v);
   return v;
 default:
   yyerror("Unknown built-in function %d", functype);
   return 0.0;
 }
}
```

calluser 方法是识别用户自定义函数的。

```
static double
calluser(struct ufncall *f)
{
  struct symbol *fn = f->s; /* function name */
  struct symlist *sl;   /* dummy arguments */
  struct ast *args = f->l;  /* actual arguments */
  double *oldval, *newval;  /* saved arg values */
  double v;
  int nargs;
  int i;

  if(!fn->func) {
    yyerror("call to undefined function", fn->name);
    return 0;
  }

  /* count the arguments */
  sl = fn->syms;
  for(nargs = 0; sl; sl = sl->next)
    nargs++;

  /* prepare to save them */
  oldval = (double *)malloc(nargs * sizeof(double));
```

```
newval = (double *)malloc(nargs * sizeof(double));
if(!oldval || !newval) {
  yyerror("Out of space in %s", fn->name); return 0.0;
}

/* evaluate the arguments */
for(i = 0; i < nargs; i++) {
  if(!args) {
    yyerror("too few args in call to %s", fn->name);
    free(oldval); free(newval);
    return 0;
  }

  if(args->nodetype == 'L') { /* if this is a list node */
    newval[i] = eval(args->l);
    args = args->r;
  } else {      /* if it's the end of the list */
    newval[i] = eval(args);
    args = NULL;
  }
}

/* save old values of dummies, assign new ones */
sl = fn->syms;
for(i = 0; i < nargs; i++) {
  struct symbol *s = sl->sym;

  oldval[i] = s->value;
  s->value = newval[i];
  sl = sl->next;
}

free(newval);

/* evaluate the function */
v = eval(fn->func);

/* put the dummies back */
sl = fn->syms;
for(i = 0; i < nargs; i++) {
  struct symbol *s = sl->sym;

  s->value = oldval[i];
  sl = sl->next;
```

```
  }

  free(oldval);
  return v;
}
```

定义了删除抽象语法树的函数 treefree，是一种递归函数，要删除本语法树首先要删除其所有子节点。不同的节点类型会进行不同的操作。

```
void
treefree(struct ast *a)
{
  switch(a->nodetype) {
    /* two subtrees */
  case '+':
  case '-':
  case '*':
  case '/':
  case '1':  case '2':  case '3':  case '4':  case '5':  case
'6':
  case 'L':
    treefree(a->r);
    /* one subtree */
  case '|':
  case 'M': case 'C': case 'F':
    treefree(a->l);
    /* no subtree */
  case 'K': case 'N':
    break;
  case '=':
    free( ((struct symasgn *)a)->v);
    break;
  case 'I': case 'W':
    free( ((struct flow *)a)->cond);
    if( ((struct flow *)a)->tl) free( ((struct flow *)a)->tl);
    if( ((struct flow *)a)->el) free( ((struct flow *)a)->el);
    break;
  default: printf("internal error: free bad node %c\n", a-
>nodetype);
  }
    free(a); /* always free the node itself */
}
```

出错处理。输出错误及其行号。

```
void
yyerror(char *s, ...)
{
  va_list ap;
  va_start(ap, s);

  fprintf(stderr, "%d: error: ", yylineno);
  vfprintf(stderr, s, ap);
  fprintf(stderr, "\n");
}
```

主函数，先输出行引导符号，再调用 yyparse 进行分析。

```
int
main()
{
  printf("> ");
  return yyparse();
}
```

当输入内容较为复杂，或是函数层层嵌套调用较深时，通过 debug 打印输出，可以观察计算器的整个求值语法树。

```
int debug = 0;
void
dumpast(struct ast *a, int level)
{

  printf("%*s", 2*level, ""); /* indent to this level */
  level++;

  if(!a) {
    printf("NULL\n");
    return;
  }

  switch(a->nodetype) {
    /* constant */
  case 'K': printf("number %4.4g\n", ((struct numval *)a)-
>number); break;

    /* name reference */
  case 'N': printf("ref %s\n", ((struct symref *)a)->s->name);
```

```
break;

    /* assignment */
  case '=': printf("= %s\n", ((struct symref *)a)->s->name);
    dumpast( ((struct symasgn *)a)->v, level); return;

    /* expressions */
  case '+': case '-': case '*': case '/': case 'L':
  case '1': case '2': case '3':
  case '4': case '5': case '6':
    printf("binop %c\n", a->nodetype);
    dumpast(a->l, level);
    dumpast(a->r, level);
    return;

  case '|': case 'M':
    printf("unop %c\n", a->nodetype);
    dumpast(a->l, level);
    return;

  case 'I': case 'W':
    printf("flow %c\n", a->nodetype);
    dumpast( ((struct flow *)a)->cond, level);
    if( ((struct flow *)a)->tl)
      dumpast( ((struct flow *)a)->tl, level);
    if( ((struct flow *)a)->el)
      dumpast( ((struct flow *)a)->el, level);
    return;

  case 'F':
    printf("builtin %d\n", ((struct fncall *)a)->functype);
    dumpast(a->l, level);
    return;

  case 'C': printf("call %s\n", ((struct ufncall *)a)->s-
>name);
    dumpast(a->l, level);
    return;

  default: printf("bad %c\n", a->nodetype);
    return;
  }
}
```

6.4.4 *fb3-2.h*

定义符号表。

```
struct symbol {   /* a variable name */
  char *name;
  double value;
  struct ast *func; /* stmt for the function */
  struct symlist *syms; /* list of dummy args */
};
```

定义内建函数。

```
enum bifs {     /* built-in functions */
  B_sqrt = 1,
  B_exp,
  B_log,
  B_print
};
```

定义抽象语法树结构体，包括当前节点类型，以及左右子树的指针。

```
/* nodes in the Abstract Syntax Tree */
struct ast {
  int nodetype;
  struct ast *l;
  struct ast *r;
};
```

定义数字节点的结构体，包括当前节点类型 K，以及数字节点的值。

```
struct numval {
  int nodetype;     /* type K */
  double number;
};
```

还定义了 fncall、ufncall、flow、numval、symref、symasgn。形式基本相同。

```
struct fncall {     /* built-in function */
  int nodetype;     /* type F */
  struct ast *l;
  enum bifs functype;
};

struct ufncall {    /* user function */
```

```
  int nodetype;      /* type C */
  struct ast *l;     /* list of arguments */
  struct symbol *s;
};

struct flow {
  int nodetype;      /* type I or W */
  struct ast *cond;   /* condition */
  struct ast *tl;    /* then or do list */
  struct ast *el;    /* optional else list */
};

struct numval {
  int nodetype;      /* type K */
  double number;
};

struct symref {
  int nodetype;      /* type N */
  struct symbol *s;
};

struct symasgn {
  int nodetype;      /* type = */
  struct symbol *s;
  struct ast *v;     /* value */
};
```

在.*h* 文件声明需要建立的下面几个函数，这些函数已经在.*c* 文件中编写。

```
struct ast *newast(int nodetype, struct ast *l, struct ast *r);
struct ast *newcmp(int cmptype, struct ast *l, struct ast *r);
struct ast *newfunc(int functype, struct ast *l);
struct ast *newcall(struct symbol *s, struct ast *l);
struct ast *newref(struct symbol *s);
struct ast *newasgn(struct symbol *s, struct ast *v);
struct ast *newnum(double d);
struct ast *newflow(int nodetype, struct ast *cond, struct ast
*tl, struct ast *tr);
/* define a function */
void dodef(struct symbol *name, struct symlist *syms, struct
ast *stmts);

/* evaluate an AST */
```

```
double eval(struct ast *);
/* delete and free an AST */
void treefree(struct ast *);
/* interface to the lexer */
extern int yylineno; /* from lexer */
void yyerror(char *s, ...);
extern int debug;
void dumpast(struct ast *a, int level);
```

6.5 构建抽象语法树

6.5.1 sq(10)构建过程

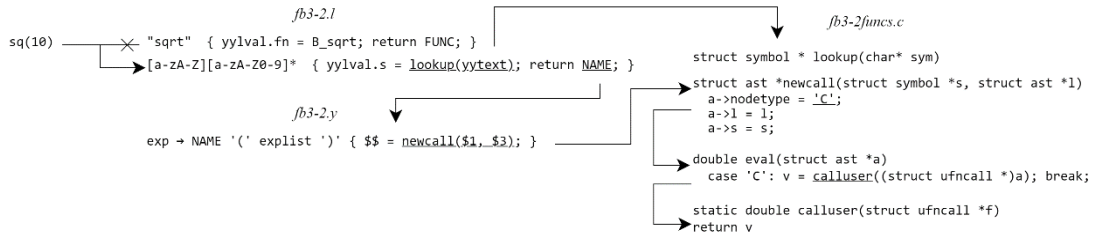

图 3 sq(10)的构建过程

当我们向计算器输入 sq(10)时，先是用词法分析器对其进行分析，判断出是内置函数或自定义函数，返回相应的 token 告诉语法分析器。

语法分析器收到词法分析结果，建立相应的语法树节点，并且设置好节点的参数。在 eval 函数计算结果时，根据节点的类型，调用对应的函数代码，计算出正确的结果。

以下结合代码进行详细的分析。

首先将计算后的结果在内建函数里面查找，发现没有，则是自定义函数。计算器由这条规则来处理 NAME 标识的用户自定义函数。

```
Exp：  NAME '(' explist ')' { $$ = newcall($1, $3); }
```

其中的 explist 定义了一个表达式列表，它用于创建抽象语法树来作为函数调用所需要的实参。

```
explist: exp
```

symlist 定义了一个符号列表，创建了这些符号的链表用于函数定义时所带的形参。

```
symlist: NAME { $$ = newsymlist($1, NULL); }
```

建立语法树节点的两个函数 newfunc、newast。

本次实验制作的计算器的重点是利用抽象语法树实现了流程的控制和用户自定义函数的功能。

对表达式的计算过程，在以上已经描述。计算器的核心是 eval，它用于计算抽象语法树。其中比较表达式根据比较的结果返回 1 或者。在计算例如 if、then、 else 的选择分支时，他会通过计算表达式的抽象语法树的值来决定选择哪一个分支继续。在计算例如 do、while 循环时，eval 会循环地计算条件表达式的值，然后建立语句的抽象与法树，直到表达式不再成立为止。

关于内置函数的实现：计算器先确定用户调用了什么函数，然后再调用相应的代码来执行即可。

关于用户自定义函数：用户自定义的函数包括函数名、形参列表、函数体

当自定义函数被调用时：计算实参的值、保存形参的值，将实参赋值给形参、执行函数体，构建抽象语法树、将形参恢复原来的值、返回函数的返回值。

6.5.2 sq(10)构建结果

构建示例如下所示。

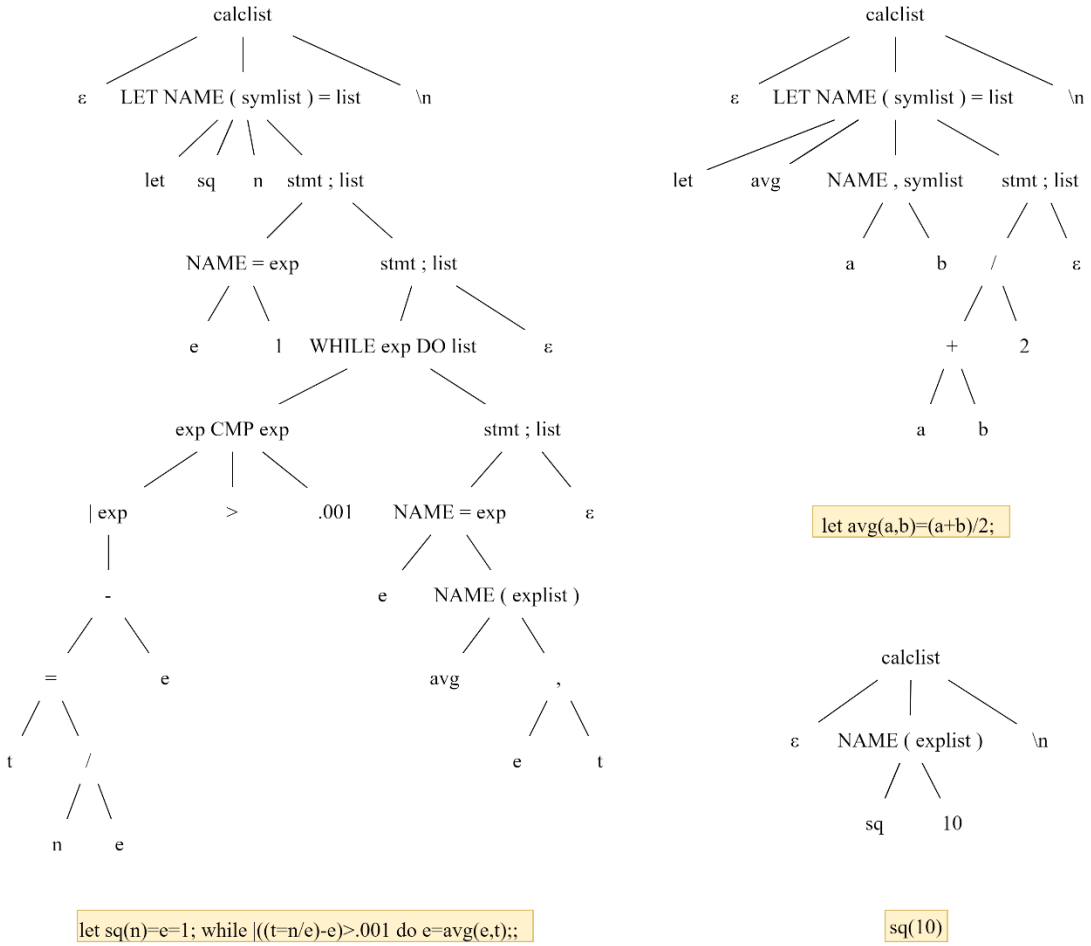

图 4 构建本实验部分抽象语法树

6.6 实验总结

6.6.1 问题

没有遇到运行代码的问题，示例代码适当修改声明，程序就直接运行出来了。实验过程中主要是阅读源码存在困难，关于协同工作的过程需要自己花时间理清，问题在阅读参考书、查阅论坛社区解答、询问同学后都得到了解决。

6.6.2 评价

本次实验完成的程序是一个可编程计算器，相比上一个实验的四则运算，又加入了编程语言，使得计算器的功能更加丰富，功能的客制化让计算器可以为用户所自行设计算法并使用。同时自己通过阅读源码绘制了本实验的抽象语法树，从而加深了对抽象语法树和语法分析的理解。

6.6.3 收获

通过对函数调用是传来的实参列表对应的抽象语法树深入分析，我了解了实参列表的结构以及计算参数值的方法，因此可以找到每个参数并计算其值。代码较前面几次实验都复杂了很多，需要自己仔细研究源码的写法，因此也更加深入地理解了抽象语法树的构造过程以及如何应用，同时还学会了使用符号表对系统的符号进行管理。在语法分析器编程上，还有很多东西要实践。