華南師範大學

本科学生实验（实践）报告

院　　系：计 算 机 学 院

实验课程：编译原理

实验项目：TINY 扩充语言的语法分析

指导老师：黄煜廉

专　　业：计算机科学与技术

班　　级：2019 级 4 班

学　　生：欧阳盈盈

学　　号：20183602151

实验四：TINY 扩充语言的语法分析

一、 实验内容

1、 扩充的语法规则有：实现 do while 循环，for 循环，扩充算术表达式的运算符号：-= 减法赋值运算符号（类似于 C 语言的-=）、求余%、乘方^

2、 扩充比较运算符号：==（等于），>(大于)、<=(小于等于)、>=(大于等于)、<>(不等于)等运算符号

3、 新增支持正则表达式以及用于 repeat 循环、do while 循环、if 条件语句作条件判断的逻辑表达式：运算符号有 and（与）、 or（或）、 not（非）

4、 具体文法规则自行构造

可参考：云盘中参考书 P97 及 P136 的文法规则。

a) Dowhile-stmt-->do stmt-sequence while(exp);
b) for-stmt-->for identifier:=simple-exp to simple-exp do stmt-sequence enddo 步长递增 1
c) for-stmt-->for identifier:=simple-exp downto simple-exp do stmt-sequence enddo 步长递减 1
d) -= 减法赋值运算符号、求余%、乘方^、>=(大于等于)、<=(小于等于)、>(大于)、<>(不等于)运算符号的文法规则请自行组织。
e) 把 tiny 原来的赋值运算符号(:=)改为(=),而等于的比较符号符号（=）则改为（==）
f) 为 tiny 语言增加一种新的表达式——正则表达式,其支持的运算符号有或(|) 、连接(&)、闭包(#)、括号( ) 以及基本正则表达式 。
g) 为 tiny 语言增加一种新的语句，ID:=正则表达式
h) 为 tiny 语言增加一种新的表达式——逻辑表达式,其支持的运算符号有 and(与) 、or (或)、非(not)。
i) 为了实现以上的扩充或改写功能,还需要对原 tiny 语言的文法规则做好相应的改造处理。

二、 实验要求

1、 要提供一个源程序编辑界面，以让用户输入源程序（可保存、打开源程序）
2、 可由用户选择是否生成语法树，并可查看所生成的语法树
3、 应该书写完善的软件文档
4、 要求应用程序应为 Windows 界面

三、 软件应用说明

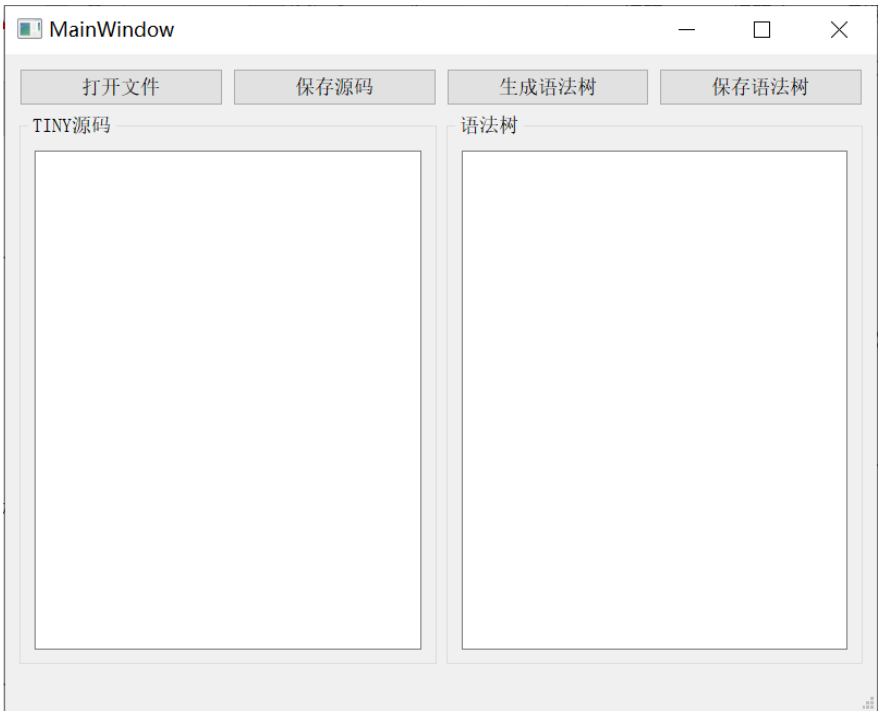
1、 开发环境

采用 c++作为开发语言，并使用 Qt 进行图形界面设计，开发工具为 Qt5.14.2

2、 使用说明

用户可选择【打开文件】/直接在左侧文本框输入源程序，并点击【生成语法树】即可在右侧文本框查看结果，也可保存生成的语法树。

界面如下：



四、 最终文法

Program→stmt-sequence

stmt-sequence→statement{;statement}

statement→if-stmt|repeat-stmt|assign-stmt|read-stmt|write-stmt|dowhile-stmt|for-stmt

If-stmt→if logic-exp then stmt-sequence [else stmt-sequence] end

dowhile-stmt→do stmt-sequence while logic-exp

for-stmt→for assign-stm to simple-exp do stmt-sequence enddo

for-stmt→for assign-stm downto simple-exp do stmt-sequence enddo

logic-exp→logic-term { or logic-term }

logic-term→logic-factor { and logic-factor }

logic-factor→not logic-factor | exp

repeat-stmt→repeat stmt-sequence until logic-exp

assign-stmt→identifier = exp| identifier -= exp | identifier := re-exp

read-stmt→read identifier

write-stmt→write exp

exp→simple-exp[comparision-op simple-exp]

comparision-op →< | > | == | >= | <= | <>

simple-exp→term{addop term}

addop→+ | -

term→power-term {mulop power-term }

mulop→* | / | %

power-term→factor{power factor}

factor→(exp) | number | identfier

re-exp→re-termA { | re-termA }

re-termA→re-termB { & re-termB }

re-termB→re-factor[#]

re-factor→(re-exp) | letter

五、 实验过程

按照实验要求，这里逐一介绍要扩充的语法规则。

（一） 统一处理

1、 在 globlas.h 文件中

a) 修改最大保留字数量#define MAXRESERVED 18

b) 在 typedef enum{} TokenType 结构中增加保留字 WHILE,DO,TO,DOWNTO, FOR, ENDDO, ENDWHILE, MOD；增加特殊符号 MinusEQUAL, MOD, POWER, LESSEQUAL, NEQ, MT, MOREEQUAL, AND, OR, NOT,REAND, REOR, RENOT, RECLO,REASSIGN, LETTER

c) 在 typedef enum {} StmtKind 结构中增加语句类型 DoWhileK, ForK.

2、 在 scan.c 文件中

在 static struct {} reservedWords[MAXRESERVED]中增加保留字关联字符 {"while", WHILE}, {"do", DO}, {"for", FOR}, {"endwhile", ENDWHILE}, {"to", TO}, {"enddo", ENDDO}, {"downto", DOWNTO}；新增特殊字符{"and", AND}, {"or", OR}, {"not", NOT}

（二） do while 循环

文法规则：Dowhile-stmt→do stmt-sequence while(exp)

将“while”加入 stmt-sequence 文法的 follow 集

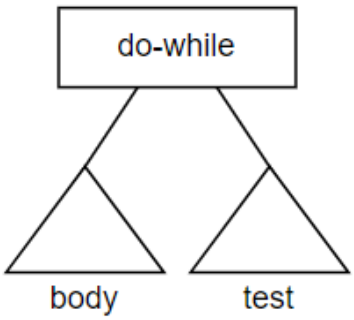


图 1 dowhile 循环的语法树结构

➢ 在 parse.c 文件中

a) 增加静态成员函数定义：

```
static TreeNode * dowhile_stmt(void);
```

代码如下：

```
/* 扩充dowhile文法 */
TreeNode * dowhile_stmt(void)
{
    TreeNode * t = newStmtNode(DowhileK);
    match(DO);
    if(t != NULL)
        t->child[0] = stmt_sequence();
    match(WHILE);
    match(LPAREN);
    if(t != NULL)
        t->child[1] = exp();
    match(RPAREN);
    return t;
}
```

b) 在 TreeNode *stmt_sequence(void)函数中增加 stmt_sequence 文法的 follow 集：增加“while”

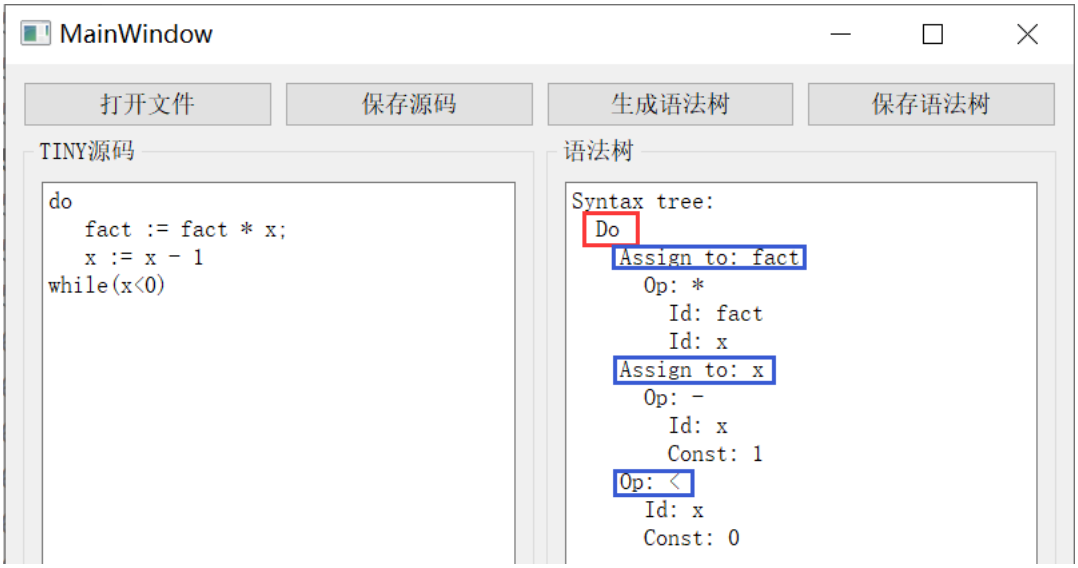
增加后为：

```
while ((token!=ENDFILE) && (token!=END) &&
       (token!=ELSE) && (token!=UNTIL) && (token!=WHILE))
```

c) 在 TreeNode * statement(void)函数中 switch (token)模块下增加状态

```
case DO: t = dowhile_stmt(); break;
```

d） 测试结果



（三） for 循环

文法规则

（1） for-stmt→for identifier:=simple-exp to simple-exp do stmt-sequence enddo 步长递增 1

（2） for-stmt→for identifier:=simple-exp downto simple-exp do stmt-sequence enddo 步长递减 1

将“enddo”加入 stmt-sequence 文法的 follow 集

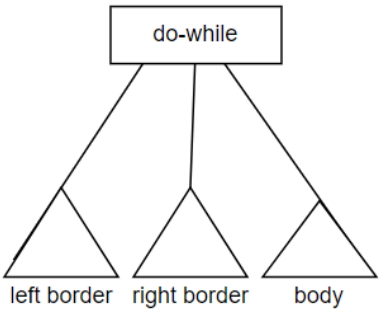


图 2 for 循环的语法树结构

➢ 在 parse.c 文件中

a） 增加静态成员函数定义：

```
static TreeNode * for_stmt(void);
```

代码如下：

```
/* 扩充for文法 */
TreeNode * for_stmt(void)
{
    TreeNode * t = newStmtNode(ForK);
    match(FOR);
    if(t != NULL)
        t->child[0] = assign_stmt();
    if(token == TO)
    {
        match(TO);
        if(t != NULL)
            t->child[1] = simple_exp();
    }
    else
    {
        match(DOWNTO);
        if(t != NULL)
            t->child[1] = simple_exp();
    }
    match(DO);
    if(t != NULL)
        t->child[2] = stmt_sequence();
    match(ENDDO);
    return t;
}
```

b) 在 TreeNode *stmt_sequence(void)函数中增加 stmt_sequence 文法的follow集：增加“enddo”

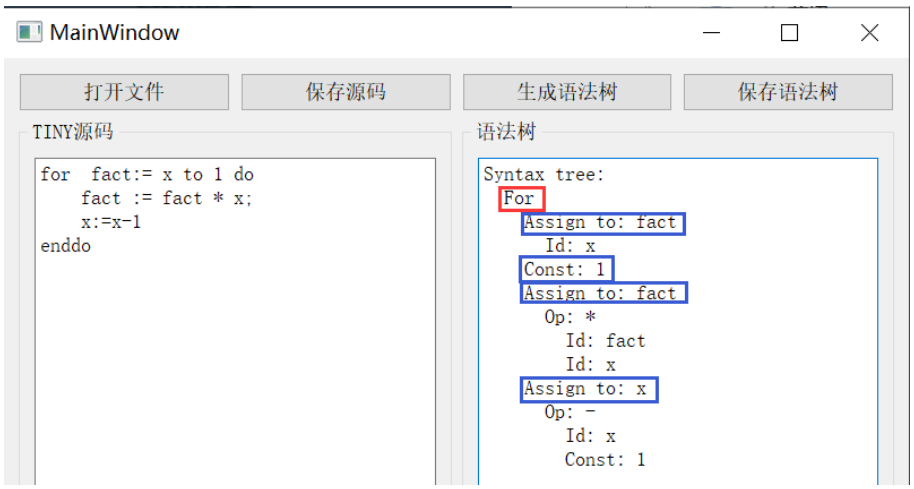
增加后为：

```
while ((token!=ENDFILE) && (token!=END) && (token!=ELSE)
       && (token!=UNTIL) && (token!=WHILE) && (token!=ENDDO))
```

c) 在TreeNode * statement(void)函数中 switch (token)模块下增加状态

```
case FOR: t = for_stmt(); break;
```

e) 测试结果

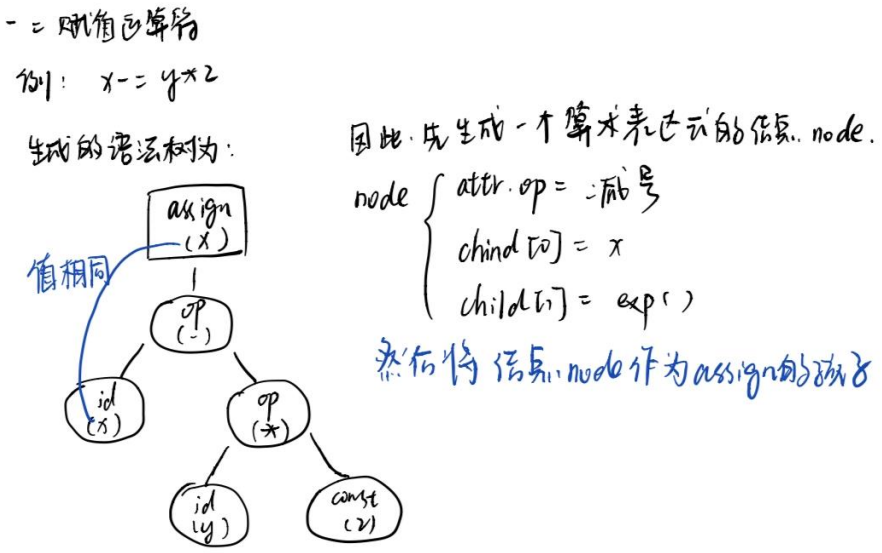


（四） -= 减法赋值运算符号

文法规则：扩充 tiny 原有的 assign-stmt，即

assign-stmt→identifier:=exp|identifier-=exp

➢ 分析过程



➢ 在 scan.cpp 中

在 TokenType getToken(void)函数中，增加扫描到‘-’时的处理：如果当前扫描到的是‘-’，使用 getNextChar()函数获取下一字符 c，如果 c 为‘=’，此时“-=”构成减法赋值运算符；否则使用 getNextChar()函数回退一个字符，此时为‘-’运算符。

代码如下：

```
case '-':
    //增加 -=
    c = getNextChar();
    if(c == '=')
        currentToken = MinusEQUAL;
    else
    {
        ungetNextChar();
        currentToken = MINUS;
    }
    break;
```

➢ 在 parse.c 文件中

在 TreeNode * assign_stmt(void)函数中，增加处理当前 token 为 MinusEQUAL（即-=）的情况：match(MinusEQUAL)后，先生成一个运算符为减号的树 p，树的左孩子为待赋值的变量，右孩子为算术表达式。然后将树 p 作为

assign 树 t 的孩子。

代码如下：

```
TreeNode * assign_stmt(void)
{
    TreeNode * t = newStmtNode(AssignK);
    if ((t != NULL) && (token == ID))
        t->attr.name = copyString(tokenString);
    //生成值为token的节点，-=时用到
    TreeNode * left = newExpNode(IdK);
    left->attr.name = copyString(tokenString);

    match(ID);
    if(token == ASSIGN)  //处理=
    {
        match(ASSIGN);
        if (t != NULL)
            t->child[0] = exp();
    }
    if(token == MinusEQUAL) //处理 -=
    {
        match(MinusEQUAL);
        //生成一个算术表达式节点
        TreeNode * p = newExpNode(OpK);
        if (p != NULL)
        {
            p->child[0] = left;
            p->attr.op = MINUS;//op为-
            p->child[1]=exp();
        }
        if(t != NULL)
            t->child[0] = p;
    }
    return t;
}
```

➢ 在 util.cpp 文件中

在 void printToken(TokenType token, const char* tokenString)函数中，增加”-=”的输出：

```
case MinusEQUAL: fprintf(listing, "-=\n"); break;
```

f） 测试结果

（五） 求%

文法规则：因为%与*和/的运算级相同，在原有的 tiny 文法上修改即可，即

mulop→*|/|%

➢ 在 scan.cpp 中

在 TokenType getToken(void)函数中，增加扫描到‘%’时的处理

```
case '%':
    currentToken = MOD;
    break;
```

➢ 在 parse.c 文件中

在 TreeNode * term(void)函数中，增加处理当前 token 为 MOD（即%）的情况。代码如下：

```
TreeNode * term(void)
{
    TreeNode * t = factor();
    while ((token == TIMES) || (token == OVER) || (token == MOD)) // * / %
    {
        TreeNode * p = newExpNode(OpK);
        if (p != NULL)
        {
            p->child[0] = t;
            p->attr.op = token;
            t = p;
            match(token);
            p->child[1] = factor();
        }
    }
    return t;
}
```

➢ 在 util.cpp 文件中

在 void printToken(TokenType token, const char* tokenString)函数中，增加%的输出：

```
case MOD: fprintf(listing, "%%\n"); break;
```

g） 测试结果

（六） 乘方^

文法规则：因为优先级顺序由高到低为^、*/、+=，乘方^的优先级最高，因此乘方^在语法树中应当更深，修改原有的 tiny 语言的文法后：

程序清单4-8 EBNF中TINY语言的文法

program → *stmt-sequence*
stmt-sequence → *statement* { **;** *statement* }
statement → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
if-stmt → **if** *exp* **then** *stmt-sequence* [**else** *stmt-sequence*] **end**
repeat-stmt → **repeat** *stmt-sequence* **until** *exp*
assign-stmt → **identifier** **:=** *exp*
read-stmt → **read** **identifier**
write-stmt → **write** *exp*
exp → *simple-exp* [*comparison-op simple-exp*]
comparison-op → **<** | **=**
simple-exp → *term* { *addop term* }
addop → **+** | **-**
term → ~~*factor*~~ { *mulop* ~~*factor*~~ }
mulop → ***** | **/**
factor → **(** *exp* **)** | **number** | **identifier**

power_term

power_term->factor { ^ factor }

➢ 在 scan.cpp 中

在 TokenType getToken(void)函数中，增加扫描到 ‘^’ 时的处理

```
case '^':
    currentToken = POWER;
    break;
```

➢ 在 parse.c 文件中

a） 增加静态成员函数定义：

```
static TreeNode * power_term(void); //增加一个优先级处理乘方
```

根据修改后的文法，修改 TreeNode * term(void)函数：

```
TreeNode * term(void)
{
    TreeNode * t = power_term();
    while ((token == TIMES) || (token == OVER) || (token == MOD)) // * / %
    {
        TreeNode * p = newExpNode(OpK);
        if (p != NULL)
        {
            p->child[0] = t;
            p->attr.op = token;
            t = p;
            match(token);
            p->child[1] = power_term();
        }
    }
    return t;
}
```

增加 treeNode * power_term(void)函数：

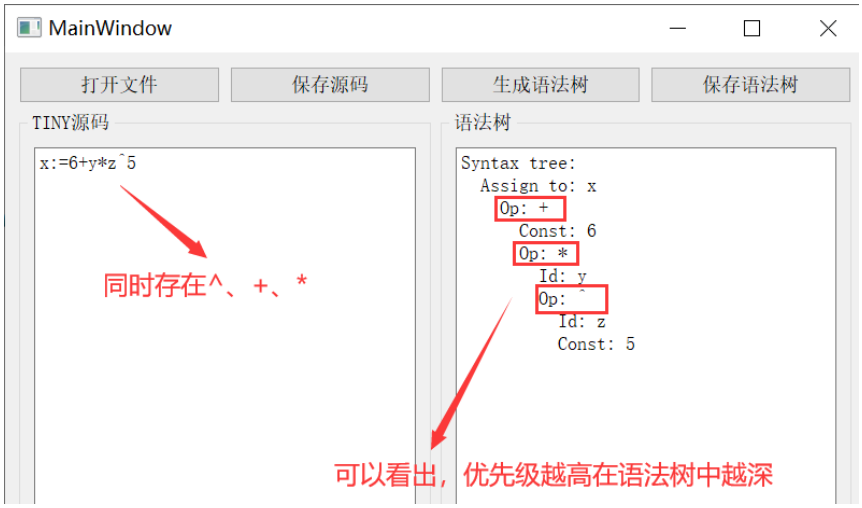
```
/* 新增处理乘方 */
treeNode * power_term(void)
{
    TreeNode * t = factor();
    while (token == POWER) // ^
    {
        TreeNode * p = newExpNode(OpK);
        if (p != NULL)
        {
            p->child[0] = t;
            p->attr.op = token;
            t = p;
            match(token);
            p->child[1] = factor();
        }
    }
    return t;
}
```

➢ 在 util.cpp 文件中

在 void printToken(TokenType token, const char* tokenString)函数中，增加^的输出：

```
case POWER: fprintf(listing, "^\n"); break;
```

h） 测试结果



（七） >=(大于等于)、<=(小于等于)、>(大于)、<>(不等于)运算符号

文法规则：因为>=、<=、>、<>与<、=的运算级相同，在原有的 tiny 文法上修改即可，即 comparison-op→<|=|<=|<>|>|>=

➢ 在 scan.cpp 中

在 TokenType getToken(void)函数中，增加扫描到‘<’时的处理：当前扫描到的是‘<’，使用 getNextChar()函数获取下一字符 c，如果 c 为‘=’，此时“<=”构成小于等于运算符；如果 c 为‘>’，此时“<>”构成不等于运算符；否则使用 getNextChar()函数回退一个字符，此时为‘<’运算符。

增加扫描到‘>’时的处理：当前扫描到的是‘>’，使用 getNextChar()函数获取下一字符 c，如果 c 为‘=’，此时“>=”构成大于等于运算符；否则使用 getNextChar()函数回退一个字符，此时为‘>’运算符。代码如下：

```
case '<':
    //增加 <= <>
    c = getNextChar();
    if(c == '=')
        currentToken = LESSEQUAL;
    else if(c == '>')
        currentToken = NEQ;
    else
    {
        ungetNextChar();
        currentToken = LT;
    }
    break;
case '>':
    //增加 > >=
    c = getNextChar();
    if(c == '=')
        currentToken = MOREEQUAL;
    else
    {
        ungetNextChar();
        currentToken = MT;
    }
    break;
```

➢ 在 parse.c 文件中

在 static TreeNode＊exp(void)函数中，增加处理当前 token 为 LESSEQUAL（<=）、NEQ（<>）、MT（>）、MOREEQUAL（>=）的情况

代码如下：

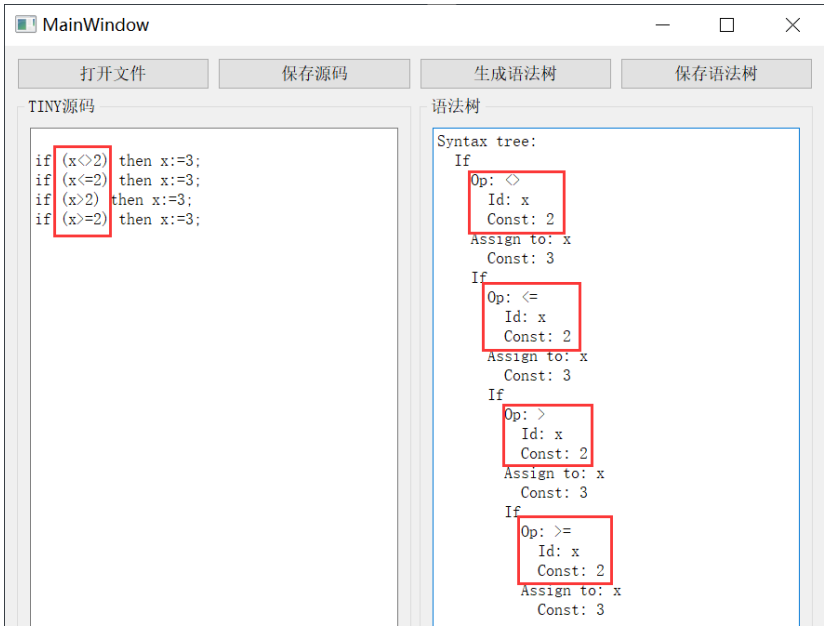
```
TreeNode * exp(void)
{
    TreeNode * t = simple_exp();
    // < = <= <> > >=
    if ((token == LT) || (token == EQ) || (token == LESSEQUAL)
        || (token == NEQ) ||(token == MT) || (token == MOREEQUAL))
    {
        TreeNode * p = newExpNode(OpK);
        if (p != NULL)
        {
            p->child[0] = t;
            p->attr.op = token;
            t = p;
        }
        match(token);
        if (t != NULL)
            t->child[1] = simple_exp();
    }
    return t;
}
```

➢ 在 util.cpp 文件中

在 void printToken(TokenType token, const char* tokenString)函数中，增加“<=”、“<>”、“>”、“>=”的输出：

```
case LESSEQUAL: fprintf(listing, "<=\n"); break;
case NEQ: fprintf(listing, "<>\n"); break;
case MT: fprintf(listing, ">\n"); break;
case MOREEQUAL: fprintf(listing, ">=\n"); break;
```

i） 测试结果



（八） 把 tiny 原来的赋值运算符号(:=)改为(=),而等于的比较符号符号（=）则改为（==）

修改原有的 tiny 语言的文法后：

程序清单4-8 EBNF中TINY语言的文法

program → *stmt-sequence*
stmt-sequence → *statement* { **;** *statement* }
statement → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
if-stmt → **if** *exp* **then** *stmt-sequence* [**else** *stmt-sequence*] **end**
repeat-stmt → **repeat** *stmt-sequence* **until** *exp*
assign-stmt → **identifier** **:=** *exp*
read-stmt → **read identifier**
write-stmt → **write** *exp*
exp → *simple-exp* [*comparison-op simple-exp*]
comparison-op → **<** | **=**
simple-exp → *term* { *addop term* }
addop → **+** | **-**
term → *factor* { *mulop factor* }
mulop → ***** | **/**
factor → **(** *exp* **)** | **number** | **identifier**

➢ 在 scan.cpp 中

在 TokenType getToken(void)函数中，删掉扫描到‘:’时的处理，增加扫描到‘=’时的处理：当前扫描到的是‘=’，使用 getNextChar()函数获取下一字符 c,如果 c 为‘=’,此时“==”构成等于的比较符号;否则使用 getNextChar()函数回退一个字符，此时为‘=’赋值运算符。

代码如下：

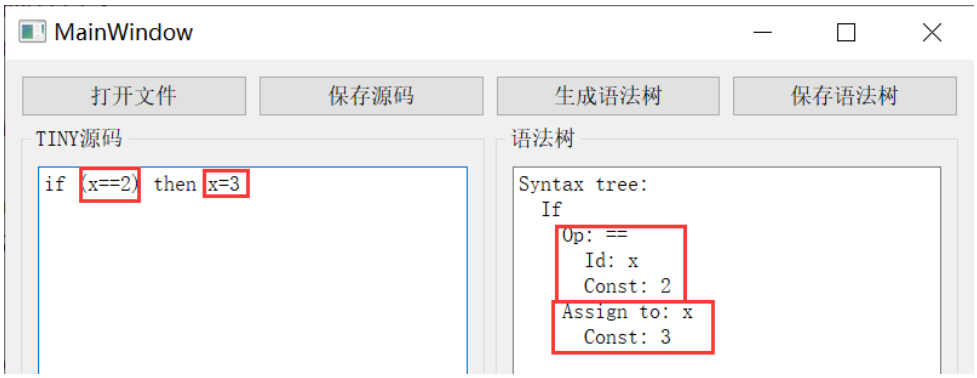
```
//else if (c == ':')
 删掉 //state = INASSIGN;

case '=':
    //=和==的情况
    c = getNextChar();
    if(c == '=')
        currentToken = EQ;
    else
    {
        ungetNextChar();
        currentToken = ASSIGN;
    }
    break;
```

➢ 在 util.cpp 文件中

在 void printToken(TokenType token, const char* tokenString)函数中，改变 ASSIGN 和 EQ 的输出：

```
case ASSIGN: fprintf(listing, "=\n"); break;
case EQ: fprintf(listing, "==\n"); break;
```

测试结果：



（九） 逻辑表达式

文法规则：（不考虑括号）

logic_exp→logic_ term { or logic_ term }

logic_term→logic_factor { and logic_factor }

logic_factor →not logic_factor | exp

➢ 在 parse.c 文件中

a）增加静态成员函数定义

```
//处理逻辑表达式
static TreeNode * logic_exp(void);
static TreeNode * logic_term(void);
static TreeNode * logic_factor(void);
```

代码如下：

```
//处理逻辑表达式
TreeNode * logic_exp(void)
{
    TreeNode * t = logic_term();
    while (token == OR) //or
    {
        TreeNode * p = newExpNode(OpK);
        if (p != NULL)
        {
            p->child[0] = t;
            p->attr.op = token;
            t = p;
            match(token);
            t->child[1] = logic_term();
        }
    }
    return t;
}
```

```
//处理逻辑表达式
TreeNode * logic_term(void)
{
    TreeNode * t = logic_factor();
    while (token == AND ) // and
    {
        TreeNode * p = newExpNode(OpK);
        if (p != NULL)
        {
            p->child[0] = t;
            p->attr.op = token;
            t = p;
            match(token);
            t->child[1] = logic_factor();
        }
    }
    return t;
}

//处理逻辑表达式
TreeNode * logic_factor(void)
{
    TreeNode * t = NULL;
    if(token==NOT){
        t = newExpNode(OpK);
        t->attr.op = token;
        match(token);
        t->child[0] = logic_factor();
    }else{
        t=exp();
    }
    return t;
}
```

b) 用于 repeat 循环、do while 循环、if 条件语句作条件判断的逻辑表达式

分别修改 TreeNode * repeat_stmt(void)、TreeNode * dowhile_stmt(void) 和 TreeNode * if_stmt(void)函数，将 exp()改为 logic_exp()

```
TreeNode * repeat_stmt(void)
{
    TreeNode * t = newStmtNode(RepeatK);
    match(REPEAT);
    if (t != NULL)
        t->child[0] = stmt_sequence();
    match(UNTIL);
    if (t != NULL)
        t->child[1] = logic_exp();
    return t;
}
```

```
/* 扩充dowhile文法 */
TreeNode * dowhile_stmt(void)
{
    TreeNode * t = newStmtNode(DowhileK);
    match(DO);
    if(t != NULL)
        t->child[0] = stmt_sequence();
    match(WHILE);
    match(LPAREN);
    if(t != NULL)
        t->child[1] = logic_exp();
    match(RPAREN);
    return t;
}
```

```
TreeNode * if_stmt(void)
{
    TreeNode * t = newStmtNode(IfK);
    match(IF);
    if (t != NULL)
    {
        match(LPAREN);
        t->child[0] = logic_exp();
        match(RPAREN);
    }
    if (t != NULL)
        t->child[1] = stmt_sequence();
    if (token == ELSE)
    {
        match(ELSE);
        if (t != NULL)
            t->child[2] = stmt_sequence();
    }
    return t;
}
```

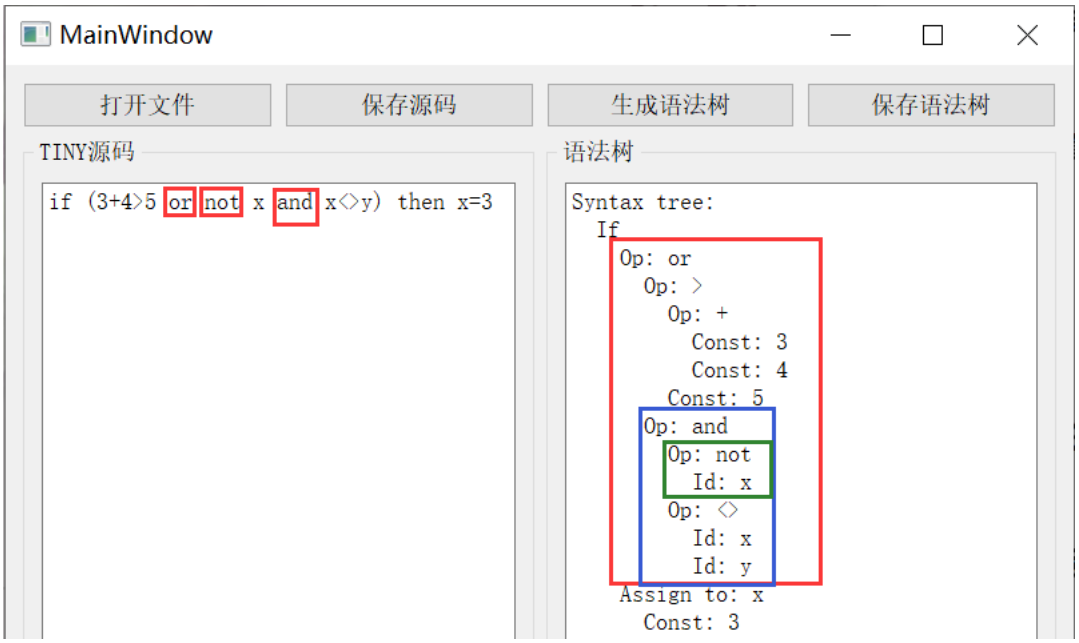
➢ 在 util.cpp 文件中

在 void printToken(TokenType token, const char* tokenString)函数中，增加 AND、OR 和 NOT 的输出：

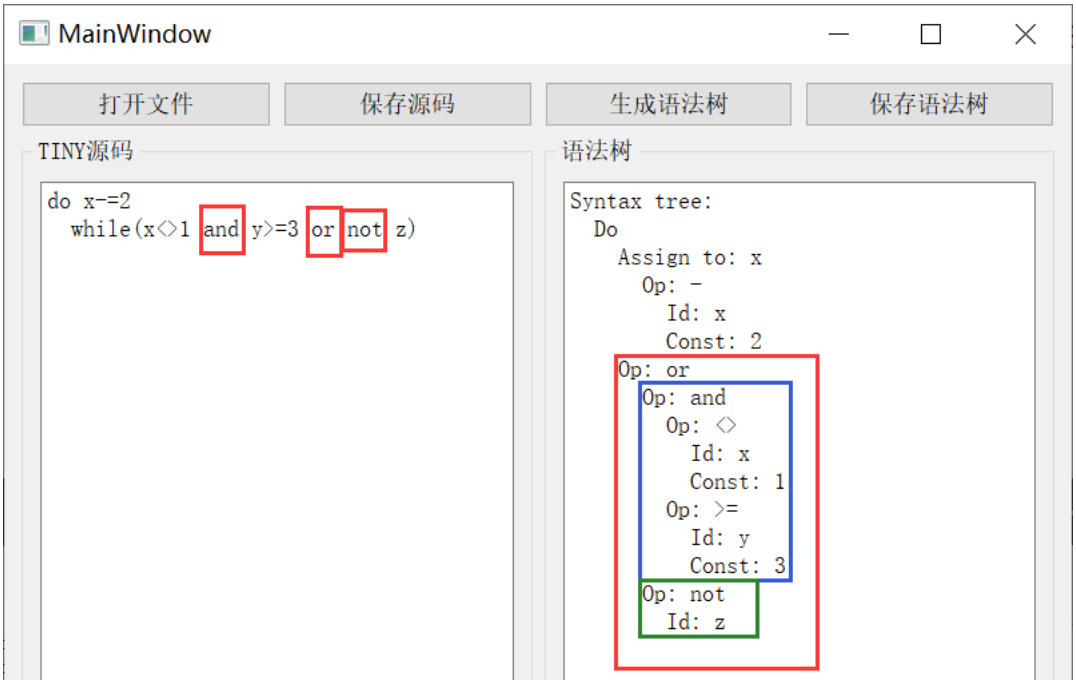
```
case AND: fprintf(listing, "and\n"); break;
case OR: fprintf(listing, "or\n"); break;
case NOT: fprintf(listing, "not\n"); break;
```

c）测试结果

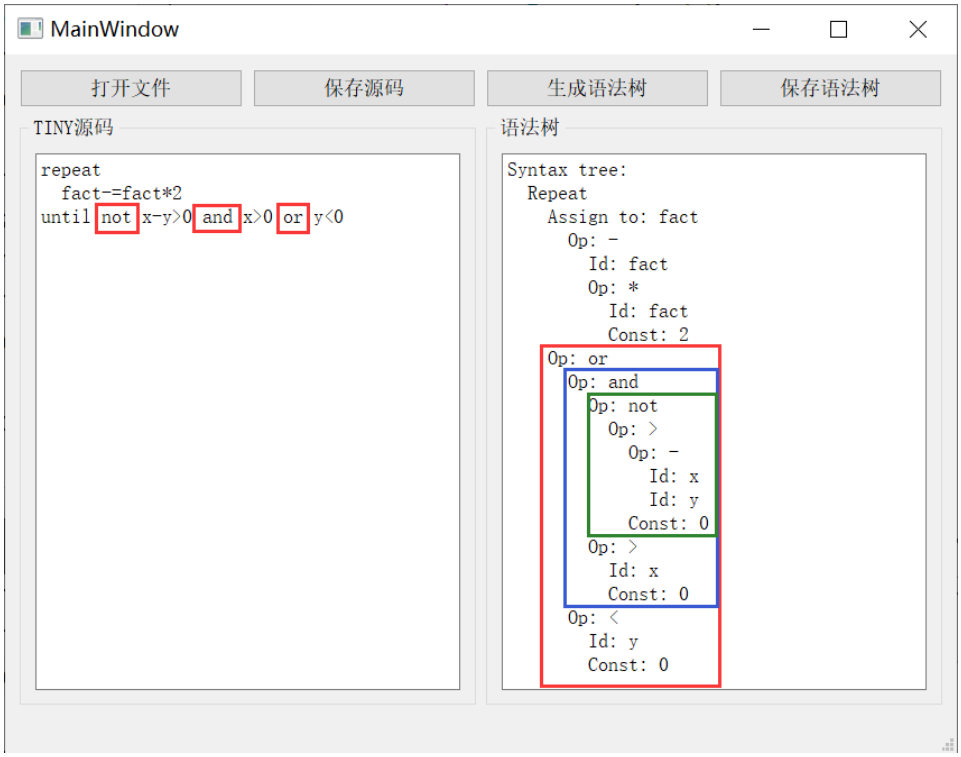
- 在 if 条件语句中测试



- 在 dowhile 循环中测试



- 在 repeat 循环中测试



（十） 新增支持正则表达式

文法规则：

re_exp→re_ termA { | re_termA }

re_termA→ re_termB { & re_termB }

re_termB→ re_ factor[#]

re_factor →(re_exp) | letter

➢ 因为正则表达式里的字符/字符串并不是变量（ID），所以我在 globals.h 中的 ExpKind 增加了 LETTERK（即可以生成 kind.exp 为 LETTERK 节点，这样打印的时候就可以打印成 Letter 而不是 ID）。

```
typedef enum {OpK, ConstK, IdK,LETTERK} ExpKind;
```

➢ 在 scan.cpp 中

在 TokenType getToken(void)函数中，增加扫描到‘：=’、‘&’、‘|’、‘#’时的处理。注意这里扫到字符/字符串时，还是先让它为 ID，在之后生成节点的时候将 kind.exp 变成 Letter 即可。

```
case '&':
    currentToken = READ;
    break;
case '|':
    currentToken = REOR;
    break;
case '#':
    currentToken = RECLO;
    break;
```

如果当前扫描到的字符为‘:’，设当前状态为 INREASSIGN。

```
else if (c == ':')
          state = INREASSIGN;
```

如果下一个字符为‘=’，则为正则表达式的赋值，将 currentToken 设为 REASSIGN

```
case INREASSIGN:
        state = DONE;
        if (c == '=')
          currentToken = REASSIGN;
        else
        { /* backup in the input */
          ungetNextChar();
          save = FALSE;
          currentToken = ERROR;
        }
        break;
```

➢ 在 parse.c 文件中

a）增加静态成员函数定义

```
//正则表达式
static TreeNode * re_exp(void);
static TreeNode * re_termA(void);
static TreeNode * re_termB(void);
static TreeNode * re_factor(void);
```

具体代码如下：

```
//处理正则表达式1
TreeNode * re_exp(void)
{
    TreeNode * t = re_termA();
    while (token == REOR) //or
    {
        TreeNode * p = newExpNode(OpK);
        if (p != NULL)
        {
            p->child[0] = t;
            p->attr.op = token;
            t = p;
            match(token);
            t->child[1] = re_termA();
        }
    }
    return t;
}
//处理正则表达式2
TreeNode * re_termA(void)
{
    TreeNode * t = re_termB();
    while (token == READ) // and
    {
        TreeNode * p = newExpNode(OpK);
        if (p != NULL)
        {
            p->child[0] = t;
            p->attr.op = token;
            t = p;
            match(token);
            t->child[1] = re_termB();
        }
    }
    return t;
}
//处理正则表达式3
TreeNode * re_termB(void)
{
    TreeNode * t = re_factor();
    if (token == RECLO) //可选，闭包#
    {
        match(token);
        TreeNode * p = newExpNode(OpK);
        if (p != NULL)
        {
            p->child[0] = t;
            p->attr.op = RECLO;
            t=p;
        }
    }
    return t;
}
```

注意这里生成节点的 kind.exp 为 LETTERK（为了输出 Letter 而不是 ID），并将 token 设为 LETTER

```
//处理正则表达式4
TreeNode * re_factor(void){
    TreeNode * t = NULL;
    if(token == LPAREN){
        match(LPAREN);
        t = re_exp();
        match(RPAREN);
    }else{
        t = newExpNode(LETTERK);
        token=LETTER;
        if ((t!=NULL) && (token==LETTER))
            t->attr.name = copyString(tokenString);
        match(LETTER);
    }
    return t;
}
```

b) 处理逻辑的思路：在对正则表达式的语句进行赋值时，首先进入 TreeNode * statement(void)函数中 case：ID 的分支：即 assign_stmt()函数。然后在 TreeNode * assign_stmt(void)函数中，增加 token 为 REASSIGN（即 ‘：=’）的处理。

```
TreeNode * assign_stmt(void)
{
    TreeNode * t = newStmtNode(AssignK);
    if ((t != NULL) && (token == ID))
        t->attr.name = copyString(tokenString);
    //生成值为token的节点，-=时用到
    TreeNode * left = newExpNode(IdK);
    left->attr.name = copyString(tokenString);
    match(ID);
    if(token == ASSIGN)  //处理=
    {
        match(ASSIGN);
        if (t != NULL)
            t->child[0] = exp();
    }
    else if(token == MinusEQUAL) //处理 -=
    {
        match(MinusEQUAL);
        //生成一个算术表达式节点
        TreeNode * p = newExpNode(OpK);
        if (p != NULL)
        {
            p->child[0] = left;
            p->attr.op = MINUS;//op为-
            p->child[1]=exp();
        }
        if(t != NULL)
            t->child[0] = p;
    }else if(token==REASSIGN){  //处理:=
        match(REASSIGN);
        if (t != NULL)
            t->child[0] = re_exp();
    }
    return t;
}
```

➢ 在 util.cpp 文件中

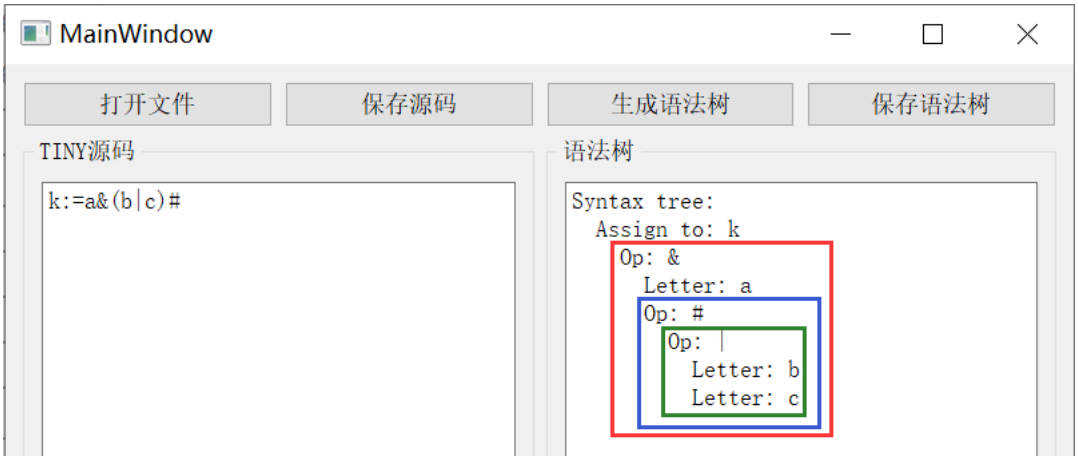
在 void printToken(TokenType token, const char* tokenString)函数中，增加 READ、REOR 和 RECLO 的输出：

```
case READ: fprintf(listing, "&\n"); break;
case REOR: fprintf(listing, "|\n"); break;
case RECLO: fprintf(listing, "#\n"); break;
```

在 void printToken(TokenType token, const char* tokenString)函数中，增加 LETTERK 时的输出：

```
case LETTERK:
    fprintf(listing, "Letter: %s\n", tree->attr.name);
    break;
```

➢ 测试结果

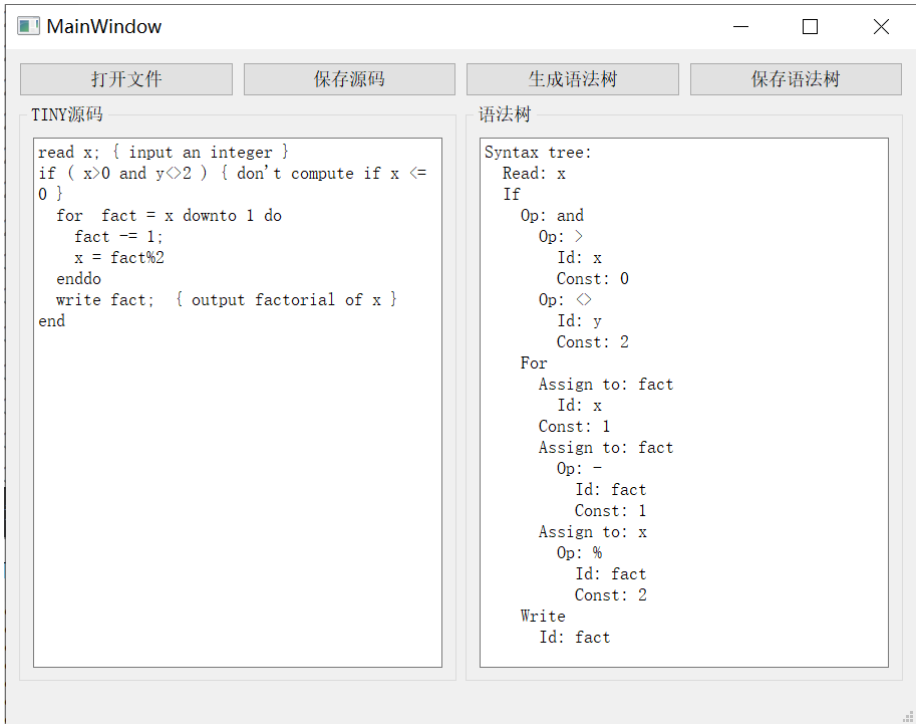


六、 全面测试

注意：测试只关注语法分析是否正确，而不必在意源程序的逻辑性

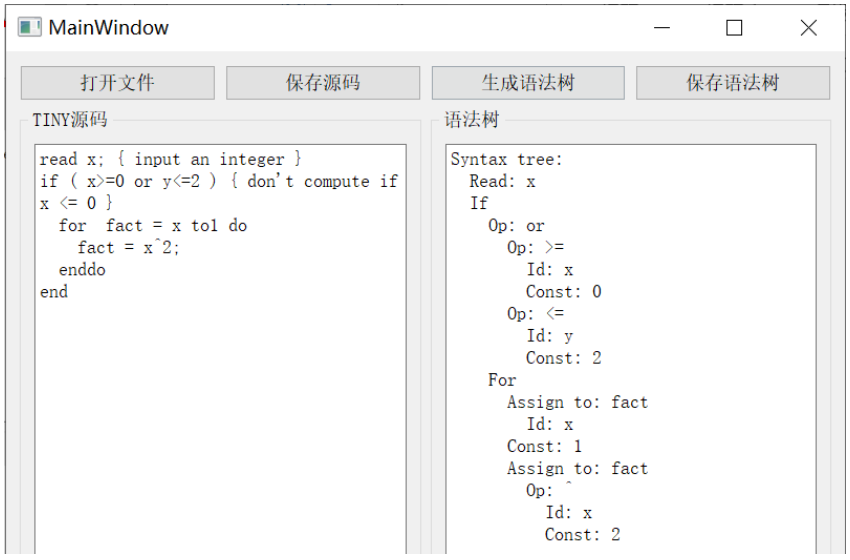
（一）测试文件 testFile1.txt

测试：>、and、<>、for 循环(downto)、-=、=、%



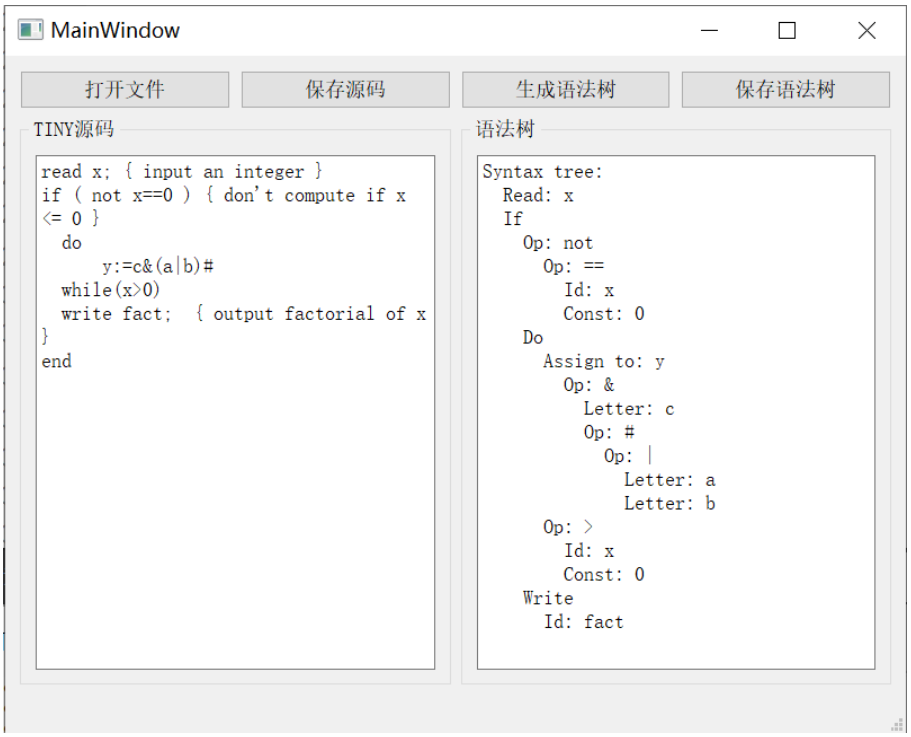
（二）测试文件 testFile2.txt

测试：>=、<=、or、for 循环（to）、^



（三）测试文件 testFile3.txt

测试：not、==、dowhile 循环、正则表达式



七、 思考总结

本次实验是根据题目的要求逐步实现的，在这个过程中也进行了相应的记录。通过本次实验，更加理解了递归向下分析法的处理过程。在理解了算术表达式的文法规则后，逻辑表达式和正则表达式的就比较好写了，不再像之前一样无从下笔。