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](#_Toc40557724)

[1.1 实验目的 1](#_Toc40557725)

[1.2 实验内容 1](#_Toc40557726)

[1.3 实验步骤 1](#_Toc40557727)

[1.3.1 Windows 1](#_Toc40557728)

[1.3.2 CentOS 2](#_Toc40557729)

[1.3.3 讨论交流 2](#_Toc40557730)

[1.4 移进规约分析 3](#_Toc40557731)

[1.4.1 移进规约分析 3](#_Toc40557732)

[1.4.2 语法分析树 4](#_Toc40557733)

[1.5 实验总结 5](#_Toc40557734)

[1.5.1 遇到的难题 5](#_Toc40557735)

[1.5.2 实验收获 5](#_Toc40557736)

[2、 综合实验2 6](#_Toc40557737)

[2.1 实验目的 6](#_Toc40557738)

[2.2 实验内容 6](#_Toc40557739)

[2.3 实验步骤 6](#_Toc40557740)

[2.3.1 CentOS 6](#_Toc40557741)

[2.3.2 Windows 7](#_Toc40557742)

[2.4 源代码分析 7](#_Toc40557743)

[2.4.1 fb3-2.l 7](#_Toc40557744)

[2.4.2 fb3-2.y 9](#_Toc40557745)

[2.4.3 fb3-2.h 11](#_Toc40557746)

[2.4.4 fb3-2funcs.c 14](#_Toc40557747)

[2.5 语法树构建 17](#_Toc40557748)

[2.5.1 常量（num） 17](#_Toc40557749)

[2.5.2 比较运算（cmp） 18](#_Toc40557750)

[2.5.3 内置函数（func） 18](#_Toc40557751)

[2.5.4 函数调用（call） 19](#_Toc40557752)

[2.5.5 变量引用（ref） 19](#_Toc40557753)

[2.5.6 变量声明（asgn） 19](#_Toc40557754)

[2.5.7 条件分支语句（flow） 20](#_Toc40557755)

[2.5.8 表达式链（symlist） 20](#_Toc40557756)

[2.5.9 函数定义（dodef） 21](#_Toc40557757)

[2.6 语法树求值（Debug输出分析） 21](#_Toc40557758)

[2.6.1 语法树求值 21](#_Toc40557759)

[2.6.2 Debug模式分析 26](#_Toc40557760)

[2.6.3 常量（num） 28](#_Toc40557761)

[2.6.4 比较运算（cmp）基本运算（+-\*/） 28](#_Toc40557762)

[2.6.5 内置函数（func） 29](#_Toc40557763)

[2.6.6 函数调用（call） 30](#_Toc40557764)

[2.6.7 函数定义（dodef） 30](#_Toc40557765)

[2.6.8 变量引用（ref） 31](#_Toc40557766)

[2.6.9 变量赋值（asgn） 31](#_Toc40557767)

[2.6.10 条件分支语句（flow） 31](#_Toc40557768)

[2.6.11 语句链表（explist） 32](#_Toc40557769)

[2.6.12 Debug综合分析 32](#_Toc40557770)

[2.7 增加内置函数pow(a,b) 33](#_Toc40557771)

[2.8 实验总结 34](#_Toc40557772)

[3、 综合实验3 35](#_Toc40557773)

[3.1 实验目的 35](#_Toc40557774)

[3.2 实验内容 35](#_Toc40557775)

[3.3 实验步骤 35](#_Toc40557776)

[3.3.1 语法修改 35](#_Toc40557777)

[3.3.2 解决移进规约冲突 38](#_Toc40557778)

[3.3.3 运行效果 38](#_Toc40557779)

[3.4 实验总结 39](#_Toc40557780)

[4、 综合实验4 40](#_Toc40557781)

[4.1 实验目的 40](#_Toc40557782)

[4.2 实验内容 40](#_Toc40557783)

[4.3 实验步骤 40](#_Toc40557784)

[4.3.1 资料搜集 40](#_Toc40557785)

[4.3.2 NDK与CMake安装 41](#_Toc40557786)

[4.3.3 新建Native C++项目 41](#_Toc40557787)

[4.3.4 搭建UI界面 44](#_Toc40557788)

[4.3.5 导入C文件修改CMakefile 45](#_Toc40557789)

[4.3.6 Flex与Bison 源代码调整 45](#_Toc40557790)

[4.3.7 演示效果 46](#_Toc40557791)

[4.4 实验总结 47](#_Toc40557792)

# 综合实验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。

## 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。

## 实验步骤

### 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。

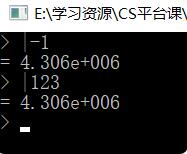


图 1‑1 Win下数值异常情况

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",&yylval.d)。如图 1‑2所示。

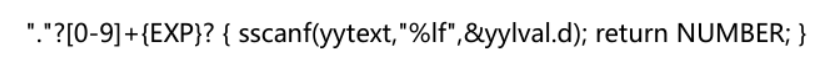


图 1‑2 模式匹配动作修改

### 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。

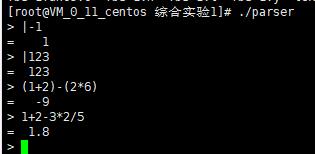


图 1‑3 CentOS下结果示意图

### 讨论交流

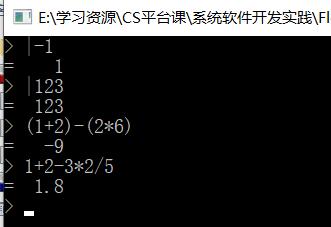


图 1‑4 Windows下运行示意图

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.l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。

## 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。

### 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term−>NUMBER |
| #(term | +2)-(2\*6)# | 规约factor−>term |
| #(factor | +2)-(2\*6)# | 规约exp−>factor |
| #(exp | +2)-(2\*6)# | 移进 |
| #(exp+ | 2)-(2\*6)# | 移进 |
| #(exp+2 | )-(2\*6)# | 规约term−>NUMBER |
| #(exp+term | )-(2\*6)# | 规约factor−>term |
| #(exp+factor | )-(2\*6)# | 规约exp−>exp+factor |
| #(exp | )-(2\*6)# | 移进 |
| #(exp) | -(2\*6)# | 规约term−>(exp) |
| #term | -(2\*6)# | 规约factor−>term |
| #factor | -(2\*6)# | 规约exp−>factor |
| #exp | -(2\*6)# | 移进 |
| #exp- | (2\*6)# | 移进 |
| #exp-( | 2\*6)# | 移进 |
| #exp-(2 | \*6)# | 规约term−>NUMBER |
| #exp-(term | \*6)# | 规约factor−>term |
| #exp-(factor | \*6)# | 移进 |
| #exp-(factor\* | 6)# | 移进 |
| #exp-(factor\*6 | )# | 规约term−>NUMBER |
| #exp-(factor\*term | )# | 规约factor−>factor∗term |
| #exp-(factor | )# | 规约exp−>factor |
| #exp-(exp | )# | 移进 |
| #exp-(exp) | # | 规约term−>(exp) |
| #exp-term | # | 规约factor−>term |
| #exp-factor | # | 规约exp−>exp−factor |
| #exp | # | 接受 |

### 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+2)-(2\*6)语法分析树

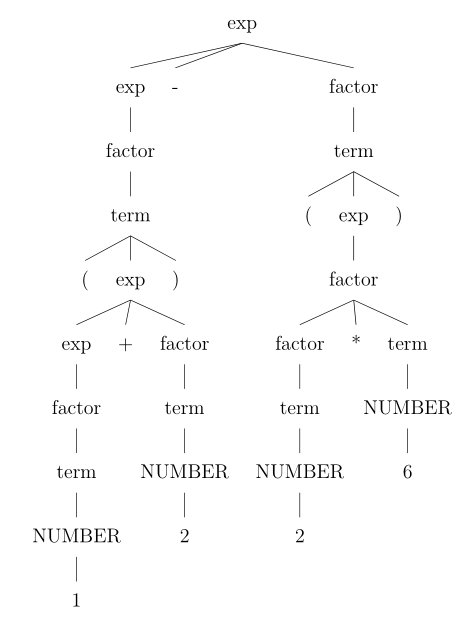


图 1‑5 (1+2)-(2\*6)语法分析树

#### 1+2-3\*2/5语法分析树

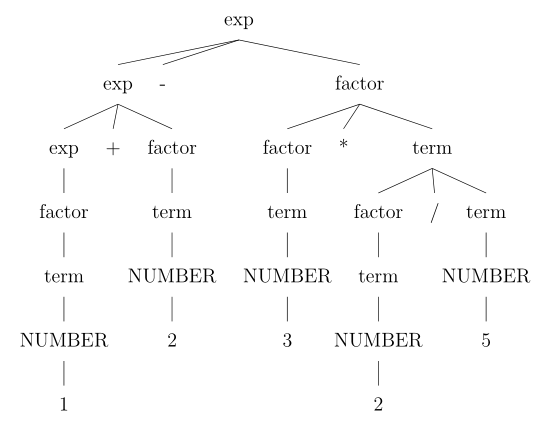


图 1‑6 1+2-3\*2/5语法分析树

## 实验总结

### 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。

### 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！

# 综合实验2

## 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。

## 实验内容

1. 阅读flex Python第三章P60~79，重点学习抽象语法树；
2. 阅读fb3-2.y、fb3-2.l、fb3-2funcs.c、fb3-2.h；
3. 使用内置函数sqrt(n)、exp(n)、log(n)；
4. 定义函数sq(n)、avg(a,b)，用于计算平方根；
5. 撰写实验报告，结合实验结果，给出抽象语法树的构建过程；
6. 提交报告和实验代码。

## 实验步骤

### CentOS

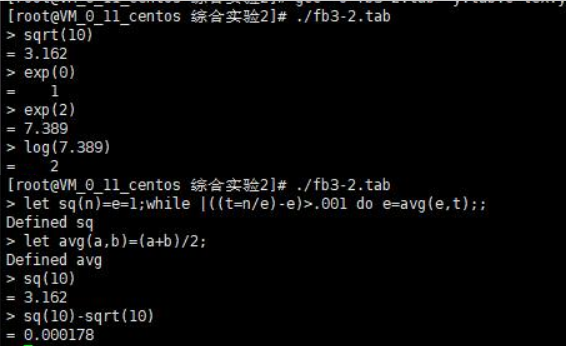


图 2‑1 CentOS运行结果图

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。

### 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

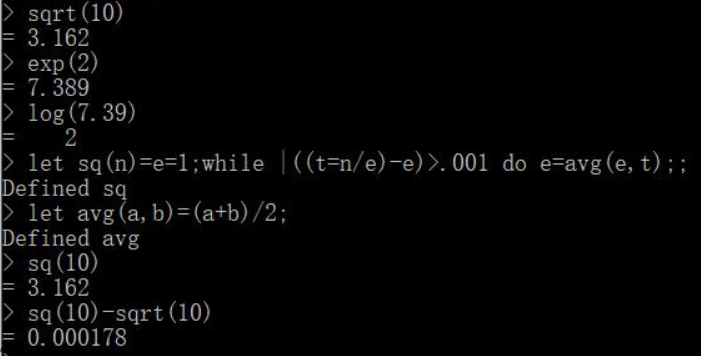


图 2‑2 Windows下运行结果图

## 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。

### 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

|  |  |
| --- | --- |
|  | **%**option noyywrap nodefault yylineno |
|  | **%{** |
|  | # include "fb3-2.h" |
|  | # include "y.tab.h" |
|  | # include <math.h> |
|  | **%}** |
|  | EXP **([**Ee**][-+]?[**0**-**9**]+)** |
|  |  |
|  | **%%** |
|  | */\* single character ops \*/* |
|  | **"+"** **|** |
|  | **"-"** **|** |
|  | **"\*"** **|** |
|  | **"/"** **|** |
|  | **"="** **|** |
|  | **"|"** **|** |
|  | **","** **|** |
|  | **";"** **|** |
|  | **"("** **|** |
|  | **")"** **{** **return** yytext**[**0**];** **}** |
|  |  |
|  | */\* comparison ops \*/* |
|  | **">"** **{** 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**;** **}** |
|  |  |
|  | */\* keywords \*/* |
|  | **"if"** **{** **return** IF**;** **}** |
|  | **"then"** **{** **return** THEN**;** **}** |
|  | **"else"** **{** **return** ELSE**;** **}** |
|  | **"while"** **{** **return** WHILE**;** **}** |
|  | **"do"** **{** **return** DO**;** **}** |
|  | **"let"** **{** **return** LET**;}** |
|  |  |
|  | */\* 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**;** **}** |
|  |  |
|  | */\* debug hack \*/* |
|  | **"debug"[**0**-**9**]+** **{** debug **=** atoi**(&**yytext**[**5**]);** printf**("debug set to %d\n",** debug**);** **}** |
|  |  |
|  | */\* names \*/* |
|  | **[**a**-**zA**-**Z**][**a**-**zA**-**Z0**-**9**]\*** **{** yylval**.**s **=** lookup**(**yytext**);** **return** NAME**;** **}** |
|  |  |
|  | **[**0**-**9**]+"."[**0**-**9**]\*{**EXP**}?** **|** |
|  | **"."?[**0**-**9**]+{**EXP**}?** **{** yylval**.**d **=**atof**(**yytext**);return** NUMBER**;}** |
|  |  |
|  | **"//".\*** |
|  | **[** \t**]** */\* ignore white space \*/* |
|  | \\\n printf**("c> ");** */\* ignore line continuation \*/* |
|  | **"\n"** **{** **return** EOL**;** **}** |
|  |  |
|  | **.** **{** yyerror**("Mystery character %c\n",** **\***yytext**);** **}** |
|  | **%%** |

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。

### 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

|  |  |
| --- | --- |
|  | **%{** |
|  | # include <stdio.h> |
|  | # include <stdlib.h> |
|  | # include "fb3-2.h" |
|  | **%}** |
|  |  |
|  | **%union** **{** |
|  | **struct** ast **\***a**;** |
|  | **double** d**;** |
|  | **struct** symbol **\***s**;** */\* which symbol \*/* |
|  | **struct** symlist **\***sl**;** |
|  | **int** fn**;** */\* which function \*/* |
|  | **}** |
|  |  |
|  | */\* declare tokens \*/* |
|  | **%**token **<**d**>** NUMBER |
|  | **%**token **<**s**>** NAME |
|  | **%**token **<**fn**>** FUNC |
|  | **%**token EOL |
|  | **%**token IF THEN ELSE WHILE DO LET |
|  |  |
|  | **%**nonassoc **<**fn**>** CMP |
|  | **%**right '=' |
|  | **%**left '+' '-' |
|  | **%**left '\*' '/' |
|  | **%**nonassoc '|' UMINUS |
|  |  |
|  | **%**type **<**a**>** exp stmt list explist |
|  | **%**type **<**sl**>** symlist |
|  |  |
|  | **%**start calclist |
|  |  |
|  | **%%** |
|  | 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 **{** $$ **=** newflow**(**'W'**,**$2**,**$4**,** NULL**);}** |
|  | **|** exp |
|  | **;** |
|  |  |
|  | list**:** */\* nothing \*/* **{** $$ **=** NULL**;** **}** |
|  | **|** stmt ';' list **{** **if** **(**$3 **==** NULL**)** |
|  | $$ **=** $1**;** |
|  | **else** |
|  | $$ **=** newast**(**'L'**,** $1**,** $3**);** |
|  | **}** |
|  | **;** |
|  |  |
|  | 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 ')' **{** $$ **=** newcall**(**$1**,** $3**);** **}** |
|  | **;** |
|  |  |
|  | explist**:** exp |
|  | **|** exp ',' explist **{** $$ **=** newast**(**'L'**,** $1**,** $3**);** **}** |
|  | **;** |
|  | symlist**:** NAME **{** $$ **=** newsymlist**(**$1**,** NULL**);** **}** |
|  | **|** NAME ',' symlist **{** $$ **=** newsymlist**(**$1**,** $3**);** **}** |
|  | **;** |
|  |  |
|  | 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**("> ");** **}** |
|  | **;** |
|  | **%%** |

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**。

### 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

|  |  |
| --- | --- |
|  | */\* symbol table \*/* |
|  | **struct** symbol **{** */\* a variable name \*/* |
|  | **char** **\***name**;** |
|  | **double** value**;** |
|  | **struct** ast **\***func**;** */\* stmt for the function \*/* |
|  | **struct** symlist **\***syms**;** */\* list of dummy args \*/* |
|  | **};** |
|  |  |
|  | */\* simple symtab of fixed size \*/* |
|  | #define NHASH 9997 |
|  | **struct** symbol symtab**[**NHASH**];** |
|  |  |
|  | **struct** symbol **\***lookup**(char\*);** |
|  |  |
|  | */\* list of symbols, for an argument list \*/* |
|  | **struct** symlist **{** |
|  | **struct** symbol **\***sym**;** |
|  | **struct** symlist **\***next**;** |
|  | **};** |
|  |  |
|  | **struct** symlist **\***newsymlist**(struct** symbol **\***sym**,** **struct** symlist **\***next**);** |
|  | **void** symlistfree**(struct** symlist **\***sl**);** |
|  |  |
|  | */\* node types* |
|  | *\* + - \* / |* |
|  | *\* 0-7 comparison ops, bit coded 04 equal, 02 less, 01 greater* |
|  | *\* M unary minus* |
|  | *\* L statement list* |
|  | *\* I IF statement* |
|  | *\* W WHILE statement* |
|  | *\* N symbol ref* |
|  | *\* = assignment* |
|  | *\* S list of symbols* |
|  | *\* F built in function call* |
|  | *\* C user function call* |
|  | *\*/* |
|  |  |
|  | **enum** bifs **{** */\* built-in functions \*/* |
|  | B\_sqrt **=** 1**,** |
|  | B\_exp**,** |
|  | B\_log**,** |
|  | B\_pow**,** |
|  | B\_print |
|  | **};** |
|  |  |
|  | */\* nodes in the Abstract Syntax Tree \*/* |
|  | */\* all have common initial nodetype \*/* |
|  |  |
|  | **struct** ast **{** |
|  | **int** nodetype**;** |
|  | **struct** ast **\***l**;** |
|  | **struct** ast **\***r**;** |
|  | **};** |
|  |  |
|  | **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 \*/* |
|  | **};** |
|  |  |
|  | */\* build an AST \*/* |
|  | **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**);** |

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为了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]](#footnote-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()。

### fb3-2funcs.c

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。

#### 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

|  |  |
| --- | --- |
|  | */\* symbol table \*/* |
|  | */\* hash a symbol \*/* |
|  | **static** **unsigned** symhash**(char** **\***sym**)** |
|  | **{** |
|  | **unsigned** **int** hash **=** 0**;** |
|  | **unsigned** c**;** |
|  | **while(**c **=** **\***sym**++)** **{** |
|  | hash **=** hash**\***9 **^** c**;** |
|  | printf**("%c hash: %d\n",**c**,**hash**);** |
|  | **}** |
|  | **return** hash**;** |
|  | **}** |
|  |  |
|  | **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**;** |
|  | **}** |
|  | */\* try the next entry \*/* |
|  | **if(++**sp **>=** symtab**+**NHASH**)** sp **=** symtab**;** |
|  | **}** |
|  | yyerror**("symbol table overflow\n");** |
|  | abort**();** */\* tried them all, table is full \*/* |
|  | **}** |

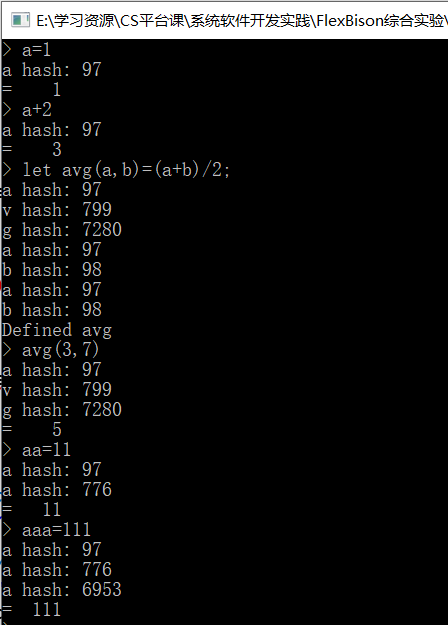


图 2‑3 散列符号表输出分析

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。

#### 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\*)

|  |  |
| --- | --- |
|  | **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**;** |
|  | **}** |

#### 抽象语法树释放

Treefree(struct ast\*)函数，通过针对不同字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。

## 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（symasgn）。下面结合fb3-2funcs.c中的树节点建立代码详细分析。

### 常量（num）

代码 2‑6 newnum(double)

|  |  |
| --- | --- |
|  | **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**;** |
|  | **}** |

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。

### 比较运算（cmp）

代码 2‑7 newcmp(int,struce ast\*,strcut ast\*)

|  |  |
| --- | --- |
|  | **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**;** |
|  | **}** |

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。

### 内置函数（func）

代码 2‑8 newfunc(int ,struct ast\*)

|  |  |
| --- | --- |
|  | **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**;** |
|  | **}** |

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。

### 函数调用（call）

代码 2‑9 newcall(struct symbol\*,struct ast\*)

|  |  |
| --- | --- |
|  | **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**;** |
|  | **}** |

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。

### 变量引用（ref）

代码 2‑10 newref(struct symbol\*)

|  |  |
| --- | --- |
|  | **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**;** |
|  | **}** |

引用变量部分，节点类型‘N’，包含一个子树，链接对应的变量。

### 变量声明（asgn）

代码 2‑11 newasgn(struct symbol\*, struct ast\*)

|  |  |
| --- | --- |
|  | **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**;** |
|  | **}** |

变量声明部分，节点类型‘=’，子树分别为变量名以及变量值。

### 条件分支语句（flow）

代码 2‑12 newflow(int, struct ast\*)

|  |  |
| --- | --- |
|  | **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**;** |
|  | **}** |

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。

### 表达式链（symlist）

代码 2‑13 newsymlist(struct symbol \*, struct symlist \*)

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

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。

### 函数定义（dodef）

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

|  |  |
| --- | --- |
|  | **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**;** |
|  | **}** |

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。

## 语法树求值（Debug输出分析）

### 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\*)

|  |  |
| --- | --- |
|  | **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**;** |
|  | **}** |

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。

#### 计算内置函数返回值

代码 2‑16 callbuiltin(struct fncall\*)

|  |  |
| --- | --- |
|  | **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**);** |
|  | **return** v**;** |
|  | **default:** |
|  | yyerror**("Unknown built-in function %d",** functype**);** |
|  | **return** 0.0**;** |
|  | **}** |
|  | **}** |

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。

#### 计算用户自定义函数返回值

代码 2‑17 calluser(struct ufncall \*)

|  |  |
| --- | --- |
|  | **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**;** |
|  | **}** |

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。

### 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dumpast()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dumpast(struct ast\*,int)打印函数如代码 2‑18所示。

代码 2‑18 dumpast(struct ast \*, int)

|  |  |
| --- | --- |
|  | **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;** |
|  | **}** |
|  | **}** |

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。

### 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。

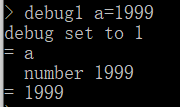
 

图 2‑4 num节点类型与debug输出（a=1999）

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。

### 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 < 22”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.

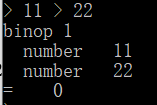
 

图 2‑5 cmp节点类型与debug输出（11<22）

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。

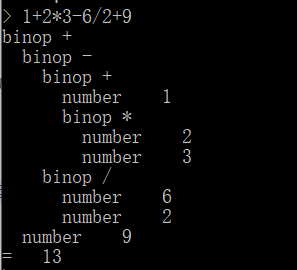


图 2‑6 四则运算综合（1+2\*3-6/2+9）

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。

### 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。

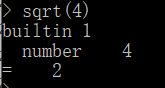
 

图 2‑7 func节点类型及debug输出（sqrt(4)）

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。

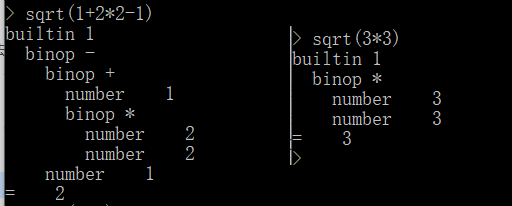


图 2‑8 内置函数参数复杂时debug输出

### 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。

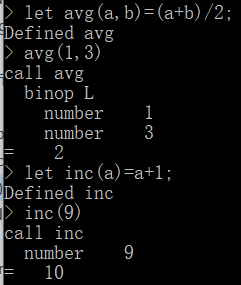
 

图 2‑9 用户定义函数类型与debug输出

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。

### 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\*)

|  |  |
| --- | --- |
|  | **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**;** |
|  | **}** |

### 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。

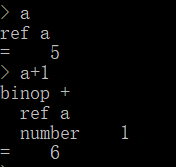
 

图 2‑10 变量引用节点类型与debug输出（a）

### 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。

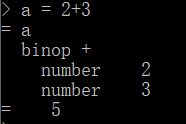
 

图 2‑11 变量声明节点类型与debug输出（a=2+3）

### 条件分支语句（flow）

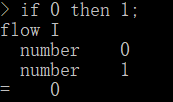
 

图 2‑12 条件分支语句与debug输出（if 0 then 1;）

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。

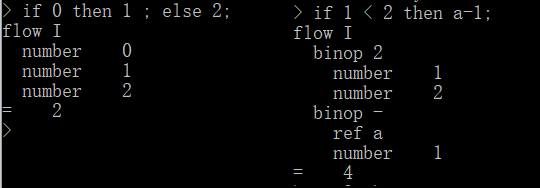


图 2‑13 稍微复杂的条件分支语句输出

当进行更加负责的条件分支语句时，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。

### 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。

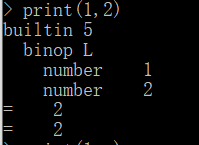
 

图 2‑14 语句链表与debug输出（1,2）

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。

### 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debug打印的抽象语法树。

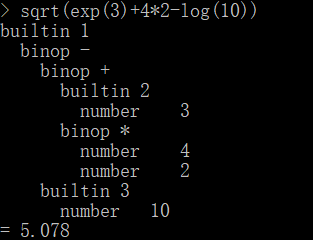
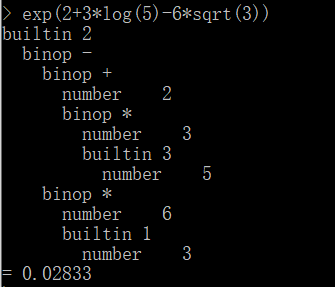
 

图 2‑15 较为复杂情况下debug打印抽象语法树

## 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.l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。

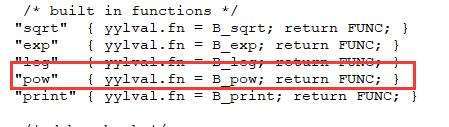


图 2‑16 fb3-2.l修改内容

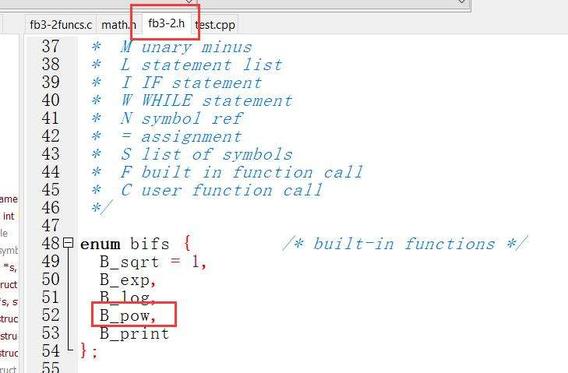


图 2‑17 fb3-2.h修改内容

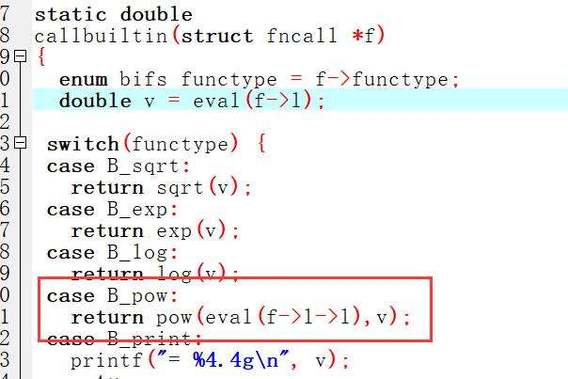


图 2‑18 fb3-2funcs.c文件修改内容

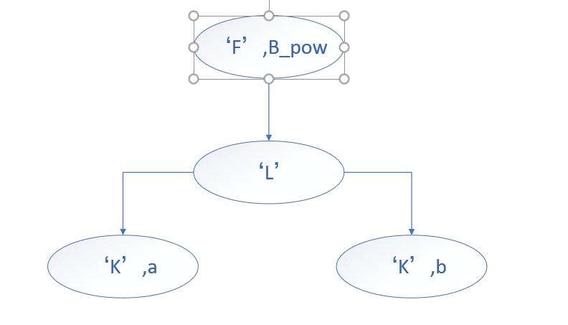
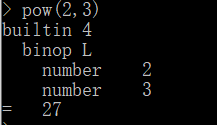
 

图 2‑19 pow(a,b)语法树简图与debug效果图

## 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。

# 综合实验3

## 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。

## 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。

## 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。

### 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

|  |  |
| --- | --- |
|  | **%{** |
|  | # include <stdio.h> |
|  | # include <stdlib.h> |
|  | # include "fb3-3.h" |
|  | **%}** |
|  |  |
|  | **%union** **{** |
|  | **struct** ast **\***a**;** |
|  | **double** d**;** |
|  | **struct** symbol **\***s**;** */\* which symbol \*/* |
|  | **struct** symlist **\***sl**;** |
|  | **int** fn**;** */\* which function \*/* |
|  | **}** |
|  |  |
|  | */\* declare tokens \*/* |
|  | **%**token **<**d**>** NUMBER |
|  | **%**token **<**s**>** NAME |
|  | **%**token **<**fn**>** FUNC |
|  | **%**token EOL |
|  |  |
|  | **%**token IF THEN ELSE WHILE DO LET |
|  |  |
|  |  |
|  | **%**nonassoc **<**fn**>** CMP |
|  | **%**right '=' |
|  | **%**left '+' '-' |
|  | **%**left '\*' '/' |
|  | **%**nonassoc '|' UMINUS |
|  | **%**nonassoc LOWER |
|  | **%**nonassoc ELSE |
|  | **%**left '(' |
|  |  |
|  | **%**type **<**a**>** exp stmt list explist |
|  | **%**type **<**sl**>** symlist |
|  |  |
|  | **%**start calclist |
|  |  |
|  | **%%** |
|  |  |
|  | stmt**:** IF exp THEN list **%**prec LOWER **{** $$ **=** newflow**(**'I'**,** $2**,** $4**,** NULL**);** **}** |
|  | **|** IF exp THEN list ELSE list **{** $$ **=** newflow**(**'I'**,** $2**,** $4**,** $6**);** **}** |
|  | **|** WHILE exp DO '{' list '}' **{** $$ **=** newflow**(**'W'**,** $2**,** $5**,** NULL**);** **}** |
|  | **|** exp ';' **%**prec LOWER |
|  | **;** |
|  |  |
|  | list**:**'{' list '}' **{** $$ **=** $2**;** **}** |
|  | **|** '{' list stmt '}' **{** $$ **=** newast**(**'L'**,**$2**,**$3**);}** |
|  | **|** stmt **{** $$ **=** $1**;}** |
|  | **|** exp **%**prec LOWER**{**$$ **=** $1**;}** |
|  | **;** |
|  |  |
|  | exp**:** exp CMP exp **{** $$ **=** 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 ')'**{** $$ **=** newfunc**(**$1**,** $3**);** **}** |
|  | **|** NAME **%**prec LOWER **{** $$ **=** newref**(**$1**);** **}** |
|  | **|** NAME '=' exp **{** $$ **=** newasgn**(**$1**,** $3**);** **}** |
|  | **|** NAME '(' explist ')' **{** $$ **=** newcall**(**$1**,** $3**);** **}** |
|  | **;** |
|  |  |
|  | explist**:** exp |
|  | **|** exp ',' explist **{** $$ **=** newast**(**'L'**,** $1**,** $3**);** **}** |
|  | **;** |
|  | symlist**:** NAME **{** $$ **=** newsymlist**(**$1**,** NULL**);** **}** |
|  | **|** NAME ',' symlist **{** $$ **=** newsymlist**(**$1**,** $3**);** **}** |
|  | **;** |
|  |  |
|  | 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**,** $7**);** |
|  | printf**("Defined %s\n> ",** $3**->**name**);** **}** |
|  |  |
|  | **|** calclist error EOL **{** yyerrok**;** printf**("> ");** **}** |
|  | **;** |
|  | **%%** |

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 shitf/reduce .

### 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。

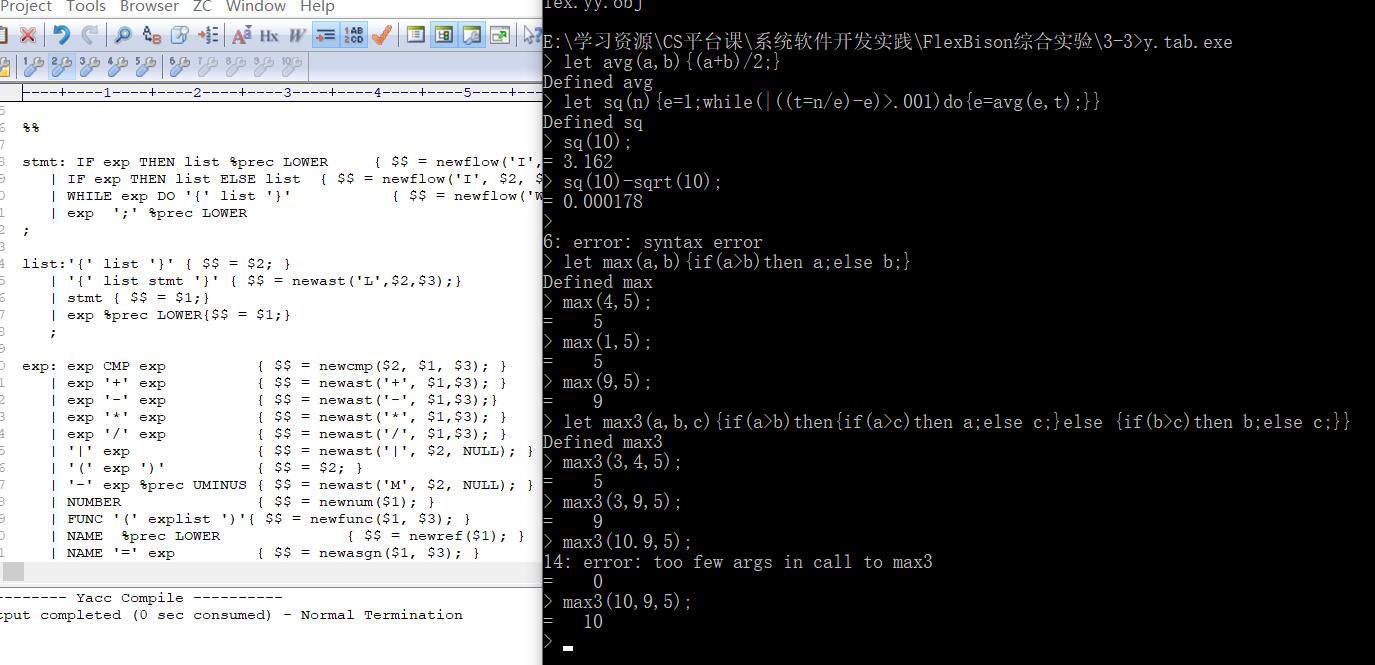
**原因一：**代码 3‑1中第62、63行，当移进NAME后，遇到“(”时程序不能判断是否进行规约或者移进，所以这里需要修改优先级，将NAME改为没有结合性，“(”改为左结合性。在定义部分增加%nonassoc LOWER与%left ’(’即可解决该冲突。

**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 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。

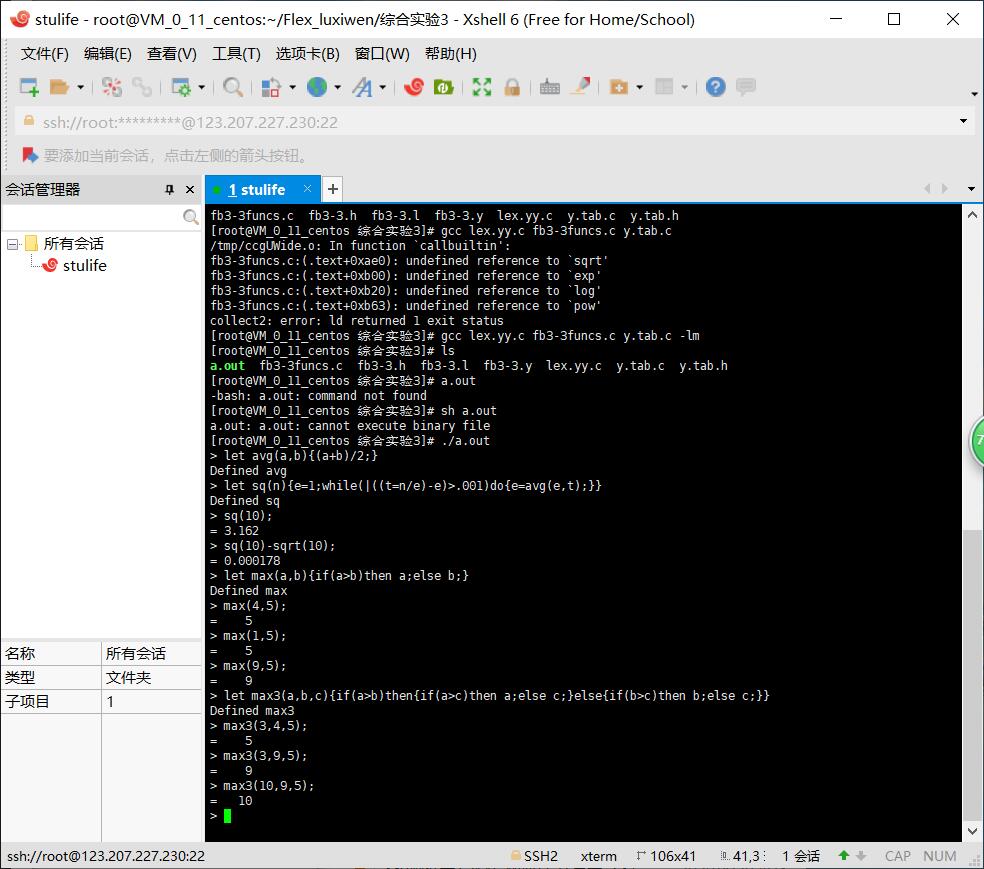
### 运行效果

#### Windows



代码 3‑2 实验三Windows下运行结果

#### CentOS



代码 3‑3 实验三CentOS下运行结果

## 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。

# 综合实验4

## 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：

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。

## 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。

## 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.

### 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/105293836> 。

### 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。

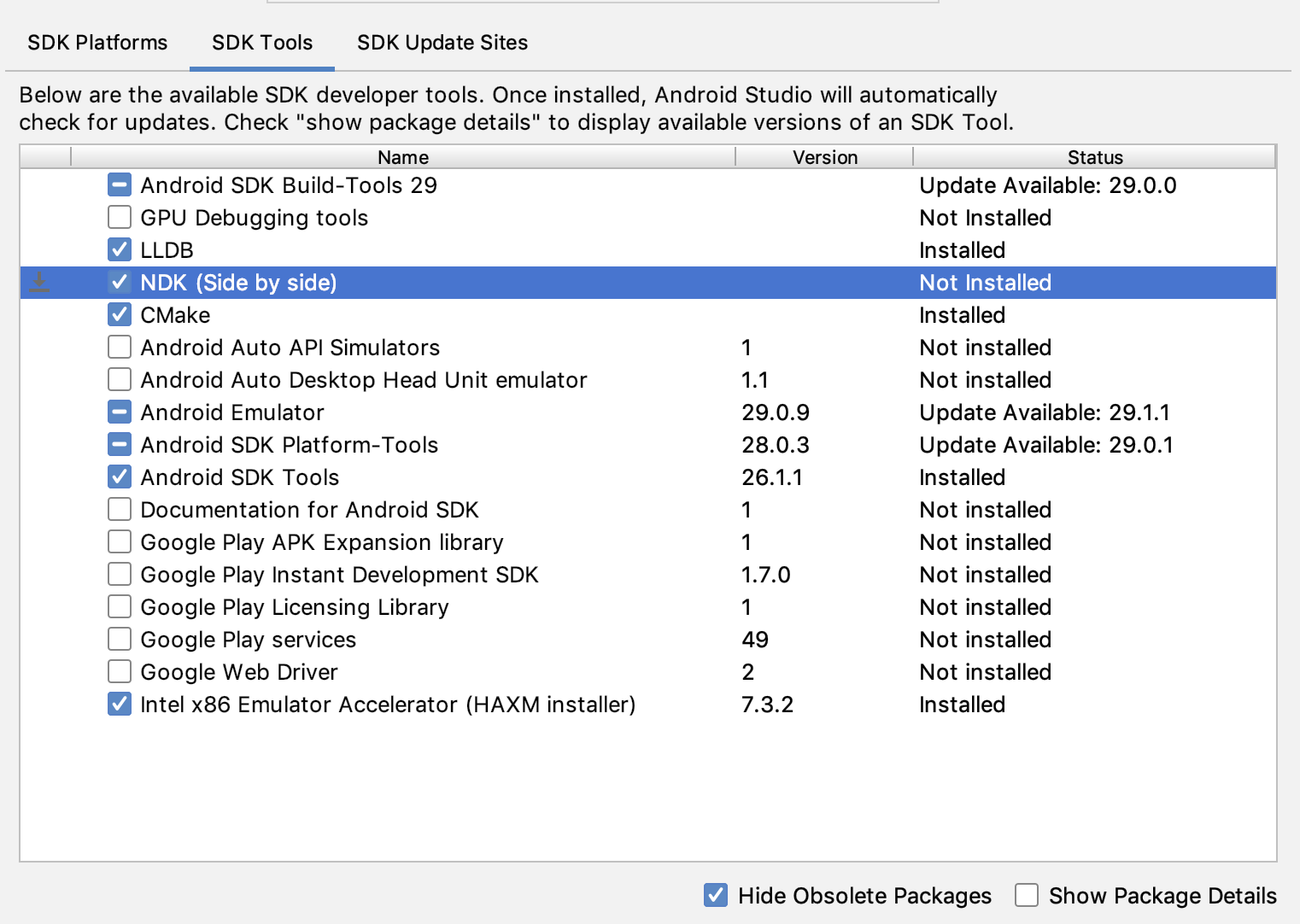


图 4‑1 NDK与CMake安装

### 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。

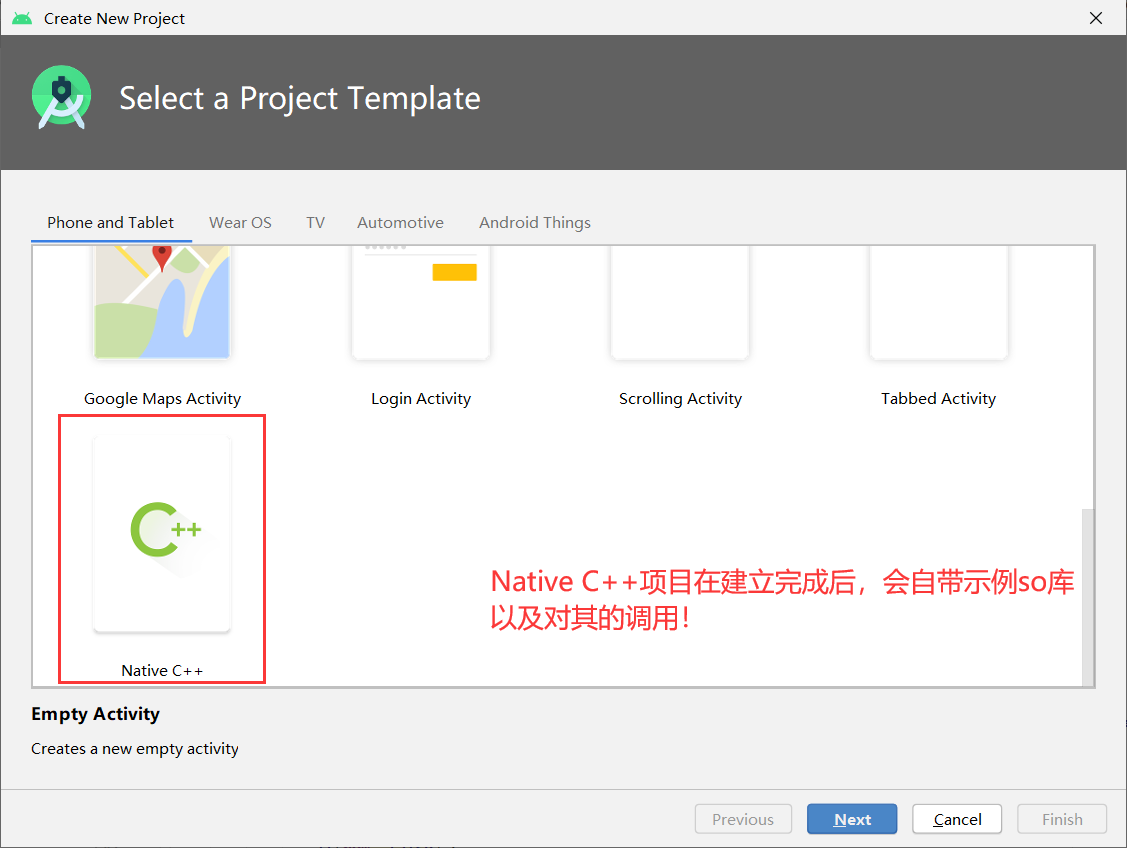


图 4‑2 新建Native C++项目

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]](#footnote-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）

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | **extern "C" JNIEXPORT** jstring **JNICALL** Java\_com\_xiwen\_flexcaculator\_Caculate\_calc(JNIEnv \*env, jclass thiz,  jstring expr) { | |  | **const char**\* s = env->GetStringUTFChars(expr, 0); | |  | jdouble ans = 0; | |  | **return** env->NewStringUTF(calc(s)); | |  | } | |

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。

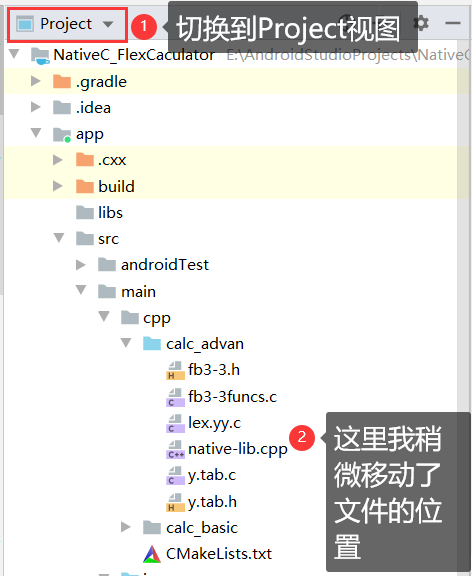


图 4‑3 文件组织示意图

整个项目的文件目录结构如图 4‑4所示（Android视图）。

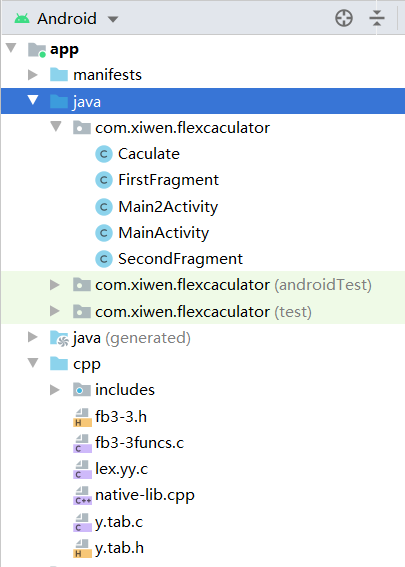


图 4‑4 整体项目目录结构

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

|  |  |
| --- | --- |
|  | **package** com.xiwen.flexcaculator; |
|  |  |
|  | **public class** Caculate { |
|  | **static** { |
|  | System.*loadLibrary*(**"native-lib-flex"**); |
|  | } |
|  | **public static native** String calc(String expr); |
|  | } |

### 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/bestxiaoxiaoming-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。

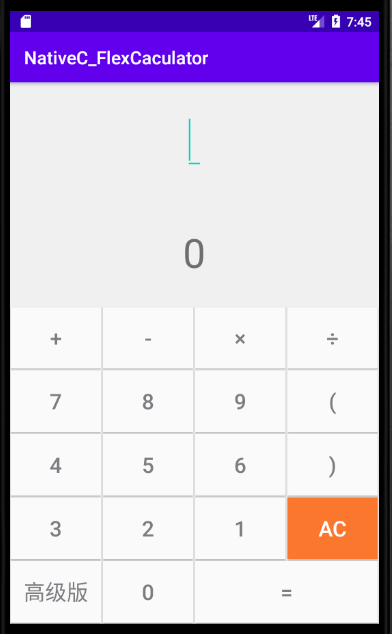
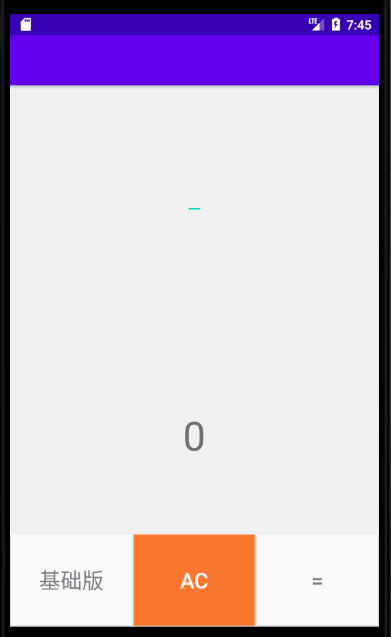
 

图 4‑5 安卓计算器主界面

### 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

|  |  |
| --- | --- |
|  | cmake\_minimum\_required(**VERSION 3.4.1**) |
|  | add\_library(**native-lib-flex** |
|  |  |
|  | **SHARED** |
|  |  |
|  | **calc\_advan/native-lib.cpp calc\_advan/fb3-3funcs.c calc\_advan/lex.yy.c calc\_advan/y.tab.c** |
|  | ) |
|  | find\_library( |
|  | **log-lib** |
|  |  |
|  | **log** ) |
|  |  |
|  | target\_link\_libraries( |
|  | **native-lib-flex** |
|  |  |
|  | **${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。

### 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.l中)

|  |  |
| --- | --- |
|  | **char\*** calc**(const** **char\*** expr**)** |
|  | **{** |
|  | size\_t len **=** strlen**(**expr**);** |
|  | result **=** **(char\*)**malloc**(sizeof(char)\*(**len**+**2**));** |
|  | yy\_switch\_to\_buffer**(**yy\_scan\_string**(**expr**));** |
|  | yyparse**();** |
|  | **return** result**;** |
|  | **}** |

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

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

### 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。

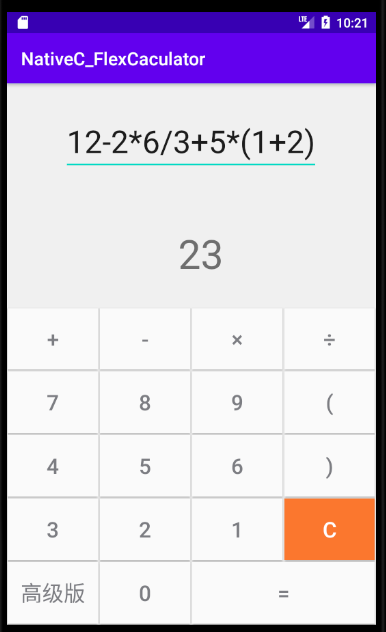
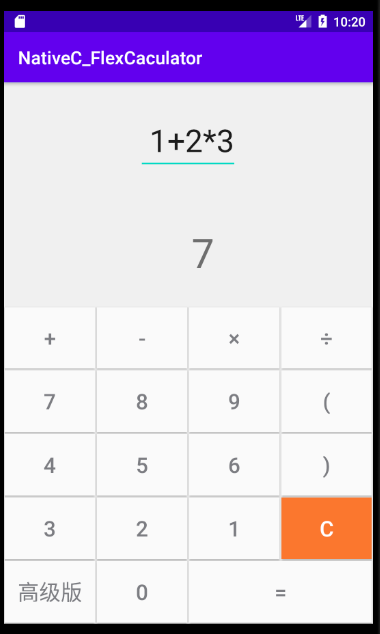


图 4‑6 简单四则运算

自定义函数并调用如图 4‑7、图 4‑8所示。

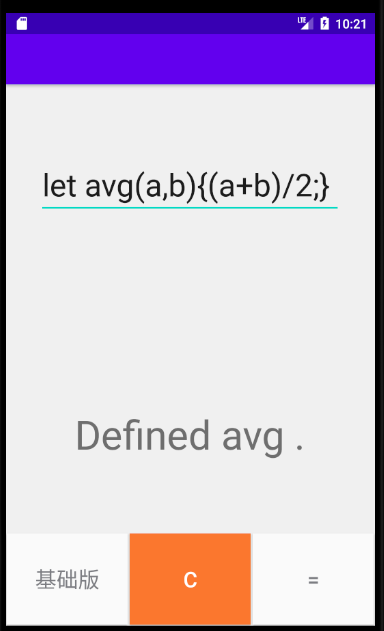


图 4‑7 自定义求平均值

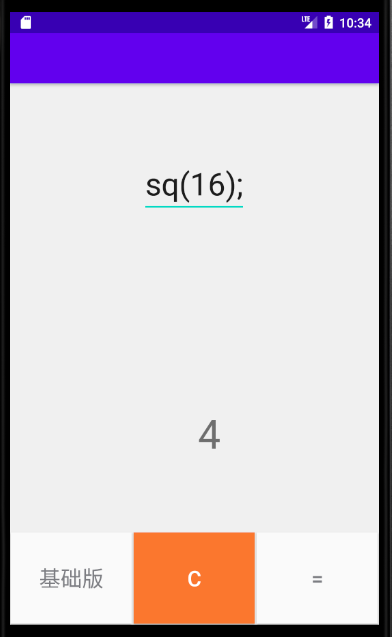
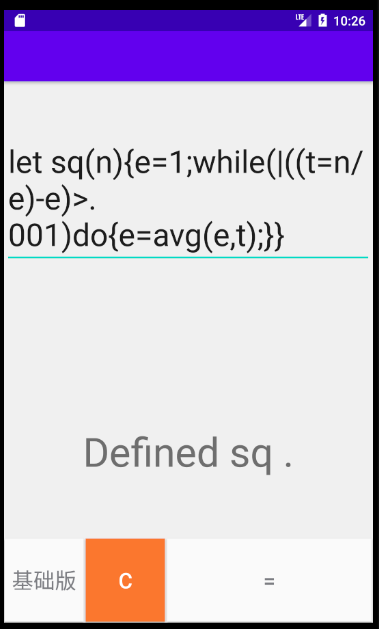


图 4‑8 自定义求平方根

## 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。

1. 此处代码本人已修改，源代码不包含pow(a,b) [↑](#footnote-ref-1)
2. 新建项目无calc-advan目录，该目录为本人实验过程中修改。 [↑](#footnote-ref-2)