



华南师范大学  
SOUTH CHINA NORMAL UNIVERSITY

2021—2022 年度编译原理

## 实验四报告

实验名称：TINY 扩充语言的语法分析

学生姓名：徐浩耀

学生学号：20192131005

指导教师：黄煜廉

所在院系：计算机学院

实验日期：2021/12/24

共青团华南师范大学委员会制

2021.4

## 实验四

### 一、实验内容：

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：  
-= 减法赋值运算符（类似于 C 语言的 -=）、求余%、乘方^，

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

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：运算符有 **and**（与）、**or**（或）、**not**（非）。

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。

(1) **Dowhile-stmt-->do stmt-sequence while(exp);**

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

(3) **for-stmt-->for identifier:=simple-exp downto simple-exp do stmt-sequence enddo** 步长递减 1

(4) -= 减法赋值运算符、求余%、乘方^、>=（大于等于）、<=（小于等于）、>（大于）、<>（不等于）运算符的文法规则请自行组织。

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

(6) 为 **tiny** 语言增加一种新的表达式——正则表达式，其支持的运算符有 或(|)、连接(&)、闭包(#)、括号() 以及基本正则表达式。

(7) 为 **tiny** 语言增加一种新的语句，**ID:=正则表达式**

(8) 为 **tiny** 语言增加一种新的表达式——逻辑表达式，其支持的运算符有 **and**(与)、**or**(或)、**非(not)**。

(9) 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。

## 实现 do while

### 1.根据文法写出对应代码[parse.c]

```
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);
    //match(SEMI);
    return t;
}
```

### 2.增加 statement 的种类 dowhile [globals.h]

```
typedef enum {IfK, RepeatK, AssignK, ReadK, WriteK, DowhileK} StmtKind;
```

### 3.增加关键字,do 和 while[scan.c], 关键字的数量也要增加[globals.h]

```
/* Lookup table of reserved words */
static struct
{
    char* str;
    TokenType tok;
} reservedWords[MAXRESERVED]
= {{ "if", IF }, { "then", THEN }, { "else", ELSE }, { "end", END },
   { "repeat", REPEAT }, { "do", DO }, { "while", WHILE }, { "until", UNTIL }, { "read", READ },
   { "write", WRITE } };

#define MAXRESERVED 10
```

### 4.由于语法树上多了一种 statement 类型,因此打印树的函数也要修改[util.c]

```
//在 util.c 中增加此行代码
case DowhileK:
    fprintf(listing, "Dowhile\n");
    break;
```

### 5.测试代码

```

D:\project\tiny\Debug\SAMPLE.TNY - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences

1 read x;
2 if ( x<0 ) then { x >= 0 }
3   fact := 1;
4   do
5     fact := fact * x;
6     x := x - 1
7   while((x<0));
8   write fact { output factorial of x }
9 end

C:\Windows\System32\cmd.exe
D:\project\tiny\Debug>tiny.exe SAMPLE

TINY COMPILATION: SAMPLE.tny

Syntax tree:
Read: x
If
  Op: <
  Id: x
  Const: 0
  Assign to: fact
  Const: 1
  Do While
    Assign to: fact
    Op: *
    Id: fact
    Id: x
    Assign to: x
    Op: -
    Id: x
    Const: 1
    Op: <
    Id: x
    Const: 0
  Write
  Id: fact

```

## 实现 for 的步长递增和递减

```

//(2) for-stmt-->for identifier:=simple-exp to simple-exp do stmt-sequence enddo    步长递增1
//(3) for-stmt-->for identifier:=simple-exp downto simple-exp do stmt-sequence enddo    步长递减1

```

合并得到

```

for-stmt-->for identifier:=simple-exp (to | downto) simple-exp
do stmt-sequence enddo

```

## 1.根据文法写出对应代码[parse.c]

```

TreeNode* for_stmt(void)
{
    TreeNode* t = newStmtNode(Fork);
    match(FOR);
    if ((t != NULL) && (token == ID))
        t->attr.name = copyString(tokenString);
    match(ID);
    match(ASSIGN);
    if (t != NULL)
        t->child[0] = simple_exp();
    if (token == TO || token == DOWNTO) {
        match(token);

```

```

        if (t != NULL)
            t->child[1] = simple_exp();
    }
    match(DO);
    if (t != NULL)
        t->child[2] = stmt_sequence();
    match(ENDDO);
    return t;
}

```

## 2.增加 statement 的种类 for[globals.h]

```
typedef enum {IfK, RepeatK, AssignK, ReadK, WriteK, DoWhileK, forK} StmtKind;
```

## 3.增加关键字,do 和 while[scan.c], 关键字的数量也要增加[globals.h]

```

/* Lookup table of reserved words */
static struct
{
    char* str;
    TokenType tok;
} reservedWords[MAXRESERVED]
= {{"if", IF}, {"then", THEN}, {"else", ELSE}, {"end", END},
   {"repeat", REPEAT}, {"do", DO}, {"while", WHILE}, {"until", UNTIL}, {"read", READ}, {"for", FOR},
   {"to", TO}, {"downto", DOWNTOW}, {"enddo", ENDDO}, {"write", WRITE}};

```

```
#define MAXRESERVED 14
```

## 4.由于语法树上多了一种 statement 类型,因此打印树的函数也要修改[util.c]

```

//在 util.c 中增加此行代码
case ForK:
    fprintf(listing, "for\n");
    break;

```

## 5.测试代码

The screenshot shows a compiler IDE with three tabs: PARSE.C, SCAN.C, and CGEN.H. The PARSE.C tab is active, displaying the following code:

```

1 read x; { input an integer }
2 if ( x<0 ) then
3     for fact := x downto 1 do
4         fact := fact * x
5     enddo;
6     write fact { output factorial of x }
7 end

```

To the right, the 'Syntax tree:' panel shows the following structure:

```

Syntax tree:
Read: x
If
  Op: <
  Id: x
  Const: 0
  for
    Id: x
    Const: 1
    Assign to: fact
    Op: *
    Id: fact
    Id: x
  Write
    Id: fact

```

The status bar at the bottom right indicates the path: D:\project\tiny\Debug>

-= 减法赋值运算符、求余%、乘方^、>=(大于等于)、<=(小于等于)、>(大于)、<>(不等于)运算符的扩充

-=减法赋值运算符的扩充

1.scan.c 中，当扫描到字符-的时候，判断下一个字符是不是=，若是的话，就是-=，否则回退[此时是减号]

```

case '-':
    c = getNextChar();
    if (c == '=') {
        currentToken = MINUSEQ;
    }
    else {
        ungetNextChar();
        currentToken = MINUS;
    }
    break;

```

2.在 GLOBALS.H 中的 TokenType 中增加 MINUSEQ 的字段

3.把 x-=1 看成是 x := x - 1,因此在 assign\_stmt 函数中改动【parse.c】

```

TreeNode * assign_stmt(void)
{
    TreeNode * t = newStmtNode(AssignK);
    if ((t!=NULL) && (token==ID))
        t->attr.name = copyString(tokenString);
    match(ID);
}

```

```

    if (token == MINUSEQ) { //处理-=逻辑:就是 x -= 1 转变为跟以前语法树上
        展示 x = x - 1 是一样的效果
        match(MINUSEQ);
        TreeNode* opNode = newExpNode(OpK);
        if (opNode != NULL) {
            TreeNode* idNode = newExpNode(IdK);
            if (t != NULL)
                idNode->attr.name = t->attr.name;
            if (opNode != NULL) {
                opNode->child[0] = idNode;
                opNode->attr.op = MINUS;
                opNode->child[1] = exp();
            }
        }
        if (t != NULL) {
            t->child[0] = opNode;
        }
    }
    else if (token == ASSIGN) { //处理:=
        match(ASSIGN);
        if (t != NULL) t->child[0] = exp();
    }
    return t;
}

```

```

D:\project\tiny\Debug\SAMPLE.TNY - Sublime Text (UNREGISTERED)
le Edit Selection Find View Goto Tools Project Preferences Hs
PARSEC x | SCAN.C x | CGEN.H x | U
1 read x;
2 if ( x<0 ) then { x >= 0 }
3 fact -= x + 11;
4 x -= 1;
5 write fact { output factorial of x }
6 end

C:\Windows\System32\cmd.exe
D:\project\tiny\Debug>tiny.exe SAMPLE
TINY COMPILATION: SAMPLE.tny

Syntax tree:
Read: x
If
  Op: <
    Id: x
    Const: 0
  Assign to: fact
    Op: -
      Id: fact
      Op: +
        Id: x
        Const: 11
    Assign to: x
      Op: -
        Id: x
        Const: 1
  Write
    Id: fact

D:\project\tiny\Debug>

```

## 求余%的扩充

1.在 GLOBALS.H 的 tokenType 中增加 MOD 字段，表示求余

## 2.scan.c 中增加一个判断

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

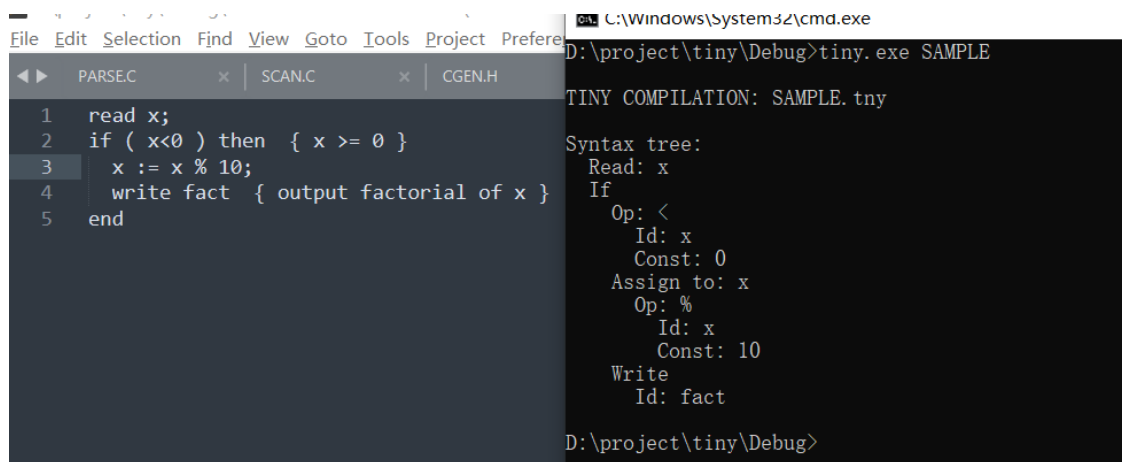
## 3.改写 parse.c 中的 term()代码

```
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;
}
```

## 4.在 util.c 中的 printToken 中增加打印 MOD 的代码

此处需要两个%%用来转义，一个%的话显示不出来，可能跟函数底层代码逻辑有关系

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



```
File Edit Selection Find View Goto Tools Project Preferences
1 read x;
2 if ( x<0 ) then { x >= 0 }
3 x := x % 10;
4 write fact { output factorial of x }
5 end

D:\project\tiny\Debug>tiny.exe SAMPLE
TINY COMPILATION: SAMPLE.tny
Syntax tree:
Read: x
If
Op: <
Id: x
Const: 0
Assign to: x
Op: %
Id: x
Const: 10
Write
Id: fact

D:\project\tiny\Debug>
```

>的扩充

## 1.GLOBALS.H 中的 TokenType 增加 GT 字段



2.scan.c 中 getToken 函数增加以下代码

```
case '>':
    currentToken = GT;
    break;
```

3.parse.c 中 exp 函数增加对于 GT 的判断

```
if ((token==LT) || (token==EQ) || (token==GT))
```

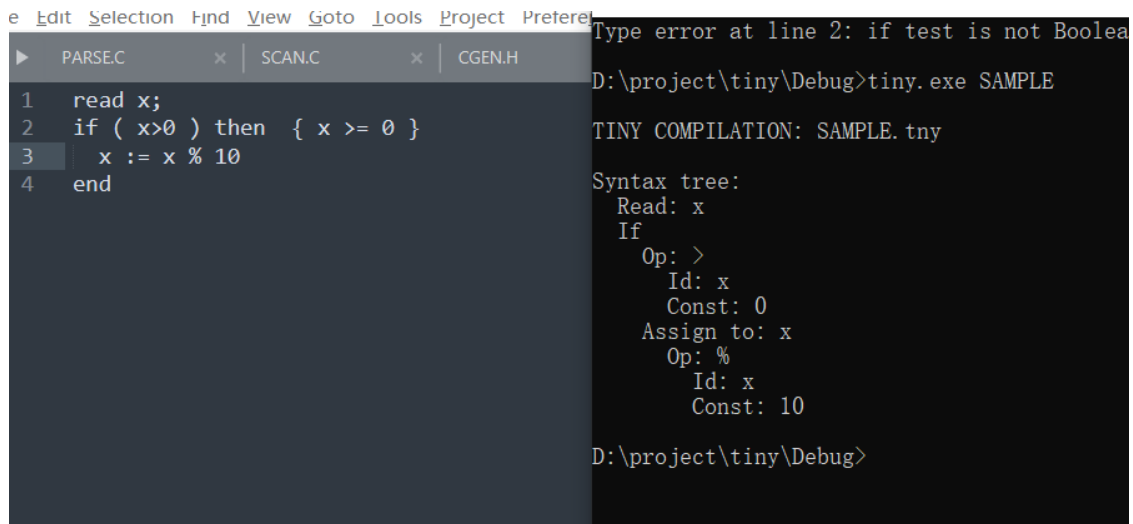
4.util.c 中 printToken 函数增加打印>的功能

```
case GT: fprintf(listing, ">\n"); break;
```

5.ANALYZE.C 中增加对于>判断的逻辑

checkNode 函数修改如下

```
if ((t->attr.op == EQ) || (t->attr.op == LT) || (t->attr.op == GT))
    t->type = Boolean;
```



```
e Edit Selection Find View Goto Tools Project Preferences
Type error at line 2: if test is not Boolean
D:\project\tiny\Debug>tiny.exe SAMPLE
TINY COMPILATION: SAMPLE.tny
Syntax tree:
Read: x
If
  Op: >
  Id: x
  Const: 0
Assign to: x
  Op: %
  Id: x
  Const: 10
D:\project\tiny\Debug>
```

<=的扩充: 详细版, >=和<>的跟<=类似处理即可

1.增加 LTEQ 字段【GLOBALS.H】

2.scan.c 中, 对于 < 的逻辑, 修改如下

即查看后面那个字符是不是=

```
case '<':
    c = getNextChar();
```

```

        if (c == '=') {
            currentToken = LTEQ;
        }
        else {
            ungetNextChar();
            currentToken = LT;
        }
    }
    break;

```

3.parse.c 中 **exp** 函数增加对于<=的判断

```
if ((token==LT) || (token==EQ) || (token==GT) || (token==LTEQ))
```

4.analyze.c 中 **checkNode** 函数增加对于<=的判断

```
if ((t->attr.op == EQ) || (t->attr.op == LT) || (t->attr.op == GT) ||
(t->attr.op == LTEQ))
```

5.util.c 中，增加打印<=的功能

```
case LTEQ: fprintf(listing, "<=\n"); break;
```

```

1  read x;
2  if ( x<=0 ) then
3      x := x % 10
4  end

```

```

D:\project\tiny\Debug>tiny.exe SAMPLE
TINY COMPILATION: SAMPLE.tny

Syntax tree:
Read: x
If
  Op: <=
  Id: x
  Const: 0
Assign to: x
  Op: %
  Id: x
  Const: 10
D:\project\tiny\Debug>

```

*>=的扩充*

跟<=的十分类似，这里只给出测试代码

```
File Edit Selection Find View Tools Project Help
<< >> PARSE.C x SCAN.C x CGEN.H
1 read x;
2 if ( x>=0 ) then
3   x := x % 10
4 end

D:\project\tiny\Debug>tiny.exe SAMPLE
TINY COMPILATION: SAMPLE.tny

Syntax tree:
Read: x
If
  Op: >=
  Id: x
  Const: 0
Assign to: x
  Op: %
  Id: x
  Const: 10

D:\project\tiny\Debug>_
```

### <>的扩充

<>的扩充与<=的十分类似，这里就不给出步骤了，测试代码如下

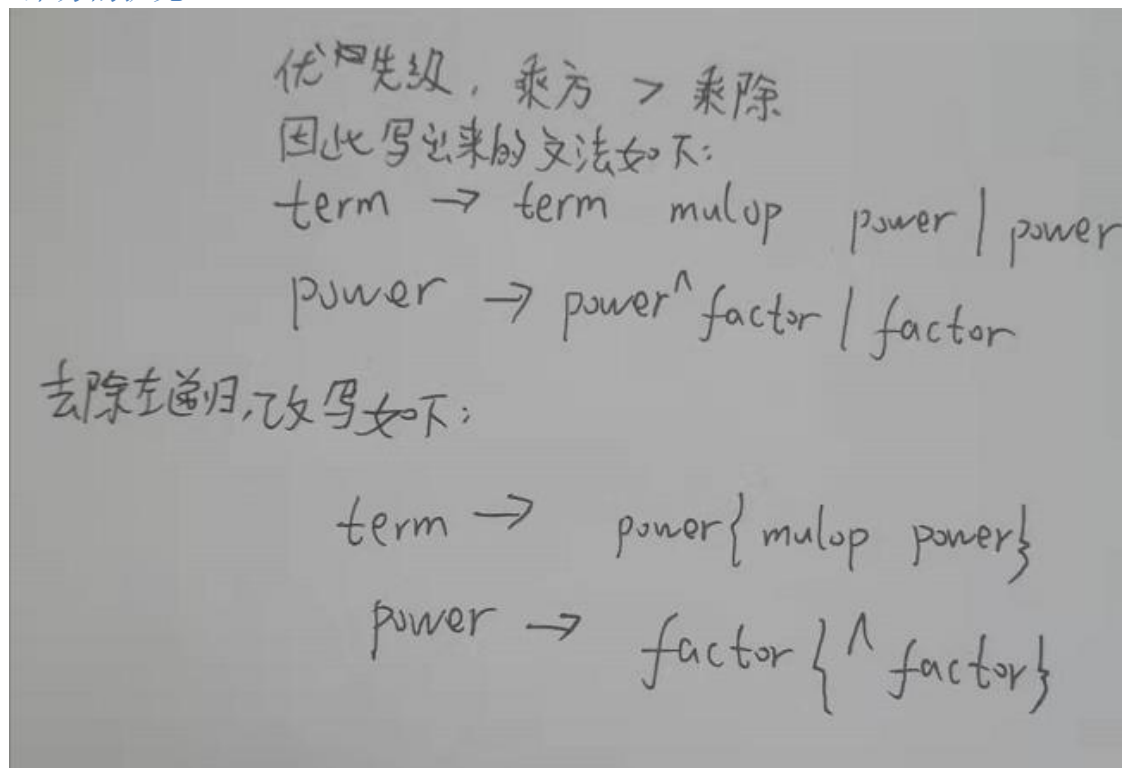
```
File Edit Selection Find View Tools Project Help
<< >> PARSE.C x SCAN.C x CGEN.H
1 read x;
2 if ( x<>0 ) then
3   x := x % 10
4 end

D:\project\tiny\Debug>tiny.exe SAMPLE
TINY COMPILATION: SAMPLE.tny

Syntax tree:
Read: x
If
  Op: <>
  Id: x
  Const: 0
Assign to: x
  Op: %
  Id: x
  Const: 10

D:\project\tiny\Debug>
```

## ^乘方的扩充



1.GLOBALS.H 的 TokenType 添加字段 POWER 用来表示乘方

2.scan.c 中, 增加对于乘方^的判断

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

3.util.c 中, 增加对于乘方^的输出

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

4.parse.c 中改写 term() 函数以及新增一个 power 函数用来处理乘方

```
TreeNode * term(void)  
{  
    TreeNode * t = power();  
    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();  
        }  
    }  
}
```

```

    }
    return t;
}

TreeNode* power(void)
{
    TreeNode* t = factor();
    while (token == POWER) {
        TreeNode* p = newExpNode(OpK);
        if (p != NULL) {
            p->child[0] = t;
            p->attr.op = POWER;
            t = p;
            match(POWER);
            p->child[1] = factor();
        }
    }
    return t;
}

```

## 5.测试代码如下:

```

D:\project\tiny\Debug\SAMPLE.TNY - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences

1 read x;
2 if ( x<>0 ) then
3   x := x ^ 10 * 2
4 end

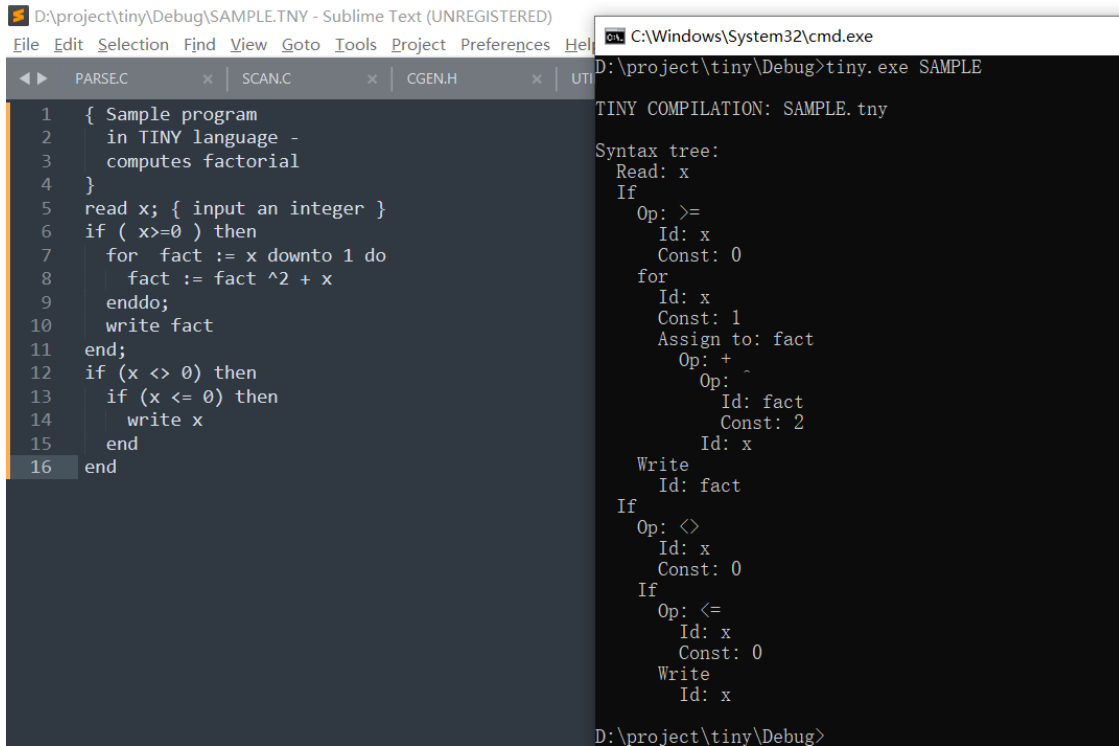
C:\Windows\System32\cmd.exe
D:\project\tiny\Debug>tiny.exe SAMPLE
TINY COMPILATION: SAMPLE.tny

Syntax tree:
Read: x
If
  Op: <>
  Id: x
  Const: 0
  Assign to: x
  Op: *
  Op: ^
  Id: x
  Const: 10
  Const: 2

D:\project\tiny\Debug>

```

## 目前为止的综合测试



The screenshot shows a Sublime Text editor window titled "D:\project\tiny\Debug\SAMPLE.TNY - Sublime Text (UNREGISTERED)". The editor contains a TINY program with 16 lines of code. To the right, a command prompt window titled "C:\Windows\System32\cmd.exe" shows the output of running "tiny.exe SAMPLE". The output includes the text "TINY COMPILATION: SAMPLE.tny" and a detailed syntax tree for the program.

```
1 { Sample program
2   in TINY language -
3   computes factorial
4 }
5 read x; { input an integer }
6 if ( x>=0 ) then
7   for fact := x downto 1 do
8     fact := fact ^2 + x
9   enddo;
10  write fact
11 end;
12 if (x <> 0) then
13   if (x <= 0) then
14     write x
15   end
16 end
```

TINY COMPILATION: SAMPLE.tny

Syntax tree:

```
Read: x
If
  Op: >=
  Id: x
  Const: 0
  for
    Id: x
    Const: 1
    Assign to: fact
    Op: +
    Op: ^
    Id: fact
    Const: 2
    Id: x
  Write
    Id: fact
If
  Op: <>
  Id: x
  Const: 0
  If
    Op: <=
    Id: x
    Const: 0
    Write
      Id: x
```

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

改动思路：照葫芦画瓢

在 **scan.c** 中，DFA 图中添加一个状态 **INEQUAL**，扫描到**=**的时候，表示目前的状态是正在进行比较

因为:在改动的过程中已经没有作用了，去掉它，取而代之的是**=**

### 1.在 **scan.c** 中的 **getToken** 改动代码

之前的代码

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

改动之后的代码

```
else if (c == '=')
    state = INEQUAL;
```

之前的代码

```
case INASSIGN:
    state = DONE;
    if (c == '=')
        currentToken = ASSIGN;
```

```

else
{ /* backup in the input */
  ungetNextChar();
  save = FALSE;
  currentToken = ERROR;
}
break;

```

改动之后的代码

```

case INEQUAL:
  state = DONE;
  if (c == '=')
    currentToken = EQ;
  else
  { /* backup in the input */
    ungetNextChar();
    currentToken = ASSIGN;
  }
  break;

```

去除这段代码，因为状态变为 INEQUAL 之后，这段代码就执行不到了

```

case '=':
  currentToken = EQ;
  break;

```

## 2.util.c 中的 printToken 函数改动代码

```

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

```

测试代码如下：

The screenshot shows two windows. The left window is Sublime Text (UNREGISTERED) editing 'D:\project\tiny\Debug\SAMPLE.TNY'. It contains the following C code:

```

1 { Sample program
2   in TINY language -
3   computes factorial
4 }
5 read x; { input an integer }
6 if ( 0<>x ) then { don't compute if x <= 0 }
7   for fact = x downto 1 do
8     fact = fact * x
9   enddo;
10  write fact { output factorial of x }
11 end

```

The right window is a Windows command prompt (C:\Windows\System32\cmd.exe) showing the output of the program:

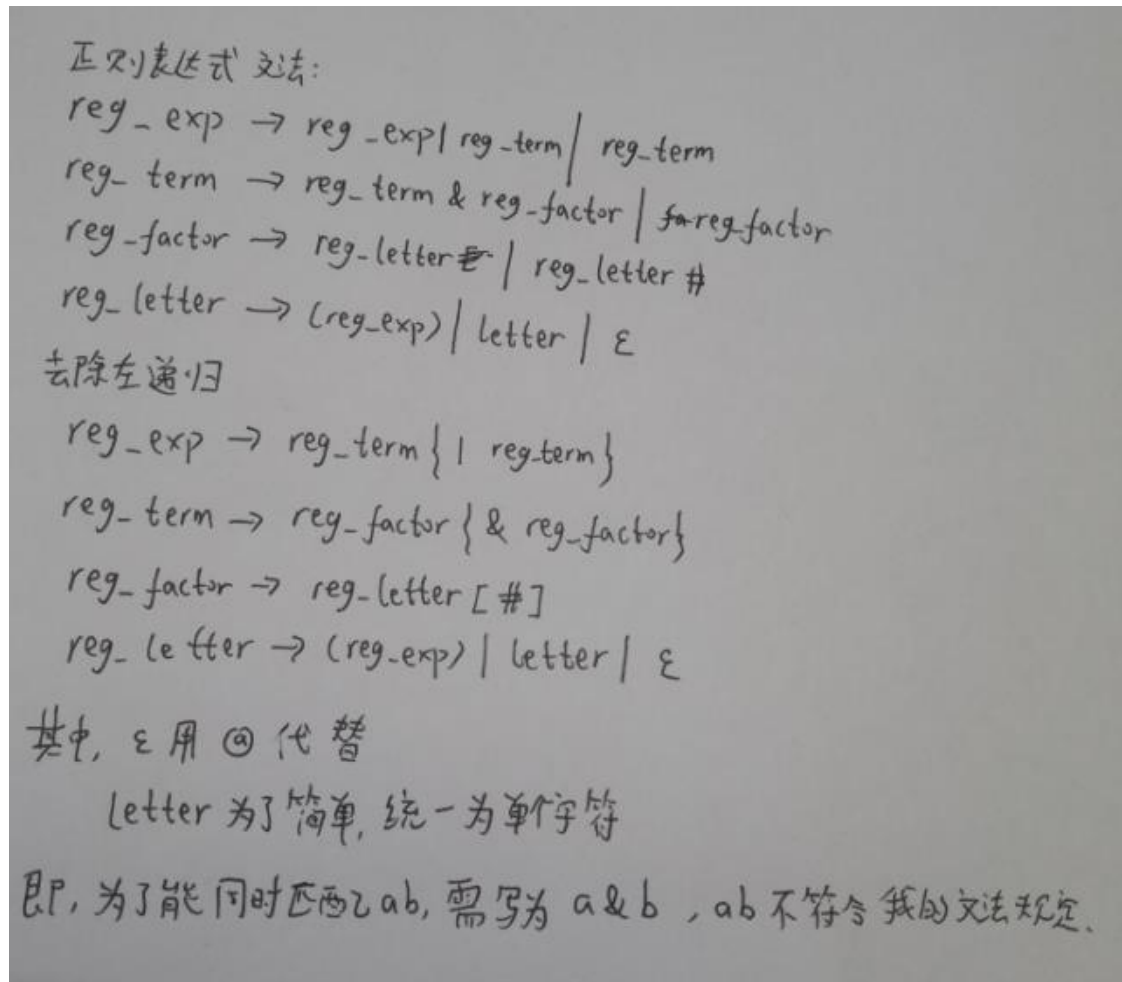
```

Syntax tree:
Read: x
If
  Op: <>
  Const: 0
  Id: x
for
  Id: x
  Const: 1
  Assign to: fact
  Op: *
  Id: fact
  Id: x
Write
  Id: fact
D:\project\tiny\Debug>

```

为 tiny 语言增加一种新的表达式——正则表达式，其支持的运算符有 或(|)、连接(&)、闭包(#)、括号() 以及基本正则表达式。

### 1.写出文法



### 2.GLOBALS.H 的 TokenType 添加记号

REG_OR: 或	
REG_AND: 连接	&
CLOSURE: 闭包	#
EPSILON: 空串	@

### 3.scan.c 中 getToken 函数增加对于记号的判断

```
case '&':  
    currentToken = REG_AND;  
    break;
```



```

    case '|':
        currentToken = REG_OR;
        break;
    case '#':
        currentToken = CLOSURE;
        break;
    case '@':
        currentToken = EPSILON;
        break;

```

#### 4.util.c 中 printToken 函数增加打印记号的功能

```

    case REG_AND : fprintf(listing, "&\n"); break;
    case REG_OR : fprintf(listing, "| \n"); break;
    case CLOSURE : fprintf(listing, "#\n"); break;
    case EPSILON : fprintf(listing, "@\n"); break;

```

#### 5.parse.c 增加函数定义

```

static TreeNode* reg_exp(void);
static TreeNode* reg_term(void);
static TreeNode* reg_factor(void);
static TreeNode* reg_letter(void);

```

#### 6.parse.c 实现新定义的函数

```

TreeNode* reg_exp(void)
{
    TreeNode* t = reg_term();
    while (token == REG_OR) {
        TreeNode* p = newExpNode(OpK);
        if (p != NULL) {
            p->attr.op = REG_OR;
            p->child[0] = t;
            t = p;
            match(REG_OR);
            p->child[1] = reg_term();
        }
    }
    return t;
}

```

```

TreeNode* reg_term(void)
{
    TreeNode* t = reg_factor();
    while (token == REG_AND) {
        TreeNode* p = newExpNode(OpK);
        if (p != NULL) {
            p->attr.op = REG_AND;
            p->child[0] = t;
            t = p;

```

```

        match(REG_AND);
        p->child[1] = reg_factor();
    }
}
return t;
}

```

```

TreeNode* reg_factor(void)
{
    TreeNode* t = reg_letter();
    if (token == CLOSURE) {
        TreeNode* p = newExpNode(OpK);
        if (p != NULL) {
            p->attr.op = CLOSURE;
            t = p;
            match(CLOSURE);
            p->child[0] = t;
        }
    }
    return t;
}

```

```

TreeNode* reg_letter(void)
{
    TreeNode* t;
    switch (token) {
        case LPAREN:
            match(LPAREN);
            t = reg_exp();
            match(RPAREN);
            break;
        case ID:
            t = newExpNode(IdK);
            if (t != NULL) {
                t->attr.name = copyString(tokenString);
            }
            match(ID);
            break;
        case EPSILON:
            t = newExpNode(IdK);
            if (t != NULL) {
                t->attr.name = copyString(tokenString);
            }
            match(EPSILON);
            break;
        default:
            syntaxError("unexpected token -> ");
            printToken(token, tokenString);
            token = getToken();
            break;
    }
}

```

```

    }
    return t;
}

```

为 tiny 语言增加一种新的语句，ID:=正则表达式

1.globals.h 新定义符号 REG\_ASSIGN (:=)

2.scan.c 中 DFA 图的状态变量新增一个 INREGASSIGN

```

switch (state)
{ case START:
    if (isdigit(c))
        state = INNUM;
    else if (isalpha(c))
        state = INID;
    else if (c == ':')
        state = INREGASSIGN;

    // 若在 INREGASSIGN 状态没有接受到=, 那么出错
    case INREGASSIGN:
        if (c == '=') {
            currentToken = REG_ASSIGN;
            break;
        }
    case DONE:
    default: /* should never happen */
        fprintf(listing, "Scanner Bug: state= %d\n", state);
        state = DONE;
        currentToken = ERROR;
        break;
}

```

3.parse.c 中 assign\_stmt 函数增加一个判断即可

```

else if (token == REG_ASSIGN) { // 处理:=
    match(REG_ASSIGN);
    if (t != NULL) t->child[0] = reg_exp();
}

```

4.util.c 中增加打印:=的功能

```

case REG_ASSIGN: fprintf(listing, ":=\n"); break;

```

## 正则表达式测试

```
PARSEC x | SCAN.C x | CGEN.H x
1 x := a & b | c;
2 y := a & (b | @)

D:\project\tiny\Debug>tiny.exe test-demand-reg.TNY
TINY COMPILATION: test-demand-reg.TNY

Syntax tree:
Assign to: x
Op: |
Op: &
Id: a
Id: b
Id: c
Assign to: y
Op: &
Id: a
Op: |
Id: b
Id: @

D:\project\tiny\Debug>_
```

为 tiny 语言增加一种新的表达式——逻辑表达式，其支持的运算符有 **and(与)**、**or(或)**、**非(not)**。

### 1. 写出文法规则

逻辑表达式文法: (不考虑括号)

$$\begin{aligned} \text{logic\_exp} &\rightarrow \text{logic\_exp} \mid \text{logic\_term} \mid \text{logic\_term} \\ \text{logic\_term} &\rightarrow \text{logic\_term} \& \text{logic\_factor} \mid \text{logic\_factor} \\ \text{logic\_factor} &\rightarrow ! \text{logic\_factor} \mid \text{exp} \end{aligned}$$

去除左递归

$$\begin{aligned} \text{logic\_exp} &\rightarrow \text{logic\_term} \{ \text{or } \text{logic\_term} \} \\ \text{logic\_term} &\rightarrow \text{logic\_factor} \{ \text{and } \text{logic\_factor} \} \\ \text{logic\_factor} &\rightarrow \text{not } \text{logic\_factor} \mid \text{exp} \end{aligned}$$

## 2.GLOBALS.H 的 TokenType 新增字段 AND,OR,NOT

### 3.parse.c 新增函数定义

```
static TreeNode* logic_exp(void);
static TreeNode* logic_term(void);
static TreeNode* logic_factor(void);
```

### 4.实现如下:

```
TreeNode* logic_exp(void) {
    TreeNode* t = logic_term();
    while (token == OR) {
        TreeNode* p = newExpNode(OpK);
        if (p != NULL) {
            p->child[0] = t;
            t = p;
            match(OR);
            p->child[1] = logic_term();
        }
    }
    return t;
}

TreeNode* logic_term(void) {
    TreeNode* t = logic_factor();
    while (token == AND) {
        TreeNode* p = newExpNode(OpK);
        if (p != NULL) {
            p->child[0] = t;
            t = p;
            match(AND);
            p->child[1] = logic_factor();
        }
    }
    return t;
}

TreeNode* logic_factor(void) {
    TreeNode* t;
    if (token == NOT) {
        match(NOT);
        t = logic_factor();
    }
    else {
        t = exp();
    }
    return t;
}
```

### 5.util.c 中 printToken 新增打印逻辑表达式的逻辑

```

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

```

6.由于文法的写法原因，这里需要改写 if，在 if 的时候就判断有没有左括号，改写如下

```

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

```

7.然后对于文法的其他位置，出现 exp 的地方，都替换成 logic\_exp 即可

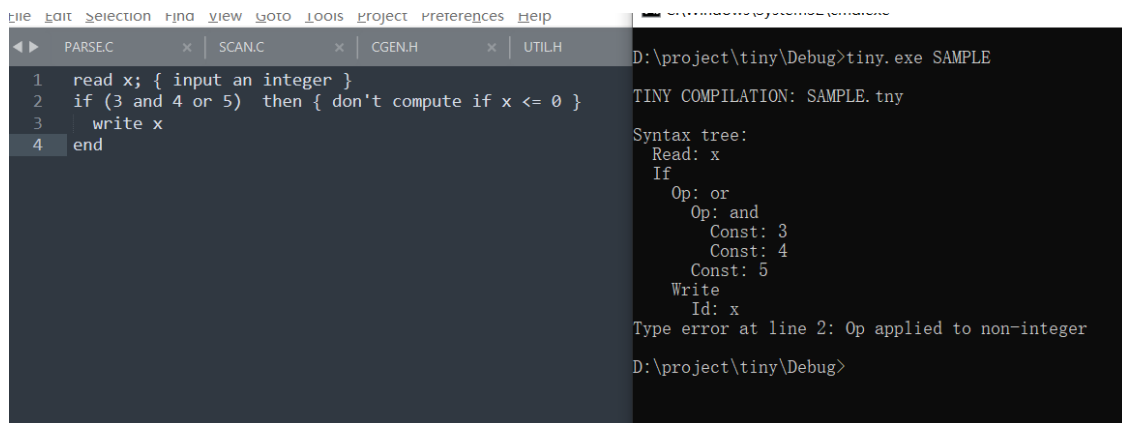
8.最后忘了说了，在 scan.c 中，and,or,not 先被判定为标识符，最后再修改为相应的逻辑符号

```

    if (currentToken == ID) {
        if (!strcmp(tokenString, "and\0")) {
            currentToken = AND;
        }
        else if (!strcmp(tokenString, "or\0")) {
            currentToken = OR;
        }
        else if (!strcmp(tokenString, "not\0")) {
            currentToken = NOT;
        }
        else {
            currentToken = reservedLookup(tokenString);
        }
    }
}

```

9.测试代码如下



The screenshot shows the Tiny compiler IDE. The code editor on the left contains the following code:

```
1 read x; { input an integer }
2 if (3 and 4 or 5) then { don't compute if x <= 0 }
3   write x
4 end
```

The console window on the right shows the output of the compilation:

```
D:\project\tiny\Debug>tiny.exe SAMPLE
TINY COMPILATION: SAMPLE.tny
Syntax tree:
Read: x
If
  Op: or
  Op: and
    Const: 3
    Const: 4
  Const: 5
Write
  Id: x
Type error at line 2: Op applied to non-integer
D:\project\tiny\Debug>
```

总结：此次实验 4 十分有趣，自己根据所学的知识修改 **tiny** 源码！而且自己定义文法实现了简单的正则表达式等等。每做完一个小步骤就编写测试代码看看自己有没有写对，语法树正确无误显示出来的时候挺满足的！为了保持修改源码的完整度，此次是直接在源码上改动，没有将代码迁移至 **QT**，相关的测试文件保留在了对应文件夹。