

[一、实验内容 2](#_Toc121739629)

[1.概述 3](#_Toc121739630)

[2.要求 4](#_Toc121739631)

[二、实验目的 4](#_Toc121739632)

[三、实验过程 4](#_Toc121739633)

[（一）理论准备 4](#_Toc121739634)

[1.定义 4](#_Toc121739635)

[2.算法 5](#_Toc121739636)

[（1）枚举 5](#_Toc121739637)

[（2）基础代码架构 5](#_Toc121739638)

[（3）输入输出 6](#_Toc121739639)

[（4）赋值 6](#_Toc121739640)

[（5）算数运算 6](#_Toc121739641)

[（6）逻辑运算 6](#_Toc121739642)

[（7）if条件转移语句 7](#_Toc121739643)

[（8）repeat循环语句 7](#_Toc121739644)

[（9）正则表达式 7](#_Toc121739645)

[（二）代码架构 8](#_Toc121739646)

[1.读取用户源程序 8](#_Toc121739647)

[2.生成语法树 8](#_Toc121739648)

[（1）语义分析结构 8](#_Toc121739649)

[（2）前序遍历输出语法树 9](#_Toc121739650)

[3.界面 9](#_Toc121739651)

[（四）测试数据 10](#_Toc121739652)

[测试文件1： 10](#_Toc121739653)

[测试文件2： 10](#_Toc121739654)

[测试文件3： 11](#_Toc121739655)

[测试文件4： 11](#_Toc121739656)

[测试文件5： 12](#_Toc121739657)

[测试文件6： 12](#_Toc121739658)

[测试文件7： 12](#_Toc121739659)

[测试文件8： 13](#_Toc121739660)

[四、实验总结 14](#_Toc121739661)

[五、参考文献 14](#_Toc121739662)

一、实验内容

## 1.概述

扩充语法规则：

①实现改写书写格式的新if语句，

②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^，

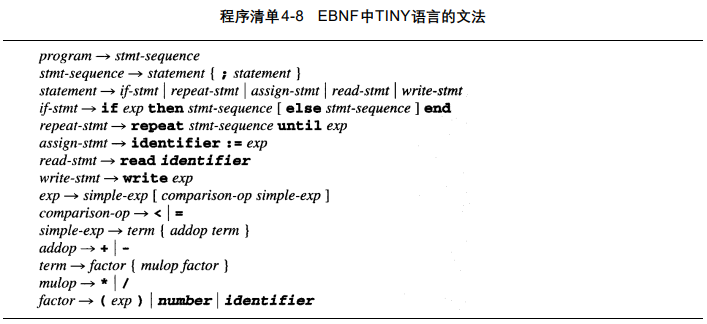
③扩充比较运算符号：==（等于），>(大于)、<=(小于等于)、>=(大于等于)、<>(不等于)等运算符号，

④新增支持正则表达式以及用于repeat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） 。

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。



（原文法）



（消除了左递归之后的文法）

备注：

(1) -= 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。

(2)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（==）

(3)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 。

(4)为tiny语言增加一种新的语句，ID:=正则表达式

(5)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)。

(6) 把TINY语言原有的if语句书写格式

if\_stmt-->if exp then stmt-sequence end | | if exp then stmt-sequence else stmt-sequence end

改写为：

if\_stmt-->if(exp) stmt-sequence else stmt-sequence | if(exp) stmt-sequence

(7)为了实现以上的扩充或改写功能，还需要注意对原tiny语言的文法规则做一些相应的改造处理。

## 2.要求

（1）要提供一个源程序编辑的界面，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。

**截止时间：12月12日**

# 二、实验目的

1、深化对编译过程中词法分析和语法分析过程的理解

2、熟练改写与创造文法规则

3、增强编码能力

# 三、实验过程

## （一）理论准备

### 1.定义

编译原理这门课就是学怎么将计算机语言**翻译**为可以在计算机硬件上直接运行的机器语言。这个过程涉及词法分析、语法分析和语义分析。词法分析就是扫描代码，断词，把字段标记为计算机语言能够识别的基本单元。语法分析就是根据特定的文法规则，规约经过词法分析处理后的基本单元，产生系统的、有逻辑的语法树。语义分析我还没懂，好像就是把代码转译为编译指令。

TINY语言是一种小巧却又能完成编译任务的语言。

我所理解的tiny语言源码的架构思路是，将读入的代码记录在全局变量buffer中，开辟一个位置指针pos（其实是一个索引，是整型数）指向当前读到的代码字符。定义一系列**枚举**，用于记录词法分析时判断出的基本单元的类型。定义一个结构体**TokenStru{ TokenID ID; int num; string word; }**，记录当前读到的基本单元。上述变量都是全局变量。

核心函数有：

**void GetToken()** ：进行词法分析，疯狂if-else判断当前单位，得出其**词性**（关键字、运算符、标识符？），值（数字、字符？）。

**void match(TokenID expecttokenid)**：语义分析的步骤之一，判断当前token的类型与预期类型expecttokenid是否匹配。如果匹配，则继续读入下一个字符；不匹配，报错。（简单粗暴）

**void error()**：记录输出位置，停止分析操作。

语法树是保存了语法分析过程的数据结构。越**深**的结点越**优先**分析处理。

### 2.算法

#### （1）枚举

定义一系列枚举，区分不同的词性。对于关键字，还需另外定义一个字典，映射枚举与关键字的一一对应关系。

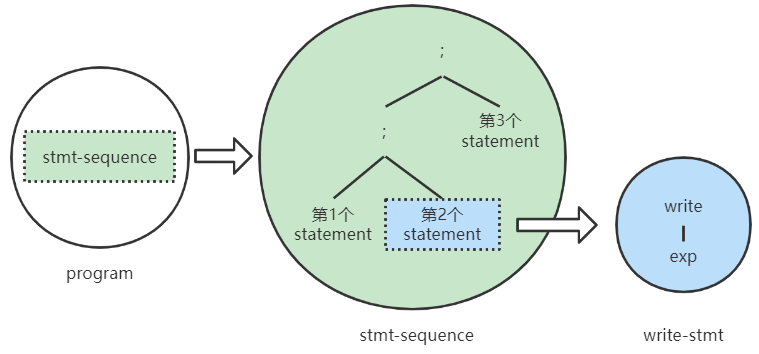
这些枚举将用于void GetToken()函数中进行词法分析。注意，tiny语言中{ }大括号内的字段为注释，在分析词性的时候要注意跳过注释。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **单元**  **类型** | **单元** | **词素**(例) | **含义** | **单元**  **类型** | **单元** | **词素**(例) | **含义** |
| 自定义符 | ID | myWord | 一般标识符 |  | SEMICOLON | ; | 分号 |
| NUMBER | 123 | 数字 | 关键字 | IF | if | 条件转移语句 |
| 运算符 | ADD | + | 算数表达式 | THEN | then |
| SUB | - | ELSE | else |
| MUL | \* | END | end |
| DIV | / | REPEAT | repeat | 循环语句 |
| SUB\_ASSIGN | -= | UNTIL | until |
| REMAIN | % | READ | read | 输入输出语句 |
| POWER | ^ | WRITE | write |
| MORE\_OR\_EQUAL | >= | 条件判断 | 报错 | ERROR |  |  |
| LESS\_OR\_EQUAL | <= |  | ENDINPUT | end |  |
| MORE | > | while (buffer[pos]==' ' || buffer[pos]=='\t' || buffer[pos]=='\n' || buffer[pos]=='{') // 遇到 空格或制表符或换行符  {  if (buffer[pos] == '{')// 遇到 注释  {  while(buffer[pos]!='}')  pos++;// 持续向前扫描  pos++;//直到扫过注释结束符  }  else  pos++; // 持续向前扫描  } | | | |
| NOT\_EQUAL | <> |
| EQUAL | == |
| ASSIGN | = | 赋值 |
| AND | and | 逻辑运算 |
| OR | or |
| NOT | not |
| REGULAR\_OR | | | 正则表达式 |
| REGULAR\_CONNECT | & |
| REGULAR\_CLOSURE | # |
| REGULAR\_ASSIGN | := |
| LBRACKET | ( |  |
| RBRACKET | ) |  |

#### （2）基础代码架构

|  |
| --- |
| program -> stmt-sequence //程序入口  stmt-sequence -> statement { **;** statement } //代码行集合  statement -> if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt | **end** //一行代码 |

分析的过程要把对应的文法规则转换为语法树，树形结构参考如下：



（若为单子结点，则赋值给左子结点，右子节点留为空指针，因为代码中的语法树遍历算法是“根左右”前序遍历）

#### （3）输入输出

|  |
| --- |
| read-stmt -> **read identifier** //读入字符  write-stmt ->**write** exp // 输出表达式 |

#### （4）赋值

|  |
| --- |
| assign-stmt -> **identifier** assign-op exp // 基本表达式的赋值  assign-stmt -> **identifier** **:=** regular-exp // 正则表达式的赋值  assign-op -> **=** | **-=** |

#### （5）算数运算

|  |
| --- |
| ~~exp -> arithmetic-exp [ comparison-op arithmetic-exp]~~  arithmetic-exp -> term { addop term }  term -> factor { mulop factor }  comop -> **-=** //复杂运算符  addop -> **+** | **-** //加减运算符  mulop -> **\*** | **/** | **%** | **^** //乘除运算符  factor -> **(** exp **)** | **number** | **identifier** // 基本算子 |

设计运算就离不开设计**运算符**，设计运算符就离不开设计运算**优先级**。越优先的运算符，越在文法规则的**深**处。

#### （6）逻辑运算

尝试设计文法如下：

|  |
| --- |
| exp -> bool-exp  bool-exp -> bterm { **or** bterm }  bterm -> mid-bfactor { **and** mid-bfactor}  mid-bfactor -> [ **not** ] bfactor  bfactor ->arithmetic-exp [ comparison-op arithmetic-exp ] // 基本算子  comparison-op -> **<** | **<=** | **==** | **>** | **>=** | **<>** // 比较运算符 |

若

exp -> arithmetic-exp [ comparison-op arithmetic-exp]

exp -> bool-exp

则exp文法存在左公因子arithmetic-exp，且

bfactor -> arithmetic-exp comparison-op arithmetic-exp

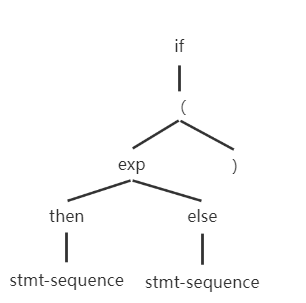
exp -> arithmetic-exp [ comparison-op arithmetic-exp]

两者的右部很像。好像我一开始就设计错了，消除左公因子算法不适用。

于是**删除文法**exp -> arithmetic-exp [ comparison-op arithmetic-exp]，表明先后顺序：**算数运算优先于逻辑运算**。

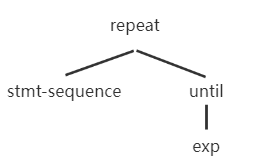
#### （7）if条件转移语句

|  |
| --- |
| if-stmt -> **if (**exp**)** **then** stmt-sequence [ **else** stmt-sequence ] |

树形结构为：

#### （8）repeat循环语句

|  |
| --- |
| repeat-stmt -> **repeat** stmt-sequence **until** exp |

树形结构为：

#### （9）正则表达式

|  |
| --- |
| ~~exp -> regular-exp~~  regular-exp -> regular-term { binary-regop regular-tterm }  regular-term -> [unary-regop] **identifer**  binary-regop -> **|** | **&**  unary-regop -> **#** |

## （二）代码架构

### 1.读取用户源程序

定义一个Loader类，专门作为读取文件的工具，因此将其设为**单例**。C++的饿汉式单例模式实现起来非常简单粗暴，用静态变量和静态函数即可。但存在线程安全问题，但不在本次作业的考虑范围内。

### 2.生成语法树

#### （1）语义分析结构

对于右部**最多**只有**一个元素**的文法，它的代码是这样的：

|  |
| --- |
| BTreeNode\* Tiny::**program**()  {  BTreeNode\* root = stmt\_sequence();  return root;  } |

其中，若有**多个**右部，则需要用到switch-case语句区分出口。注意**消除左公因子**。

|  |
| --- |
| switch (token.ID)  {  case WRITE:  node = write\_stmt();  break;  case END:  node = new BTreeNode(token);  break;  default:  error();  } |

对于右部**至少**有**一个元素**，**且存在右递归**（递增2）的文法，它的代码是这样的：

|  |
| --- |
| BTreeNode\* Tiny::**stmt\_sequence**()  {  BTreeNode\* nodex, \* nodey;  nodex = statement(); // 第1个元素是根节点  while (token.ID == SEMICOLON) // 【右递归】若存在第2个元素  {  nodey = new BTreeNode(token); // 取第2个元素生成新的根结点  nodey->lc = nodex; // 左子结点是第1个元素  match(token.ID);  nodey->rc = statement(); // 右子节点是第3个元素  nodex = nodey; // 更新根节点  }  return nodex;  } |

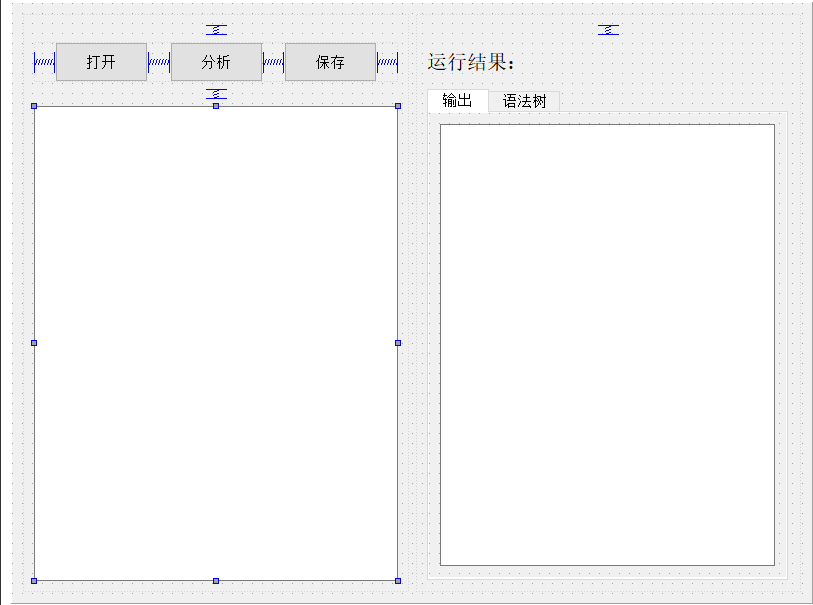
对于右部**固定**有**三个元素**（不存在右递归），则把while语句改为if-else语句，且else error()。

对于右部**至多**有**三个元素**（**存在选择项**），则把while语句改为if语句，不需要else。

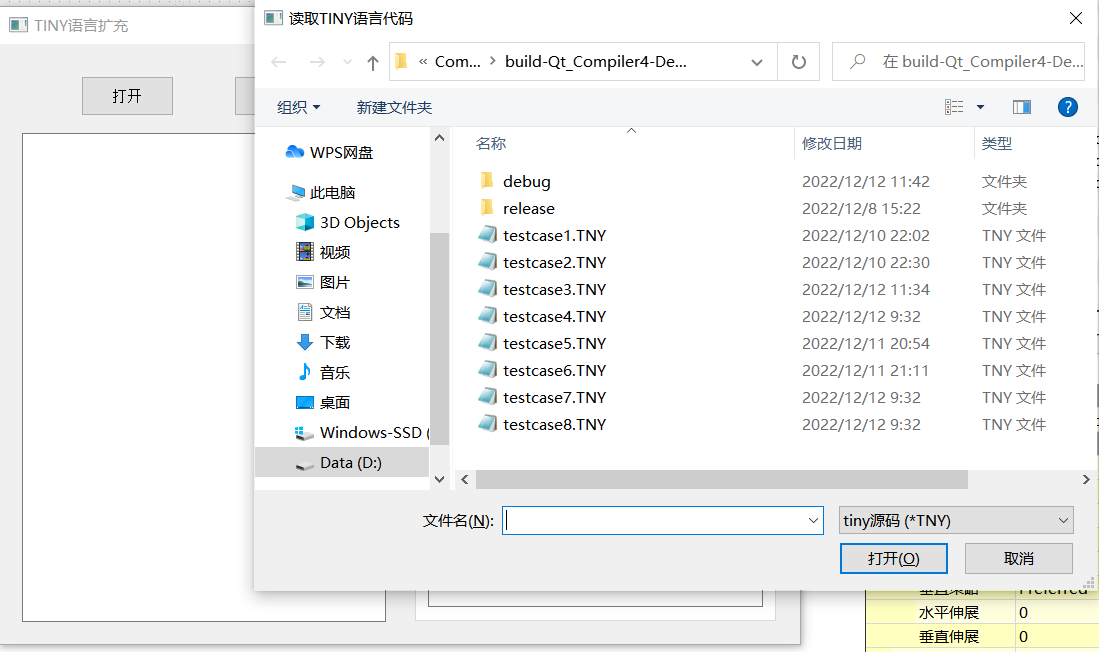
#### （2）前序遍历输出语法树

|  |
| --- |
| void Tiny::**preorder**(BTreeNode\* t, int deepth)  {  if (t == 0) return;  for(int i = 0; i < deepth; i++)  syntaxTree += " ";  if (t->data.ID == NUMBER)  syntaxTree = syntaxTree + to\_string(t->data.num) + '\n' ;  else  syntaxTree = syntaxTree + t->data.word + '\n' ;  preorder(t->lc, deepth+1);  preorder(t->rc, deepth+1);  } |

### 3.界面



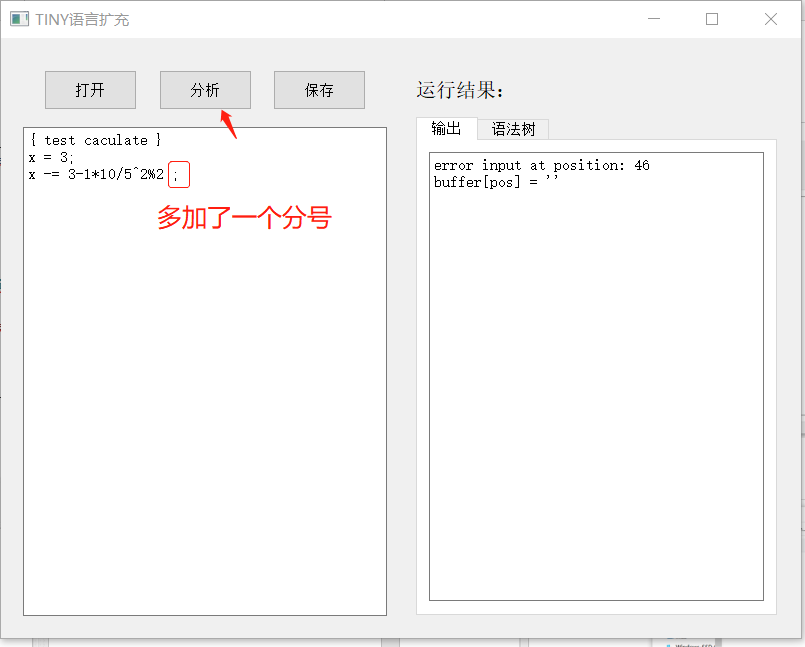
点击“打开”按钮，打开.TNY文件之后自动分析。



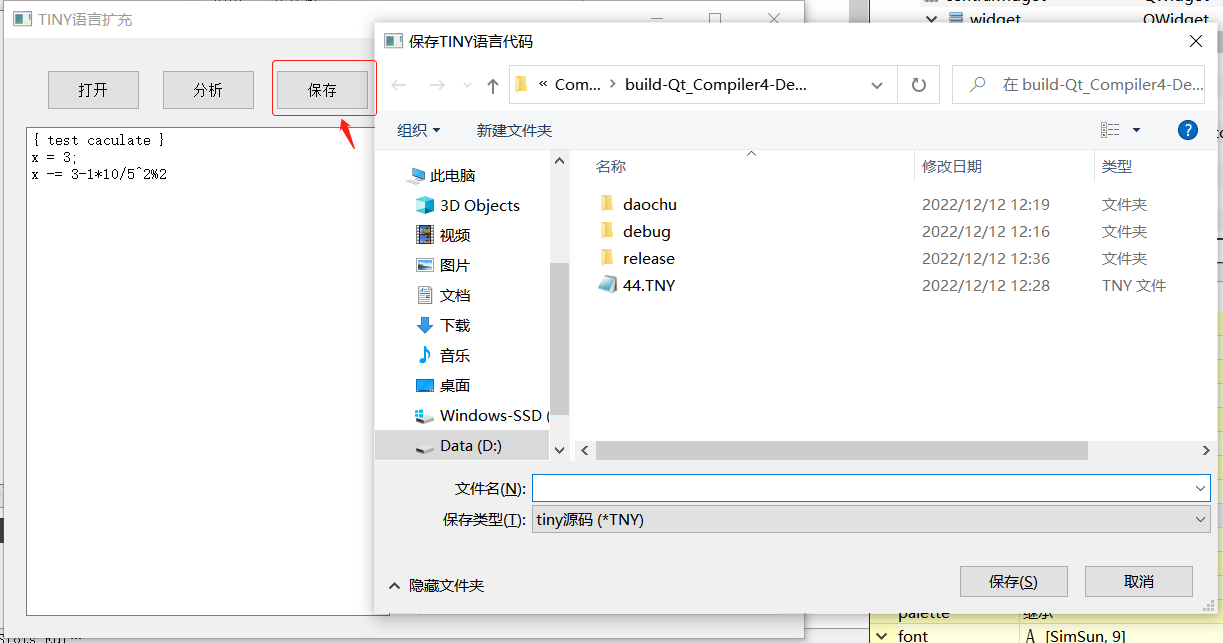
“输出”栏可查看分析过程中的错误位置。点击“语法树”一栏，可以查看语法树。



可以修改代码，立即分析。



点击“保存”按钮可保存代码文件。



## （四）测试数据

### 测试文件1：

|  |  |
| --- | --- |
| { test read and write }  read x;  write x |  |

### 测试文件2：

|  |  |  |
| --- | --- | --- |
| { test caculate }  x = 3;  x -= x+3-1\*10/5^2%2 |  |  |

### 测试文件3：

|  |  |
| --- | --- |
| { test bool-exp }  y = 4 or 0 and 1==1;  x = (5\*9<100 or not 0) and 2 |  |

### 测试文件4：

|  |  |
| --- | --- |
| { test if-statment }  If(1==2)  then write x  else read x |  |

### 测试文件5：

|  |  |
| --- | --- |
| { test repeat-statment }  repeat write x  until 1==1 |  |

### 测试文件6：

|  |  |
| --- | --- |
| { test regular-exp }  x := #a | b&c |  |

### 测试文件7：

|  |  |
| --- | --- |
| { Sample program  in TINY language -  computes factorial  }  read x; { input an integer }  if (0<x) then { don't compute if x <= 0 }  fact = 1;  repeat  fact = fact \* x;  x = x - 1  until x == 0;  write fact { output factorial of x } |  |

测试文件8：

|  |  |
| --- | --- |
| { Sample program  in TINY language -  computes factorial  }  read x; { input an integer }  if (x>0) then { don't compute if x <= 0 }  fact = 1;  repeat  fact = fact \* x;  x = x - 1  until x == 0;  write fact { output factorial of x } |  |

# 四、实验总结

通过本次实验，我加深了对编译原理过程中词法分析和语法分析的理解，并深刻地体会到了这种算法结构的健壮性、易拓展性和巧妙。

# 五、参考文献

1. <https://blog.csdn.net/StuGeek/article/details/122890698>

2.老师的参考代码“算术表达式---计算、语法树生成、汇编生成及解释执行 等源程序”