

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 2020 年 5 月 10 日

学生姓名 陆玺文

学 号 03170908

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，采用软件工程师进行系统分析、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

1、 综合实验 1	1
1.1 实验目的	1
1.2 实验内容	1
1.3 实验步骤	1
1.3.1 Windows.....	1
1.3.2 CentOS	2
1.3.3 讨论交流.....	2
1.4 移进规约分析	3
1.4.1 移进规约分析.....	3
1.4.2 语法分析树.....	4
1.5 实验总结	5
1.5.1 遇到的难题.....	5
1.5.2 实验收获.....	5
2、 综合实验 2	6
2.1 实验目的	6
2.2 实验内容	6
2.3 实验步骤	6
2.3.1 CentOS	6
2.3.2 Windows.....	7
2.4 源代码分析	7
2.4.1 fb3-2.l.....	7
2.4.2 fb3-2.y.....	9
2.4.3 fb3-2.h.....	11
2.4.4 fb3-2funcs.c.....	14
2.5 语法树构建	17
2.5.1 常量 (num)	17
2.5.2 比较运算 (cmp)	18
2.5.3 内置函数 (func)	18
2.5.4 函数调用 (call)	19
2.5.5 变量引用 (ref)	19
2.5.6 变量声明 (asgn)	19
2.5.7 条件分支语句 (flow)	20
2.5.8 表达式链 (symlist)	20
2.5.9 函数定义 (dodef)	21
2.6 语法树求值 (DEBUG 输出分析)	21
2.6.1 语法树求值.....	21
2.6.2 Debug 模式分析	26
2.6.3 常量 (num)	28
2.6.4 比较运算 (cmp) 基本运算 (+-*/)	28
2.6.5 内置函数 (func)	29

2.6.6 函数调用 (call)	30
2.6.7 函数定义 (dodef)	30
2.6.8 变量引用 (ref)	31
2.6.9 变量赋值 (asgn)	31
2.6.10 条件分支语句 (flow)	31
2.6.11 语句链表 (explist)	32
2.6.12 Debug 综合分析	32
2.7 增加内置函数 POW(A,B)	33
2.8 实验总结	34
3、 综合实验 3	35
3.1 实验目的	35
3.2 实验内容	35
3.3 实验步骤	35
3.3.1 语法修改.....	35
3.3.2 解决移进规约冲突.....	38
3.3.3 运行效果.....	38
3.4 实验总结	39
4、 综合实验 4	40
4.1 实验目的	40
4.2 实验内容	40
4.3 实验步骤	40
4.3.1 资料搜集.....	40
4.3.2 NDK 与 CMake 安装	41
4.3.3 新建 Native C++项目	41
4.3.4 搭建 UI 界面.....	44
4.3.5 导入 C 文件修改 CMakefile.....	45
4.3.6 Flex 与 Bison 源代码调整	45
4.3.7 演示效果.....	46
4.4 实验总结	47

1、综合实验 1

1.1 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：

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

1.2 实验内容

- 1. 阅读 flex Python 第三章 P47~60，重点学习抽象语法树；
- 2. 阅读 fb3-1.y、fb3-1.l、fb3-1funcs.c、fb3-1.h；
- 3. 撰写实验报告，结合实验结果，给出移进规约过程，即抽象语法树的构建过程，如 $(1+2)-(2*6)$ 、 $1+2-3*2/5$ ；
- 4. 提交报告和实验代码。

1.3 实验步骤

1.3.1 Windows

在 Windows 下编译运行 flex 与 bison 文件后，编译与链接的过程比较顺利，没有出现 bug 即通过了。但在开始运行示例程序时，出现了如图 1-1 所示的数值异常情况。

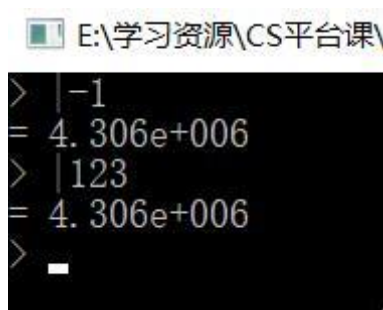


图 1-1 Win 下数值异常情况

根据编程的经验判断，应该是 windows 系统下浮点数运算出现了问题。返回 lex 源文件去观察，在模式匹配的数值匹配动作部分，lex 返回值中，使用了

atoi()函数，该函数的作用是将字符串转化为 int 整数。所以推测是在 windows 系统下，该函数失灵的缘故。导致最终输出的浮点数异常。

在查阅资料之后，选择将该函数修改为 sscanf(yytext,"%lf",&yyval.d)。如图 1-2 所示。

```
."?[0-9]+{EXP}? { sscanf(yytext,"%lf",&yyval.d); return NUMBER; }
```

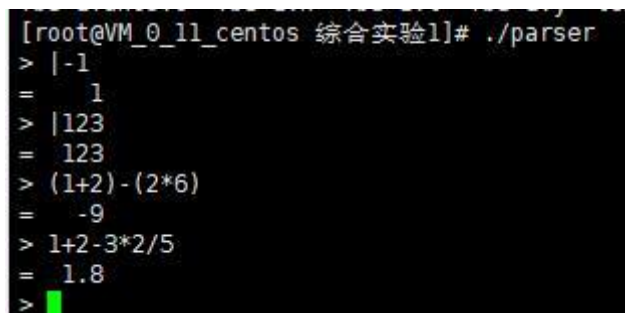
图 1-2 模式匹配动作修改

1.3.2 CentOS

在 CentOS 系统下整体运行非常顺利。输入如下编码即可成功编译 flex、bison 源文件。

```
# flex fb3-1.l
# bison -d fb3-1.y
# gcc -o parser fb3-1funcs.c lex.yy.c y.tab.c
```

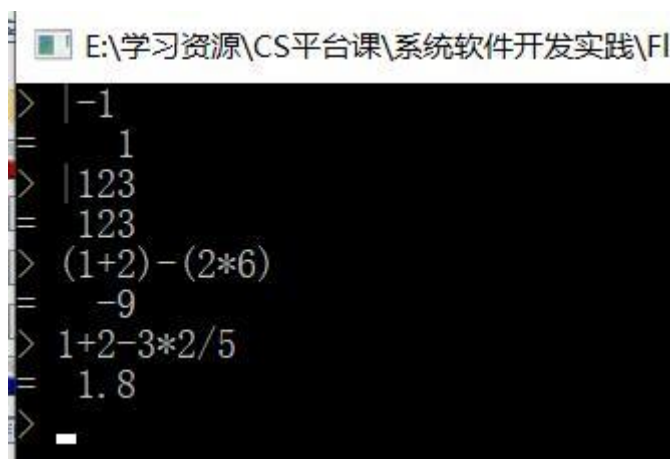
运行所给代码段，效果如图 1-3 所示。



```
[root@VM_0_11_centos 综合实验1]# ./parser
> |-1
= 1
> |123
= 123
> |(1+2)-(2*6)
= -9
> |1+2-3*2/5
= 1.8
>
```

图 1-3 CentOS 下结果示意图

1.3.3 讨论交流



```
E:\学习资源\CS平台课\系统软件开发实践\FI
> |-1
= 1
> |123
= 123
> |(1+2)-(2*6)
= -9
> |1+2-3*2/5
= 1.8
>
```

图 1-4 Windows 下运行示意图

在讨论区和其他同学针对 1.3.1 中 Win 下的问题进行了更加深入的讨论，

发现其本质原因在于 `atoi()` 函数的正常运行，需要导入 `<math.h>` 库函数。而在 CentOS 下，因为采用 `gcc` 编译链接的方式，故可以自动导入 `math.h` 标准库。而在 Windows 下 VC 编译链接时，则无此效果。

所以，可以通过在 fb3-1.1 中显示的导入 `<math.h>` 库来解决此问题。最终运行效果如图 1-4 所示。

1.4 移进规约分析

Flex 与 Bison 组合构成的编译器遵从 LALR 分析方法，向前看一个字符。所以根据在 `lex` 文件识别的标识符与 `yacc` 文件中规定的文法以及识别动作，可以做如下移进规约分析。

1.4.1 移进规约分析

以 $(1+2)-(2*6)$ 为例，其移进规约分析如表 1-1 所示。

表 1-1 移进规约分析表

栈	输入	动作
#	$(1+2)-(2*6)\#$	移进
#($1+2)-(2*6)\#$	移进
#(1	$+2)-(2*6)\#$	规约 <code>term→NUMBER</code>
#(term	$+2)-(2*6)\#$	规约 <code>factor→term</code>
#(factor	$+2)-(2*6)\#$	规约 <code>exp→factor</code>
#(exp	$+2)-(2*6)\#$	移进
#(exp+	$2)-(2*6)\#$	移进
#(exp+2	$)-(2*6)\#$	规约 <code>term→NUMBER</code>
#(exp+term	$)-(2*6)\#$	规约 <code>factor→term</code>
#(exp+factor	$)-(2*6)\#$	规约 <code>exp→exp+factor</code>
#(exp	$)-(2*6)\#$	移进
#(exp)	$-(2*6)\#$	规约 <code>term→(exp)</code>
#term	$-(2*6)\#$	规约 <code>factor→term</code>
#factor	$-(2*6)\#$	规约 <code>exp→factor</code>
#exp	$-(2*6)\#$	移进
#exp-	$(2*6)\#$	移进
#exp-($2*6)\#$	移进
#exp-(2	$*6)\#$	规约 <code>term→NUMBER</code>
#exp-(term	$*6)\#$	规约 <code>factor→term</code>
#exp-(factor	$*6)\#$	移进
#exp-(factor*	$6)\#$	移进
#exp-(factor*6	$)\#$	规约 <code>term→NUMBER</code>
#exp-(factor*term	$)\#$	规约 <code>factor→factor*term</code>
#exp-(factor	$)\#$	规约 <code>exp→factor</code>
#exp-(exp	$)\#$	移进
#exp-(exp)	$\#$	规约 <code>term→(exp)</code>

#exp-term	#	规约 factor→term
#exp-factor	#	规约 exp→exp-factor
#exp	#	接受

1.4.2 语法分析树

根据移进规约分析，可以理顺其 flex 与 bison 组合生成代码的工作过程。同时，根据 bison 的推导规则，可以得到如图 1-5、图 1-6 所示的语法分析树。

1.4.2.1 (1+2)-(2*6)语法分析树

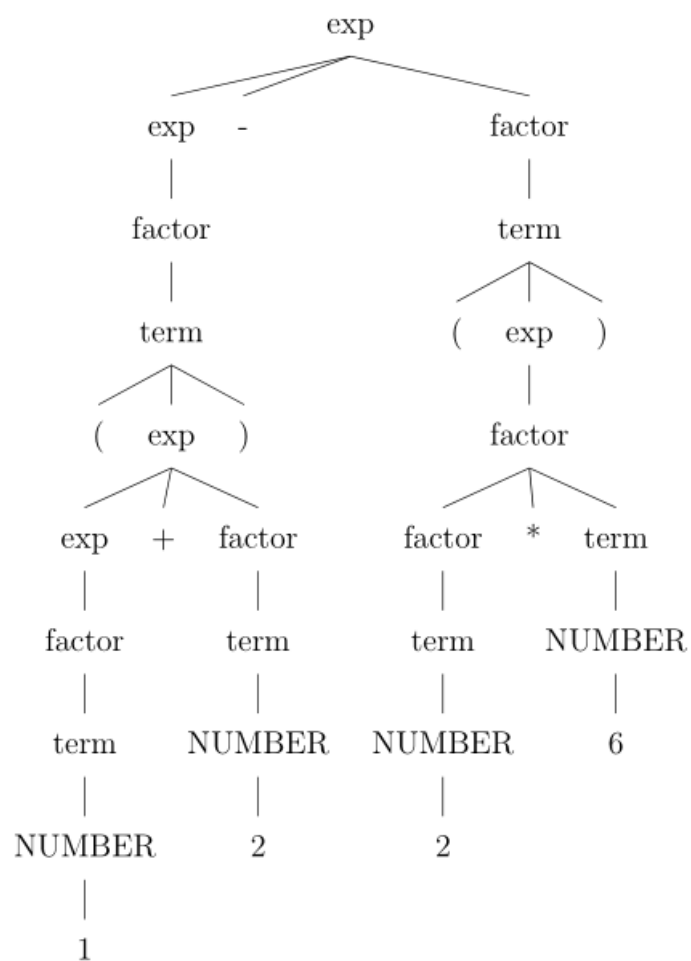


图 1-5 (1+2)-(2*6)语法分析树

1.4.2.2 1+2-3*2/5 语法分析树

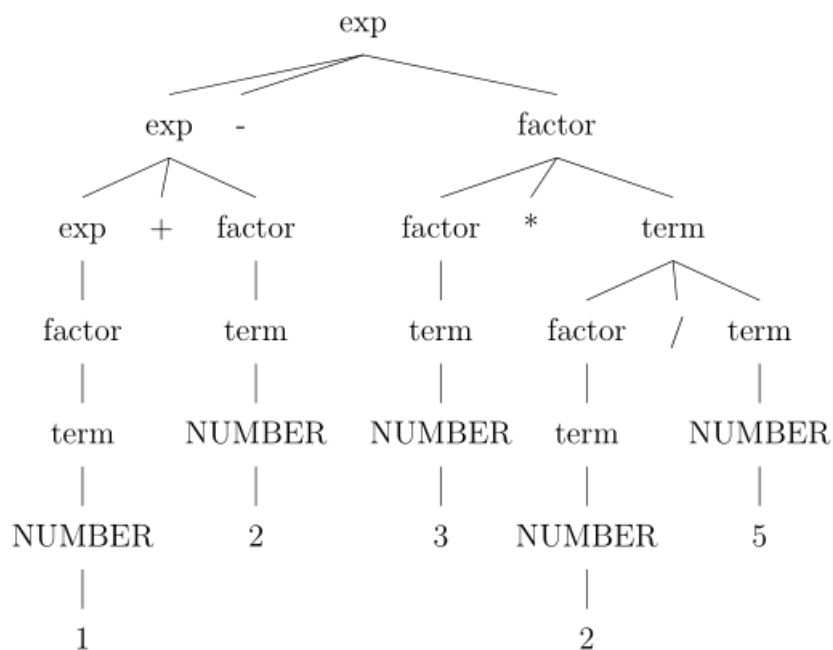


图 1-6 1+2-3*2/5 语法分析树

1.5 实验总结

1.5.1 遇到的难题

在实验指导 PDF 的帮助下，这次实验总体上十分顺利，遇到的小错误主要在于 `atoi()` 函数的处理。当然，在讨论区的充分交流下，也是学习到了关于编译命令以及 `gcc` 库的相关知识。

具体解决步骤已放于 1.3.1 中。

1.5.2 实验收获

这一次实验体验了使用 `Flex` 和 `yacc` 联合编写简单计算器并工作的步骤，对于一款《编译原理》的运用有了更加进一步的感受。

此外，在 1.3.1 的问题上，感觉到了大家讨论的智慧！百花齐放春满园！

2、综合实验 2

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：

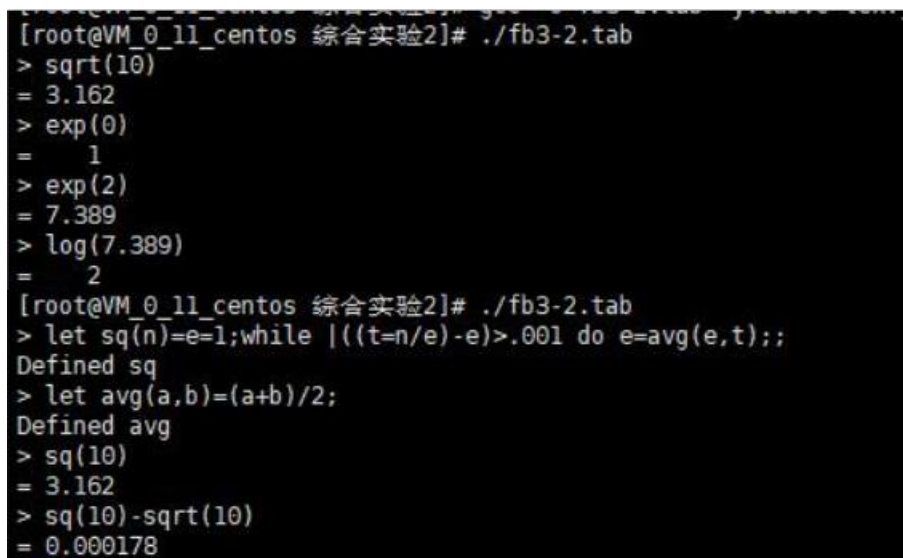
- g) 支持变量；
- h) 实现复制功能；
- i) 实现比较表达式（大于小于等）；
- j) 实现 if/then/else 和 do/while 流程控制；
- k) 用户可以自定义函数；
- l) 简单的错误恢复机制。

2.2 实验内容

- 5. 阅读 flex Python 第三章 P60~79，重点学习抽象语法树；
- 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；
- 10. 提交报告和实验代码。

2.3 实验步骤

2.3.1 CentOS



```
[root@VM_0_11_centos 综合实验2]# ./fb3-2.tab
> sqrt(10)
= 3.162
> exp(0)
= 1
> exp(2)
= 7.389
> log(7.389)
= 2
[root@VM_0_11_centos 综合实验2]# ./fb3-2.tab
> 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 CentOS 运行结果图

在 CentOS 下执行比较顺利，直接编译 flex 与 yacc 文件，同时 gcc 加入 -lm 命令编译链接 c 文件即可，运行示例如图 2-1 所示。

2.3.2 Windows

在综合实验 1 的基础上，实验 2 比较顺利，通过添加 <math.h> 库后，编译完 flex 与 bison 源文件后执行如下的代码编译链接 C 文件即可顺利执行。运行结果如图 2-2 所示。

```
cl yy.lex.c fb3-2funcs.c y.tab.c

> sqrt(10)
= 3.162
> exp(2)
= 7.389
> log(7.39)
= 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 Windows 下运行结果图

2.4 源代码分析

脱离了代码的任何分析都是无本之木，故在综合实验 2 中，结合此次实验的 flex、yacc、c 文件，详细分析源代码，进而分析出其各种数据类型树节点的建立遍历过程。

下面，将按照从高级到底层的顺序，详细介绍在实验中我分析源代码的顺序方法，以及分析结果。

2.4.1 fb3-2.l

fb3-2.l 是词法分析的 lex 文件，在多次实验的基础上，对其内容比较熟悉了。如代码 2-1 所示。

代码 2-1 fb3-2.l

```
1. %option noyywrap nodefault yylineno
2. %{
3. # include "fb3-2.h"
4. # include "y.tab.h"
```

```
5. #include <math.h>
6. %}
7. EXP ([Ee][--]?[0-9]+)
8.
9. %%
10. /* single character ops */
11. "+" |
12. "-" |
13. "*" |
14. "/" |
15. "=" |
16. "|" |
17. "," |
18. ";" |
19. "(" |
20. ")" { return yytext[0]; }
21.
22. /* comparison ops */
23. ">" { yylval.fn = 1; return CMP; }
24. "<" { yylval.fn = 2; return CMP; }
25. "<>" { yylval.fn = 3; return CMP; }
26. "==" { yylval.fn = 4; return CMP; }
27. ">=" { yylval.fn = 5; return CMP; }
28. "<=" { yylval.fn = 6; return CMP; }
29.
30. /* keywords */
31. "if" { return IF; }
32. "then" { return THEN; }
33. "else" { return ELSE; }
34. "while" { return WHILE; }
35. "do" { return DO; }
36. "let" { return LET; }
37.
38. /* built in functions */
39. "sqrt" { yylval.fn = B_sqrt; return FUNC; }
40. "exp" { yylval.fn = B_exp; return FUNC; }
41. "log" { yylval.fn = B_log; return FUNC; }
42. "pow" { yylval.fn = B_pow; return FUNC; }
43. "print" { yylval.fn = B_print; return FUNC; }
44.
45. /* debug hack */
46. "debug"[0-9]+ { debug = atoi(&yytext[5]); printf("debug
    set to %d\n", debug); }
47.
```

```

48. /* names */
49. [a-zA-Z][a-zA-Z0-9]* { yyval.s = lookup(yytext); return
    NAME; }
50.
51. [0-9]+."[0-9]*{EXP}? |
52. ."? [0-9]+{EXP}? { yyval.d =atof(yytext);return NUMBER;}
53.
54. "/".*
55. [ \t] /* ignore white space */
56. \\n printf("c> "); /* ignore line continuation */
57. "\n" { return EOL; }
58.
59. . { yyerror("Mystery character %c\n", *yytext); }
60. %%

```

其中需要注意，在第 38-40 行，定义了程序中的内置函数，识别之后，直接返回 FUNC。

第 52 行，针对浮点数，则直接返回 NUMBER，同时程序中的节点采用共用体形式（代码 2-2 中第 7-12 行）来存储数据值，浮点数的值存储在 double 型成员 d 中。

第 57 行，规定了识别到 '\n' 时，返回 EOL，仅此符号可以。

2.4.2 fb3-2.y

fb3-2.y 是语法分析的 yacc 文件，其中包含了所有能够识别的文法规则，并采用语法制导翻译的方式，给出了每一次匹配之后相应的执行动作。对这些动作的理解，是理解构造语法树的关键，更是遍历语法树计算表达式值的基础前提。

代码 2-2 fb3-2.y

```

1. %{
2. # include <stdio.h>
3. # include <stdlib.h>
4. # include "fb3-2.h"
5. %}
6.
7. %union {
8. struct ast *a;
9. double d;
10. struct symbol *s; /* which symbol */
11. struct symlist *sl;
12. int fn; /* which function */
13. }
14.

```

```

15. /* declare tokens */
16. %token <d> NUMBER
17. %token <s> NAME
18. %token <fn> FUNC
19. %token EOL
20. %token IF THEN ELSE WHILE DO LET
21.
22. %nonassoc <fn> CMP
23. %right '='
24. %left '+' '-'
25. %left '*' '/'
26. %nonassoc '|' UMINUS
27.
28. %type <a> exp stmt list explist
29. %type <sl> symlist
30.
31. %start calclist
32.
33. %%
34. stmt: IF exp THEN list    { $$=newflow('I',$2,$4,NULL); }
35.    | IF exp THEN list ELSE list { $$ = newflow('I', $2,
    $4, $6); }
36.    | WHILE exp DO list { $$ = newflow('W',$2,$4, NULL); }
37.    | exp
38.    ;
39.
40. list: /* nothing */ { $$ = NULL; }
41.    | stmt ';' list { if ($3 == NULL)
42.        $$ = $1;
43.        else
44.        $$ = newast('L', $1, $3);
45.        }
46.    ;
47.
48. exp: exp CMP exp          { $$ = newcmp($2, $1, $3); }
49.    | exp '+' exp          { $$ = newast('+', $1,$3); }
50.    | exp '-' exp          { $$ = newast('-', $1,$3); }
51.    | exp '*' exp          { $$ = newast('*', $1,$3); }
52.    | exp '/' exp          { $$ = newast('/', $1,$3); }
53.    | '|' exp              { $$ = newast('|', $2, NULL); }
54.    | '(' exp ')'          { $$ = $2; }
55.    | '-' exp %prec UMINUS { $$ = newast('M', $2, NULL); }
56.    | NUMBER                { $$ = newnum($1); }
57.    | FUNC '(' explist ')' { $$ = newfunc($1, $3); }

```

```

58. | NAME                                { $$ = newref($1); }
59. | NAME '=' exp                        { $$ = newasgn($1, $3); }
60. | NAME '(' explist ')' { $$ = newcall($1, $3); }
61. ;
62.
63. explist: exp
64. | exp ',' explist { $$ = newast('L', $1, $3); }
65. ;
66. symlist: NAME { $$ = newsymlist($1, NULL); }
67. | NAME ',' symlist { $$ = newsymlist($1, $3); }
68. ;
69.
70. calclist: /* nothing */
71. | calclist stmt EOL {
72.     if(debug) dumpast($2, 0);
73.     printf("= %4.4g\n> ", eval($2));
74.     treefree($2);
75. }
76. | calclist LET NAME '(' symlist ')' '=' list EOL {
77.     dodef($3, $5, $8);
78.     printf("Defined %s\n> ", $3->name); }
79. | calclist error EOL { yyerrok; printf("> "); }
80. ;
81. %%

```

代码 2-2 中，可以看到，在第 72 行中给出了`debug`模式的动作规则，说明在这一份示例的计算器程序中支持 debug 模式的开启。在 2.6.2 中将结合本节对于代码的分析，详细介绍 Debug 模式的输出结果含义，以及作用。

代码 2-2 中，几乎所有的动作中，都调用了相应类型的创建节点函数。仅在 73 行中，调用了 eval()来进行结果打印。可以分析出，程序的工作时序是先创建语法树节点，后通过语法树计算结果值。

2.4.3 fb3-2.h

这一份是内置的头文件，给出了各种类型节点的定义，以及将要实现的函数部分，详细代码如代码 2-3 所示。

代码 2-3 fb3-2.h

```

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];
12.
13. struct symbol *lookup(char*);
14.
15. /* list of symbols, for an argument list */
16. struct symlist {
17.     struct symbol *sym;
18.     struct symlist *next;
19. };
20.
21. struct symlist *newsymlist(struct symbol *sym, struct
    symlist *next);
22. void symlistfree(struct symlist *sl);
23.
24. /* node types
25. * + - * / |
26. * 0-7 comparison ops, bit coded 04 equal, 02 less, 01
greater
27. * M unary minus
28. * L statement list
29. * I IF statement
30. * W WHILE statement
31. * N symbol ref
32. * = assignment
33. * S list of symbols
34. * F built in function call
35. * C user function call
36. */
37.
38. enum bifs { /* built-in functions */
39.     B_sqrt = 1,
40.     B_exp,
41.     B_log,
42.     B_pow,
43.     B_print
44. };
45.
46. /* nodes in the Abstract Syntax Tree */
47. /* all have common initial nodetype */
48.
```

```
49. struct ast {
50.     int nodetype;
51.     struct ast *l;
52.     struct ast *r;
53. };
54.
55. struct fncall {          /* built-in function */
56.     int nodetype;        /* type F */
57.     struct ast *l;
58.     enum bifs functype;
59. };
60.
61. struct ufncall {         /* user function */
62.     int nodetype;        /* type C */
63.     struct ast *l;       /* list of arguments */
64.     struct symbol *s;
65. };
66.
67. struct flow {
68.     int nodetype;        /* type I or W */
69.     struct ast *cond;     /* condition */
70.     struct ast *tl;       /* then or do list */
71.     struct ast *el;       /* optional else list */
72. };
73.
74. struct numval {
75.     int nodetype;        /* type K */
76.     double number;
77. };
78.
79. struct symref {
80.     int nodetype;        /* type N */
81.     struct symbol *s;
82. };
83.
84. struct symasn {
85.     int nodetype;        /* type = */
86.     struct symbol *s;
87.     struct ast *v;       /* value */
88. };
89.
90. /* build an AST */
91. struct ast *newast(int nodetype, struct ast *l, struct ast
    *r);
```

```

92. struct ast *newcmp(int cmptype, struct ast *l, struct ast
    *r);
93. struct ast *newfunc(int functype, struct ast *l);
94. struct ast *newcall(struct symbol *s, struct ast *l);
95. struct ast *newref(struct symbol *s);
96. struct ast *newasgn(struct symbol *s, struct ast *v);
97. struct ast *newnum(double d);
98. struct ast *newflow(int nodetype, struct ast *cond, struct
    ast *tl, struct ast *tr);
99.
100 /* define a function */
101 void dodef(struct symbol *name, struct symlist *syms,
    struct ast *stmts);
102 /* evaluate an AST */
103 double eval(struct ast *);
104 /* delete and free an AST */
105 void treefree(struct ast *);
106 /* interface to the lexer */
107 extern int yylineno; /* from lexer */
108 void yyerror(char *s, ...);
109
110 extern int debug;
111 void dumpast(struct ast *a, int level);

```

第 1-7 行，给出了符号表的数据结构定义，11、13 行给出了对于符号表的存储于查找。

15-22 行号，给出了符号列表，参数列表的定义（这一部分个人认为是在函数定义中可以并列放置多条语句，是一个 **exp** 简单表达式链），对应于代码 2-2 中 64 行。

38-44 行，给出了内置函数的枚举型变量类型¹，便于查找调用。

第 49-53 行，定义了抽象语法树的通用节点结构，包含节点类型值、左子树、右子树。

第 57-8 行，分别定义了，针对具体类型的具体节点结构，包括内置函数、用户定义函数、条件分支语句、常数、变量声明、变量引用。

第 91-98 行，定义了所有将要实现的，对于不同类型节点的新建函数。

第 103 行，声明了求值函数 `eval()`。

第 112 行，声明了在 `debug` 工作模式下的打印函数 `dumpast()`。

2.4.4 fb3-2funcs.c

¹ 此处代码本人已修改，源代码不包含 `pow(a,b)`

fb3-2funcs.c 是给出了在 fb3-2.h 头文件中函数声明的具体实现，其详细代码较多，分块介绍如下。

2.4.4.1 散列符号表

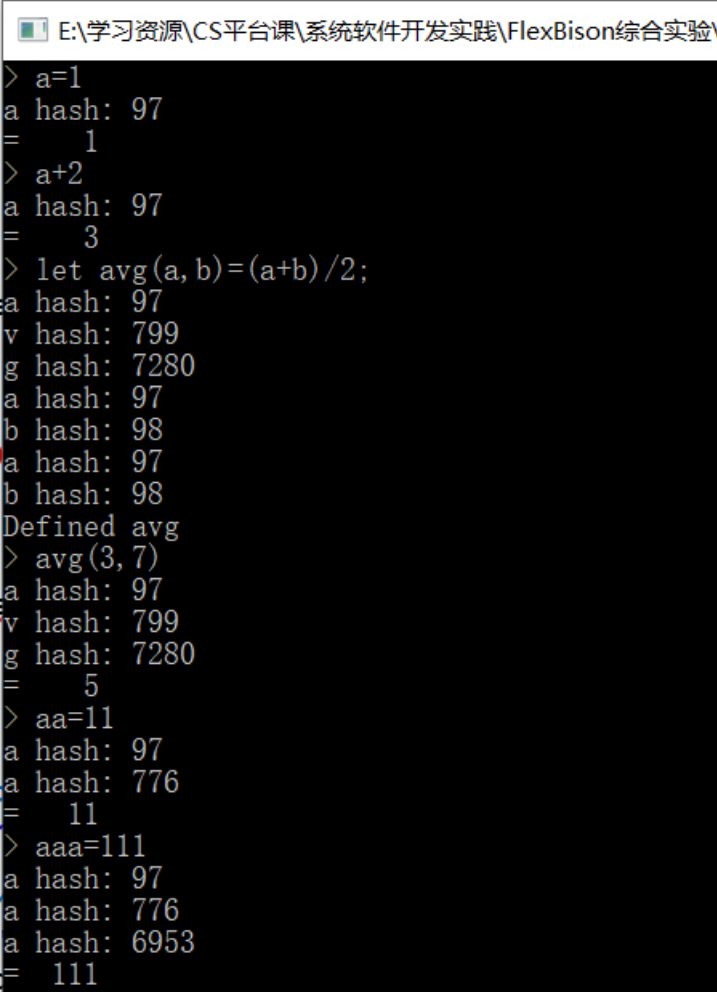
在代码的开头，给出了符号表求 hash 的函数 symhash(char*)，通过不断乘 9 取异或来获得该符号的 hash 值。通过在代码 2-4 中增加第 9 行，打印对应字符时的 hash，来分析部分语句的输出效果如图 2-3 所示。

代码 2-4 fb3-2funcs.c 中散列符号表操作

```

1.  /* symbol table */
2.  /* hash a symbol */
3.  static unsigned symhash(char *sym)
4.  {
5.      unsigned int hash = 0;
6.      unsigned c;
7.      while(c = *sym++) {
8.          hash = hash*9 ^ c;
9.          printf("%c hash: %d\n",c,hash);
10.     }
11.     return hash;
12. }
13.
14. struct symbol * lookup(char* sym)
15. {
16.     struct symbol *sp = &syntab[symhash(sym)%NHASH];
17.     int scout = NHASH;    /* how many have we looked at */
18.
19.     while(--scout >= 0) {
20.         if(sp->name && !strcmp(sp->name, sym)) { return sp; }
21.
22.         if(!sp->name) {    /* new entry */
23.             sp->name = strdup(sym);
24.             sp->value = 0;
25.             sp->func = NULL;
26.             sp->syms = NULL;
27.             return sp;
28.         }
29.         /* try the next entry */
30.         if(++sp >= syntab+NHASH) sp = syntab;
31.     }
32.     yyerror("symbol table overflow\n");
33.     abort(); /* tried them all, table is full */
34. }

```

A screenshot of a terminal window with a black background and white text. The window title is "E:\学习资源\CS平台课\系统软件开发实践\FlexBison综合实验\". The terminal shows a series of commands and their outputs, illustrating how the hash of the variable 'a' changes as its value is updated. The commands and outputs are: 1. Command: > a=1; Output: a hash: 97, = 1. 2. Command: > a+2; Output: a hash: 97, = 3. 3. Command: > let avg(a,b)=(a+b)/2; Output: a hash: 97, v hash: 799, g hash: 7280, a hash: 97, b hash: 98, a hash: 97, b hash: 98, Defined avg. 4. Command: > avg(3,7); Output: a hash: 97, v hash: 799, g hash: 7280, = 5. 5. Command: > aa=11; Output: a hash: 97, a hash: 776, = 11. 6. Command: > aaa=111; Output: a hash: 97, a hash: 776, a hash: 6953, = 111.

```
> a=1
a hash: 97
= 1
> a+2
a hash: 97
= 3
> let avg(a,b)=(a+b)/2;
a hash: 97
v hash: 799
g hash: 7280
a hash: 97
b hash: 98
a hash: 97
b hash: 98
Defined avg
> avg(3,7)
a hash: 97
v hash: 799
g hash: 7280
= 5
> aa=11
a hash: 97
a hash: 776
= 11
> aaa=111
a hash: 97
a hash: 776
a hash: 6953
= 111
```

图 2-3 散列符号表输出分析

从图 2-3 的输出中，可以看到，定义了单个变量 a 后，程序执行 $0*9^a$ ，即 0^a ，输出不变，为 a 的 ASCII 码值 97。

在执行 $aa=11$ 语句时，第二个 a 的 hash 计算式为 $97*9^a=873^97=776$ 。

同理可以分析，当执行 $aaa=111$ 语句时，第三个 a 的 hash 计算式为 $776*9^a=6984^97=6953$ 。

故，当用户声明了变量或定义了函数后，符号表对应 hash 索引处便存储了该抽象语法树的根，当再次识别到该符号时，则调用相关根节点，进行访问。完成函数调用。

2.4.4.2 抽象语法树根节点

ast 是抽象语法树的根节点，也是最为基础的树节点，其数据类型在代码

2-3 中有所展示，非常简单，包含节点类型 `Int` 值，以及左右子树。在代码 2-5 中展示了创建 `ast` 节点的代码，基本二元运算或者表达式列表，均会调用之。

代码 2-5 `newast(int, struct ast*, struct ast*)`

```

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

```

2.4.4.3 抽象语法树释放

`Treefree(struct ast*)` 函数，通过针对不同子树数量的语法树节点操作，释放相应的语法树。其代码较为简单，通过 `switch` 语句判断各个节点类型后，调用相应的释放操作，节省篇幅，此处不放置源代码。

2.5 语法树构建

在 2.4 部分，了解了各个函数的作用，以及各个文件中函数的相互调用。这一部分，将结合具体的示例，详细分析各种类型的语法树节点建立过程。语法树的构建，是表达是求值的基础。

为了进一步的更好分析，首先罗列 `fb3-2.h` 中支持的类型，在代码 2-3 中可以看到，定义了内置函数（`fncall`）、用户函数（`ufncall`）、分支语句（`flow`）、数值（`numval`）、符号引用（`symref`）、符号声明（`symasn`）。下面结合 `fb3-2funcs.c` 中的树节点建立代码详细分析。

2.5.1 常量（`num`）

代码 2-6 `newnum(double)`

```

1. struct ast * newnum(double d)
2. {
3.     struct numval *a = malloc(sizeof(struct numval));
4.
5.     if(!a) {

```

```
6.     yyerror("out of space");
7.     exit(0);
8. }
9. a->nodetype = 'K';
10. a->number = d;
11. return (struct ast *)a;
12. }
```

常量节点类型比较简单，包含节点类型‘K’，数值 number，最后转换为 ast* 返回。

2.5.2 比较运算 (cmp)

代码 2-7 newcmp(int, struct ast*, struct ast*)

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

比较运算节点类型包含左右子树，分别对应前后两个操作数，此外，节点类型为数值 cmptype，在代码 2-1 中第 21-28 行，定义了各种不同比较运算的内置 cmptype 值。进而通过其可以确定具体是何种比较运算。

2.5.3 内置函数 (func)

代码 2-8 newfunc(int, struct ast*)

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

```

11. a->functype = functype;
12. return (struct ast *)a;
13. }

```

内置函数节点中，节点类型为‘F’，包含子树 l，此外还有枚举型 bifs（代码 2-3 中 38-44 行）函数类型 functype。初始时，定义了 sqrt、exp、log、print 这 4 种内置函数类型。

2.5.4 函数调用 (call)

代码 2-9 newcall(struct symbol*,struct ast*)

```

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

```

用户自定义函数部分，包含函数节点类型‘C’，以及左右子树，左子树链接函数名对应的语法树，右子树对应参数列表。

2.5.5 变量引用 (ref)

代码 2-10 newref(struct symbol*)

```

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

```

引用变量部分，节点类型‘N’，包含一个子树，链接对应的变量。

2.5.6 变量声明 (asgn)

代码 2-11 newasgn(struct symbol*, struct ast*)

```

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

```

变量声明部分，节点类型 ‘=’，子树分别为变量名以及变量值。

2.5.7 条件分支语句 (flow)

代码 2-12 newflow(int, struct ast*)

```

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

```

条件分支语句部分，包含节点类型 ‘I’、‘M’（代码 2-2 中 34-38 行），分别表示 if 语句与 while 语句。子树记录了状态 condition，以及接下来的语句链表，或是可能的 else 分支。

2.5.8 表达式链 (symlist)

代码 2-13 newsymlist(struct symbol *, struct symlist *)

```

1. struct symlist * newsymlist(struct symbol *sym, struct
   symlist *next)
2. {

```

```

3.     struct symlist *sl = malloc(sizeof(struct symlist));
4.
5.     if(!sl) {
6.         yyerror("out of space");
7.         exit(0);
8.     }
9.     sl->sym = sym;
10.    sl->next = next;
11.    return sl;
12. }
13.
14. void symlistfree(struct symlist *sl)
15. {
16.     struct symlist *nsl;
17.
18.     while(sl) {
19.         nsl = sl->next;
20.         free(sl);
21.         sl = nsl;
22.     }
23. }

```

Symlist 主要在函数参数列表中，用于维护一个参数链表。其不使用 ast 的数据结构，仅维护自身符号与下一个节点的符号。

2.5.9 函数定义 (dodef)

代码 2-14 dodef(struct symbol*, struct symlist *, struct ast *)

```

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

```

函数定义部分，首先，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。

2.6 语法树求值 (Debug 输出分析)

2.6.1 语法树求值

在 2.5 中介绍了，由 yacc 语句与各个 C 函数的作用下，不同语法树节点的

构建。在构建完成语法树，语句规约到 S0 状态，即执行代码 2-2 中第 73 行语句时，程序会调用相应的 eval(struct ast *)函数，这也是程序通过语法树求值的关键函数，其代码如代码 2-15 所示。

代码 2-15 eval(struct ast*)

```

1.  double eval(struct ast *a)
2.  {
3.      double v;
4.
5.      if(!a) {
6.          yyerror("internal error, null eval");
7.          return 0.0;
8.      }
9.
10.     switch(a->nodetype) {
11.         /* constant */
12.         case 'K': v = ((struct numval *)a)->number; break;
13.
14.         /* name reference */
15.         case 'N': v = ((struct symref *)a)->s->value; break;
16.
17.         /* assignment */
18.         case '=': v = ((struct symasn *)a)->s->value =
19.             eval(((struct symasn *)a)->v); break;
20.
21.         /* expressions */
22.         case '+': v = eval(a->l) + eval(a->r); break;
23.         case '-': v = eval(a->l) - eval(a->r); break;
24.         case '*': v = eval(a->l) * eval(a->r); break;
25.         case '/': v = eval(a->l) / eval(a->r); break;
26.         case '|': v = fabs(eval(a->l)); break;
27.         case 'M': v = -eval(a->l); break;
28.
29.         /* comparisons */
30.         case '1': v = (eval(a->l) > eval(a->r))? 1 : 0; break;
31.         case '2': v = (eval(a->l) < eval(a->r))? 1 : 0; break;
32.         case '3': v = (eval(a->l) != eval(a->r))? 1 : 0; break;
33.         case '4': v = (eval(a->l) == eval(a->r))? 1 : 0; break;
34.         case '5': v = (eval(a->l) >= eval(a->r))? 1 : 0; break;
35.         case '6': v = (eval(a->l) <= eval(a->r))? 1 : 0; break;
36.
37.         /* control flow */
38.         /* null if/else/do expressions allowed in the grammar,
           so check for them */

```

```

39.  case 'I':
40.     if( eval( ((struct flow *)a)->cond) != 0) {
41.         if( ((struct flow *)a)->t1) {
42.             v = eval( ((struct flow *)a)->t1);
43.         } else
44.             v = 0.0;      /* a default value */
45.         } else {
46.             if( ((struct flow *)a)->el) {
47.                 v = eval(((struct flow *)a)->el);
48.             } else
49.                 v = 0.0;      /* a default value */
50.         }
51.         break;
52.
53.  case 'W':
54.     v = 0.0;      /* a default value */
55.
56.     if( ((struct flow *)a)->t1) {
57.         while( eval(((struct flow *)a)->cond) != 0)
58.             v = eval(((struct flow *)a)->t1);
59.     }
60.     break;      /* last value is value */
61.
62.  case 'L': eval(a->l); v = eval(a->r); break;
63.
64.  case 'F': v = callbuiltin((struct fncall *)a); break;
65.
66.  case 'C': v = calluser((struct ufncall *)a); break;
67.
68.  default: printf("internal error: bad node %c\n",
69.                 a->nodetype);
70.  }
71.  return v;
72. }

```

在代码 2-15 中，主体上根据不同的节点类型，可以结合存储时的数据结构，返回其语义值，比如常量‘K’型直接返回其 number，分支语句则根据状态 cond 进行判断该执行那一条。

其中，需要注意的是第 62 行，对于表达式语句的计算，会计算遍历访问计算左子树的节点，但是 return 值 v 时，该值仅表示右子树值 a->r，这一点符合我们在高级语言中，逗号表达式的用法。

此外，还可以看到对于调用了内置函数（第 64 行）以及用户自定义函数（第 66 行）的两种计算，代码分别如代码 2-16、代码 2-17 所示。

2.6.1.1 计算内置函数返回值

代码 2-16 callbuiltin(struct fncall*)

```
1. static double callbuiltin(struct fncall *f)
2. {
3.     enum bifs functype = f->functype;
4.     double v = eval(f->l);
5.
6.     switch(functype) {
7.     case B_sqrt:
8.         return sqrt(v);
9.     case B_exp:
10.        return exp(v);
11.    case B_log:
12.        return log(v);
13.    case B_pow:
14.        return pow(eval(f->l->l),v);
15.    case B_print:
16.        printf("= %4.4g\n", v);
17.        return v;
18.    default:
19.        yyerror("Unknown built-in function %d", functype);
20.        return 0.0;
21.    }
22. }
```

在内置函数计算部分，通过 eval 求出函数中参数值，再通过节点类型中 functype 的判断，来找到具体是哪一种内置函数，进而调用相应函数去求值。

2.6.1.2 计算用户自定义函数返回值

代码 2-17 calluser(struct ufncall *)

```
1. static double calluser(struct ufncall *f)
2. {
3.     struct symbol *fn = f->s; /* function name */
4.     struct symlist *sl;      /* dummy arguments */
5.     struct ast *args = f->l; /* actual arguments */
6.     double *oldval, *newval; /* saved arg values */
7.     double v;
8.     int nargs;
9.     int i;
10. }
```

```
11.  if(!fn->func) {
12.      yyerror("call to undefined function", fn->name);
13.      return 0;
14.  }
15.
16.  /* count the arguments */
17.  sl = fn->syms;
18.  for(nargs = 0; sl; sl = sl->next)
19.      nargs++;
20.
21.  /* prepare to save them */
22.  oldval = (double *)malloc(nargs * sizeof(double));
23.  newval = (double *)malloc(nargs * sizeof(double));
24.  if(!oldval || !newval) {
25.      yyerror("Out of space in %s", fn->name); return 0.0;
26.  }
27.
28.  /* evaluate the arguments */
29.  for(i = 0; i < nargs; i++) {
30.      if(!args) {
31.          yyerror("too few args in call to %s", fn->name);
32.          free(oldval); free(newval);
33.          return 0;
34.      }
35.
36.      if(args->nodetype == 'L') { /* if this is a list node
37.          */
38.          newval[i] = eval(args->l);
39.          args = args->r;
40.      } else { /* if it's the end of the list */
41.          newval[i] = eval(args);
42.          args = NULL;
43.      }
44.  }
45.  /* save old values of dummies, assign new ones */
46.  sl = fn->syms;
47.  for(i = 0; i < nargs; i++) {
48.      struct symbol *s = sl->sym;
49.
50.      oldval[i] = s->value;
51.      s->value = newval[i];
52.      sl = sl->next;
53.  }
```

```

54. free(newval);
55.
56. /* evaluate the function */
57. v = eval(fn->func);
58.
59. /* put the dummies back */
60. sl = fn->syms;
61. for(i = 0; i < nargs; i++) {
62.     struct symbol *s = sl->sym;
63.
64.     s->value = oldval[i];
65.     sl = sl->next;
66. }
67.
68. free(oldval);
69. return v;
70. }

```

这段用户自定义函数的调用过程，有比较多的信息。首先，传入的是一个函数名、参数列表。

第一步是计数参数个数，校验是否与内置的该函数参数个数匹配。

在匹配之后所要做的工作就是根据该函数的抽象语法树结构来计算相应的函数返回值，在此之前，需要将该抽象语法树结构中的节点值替换为传入函数的参数。这就是第二步的工作。首先读取传入的语句列表值，放置在 `newval[]` 中，然后保存原来的函数节点值，或者是重名的符号值（函数形参与全局变量重名），保存在 `oldval[]` 中。接着，遍历函数语法树，替换节点值为 `newval[]`，然后执行 `eval`，返回结果即是函数计算结果。

第三步，函数计算完成了，当然是复原原抽象语法树的节点值，再将 `oldval[]` 拿出来放回原来的语法树节点。

至此，用户调用函数的分析便完成。

2.6.2 Debug 模式分析

在 2.4.2 中介绍了，示例代码在识别到标识符 `debug` 之后，支持开启 `debug` 模式。在代码 2-2 中可以看到，开启了 `debug` 模式后，将调用 `dumppast()` 这一树遍历函数。因而，实质上，其所输出打印的，即为该处节点的语法分析树，也是递归遍历求值时的树。

与之相关的主要是 `dumppast(struct ast*,int)` 打印函数如代码 2-18 所示。

代码 2-18 `dumppast(struct ast *, int)`

```

1. void dumppast(struct ast *a, int level)
2. {

```

```

3.  printf("%*s", 2*level, "");  /* indent to this level */
4.  level++;
5.
6.  if(!a) {
7.      printf("NULL\n");
8.      return;
9.  }
10. switch(a->nodetype) {
11.     /* constant */
12.     case 'K': printf("number %4.4g\n", ((struct numval
        *)a)->number); break;
13.
14.     /* name reference */
15.     case 'N': printf("ref %s\n", ((struct symref
        *)a)->s->name); break;
16.
17.     /* assignment */
18.     case '=': printf("= %s\n", ((struct symref
        *)a)->s->name);
19.         dumpast( ((struct symasn *)a)->v, level); return;
20.
21.     /* expressions */
22.     case '+': case '-': case '*': case '/': case 'L':
23.     case '1': case '2': case '3':
24.     case '4': case '5': case '6':
25.         printf("binop %c\n", a->nodetype);
26.         dumpast(a->l, level);
27.         dumpast(a->r, level);
28.         return;
29.
30.     case '|': case 'M':
31.         printf("unop %c\n", a->nodetype);
32.         dumpast(a->l, level);
33.         return;
34.
35.     case 'I': case 'W':
36.         printf("flow %c\n", a->nodetype);
37.         dumpast( ((struct flow *)a)->cond, level);
38.         if( ((struct flow *)a)->t1)
39.             dumpast( ((struct flow *)a)->t1, level);
40.         if( ((struct flow *)a)->el)
41.             dumpast( ((struct flow *)a)->el, level);
42.         return;
43.

```

```

44. case 'F':
45.     printf("builtin %d\n", ((struct fncall
        *)a)->functype);
46.     dumpast(a->l, level);
47.     return;
48.
49. case 'C': printf("call %s\n", ((struct ufncall
        *)a)->s->name);
50.     dumpast(a->l, level);
51.     return;
52.
53. default: printf("bad %c\n", a->nodetype);
54.     return;
55. }
56. }

```

从代码 2-18 中可以看出，其遍历思路仍然是通过 `switch()` 语句判断节点类型，进而执行特定的访问操作。对于双子树节点，使用前序遍历。

接下来，通过列举几种典型的节点类型来分析其输出，同时会根据 2.5 中所分析的抽象语法树节点类型，来比较两者。

2.6.3 常量 (num)

常量类型非常简单，根据 2.5.1 中的分析，可以得到如所示的节点类型，此外 `debug` 模式下定义常量后输出如图 2-4 所示。

Numval:	Int nodetype	Double number	<pre> > debug1 a=1999 debug set to 1 = a number 1999 = 1999 </pre>
A=1999:	K	1999	

图 2-4 num 节点类型与 debug 输出 (a=1999)

结合代码 2-18 中第 12 行，`debug` 所打印的，即是 `number` 常量节点的存储值，遍历语法树节点计算结果即为 1999。很好的与抽象语法树的节点对应了。

2.6.4 比较运算 (cmp) 基本运算 (+-*/)

在 2.5.2 中分析到，`cmp` 节点使用的存储结构同样为双子树节点 `ast`，其中节点类型为根据 `lex` 文件的返回值，为内置的比较运算的编号值。以 “11 < 2” 为例，‘<’ 的内置编号为 1，可以看到图 2-5 输出了运算符 `binop` 编号 1，

以及左右子树，均为 **number** 节点，左部值 11，右部值 22。

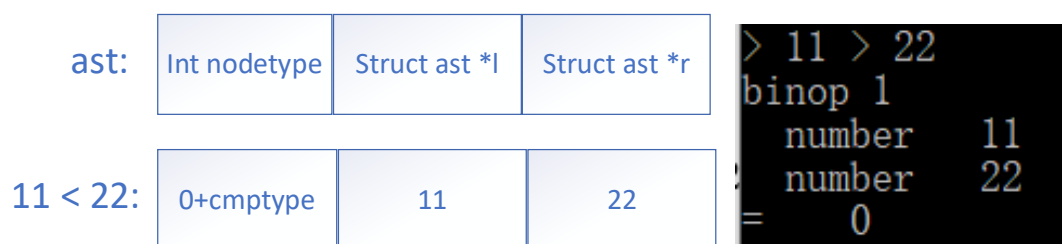
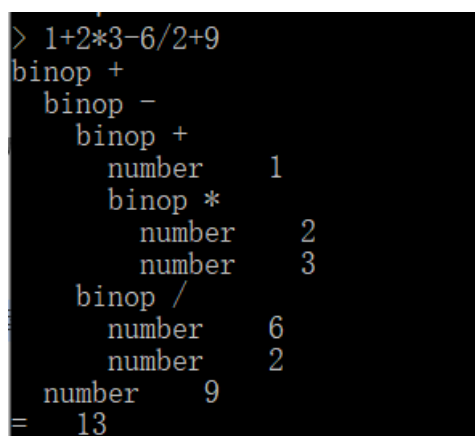


图 2-5 cmp 节点类型与 debug 输出 (11<22)

其他的比较运算符与之相类似，基本运算加减乘除因为相同的构造方式，同样可以相同的分析。



```

> 1+2*3-6/2+9
binop +
  binop -
    binop +
      number 1
      binop *
        number 2
        number 3
    binop /
      number 6
      number 2
  number 9
= 13
  
```

图 2-6 四则运算综合 (1+2*3-6/2+9)

图 2-6 中展示了相对综合的四则运算，可以看到此处构建了一颗分析树。根据在 2.6.2 中的分析，对于双子树根节点，采用前序遍历的访问方式。所以，依据这颗树，计算顺序依次为 $2*3=6$ ， $6+1=7$ ， $6/2=3$ ， $7-3=4$ ， $4+9=13$ 。

2.6.5 内置函数 (func)

在 2.5.3 中已分析到，内置函数节点类型为 **K**，包含一个 **ast*** 子树，一个函数类别值。当调用时，**ast*** 子树会根据参数而链接不同的类型，以 **sqrt(4)** 为例，如图 2-7 所示，其中链接了常量类型 **K**。

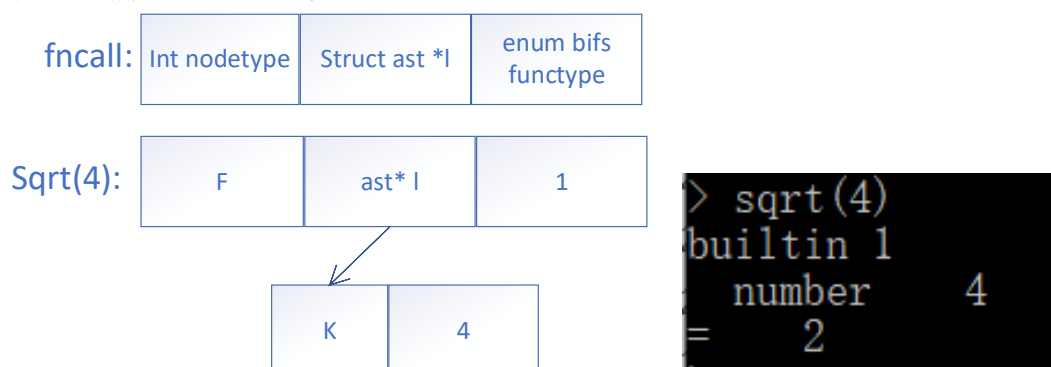


图 2-7 func 节点类型及 debug 输出 (sqrt(4))

当函数的参数更加负责，或者是一个表达式时，其语法树也相应的会有所改变，如图 2-8 所示。

```
> sqrt(1+2*2-1)
builtin 1
  binop -
    binop +
      number 1
      binop *
        number 2
        number 2
    number 1
  = 2

> sqrt(3*3)
builtin 1
  binop *
    number 3
    number 3
  = 3
>
```

图 2-8 内置函数参数复杂时 debug 输出

2.6.6 函数调用 (call)

用户定义函数 `ufncall` 类型，包含节点类型为 `C`，子树 `ast` 链接参数，此外符号类型存储函数名。在调用时，可以看到同样层层输出。

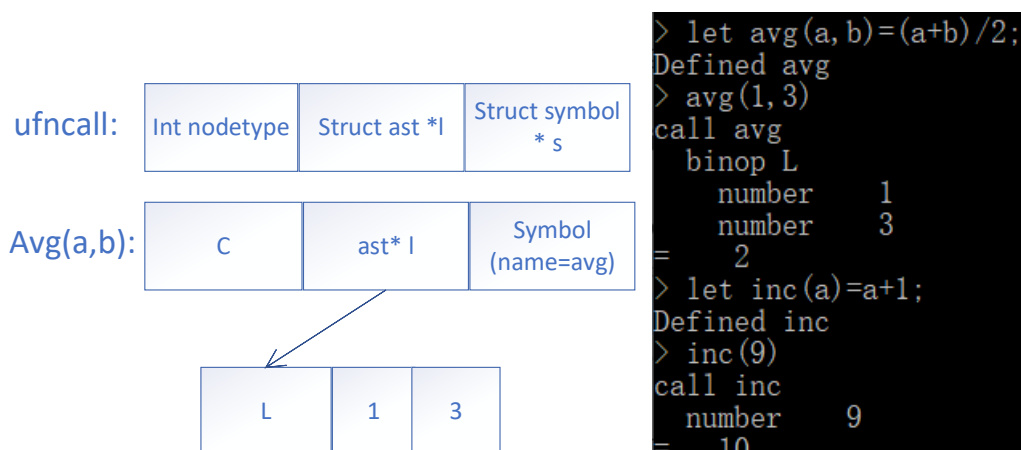


图 2-9 用户定义函数类型与 debug 输出

在图 2-9 中，分别展示了双目函数 `avg(a,b)` 与，单目函数 `inc(a)`，可以看到内部子树稍有区别。

2.6.7 函数定义 (dodef)

在语法树求值过程中设计到用户定义函数部分，如代码 2-19 所示。相对比较容易理解，如果重名冲突，则释放旧的定义，同时增加新的符号表定义。

代码 2-19 `dodef(struct symbol*,struct symlist*,struct ast*)`

```
1. void dodef(struct symbol *name, struct symlist *syms,
   struct ast *func)
2. {
3.   if(name->syms) symlistfree(name->syms);
4.   if(name->func) treefree(name->func);
5.   name->syms = syms;
6.   name->func = func;
```

7. }

2.6.8 变量引用 (ref)

在定义完变量后，当需要引用时，直接使用的名即可，根据所示节点构建，可以得到如下的节点类型示意。当引用 a 时，其 debug 模式输出如图 2-10 所示。与之有着很好的对应。

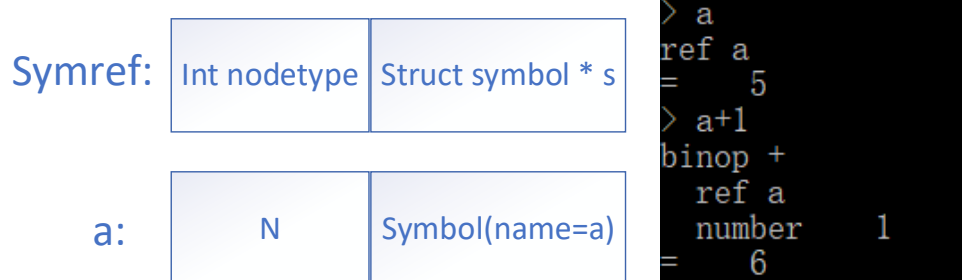


图 2-10 变量引用节点类型与 debug 输出 (a)

2.6.9 变量赋值 (asgn)

当给一个变量赋值时，其节点类型固定为“=”，包含子树记录计算值，此外符号节点 symbol 记录变量名称，以“a=2+3 为例”，其 debug 输出如图 2-11 所示。

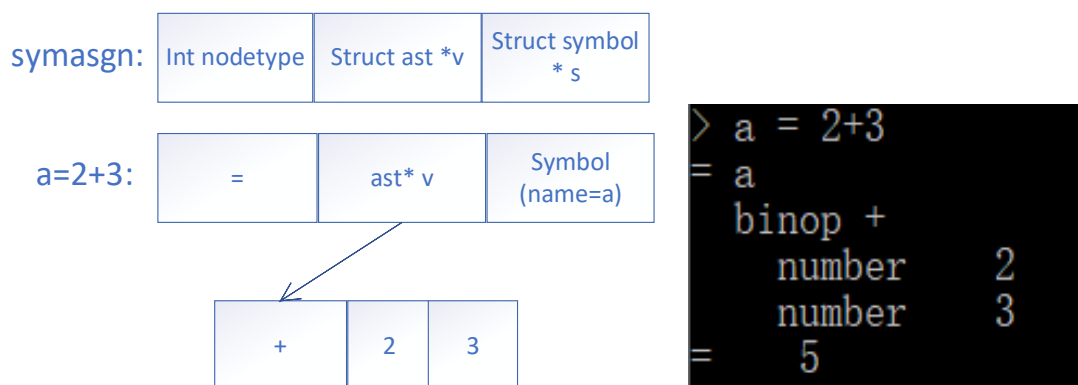


图 2-11 变量声明节点类型与 debug 输出 (a=2+3)

2.6.10 条件分支语句 (flow)

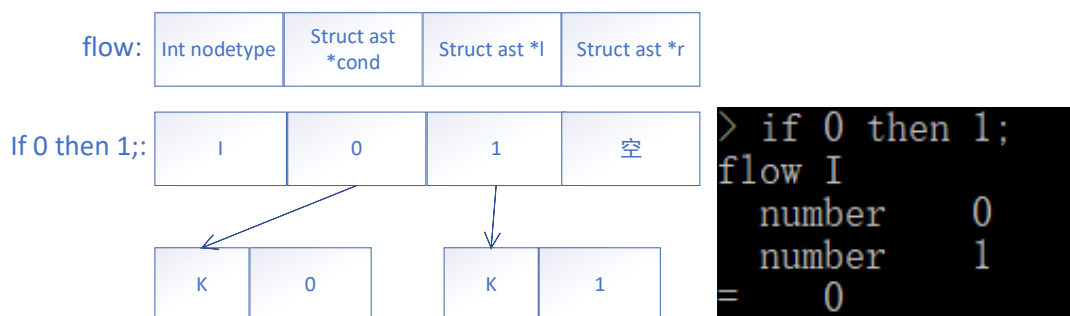


图 2-12 条件分支语句与 debug 输出 (if 0 then 1;)

条件分支语句包含 if 语句与 while 语句，根据 2.5.7 中分析，其节点类型分别为“I”与“W”。此外包含有三个节点，以“if 0 then 1;”为例，其 debug 模式输出如图 2-12 所示。

```
> if 0 then 1 ; else 2;      > if 1 < 2 then a-1;
flow I                      flow I
number    0                binop 2
number    1                number  1
number    2                number  2
=    2                    binop -
>                          ref a
                           number  1
                           =    4
```

图 2-13 稍微复杂的条件分支语句输出

当进行更加复杂的条件分支语句时，观察其 debug 输出，可以发现，仍然是抽象语法树的类似输出，如图 2-13 所示。

2.6.11 语句链表 (explist)

在 fb3-2.y 中第 63-65 行定义了语句链表 `explist` 的执行动作，中也已分析到，其本质上类似高级语言中的逗号表达式，在 `eval` 计算时以最右边表达式值为基础。节点类型使用基础的 `ast` 类型，`nodetype` 固定为“L”，包含左右子树。

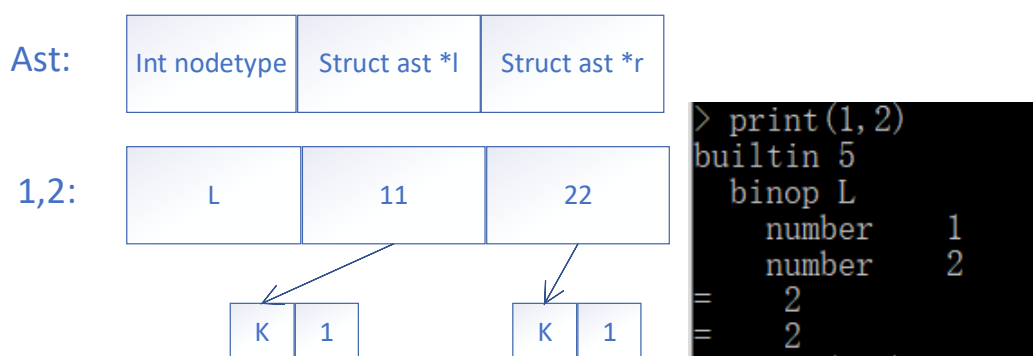


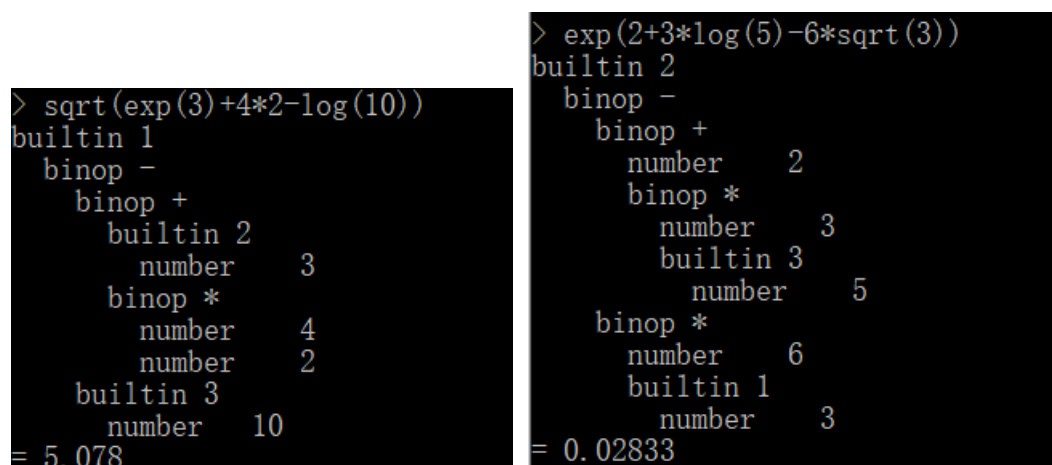
图 2-14 语句链表与 debug 输出 (1,2)

在测试 `print(a)` 是一个单目内置函数，此处测试 `print(1,2)` 输出 2，同时 debug 下看到输出 `binop L`，证实了在调用了 `explist` 时，输出最右值。如图 2-14 所示。

2.6.12 Debug 综合分析

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。通过肉眼 `eval()` 的方式，计算其值。进而可以帮助纠正错误。图 2-15 展示了两种较为复杂的函数调用，并展示了其 d

ebug 打印的抽象语法树。



```

> sqrt(exp(3)+4*2-log(10))
builtin 1
  binop -
    binop +
      builtin 2
        number 3
      binop *
        number 4
        number 2
      builtin 3
        number 10
= 5.078

> exp(2+3*log(5)-6*sqrt(3))
builtin 2
  binop -
    binop +
      number 2
      binop *
        number 3
        builtin 3
          number 5
    binop *
      number 6
      builtin 1
        number 3
= 0.02833

```

图 2-15 较为复杂情况下 debug 打印抽象语法树

2.7 增加内置函数 pow(a,b)

在的分析基础上，增加相近的内容则会相对容易不少。首先，内置函数的语法树构建及其工作时序在中有详细的介绍。

所以首先可以明确如下几个步骤：

1. fb3-2.1 中增加"pow"的模式匹配。如图 2-16 所示
2. fb3-2.h 中枚举型 bifs 中增加 Pow。如图 2-17 所示

3. fb3-2funcs.c 中 callbuiltin()函数中增加对于 B_pow 的动作，如图 2-18 所示。因为 pow(.)为双元函数，与之前的 log 等稍有区别，需要分析语法树，如图 2-19 所示。

```

/* built in functions */
"sqrt" { yylval.fn = B_sqrt; return FUNC; }
"exp" { yylval.fn = B_exp; return FUNC; }
"log" { yylval.fn = B_log; return FUNC; }
"pow" { yylval.fn = B_pow; return FUNC; }
"print" { yylval.fn = B_print; return FUNC; }

```

图 2-16 fb3-2.1 修改内容

```

enum bifs {
  B_sqrt = 1,
  B_exp,
  B_log,
  B_pow,
  B_print
};

```

图 2-17 fb3-2.h 修改内容

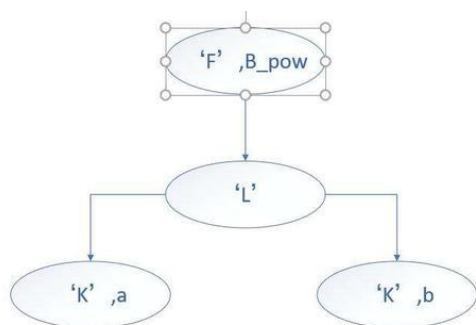
```

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_pow:
        return pow(eval(f->l->l), v);
    case B_print:
        printf("= %4.4g\n", v);
    }
}

```

图 2-18 fb3-2funcs.c 文件修改内容



```

> pow(2,3)
builtin 4
  binop L
    number 2
    number 3
= 27

```

图 2-19 pow(a,b)语法树简图与 debug 效果图

2.8 实验总结

实验 2 收获颇多。

在实验 2 中，通过仔细阅读源代码，发现了包括散列符号表、debug 模式等许多示例代码支持的方式。同时通过深入的学习代码中关于抽象语法树的构建和遍历，对于整个桌面计算器的工作全流程有了更加深入的了解。

3、综合实验 3

3.1 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：

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

3.2 实验内容

- 1. 阅读 flex Python 第三章 P79 习题 1，重点学习抽象语法树；
- 2. 修改 fb3-2 相关代码，并保存为 fb3-3，使得支持特定函数；
- 3. 撰写实验报告，结合实验结果，给出抽象语法树的构建过程；
- 4. 提交报告和实验代码。

3.3 实验步骤

本次实验重点考察对于语法制导翻译的理解与运用，并修改相应的语法规则使得支持特定语句的翻译。

3.3.1 语法修改

在实验的要求中，自定义函数部分 let 需要能够支持“{……}”的函数段落定义结构，同时在选择判断分支语句部分，同样需要可以支持“{……}”的括弧定义。

观察 fb3-3.y 文件中原有的语句定义，考虑对涉及的关键语句 list 部分进行如下的修改。修改后的代码如代码 3-1 所示。

代码 3-1 fb3-3.y

```
1. %{  
2. # include <stdio.h>  
3. # include <stdlib.h>  
4. # include "fb3-3.h"  
5. %}  
6.  
7. %union {
```

```

8.  struct ast *a;
9.  double d;
10. struct symbol *s;  /* which symbol */
11. struct symlist *sl;
12. int fn;            /* which function */
13. }
14.
15. /* declare tokens */
16. %token <d> NUMBER
17. %token <s> NAME
18. %token <fn> FUNC
19. %token EOL
20.
21. %token IF THEN ELSE WHILE DO LET
22.
23.
24. %nonassoc <fn> CMP
25. %right '='
26. %left '+' '-'
27. %left '*' '/'
28. %nonassoc '|' UMINUS
29. %nonassoc LOWER
30. %nonassoc ELSE
31. %left '('
32.
33. %type <a> exp stmt list explist
34. %type <sl> symlist
35.
36. %start calclist
37.
38. %%
39.
40. stmt: IF exp THEN list %prec LOWER      { $$ = newflow('I',
    $2, $4, NULL); }
41.    | IF exp THEN list ELSE list { $$ = newflow('I', $2,
    $4, $6); }
42.    | WHILE exp DO '{' list '}'      { $$ = newflow('W', $2,
    $5, NULL); }
43.    | exp ';' %prec LOWER
44. ;
45.
46. list: '{' list '}' { $$ = $2; }
47.    | '{' list stmt '}' { $$ = newast('L', $2, $3); }
48.    | stmt { $$ = $1; }

```

```

49. | exp %prec LOWER{ $$ = $1; }
50. ;
51.
52. exp: exp CMP exp      { $$ = newcmp($2, $1, $3); }
53. | exp '+' exp        { $$ = newast('+', $1, $3); }
54. | exp '-' exp        { $$ = newast('-', $1, $3); }
55. | exp '*' exp        { $$ = newast('*', $1, $3); }
56. | exp '/' exp        { $$ = newast('/', $1, $3); }
57. | '|' exp            { $$ = newast('|', $2, NULL); }
58. | '(' exp ')'        { $$ = $2; }
59. | '-' exp %prec UMINUS { $$ = newast('M', $2, NULL); }
60. | NUMBER              { $$ = newnum($1); }
61. | FUNC '(' explist ')' { $$ = newfunc($1, $3); }
62. | NAME %prec LOWER    { $$ = newref($1); }
63. | NAME '=' exp        { $$ = newasgn($1, $3); }
64. | NAME '(' explist ')' { $$ = newcall($1, $3); }
65. ;
66.
67. explist: exp
68. | exp ',' explist { $$ = newast('L', $1, $3); }
69. ;
70. symlist: NAME      { $$ = newsymlist($1, NULL); }
71. | NAME ',' symlist { $$ = newsymlist($1, $3); }
72. ;
73.
74. calclist: /* nothing */
75. | calclist stmt EOL {
76.     if(debug) dumpast($2, 0);
77.     printf("= %4.4g\n> ", eval($2));
78.     treefree($2);
79. }
80. | calclist LET NAME '(' symlist ')' list EOL {
81.     dodef($3, $5, $7);
82.     printf("Defined %s\n> ", $3->name); }
83.
84. | calclist error EOL { yyerror; printf("> "); }
85. ;
86. %%

```

相较于 fb3-2.y, 有如下改动:

第 80 行, 去掉原 “=” ;

第 41-42 行, 增加 “{”、“}” ;

第 46-50 行, 增加 “{”、“}”, 以及 list -> exp;

从而在文法设计上实现了对于题中句式的支持, 但是在编译运行时, 提示

如下移进规约冲突。

Fb3-3.y: conflicts: 3 shift/reduce .

3.3.2 解决移进规约冲突

针对移进规约冲突，可以在 **bison** 编译时通过 **-v** 命令生成状态机描述文件，从而查看定位具体的冲突发生位置。

State 5 conflicts: 1 shift/reduce

State 49 conflicts: 1 shift/reduce

State 50 conflicts: 1 shift/reduce

打开.output 文件，具体查找对应的状态 5、49、50，分析其冲突发生原因。

原因一：代码 3-1 中第 62、63 行，当移进 NAME 后，遇到“(”时程序不能判断是否进行规约或者移进，所以这里需要修改优先级，将 NAME 改为没有结合性，“(”改为左结合性。在定义部分增加%nonassoc LOWER 与%left ‘(’即可解决该冲突。

原因二：代码 3-1 中第 40、41 行，当移进 list 后，遇到 ELSE 时程序不能判断是否进行规约或者移进，所以在此处修改优先级，提高“stmt:IF exp THEN N list”的优先级。

原因三：代码 3-1 中第 43、49 行，当遇到 **exp** 时，程序无法区分是应该继续移进 “;”，还是将 **exp** 规约为 **list**。解决方案为在第 43 行后增加 **%prec LOWER** 降低其规约的优先级。

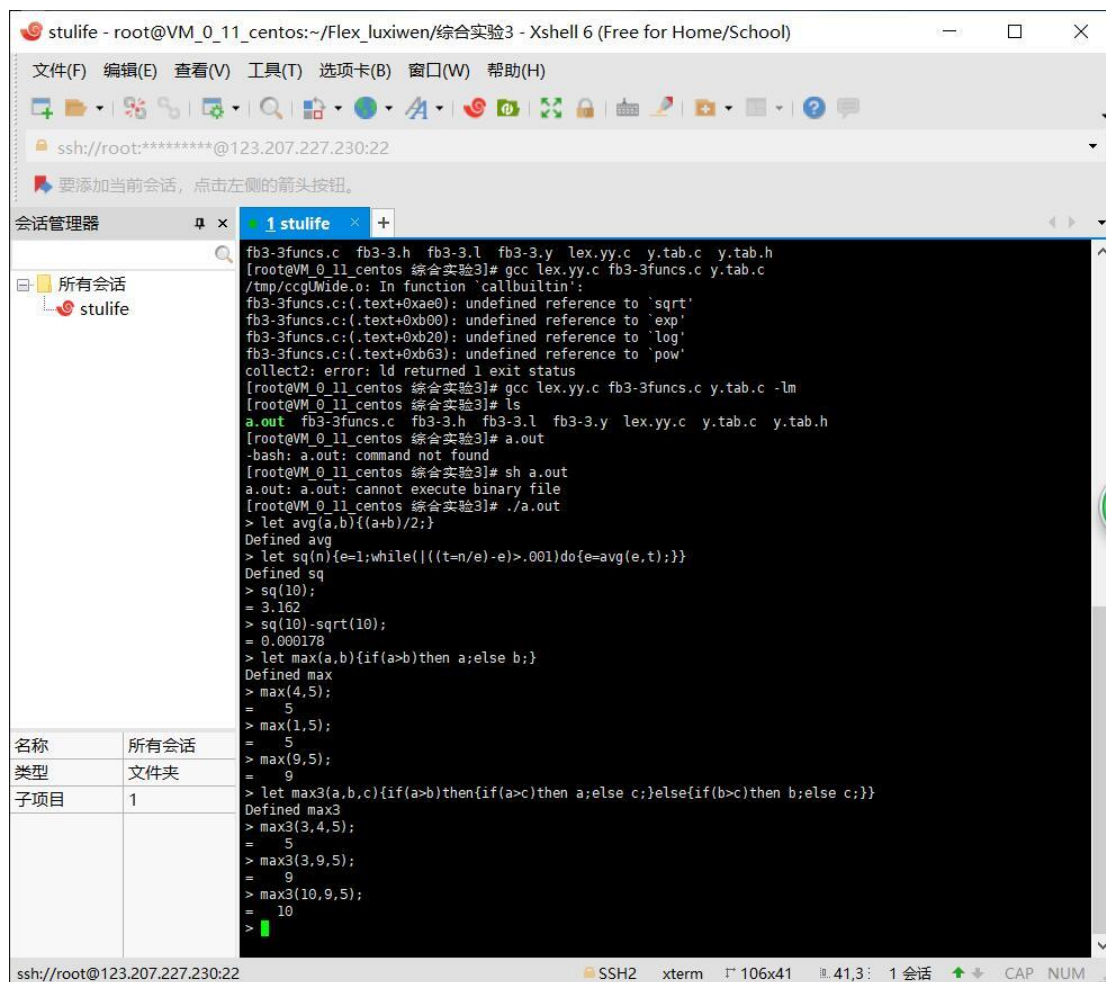
3.3.3 运行效果

3.3.3.1 Windows

[illegible]

代码 3-2 实验三 Windows 下运行结果

3.3.3.2CentOS



```
stulife - root@VM_0_11_centos:~/Flex_luxuwen/综合实验3 - Xshell 6 (Free for Home/School)
文件(F) 编辑(E) 查看(V) 工具(T) 选项卡(B) 窗口(W) 帮助(H)
ssh://root:*****@123.207.227.230:22
要添加当前会话，点击左侧的箭头按钮。
会话管理器
所有会话
stulife
fb3-3funcs.c fb3-3.h fb3-3.l fb3-3.y lex.yy.c y.tab.c y.tab.h
[root@VM_0_11_centos 综合实验3]# gcc lex.yy.c fb3-3funcs.c y.tab.c
/tmp/ccgUWide.o: In function 'callbuiltin':
fb3-3funcs.c:(.text+0xae0): undefined reference to 'sqrt'
fb3-3funcs.c:(.text+0xb00): undefined reference to 'exp'
fb3-3funcs.c:(.text+0xb20): undefined reference to 'log'
fb3-3funcs.c:(.text+0xb63): undefined reference to 'pow'
collect2: error: ld returned 1 exit status
[root@VM_0_11_centos 综合实验3]# gcc lex.yy.c fb3-3funcs.c y.tab.c -lm
[root@VM_0_11_centos 综合实验3]# ls
a.out fb3-3funcs.c fb3-3.h fb3-3.l fb3-3.y lex.yy.c y.tab.c y.tab.h
[root@VM_0_11_centos 综合实验3]# a.out
-bash: a.out: command not found
[root@VM_0_11_centos 综合实验3]# sh a.out
a.out: a.out: cannot execute binary file
[root@VM_0_11_centos 综合实验3]# ./a.out
> let avg(a,b){(a+b)/2;}
Defined avg
> let sq(n){e=1;while((((t=n/e)-e)>.001)do{e=avg(e,t);}}
Defined sq
> sq(10);
= 3.162
> sq(10)-sqrt(10);
= 0.000178
> let max(a,b){if(a>b)then a;else b;}
Defined max
> max(4,5);
= 5
> max(1,5);
= 5
> max(9,5);
= 9
> let max3(a,b,c){if(a>b)then{if(a>c)then a;else c;}else{if(b>c)then b;else c;}}
Defined max3
> max3(3,4,5);
= 5
> max3(3,9,5);
= 9
> max3(10,9,5);
= 10
>
```

名称	所有会话
类型	文件夹
子项目	1

代码 3-3 实验三 CentOS 下运行结果

3.4 实验总结

实验三总体上比较简单，可以通过对于所需要文法的改进支持。同时，在面对移进规约冲突时，通过观察状态机描述文件，独立解决了该问题，进一步增强了用 Flex 与 Bison 设计文法并工作的能力，同时很好的回顾了 Bison 实验 2 的内容。

4、综合实验 4

4.1 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桌面计算器，包括如下功能：

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

4.2 实验内容

- 1. 搜集资料，能够将 flex 与 bison 代码生成动态链接库；
- 2. 选择合适的开发语言调用动态链接库实现一个具有完整功能的桌面计算器；
- 3. 撰写实验报告，结合实验结果，分析整个步骤；
- 4. 提交报告和实验代码。

4.3 实验步骤

本次实验主要是构造一个前端框架能够调用 flex 与 bison 的编译器后端，考虑到往届的学长学姐尚未能够在安卓端有所突破，故选择安卓 APP 开发作为本次实验的展示面。

实验环境：Android studio 3.6.

编程语言：Java.

4.3.1 资料搜集

凡事预则立不预则废。动态链接库的调用与安卓端 APP 开发于我皆是全新的领域，故进行了详细的资料搜集过程，以及流程步骤整理。

首先，在新版的 Android studio 中已经支持通过 NDK 和 CMake 进行 C 和 C++代码的添加了。最权威的资料莫过于安卓开发者官网：<https://developer.android.com/studio/projects/add-native-code?hl=zh-cn>。

此外，在学习过程中，大量的技术实践博客也给予了我启示与帮助。其中直接给予我启示的是这一篇：<https://blog.csdn.net/andylauren/article/details/10529>

[3836](#)。

4.3.2 NDK 与 CMake 安装

在安卓开发者官网上，给出了 NDK 与 CMake 安装的详细步骤，其示意图如图 4-1 所示。

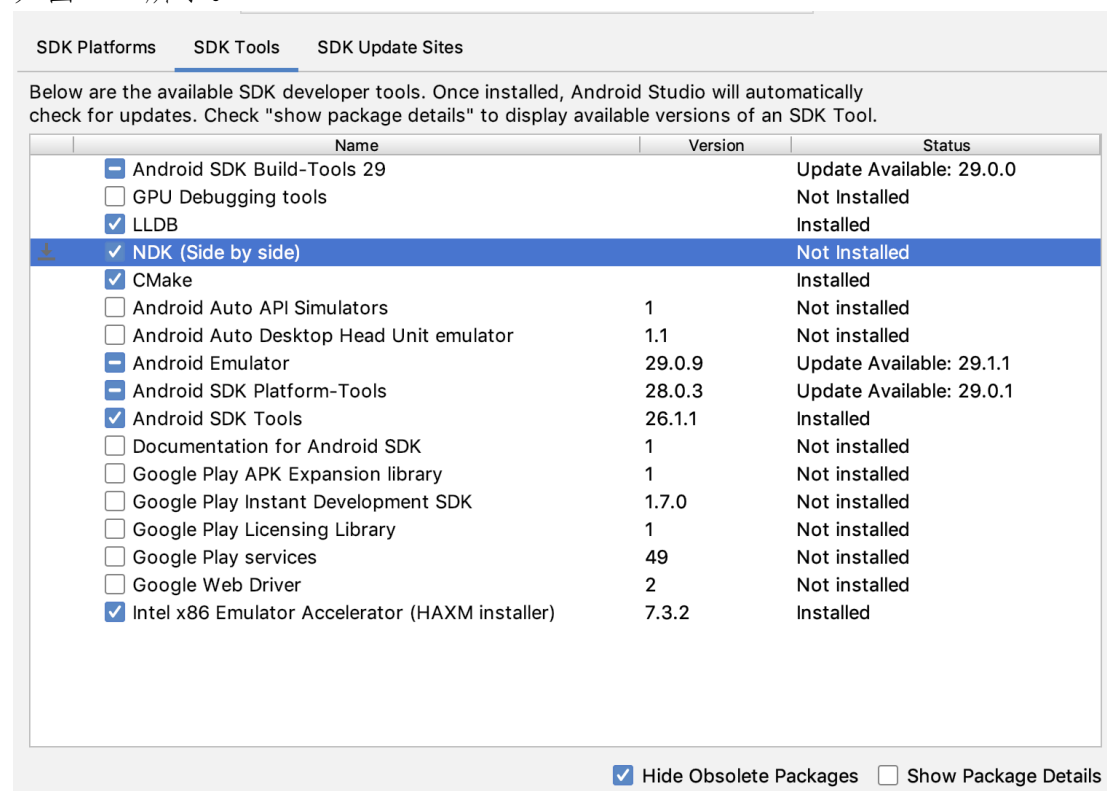


图 4-1 NDK 与 CMake 安装

4.3.3 新建 Native C++项目

传统的完整的在 java 中导入 C/C++方法为：

1. 编写 Java 调用 C 方法的类；
2. 使用 JNI 编译该类，生成符合 JNI 的头文件.h；
3. 根据.h 文件，编写实现的.c 文件，从而能够调用其他 C 方法；
4. 编译.c 文件与其他 C 文件，生成动态链接.so 库；
5. 在 Java 主程序段中调用 System.load()方法，导入.so 库；
6. 调用第一步中的 Java 类，从而实现对于 C 方法的调用。

而在 Android Studio 中，简化了上述的第 2 步，在编写完调用 C 方法的 Java 类后，可以自动生成对应的 C 文件，在其中完成修改实现对于自己的 C 函数调用即可。同时 Android Studio 提供了 CMake 编译生成.so 库，详细步骤在官方文档中有具体介绍：<https://developer.android.com/studio/projects/add-native-code?hl=zh-cn>。

为了更加便捷的使用 **Android Studio**，选择直接新建 **Native C++**项目，从而可以略去配置 **Makefile** 的许多步骤。新建项目方式如图 4-2 所示。

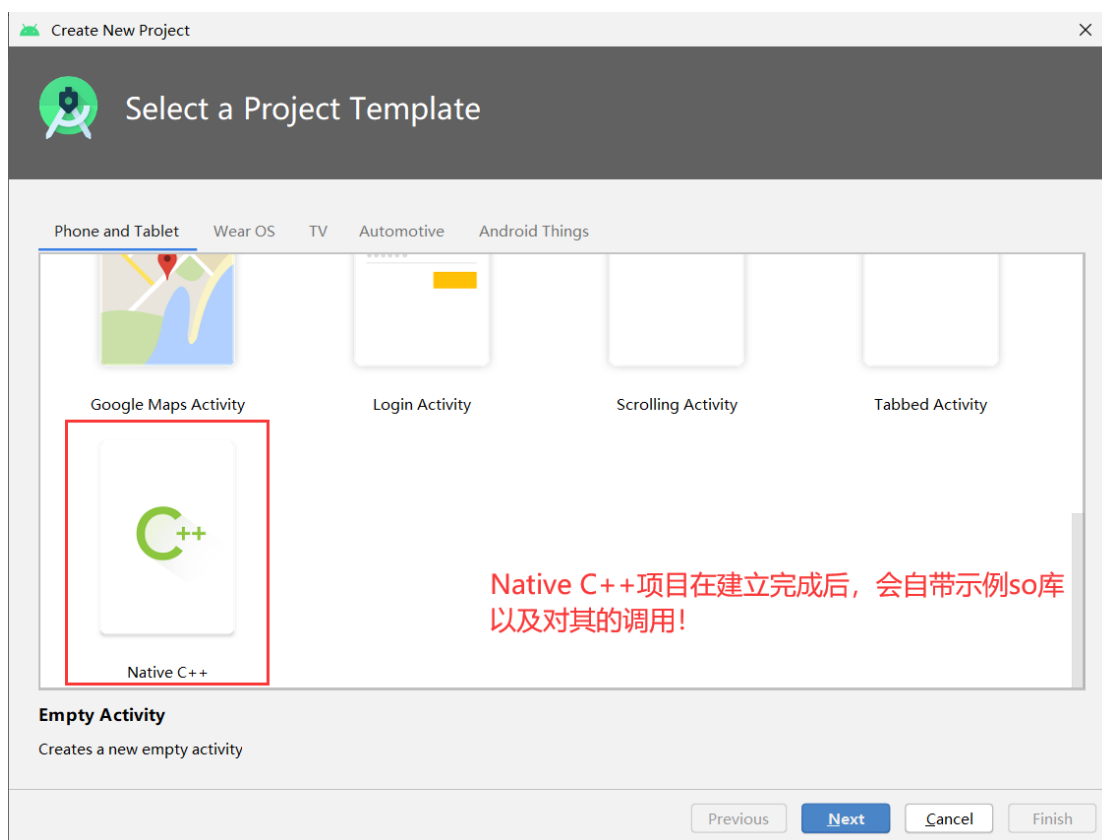


图 4-2 新建 Native C++项目

根据提示新建完成后，即可进入主界面。此时切换到 **Project** 视图，在`app ->src->main->cpp`路径下，可以看到示例的 **native-lib.cpp** 文件，如图 4-3 所示²。

打开 **native-lib.cpp**，可以看到其中所提供的一段示例的根据 **JNI** 规范编写的 **C** 文件代码，其他的 **C** 函数，都将在其中被调用。而其所对应的 **JAVA** 函数，则将被其他 **JAVA** 类调用。最终实现了通过 **Java** 调用 **C** 函数。

代码 4-1 native-lib.cpp（已修改）

```

1. extern "C" JNIEXPORT jstring JNICALL
   Java_com_xiwen_flexcalculator_Caculate_calc(JNIEnv *env, jclass
   thiz,
       jstring expr) {
2.     const char* s = env->GetStringUTFChars(expr, 0);
3.     jdouble ans = 0;
4.     return env->NewStringUTF(calc(s));
5. }

```

² 新建项目无 calc-advan 目录，该目录为本人实验过程中修改。

代码 4-1 中展示了本次实验中，我所编写的 `java` 类自动生成的 `C` 文件，实现了调用 `flex` 与 `bison` 的后端代码。

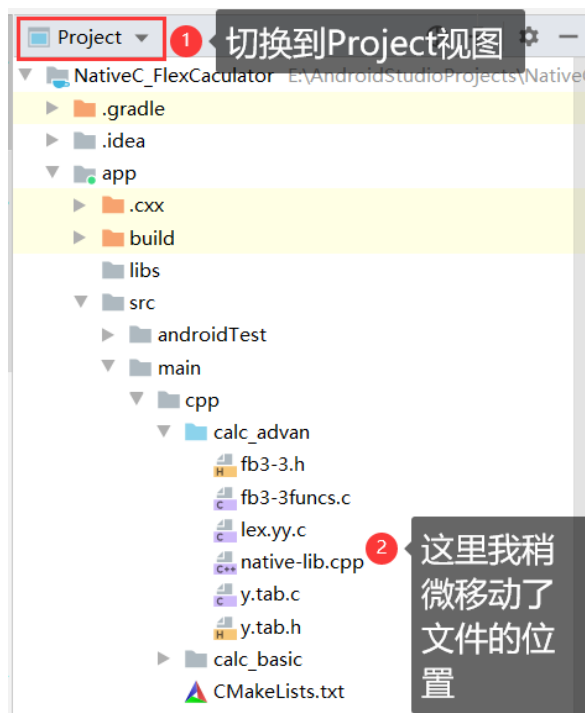


图 4-3 文件组织示意图

整个项目的文件目录结构如图 4-4 所示（Android 视图）。

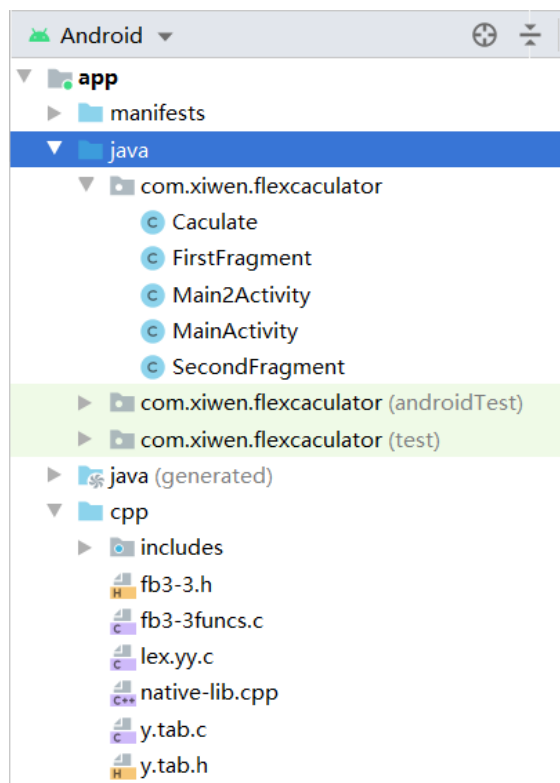


图 4-4 整体项目目录结构

与代码 4-1 相对应的，执行调用任务的 JAVA 类如代码 4-2 所示。

代码 4-2 Caculate 类

```
1. package com.xiwen.flexcalculator;
2.
3. public class Caculate {
4.     static {
5.         System.loadLibrary("native-lib-flex");
6.     }
7.     public static native String calc(String expr);
8. }
```

4.3.4 搭建 UI 界面

考虑到初次学习 Android 开发，在 UI 界面上选择原型开发模式，通过 github 搜索，最终选择更新日期较近的一款小项目作为项目原型：<https://github.com/bestxiaxiaoming-hm/AndroidStudio---java->。

在修改完成之后，实验计算器主界面如图 4-5 所示。相比较于 github 原项目的代码，在基础版修改其小数点“.”为高级版切换按钮，高级版则沿用其网格布局，同时简化其他所有按钮。加大输入输出文本框的占位。更加便于自定义函数以及调用。

程序整体上使用网格布局（9 行 4 列），保证每个控件的大小均匀。同时最下方的“=”占位两个字符宽度，以使其更加突出。

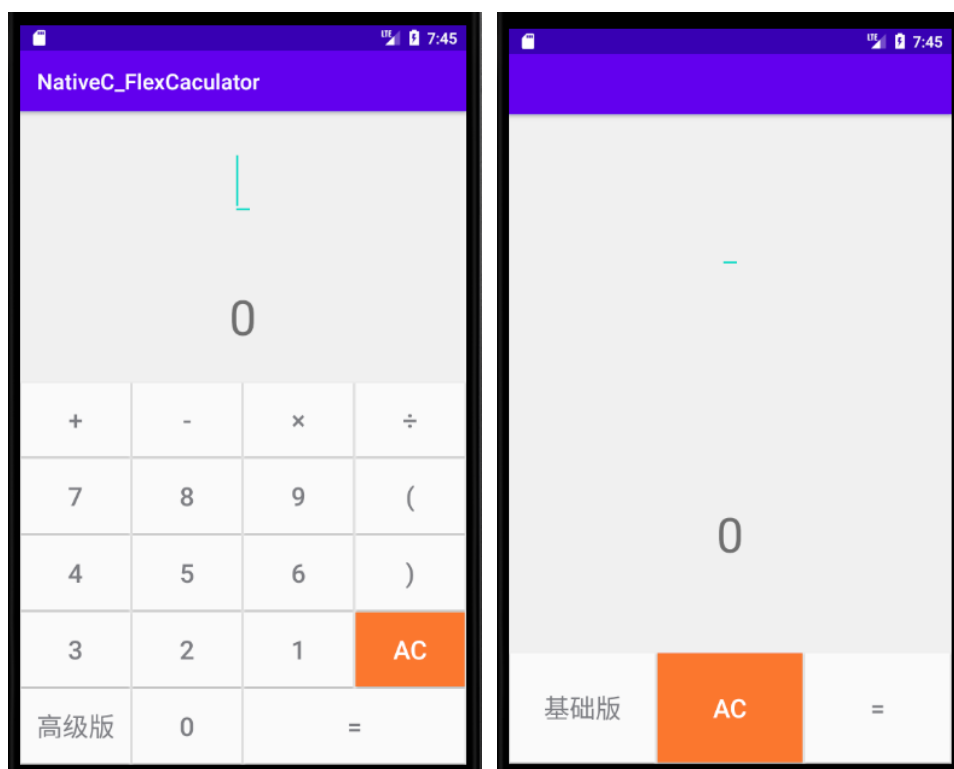


图 4-5 安卓计算器主界面

4.3.5 导入 C 文件修改 CMakefile

如图 4-3 所示，为了使用实验 3 中的源代码，在调试完成后，将必要的.c 文件以及.h 文件，组织放置在 calc_advan 目录中，同时修改 CMakelist.txt 内容如代码 4-3 所示。第 6 行为所需编译的 c 文件。

代码 4-3 CMakelist.txt

```
1. cmake_minimum_required(VERSION 3.4.1)
2. add_library(native-lib-flex
3.
4.         SHARED
5.
6.         calc_advan/native-lib.cpp calc_advan/fb3-3funcs.c
   calc_advan/lex.yy.c calc_advan/y.tab.c
7.     )
8. find_library(
9.         log-lib
10.
11.         log )
12.
13. target_link_libraries(
14.         native-lib-flex
15.
16.         ${log-lib} )
```

其他内容，诸如 build.gradle 中 CMake 配置不做修改，沿用默认配置即可。

4.3.6 Flex 与 Bison 源代码调整

首先，为了编译成.so 文件供 java 调用函数，需要去掉源文件中 main() 函数，改增 char* calc(char*) 函数，如代码 4-4 所示。

代码 4-4 新增 calc(char*) 函数(fb3-3.1 中)

```
1. char* calc(const char* expr)
2. {
3.     size_t len = strlen(expr);
4.     result = (char*)malloc(sizeof(char)*(len+2));
5.     yy_switch_to_buffer(yy_scan_string(expr));
6.     yyparse();
7.     return result;
8. }
```

此外，为了能够直接通过字符串传递输入输出，修改.y 文件中的部分语法规则，使得支持将结果值传入全局变量 result 中（result 定义在头文件 fb3-3.h 中），所修改部分代码如代码 4-5 所示。

代码 4-5 修改 fb3-3.y 文件部分代码

```

1. calclist: /* nothing */
2. | calclist stmt {
3.     //if(debug) dumpast($2, 0);
4.     sprintf(result, "%4.4g", eval($2));
5.     treefree($2);
6. }
7. | calclist LET NAME '(' symlist ')' list{
8.     dodef($3, $5, $7);
9.     sprintf(result, "Defined %s .", $3->name); }

```

4.3.7 演示效果

整个实验的完整演示视频已发布于：https://v.youku.com/v_show/id_XNDY3NTU5MzI0NA==.html。

基本数值四则运算如图 4-6 所示。

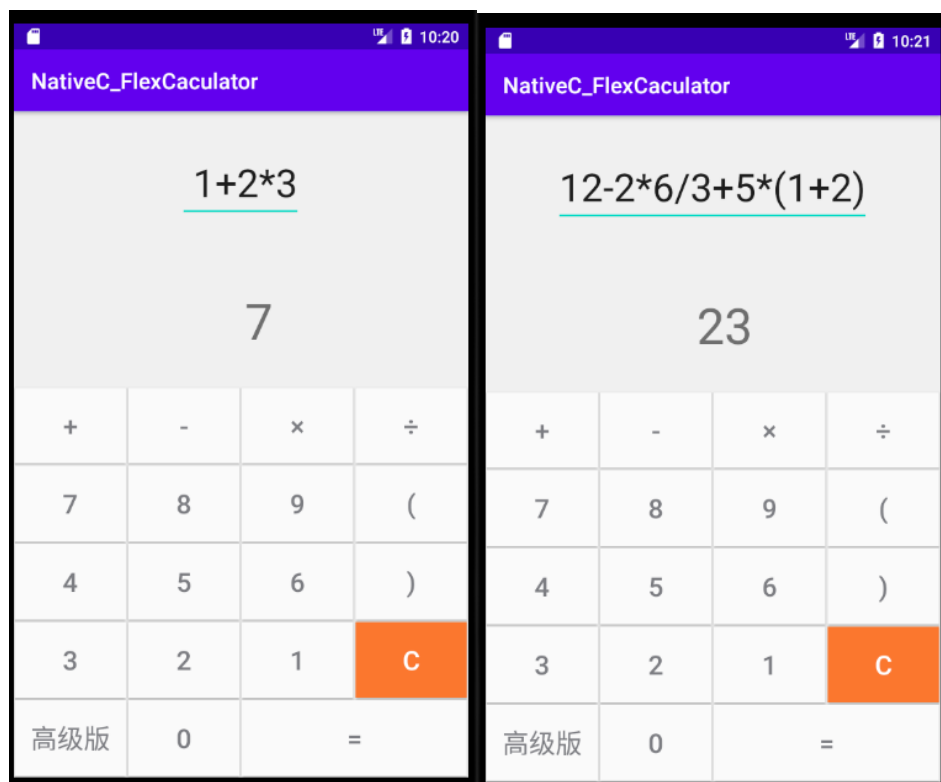


图 4-6 简单四则运算

自定义函数并调用如图 4-7、图 4-8 所示。

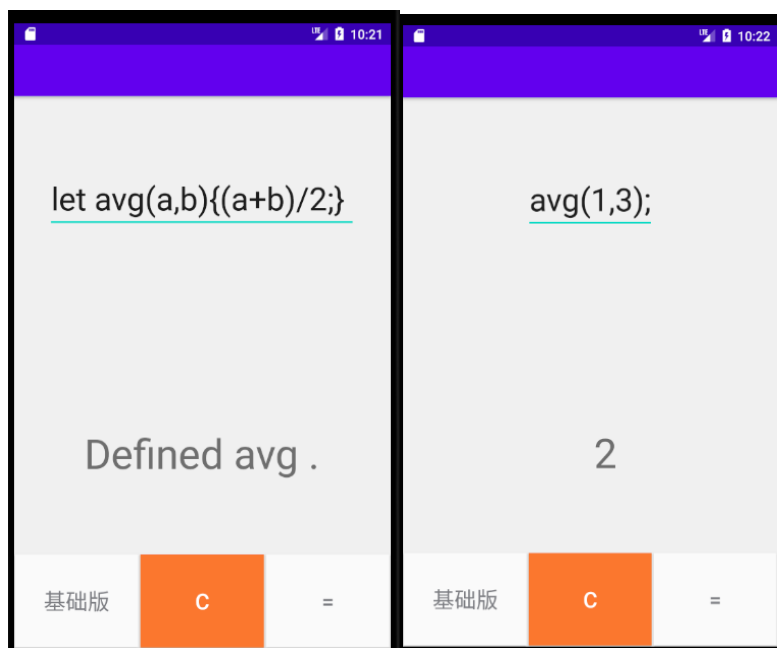


图 4-7 自定义求平均值

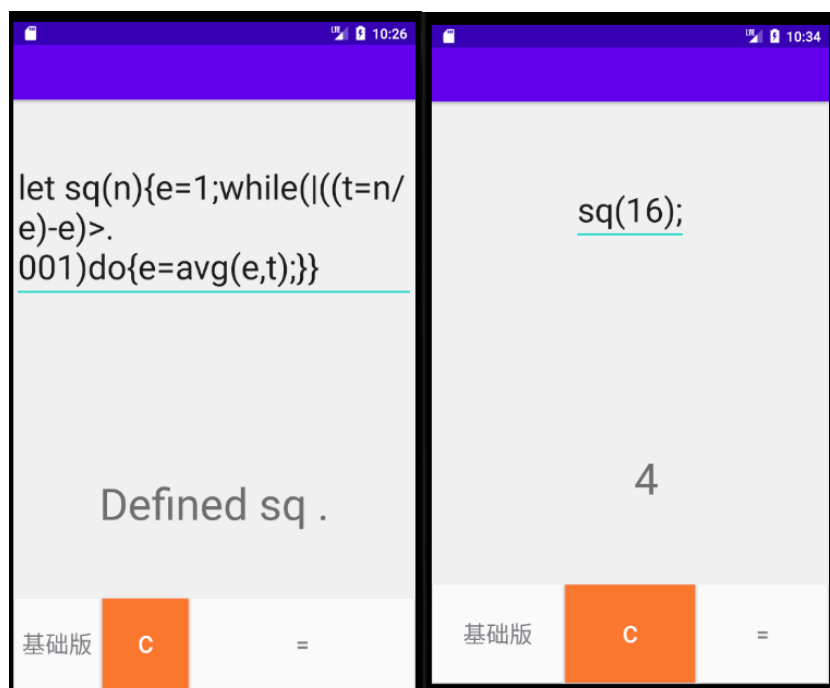


图 4-8 自定义求平方根

4.4 实验总结

实验四独立完成了将 Flex 与 Bison 部署在安卓端，期间搜集了大量资料，对于安卓调用 C、C++ 文件有了直观的体验和感受。

也正是在实际编写软件过程中，发现了许多细节上的不到位，比如对于输入不符合语法的错误处理等，还需要进一步加强。