

[一、项目内容 3](#_Toc135083042)

[二、项目目的 3](#_Toc135083043)

[三、项目文档 3](#_Toc135083044)

[（一）需求分析 3](#_Toc135083045)

[（二）代码设计 4](#_Toc135083046)

[1. 算法设计 4](#_Toc135083047)

[2、架构设计 9](#_Toc135083048)

[3、数据结构设计 9](#_Toc135083049)

[4、平台实现设计 10](#_Toc135083050)

[（三）程序实现 10](#_Toc135083051)

[1、代码 10](#_Toc135083052)

[2、界面 10](#_Toc135083053)

[（四）程序测试 10](#_Toc135083054)

[1、测试用例与结果 10](#_Toc135083055)

[2、测试评估 13](#_Toc135083056)

[四、实验总结（心得体会） 14](#_Toc135083057)

[（一）编码心得 14](#_Toc135083058)

[（二）学习心得 15](#_Toc135083064)

[五、参考文献 15](#_Toc135083065)

[六、项目自评 16](#_Toc135083066)

# 一、项目内容

（1）以文本文件的方式输入某一高级程序设计语言的所有语法对应的BNF文法，因此系统需要提供一个操作界面，让用户打开某一语言的所有语法对应的BNF文法的文本文件，该文本文件的具体格式可根据自己实际的需要进行定义。

（2）需要提供窗口以便用户可以查看**文法化简**后的结果（可用表格形式进行呈现）

（3）需要提供窗口以便用户可以查看**消除左公共因子和左递归**之后的新文法（可用表格形

式进行呈现）

（4）求出改造后文法的每个非终结符号的 **First 集**合和 **Follow 集**合，并需要提供窗口以便用户可以查看该结果（可用两张表格的形式分别进行呈现）

（5）构造出 **LL(1)分析表**，并需要提供窗口以便用户可以查看该结果（可用表格形式进行

呈现）

（6）采用 LL(1)语法分析方法进行**语法分析**并生成相应的语法树，每个语句的语法树结构

可根据实际的需要进行定义。（语法树需要采用树状形式进行呈现）

（7）对系统进行测试：以 TINY 语言的所有语法以及第一项任务的测试结果 sample.lex 作

为测试，进行 LL(1)语法分析并生成对应的语法树。

（8）要求应用程序为 **Windows 界面**

（9）书写完善的软件文档

# 二、项目目的

（1）增强对文法规则的定义、处理算法的理解与运用

（2）增强对LL(1)分析表、LL(1)语法分析方法的理解与运用

（3）熟悉语法树的定义和构建方法，深化对语法分析的理解

（4）积累程序设计、开发和调试经验，积累交互式界面的设计和实现经验，熟悉软件文档的撰写和规范

# 三、项目文档

## （一）需求分析

要完成这个作业，我需要做到：

1. 使用Qt和C++，编写具有交互窗口的可执行程序。窗口需要支持打开本地文件、显示图表、显示文本框，为用户提供自主操作的按钮
2. 了解题目中各个专有名词的定义，例如BNF文法（上下文无关文法），文法化简， First 集合和 Follow 集合，LL(1)分析表，语法树
3. 了解语法分析程序的算法
4. 了解TINY语言，能够独立设计测试用例并检验程序的正确性

## （二）代码设计

### 1. 算法设计

#### 1.1. 文法规则转LL(1)分析表【语法分析】

##### 1.1.0. 预处理

读入的文法规则的各个字段（Symbol）可能很长，但在算法中，我实际处理时，我会为每个string Symbol分配特定的int ID，对ID做算法。ID完全是根据读入的Rule生成的。所以读入字符串、生成文法规则list<Rule> Grammar、生成map<int, string> ID2Word这三个过程可以**解耦**。

规定：ID2Word[100] = "epslion";TempID2Word[101] = "$";

##### 1.1.1.化简文法

化简文法就是去除其中的无效规则，其中无效规则包括了有害规则和多余规则。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **分类** | | **定义** | **步骤** | **举例** |
| 有害  规则 | | 导致文法出现二义性的规则。 | 直接找左部符号和右部符号一样的文法规则结构体，删了。【比较】 | A->A，B->B |
| 多余规则 | 不可到达规则 | 文法中任何句子的推导都不会用到的规则 | 将S放入“可到达集合”中。从S开始遍历每个文法，若左部在集合中，则将右部的每个非终结符也放入“可到达集合”中。再次遍历每个文法，若左部不在集合中，则删去。【并查集】 |  |
| 不可终止规则 | ①从开始符号为左部起步，遍历其对应的每个文法。设当前左部符号为A，检查是否满足A->αAβ形式。 | S->Be，B->Ce，B->Af，A->e ，A->Ae，A->E，E->e，C->Cf，D->f  处理后： |
| ②若满足，则标记该文法，查找以A为左部、可终止的文法。若不存在可终止，则删除所有以A为左部，或右部含有A的文法，返回。 |
| ③若不满足，且右部存在非终结符B，则以B为左部重复步骤①②。 |

##### 1.1.2 消除左公因子

左公因子就是两个或多个文法规则共享的通用前缀串。判断是否存在左公因子的**依据**是，右部的第一个符号的**first集是否存在交集**。其依赖关系如下：



因此，消除左公因子应该在消除左递归之后进行。

当右部的第一个符号是终结符和非终结符时，左公因子分为直接左公因子和间接左公因子。

消除左公因子的**目的**是避免在递归向下分析时产生回溯。例如下一个字符是a，如果A->ab且A->ac，语法分析过程将不知选择哪个文法继续，这会导致多次选择（回溯）。

|  |  |  |  |
| --- | --- | --- | --- |
| **左公因子** | | **化简后** | **备注** |
| 直接左公因子 | A->**a**b，A->**a**c | A -> aA’, A’ -> b, A’ -> c |  |
| B->**ac**mm，B->**ac**d | B -> acB’, B’ -> mm, B’ -> d | 手工 |
| B->cB’, B’->cB’’, B’’->mm, B’’->d | 代码 |
| 间接左公因子 | A -> Be, A->Cf, B->**a**b, B->**b**, C->**a**c, C->**b** | A->**a**A’, A’->be, A’->cf, A->**b**A’’, A’’->ε | 手工 |
| A->**a**A’, A->**b**A’, A’->B’e, A’->C’f, B->ab, B->b, C->ac, C->b, B’->b, B’->ε, C’->c, C’->ε | 代码 |

举例

|  |  |
| --- | --- |
| **手工算法** | **代码算法** |
| ①找出左部相同的一批文法，并求出对应的first集。 | ①遍历每个非终结符i，记录以i为左部的所有文法，并求出对应的first集 |
| （消除间接左公因子）  ②若first集之间存在交集，则不断**代入**产生新文法，再次求first集，直到first集之间要么完全相等，要么完全不等。  ③若first集之间相等，则不断代入产生新文法，直到满足相等的first集对应的文法的右部第一个符号也是相同的。  例如A->Db, A->Dc | ② |
| （消除直接左公因子）  ④**新增**文法A->DA’，**将**A->Db改为A’->b，将A->Dc改为A’->c | ③采用并查集，遍历以Vni为左部的文法。值为-2代表这是父节点，-1代表不存在公因子，自然数代表该子节点的父节点的索引。  ④遍历并查集，若发现union[Index 1]=-2，找到第Index 1个文法，其原形式为A->aB，新增非终结符**C**，将A->aB改为A->aC，新增C->B。找到所有union[…]=n的文法A->aB’，改为C->B’。 |
| 重复执行上述两步，直到遍历所有非终结符 |  |

##### 1.1.3 消除左递归

文法经过一次或多次推导之后，出现如下形式**A**->**A**α，则称该文法是左递归的。左递归会在自顶向下分析的过程中产生不好的回溯。

①直接左递归：经过一次推导就可以看出文法存在左递归。如A->Aα|β。递归结果为A->Aα->Aαα->Aααα（总在左边增加α）=>βααα= β{α}（总在右边增加α）

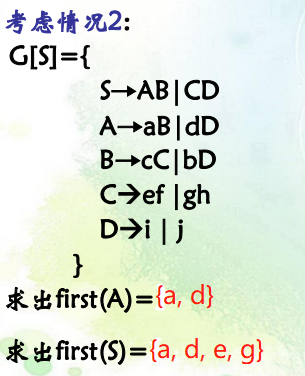
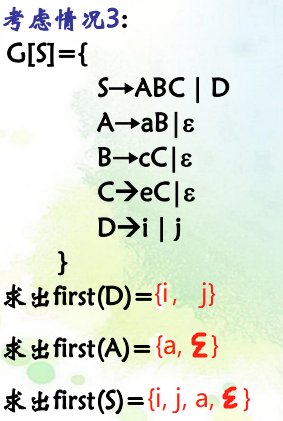
②间接左递归：需多次推导才可以看出文法存在左递归。如文法：S->Qc | c，Q->Rb | b，R->Sa | a，有S->Qc->Rbc->Sabc。

|  |  |  |
| --- | --- | --- |
| **手工算法** | **举例** | **代码算法** |
|  |  | ①遍历每个非终结符号，设当前符号为Vni  ②遍历以Vnj为左部的文法，检查是否存在左递归 |
| ①举一个最简单的例子，把A->Aa | β直接改为A->βA’，A’ ->aA’ | ε（右递归）。  更一般化的形如P → P**X** | **Y**（其中X和Y看作一个整体，比如：P → Pabc|ab|b，X就是abc，Y就是ab|b），可改写为P->**Y**P'，P'->**X**P' | **ε**。 | A -> A a  A -> b  改为右递归：  A -> b A’  A’ -> a A’  A’ -> ε | ④判断Vni是否存在形式为A→ AX的文法。若存在，则新增非终结符A’，将文法A->AX修改为文法A’->XA’，新增文法A’->ε。遍历其它以Vni为左部的文法，将A’追加到最右部。（消除直接左递归） |
| ②将消除直接左递归后的新文法**代入**未解决的文法中（即间接左递归），得到新的直接左递归，按照步骤①再次消除 |  | ③若存在间接左递归，且右部第一个符号与左部不同，说明是间接左递归，则代入。（消除间接左递归） |
| ③反复实施，直到不可代入 |  |

考虑一种边界情况：A -> A a，A -> ε，则消除左递归结果应为A -> A’，A’ -> a A’，A’ -> ε，也就是说，在右部追加A’时，要检查右部是否为ε。

##### 1.1.4 求出First集合

First集合是可以从**非**终结符号或符号串X**推导出**的所有串首**终结符**构成的**集合**。如果X->∗ε，那么 ε也在FIRST（X）中。举例：

|  |  |
| --- | --- |
| **手工算法** | **代码算法** |
|  | 定义【递归】函数GetFirst(int X) |
| 以X为目标，从右部符号串取第一个字符a | 遍历以X为左部的文法，取文法右部第一个字符假设为a |
| 若a是终结符或ε，则填入Follow集中 | 若a是终结符或ε，则填入当前Follow集中，返回Follow集 |
| 若a是非终结符，则继续重复上述步骤，直到其右部符号串的**串首**是终结符或ε | 若为非终结符且不为X（**避免进入死循环**），则递归进入GetFirst(int a) |

##### 1.1.5 求出Follow集合

给定一个在右部的非终结符A，则Follow(A)为**紧跟**在其后的每个**终结符号**或$（右端结束标记）的**集合**。举例：

|  |  |  |
| --- | --- | --- |
| **文法规则** | **Follow集** | **参考资料** |
| E -> TE’  E’-> +TE’| ε  T -> FT’  T’-> \*FT’| ε  F -> (E) | a | Follow(E)={ $, ) }  Follow(E’)={ $, ) }  Follow(T)={ +, $, ) }  Follow(T’)={ +, $, ) }  Follow(F)={ \*, +, $, ) } | [编译原理中Follow集的求法\_杨博东的博客的博客-CSDN博客\_follow集](https://blog.csdn.net/yangbodong22011/article/details/52950436?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-0-52950436-blog-106379883.pc_relevant_recovery_v2&spm=1001.2101.3001.4242.1&utm_relevant_index=3) |
| S→ABc  A→a|ε  B→b|ε | Follow（S）=｛＃｝  Follow（A）=｛b，c｝  Follow（B）=｛c｝ | [编译原理 First集 Follow集 select集 通俗易懂的讲解 + 实例\_CooperNiu的博客-CSDN博客\_select集](https://blog.csdn.net/CooperNiu/article/details/78524688?spm=1001.2101.3001.6650.6&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-6-78524688-blog-115911002.pc_relevant_aa2&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-6-78524688-blog-115911002.pc_relevant_aa2&utm_relevant_index=7) |
| S→AB  S→bC  A→ε  A→b  B→ε  B→aD  C→AD  C→b  D→aS  D→cz | FOLLOW(S)={ $ }  FOLLOW(A)= { a, $, c }  FOLLOW(B)= { $ }  FOLLOW(C)={ $ }  FOLLOW(D)={ $ } | [判断LL(1)文法（first集、follow集、select集）\_内存不足°的博客-CSDN博客\_判断ll(1)文法](https://blog.csdn.net/qq_44922497/article/details/111880076?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-4-111880076-blog-78524688.pc_relevant_recovery_v2&spm=1001.2101.3001.4242.3&utm_relevant_index=7) |

依照这个特性，要求文法规则不能存在形如A -> A B, B -> ε的直接或间接左递归。

|  |  |  |
| --- | --- | --- |
| **手工算法** | **解释** | **代码算法** |
| 不断应用下面的规则，直到再没有新的终结符号可以被加入到任意的follow集合中为止： |  | 定义【递归】函数GetFollow(int X)。遍历每个文法： |
| ①将 **$** 放到follow（S）中，其中S是文法的开始符号。 |  | ①若X是文法开始符号，则将$填入当前Follow集中。 |
| ②当A是最右部的时候，将 **$** 加入到follow(A)中 |  | ②若X在文法最右部，则将$填入当前Follow集中。 |
| ③如果存在一个产生式A→α**B**β，那么follow（B）**包含first（β）-ε**。 | follow(B)是求跟在B后的终结符或$组成的集合，因此对于跟在B后的β，它的first集合就是follow(B)的子集 | ③若存在A->αXβ形式，则将GetFirst(β)-ε填入当前Follow集中。 |
| ④如果存在一个产生式A→α**B**，或存在产生式A→α**B**β且first（β）包含ε，那么follow（B）**包含follow（A）**。 | 对于A→α**B**β,且β多步推导出ε ，那么可以用αB替换A, B后面紧跟的字符就是A后面紧跟的字符 | 若GetFirst(β)包含ε，则将GetFollow(A) 填入当前Follow集中 |
| ④若存在A->αX形式，即X在文法最右部，则将GetFollow(A) 填入当前Follow集中。注意A≠X，避免死循环。 |

##### 1.1.6 生成LL(1)分析表

若BNF文法是LL(1)文法，则同时满足以下**条件**：①对于相同的左部，其右部的first集都没有交集；【消除左公因子】；②若每个非终结符A的first集都包含了ε，则first(A)∩follow(A) = Ø。【消除左递归】

步骤：①遍历每条文法，①对于First(α)中的每个记号a，都将A->α添加到项目M[A,a]中。②若**ε**在First(a)中，则对于Follow(A) 的每个元素a（记号或是$），都将A->α添加到M[A,a]中。

##### 1.1.7 其它整理

|  |  |  |
| --- | --- | --- |
| 过程 | 是否删除字符 | 是否增加字符 |
| 化简文法 | 是 | 否 |
| 消除左公因子 | 否 | 是 |
| 消除左递归 | 否 | 是 |

注意事项：初始化时、当有字符被删除时，最好调用OrganizeID2Word()。当有字符增加时（文法增加）时，调用SortGrammar()。

#### 1.2. 根据分析表自动分析代码，并生成语法树【语法分析】

##### 1.2.1 语法树

一般的语法树，以文法左部为树的根节点，文法右部为树的子节点。

抽象语法树，全部以非终结符作为树的节点，其中文法右部的第一个非终结符是根节点，其余非终结符是子节点。

|  |  |  |  |
| --- | --- | --- | --- |
| 文法规则 | 读入字符串 | 一般语法树 | 抽象语法树 |
| S -> A a A  A -> B  A -> a  B -> b A | aaba |  |  |
|  |  |  |  |

在实际测试的过程中，可能出现应为兄弟关系的两个节点，变成了父子关系的两个节点，这是因为消除左公因子的时候把右部的连贯性打断了。s

##### 1.2.2 语法分析

程序读入的是源代码，但做词法分析的时候是围绕int ID，也就是文法中的所有终结符和非终结符的索引号做算法的。

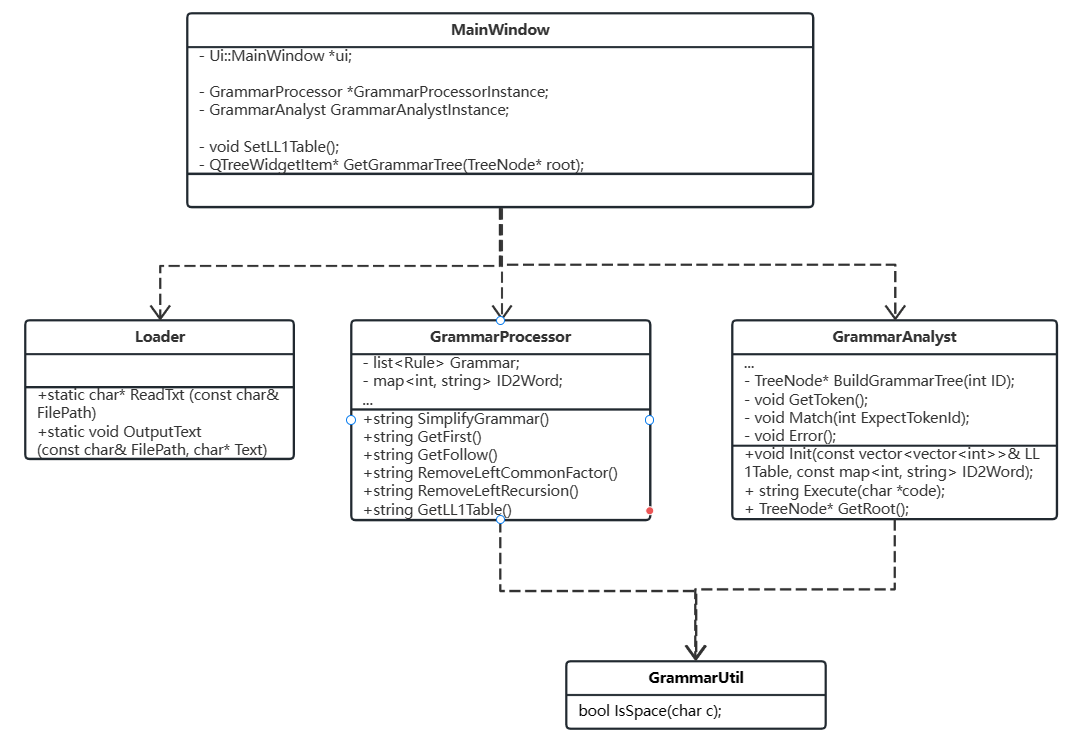
1. 读入源码，取当前字符Buffer[Pos]，并以此匹配终结符，得到Token。如果任何非终结符都不能匹配，则报错。

2. 执行递归。从起始符号S开始，根据非终结符a所在的列，查询LL1分析表，得到文法规则，假设为S -> A B。以A所在的行、非终结符所在的列再次查表，直到遇到终结符，则执行match函数。若匹配，则返回，查询B；若不匹配，则报错。

会存在得不到文法规则的情况：例如S -> T A, T -> epsilon, T -> b, A -> a，当以T所在的行、a所在的列查表时，会直接返回空值，进入对A的查询。

3. 在递归过程中构造语法树，以文法的**非终结符**为根节点，终结符所对应的**代码中的字符**为子节点，最后返回树的根节点。

### 2、架构设计



### 3、数据结构设计

#### （1）BNF文法规则

巴科斯范式（BNF）所描述的语法是与上下文无关的。其语法为：在双引号中的字("word")代表着这些字符本身。而double\_quote用来代表双引号。在双引号外的字（有可能有下划线）代表着语法部分。尖括号( < > )内包含的为必选项。方括号( [ ] )内包含的为可选项。大括号( { } )内包含的为可重复0至无数次的项。括号 () 表示分组的意思。竖线( | )表示在其左右两边任选一项，相当于"OR"的意思。::= 是“被定义为”的意思。

例如，TINY语言的stmt-sequence的文法，用BNF表示为：stmt-sequence := statement { ; statement }，用一般文法规则表示为：stmt-sequence -> stmt-sequence ; stmt-sequence | stmt-sequence。**虽然格式不一样，但含义是一样的**。

因为历史遗留问题，我选择了**后者**的格式。

|  |
| --- |
| struct **Rule**{  int left;  vector<int> right;  }; |

#### （2）Token

语法树上的每一个节点称为一个Token。

定义Token结构体如下：

|  |
| --- |
| struct {  int ID;  string Word;  } Token; |

（3）其它规定

number必为数字，identifier必为标识符，不可设为其它非终结符。

### 4、平台实现设计

运行环境：Windows x64

开发语言：C++ 11

编码工具：Qt 4.12.2，Visual Studio 2022

## （三）程序实现

### 1. 代码

### 2. 界面

### 3. 迭代

#### 3.1. erase和指针--

|  |  |  |
| --- | --- | --- |
|  | **代码** | **解释** |
| **迭代前** | for(int i = 0; i < Grammar.size(); i++, it++)  if(Reachable[(\*it).Left] == false)  {  it = Grammar.erase(it); // 删除不可达的文法规则  it--;  } | 代码存在漏洞：当it=Grammar.begin()时，erase和it--会导致指针越界。  写代码时应该避免指针回退，这是少见的操作。如果要回退，注意使用it!=.begin()保护 |
| **迭代后** | for (int i = 0; i < Grammar.size(); i++)  {  if (Reachable[(\*it).Left] == false)  it = Grammar.erase(it); // 删除不可达的文法规则  else  it++;  } |

#### 3.2. if条件句排版

|  |  |  |
| --- | --- | --- |
|  | **代码** | **解释** |
| **迭代前** | for(int i = 0; i < Grammar.size(); i++)  if(a)  处理1  for(int j = 0; j…)  if(b)  处理2 | 整个语法分析算法就是一堆的暴力查询，其中大量的for循环和if句叠加，会让代码很宽很“胖”，使不雅观。并且debug时跳跃太大.  更优雅的写法是善用continue。 |
| **迭代后** | for(int i = 0; i < Grammar.size(); i++)  if(!a)  continue;  处理1  for(int j = 0; j…)  if(b)  continue;  处理2 |

## （四）程序测试

### 1、测试用例与结果

输入要求：左部和右部之间用“ -> ”连接。文法文本中不得出现除了空格、换行符、制表符以外的间隔符，文法中不得出现或运算”|”，或者其它BNF标准里的处理符号，因为写这样的转化代码意义小成本大。

#### （1）算法测试

…\ComprehensiveExp2\TestData\data1.txt

测试：消除有害规则（G->G）、不可达规则（D->f）、不可终止规则（B->Ce、C->Cf），消除左公因子（B->eb、B->E、E->e）。

|  |
| --- |
| S -> B e  B -> C e  B -> A f  B -> e b  B -> E  A -> E  A -> d e  E -> e  C -> C f  G -> G  D -> f |

…\ComprehensiveExp2\TestData\data2.txt

测试：消除左公因子（A->dB、A->dC）

|  |  |  |
| --- | --- | --- |
| **测试用例** | **预期结果** | **实际结果** |
| A -> d e B  A -> d e C  A -> d B  A -> a  B -> c d  C -> e |  |  |

…\ComprehensiveExp2\TestData\data3.txt

测试：消除左递归。

|  |
| --- |
| A -> A a  A -> B b c  A -> c  B -> A e f  B -> g A h  B -> a  C -> c |

#### （2）TINY

|  |
| --- |
| program -> stmt-sequence  stmt-sequence -> stmt-sequence **;** statement  stmt-sequence -> statement  statement -> if-stmt  statement -> repeat-stmt  statement -> assign-stmt  statement -> read-stmt  statement -> write-stmt  if-stmt -> if exp then stmt-sequence end  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-exp simple-exp  exp -> simple-exp  comparison-exp -> <  comparison-exp -> =  simple-exp -> simple-exp addop term  simple-exp -> term  addop -> +  addop -> -  term -> term mulop factor  term -> factor  mulop -> \*  mulop -> /  factor -> ( exp )  factor -> number  factor -> identifier |

### 2、测试评估

三、项目文档：将按软件工程规范书写的文档附加在这里。该部分应该有详细的项目分析、设计、实现及测试内容，例如数据结构的选择、关键算法的设计方案等。阐述时，应该尽量使用文字或图表的方式而不是简单地把项目的源代码粘贴进去，确保源代码的行数要比非源代码的行数要少；这里也不能只是粘贴大量的测试结果图片[在上交的报告书中应把这些红色字删除]

# 四、实验总结（心得体会）

## （一）编码心得

### 1.中文路径问题

在项目一的过程中，我成功实现了读取中文路径，不乱码，但突然有一天，这个代码就一直无法读取中文路径下的文件。我遇到了和[这篇博客](https://blog.csdn.net/qq_40015157/article/details/120350871)一样的问题。考虑到实现成本太高，而且不重要，我跳过了这个功能。所以本项目不可读取中文路径。

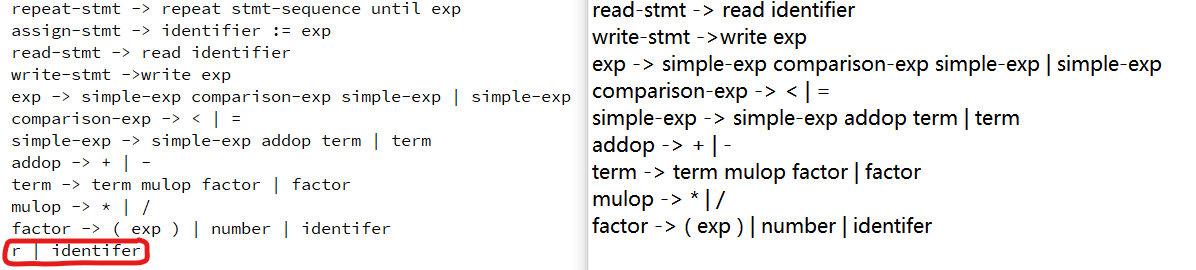
另外，当单正斜杠路径/转化为双反斜杠路径\\时，使用QDir::toNativeSeparators()只能得到单反斜杠结果\，这样书写的路径是非法的。不知为何这个函数功能要这样安排。

于是我抄到了这段代码：

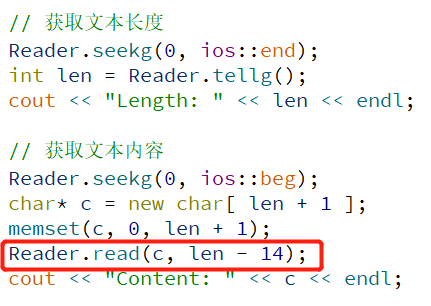
|  |
| --- |
| #ifdef Q\_OS\_LINUX  FileName = FileName.replace("\\\\", "/");  #endif  #ifdef Q\_OS\_WIN32  FileName = FileName.replace("/", "\\\\");  #endif |

### 2.文件流问题

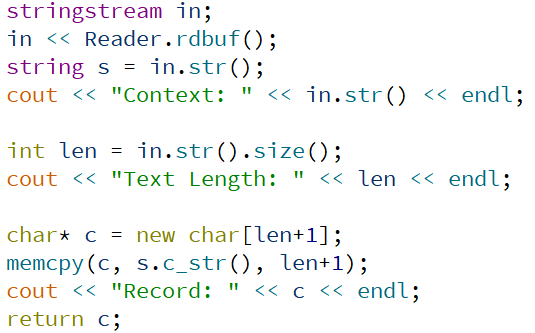
在读取文本文件时，会莫名其妙多出一些不存在的字符。问题暂未解决。



经过排查，发现需要将读取的字符串长度减14。原因不明。



经过我长达半个月的断断续续的debug，我发现应该是直接使用ifstream的读取方式不对，它总是会不确定性的读超。因此引入sstream，借助string完成字符串的创建。（曲线救国）



### 3.遍历问题

遍历容器一般有两种方法：①iterator迭代器；②for(int i = 0; i < size; i++){容器[i]…}。C++ 11的新特性for(auto it : container)的思想和迭代器差不多。三者性能相近，但如果需要在遍历的过程中剔除某个元素，则最好用②。

参考资料：[C++ Vector遍历的几种方式及性能对比\_蜗牛lx的博客-CSDN博客](https://blog.csdn.net/woniulx2014/article/details/83176360)

### 4. 控制台打印问题

因为觉得Qt的debug环境不友好，所以我尽量用c++原生语法写算法，这样即使出大问题也可以搬回VS 2022里调试。

在Qt，cout经常会不输出，因为它是先存储到缓冲区，等满了再输出。可是当出现crash时，会导致几乎没有任何输出，程序就终止了。因此被迫用上cout.flush()。

### 5. 调试效率问题

Qt调试实在是太不智能了，很多返回值不显示，导致调试效率低。写到消除间接左公因子的时候简直两眼一黑，就把算法代码搬回VS里测试了。

愿来生没有Qt。

## （二）学习心得

通过本次大作业的学习，我深刻地了解到了语法分析的重要性和复杂性。1、语法分析是将源代码转化为抽象语法树的过程，也是之后进行语义分析和代码生成的基础。2、在学习语法分析的过程中，我必须深入了解正则表达式和上下文无关语法的概念，对文法处理算法越发熟练。3、通过这个项目的实践，我对编译原理有了更深入的理解和掌握，也获得了更多的编程技能。

# 五、参考文献

1、https://blog.csdn.net/XMPTFQ/article/details/111412023

2、老师的课件

# 六、项目自评

1. 项目完成情况

（1）完成了对文法规则的处理，包括化简文法、消除左递归（直接、间接）、消除左公因子（直接、间接）。

（2）完成了语法分析任务，包括生成LL1分析表、生成语法树。

（3）实现交互界面，编写了详细的使用说明、测试文档、设计文档。

2. 自评分数

95

3. 原因说明

（1）独立完成：我在整个项目过程中全部独立完成了编译器的设计和实现，证明了我对编译理论的深刻理解和对编程技能的熟练掌握。

（2）良好的文档：我书写的实验文档详细描述了语法分析器器的结构和工作原理，体现了我的思考和对语法分析知识的深入理解。

（3）优秀的代码质量：所有代码都是由我自己编写的，我不断重构、优化代码以提高效率和可读性，并在完成作业的过程中有意识地养成良好的编程习惯和规范。

（4）测试充分：我设计了一系列的测试用例，对我的代码进行了充分的测试和验证，保证了代码的正确性和鲁棒性。