成绩

**编译原理实验报告**

通过计算器理解词法、语法与语法制导翻译

学院：计算机学院

班级：10012003

姓名：徐金海

学号：2020302703

日期：2023.05.18

**目录**

**[1实验概述 2](#_Toc11862)**

[1.1要求与目标 2](#_Toc23418)

[1.2总体完成情况 3](#_Toc17985)

**[2主要功能 3](#_Toc25864)**

[2.1现有计算器功能分析 3](#_Toc2642)

[2.2对实数的支持以及对新运算符的支持 4](#_Toc30918)

[2.3对实数运算的支持 4](#_Toc10466)

[2.4语义分析对实数运算的支持 5](#_Toc8919)

**[3软件总体结构 6](#_Toc24586)**

[3.1软件开发环境 6](#_Toc2408)

[3.2软件运行环境 6](#_Toc984)

[3.3软件组成架构 6](#_Toc9373)

[3.4源代码组织与构建 8](#_Toc30463)

**[4详细设计 8](#_Toc29426)**

[4.1变量、整数和实数的表示 8](#_Toc12864)

[4.2加减乘除运算及整数取模运算的实现 12](#_Toc30097)

[4.3对负数表示和负数运算的支持 20](#_Toc6110)

[4.4生成抽象语法树时对实数的支持 21](#_Toc23661)

[4.5线性IR生成的实现 22](#_Toc31697)

[4.6问题：现有计算器代码的优缺点评价 24](#_Toc18466)

**[5测试与结果 24](#_Toc2073)**

[5.1现有计算器的测试 25](#_Toc11312)

[5.2强化计算器整数类型数据完整功能测试 26](#_Toc6149)

[5.3小数点表示的实数测试 27](#_Toc2664)

[5.4科学计数法表示的实数测试 29](#_Toc5897)

[5.5求负运算测试 30](#_Toc30075)

[5.6整数和实数混合运算测试 31](#_Toc9671)

**[6实验总结 32](#_Toc8884)**

[6.1调试和问题修改总结 32](#_Toc26204)

[6.2实验小结 32](#_Toc26635)

**[7实验建议 33](#_Toc18750)**

1实验概述

1.1要求与目标

实验要求：

（1）以计算器为例，理解与掌握Flex如何进行词法分析；

（2）以计算器为例，理解与掌握Bison如何进行语法制导翻译；

（3）以计算器为例，理解与掌握递归下降分析法实现语法制导翻译；

（4）以计算器为例，掌握如何生成抽象语法树，以及深度优先遍历抽象语法树进行算术表达式的运算和产生线性IR；

（5）掌握用Graphviz工具输出抽象语法树；

（6）通过增加实数常量的支持强化计算器，可进行加减乘除求余数求负等

运算。

实验目标：

熟悉使用词法分析程序自动构造工具Flex；熟悉使用语法分析程序自动构造工具Bison；熟悉使用递归下降分析法进行语法分析；理解语法制导翻译生成抽象语法树；理解遍历抽象语法树生成线性IR；理解遍历抽象语法树进行计算器的运算；能够使用Graphviz工具编程实现抽象语法树的图形化显示。

在实验过程中，需要对老师提供的现有计算器的代码进行阅读，并进行优缺点分析。同时，需要在其基础上，完成强化计算器（追加对实数的支持，如小数、科学计数法表示的实数等，以及新运算符的支持等；修改或增加对实数的加减乘除求余以及求负运算的支持；在语法制导中生成抽象语法树时扩展整数到实数的支持；实现整数和实数的混合运算）的设计与实现，同时编写多个测试用例对其加以验证。

1.2总体完成情况

（要定量与定性相结合描述总体完成情况，给出本次实验的自我评价）

2主要功能

下面是对老师提供的现有计算器的功能分析和在实验过程中完成的强化计算器的功能说明。

2.1现有计算器功能分析

现有计算器的源程序是一个基于Flex和Bison的简易计算器程序，可以执行基本的算术运算和赋值操作，并支持变量和常量的定义。程序首先将输入的表达式转化为抽象语法树，然后再通过语义分析将其转为中间代码，在最终执行前完成优化。

现有计算器代码文件calculator.l是Flex语法描述文件，主要实现基于正则表达式的词法分析器，用于将输入的代码流分解成对应的token，被作为后续语法分析器的输入。在这个文件中，每个正则表达式对应一个符号，并且每个符号都有对应的token值，都被定义为枚举类型的值（如T\_ASSIGN），用于后续语法分析器进行调用。这个文件同时还实现了注释和行注释的识别，忽略空格符等和代码相关的字符，对不符合的字符报错。其中还有一些语句用于将正则表达式的识别与对应的token值进行关联，比如"(",")","\*"和"+"等。

而代码文件calculator.y中使用了Lex和Yacc工具实现简单计算器的程序。它可以分析输入的表达式并生成AST（抽象语法树）表示该表达式，同时执行语义动作以便计算该表达式的值。程序定义了五种语法规则，分别对应于语句块、语句、表达式、项和因子，将输入的表达式递归地拆分为语法单元并构造AST。其中，递归下降的语法规则类似于传统的文法定义。程序同时定义了终结符和非终结符的属性，以便在AST构造时进行类型检查和语义分析。例如，标识符被赋值时，程序将其取值赋给对应的AST节点，方便用户后续引用和修改。另外，该程序支持语法错误检查和提示，当输入的表达式不符合语法规则时，程序可以输出相关的错误信息。

同时，现有计算器已经实现了变量定义和整型数据的加法和乘法运算操作，同时也实现了单行和多行注释的识别，以及含括号运算的识别与匹配，后续会加以验证。

其中，flex源文件中给出的变量标识符的正则表达式为{l}+({d}|{l})\*，整型数据的正则表达式为{d}+，加法运算符+和乘法运算符\*返回的语法分析器可识别的token分别为（T\_ADD）和（T\_MUL），单行注释的正则表达式为"//".\*\n，而多行注释使用了LEX中的多重入口机制，当遇到/\*时，使程序进入条件，当条件中出现\n时，不进行任何操作，当出现\*/时，将条件重新设置为初始，退出此条件。如果一直没有/\*的出现，则输出错误信息，并返回一个错误代码。左右括号()返回的token分别为T\_LPAREN和T\_RPAREN。

2.2对实数的支持以及对新运算符的支持

现有计算器仅支持整型数据和其乘法和加法计算，在强化计算器中，我实现了实数数据（DOUBLE）的表示，包括小数、科学计数法表示的实数。

flex源文件中定义了小数表示实数的正则表达式为{d}\*\.{d}+，科学计数法表示的实数的正则表达式为{d}+(\.{d}+)?([eE][-+]?{d}+)，其中分别讨论了正指数和负指数幂的计算，两者返回的token均为DOUBLE。

同时，在原有的+（T\_ADD）和\*（T\_MUL）两个运算符的基础上，在flex源程序中又实现了除法/（T\_DIV），减法-（T\_SUB），取模%（T\_MOD）三个运算符符号的定义，返回一个语法分析器可识别的token。

2.3对实数运算的支持

在原有的计算器代码中，仅支持对整型数据的加法和乘法操作，而经过上面的强化，计算器便可以实现整数和实数的加减乘除运算以及其混合运算，同时也支持整型数据的取模运算。总的来说，强化计算器最终实现了如下的文法：

Statement->IDASSIGNExpression;|Expression;|Expression

Expression->Expression+Term|Expression-Term|Term

Term->Term\*Factor|Term/Factor|Term%Factor|-|Factor

-->-Factor

Factor->(Expression)|DIGIT|DOUBLE|ID

显然，这是一个针对表达式语句的上下文无关文法，共有五个非终结符和四个终结符。

其中，Statement代表一个语句，可以分为三种情况：1.以标识符（ID）为左值的赋值语句，形式为IDASSIGNExpression;2.非赋值的表达式语句，形式为Expression;3.只有一个Expression的语句。

Expression代表一个表达式，可以由下列三种情况递归定义：1.两个Expression的加法或减法运算，形式为Expression+Term或Expression-Term；2.一个Term；3.一个负号(-)，形式为-Expression。

Term代表一个项，可以由下列四种情况递归定义：1.两个Term的乘法、除法或取余运算，分别以Term\*Factor、Term/Factor和Term%Factor形式表示；2.一个负号(-)，形式为-；3.一个因子(Factor)。

负号(-)只能作为Term或Expression的一部分，表示取负操作。

Factor代表一个因子，可以由下列三种情况递归定义：1.用括号括起来的一个Expression，形式为(Expression)；2.一个数字(DIGIT)，形式为一个整数；3.一个浮点数(DOUBLE)；4.一个标识符(ID)。

2.4语义分析对实数运算的支持

在本次实验中采用Graphviz工具生成抽象语法树图片。原理为先遍历左侧已经生成的抽象语法树，再遍历右侧已经生成的抽象语法树，使用Graphviz工具为每个节点创建图形，遍历方法采用深度优先遍历。AST的遍历顺序是从父节点到子节点，在清理AST资源时采用了递归的方式进行清除。在ast.cpp文件中的free\_ast\_node()函数中，首先对节点的所有子节点进行递归清理（因为子节点是嵌套在节点中的，需要先清理子节点资源才能清理节点资源），然后清理所有子节点，将已被删除的子节点从节点的子节点列表中删除，最后再清理节点的资源，从而确保不会出现资源泄漏情况。在递归调用清理子节点的各个节点时，也是从父节点到子节点的进行遍历。

生成线性IR的方法同生成抽象语法树图片的方法类似，通过遍历抽象语法树，生成相应的中间IR，并输出到相应的txt文件。

3软件总体结构

3.1软件开发环境

Windows10+Flex+Bison+YACC，IDE使用的是老师提供的CompileTools中的64为的VScode。

3.2软件运行环境

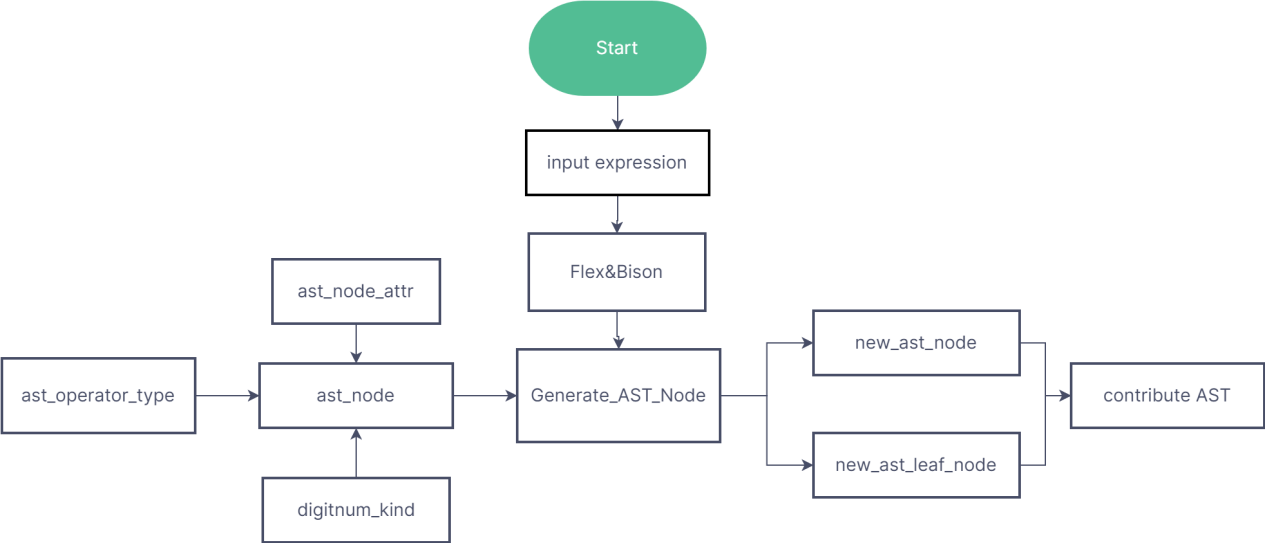
本次实验的C++项目的运行依赖库为Graphviz。

3.3软件组成架构

以下是对现有计算器程序头文件功能的相关说明，便于对整个计算器代码的逻辑进行理解：

| **文件名** | **功能** |
| --- | --- |
| IRCode.h | 中间代码生成部分，定义了一个描述中间代码的类InterCode，以及与其相关的函数和变量。 |
| IRInst.h | 中间代码指令部分，定义了在中间代码中可以使用的各种指令和操作符，以及与其相关的函数和变量。 |
| ast.h | 抽象语法树部分，定义了处理抽象语法树的各种结构体和函数，以及与其相关的变量和宏定义。 |
| calculator\_lex.h | 词法分析器部分，用于声明识别和处理表达式中的单词； |
| calculator\_yacc.h | 语法解析器部分，用于定义对输入表达式进行解析和树构建的函数及相关数据类型的初始化等； |
| common.h | 通用函数部分，包含了对抽象语法树和单词进行操作的函数的定义。 |
| expr.h | AST树表达式计算部分。 |
| graph.h | 输出AST树部分，实现了将抽象语法树以文本格式输出到文件中的函数。 |
| lexer.h | 词法分析程序的头文件，包括一些宏定义和函数声明。 |
| parser.h | 语法分析程序的头文件，定义了语法分析器相关的数据结构和函数。 |
| symbol.h | 符号表部分，定义了在编译过程中会用到的数据结构和函数，包括临时数值、常量和变量型的数值，以及数值的查找和创建等操作。 |

该C++项目的模块层次结构可以概括为下图：



词法分析模块：包括calculator\_lex.l和calculator\_lex.h两个文件，以及lexer.h头文件，包括对应的cpp文件，用于将输入的字符流转换为符号流。

语法分析模块：包括calculator\_yacc.y和calculator\_yacc.h两个文件，以及parser.h头文件，包括对应的cpp文件，用于将符号流转换为抽象语法树。

中间代码模块：包括IRCode和IRInst文件，用于生成和管理中间代码。

符号表模块：包括symbol文件，用于存储和管理变量、临时变量和常量等符号的信息。

AST模块：包括ast文件，用于构建和管理抽象语法树。

输出模块：包括graph文件，用于将AST树输出到文件。

计算模块：包括expr文件，用于对表达式树求值。

这些模块之间存在着相互引用和依赖的关系：

词法分析模块和语法分析模块通过Flex和Bison生成的词法分析器和语法分析器文件相互引用。语法分析模块和中间代码模块之间通过Bison生成的语法分析器和IRCode中声明的中间代码指令类相互交互。符号表模块和中间代码模块之间通过TEMP和VAR指令使用符号表中对应的数据信息。抽象语法树模块和语法分析模块之间通过Bison生成的语法分析器和ast.h中定义的抽象语法树节点类相互交互。输出模块和抽象语法树模块之间通过graph中声明的输出AST树函数相互交互。计算模块和语法分析模块和符号表模块之间通过expr.h中声明的表达式计算函数和symbol中定义的Value类及其子类相互交互。

3.4源代码组织与构建

源代码整体构架为使用Flex工具生成词法分析器，将输入的表达式字符串转化为词法单元（Token）序列。使用Bison工具生成语法分析器，根据词法单元序列分析语法结构并生成抽象语法树。再基于语法分析器生成的语法树，将表达式解析为一个可以进行计算的内部表示形式，为代码生成器（即IR生成器）提供支持。然后将抽象语法树转换为中间代码，也就是具体的计算过程。If语句和Loop语句也可以通过中间代码实现。

在计算表达式时符号表中存储和管理变量和函数信息，可执行计算器程序提供对工程中各种实现的功能的调用入口。

4详细设计

下面将从以下几个方面对整个计算器的实现设计进行分析。

4.1变量、整数和实数的表示

首先是变量、整数、小数点表示的实数、科学计数法表示的实数的正则表达式定义。可以根据正则表达式画出其有限自动图，以便于理解正则式。

在这之前，我们定义了一些数字和字符的宏定义。我在下面通过注释的方式解释其含义：

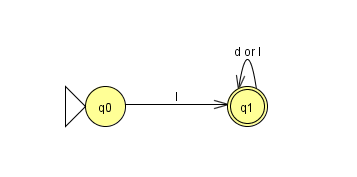
d [0-9] //匹配0到9之间的任意数字

l [a-zA-Z] //匹配从大写字母A到Z和小写字母a到z之间的任意字母

white [\t\040] //\t为Tab，\040为空格（对应的ASCII码为40）

white\_line [\r\n] //分别对应回车和换行符

4.1.1变量：{l}+({d}|{l})\*：



定义其的flex词法规则为：

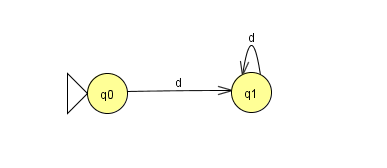
strncpy(yylval.var\_id.id, yytext, sizeof(yylval.var\_id.id));

yylval.var\_id.lineno = yylineno;

return T\_ID;

这段代码的作用是在词法分析器中识别输入中的标识符（变量名），将其复制到yylval结构体变量中，其通过strncpy函数用于将匹配到的标识符（yytext）复制到yylval.var\_id.id字段中，并存储行号信息，然后返回token T\_ID给语法分析器。

4.1.2整数：{d}+：



定义其的flex词法规则为：

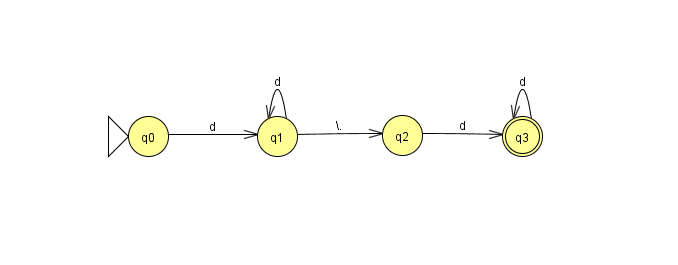
yylval.integer\_num.val = (int)strtol(yytext, (char \*\*)NULL, 10);

yylval.integer\_num.lineno = yylineno;

return T\_DIGIT;

这段代码的作用是在词法分析器中识别输入中的数字，并将其转换为整数类型，这是通过strtol函数实现，通过强制转换为int将其转换为整数。并将结果存储在yylval结构体变量中，同时返回标记token T\_DIGIT给语法分析器。

4.1.3小数点表示的实数：{d}+\.{d}+（注意此处的\为转义符）：



定义其的flex词法规则为：

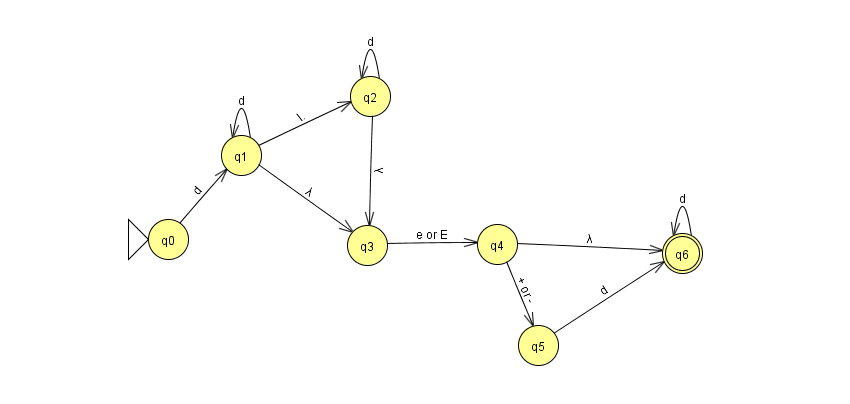
yylval.float\_num.val = atof(yytext);

yylval.float\_num.lineno = yylineno;

return T\_DOUBLE;

这段代码的作用是在词法分析器中识别输入中的浮点数，将其转换为浮点数类型，其是通过atof函数将字符串转换为浮点数，并将结果存储在yylval结构体变量中，同时返回token T\_DOUBLE给语法分析器。

4.1.4科学计数法表示的实数：{d}+(\.{d}+)?([eE][-+]?{d}+)：



注意，上图中的转换边标识为λ为空串。

定义其的flex词法规则为：

int n = strlen(yytext);

double result = 0; //临时变量，暂时转换后的保存结果

char base[n]; //基数部分

char exp[n]; //指数部分

int i = 0; //用于指示输入串的当前位置

int j = 0;

//当碰到e或E时，说明基数部分已经处理完毕

for (i = 0; yytext[i] != 'e' && yytext[i] != 'E'; i++)

base[i] = yytext[i];

//进入指数部分

i++;

result = atof(base);

//若指数为负

if (yytext[i] == '-'){

i++;

for (j = 0; i < n; j++, i++)

exp[j] = yytext[i];

int exponent = atof(exp);

for (j = 0; j < exponent; j++)

result = result / 10;

}

//若指数为正

else if (yytext[i] == '+'){

i++;

for (j = 0; i < n; j++, i++)

exp[j] = yytext[i];

int exponent = atof(exp);

for (j = 0; j < exponent; j++)

result = result \* 10;

}

//没有+号，默认为正

else{

for (j = 0; i < n; j++, i++)

exp[j] = yytext[i];

int exponent = atof(exp);

for (j = 0; j < exponent; j++)

result = result \* 10;

}

yylval.float\_num.val = result;

yylval.var\_id.lineno = yylineno;

return T\_DOUBLE;

这段代码的作用是在词法分析器中识别以科学计数法格式输入的浮点数，其技术部分与之前的处理类似，便不再赘述，重点是e或E之后的指数部分的处理。在上述代码中，根据指数部分的正负情况，进行不同的处理：

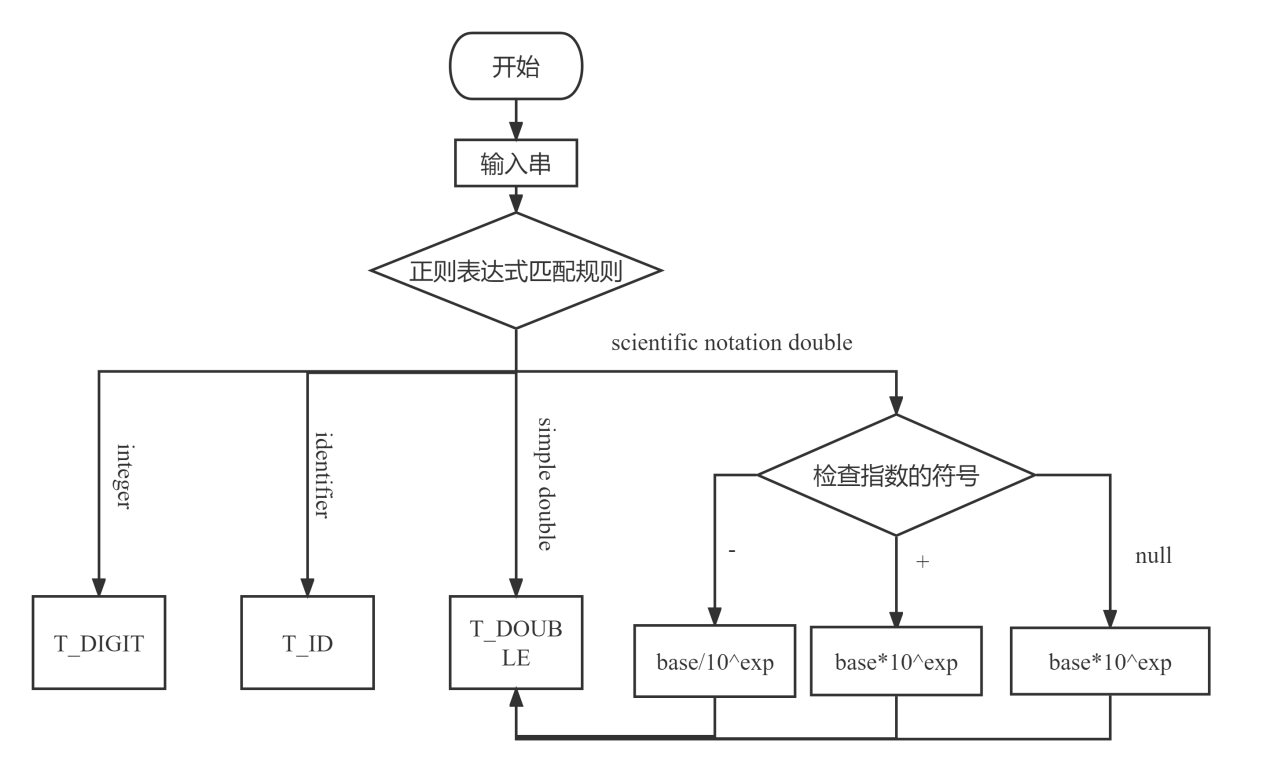
（1）如果指数部分以'-'开始，跳过符号字符，将剩余的字符存储到exp数组中，并将其转换为整数类型（指数部分的值）。循环将result除以10的绝对值次方，用于计算科学计数法中的指数部分。

（2）如果指数部分以'+'开始，跳过符号字符，将剩余的字符存储到exp数组中，并将其转换为整数类型（指数部分的值）。循环将result乘以10的绝对值次方，用于计算科学计数法中的指数部分。

（3）如果指数部分没有正负符号，则默认为正指数将剩余的字符存储到exp数组中，并将其转换为整数类型（指数部分的值）。循环将result乘以10的绝对值次方，用于计算科学计数法中的指数部分。

都转化完成后，再将转换后的浮点数值result赋值给yylval.float\_num.val，用同时返回token T\_DOUBLE给语法分析器。

最终的变量和数据类型的词法定义可用以下流程图表示：



4.2加减乘除运算及整数取模运算的实现

为了实现加减乘除以及整数取模的运算，我们首先要在calcuulator.l文件中给出个运算符的token定义，以便后续的文法分析识别到这些运算符。运算符的定义如下：

"\*" {return T\_MUL;}

"+" {return T\_ADD;}

"/" {return T\_DIV;}

"-" {return T\_SUB;}

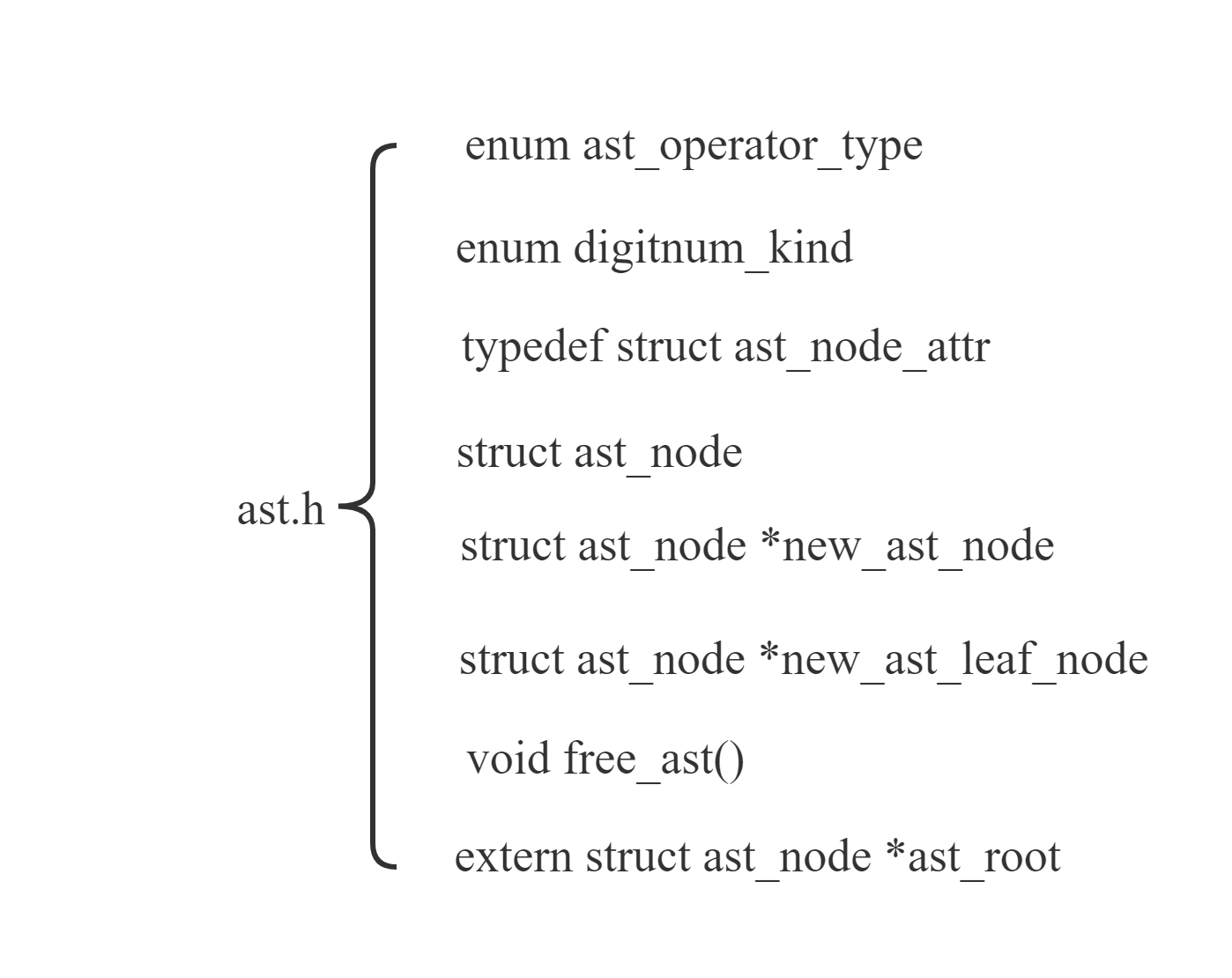
"%" {return T\_MOD;}

接下来我们来分别说明以上几种运算的文法以及计算。

4.2.1 AST的构建

在编译器中，抽象语法树用于表示源代码的语法结构和语义信息。所以在说明具体的计算器文法之前，我们先需要对AST的构建进行定义，以便将文法规则翻译为对应的抽象语法树结构，实现将源代码转换为更高级别的语法结构的过程。

关于AST的构建体现在ast.h和ast.cpp两个源代码文件中，其中ast.h中定义了以下内容（此处不展示代码细节，具体可见src目录下的源代码文件）：

  
 ast\_operator\_type：枚举类型，定义了不同的AST节点类型，包括终结符节点和各种运算符节点。

digitnum\_kind：枚举类型，定义了不同的数字类型，包括整型字面量、实数字面量和变量名称。

ast\_node\_attr：结构体，表示叶子节点的属性值，包括数字类型、行号信息和具体的值。

ast\_node：结构体，表示AST中的一个节点，包含父节点指针、孩子节点指针列表、节点类型、属性值、线性IR指令块和值。

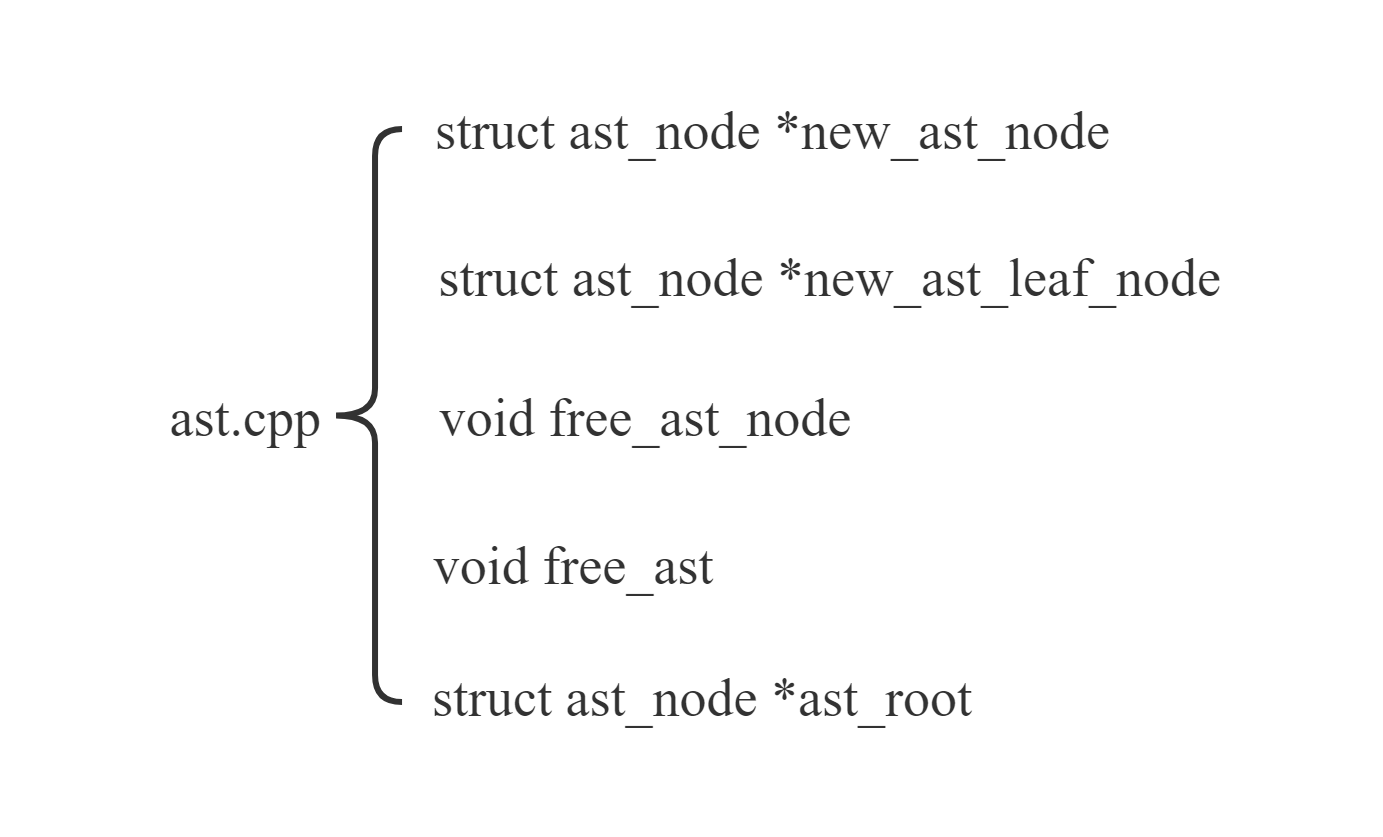
new\_ast\_node：函数，用于创建一个最多有三个孩子节点的AST节点，根据提供的参数创建节点并将孩子节点添加到节点的孩子列表中。

new\_ast\_leaf\_node：函数，用于创建叶子节点或终结符节点，根据提供的属性值创建节点。

free\_ast：函数，用于释放整个抽象语法树的资源，递归地释放节点及其子节点的资源。

ast\_root：指向抽象语法树根节点的全局指针。

而ast.cpp源代码文件中则给出了ast.h中定义的一些函数的具体定义，函数定义的具体内容如下：

  
 new\_ast\_node：具体实现，根据提供的参数创建一个新的AST节点，并将孩子节点添加到节点的孩子列表中。

new\_ast\_leaf\_node：具体实现，根据提供的属性值创建一个新的叶子节点或终结符节点。

free\_ast\_node：递归地释放给定节点及其子节点的资源。

free\_ast：具体实现，释放整个抽象语法树的资源，包括根节点及其所有子节点。

ast\_root：对抽象语法树根节点的全局指针，在ast.cpp中进行定义。

这两个源代码文件提供了创建和操作抽象语法树的基本功能，以及释放资源的函数。便于表示源代码的结构和语义，以便进行后续的语义分析、IR生成等处理。

4.2.2 各运算文法定义

在完成抽象语法树的定义后，我们便可以对计算器实现的文法进行定义，这部分代码位于calculator.y源文件中，该代码是一个Bison（GNU Parser Generator）的输入文件，用于定义一个基本的语法规则，以及对应的语义动作。Bison根据这些规则生成语法分析器（parser）代码，用于解析输入的源代码。

在计算器功能介绍部分已经提到了相关的文法，具体内容如下：

Statement->IDASSIGNExpression;|Expression;|Expression

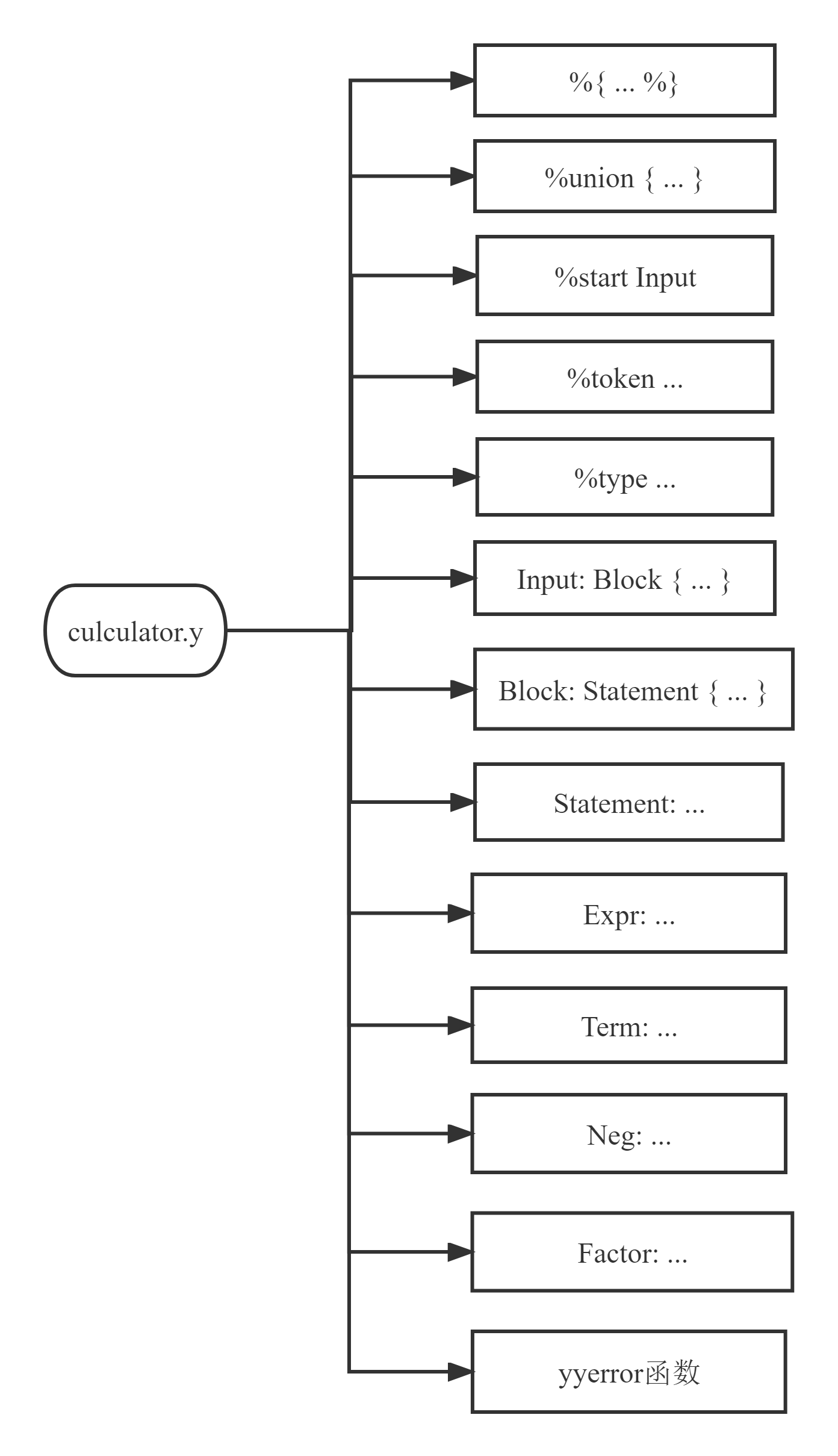
Expression->Expression+Term|Expression-Term|Term

Term->Term\*Factor|Term/Factor|Term%Factor|-|Factor

-->-Factor

Factor->(Expression)|DIGIT|DOUBLE|ID

calculator.y文件中定义了计算器语法的文法规则以及语义动作程序的执行过程，用于构建抽象语法树（AST）来表示计算器输入的语法结构。，该文件含以下内容：



%{ ... %}：这是一个声明部分，其中包含了一些必要的C头文件和函数声明。

%union { ... }：这是一个联合体声明，用于指定终结符和非终结符的属性类型。

%start Input：指定文法的开始符号是Input。

%token ...：指定文法的终结符号，可以指定终结符的属性类型。

%type ...：指定文法的非终结符号，可以指定非终结符的属性类型。

Input: Block { ... }：定义了语法规则Input，表示整个输入。

Block: Statement { ... }：定义了语法规则Block，表示一个语句块。

Statement: ...：定义了语法规则Statement，表示一个语句。

Expr: ...：定义了语法规则Expr，表示一个表达式。

Term: ...：定义了语法规则Term，表示一个项。

Factor: ...：定义了语法规则Factor，表示一个因子。

Neg: ...：定义了语法规则Neg，表示一个负号。

yyerror函数：定义了语法识别错误时的错误处理函数。

下面以Expression->Expression+Term|Expression-Term|Term这句文法的定义为例进行分析，其代码为：

Expr : Expr T\_ADD Term

{

/\* Expr = Expr + Term \*/

$$ = new\_ast\_node(AST\_OP\_ADD, $1, $3);

}

| Expr T\_SUB Term

{

/\* Expr = Expr - Term \*/

$$ = new\_ast\_node(AST\_OP\_SUB, $1, $3);

}

| Term

{

/\* Expr = Term \*/

$$ = $1;

};

上述代码的内容包含三个产生式规则，用于表示不同的表达式形式。

Expr : Expr T\_ADD Term：这个产生式表示表达式由两个表达式和一个加法运算符组成。语义动作程序将创建一个新的抽象语法树节点，类型为AST\_OP\_ADD，它的孩子节点是前一个表达式$1和后面的项$3。$$表示产生式的结果，即新创建的抽象语法树节点。

Expr : Expr T\_SUB Term：这个产生式表示表达式由两个表达式和一个减法运算符组成。语义动作程序将创建一个新的抽象语法树节点，类型为AST\_OP\_SUB，它的孩子节点是前一个表达式$1和后面的项$3。$$表示产生式的结果，即新创建的抽象语法树节点。

Expr : Term：这个产生式表示表达式只有一个项。在这种情况下，不需要创建新的节点，直接将项$1作为结果返回。$$表示产生式的结果，即项本身。

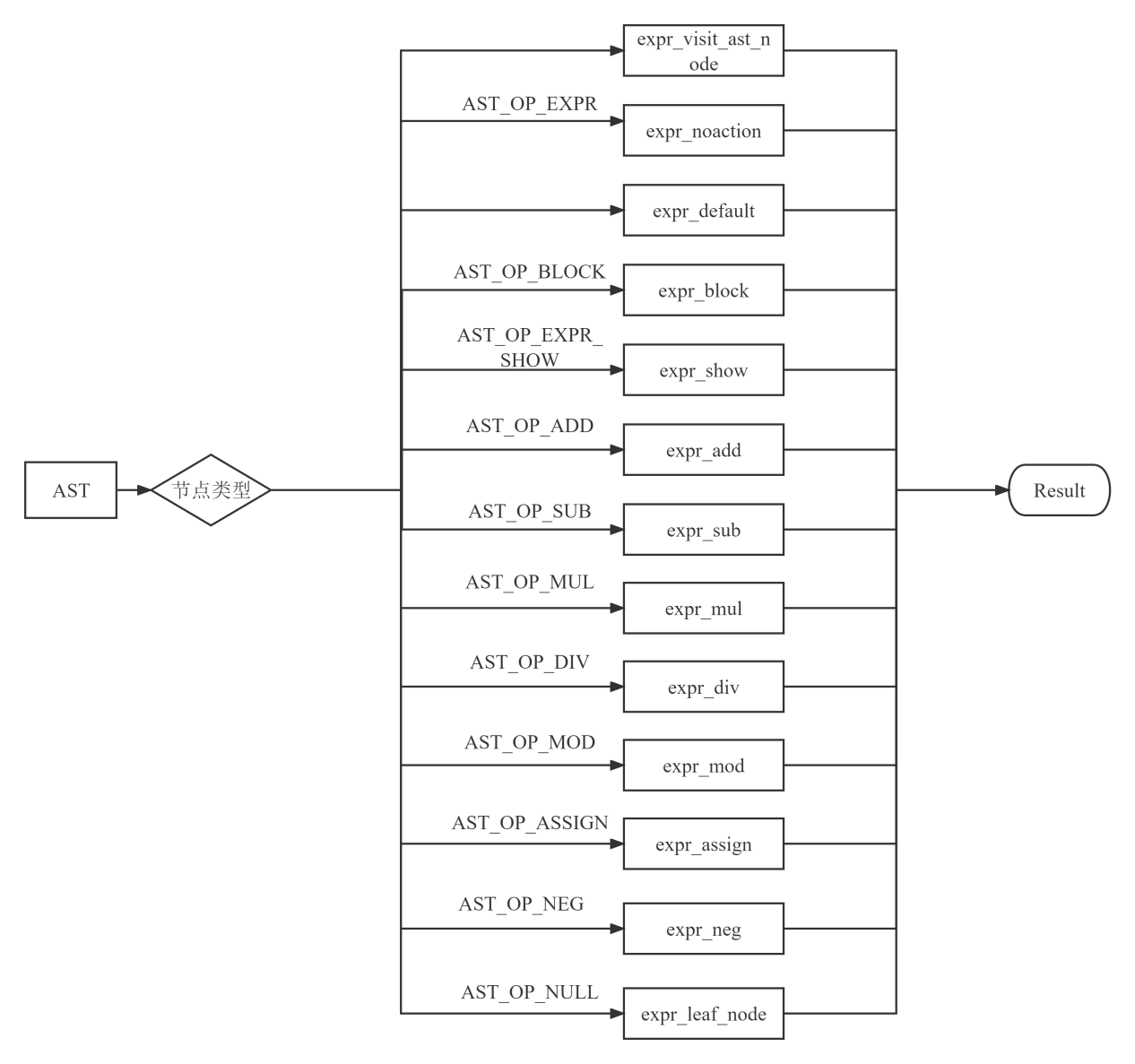
这些产生式规则定义了表达式的不同形式，包括加法运算、减法运算和单个项的情况。根据输入的语法结构，相应的语义动作程序将构建相应的抽象语法树节点，用于表示表达式的结构和含义。

其余正则式的分析与其类似，此处不再赘述。

4.2.3 各运算计算实现

这部分功能主要通过表达式解析器实现，这部分代码主要定义在eper.cpp文件中，其中定义了处理表达式的程序。其中包含了一系列函数，每个函数处理不同类型的表达式节点，并返回一个布尔值表示处理是否成功。

以下是代码中定义的函数及其功能，其可以用于遍历AST：



expr\_visit\_ast\_node函数：用于遍历抽象语法树的节点，并根据节点类型调用相应的处理函数。

expr\_noaction函数：空操作函数，无需执行任何动作，始终返回true。

expr\_default函数：默认操作函数，用于处理不认识的节点操作符，打印未知节点类型并返回false。

expr\_block函数：处理代码块（Block）节点，遍历块中的每个语句，并调用expr\_visit\_ast\_node函数进行处理。

expr\_show函数：处理表达式打印节点，根据节点的值类型打印对应的结果。

expr\_add函数：处理加法节点，根据操作数的类型执行加法操作，并将结果存储在节点的值中。

expr\_sub函数：处理减法节点，根据操作数的类型执行减法操作，并将结果存储在节点的值中。

expr\_mul函数：处理乘法节点，根据操作数的类型执行乘法操作，并将结果存储在节点的值中。

expr\_div函数：处理除法节点，根据操作数的类型执行除法操作，并将结果存储在节点的值中。

expr\_mod函数：处理取模运算节点，执行取模运算，并将结果存储在节点的值中。

expr\_assign函数：处理赋值节点，根据赋值语句的右侧值和左侧变量的类型进行赋值操作，并将结果存储在节点的值中。

expr\_neg函数：处理取负节点，对操作数执行取负操作，并将结果存储在节点的值中。

expr\_leaf\_node函数：处理叶子节点，根据节点的类型创建对应类型的值，并将其存储在节点的值中。

以上函数的具体代码实现此处便不一一说明，此处以除法的函数expr\_div的代码为例进行详细说明，其余函数实现与其类似：

static bool expr\_div(struct ast\_node \*node)

{

// TODO real number mul

struct ast\_node \*son1\_node = node->sons[0];

struct ast\_node \*son2\_node = node->sons[1];

struct ast\_node \*left = expr\_visit\_ast\_node(son1\_node);

if (!left) {

return false;

}

// 乘法的右边操作数

struct ast\_node \*right = expr\_visit\_ast\_node(son2\_node);

if (!right) {

// 某个变量没有定值

return false;

}

Value \*leftValue = left->val;

Value \*rightValue = right->val;

if (leftValue->type == ValueType::ValueType\_Int && rightValue->type == ValueType::ValueType\_Int) {

if (rightValue->intVal == 0) {

printf("Dividend can't be 0");

} else {

// 整数/整数

node->val = newConstValue(leftValue->intVal / rightValue->intVal);

}

} else {

if (leftValue->type == ValueType::ValueType\_Int) {

// 左整型，右实数

leftValue->type = ValueType::ValueType\_Real;

leftValue->realVal = leftValue->intVal;

}

if (rightValue->type == ValueType::ValueType\_Int) {

// 左实数，右整型

rightValue->type = ValueType::ValueType\_Real;

rightValue->realVal = rightValue->intVal;

}

if (rightValue->realVal == 0) {

printf("Dividend can't be 0");

} else {

node->val = newConstValue(leftValue->realVal / rightValue->realVal);

}

}

return true;

}

函数的具体解释如下：

（1）函数定义：static bool expr\_div(struct ast\_node \*node)

这是一个静态函数，只在当前文件中可见，用于处理除法表达式。函数接受一个指向struct ast\_node类型的指针作为参数，并返回一个布尔值。

（2）变量定义：

struct ast\_node \*son1\_node = node->sons[0];定义了一个指向struct ast\_node类型的指针son1\_node，并将其初始化为node的第一个子节点。struct ast\_node \*son2\_node = node->sons[1];定义了一个指向struct ast\_node类型的指针son2\_node，并将其初始化为node的第二个子节点。struct ast\_node \*left = expr\_visit\_ast\_node(son1\_node);定义了一个指向struct ast\_node类型的指针left，并将其初始化为调用expr\_visit\_ast\_node函数处理son1\_node的结果。expr\_visit\_ast\_node函数的作用是访问和处理AST节点，返回处理后的结果。

（3）判断左操作数是否有效：

if (!left) { return false; }如果左操作数无效（即left为空指针），则返回false，表示除法表达式无法计算。

（4）处理右操作数：

struct ast\_node \*right = expr\_visit\_ast\_node(son2\_node);定义了一个指向struct ast\_node类型的指针right，并将其初始化为调用expr\_visit\_ast\_node函数处理son2\_node的结果。同样，这里处理了除法表达式的右操作数。

（5）判断右操作数是否有效：

if (!right) { return false; }如果右操作数无效（即right为空指针），则返回false，表示除法表达式无法计算。

（6）获取操作数的值：

Value \*leftValue = left->val;将左操作数的值存储在leftValue中。Value \*rightValue = right->val;将右操作数的值存储在rightValue中。

（7）根据操作数类型进行除法计算：

**如果左右操作数都是整数类型：**

if (rightValue->intVal == 0) { printf("Dividend can't be 0"); }如果除数为0，则打印错误信息"Dividend can't be 0"。else { node->val = newConstValue(leftValue->intVal / rightValue->intVal); }否则，计算两个整数的商，并将结果存储在node->val中。

**如果操作数类型不全是整数：**

1、如果左操作数是整数类型：

leftValue->type = ValueType::ValueType\_Real;将左操作数的类型更改为实数类型。leftValue->realVal = leftValue->intVal;将左操作数的实数值设为原整数值。

2、如果右操作数是整数类型：

rightValue->type = ValueType::ValueType\_Real;将右操作数的类型更改为实数类型。rightValue->realVal = rightValue->intVal;将右操作数的实数值设为原整数值。

if (rightValue->realVal == 0) { printf("Dividend can't be 0"); }如果除数为0，则打印错误信息"Dividend can't be 0"。else { node->val = newConstValue(leftValue->realVal / rightValue->realVal); }否则，计算两个实数的商，并将结果存储在node->val中。

（8）返回处理结果：

return true;表示除法表达式已成功计算。

4.3对负数表示和负数运算的支持

关于这部分在calculator.y文件中的文法定义已经在上面提到过，此处便不再说明，此处我将说明其计算表达式的实现：

static bool expr\_neg(struct ast\_node \*node)

{

struct ast\_node \*son\_node = node->sons[0];

// 操作数

struct ast\_node \*right = expr\_visit\_ast\_node(son\_node);

if (!right) {

// 某个变量没有定值

return false;

}

Value \*rightValue = right->val;

if (rightValue->type == ValueType::ValueType\_Int) {

node->val = newConstValue(0 - rightValue->intVal);

}

if (rightValue->type == ValueType::ValueType\_Real) {

node->val = newConstValue(0 - rightValue->realVal);

}

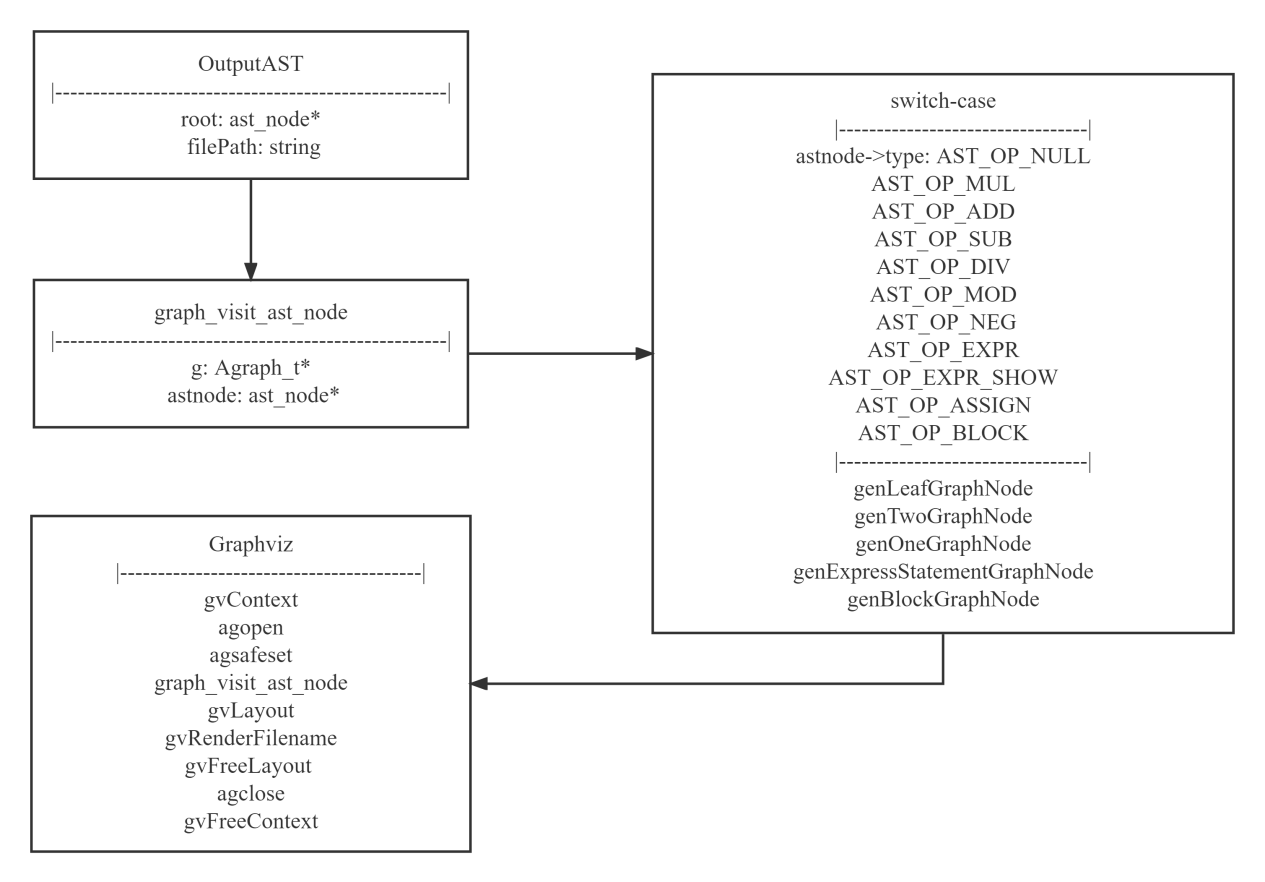
return true;

}

从上面代码可以看出，实现求负计算的核心是将左操作数置为0，再与右操作数（被求负的数）做减法操作，然后将其值赋给对应的根节点。

4.4生成抽象语法树时对实数的支持

生成抽象语法树的代码主要是通过调用Graphviz库来实现的，其具体实现定义在graph.cpp中。下图展示其函数调用关系：

 该代码文件的结构如下：

引用和头文件包含：代码开始部分包含了一些引用和头文件的包含，其中包括<gvc.h>、<iostream>、<string>、<vector>以及自定义的头文件 "ast.h" 和 "common.h"。

全局函数：代码中定义了一些全局函数，如getNodeLabelName()和getEdgeLabelName()。

函数实现：接下来是一系列函数的实现，包括生成不同类型图形节点的函数genExpressStatementGraphNode()、genTwoGraphNode()、genOneGraphNode()、genLeafGraphNode()和genBlockGraphNode()，以及用于遍历抽象语法树节点的函数graph\_visit\_ast\_node()。

OutputAST()函数：该函数是整个程序的入口函数。它接受抽象语法树的根节点和输出文件路径作为参数。函数内部首先创建了Graphviz的上下文和一个图形对象，然后调用graph\_visit\_ast\_node()函数遍历抽象语法树并生成图形表示。接着设置图形的布局为"dot"，并将图形渲染为指定格式的文件，最后清理图形上下文和释放资源。

综上，这个代码文件通过调用不同的函数来生成抽象语法树的图形表示。它利用Graphviz库创建和布局图形，并将图形保存为文件。代码结构清晰，将不同的功能模块化，使得代码的可读性和可维护性都比较高。

4.5线性IR生成的实现

genIR.cpp是用于生成中间代码（Intermediate Code）的代码。它接受一个AST作为输入，并通过遍历抽象语法树生成相应的中间代码。

函数genIR是生成中间代码的入口函数，它接受一个抽象语法树和一个用于保存中间代码的对象（IRCode）。它首先调用ir\_visit\_ast\_node函数对抽象语法树进行遍历，然后将生成的中间代码添加到IRCode中。

函数ir\_visit\_ast\_node是一个递归函数，用于遍历抽象语法树的节点并生成中间代码。它根据节点的类型执行不同的操作。以下是每个节点类型对应的操作：

AST\_OP\_NULL: 叶子节点，根据节点的属性生成相应的值（变量或常量）。

AST\_OP\_MUL: 乘法节点，根据左右子节点生成乘法的中间代码。

AST\_OP\_DIV: 除法节点，根据左右子节点生成除法的中间代码。

AST\_OP\_ADD: 加法节点，根据左右子节点生成加法的中间代码。

AST\_OP\_SUB: 减法节点，根据左右子节点生成减法的中间代码。

AST\_OP\_EXPR: 表达式节点，不显示表达式的值，生成相应的中间代码。

AST\_OP\_EXPR\_SHOW: 表达式节点，显示表达式的值，生成相应的中间代码。

AST\_OP\_ASSIGN: 赋值语句节点，根据左右子节点生成赋值的中间代码。

AST\_OP\_BLOCK: 块节点，遍历块中的每个语句节点并生成相应的中间代码。

在生成中间代码过程中，会涉及一些操作数的处理，例如根据节点的属性生成变量或常量的值，并将生成的中间代码添加到节点的blockInsts属性中。

上面内容的是实现是基于IRInst.cpp中定义的4个类实现的，（代码冗长，报告内不便展示）下面我将介绍这些类：

IRInst类：

IRInst类是IR指令的基类，它表示一个通用的IR指令。

构造函数IRInst::IRInst()用于初始化IR指令的操作码（op）为最大值。另一个构造函数IRInst::IRInst(IRInstOperator \_op, Value \*\_result)用于初始化IR指令的操作码（op）和目标操作数（dstValue）。析构函数IRInst::~IRInst()为空，没有特定的实现。IRInst::getOp()函数用于获取指令的操作码。IRInst::getSrc()函数用于获取源操作数列表。IRInst::getDst()函数用于获取目标操作数或结果操作数。

IRInst::toString()函数用于将IR指令转换为字符串表示。

BinaryIRInst类：

BinaryIRInst类继承自IRInst类，表示二元运算的IR指令。

构造函数BinaryIRInst::BinaryIRInst(IRInstOperator \_op, Value \*\_result, Value \*\_srcVal1, Value \*\_srcVal2)用于初始化二元运算指令的操作码、目标操作数和源操作数。析构函数BinaryIRInst::~BinaryIRInst()为空，没有特定的实现。BinaryIRInst::toString()函数根据操作码将二元运算指令转换为对应的字符串表示。

FuncCallIRInst类：

FuncCallIRInst类继承自IRInst类，表示函数调用的IR指令。

构造函数FuncCallIRInst::FuncCallIRInst(std::string \_name)用于初始化函数调用指令的函数名。另一个构造函数FuncCallIRInst::FuncCallIRInst(std::string \_name, Value \*\_srcVal1, Value \*\_result)用于初始化函数调用指令的函数名、源操作数和目标操作数。还有一个构造函数FuncCallIRInst::FuncCallIRInst(std::string \_name, std::vector<Value \*> &\_srcVal, Value \*\_result)用于初始化函数调用指令的函数名、源操作数列表和目标操作数。析构函数FuncCallIRInst::~FuncCallIRInst()为空，没有特定的实现。FuncCallIRInst::toString()函数将函数调用指令转换为对应的字符串表示。

AssignIRInst类：

AssignIRInst类继承自IRInst类，表示赋值操作的IR指令。

构造函数AssignIRInst::AssignIRInst(Value \*\_result, Value \*\_srcVal1)用于初始化赋值指令的目标操作数和源操作数。函数AssignIRInst::~AssignIRInst()为空，没有特定的实现。AssignIRInst::toString()函数将赋值指令转换为对应的字符串表示。

4.6问题：现有计算器代码的优缺点评价

代码解读和细节分析在上面介绍计算器的内容中已经提到，此处便不再继续说明，此处总结一下现有计算器代码的优缺点：

**优点：**

自动化：使用Flex和Bison工具可以自动生成对应的词法分析器和语法分析器，减少手动编写编译器所需的工作量，提高开发效率。

可维护性：模块化结构及自动生成的代码结构使得程序易于维护和修改。

灵活性高：使用Flex和Bison不仅可以方便地添加新的功能，同时还可以灵活地改变语法规则。这使我们可以很容易地创建自定义编程语言、模板库以及代码生成器等工具。

可读性：由于使用了模块化结构以及自动生成的代码结构，Flex和Bison生成的代码都比较规范，易于阅读和理解。

可靠性：由于这些工具经过良好的测试和成熟的改进，因此它们通常是相对稳定的，也拥有大量的开发社区支持，有各种示例和教程，在使用过程中可以快速得到帮助。

**缺点：**

在编写语法规则时，需要了解工具的相关知识，这可能成为使用这些工具的门槛。

系统错误提示不一定友好，需要花费时间编写适当的错误提示规则，以便更好地调试程序。

可能出现编译时间过长的问题，由于自动生成代码方式会产生很多代码，可能导致编译时间过长。

生成的代码结构可能相对复杂，在开发人员不熟悉代码的情况下，可能会增加调试和维护的难度。

可能出现性能问题，语法分析器可能会面临一些性能问题，因为词法分析和语法分析可能非常繁重，而生成的代码可能不够高效。

5测试与结果

在分析具体测试之前，我们可以先对readme.md中的5条测试指令的功能进行分析。

.\cmake-build-debug\calculator-flex-bison.exe-atest.txt

在执行该指令时，程序calculator-flex-bison.exe将会针对输入文件test.txt进行词法分析和语法分析，然后输出生成的抽象语法树。

.\cmake-build-debug\calculator-flex-bison.exe-a-otest.pdftest.txt

该指令与第一个指令的功能一致，不同之处在于它额外指定了输出文件的文件名为test.pdf。因此，在执行该指令时，程序将会在生成的抽象语法树的基础上，将其输出到指定的文件中。

.\cmake-build-debug\calculator-flex-bison.exe-rtest.txt

该指令表明程序将会对输入文件test.txt进行词法分析和语法分析，并输出生成的语法树。不同于第一个指令，该指令输出语法树而不是抽象语法树。语法树比抽象语法树更详细，同时也更为复杂。

.\cmake-build-debug\calculator-flex-bison.exe-r-oir.txttest.txt

该指令表明程序将会对输入文件test.txt进行词法分析和语法分析，并输出生成的中间代码。不同于前两个指令，该指令生成的不再是语法树，而是中间代码。该指令还额外指定将中间代码输出到ir.txt。

.\cmake-build-debug\calculator-flex-bison.exe-Rtest.txt

该指令与第四个指令类似，不过它生成的不是中间代码，而是汇编代码。在执行该指令时，程序将会对输入文件test.txt进行完整的语法分析，并将运行结果输出到屏幕上。

5.1现有计算器的测试

在实验中，编写了一个测试文件integerTest.txt文件对现有计算器的整型数据的加法和乘法运算进行了测试。其中测试文件为：

a=5;

b=3\*a+7;

//显示a的值

a

//显示b的值

b

/\*

\*显示a+b的值

\*/

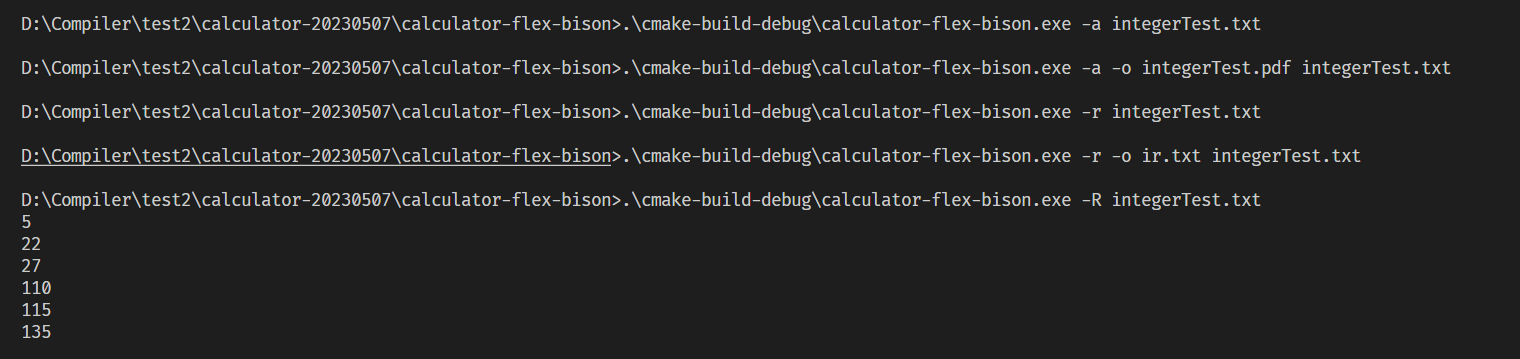
a+b

//显示a\*b的值

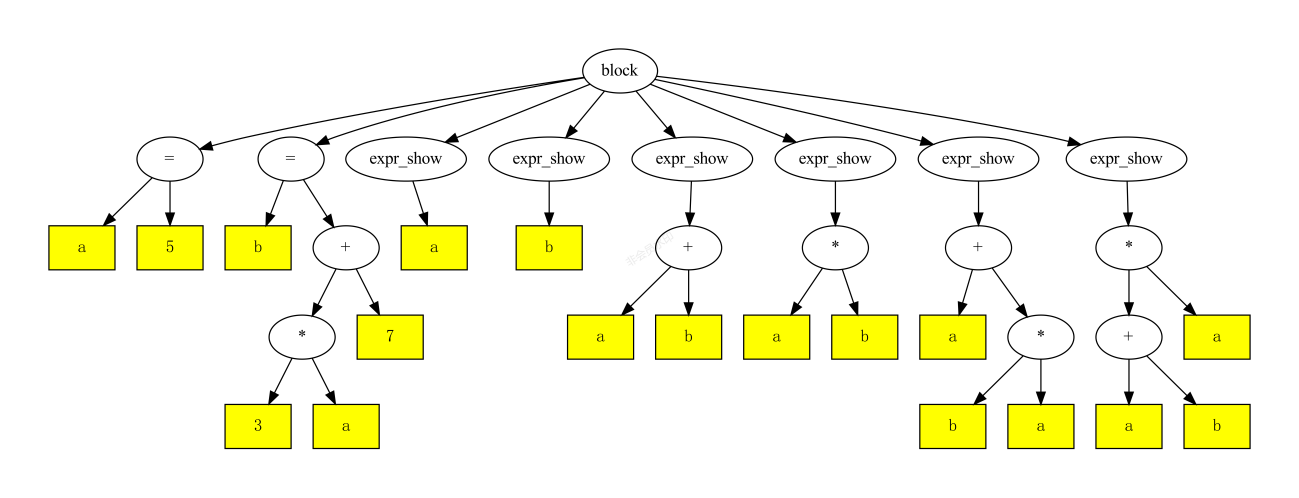
a\*b

//验证括号

a+b\*a

(a+b)\*a

测试文件中包含了多行和单行注释，整数的定义，实现了两个整型数据的加法和乘法计算，同时含有乘法和加法优先级的验证以及含括号的表达式计算。理论上程序应该依次输出5，22，27，110，115，135这6个数，上图是测试结果，与预期值一致，成功通过测试。



同时，在测试过程中，产生了上图所示的AST，与测试文件integerTest.txt中的测试内容完全一致。尤其是倒数的两个子树，正确地展示了乘法和加法的优先级，同时也展示了括号在表达式中的绝对优先级。

5.2强化计算器整数类型数据完整功能测试

在这部分测试中，使用了已经设计好的强化计算器程序进行测试，测试内容包含了整数的加减乘除，以及取模运算，同时包含含括号的表达式计算，测试文件为integerAllFuncTest.txt。

//定义4个初始测试值

integer1 = 123;

integer2 = 456;

integer3 = 45;

integer4 = 9;

//加减乘除测试

//理论值依次为:168,447,0,405

integer1 + integer3

integer2 - integer4

integer1 / integer2

integer3 \* integer4

//取模运算以及含括号的表达式的计算

//理论值依次为:123,0,20643,26055,133,12

integer1 % integer2

integer3 % integer4

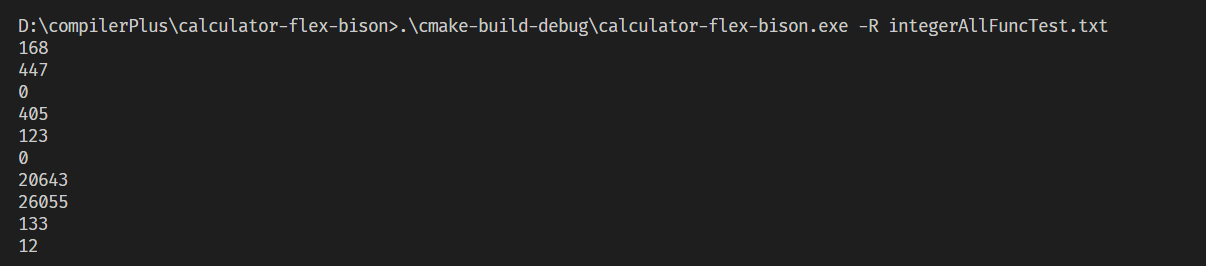
integer1 + integer2 \* integer3

(integer1 + integer2) \* integer3

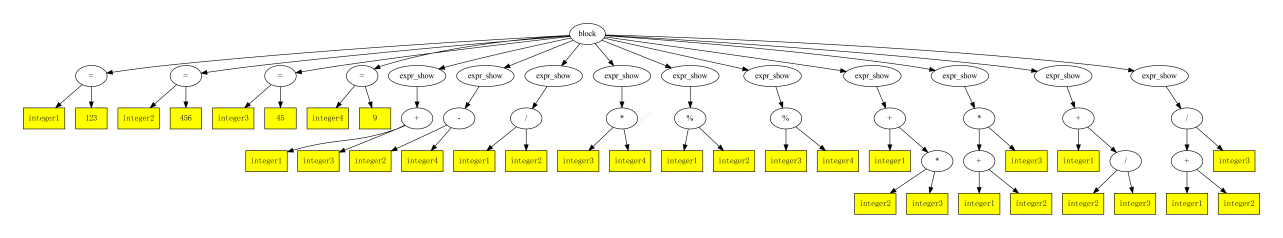
integer1 + integer2 / integer3

(integer1 + integer2) / integer3

运行程序测试的结果如下图：



很显然，测试的结果与测试文件中的理论值结果完全一致。同时，我生成了对应测试程序的抽象语法树，其如下图所示：



抽象语法数展示的内容也与上面的测试文件完全吻合，通过测试。

5.3小数点表示的实数测试

在这部分测试中，使用了已经设计好的强化计算器程序进行测试，测试内容包含了小数点表示的小数的加减乘除运算，同时包含含括号的表达式计算，测试文件为doubleTest.txt。其内容如下：

double1 = 3.14;

double2 = -1.23;

double3 = -5.34;

double4 = 2.56;

//初始值检查

double1

double2

double3

double4

//理论值依次为1.910000,-4.370000,-3.862200,-0.391720

double1 + double2

double2 - double1

double1 \* double2

double2 / double1

//理论值依次为-2.780000,7.900000,-13.670400,-0.4794001

double3 + double4

double4 - double3

double3 \* double4

double4 / double3

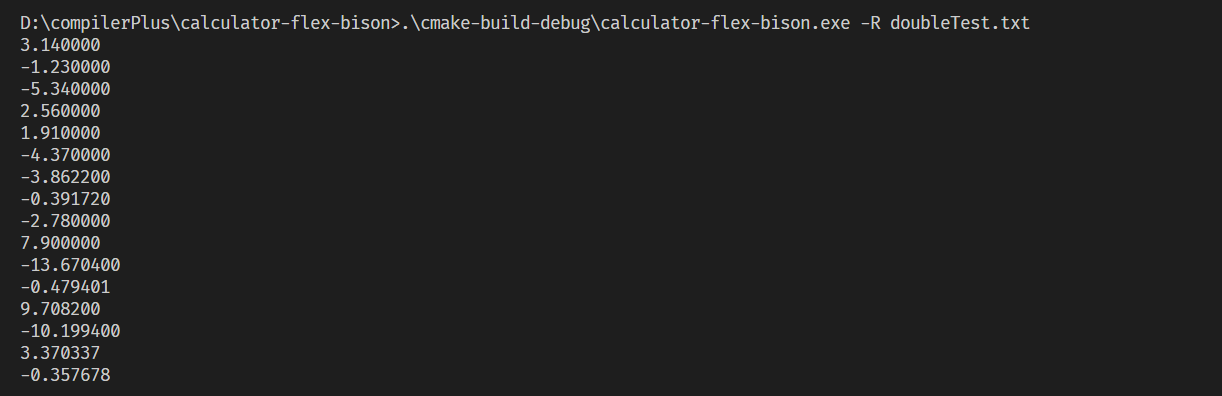
//理论值依次为9.708200,10.199400,3.370337,-0.357678

double1 + double2 \* double3

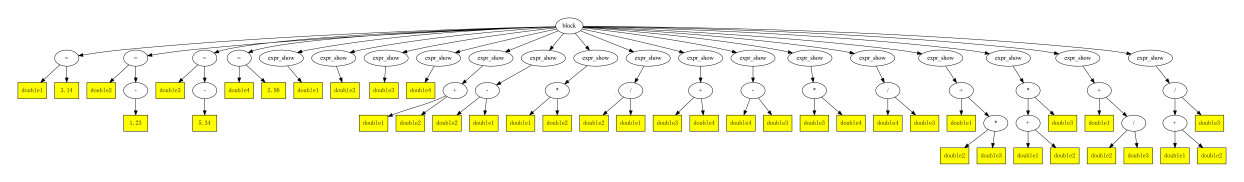
(double1 + double2) \* double3

double1 + double2 / double3

(double1 + double2) / double3

运行计算器程序进行测试，得到的结果如下所示：  


很显然，测试的结果与测试文件中的理论值结果完全一致。计算器程序功能正常、正确。同时，我生成了对应测试程序的抽象语法树，其如下图所示：



抽象语法数展示的内容也与上面的测试文件完全吻合，通过测试。

5.4科学计数法表示的实数测试

在这部分测试中，使用了已经设计好的强化计算器程序进行测试，测试内容包含了小数点表示的小数的加减乘除运算，同时包含含括号的表达式计算，测试文件为doubleTest.txt。其内容如下：

//定义4个初始测试值

scidouble1 = 2e-4;

scidouble2 = 3e+3;

scidouble3 = 2.1e-3;

scidouble4 = 6.3e+3;

//初始值检查

scidouble1

scidouble2

scidouble3

scidouble4

//理论值依次为3000.000200,-2999.999800,0.600000,15000000.000000

scidouble1 + scidouble2

scidouble2 - scidouble1

scidouble1 \* scidouble2

scidouble2 / scidouble1

//理论值依次为6300.002100,6299.997900,13.230000,3000000.000000

scidouble3 + scidouble4

scidouble4 - scidouble3

scidouble3 \* scidouble4

scidouble4 / scidouble3

//理论值依次为6.300200,6.300000,1428571.428771,1428571.523810

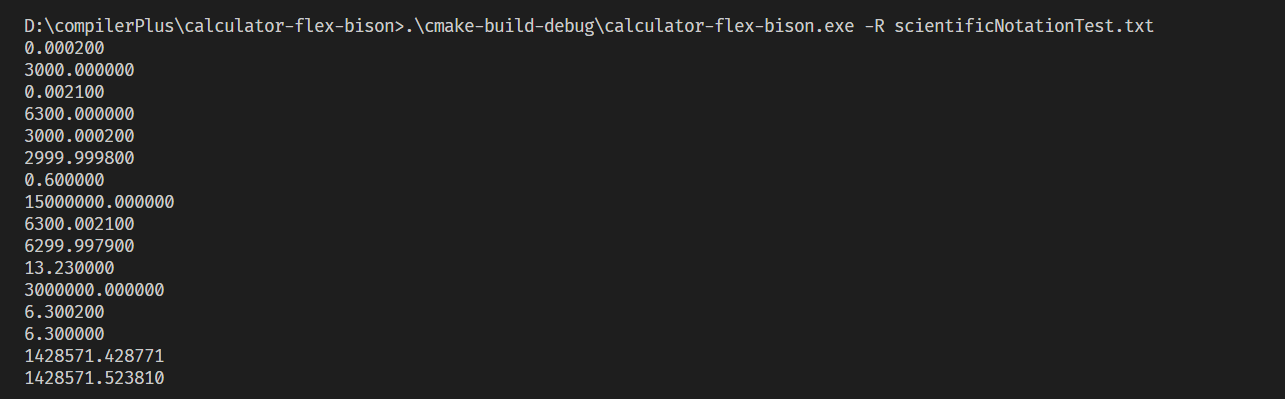
scidouble1 + scidouble2 \* scidouble3

(scidouble1 + scidouble2) \* scidouble3

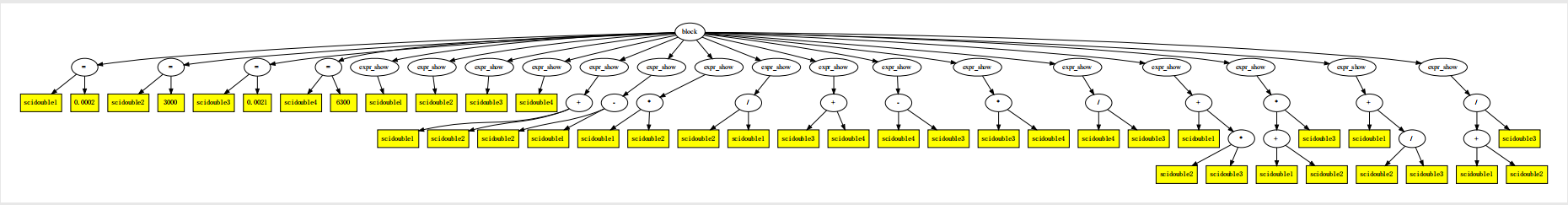
scidouble1 + scidouble2 / scidouble3

(scidouble1 + scidouble2) / scidouble3

运行程序测试的结果如下图：



很显然，测试的结果与测试文件中的理论值结果完全一致。同时，我生成了对应测试程序的抽象语法树，其如下图所示：



抽象语法数展示的内容也与上面的测试文件完全吻合，通过测试。

5.5求负运算测试

在这部分测试中，使用了已经设计好的强化计算器程序进行测试，测试内容包含了对各种数据类型的数据的求负运算，测试文件为negNumTest.txt。其内容如下：

//初始化3个类型的测试数据

integertest = 3;

doubletest = 3.14;

scidoubletest = 5e3;

//检查测试值是否与初始化的一致

integertest

doubletest

scidoubletest

//求负运算

integertest -integertest;

doubletest = -doubletest;

scidoubletest = -scidoubletest;

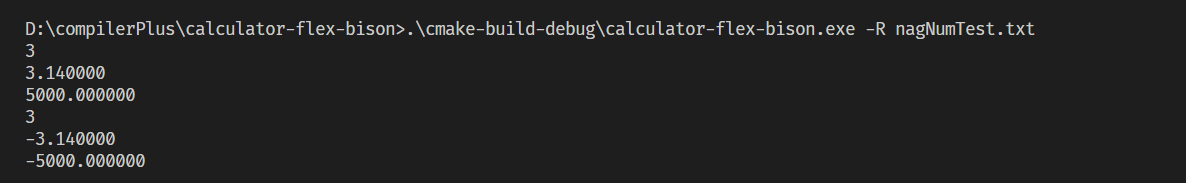
//求负结果 理论值为：3，-3.140000，-5000.000000

integertest

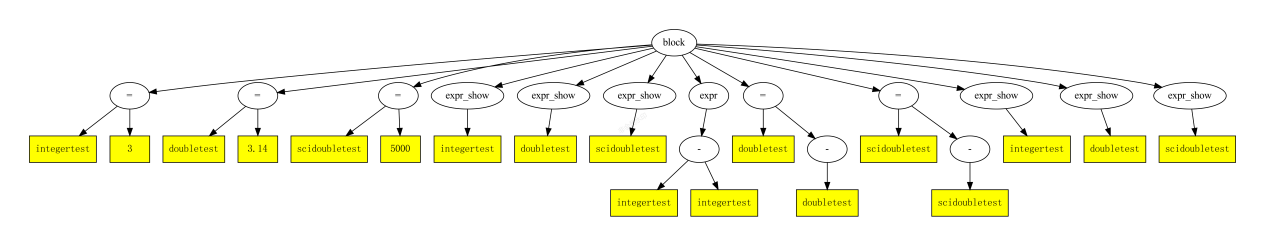
doubletest

scidoubletest

运行程序测试的结果如下图：



通过上面结果，可以清楚看到定义的各种类型的数据被正确地打印在终端，且经过求负运算后，对应的数值也变成了其相应的负数。然后，生成了测试文件得到的抽象语法树：



生成的抽象语法树也与测试文件完全对应，求负值功能也通过了测试。

5.6整数和实数混合运算测试

强化计算器还要求支持整数与实数的混合计算，故编写了以下测试文件combineTest.txt文件来进行测试：

//初始化一些测试数据

int1 = 2;

int2 = 7;

double1 = 2.0;

double2 = 7.0;

//混合计算测试,理论值依次为3，14，3.500000，3.500000，14.000000，14.000000

int2 / int1

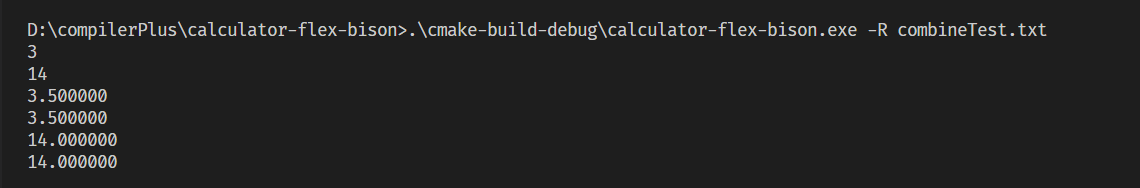
int1 \* int2

double2 / double1

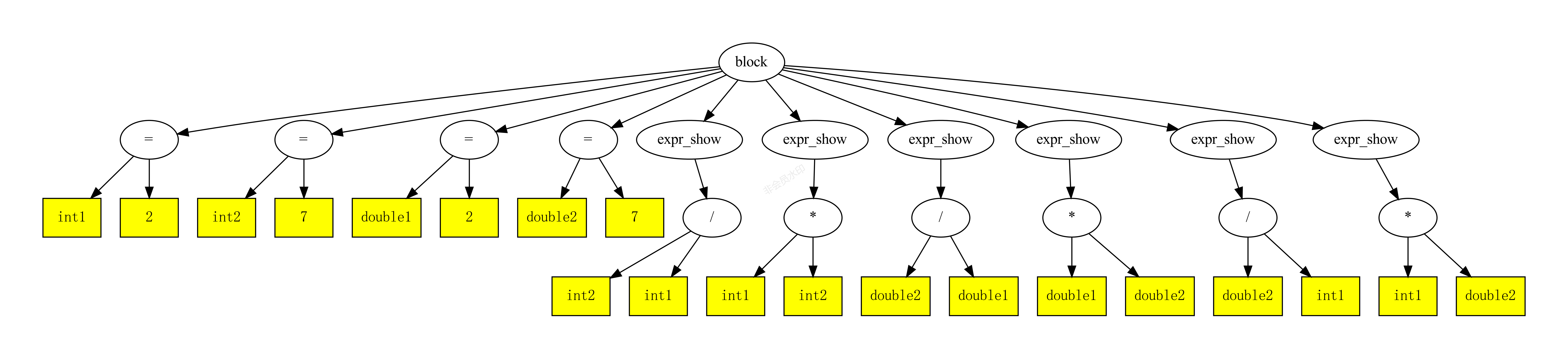
double1 \* double2

double2 /int1

int1 \* double2



显然，我们可以分析7 / 2=3，7.0 / 2 = 3.5，得到的计算结果与理论值一致，同时，计算的结果也与理论值一致， 计算结果正确可靠。下图为对应的抽象语法树：



生成的语法树与测试程序一致，通过测试。

6实验总结

6.1调试和问题修改总结

问题：在进行用科学计数法表示的实数的匹配定义时，出现了问题：形如3e5的串无法被识别，而必须用3e+5来识别，这显然不满足我们对用科学计数法表示的实数的要求。

解决办法：在calculator.l中对于用科学计数法表示的实数的正则式原有匹配规则之后，又添加了以下匹配规则：  
 else{

for (j = 0; i < n; j++, i++)

exp[j] = yytext[i];

int exponent = atof(exp);

for (j = 0; j < exponent; j++)

result = result \* 10;

}

这样便直接默认了指数部分没有符号时，默认指数部分为正数。

其余还有一些小问题都是由于对项目结构和代码逻辑不熟悉导致，经过慢慢调试之后便一一解决了。

6.2实验小结

在本次实验中，最难解决的部分是对项目结构和代码逻辑不熟悉，多亏在实验中参考了《Flex & Bison》一书，里面也有类似的强化计算器的设计内容，给了我很大的指导作用。本次实验让我对flex和bison有了一定的了解：

Flex:

Flex是一个生成词法分析器的工具。它根据用户提供的正则表达式规则，生成C/C++代码，用于将输入文本流划分为一系列的记号（Tokens）。Flex能够高效地处理大量的输入，并生成高性能的词法分析器。词法分析器负责识别和提取源代码中的关键字、标识符、运算符和常量等词法单元。

Bison:

Bison是一个生成语法分析器的工具。它根据用户提供的上下文无关文法规则，生成C/C++代码，用于分析和解析输入的记号流。Bison使用LALR(1)语法分析算法，可以处理大部分上下文无关文法，并生成解析树或抽象语法树。语法分析器负责根据语法规则检查输入的记号流是否符合语法规范，并构建语法结构，如抽象语法树，用于后续的语义分析和代码生成。

Flex & Bison:

使用Flex编写词法分析器规则文件（通常以.l或.lex为后缀），其中定义了正则表达式模式和对应的动作。

使用Bison编写语法分析器规则文件（通常以.y或.yy为后缀），其中定义了上下文无关文法规则和对应的语义动作。

使用Flex将词法分析器规则文件生成C/C++代码，编译生成词法分析器。

使用Bison将语法分析器规则文件生成C/C++代码，编译生成语法分析器。

将词法分析器和语法分析器结合起来，用于解析和处理源代码。

7实验建议

建议将实验与理论课结合得更加密切一点，总感觉理论课与实验课有些脱节。具体来说，可以将实验课与理论课一体化，将实验内容更加细化，比如将实验课与理论课同时开课，每周总课时固定，再由老师根据两者协同进度自由调整理论与实践课程的时间，做到让实践贯穿理论。