

同济大学计算机系

## 编译原理课程语法分析器报告



小组成员：陈开煦 何征昊 黎可杰

# 目录

<b>1</b>	<b>需求分析</b>	<b>4</b>
1.1	程序功能 . . . . .	4
1.2	程序任务输入 . . . . .	4
1.3	程序任务输出 . . . . .	5
<b>2</b>	<b>概要设计</b>	<b>5</b>
2.1	主程序流程 . . . . .	5
2.2	模块之间的调用关系 . . . . .	6
2.3	词法分析 . . . . .	7
2.3.1	具体流程总纲 . . . . .	7
2.3.2	数据类型的定义 . . . . .	8
2.3.3	类 C 程序文件的读入 . . . . .	9
2.3.4	调用五类 DFA 进行 Token 的识别 . . . . .	9
2.3.5	将识别的 Token 串与相关信息输出到相应文件 . . . . .	9
2.3.6	模块间的调用关系 . . . . .	10
2.4	语法分析 . . . . .	10
2.4.1	语法文件的读入 . . . . .	10
2.4.2	FIRST 集合的构建 . . . . .	11
2.4.3	项目集规范族 DFA 的建立 . . . . .	12
2.4.4	ACTION-GOTO 表的建立 . . . . .	13
2.4.5	移进规约过程 . . . . .	14
2.5	前端 UI 设计 . . . . .	15
2.6	三个模块之间的数据交互 . . . . .	16
<b>3</b>	<b>详细设计</b>	<b>16</b>
3.1	词法分析 . . . . .	16
3.2	语法分析 . . . . .	19
<b>4</b>	<b>调试分析</b>	<b>21</b>

4.1	基本功能调试分析 . . . . .	21
4.1.1	测试数据以及测试结果 . . . . .	22
4.1.2	时间复杂度分析 . . . . .	29
4.2	调试问题与解决 . . . . .	31
4.2.1	词法分析 . . . . .	31
4.2.2	语法分析 . . . . .	31
<b>5</b>	<b>程序使用说明</b>	<b>32</b>
<b>6</b>	<b>总结与收获</b>	<b>37</b>
	<b>参考文献</b>	<b>38</b>

# 1 需求分析

## 1.1 程序功能

本次实践要求我们根据 LR(1) 分析方法，编写一个类 C 语言的语法分析程序。我们需要一个文法文件输入给程序，由程序自动生成 ACTION-GOTO 表，对指定的源程序进行词法分析和语法分析，输出分析结果、分析过程和语法树。

## 1.2 程序任务输入

程序输入: 一套类 C 语言的文法规则和一个类 C 语言代码的文本文件。

文法支持的 token 表如下

token	类型	值
int	保留字	int
double	保留字	double
char	保留字	char
float	保留字	float
break	保留字	break
continue	保留字	continue
while	保留字	while
if	保留字	if
else	保留字	else
for	保留字	for
return	保留字	return
function	保留字	function
identifier	标识符	由字母，下划线和数字 (0-9) 组成，首字符不能是数字 支持整型数字和科学计数法的数字
number	数字	
+	算术运算符	+
-	算术运算符	-
*	算术运算符	*
/	算术运算符	/
%	算术运算符	%
>	关系运算符	>
<	关系运算符	<
>=	关系运算符	>=
<=	关系运算符	<=
^	位运算符	^

&	位运算符	&
,	界符	,
;	界符	;
(	界符	(
)	界符	)
{	界符	{
}	界符	}

文法的语法部分共包含 78 个产生式，涉及 31 个非终结符，在报告中不再给出，以文本形式存储在了可执行文件同级目录下的 *grammer.syn* 中。

### 1.3 程序任务输出

程序需要以图形化界面的方式对词法分析和语法分析的结果进行展示。假如词法分析成功，显示分析成功的 token 串；假如语法分析成功，我们需要显示移进规约过程和对应的语法树。

## 2 概要设计

首先，我们将任务大致地划分为了词法分析、语法分析和前端 UI 设计三个部分。接下来，我们找到了一份长度适中的语法及其对应的词法，并以此作为分析源程序的基础。

程序的设计过程采用了前后端分离的思想和模块化的思想。我们将开发过程分为前端和后端两部分，后端主要完成词法分析和语法分析的部分，前端主要完成用户交互和功能展示的功能。前后端之间的数据传递主要通过文件来进行。

后端部分采用 C++ 11 语言，在 *VisualStudio* IDE 上开发，也使用了一小部分 C++ 17 的特性。前端部分采用 *Qt* 5.9.9 框架，基于 *QCreator* IDE 开发。

### 2.1 主程序流程

从用户使用的角度来看，本程序的主程序流程如下图：

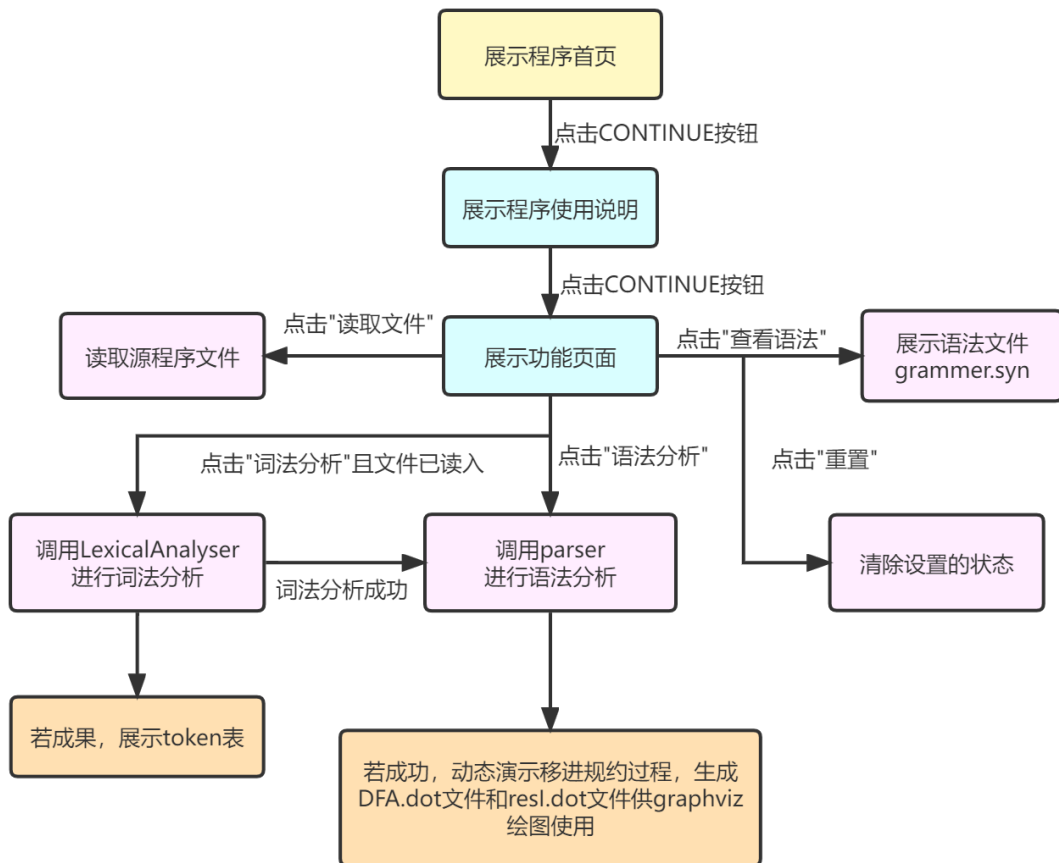


图 1: 主程序流程图

## 2.2 模块之间的调用关系

本程序模块之间的调用关系较为简单，我们将所有词法分析的部分封装成了 *LexicalAnalyser* 类，将语法分析的部分封装为了 *parser* 类。*LexicalAnalyser* 类在完成语法分析之后会生成一个包含 token 串的文件，而 *parser* 类会对这个文件进行读取以完成语法分析的步骤。这两个类分别对外提供了一个接口以供前端部分在获取用户的对应指令之后调用，分析的返回值将以函数的返回值和相关中间文件来供前端部分解析输出。

接下来，我们将对词法分析、语法分析以及前端 UI 设计三个部分的设计做说明，包括每个部分的流程，具体使用的算法和完成的任务等。在本节末尾，我们会给出这三个模块之间的数据交互示意图。

## 2.3 词法分析

词法分析过程有着较为严格的一套流程，我们据此将词法分析分解为从文件中读入类 C 程序并程序预处理得到去掉注释，统一非换行空白符的类 C 程序字符串、调用五类识别 Token 的 DFA 进行 Token 的识别，将识别得到的 Token 串与可能的报错信息输出到相应文件等多个顺序执行的任务，而在这些顺序执行的任务之前，还前置了一些任务，比如在进行词法分析前，我们需要数据类型的定义；根据语法文件的规则将可以被识别的保留字 ReservedWord、界符 Delimiter 与算符 Operator 分别进行存储。接下来，我们将先介绍具体流程，再逐个对这些子任务进行介绍。

### 2.3.1 具体流程总纲

**程序预处理** 目的：编译预处理，剔除无用的字符和注释。

1. 若为单行注释“//”，则去除注释后面的东西，直至遇到回车换行；
2. 若为多行注释“/\*...\*/”则去除该内容；
3. 若出现无用字符，则过滤；否则加载；最终产生净化之后的源程序。

**程序总体实现思想与流程** 以 DFA 以及状态转移作为词法分析器的总纲。

- 1、首先进入总控程序总的大类 LexicalAnalyzer。
- 2、调用 preprocess 函数对读入的程序进行预处理，即去掉注释等，若有报错则会报错，若无报错则进入 token 的识别。
- 3、初始化行列，start,end 指针位置信息。
- 4、每次读入一个字符，首先进行换行、空格的去除，然后对其进行预判断，即判断它是否能进入某个类别 DFA，例如根据当前字符‘{’就可以知道当进入界符的 DFA 进行识别，而不会进入其他的 DFA。
- 5、而在进入 DFA 之前会将 start 置为 end+1，即将原来的区间 [start,end] 转为 [end+1,?]，然后每个 DFA 在内部会进行更为细致的判断，若识别成功，则会更新 end 指针指向当前识别出 token 的末尾，否则不该变 end 的值，并返回报错的位置信息及具体类别与值。
- 6、若 DFA 未识别完而输入程序字符串已经读完，同样参照上条报错。
- 7、若成功识别出整个程序的所有 token，则将信息输出到 Token\_result.txt 文件里，留给语法分析程序读取，并返回分析成功信号；若中间有出错，则词法分析程序结束，把报错信息输出到 WrongInfo.txt 文件里，并返回错误信号。

词法分析程序总体框图如下：



**出错处理** 当出现程序预处理错误，程序无法分析的情况，词法分析程序会在终端终端提示词法分析失败，并输出出错定位及出错信息。

当出现词法错误，程序无法分析的情况，词法分析程序会在程序终端提示词法分析失败，并输出出错定位及出错信息。

具体实现思路在于按照一定的判别顺序判断读入的信息是否是关键字、标识符、数字、符号；若这些均无法识别，则将读入的信息判别为错误。在此大基础上，在各自判断类别内特判一些个别错误（如开头读入数字，判断后续是否还出现字符等）。

### 2.3.2 数据类型的定义

考虑到实际上词法分析主要是识别出每个 token，并且采用的是 dfa 的转移方法，所以根据识别出 token 的类别划分出保留字 *ReservedWord*，标识符 *Identifier*，数字 *Digit*，界符 *Delimiter*，算符 *Operator*，并以此生成对应五个类，分别为：

*ReservedWordAnalyzer*, *IdentifierAnalyzer*, *DigitAnalyzer*, *DelimiterAnalyzer* 以及 *OperatorAnalyzer*，而考虑到调用的便捷统一性以及类功能的类似性，统一封装成了一个虚类 *AbstractDFA*，调用统一的虚函数。

```
1 virtual int isAccepted(const std::string &str, unsigned int &start, unsigned
    int &end, unsigned int &line, unsigned int &col);
```

总体关系如下图。



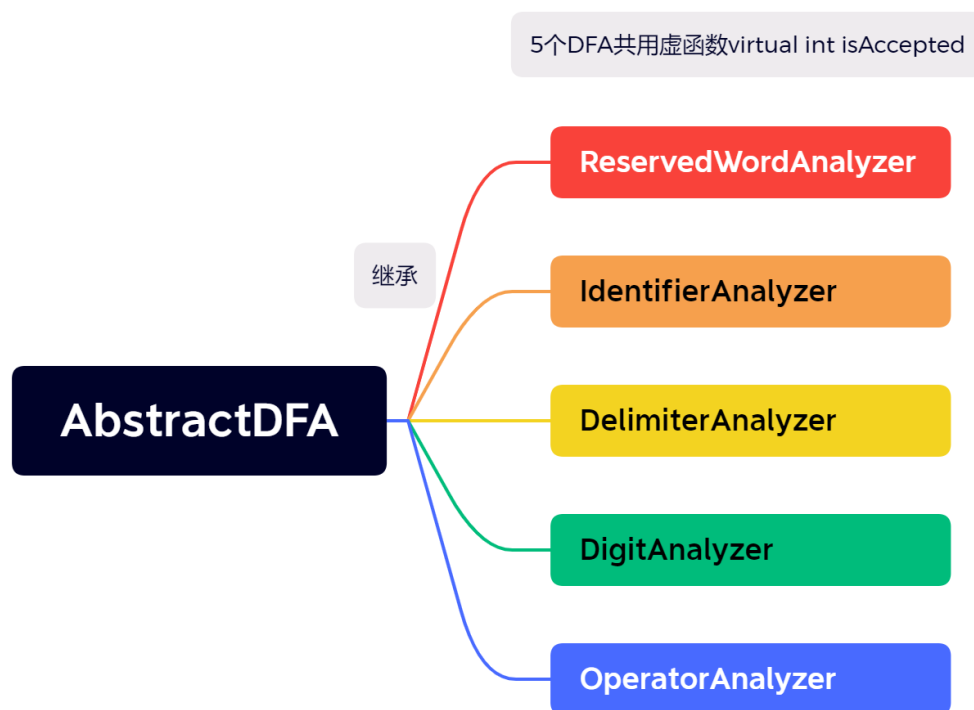


图 3: 词法分析总体关系

对于识别的定位，采用将整个输入程序用一个 `std::string` 类型变量存储，采用 `int start,int end` 模拟两个指针用于定位词法分析器当前识别位置，并且采用 `int line,int col` 记录当前读到的行列数，主要用于报错的位置定位。

### 2.3.3 类 C 程序文件的读入

采用文件读入，然后用字符串流直接将缓冲区的内容转成字符串，传进函数。

### 2.3.4 调用五类 DFA 进行 Token 的识别

依次读入字符，并调用 `ReservedWordAnalyzer`,`IdentifierAnalyzer`,`DigitAnalyzer`,`DelimiterAnalyzer`,`OperatorAnalyzer` 的 `IsAccept` 函数来识别 Token。

### 2.3.5 将识别的 Token 串与相关信息输出到相应文件

1. 每个 Token 以每行——`line,col,Token 类型,Token 值`的方式输出到 `token_result.txt` 中。
2. 预处理错误信息以 `line,col,Preprocess` 方式输出到 `WrongInfo.txt` 中。

3. 预处理后的词法分析错误信息以 line, col, Token 类型, Token 值的方式输出到 WrongInfo.txt 中。

### 2.3.6 模块间的调用关系

具体见下图。

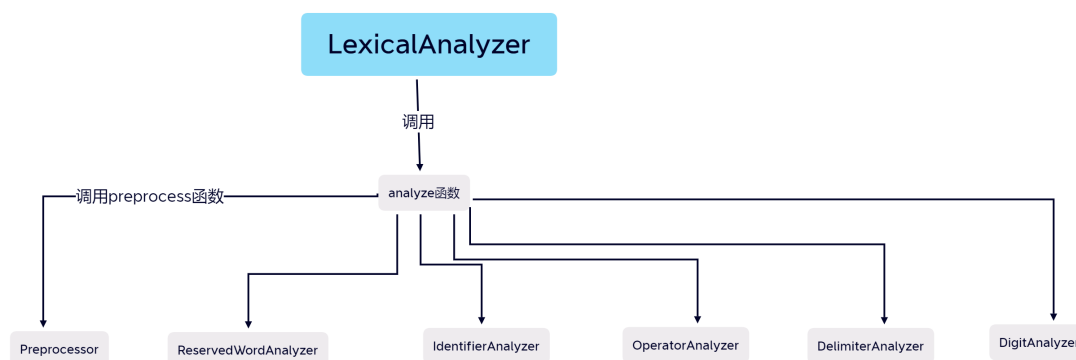


图 4: 词法分析器模块间的调用关系

## 2.4 语法分析

语法分析过程有着较为严格的一套流程，我们据此将语法分析分解为从文件中读入语法并把不同的产生式编号、构建非终结符的 FIRST 集合、项目集 DFA 的构建、ACTION-GOTO 表的构建等多个顺序执行的任务，而在这些顺序执行的任务中，还并行执行着一些任务，比如在读入语法文件的同时，我们将符号区分为为了终结符和非终结符并分别进行存储。接下来，我们将逐个对这些子任务进行介绍。

### 2.4.1 语法文件的读入

在本次作业中，由于要求对类 C 语言做语法分析，并且一套完整的语法还应该包含一整套完整的 Token 定义，我们认为让用户来指定语法是不合适的，因此我们定义里一套类 C 语言语法作为这一环节的输入，该文件名为 `grammer.syn`, 在可执行文件的同级目录下给出。

语法文件为纯文本文件，一行内容为一个产生式。我们采用两个 pass 来读入语法文件。在第一个 pass 的时候，我们可以根据每一条产生的语法确定所有的非终结符。接下来程序执行第二个 pass，由于我们已经知道的非终结符的数量和种类，因此在这个 pass 中我们就可以将所有的符号分为非终结符和终结符两类。与

此同时按照产生式读入的顺序构建每一条产生式并编号以便在构建 GOTO 表和最后的移进规约过程中使用。

不仅如此，在第二个 pass 中，我们为不同非终结符对应的产生式建立了一张查找表，可以通过非终结符的值找到其对应的所有产生式。这样设计主要是为了方便下一步的 FIRST 集构建。

## 2.4.2 FIRST 集合的构建

接下来，我们会在上一节中查找表的基础上进行 FIRST 集合的构建。建立 FIRST 集合是进一步构造项目集规范族 DFA 必要的前序工作，因为在 LR(1) 文法构造项目集闭包的算法中，通过已有项目添加新项目的规则是这样的：

若项目  $[A \rightarrow \alpha.B\beta, a]$  属于  $CLOSURE(I)$ ,  $B \rightarrow \xi$  是一个产生式，那么对于  $FIRST(\beta a)$  中的每个终结符  $b$ ，如果  $B \rightarrow \xi, b$  不在  $CLOSURE(I)$  中，就把它加进去。

构造 FIRST 集合的过程基于递归的思想，算法步骤如下：

1. 对于每个非终结符  $A$ ，执行下列步骤：
2. 遍历  $A$  的所有产生式，对于每条产生式，执行下列步骤
  - (a) 假如  $A$  可以推导出  $\epsilon$ ，则把  $\epsilon$  加入到 FIRST 集合中。
  - (b) 假如  $A$  右侧的第一个符号为非  $\epsilon$  的终结符，则将该终结符加入到 FIRST 集合中。
  - (c) 假如  $A$  右侧第一个符号为非终结符。
3. 假如该非终结符的 FIRST 集合中不包含  $\epsilon$ ，则将该终结符的 FIRST 集并入到当前非终结符的 FIRST 集，结束。
4. 若该终结符可以推导出  $\epsilon$ ，则把该终结符的 FIRST 集中除了  $\epsilon$  的所有符号假如  $A$  的 FIRST 集合，继续向后遍历，直到遍历到 FIRST 集中不含  $\epsilon$  的非终结符或终结符为止。

假如  $A$  的产生式中的所有符号的 FIRST 集中都含有  $\epsilon$ ，则将  $\epsilon$  加入到  $A$  的 FIRST 集合中去。

从上述流程中可以看出，求解一个终结符的 FIRST 集合的过程可能需要用到其他终结符的 FIRST 集合，这就要求我们运用递归的思想，当需要求解其他符号的 FIRST 集合时递归调用函数自身，直到完全求解出某个符号的 FIRST 集合，再回溯回来。重点函数和重点变量将在下一节 详细设计 一节给出，在此不再赘述。

### 2.4.3 项目集规范族 DFA 的建立

项目集规范族的建立过程可以大致分为两个交替进行的子任务，首先我们需要根据初始项目

$$sstart \rightarrow .start, \#$$

构造第一个项目集的闭包。

在获取第一个闭包后，我们建立一个队列的数据结构，并将第一个闭包推入队列中。

接下来，我们执行下列的步骤直至队列为空。

1. 从队列中弹出一个项目集 A。
2. 遍历所有符号，对于每个符号，查找 A 中的产生式在输入当前符号时可能转换到的新项目，并对新项目求闭包，得到一个新的项目集合，假如这个项目集之前从未出现过，则对其进行编号，将这个新项目集和 A 建立联系。将其推入队列中。

假如这个“新项目集”在之前出现过，则获取它的编号，将其与 A 建立联系。

通过这种方法，我们就可以得到完整的项目集规范族及包含转换关系的自动机。

为了更好地展现项目集之间的转换关系，我们利用了开源的绘图工具 *graphviz* 对于项目集 DFA 进行了可视化展示，由于该 DFA 拥有 299 个状态，转换关系十分复杂，我们省去了每个项目集的内容以及转换时需要的符号，得到的图片将在后续的章节进行展示。

建立项目集规范族以及 DFA 的过程如下页的流程图所示。

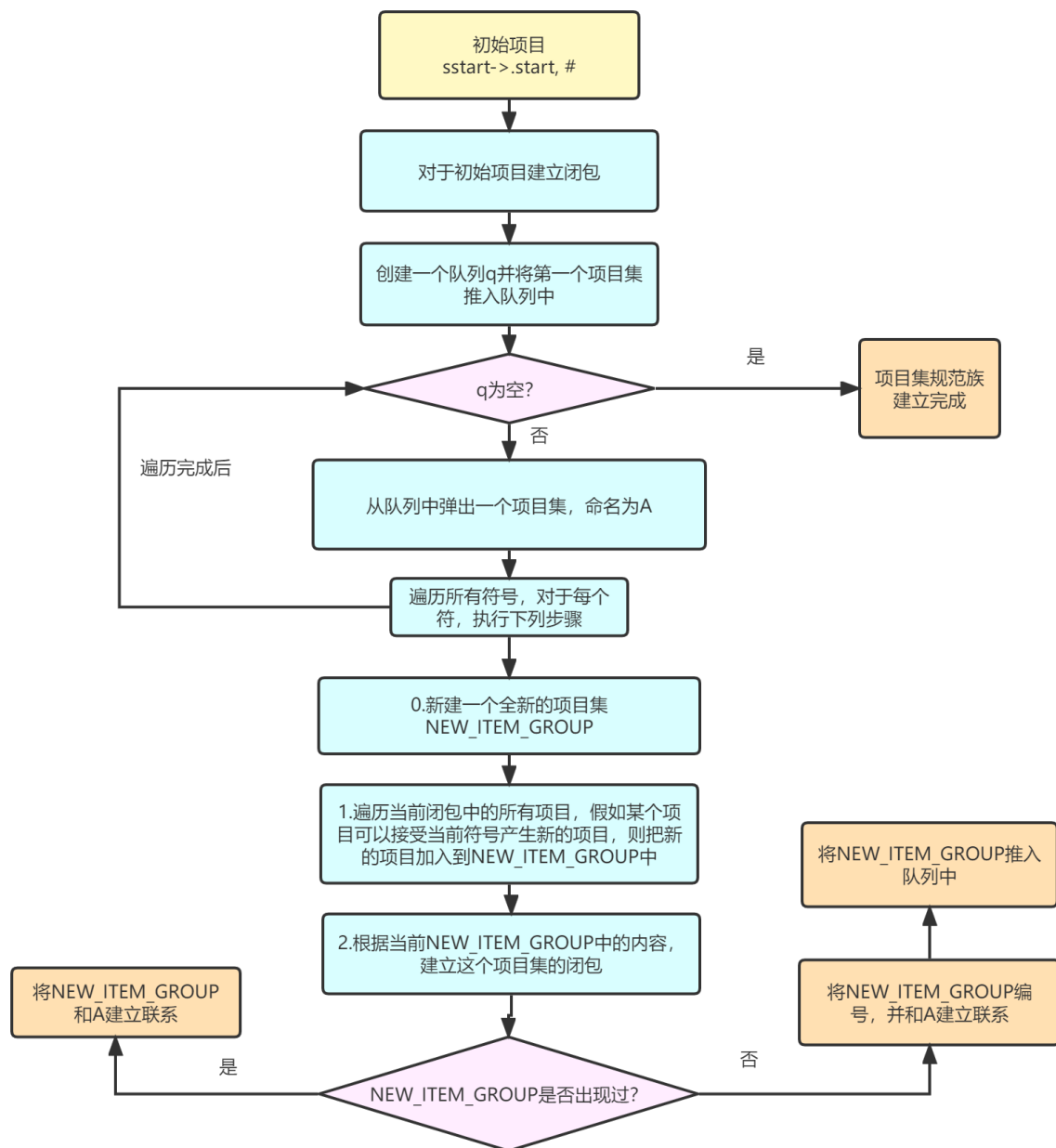


图 5: 项目集 DFA 构造流程图

#### 2.4.4 ACTION-GOTO 表的建立

在建立好语法项目集规范族之后，我们需要建立好 ACTION-GOTO 表，有了这张表，我们就可以愉快地开展移进规约过程了。

ACTION-GOTO 表可以被看做一张二维的表格，行下标代表规范集项目族不同的状态，列下标代表不同的符号，表中每一个格子的内容代表当移进-规约过程时符号栈顶的符号为 ACTION-GOTO 表的列下标，状态栈顶的符号为行下标时程序应当采取的行为。

建立 ACTION-GOTO 表的算法流程如下所示，具体的数据结构设计以及程序中的细节将在下一节 详细设计中给出。

- 遍历所有的项目集 (标号为  $i$  的项目集记作  $I_i$ )。
- 对其中的项目  $I_k (0 \leq k \leq size)$  执行以下步骤
  - 若该项目满足  $A \rightarrow \alpha.a\beta, b$  的格式且满足  $GO(I_k, a) = I_j, a$  为终结符，那么置  $ACTION[K, a] = s_j$ 。
  - 若该项目满足  $A \rightarrow .\alpha, a$  的格式且满足  $GO(I_k, a) = I_j$ ，那么置  $ACTION[K, a] = r_j$ 。
  - 其中  $A \rightarrow \alpha$  为文法的第  $j$  条产生式。
  - 若项目为  $sstart \rightarrow start., \#$ ，则置  $ACTION[K, \#]$  为  $ACC$ 。
- 遍历所有的非终结符，若该项目集 ( $I_k$  在接收一个  $S$  符号输入后会转移到项目集  $j$ ，则置  $GOTO[k, S] = j$ 。

#### 2.4.5 移进规约过程

首先，我们需要设计两个栈：符号栈和状态栈。前者压入输入的符号以及归约得到的符号，后者负责压入移进得到的状态号。在移进过程中，我们首先查 action 表，如果 action 表中是移进项目，那么就把读入的符号压到符号栈，然后根据移进的项目号把项目号压入项目栈。如果查 action 表查到的是移进项目，那么就按照移进项目的下标所对应的产生式进行归约，由于之前都是合法入栈的，所以直接将符号栈弹出  $n$  个元素即可， $n$  就是归约产生式的右边的长度，然后对应的项目栈也要弹出  $n$  个元素，然后将归约产生式的左边符号压栈。然后查 goto 表，根据此时项目栈的栈顶元素以及符号栈的栈顶元素查询对应的项目号，将该项目号压入项目栈。不断重复以上内容，知道最后输入  $\#$  得到  $ACC$ ，具体的流程图如下页：

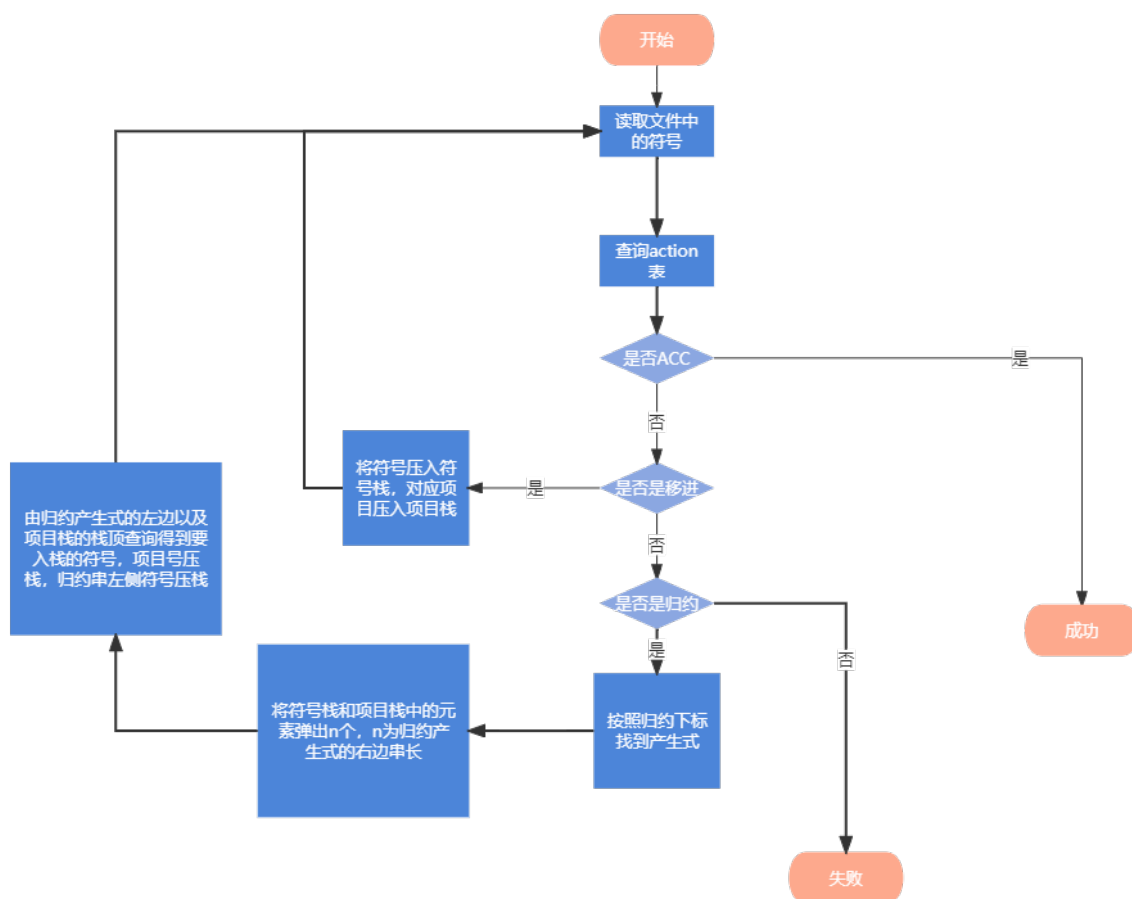


图 6: 移进规约过程流程图

## 2.5 前端 UI 设计

在前端方面我们采用 `C++ Qt5.9.9` 进行开发，和词法分析，语法分析的部分主要通过中间文件的方式进行数据交互和展示。

UI 部分大致分为程序使用说明和功能展示两个主要页面。

程序使用说明部分页面描述了本程序的使用规则，包括使用的文法，可能输出的一些中间文件 (FIRST 集合，ACTION-GOTO 表等) 等。

功能展示页面中我们设计了“选择文件”，“词法分析”，“语法分析”，“查看语法”，“重置”五个按钮，其功能分别如下：

- “选择文件”：用户在文件选择窗口中选择一个输出文件作为源程序。
- “词法分析”：对选择的源程序文件进行词法分析。假如用户此时并未选择文件，程序会弹出警告框。

完成词法分析后，UI 上会显示是否分析成功，并会显示分析成功的文件的 token 串。

- “语法分析”：对经过词法分析的 Token 串进行语法分析，UI 上回显示是否分析成功，并会显示分析栈和 token 串中首符号的动态变化过程。
- “查看语法”：显示语法文件 *grammer.syn* 中的所有产生式。
- 清空既有状态，重新选择文件进行分析。

前端界面的功能演示图片和对于用户操作的鲁棒性将在后续的成果展示一部分进一步描述。

## 2.6 三个模块之间的数据交互

三个模块之间数据交互示意图如下所示

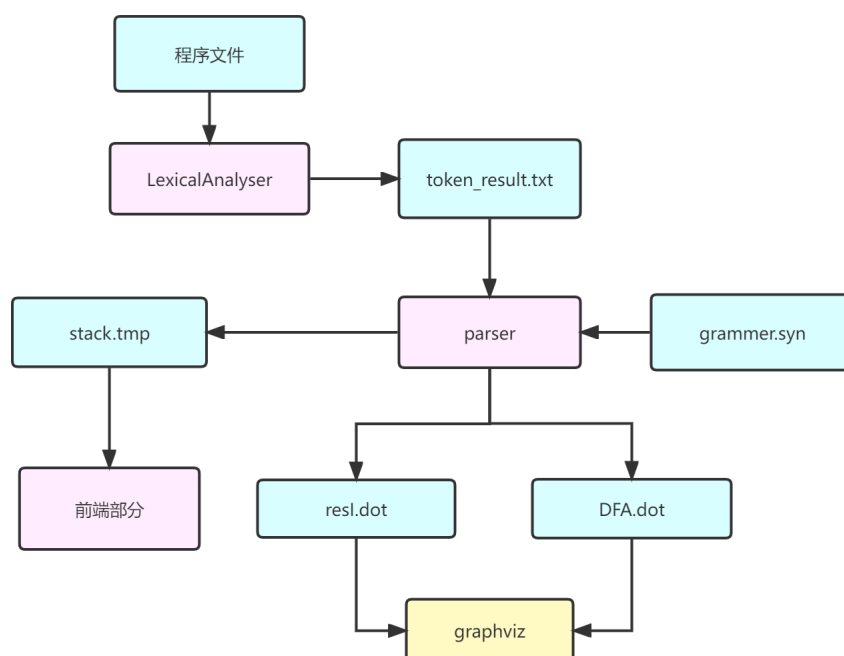


图 7: 数据传输示意图

## 3 详细设计

### 3.1 词法分析

词法分析过程中主要使用的函数为共用的虚函数 **virtual int isAccepted** 以及各个类里的具体实现。具体函数声明如下：

```

1 virtual int isAccepted(const std::string &str, unsigned int &start, unsigned
    int &end, unsigned int &line, unsigned int &col)=0;
  
```



下具体说明该虚函数在五个类别识别的 DFA 里的具体实现思路与部分代码展示。

### 保留字识别 DFAReservedWordAnalyzer:

对于保留字的识别,只需要记录语法文件中允许的若干个保留字,分别为"int", "double", "char", "float", "break", "continue", "while", "if", "else", "for", "return", "function"。而考虑到各 DFA 均需要与 LexicalAnalyzer 类进行"互动",所以考虑直接把保留字表直接存在 LexicalAnalyzer.cpp 中,方便 LexicalAnalyzer 直接访问判断。

对于具体实现方面,每次对于当前字符判断是否为字母/数字,直到第一个非字母且非数字的字符,将前面得到的字符串去查找

```
std::string ReserveWord[RESERVED_WORD_NUM]
```

表,判断是否为保留字。

### 标识符识别 DFA IdentifierAnalyzer:

对于标识符的识别,由于只需要满足为字符/数字,所以不需要预先存表,只需要将每次出现的表存入标识表中,用于后续语法分析来调用。

对于具体实现方面,每次对于当前字符判断是否为字母/数字,直到第一个非字母且非数字的字符,将前面得到的字符串直接插入标识表中。而为了插入操作的简洁性,直接考虑采用自动去重的 std::set 来存储标识符,而每次只需要 insert 当前得到的标识符而不需要考虑是否重复的问题。

### 运算符识别 DFA OperatorAnalyzer:

对于运算符的识别,只需要记录语法文件中允许的若干个运算符。而考虑到各 DFA 均需要与 LexicalAnalyzer 类进行"互动",所以考虑直接把运算符表直接存在 LexicalAnalyzer.cpp 中,方便 LexicalAnalyzer 直接访问判断。

对于具体实现方面,考虑到算符的特殊性,决定不采用纯粹的 DFA 转换进行判断,而是每次对于当前字符判断是否为运算符,对特定读到的字符进行一至两步的展望,来判断当前读到的运算符的类别以及是否为运算符,下为部分代码。

```
1 else if (ch=='>') {
2     char ch1=str[pos+1];
3     if(ch1=='>') {
4         char ch2=str[pos+2];
5         if(ch2=='=') {
6             end=pos+2;
7         }
8         else {
9             end=pos+1;
10        }
11    }
12    else if (ch1 == '=' ) {
```

```

13         end = pos + 1;
14     }
15     else {
16         end=pos;
17     }
18 }

```

### 界符识别 DFA DelimitersAnalyzer:

对于界符的识别,只需要记录语法文件中允许的若干个界符,分别为"(",")","", "'", "[", "]", ";", "。而考虑到各 DFA 均需要与 LexicalAnalyzer 类进行"互动",所以考虑直接把界符表直接存在 LexicalAnalyzer.cpp 中,方便 LexicalAnalyzer 直接访问判断。

对于具体实现方面,由于界符为单字符,所以直接对于当前字符判断是否为界符即可,将得到的字符串去查找 `std::stringDelimiter[DELIMITER_NUM]` 表,判断是否为保留字。

### 数字识别 DFA DigitAnalyzer:

对于数字的识别,考虑数字主要有以下几种类别,下分别以具体数值举例。

1. 纯整数:2345678
2. 整数 + 小数:234.56
3. 科学计数纯整数:35e5
4. 科学计数整数 + 小数:234.56e7

所以数值识别可以采用纯 DFA 的识别方式,具体 DFA 及状态转移图如下。

相关的状态转移以代码形式举例如下:

```

1  else if (state == STATE_2) {
2      if (str[i] <= '9' && str[i] >= '0') {
3
4      }
5      else if (str[i] == 'e') {
6          state = STATE_3;
7      }
8      else if(str[i]=='.'){
9          state = STATE_ERROR;
10         break;
11     }
12     else {
13         state = STATE_TERMINATE;
14         break;
15     }
16 }

```

还有部分工具函数,具体函数声明如下:

```

1 bool isLetter(char letter);
2 bool isDigit(char digit);
3 bool isDelimiter(char ch);
4 bool isOperator(char ch);

```

它们分别用于判断当前字符(串)是否为字母, 数字, 界符, 算符以及将返回结果转为相应信号量传递给外界。

## 3.2 语法分析

语法分析的实现过程主要在 *parser* 类中完成, 接下来我们会详细地介绍类中的函数和一些数据结构。

我们将读入文件的环节设置为如下的函数

```

1 void Parser::read_grammar(const std::string path)

```

在这个过程中, 我们将语法分解为一条条的产生式, 并将非终结符和终结符分开存放。其中一个符号包括它的值, 类型以及一个指向其所有产生式的一个指针, 具体定义如下:

```

1 class symbol //
2 {
3     std::string name; // the name of symbol
4     bool type=0; // 0 presents terminator ,vice versa
5     int generators_index = -1;
6     // the index of this symbol's generators list
7 };

```

在读入产生式之后, 我们需要为每个符号建立它的 FIRST 集合, 对应函数为

```

1 void Parser::get_all_symbol_first();

```

在这个函数中会遍历所有的符号并对所有的符号执行如下函数

```

1 void Parser::get_symbol_first(const symbol& a);

```

这个函数的参数即为符号 *a*, 当 *a* 为终结符式直接返回 *a* 自身, 当 *a* 为非终结符式遍历 *a* 的所有产生式进行求算。

值得一提的是, 由于求解一个符号的 FIRST 集的过程中可能要求解其他符号的 FIRST 的集合, 因此该函数是一个递归函数。

在求解了所有符号的 FIRST 集之后, 我们需要建立项目集规范族及其转换关系, 其中一个项目的定义如下。

```

1 class item // point and prospect symbols are included
2 {

```

```

3 public:
4     const generator base; //产生式
5     unsigned int index; //圆点的位置
6     std::set<std::string> prospect_symbols; //展望
7 };

```

而一个项目集闭包的定义是这样的:

```

1     int id; //项目集编号
2     std::set<item> items; //包含的所有项目

```

在介绍求解项目集规范族的函数前, 还有三个工具函数需要介绍。

```

1     void Parser::get_closure(item_group& group);

```

在这个函数中, 我们根据一个项目集中既有的项目求算它的闭包。

```

1     void Parser::get_sequence_first(const std::vector<std::string>& seq,
2                                     std::vector<std::string>& re);
3     //the seq is the input sequence
4     //re is the reference to the results we want

```

这个函数可以求取一个符号序列的 FIRST 集合, 在上一个求取闭包的函数中被调用。

```

1     void Parser::item_group_go(const item_group& scr, item_group& dst, std
2                                ::string input);
3     //求取 scr 闭包在接受 input 符号后转换到的新闭包 dst

```

这个函数接受一个闭包 scr 和一个输入串 input, 将 scr 输入 input 后得到的新闭包存储在 dst 中。

获得项目集规范族的函数为

```

1     void Parser::get_item_group_list();

```

在这个函数中, 我们首先利用项目  $sstart \rightarrow .start, \#$  构建了第一个项目集闭包。接着利用 `std::queue` 建立了一个队列, 并将第一个项目集闭包推入队列中。

接下来的过程和概要设计一节中描述的相同, 在队列不为空的时候, 每次出队一个闭包, 我们会遍历所有符号, 并对于每个符号 input 调用 `item_group_go` 函数获得转换后的闭包 dst, 假如 dst 从未在队列中出现过, 我们就将它加入队列。

为了判断有哪些闭包曾出现过, 我们利用率 `std::set` 容器, 容器中的所有元素都是不重复的, 可以根据它的 `find` 方法实现高效查找 (内部数据结构为红黑树)。

建立好的项目集规范族存储在如下的数据结构下:

```

1     std::set<item_group> item_groups;

```

接下来我们就可以建立 ACTION-GOTO 表了，前面提到过 ACTION-GOTO 表可以被看做是一张二维的表，因此二维数组理应是首选的数据结构。

但经过观察和简单的测试不难发现，在大多数情况下 ACTION-GOTO 表相当稀疏，用二维数组存储势必浪费大量空间。而邻接链表的存储结构虽然无法做到顺序存取，但由于每个项目集对应的 ACTION 或 GOTO 项极少，时间开销的增加幅度是可以承受的。

因此我们将 ACTION-GOTO 表存储在一个 1 维数组中，数组中每一个元素的下标即为对应项目集的编号，存储的内容两个链表的头指针，两个链表分别代表 ACTION 项和 GOTO 项，而链表中每一个结点包含一个符号和一个 ACTION/GOTO 操作。

比如 1 维数组中第  $i$  项中的 ACTION 指针指向的链表中包含了一个 A 符号和  $S_j$  操作。意为当移进规约过程中符号栈中为 A，状态栈中为  $i$  时，程序应将 3 号状态推入状态栈，将 token 串第一个元素推入符号栈。

对应的变量定义如下

```
1 std::vector< movement> LR1_table;
2 //movement 中含有两个链表的头指针
```

建立 ACTION-GOTO 表对应的函数为

```
1 void get_LR1_table();
```

最后是关于移进归约的函数的实现。移进归约主要依赖 action 表和 goto 表，主要采用的数据结构是两个栈：符号栈和项目栈，具体实现依赖于 std::stack 来进行实现。这两个 stack 的作用就是在移进归约的时候存储当前项目以及当前的符号的，这体现了 LR(1) 最左归约的时间特点。具体的函数声明如下：

```
1 int parser::check(std::string path);
```

这里的函数参数就是词法分析的结果，然后返回值是出错行数，如果正确则返回宏定义 "SUCCESS=-1"

## 4 调试分析

### 4.1 基本功能调试分析

对于本实验来说，需要检测的主要可以分为两个阶段，一个是词法分析阶段，一个是语法分析阶段。

对于词法分析，我们希望能够通过输入一串字符串，得到词法中该符号对应的 token 值，方便给语法分析器进行进一步的分析；以及这个符号在串中的位置信息，方便之后的报错处理。

对于语法分析器来说，语法分析器的输入就是词法分析器的输出。从词法分析器获取到的 token 传给语法分析器，然后语法分析器利用已经根据语法求出的 action 表和 goto 表对于输入串进行移进规约。如果在移进归约的过程中遇到了错误，就把出错的字符的位置信息输出，如果移进归约到最后遇到 action 表中的 ACC (宏定义)，则表示移进归约成功，该输入串满足移进规约的要求，是一个合法的串。

下面具体介绍一下本实验中的测试数据以及测试的结果。

### 4.1.1 测试数据以及测试结果

**正确测试数据** 首先是对于定义数据类型的语法的测试，具体的测试程序如下（测试代码中会含有空换行以及空格以及注释，这些在做词法分析的预处理的时候都应该处理好）：

```
1 double aa=1.1e5;
2     int inta=1; //xss
3 char hh;
4     //bnueab
```

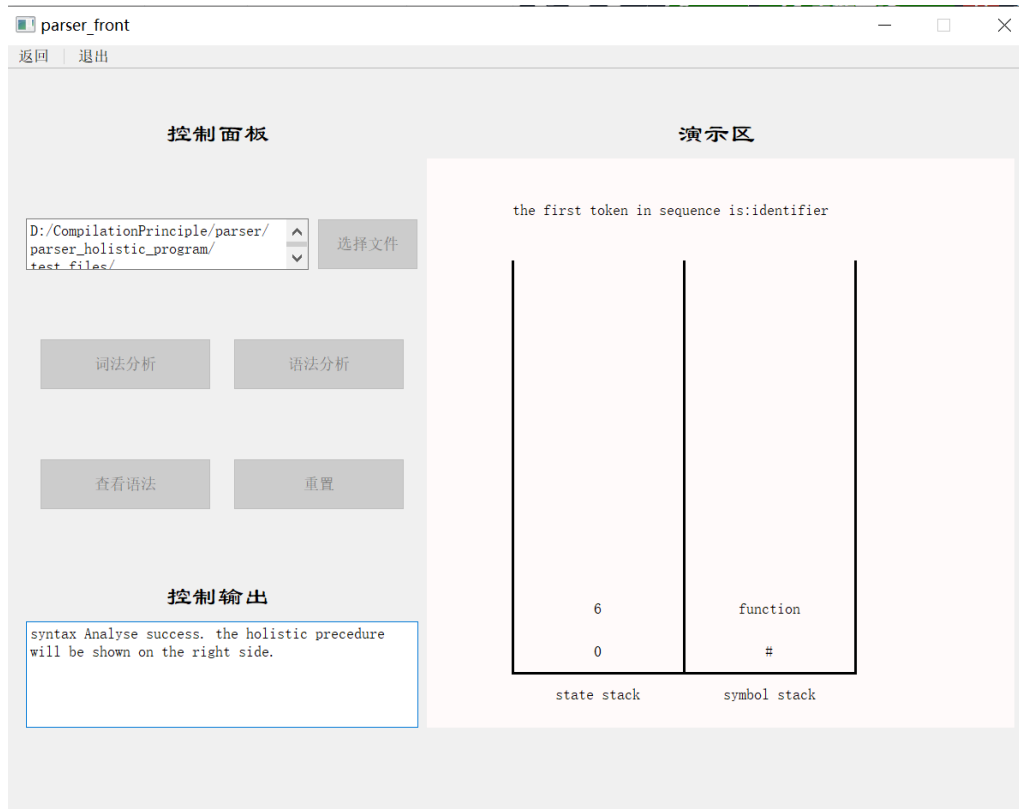
测试结果如下，语法分析成功

The screenshot shows a web-based application titled "parser\_front". It has a "控制面板" (Control Panel) on the left and a "演示区" (Demonstration Area) on the right. The Control Panel includes a file selection dropdown showing a path to "test\_files/1\_data\_correct.txt", buttons for "词法分析" (Lexical Analysis), "语法分析" (Syntax Analysis), "查看语法" (View Grammar), and "重置" (Reset). The Demonstration Area displays the message "the first token in sequence is: number" and two vertical stacks: the "state stack" with values 29, 18, 14, 10, and 0, and the "symbol stack" with values "number", "=", "identifier", "type\_specifier", and "#". A "控制输出" (Control Output) box at the bottom left shows the message: "syntax Analyse success. the holistic precEDURE will be shown on the right side."

然后是关于函数定义的语法的测试，具体的测试程序如下：

```
1 function int main(int a){ //在我们的语法中规定必须以关键字 function 作为函数的开头
2     return 0;
3 }
```

测试结果如下，语法分析成功



接着是关于条件分支语句的语法的测试，具体的测试程序如下：

```
1 function int main(int a){ //在我们的语法中规定条件分支语句必须在函数体内使用
2     if(a<1){
3         if(1){
4             return 123;
5         }
6         else{
7             return -1;
8         }
9     }
10    else
11    {
12        int c=p;
13    }
14    return 0;
15 }
```

测试结果如下，语法分析成功。

## 控制输出

syntax Analyse success. the holistic precedure  
will be shown on the right side.

接着是对于 while 语句的语法的测试，具体的测试程序如下（测试代码中会含有空换行以及空格以及注释，这些在做词法分析的预处理的时候都应该处理好）：

```
1  function int main(int a){ //在我们的语法中规定循环语句必须在函数体内使用
2      while (a<0){
3          a=a+1;
4      }
5      return 0;
6  }
```

测试结果如下，语法分析成功。

返回 | 退出

### 控制面板

D:/CompilationPrinciple/parser/  
parser\_holistic\_program/  
test\_files/4\_while\_correct.txt

选择文件

词法分析

语法分析

查看语法

重置

### 演示区

the first token in sequence is:(

15	identifier
12	type_specifier
6	function
0	#

state stack

symbol stack

### 控制输出

syntax Analyse success. the holistic precedure  
will be shown on the right side.

24



接着是对于 for 语句的语法的测试，具体的测试程序如下（测试代码中会含有空换行以及空格以及注释，这些在做词法分析的预处理的时候都应该处理好）：

```
1  function int main(int a){ //在我们的语法中规定循环语句必须在函数体内使用
2      for(int i=0;i<a;i=i+1){
3          return 123;
4      }
5      return 0;
6  }
```

测试结果如下，语法分析成功。

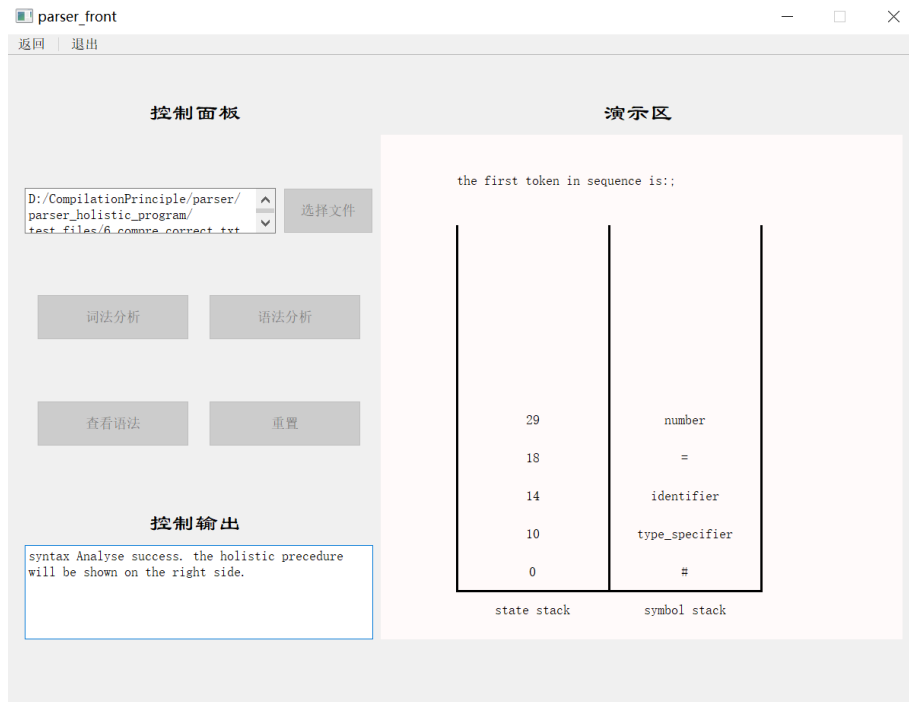
## 控制输出

```
syntax Analyse success. the holistic precedure
will be shown on the right side.
```

最后是对所有语句综合的语法的测试，具体的测试程序如下（测试代码中会含有空换行以及空格以及注释，这些在做词法分析的预处理的时候都应该处理好）：

```
1  double aa=1;
2  int inta=1;
3  char hh;
4  //bnueab
5  function int main(int a){ //在我们的语法中规定循环语句以及条件分支语句必须在函数体内使用
6      while(10086){
7          int a;
8      }
9      if(2){
10         while(2){
11             for(int i=0;i<2;i=i+1){
12                 return 123;
13             }
14         }
15     }
16     else
17     {
18         int c=p;
19     }
20 }
```

测试结果如下，语法分析成功。

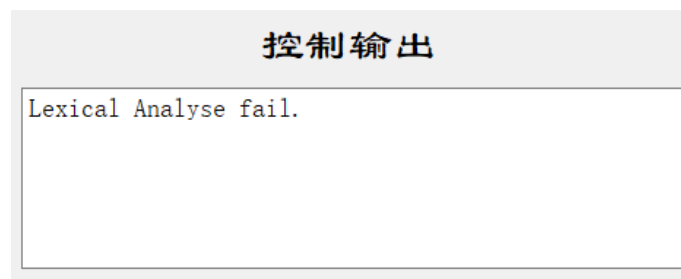


**错误测试数据** 接下来我们还构建了一些错误输入的文件，来检测我们词法分析器以及语法分析器的报错能力：

首先是错误的注释方式：

```
1 double aa=1;
2 int inta=1;# 错误注释
3 char hh; // 正确注释
```

测试结果如下，词法分析失败。



接着是错误的关键字：

这是关键字错误，会在词法分析出错

```
1 DOUBLE aa=1; // 错误关键字
2 INT inta=1; // 错误关键字
3 char hh; // 正确注释
```

测试结果如下，语法分析失败。

这是缺失关键字，会在语法分析出错

### 控制输出

```
syntax Analyse fail.the nearest error occurs in  
line 1 !
```

```
1 int main(int a){ //缺少关键字 function  
2     while(10086){  
3         int a;  
4     }  
5     if(2){  
6         while(2){  
7             for(int i=0;i<2;i=i+1){  
8                 return 123;  
9             }  
10        }  
11    }  
12    else  
13    {  
14        int c=p;  
15    }  
16 }
```

测试结果如下，语法分析失败。

### 控制输出

```
syntax Analyse fail.the nearest error occurs in  
line 1 !
```

接着测试数字格式错误

```
1 double aa=1..e5;
```

测试结果如下，词法分析失败。

### 控制输出

```
Lexical Analyse fail.
```

接着测试一些我们写代码时经常会出现的错误：

缺少";"，会在语法分析出错

```
1 int aa=1
```

测试结果如下，语法分析失败。

### 控制输出

```
syntax Analyse fail.the nearest error occurs in  
line 1 !
```

括号不匹配，会在语法分析出错

```
1 function int main(int a{ //缺少右括号  
2     while(10086){  
3         int a;  
4     }  
5     if(2){  
6         while(2){  
7             for(int i=0;i<2;i=i+1){  
8                 return 123;  
9             }  
10        }  
11    }  
12    else  
13    {  
14        int c=p;  
15    }  
16 }
```

测试结果如下，语法分析失败。

### 控制输出

```
syntax Analyse fail.the nearest error occurs in  
line 1 !
```

括号不匹配，会在语法分析出错

```
1 function int main(int a{ //缺少右括号
2     while(10086){
3         int a;
4     }
5     if(2){
6         while(2){
7             for(int i=0;i<2;i=i+1){
8                 return 123;
9             }
10        }
11    }
12    else
13    {
14        int c=p;
15    }
16 //这里缺少右大括号
```

测试结果如下，语法分析失败。

### 控制输出

```
syntax Analyse fail.the nearest error occurs in
line 1 !
```

#### 4.1.2 时间复杂度分析

首先分析词法分析器的复杂性：词法分析器主要有以下几个过程：

1. 输入字符串的预处理
2. 各 DFA 对输入字符的判断与传输

**输入字符串的预处理，**

由于需要对全部输入字符串进行判断与处理，所以时间复杂度一定为  $O(n)$ ，其中  $n$  为输入字符串的长度。

各 DFA 对输入字符的判断与传输，下面将分各类进行具体说明。

**保留字 DFA**

考虑到保留字的个数极少，所以直接采用数组的方式存储保留字，所以每次判读是否为保留字的时间复杂度为  $O(n)$ ，其中  $n$  为保留字的个数。

## 标识符 DFA

由于标识符长度不定，每次仅需判断当前字符是否为数字/字母，所以时间复杂度为  $O(m)$ ,  $m$  为标识符的长度。

## 界符 DFA

考虑到界符的个数极少，所以同样直接采用数组的方式存储保留字，所以每次判读是否为界符的时间复杂度为  $O(n)$ , 其中  $n$  为界符的个数。

## 数字 DFA

由于数字的识别采用纯 DFA 的方式进行识别，所以时间复杂度为  $O(m)$ ,  $m$  为数字的长度。

## 算符 DFA

由于算符的有限性与特殊性，直接对当前字符进行判断以及提前往后展望 1-2 个字符即可判断出算符，所以时间复杂度为  $O(1)$ 。

接着是语法分析器的复杂性：语法分析器主要有一下几个过程：

1. FIRST 集的构建
2. 项目集规范族的构建
3. 识别活前缀的 DFA 的构建
4. Action 表和 Goto 表的构建

对于 FIRST 集的构建，主要是一个递归的算法，即如果产生式的右边的第一个符号是终结符，则把终结符加入产生式左边的 FIRST 集中，如果产生式的右边第一个符号是非终结符，那就将这个非终结符的 FIRST 集去掉  $\epsilon$  之后加入到左边符号的 FIRST 集中去，同时，产生式的右边的首符的 FIRST 集中有空，那么还要将其之后的符号的 FIRST 集加入左边符号的 FIRST 集，不断重复直到加入的符号的 FIRST 集中没有空，如果直到最后都有  $\epsilon$ ，那就把  $\epsilon$  也加入到 FIRST 集中。

上述过程的复杂度是线性的，每一次递归并不会产生多出来的分治条件，因此复杂度是  $O(n)$

其次是项目集规范族的构建，首先，求项目集规范族主要分为两步：首先是按照一条产生式求项目集的闭包。第二步是由一个项目集求另一个项目集。第一步是第二步的子集，主要复杂度都集中在第一步，因此这里主要分析第一步。

由一条产生式根据圆点之后的字符后面的字符串  $\beta$ ，同时与展望字符  $a$  合在一起求 FIRST（已经由上面的算法求好了），得到的就是新的项目的展望。如果圆点之后没有任何字符，则停止求项目。这一算法的本质就是一个循环迭代的过程，其时间复杂度是： $O(n)$ 。（因为之前已经求好 FIRST 集了）。

构建识别活前缀的 DFA 其实在构建完项目集族之后就不是什么难事了，我们

在项目集族的结构体中设置了项目集的编号以及其指向的其他项目集（使用 map 实现），DFA 直接就能根据这两个参数直接求出来，复杂度为  $O(1)$ 。

完成 DFA 的构建之后求 action 和 goto 表只需要按照之前存储的 DFA 改变存储结构即可，其实也可以按照 DFA 来识别活前缀然后实现移进归约。因此只是换一个存储方式，复杂度为  $O(n)$ 。

## 4.2 调试问题与解决

### 4.2.1 词法分析

在完成词法分析器的过程中主要遇到了以下一些主要问题：

1. 测试输入文件中末尾加上若干空格和换行，结果报错

**解决方法：**发现在 LexicalAnalyzer.cpp 的 analyze 函数处理空格/换行时识别后但未检测指针是否超出输入串的长度，加上如下代码后问题解决。

```
1 if (start >= out_str.size()) {  
2     sign = RIGHT_STATUS;  
3     break;  
4 }
```

2. 一开始各 DFA 未统一 start、end 的指向先后性，导致输入字符串识别有误

**解决方法：**统一将 start 指针仅在 LexicalAnalyzer 类里修改，而 end 指针仅当当前 DFA 识别字符（串）成功时才进行修改，而每次 LexicalAnalyzer 需要调用 DFA 前都将  $start=end+1$  就成功地解决了问题。

### 4.2.2 语法分析

在完成语法分析器的过程中主要遇到了以下一些主要问题：

1. set 容器小于号重载问题。set 小于号的重载非常重要，在每一次往 set 集中插入数据的时候，会将插入的元素和集合中的元素进行比较，然后换位之后再比较，通过两次比较确定两个元素的大小关系。如果小于号的重载有问题，比如总是返回 false，那么就永远无法插入数据，加入永远返回 true，会直接弹窗报错。因此，我们需要构建一种能够区分出所有元素的小于号重载。这其实并不是一件容易的事，因为我们很多类是嵌套的，需要一级一级做重载。最底层的类是 symbol 类，我们用符号的值来直接比较，利用的是 `std::string` 已经重载好的小于号。其上层是 generator 类，存储产生式，我们为了之后的归约操作的方便，我们对于产生式都有编号，只需要比较 id 号即可区别所有 generator。再然后就是 item 类，也就是项目类。该项目的小于号的重载需要




比较每一个项目的产生式，圆点的位置，以及展望的符号进行比较，主要还是依靠 `std::string` 已经重载的小于号以及之前重载好的几个类的小于号，利用迭代器遍历 `std::vector` 对于每个元素进行比较。

2. 归约时项目栈弹出元素问题。刚开始我对于归约时对于栈的操作理解错误，每一次归约我都会弹出一个项目栈的元素，但是，弹出元素的个数应该取决于归约串的长度，如果归约到  $\epsilon$  的话，其实是不需要弹出状态的，因此，在这一点上的理解错误导致我的归约步骤出错。

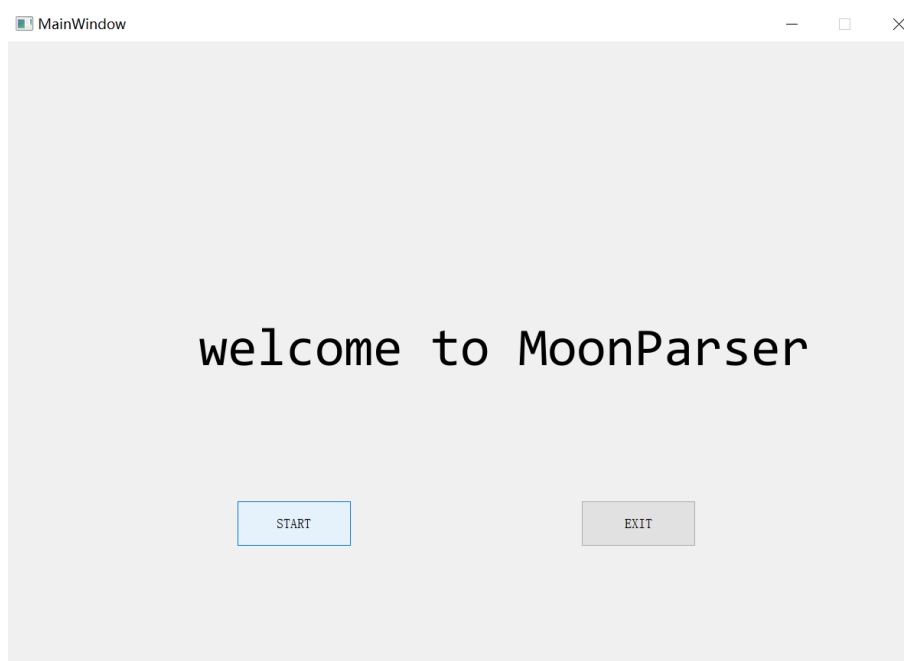
## 5 程序使用说明

在这个部分我们会介绍程序的使用方法和使用过程中的可能的输出含义。

1. 首先进入 *parser\_holistic\_program* 文件夹，其中的 *parser\_front.exe* 即为可执行文件。

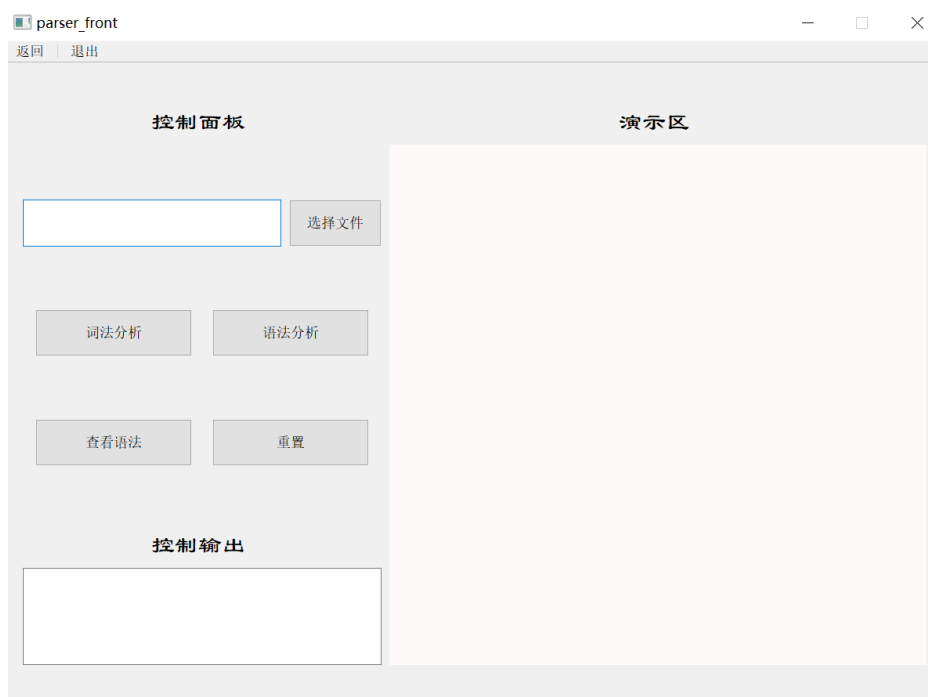
 opengl32sw.dll	2016-06-14 21:08	应用程序扩展	15,621 KB
 parser_front.exe	2022-11-13 15:10	应用程序	157 KB
 Qt5Core.dll	2022-11-13 15:14	应用程序扩展	6,004 KB

2. 点击进入，即可进入程序欢迎界面。点击左下方的 *START* 按钮即可进入规则说明页面。





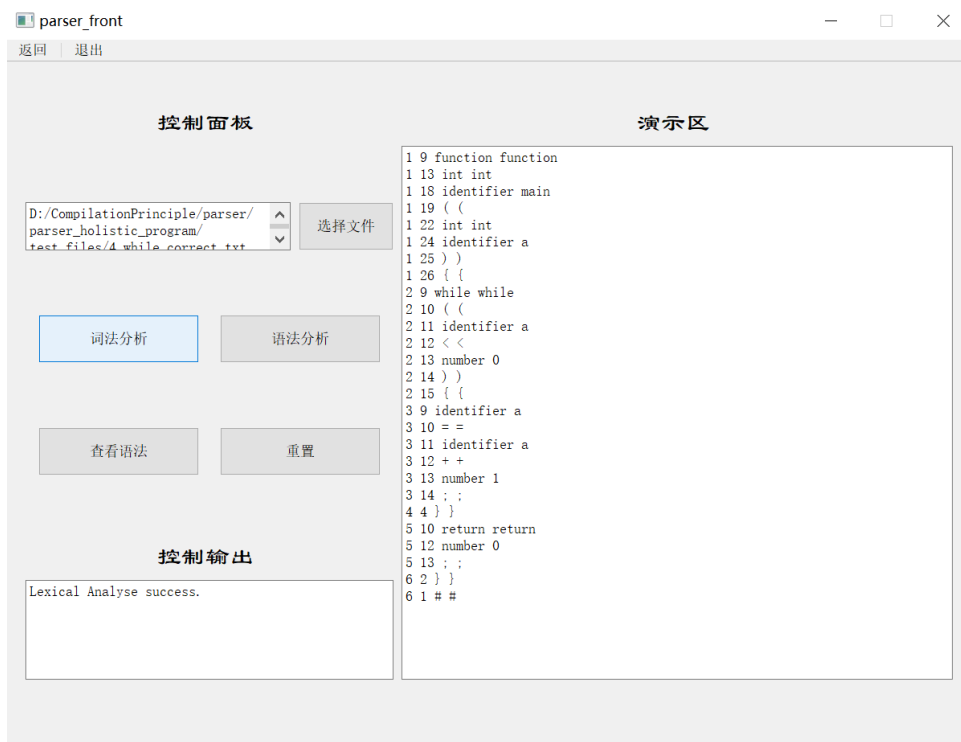
3. 查看规则后点击 *CONTINUE* 页面即可进入功能演示界面。



4. 在功能演示界面中点击“选择文件”按钮，会弹出文件对话框，确认之后，文件中的内容将会展示在界面右侧。主要注意的是，文件的路径中不能含有中文字符。



5. 在选择文件后，用户可以点击“词法分析”按钮进行词法分析，若程序文件可以通过词法分析，则会在左下方的控制输出中打印"lexical analysis success.", 在右侧面板中展示分析完毕的 Token 串。



6. 在完成词法分析后，用户可以点击“语法分析”进行语法分析，若 token 串可以成功通过语法分析，则会在左下方的控制输出中打印"syntax analysis success."，在右侧面板中动态展示移进规约的全过程。



7. 点击“查看语法”按钮，语法分析使用的文法会显示在右侧面板上；点击“重置”按钮，则会清除之前输入的文件，用户需要重新选择文件进行分析。

在成功完成一次语法分析后，可执行文件同级别目录下会出现 *DFA.dot* 和 *resI.dot* 两个文件，*DFA.dot* 文件中存储的是项目集规范族 DFA 的信息，*resI.dot* 中存储的是语法树的信息。将这两个文件输入到 *graphviz* 中即可生成对应的图片。具体方法如下。

- 在命令行中将当前目录切换为 *Graphviz* 下。
- 通过输入

*neato -Tpng path(DFA.dot) -o path(DFA.png)*

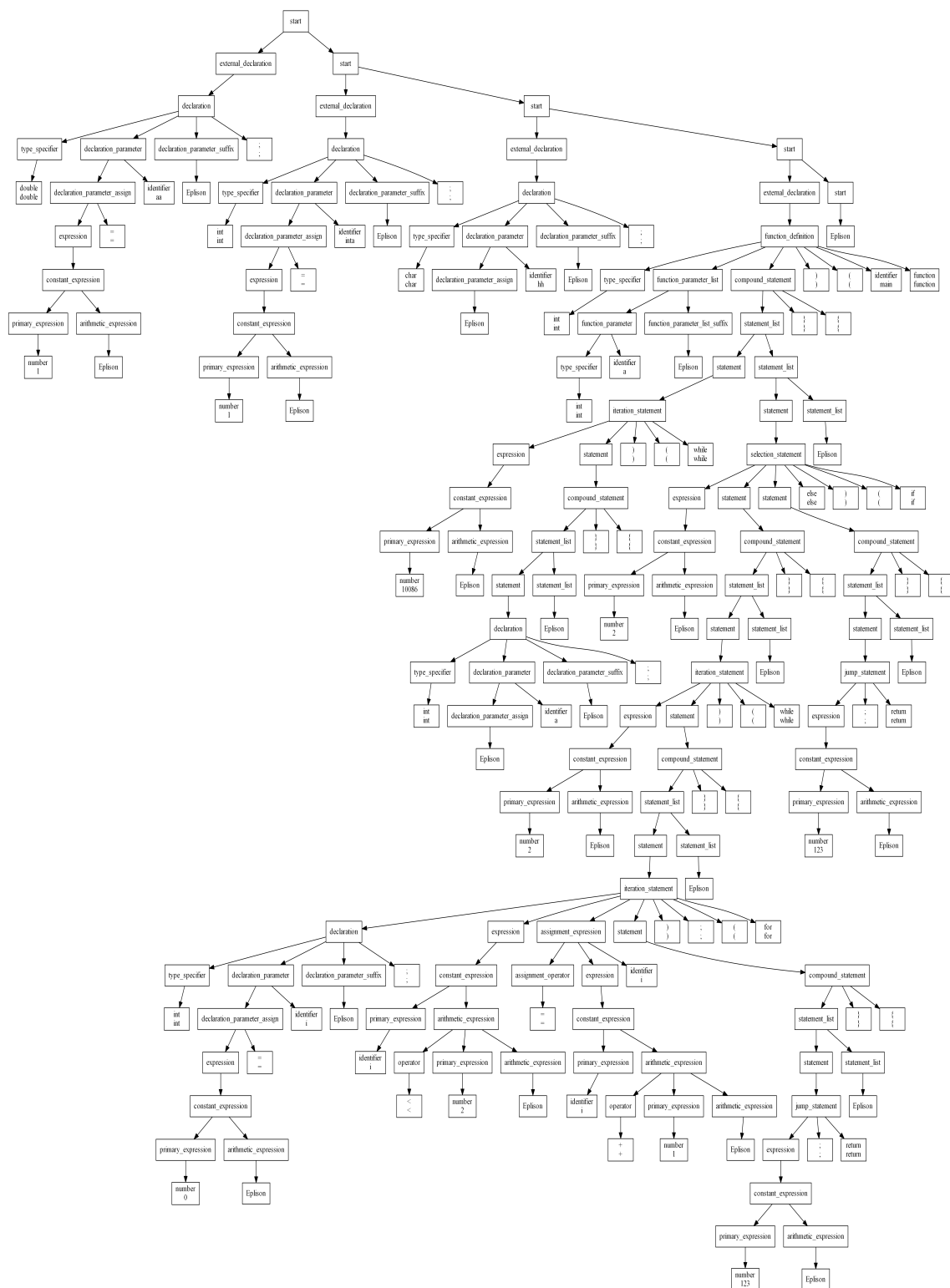
命令，即可将路径为 *path(DFA.dot)* 的 *DFA.dot* 文件转换为 *png* 文件输出到 *path(DFA.png)* 目录下。

通过输入

*dot -Tpng path(resI.dot) -o path(resI.png)*

命令，即可将路径为 *path(resI.dot)* 的 *resI.dot* 文件转换为 *png* 文件输出到 *path(resI.png)* 目录下。

画出的语法树 *resI.png* 如下图。



由于项目集规范族和示例语法树较为庞大，图片在报告中存在分辨率不够的问题，您也可以通过访问

<https://github.com/K-D-B/MoonParser1.0>

来获取这张图片。

## 6 总结与收获

本次大作业为小组任务，由陈开煦、何征昊、黎可杰三位同学组队完成。首先，我们通过课件和课本温习了词法分析和语法分析的相关知识，接下来，我们找到了一份长度适中的类 C 语言文法，并以此为基础完成相关的分析过程。

程序的开发过程采用模块化和前后端分离的思想。我们首先将任务划分为不同的模块，然后定义好相关的数据接口，再分配不同的同学来完成。整个程序开发过程持续了一周左右的时间，小组成员之间通力合作，群策群力，合作顺利且愉快。

从作品的角度来看，程序完整地满足了作业的所有要求，在具备以下特点的同时兼具了一定的可移植性和可扩展性。

- 程序具有完善性。通过读入文件的方法输入文法，可以完整地实现词法分析和语法分析的过程，在过程中输出两个 *.dot* 文件实现 DFA 和语法分析树的绘制。
- 程序具有正确性。程序可以在各种输入情况下实现正确的分析过程。部分测试数据和结果在 调试分析 一节给出。
- 程序界面友好，提供了各种不同的按钮以使用户操作程序完成不同的功能。在进行词法分析和语法分析时，用户不仅可以看到分析的结果，还可以看到词法分析产生的 token 串，语法分析的移进规约过程以及最后的语法树等等信息。
- 程序具有一定的鲁棒性。比如用户在未读入文件时无法点击按钮进行词法分析和语法分析功能，在词法分析失败时也无法进行语法分析操作等。

程序的源代码和可执行文件均托管在 github 平台上。

<https://github.com/K-D-B/MoonParser1.0>

从完成过程上来看，本次作业工程量不小，具有一定的挑战性，我们在完成过程中有很多收获和成长。

首先，通过动手实践，我们对于词法分析和语法分析的相关知识有了更深的记忆和理解，在完成书面作业中不清晰的细节也在实践过程中变得清晰明了。

其次，在小组合作的过程中，我们对于程序的模块划分，接口定义有了更深的感悟，合作能力和交流沟通能力也有了一定提高。

在实现过程中，我们对 `C++ STL` 中的一些容器的使用和特性，`Qt` 事件过滤器等知识有了更多了解。

通过这次实践过程，虽然我们只是完成了一个简易的类 `C` 语言的编译器前端的一部分，但还是深切感受到了程序编译过程的精巧和严谨，联想到我们实际开发中使用的功能完善的 IDE 和编译器，不禁对计算机科学家和工程师们的智慧感到由衷地敬佩。

## 参考文献

- [1] 陈火旺, 刘春林等. 程序设计语言编译原理 (第三版) [M]. 北京: 国防工业出版社, 2000.