编译器专题实验报告

**实验三：语法分析（独立模式）**

**一、实验目的**

掌握移进-归约技术语法分析技术，利用语法分析器生成工具Yacc/Bison 实现语法分析器的构造。

利用语法分析器生成工具 Yacc/Bison编写一个语法分析程序，结合词法分析器输出内容，能够根据语言的上下文无关文法识别输入的单词序列是否文法的句子；以LALR为基础，产生分析表。

**二、实验一（必做）**

**1.实验要求**

完成计算器编写：

（1）支持加减乘除、括号、“-”、关系运算等exper操作；

（2）考虑小数

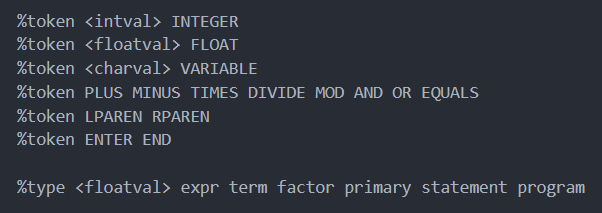
（3）考虑算符结合和优先级；

（4）计算器有26个寄存器(分别用a到z命名)。可保存数值。

（5）如果表达式为赋值，则仅保存数值到相应寄存器，否则打印表达式的值。

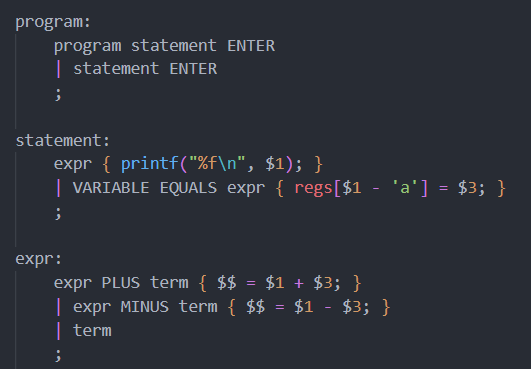
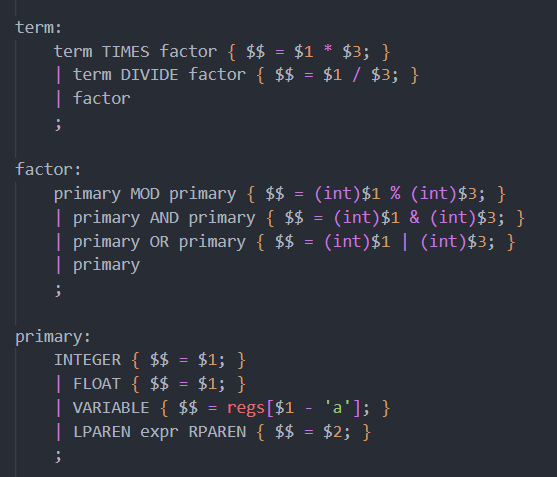
**2.使用YACC/Bison实现计算器**

**（1）Token定义**



为实现计算器运算，首先要定义一些Token用于表示识别到整数（INTEGER）、浮点数（FLOAT）、以及要求的26个变量寄存器（VARIABLE），同时还要能识别加减乘除、余数运算、括号等，具体如上图所示。

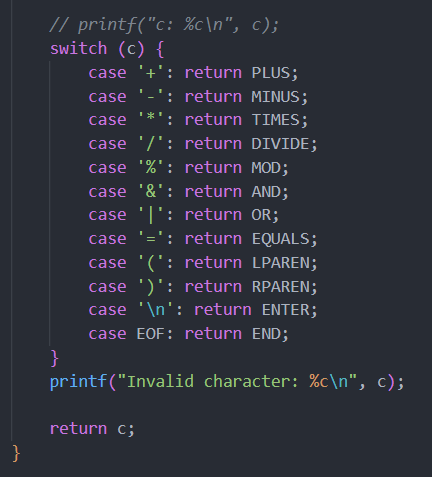
**（2）规则定义**

语法分析使用的规则如上所示，自下而上地来看，先从primary开始，首先识别数字和变量寄存器以及有括号的部分，而后识别余数运算、与或运算（factor部分），再然后识别乘除运算（term部分），最后结合加减运算（expr部分），同时，如果是类似于a=1这类为变量寄存器赋值的操作，最后会在statement部分被识别，如果不是为变量寄存器赋值，那么会在statement部分进行结果的输出。由此也可以看出，括号运算的优先级最高，其次是余数运算和与或运算，再是乘除和加减。

**（3）词素识别编写**

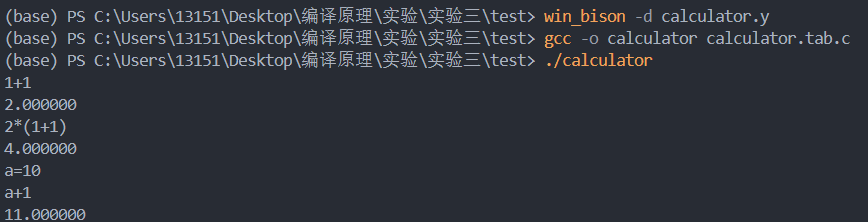
实验一仅使用YACC/Bison进行计算器的编写，因此需要编写yylex函数进行词素的识别，实验三中，将结合lex进行计算器的编写，那里就不需要在YACC/Bison文件中进行yylex函数的编写了。yylex的实现如下所示。

代码会从命令提示符中读入需要的字符，如果遇到不需要的空格或者回车就会持续读入且不做处理。当读入到的字符为数字或者小数点时，则会直接读入需要的数字，如果发现读入的不是数字，则会报错。如果发现读入的为字母，则将其作为变量寄存器的代号（a到z）并将其识别为变量。如果发现读入运算符，则将其识别为对应的Token。

**（4）实验验证**

在命令行中输入win\_bison -d calculator.y，可以得到上面编写的计算器文件生成的C语言文件calculator.tab.c和对应的头文件，而后运行对应的C语言文件就可以运行计算器，如下所示。

****

对计算器输出1+1可以的到结果2，输入2\*（1+1）可以的到结果4，同时也可以使用变量寄存器进行赋值，比如将a赋为10，输入a+1就可以11。

**三、实验二（必做）**

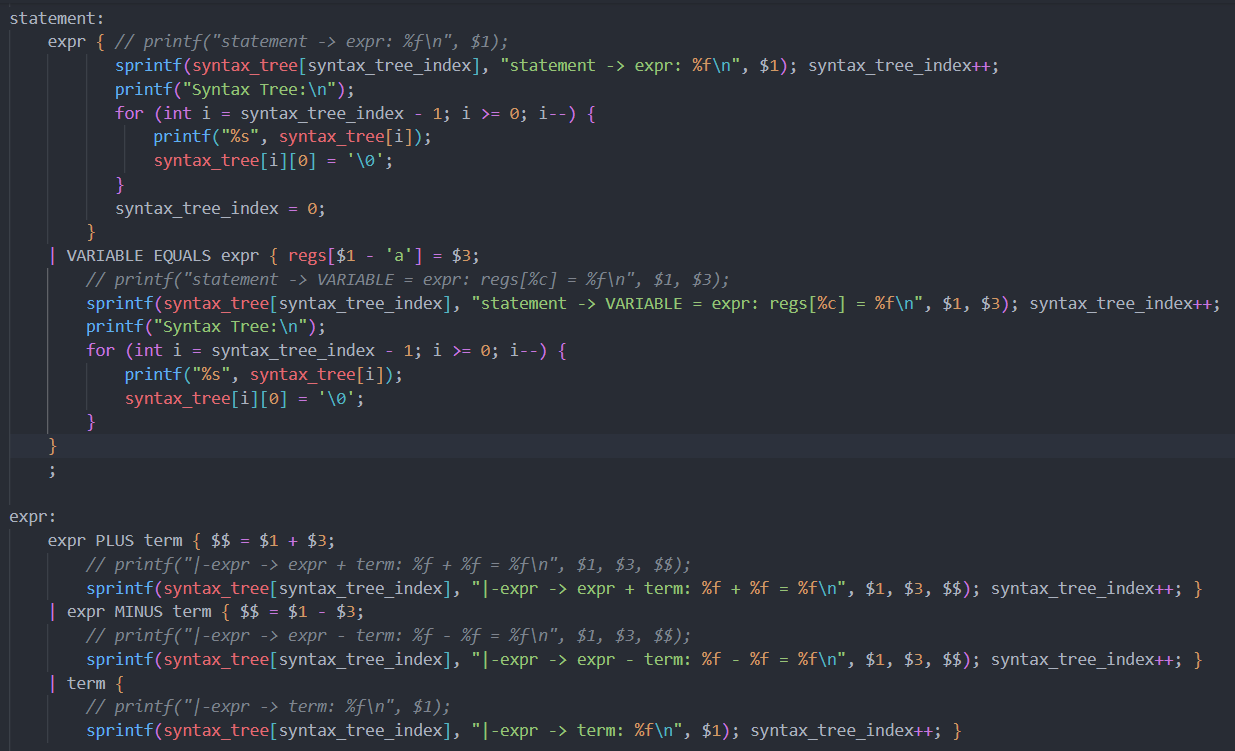
**1.实验要求**

在上一例基础上，完成计算器的语法分析，识别出终结符，考虑出错情况，包括识别运算符中的字母等不能运算的内容。

**2.计算器的编写**

**（1）规则修改**

上面的计算器仅完成了计算功能的实现，并没有体现出对应的语法分析功能，于是在这里，我们对计算器的规则进行修改，使其能够在运算的同时保存语法树并最后输出语法树。修改后的statement和expr规则如下所示，其他规则修改类似。

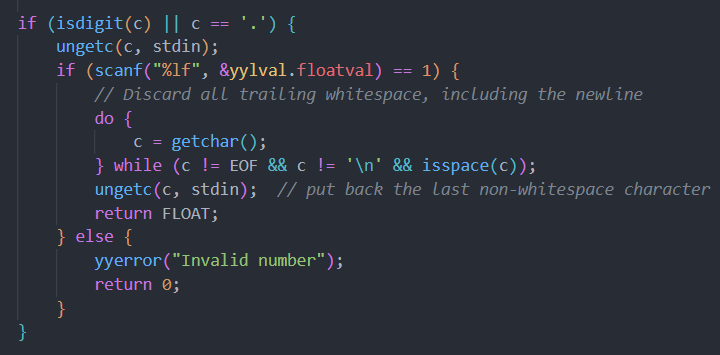


首先，我们在代码中添加了syntax\_tree数组，每进行一次语法分析就存储下使用的规则。可以看到对于statement->expr项，由于自下而上分析到分析到statement时已经完成整体分析，于是在这里将存储到的语法树进行输出，对于其他规则，则进行语法树的存储。

**（2）错误识别**

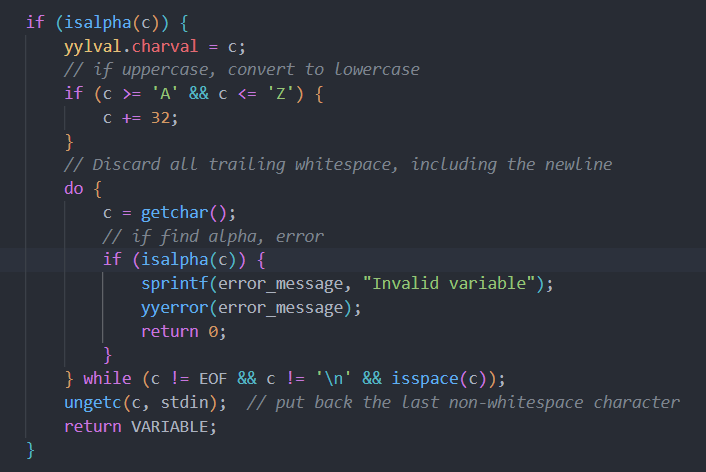
这里考虑了三种错误情况的处理，首先是识别到了错误的数字，而后是识别到了错误的变量寄存器名（这里只能用a到z字母），最后是识别到了非法的运算符。代码中对yylex进行修改以实现以上功能。

**① 错误数字识别**



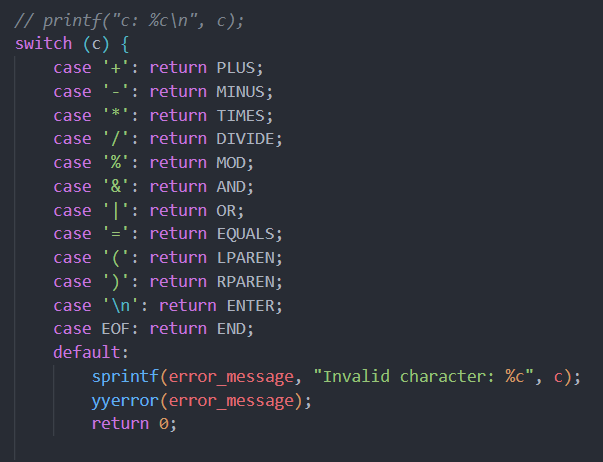
这里使用scanf函数实现，如果发现应当识别数字的时候，识别的不是所需要的数字，则返回0，抛出Invalid number的错误。

**② 错误变量识别**



只允许接受a到z的单个字母，如果发现后面跟着字母，则判定为错误的变量名，抛出Invalid variable的错误。

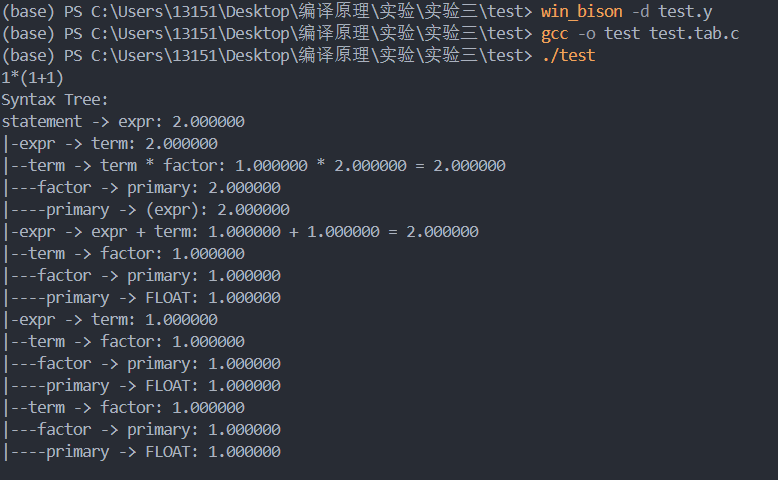
**③ 非法运算符识别**



计算器只识别之前定义的Token，所以当识别到其他的Token时会认为读到了非法字符，抛出Invalid character的异常。

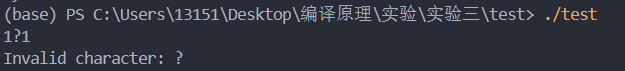
**（3）实验验证**

在命令行中输入win\_bison -d test.y，可以得到上面编写的计算器文件生成的C语言文件test.tab.c和对应的头文件，而后运行对应的C语言文件就可以运行计算器，如下所示。

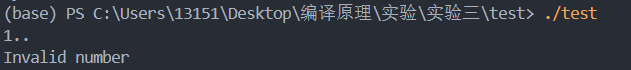


可以看到，当输入1\*（1+1）时，可以得到语法分析的语法树，可以看到，从statement->expr出发，expr->term->term \* factor->term \* primary

->term \* (expr)，这里的expr又进行转换，expr->expr + term，又利用expr -> term， term -> factor 和 factor -> primary ，expr + term 变为 primary + primary，最终识别到FLOAT + FLOAT （也就是1+1），term \* (expr)中的term也类似地识别到输入的第一个1。这里在阐述语法树的时候用的是自顶向下的思路，实际上Bison在分析的时候是自底向上进行分析的。







当输入不正确的符号、变量名或者数字时，程序会进行报错处理，比如输入？时会出现无效的字符，输入aa时会报错无效的变量名等。

**四、实验三（选做）**

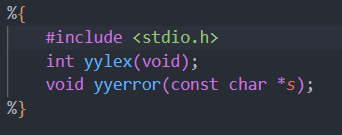
**1.实验要求**

和词法分析结合，实现flex和bison的联合编程。

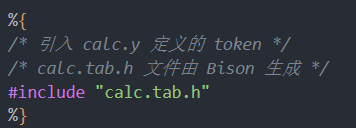
**2.实现思路**

上面的计算器程序中通过编写yylex进行词法Token的识别，但是我们也可以使用之前用过的flex进行词法识别，我们可以将其与Bison结合，通过批处理文件对.y文件和词法的.l进行处理后，得到计算器运行文件。

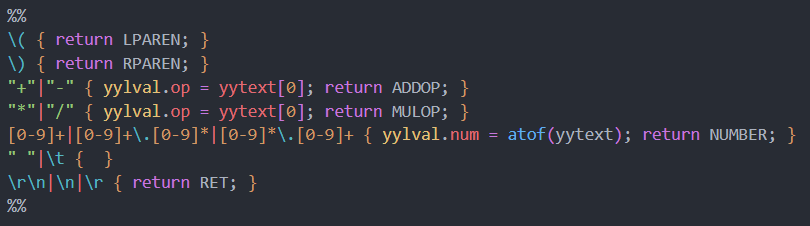
首先，在calc.y文件中，我们声明int yylex(void)函数，如下所示。



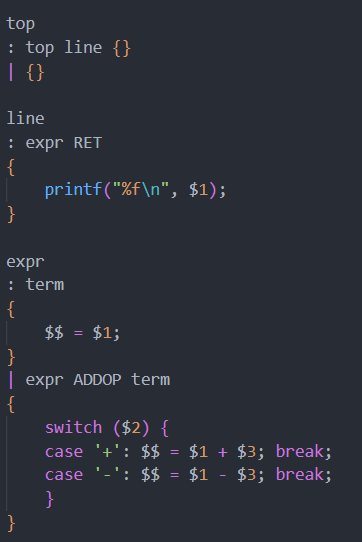
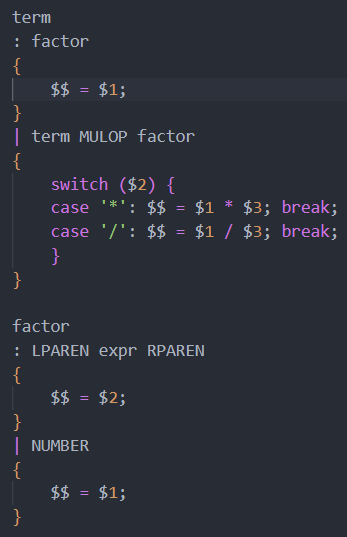
在calc.l文件中，我们需要引用calc.tab.h的头文件，如下所示。



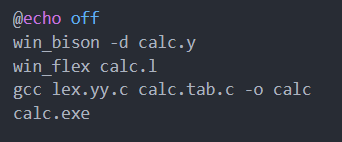
同时，我们需要定义一些需要用到的数字以及运算符的Token，如下所示。



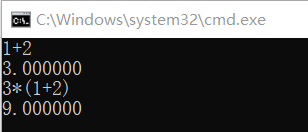
对应地，编写需要的语法分析规则，如下所示。

最后，我们将使用win flex和win bison处理上述.y和.l文件以及编译运行C语言文件的过程综合到一个批处理文件中，如下所示。



运行批处理文件，即可启动计算器，计算器测试如下所示，可以看到计算器可以进行正常的运行。

****

这里使用批处理文件的生成时间作为此实验完成的时间，用于参考。

****

**五、遇到的问题**

1.不会编写批处理脚本，参考CSDN链接

https://blog.csdn.net/hfy1237/article/details/130123285

**六、附件：代码**

calculator.y

%{

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

double regs[26];

int base;

int yylex(void);

void yyerror(char \*);

%}

%union {

int intval;

double floatval;

char charval;

}

%token <intval> INTEGER

%token <floatval> FLOAT

%token <charval> VARIABLE

%token PLUS MINUS TIMES DIVIDE MOD AND OR EQUALS

%token LPAREN RPAREN

%token ENTER END

%type <floatval> expr term factor primary statement program

%%

program:

program statement ENTER

| statement ENTER

;

statement:

expr { printf("%f\n", $1); }

| VARIABLE EQUALS expr { regs[$1 - 'a'] = $3; }

;

expr:

expr PLUS term { $$ = $1 + $3; }

| expr MINUS term { $$ = $1 - $3; }

| term

;

term:

term TIMES factor { $$ = $1 \* $3; }

| term DIVIDE factor { $$ = $1 / $3; }

| factor

;

factor:

primary MOD primary { $$ = (int)$1 % (int)$3; }

| primary AND primary { $$ = (int)$1 & (int)$3; }

| primary OR primary { $$ = (int)$1 | (int)$3; }

| primary

;

primary:

INTEGER { $$ = $1; }

| FLOAT { $$ = $1; }

| VARIABLE { $$ = regs[$1 - 'a']; }

| LPAREN expr RPAREN { $$ = $2; }

;

%%

void yyerror(char \*s) {

fprintf(stderr, "%s\n", s);

}

int main(void) {

yyparse();

return 0;

}

int yylex(void) {

int c = getchar();

while (c != EOF && c != '\n' && isspace(c))

c = getchar();

if (isdigit(c) || c == '.') {

ungetc(c, stdin);

if (scanf("%lf", &yylval.floatval) == 1) {

// Discard all trailing whitespace, including the newline

do {

c = getchar();

} while (c != EOF && c != '\n' && isspace(c));

ungetc(c, stdin); // put back the last non-whitespace character

return FLOAT;

} else {

yyerror("Invalid number");

return 0;

}

}

if (isalpha(c)) {

yylval.charval = c;

// Discard all trailing whitespace, including the newline

do {

c = getchar();

} while (c != EOF && c != '\n' && isspace(c));

ungetc(c, stdin); // put back the last non-whitespace character

return VARIABLE;

}

// printf("c: %c\n", c);

switch (c) {

case '+': return PLUS;

case '-': return MINUS;

case '\*': return TIMES;

case '/': return DIVIDE;

case '%': return MOD;

case '&': return AND;

case '|': return OR;

case '=': return EQUALS;

case '(': return LPAREN;

case ')': return RPAREN;

case '\n': return ENTER;

case EOF: return END;

}

printf("Invalid character: %c\n", c);

return c;

}

test.y

%{

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

double regs[26];

int base;

char error\_message[50];

char syntax\_tree[1000][1000];

int syntax\_tree\_index = 0;

int yylex(void);

void yyerror(char \*);

%}

%union {

int intval;

double floatval;

char charval;

}

%token <intval> INTEGER

%token <floatval> FLOAT

%token <charval> VARIABLE

%token PLUS MINUS TIMES DIVIDE MOD AND OR EQUALS

%token LPAREN RPAREN

%token ENTER END

%type <floatval> expr term factor primary statement program

%%

program:

program statement ENTER

| statement ENTER

;

statement:

expr { // printf("statement -> expr: %f\n", $1);

sprintf(syntax\_tree[syntax\_tree\_index], "statement -> expr: %f\n", $1); syntax\_tree\_index++;

printf("Syntax Tree:\n");

for (int i = syntax\_tree\_index - 1; i >= 0; i--) {

printf("%s", syntax\_tree[i]);

syntax\_tree[i][0] = '\0';

}

syntax\_tree\_index = 0;

}

| VARIABLE EQUALS expr { regs[$1 - 'a'] = $3;

// printf("statement -> VARIABLE = expr: regs[%c] = %f\n", $1, $3);

sprintf(syntax\_tree[syntax\_tree\_index], "statement -> VARIABLE = expr: regs[%c] = %f\n", $1, $3); syntax\_tree\_index++;

printf("Syntax Tree:\n");

for (int i = syntax\_tree\_index - 1; i >= 0; i--) {

printf("%s", syntax\_tree[i]);

syntax\_tree[i][0] = '\0';

}

}

;

expr:

expr PLUS term { $$ = $1 + $3;

// printf("|-expr -> expr + term: %f + %f = %f\n", $1, $3, $$);

sprintf(syntax\_tree[syntax\_tree\_index], "|-expr -> expr + term: %f + %f = %f\n", $1, $3, $$); syntax\_tree\_index++; }

| expr MINUS term { $$ = $1 - $3;

// printf("|-expr -> expr - term: %f - %f = %f\n", $1, $3, $$);

sprintf(syntax\_tree[syntax\_tree\_index], "|-expr -> expr - term: %f - %f = %f\n", $1, $3, $$); syntax\_tree\_index++; }

| term {

// printf("|-expr -> term: %f\n", $1);

sprintf(syntax\_tree[syntax\_tree\_index], "|-expr -> term: %f\n", $1); syntax\_tree\_index++; }

;

term:

term TIMES factor { $$ = $1 \* $3;

// printf("|--term -> term \* factor: %f \* %f = %f\n", $1, $3, $$);

sprintf(syntax\_tree[syntax\_tree\_index], "|--term -> term \* factor: %f \* %f = %f\n", $1, $3, $$); syntax\_tree\_index++; }

| term DIVIDE factor { $$ = $1 / $3;

// printf("|--term -> term / factor: %f / %f = %f\n", $1, $3, $$);

sprintf(syntax\_tree[syntax\_tree\_index], "|--term -> term / factor: %f / %f = %f\n", $1, $3, $$); syntax\_tree\_index++; }

| factor {

// printf("|--term -> factor: %f\n", $1);

sprintf(syntax\_tree[syntax\_tree\_index], "|--term -> factor: %f\n", $1); syntax\_tree\_index++; }

;

factor:

primary MOD primary { $$ = (int)$1 % (int)$3;

// printf("|---factor -> primary %% primary: %d %% %d = %d\n", (int)$1, (int)$3, (int)$$);

sprintf(syntax\_tree[syntax\_tree\_index], "|---factor -> primary %% primary: %d %% %d = %d\n", (int)$1, (int)$3, (int)$$); syntax\_tree\_index++; }

| primary AND primary { $$ = (int)$1 & (int)$3;

// printf("|---factor -> primary & primary: %d && %d = %d\n", (int)$1, (int)$3, (int)$$);

sprintf(syntax\_tree[syntax\_tree\_index], "|---factor -> primary & primary: %d && %d = %d\n", (int)$1, (int)$3, (int)$$); syntax\_tree\_index++; }

| primary OR primary { $$ = (int)$1 | (int)$3;

// printf("|---factor -> primary | primary: %d | %d = %d\n", (int)$1, (int)$3, (int)$$);

sprintf(syntax\_tree[syntax\_tree\_index], "|---factor -> primary | primary: %d | %d = %d\n", (int)$1, (int)$3, (int)$$); syntax\_tree\_index++; }

| primary {

// printf("|---factor -> primary: %f\n", $1);

sprintf(syntax\_tree[syntax\_tree\_index], "|---factor -> primary: %f\n", $1); syntax\_tree\_index++; }

;

primary:

INTEGER { $$ = $1; // printf("|----primary -> INTEGER: %f\n", $1);

sprintf(syntax\_tree[syntax\_tree\_index], "|----primary -> INTEGER: %f\n", $1); syntax\_tree\_index++; }

| FLOAT { $$ = $1; // printf("|----primary -> FLOAT: %f\n", $1);

sprintf(syntax\_tree[syntax\_tree\_index], "|----primary -> FLOAT: %f\n", $1); syntax\_tree\_index++; }

| VARIABLE { $$ = regs[$1 - 'a'];

// printf("|----primary -> VARIABLE: %c = %f\n", $1, $$);

sprintf(syntax\_tree[syntax\_tree\_index], "|----primary -> VARIABLE: %c = %f\n", $1, $$); syntax\_tree\_index++; }

| LPAREN expr RPAREN { $$ = $2;

// printf("|----primary -> (expr): %f\n", $2);

sprintf(syntax\_tree[syntax\_tree\_index], "|----primary -> (expr): %f\n", $2); syntax\_tree\_index++; }

;

%%

void yyerror(char \*s) {

fprintf(stderr, "%s\n", s);

}

int main(void) {

yyparse();

return 0;

}

int yylex(void) {

int c = getchar();

while (c != EOF && c != '\n' && isspace(c))

c = getchar();

if (isdigit(c) || c == '.') {

ungetc(c, stdin);

if (scanf("%lf", &yylval.floatval) == 1) {

// Discard all trailing whitespace, including the newline

do {

c = getchar();

} while (c != EOF && c != '\n' && isspace(c));

ungetc(c, stdin); // put back the last non-whitespace character

return FLOAT;

} else {

yyerror("Invalid number");

return 0;

}

}

if (isalpha(c)) {

yylval.charval = c;

// if uppercase, convert to lowercase

if (c >= 'A' && c <= 'Z') {

c += 32;

}

// Discard all trailing whitespace, including the newline

do {

c = getchar();

// if find alpha, error

if (isalpha(c)) {

sprintf(error\_message, "Invalid variable");

yyerror(error\_message);

return 0;

}

} while (c != EOF && c != '\n' && isspace(c));

ungetc(c, stdin); // put back the last non-whitespace character

return VARIABLE;

}

// printf("c: %c\n", c);

switch (c) {

case '+': return PLUS;

case '-': return MINUS;

case '\*': return TIMES;

case '/': return DIVIDE;

case '%': return MOD;

case '&': return AND;

case '|': return OR;

case '=': return EQUALS;

case '(': return LPAREN;

case ')': return RPAREN;

case '\n': return ENTER;

case EOF: return END;

default:

sprintf(error\_message, "Invalid character: %c", c);

yyerror(error\_message);

return 0;

}

printf("Invalid character: %c\n", c);

return c;

}

calc.y

%{

#include <stdio.h>

int yylex(void);

void yyerror(const char \*s);

%}

%token RET

%token <num> NUMBER

%token <op> ADDOP MULOP LPAREN RPAREN

%type <num> top line expr term factor

%start top

%union {

char op;

double num;

}

%%

top

: top line {}

| {}

line

: expr RET

{

printf("%f\n", $1);

}

expr

: term

{

$$ = $1;

}

| expr ADDOP term

{

switch ($2) {

case '+': $$ = $1 + $3; break;

case '-': $$ = $1 - $3; break;

}

}

term

: factor

{

$$ = $1;

}

| term MULOP factor

{

switch ($2) {

case '\*': $$ = $1 \* $3; break;

case '/': $$ = $1 / $3; break; // 这里会出什么问题？

}

}

factor

: LPAREN expr RPAREN

{

$$ = $2;

}

| NUMBER

{

$$ = $1;

}

%%

void yyerror(const char \*s)

{

fprintf(stderr, "%s\n", s);

}

int main()

{

yyparse();

return 0;

}

calc.l

/\* calc.l \*/

%option noyywrap

%{

/\* 引入 calc.y 定义的 token \*/

/\* calc.tab.h 文件由 Bison 生成 \*/

#include "calc.tab.h"

%}

/\* 规则部分 yylval 同样来自 calc.tab.h 文件，其类型为 YYSTYPE，用于 token 的相关属性，比如 NUMBER 对应的实际数值 \*/

/\* 在这个例子中，YYSTYPE 定义如下

typedef union YYSTYPE {

char op;

double num;

} YYSTYPE;

其同样由 .y 文件根据 %union 生成，在文件中我们的 %union 定义如下

%union {

char op;

double num;

}

\*/

%%

\( { return LPAREN; }

\) { return RPAREN; }

"+"|"-" { yylval.op = yytext[0]; return ADDOP; }

"\*"|"/" { yylval.op = yytext[0]; return MULOP; }

[0-9]+|[0-9]+\.[0-9]\*|[0-9]\*\.[0-9]+ { yylval.num = atof(yytext); return NUMBER; }

" "|\t { }

\r\n|\n|\r { return RET; }

%%

test.bat

@echo off

win\_bison -d calc.y

win\_flex calc.l

gcc lex.yy.c calc.tab.c -o calc

calc.exe