南京理工大学
NANJING UNIVERSITY OF SCIENCE & TECHNOLOGY

软件课程设计(II)
课程项目报告

班　　级 9191062301

学生姓名 孙傲歆

学　　号 919106840333

题　　目 词法、语法、语义分析器的实现

指导教师 项欣光

2022 年 3 月

摘　要

本次课程设计，一共包含三个部分：**词法分析器、语法分析器、语义分析器。**

词法分析器基于 C++进行开发设计。很好地实现了任务 1 所要求的功能。分析器可以读入外部的 3° 型文法文件和源代码文件，并基于此进行词法分析，给出三元式 token 以及非终结符的首符号集合。此外词法分析器还支持复数和科学计数法输入，并且对错误单词进行识别并报错。

语法分析器同样基于 C++进行开发设计。很好地实现了任务 2 所要求的功能。分析器可以读入外部的 2° 型文法和词法分析器输出的三元式 token 序列，并基于此进行语法分析，给出分析结果，若出错还会给出错误所在行数、错误发生的地点并给出大概的错误原因。此外为了更好地体现中间结果，分析器还可输出分析过程中创建的项目集和 Action Goto 表。

语义分析器同样基于 C++进行开发设计。基本实现了任务 3 所要求的的功能。但其不能进行外部文法输入，相当于属性文法写死在了程序当中。但其任可以识别简单的赋值语句、算术表达式、if 语句、for 循环语句以及 while 循环语句。分析器可以对外部三元式 token 文件进行翻译，基于翻译结果输出四元式。

最后，上述三个分析器均使用 Qt creator 进行了图形化界面包装，以进一步提高用户的使用体验，并且使输入-分析-输出这样一个流程更加清晰可见。

本报告分为**概述、词法分析器、语法分析器、语义分析器、图形化界面设计 、心得总结、附录**共七个部分。是一个从浅入深、层层递进的关系。**第一章概述**部分对任务进行了重述并介绍而项目的大概思路以及项目的开发环境。**第二至五章**全部是分析器的具体实现介绍，每一章节都介绍了项目的总体设计思路，然后以模块为单位进行系统设计思路的讲解，然后对代码编写过程中所构建的结构体及函数进行介绍与解释，最后给出运行截图进行功能测试。**第六章心得体会**则是经验总结和对系统开发过程进行反思。

关键字： C++编程； QT 图形化编程；词法分析；语法分析；语义分析

目　录

第一章 概述

1.1 任务重述

任务 1：创建一个词法分析程序，该程序支持分析常规单词。

程序有两个输入：一个文本文档，包括一组 3º 型文法（正规文法）的产生式；一个源代码文本文档，包含一组需要识别的字符串（程序代码）。

程序的输出包括：一个 token（令牌）表，该表由 5 种 token 组成：关键词，标识符，常量，限定符和运算符。

任务 2：创建一个使用 LL(1)方法或 LR(1)方法的语法分析程序。

程序有两个输入：1）一个是文本文档，其中包含 2º 型文法（上下文无关文法）的产生式集合；2）任务 1 词法分析程序输出的（生成的）token 令牌表。

程序的输出包括：YES 或 NO（源代码字符串符合此 2º 型文法，或者源代码字符串不符合此 2º 型文法）；错误提示文件，如果有语法错标示出错行号，并给出大致的出错原因。

任务 3：创建符合属性文法规则的语义分析程序。

程序有两个输入：1）一个是文本文档，其中包含 2º 型文法（上下文无关文法+属性文法，包含语义规则注释，可以简单以表达式计算语义为例）的产生式集合；2）任务 1 词法分析程序输出的（生成的）token 令牌表。

程序的输出包括：四元式序列，可以利用优化技术生成优化后的四元式序列

1.2 项目综述

基于上述任务要求，本次软件课程设计(II)项目一共包括三个部分：一个支持分析常规单词的词法分析程序；一个基于 LR(1)方法的语义分析程序；一个能够分析基本赋值语句、while 语句、for 语句、if 语句的符合属性文法规则的语义分析程序。

词法分析器与语法分析器严格按照任务要求进行编写，都可以读入文法来进行分析，理论上只要文法足够合理且完整，词法分析器和语法分析器所构成的语言系统可以十分接近于我们平常使用的各种高级语言。但由于本人水平有限，创造的文法或多或少会有很多不足之处，还望老师谅解。这里我设计了一种基于 C++语法体系的高级语言系统，整体语法风格十分接近于 C++,并进行了各种优化。词法分析器和语法分析器所用的文法将在附录部分给出。

对于词法分析器作出了如下优化：

- 支持复数赋值
- 支持科学计数法赋值
- 能够对非法的标识符进行识别并报错
- 能够自动略过注释部分

对于语法分析器作出了如下优化：

- 支持 while 语句、for 语句、if 语句
- 能够指出错误所发生的的行数
- 能够指出错误的大概特征

- 能够很好地体现语法处理中间过程，如输出 action-goto 表以及项目集表

而语义分析器则较为简陋，没能很好地到达任务要求。但是其任然可以对输入的源代码进行分析并输出四元式序列，可以分析的语句范围包括：赋值语句、自增自减语句、while 语句、for 语句、if 语句。

1.3 开发环境

语言：C++
系统：windows11
集成开发环境：Dev-C++、Qt creator

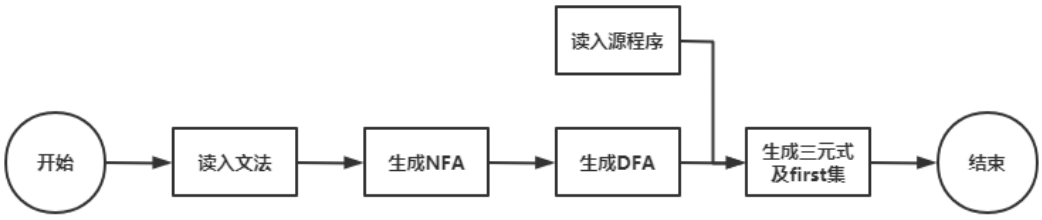
第二章 词法分析器

2.1 总体设计思路

在本项目中，词法分析器的作用是读入字符序列并将其转化为 token 表。为了便于分析，这里我将 token 的类型进一步细分，分成了如下 9 种类型：

- 关键字-keyword（如 return、for、while、break 等）
- 标识符-identifier
- 类型-type（如 int、double、float、void 等）
- 双目运算符-binary_operation（如 ‘+’、‘-’、‘*’、‘/’ 等）
- 单目运算符-unary_operation（如 “++”、“--” 等）
- 限定符-qualifier（只包含 “const”）
- 常量-constant（包括 true、false、None 以及数字）
- 字符串-string
- 分隔符-delimiters（如 ‘;’、‘)’、‘}’ 等）

本项目在思路上遵循题设要求，从头到尾从左至右识别用户输入的源代码，然后根据用户输入的 3° 型文法生成 NFA，然后确定化 DFA，根据确定化的 DFA 识别一个个单词并输出 token 三元组以及每个非终结符的 first 集合。在具体实现过程中，还将构造各种结构体对文法、NFA、DFA 进行存储，提高代码质量以及运行速度，使代码条理清晰更加易读。**下图为词法分析器的运行流程图：**



2.2 系统设计

在本项目中，词法分析器基本上包括四个模块：**一是文法输入模块；二是 NFA 生成模块、三是 DFA 确定化模块；四是源程序输入及分析模块。**下面将分别对这四个模块进行介绍。

2.2.1 文法输入模块

首先进行的必然是文法的输入。本程序所能识别的文法必须是形如<A>->a<B>形式的正规文法，其中 A、B 表示非终结符，a 表示终结符。并且将不同的文法通过“******”进行分块，这样可以有效的解决非终结符符号可能不够用的问题。其次在文法中像 26 个大写字母、26 个小写字母、数字这样的终结符数量过于庞大，且出现频率很高，如果全部写下显得过于繁琐、冗余。所以我设计了一些终结符的省略表示形式，包括：“^d”表示全体数字，“^c”表示全体小写字母，“^C”表示全体大写字母，“^s”表示含有转义字符的字符串，“^S”表示不含转义字符的字符串。

然后对于输入的文法，设计了结构体“GRAMMAR”,其存储了每个文法的左部、右部、长度以及右部是否为单一终结符。然后编写了函数“read_lexer_grammar”对外部文法文件进行读入，一边读入一边存入上述的“GRAMMAR 结构体”。必要的时候还会调用“solve”函数对省略表示的终结符进行处理。

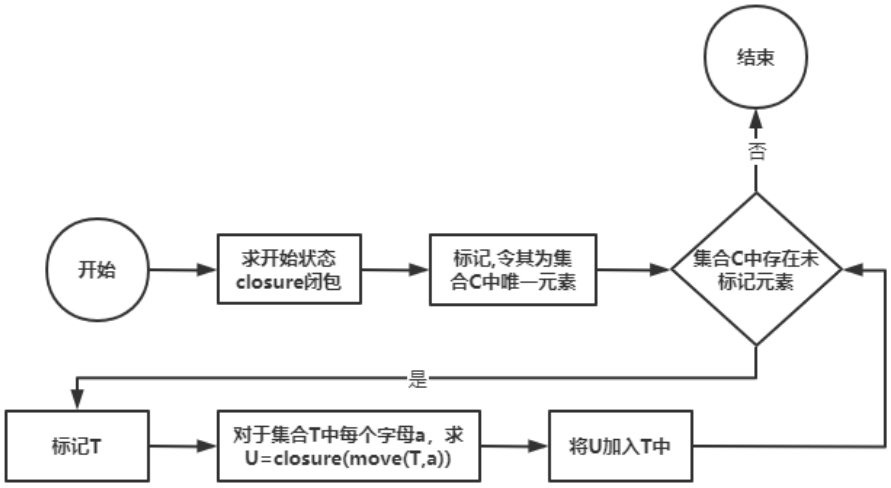
2.2.2 NFA 生成模块

同样是“read_lexer_grammar”函数，在对文法进行存储后，该函数的最后一部分便是通过已经输入好的文法进行 nfa 的构建。这里我设计了两个结构体来进行 nfa 的构建，一个 Edge 结构体和一个 NFA 结构体。这里我的思路是：将 nfa 当做一个有向图来看待，按照构建有向图的方法来构建 nfa，只不过该有向图会有多个头结点及尾结点，但只要对 nfa 中的每个点做好标记即可。

实际上这里我还加入了数组“fir”、“nonterminal_to_num”、“num_to_nonterminal”，其作用分别为存储首符号集合以及将非终结符和数字进行绑定。这一系列工作都是为了给接下来对 NFA 进行确定化生成 DFA 提供便利。以及最后 first 集合的输出。

2.2.3 DFA 确定化模块

在构建了 nfa 之后，下一步便是将 nfa 确定化为 dfa。基本的 nfa 确定化 dfa 的算法流程图，**如下图所示：**



同样地，我又设计了结构体 DFA 来存储 dfa 结构，其实 dfa 和 nfa 的结构声明十分相似。和 nfa 一样，将 dfa 也当做一个有向图来看待，同样使用构建有向

图的方法来构建 dfa。只不过 dfa 只有一个初态结点，而且其结点名是多个数字的集合。程序实际构建确定化 dfa 时，将调用“nfa_to_dfa”函数，该函数基于以上算法调用 nfa 结构体的相关函数，来求 closure 闭包和 move-a 弧转换，直至集合不再扩大。

2.2.4 源程序输入及分析模块

最后便是通过已经制作好的 DFA 对源程序进行分析并输出，这里采取的方法是：一边输入、一边分析、一边输出。也就是说这里我将上述的三个工作全部写入了一个函数“read_lexer_source”中进行。

文法的读入是按行进行的，每次读入都将判断每一行中是否含有注释，若遇到“//”则会直接跳过本行“//”后边的内容，若遇到“/*”则会跳过中间所包含的行，直到遇到“*/”为止。

对于非注释行，将遍历所有 dfa，并运行 dfa 中的相关函数来判断所输入单词的类别。程序这里采取的是“先入为主”以及“长度优先”的策略，即先识别到的 token 类型优先，识别更长长度的 token 类型优先。这样可以有效的介绍单词的识别错误，比如“++”既可以识别成两个双目运算符“+”，也可识别成一个单目运算符“++”，但由于长度优先的策略，系统还是会识别成“++”。

识别完成后便会对识别到的 token 进行输出，输出内容为：所在行号、token 类型、token 内容。这里我还设计了一个特殊的 token 类别“error”，当程序分析出“error”类别将会停止分析，并给出错误原因。

最后，程序还会调用函数“out_first”来输出所有非终结符的 first 集，该 first 集是用于后续的语法分析的，这里将其一并写入输出文件当中，可以有效较少工作量，为后续语法分析器的读入分析提供了便利。

2.3 代码设计

由于本项目的代码量较大，这里便不展示具体代码。我将对该词法分析器程序中所构建的所有结构体的成员变量及成员函数进行功能介绍，并对程序中所出现的所有函数进行解释。

2.3.1 结构体

1.GRAMMAR 结构体

该结构体主要用于存储外部输入的 3° 型文法。具体的成员变量及函数**如下表所示：**

（1）成员变量：

类型	变量名	含义
char[]	left	文法左部
char[][]	val	文法右部
bool[]	is_terminator	右部是否终结符
int	cnt	文法长度

（2）成员函数：

函数名	输入	返回值	功能描述
init	none	void	变量初始化

push_left	char[]	void	拼接左部
push_right	char[]	void	当右部为非终结符时，拼接右部
push_right	char	void	当右部为单一终结符时，拼接右部

2.Edge 结构体

该结构体为辅助性结构体，其主要用于存储前向边，是用于构建 nfa 和 dfa 有向图的。其没有成员函数，只有三个成员变量**如下表所示：**

类型	变量名	含义
int	to、nxt	目标结点，及下一个目标结点的地址
char	val	有向边的值

3.NFA 结构体

该结构体主要用于存储从 3°型文法转换而来的 nfa。具体的成员变量及函数**如下表所示：**

（1）成员变量：

类型	变量名	含义
head[]	head	头结点
int	n	顶点数
int	cnt	边数
Edge[]	E	有向边集合
char[][]	name	存储结点名称
int[]	type	表示结点类型，1 为开始结点，2 为终结点
map<string,int>	mp	创造键值对，节点名称与数字绑定
set<char>	st	某一 nfa 所包含的所有终结符，如数字、大小写字母等等

（2）成员函数：

函数名	输入	返回值	功能描述
init	none	void	变量初始化
add	(int,int,char)	void	加边
add_edge	(char[],int,char[],int,cahr)	void	存储结点名称并添加绑定信息，调用 add 函数加边
setType	(char[],int,int)	void	设置结点类型
closure	int	set<int>	求 closure 闭包
move_closure	(set<int>,char)	set<int>	求 move-a 弧转换

4.DFA 结构体

该结构体主要用于存储由 nfa 确定化后得到的 dfa。具体的成员变量及函数

如下表所示：

（1）成员变量：

类型	变量名	含义
head[]	head	头结点
int	n	顶点数
int	cnt	边数
Edge[]	E	有向边集合
map<int,set<int>>	name	存储结点名称
int[]	type	表示结点类型，1 为开始结点，2 为终结点
map<set<int>,int>	mp	创造键值对，节点名称与数字绑定

（2）成员函数：

函数名	输入	返回值	功能描述
init	none	void	变量初始化
add	(int,int,char)	void	加边
add_edge	(set<int> , set<int>,char)	void	存储结点名称并添加绑定信息，调用 add 函数加边
setType	(set<int>,int)	void	设置结点类型
run	char[]	int	对输入的单词进行匹配，返回最大匹配字符数

2.3.2 函数

该词法分析器所用的主要函数及其功能描述如下表所示：

函数名	输入	返回值	功能描述
solve	(char,bool,bool, int,int,int,int,char[], int,char[],int)	void	文法输入时处理省略表示的字符，并同时构建 nfa
read_lexer_source	none	void	读取源代码,分析并输出三元式 token 结果
read_lexer_grammar	none	void	读取文法，并同时构建 nfa，必要时调用 solve 函数进行省略字符处理
nfa_to_dfa	none	void	nfa 确定化为 dfa
out_first	none	void	输出非终结符 first 集合

2.4 功能测试

我们将使用三组测试样例来测试该词法分析器的功能，详细说明如下：

例 1：简单的复合语句测试（这里以 for 为例）

输入：

```
int main(){
int a=0;
for(int i=0;i<10;i++){
a=a+1;
}
}
```

输出：

```
(line 1,type,int)
(line 1,identifier,main)
(line 1,delimiters,()
(line 1,delimiters,))
(line 1,delimiters,{)
(line 2,type,int)
(line 2,identifier,a)
(line 2,binary_operation,=)
(line 2,constant,0)
(line 2,delimiters,;)
(line 3,keyword,for)
(line 3,delimiters,()
(line 3,type,int)
(line 3,identifier,i)
(line 3,binary_operation,=)
(line 3,constant,0)
(line 3,delimiters,;)
(line 3,identifier,i)
(line 3,binary_operation,<)
(line 3,constant,10)
(line 3,delimiters,;)
(line 3,identifier,i)
(line 3,unary_operation,++)
(line 3,delimiters,))
(line 3,delimiters,{)
(line 4,identifier,a)
(line 4,binary_operation,=)
(line 4,identifier,a)
(line 4,binary_operation,+)
(line 4,constant,1)
(line 4,delimiters,;)
(line 5,delimiters,})
(line 6,delimiters,})
```

第一组测试说明，该词法分析器能正常的处理简单复合语句，并给出相应的三元式 token 序列。

例 2：常量及正负号处理

输入：

```
int main(){
int a=-3;
int b=2E+10;
a=$10+12i;
}
```

输出：

```
(line 1,type,int)                    (line 3,type,int)
(line 1,identifier,main)             (line 3,identifier,b)
(line 1,delimiters,()                (line 3,binary_operation,=)
(line 1,delimiters,))                (line 3,constant,2E+10)
(line 1,delimiters,{)                (line 3,delimiters,;)
(line 2,type,int)                    (line 4,identifier,a)
(line 2,identifier,a)                (line 4,binary_operation,=)
(line 2,binary_operation,=)          (line 4,constant,$10+12i)
(line 2,constant,-3)                 (line 4,delimiters,;)
(line 2,delimiters,;)                (line 5,delimiters,})
```

第二组测试说明，该词法分析器能够正常的处理各种常量，包括负数常量、科学计数法以及复数。这里需要强调的是：为了和 error 类型进行区分，这里约定复数常量必须以$开头来作为复数常量的标识。

例 3：词法错误识别

输入：

```
int main(){
int 3asd=1;
}
```

输出：

```
(line 1,type,int)
(line 1,identifier,main)
(line 1,delimiters,()
(line 1,delimiters,))
(line 1,delimiters,{)
(line 2,type,int)
(line 2,error,3asd)
发现词法错误！数字不能作为标识符的首字母
```

第三组测试说明，该词法分析器能够识别出词法单词错误，并给出错误所在地方和错误原因。

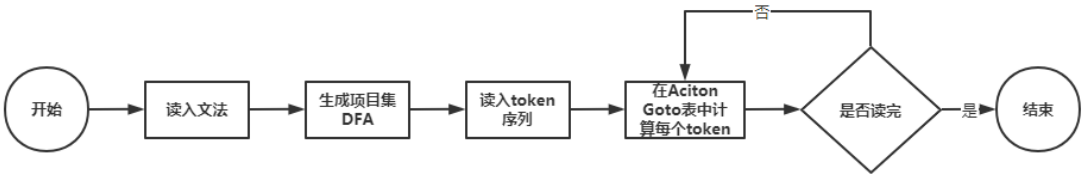
第三章 语法分析器

3.1 总体设计思路

本项目中，语法分析器的作用是对前面词法分析器所输出的 token 三元式进行语法分析，判断其是否满足文法所定义的语义规则。若满足则输出“YES”；若不满足则输出“NO”，并给出出错行数、出错位置以及大概的出错原因。

语法分析程序的处理逻辑为：根据用户输入的 2º 型文法，生成 Action 及 Goto 表，判断 token 序列是否合法。也即，通过读入语法分析器生成的文法，生成项目集的 DFA，再对所有项目集生成对应的 Action Goto 表，再读入语法分析器的输出，对于所有的 token 序列以符号栈和状态栈的形式来在 Action Goto 表上运行，以运行是否正常结束来判断该程序是否符合语法分析器的要求。

下图为语法分析器的运行流程图：



3.2 系统设计

在本项目中，词法分析器基本上包括五个模块：**一是输入模块；二是预处理模块、三是项目集 DFA 生成模块；四是 Action Goto 表生成模块；五是语法分析及结果输出模块。**下面将分别对这五个模块进行介绍。

3.2.1 输入模块

首先进行的是输入模块，该输入模块其实还包含两个部分：**一是词法分析器结果的输入；二是文法的输入。**

前面已经提到过，词法分析器的输出除了 token 三元式，还有一张非终结符的 first 集表。而这个 first 集表就是为语法分析阶段所准备的，这样语法分析器就不必再进行复杂的首符号集运算，而可以直接读入词法分析阶段计算好的结果。程序只需调用函数“read_lexer_out”，即可完成上述的工作，函数首先读入 token 三元式，这里我设计了“TOKEN”结构体来存储读入的 token 三元式，以方便后续的工作。然后我还设计了“NONTERMINAL_FIRST”结构体来存储后续读入的非终结符的 first 集，这里用“-------”来将 token 三元式和 first 集表分开，以防止读入混乱。

然后便是文法的读入。和词法分析器一样，用“<>”将文法的非终结符标记起来，由于此处输入的 2° 型文法，对文法的格式要求也不再像词法分析器那样严格。程序调用函数“read_parser_grammar”，即可完成文法的读入工作。同样地，这里也设计了结构体“GRAMMAR”来方便文法的存储,其存储了每个文法的左部、右部、长度以及右部是否为单一终结符。

3.2.2 预处理模块

由于 LR(1)文法生成项目集十分复杂繁琐，所以在生成项目集 DFA 之前，先进行一定的预处理工作，为项目集生成工作做准备。该预处理模块包含三个小模块：一是计算每个非终结符是否能推出空；二是计算每个非终结符的 first 集；三是统计所有文法。

首先是计算每个非终结符是否能推出空模块。和词法分析器时一样，将每个非终结符和数字唯一绑定，这样可以方便后续的计算。然后，创建一个专门的数组来存储每个非终结符是否能推出空，true 表示能推空、false 表示不能推空。然后，直接调用“cal_empty”函数进行计算，对于文法中，存在形如“<A>->@”（@表示空），则非终结符 A 显然能推出空，这样的非终结符便不再重复计算，直接标记为 true。而对于一般的非终结符则要遍历其所有文法，若其右部存在终结符或右部不能推出空，则不能推出空，标记为 false；而若其有右部可以推出空，则可以推出空，标记为 true。

然后是计算每个非终结符的 first 集模块。虽然在输入模块中，我们已经读入了一些非终结符的 first 集合，但是在语法分析文法中又加入了新的非终结符，对于这些新的非终结符依然需要计算它们的 first 集。程序将调用函数“cal_nonterminal_first”来进行 first 集的计算，同样是将计算结果存入“NONTERMINAL_FIRST”结构体，当然为了避免不必要的重复计算，该结构体设置了相关成员变量来标识非终结符是否已被计算过，若计算过则直接跳过，若没有计算过则将调用函数“cal_first”来进行非终结符 first 集的计算，并将结果存入到结构体当中。

最后是统计所有文法模块。这一个模块的工作十分简单，就是调用函数“cal_all_grammar”来根据文法的左部将文法统一存储在一个 set<GRAMMAR>类型的数组之中，左部相同的文法将被存入在同一个 set 中。这一步的工作主要是为了下面的项目集生成做准备，方便程序按照文法的左部来遍历文法。

3.2.3 项目集 DFA 生成模块

然后是语法分析器的最关键一步，项目集 DFA 的构建。首先先来介绍一下**项目集 DFA 的生成算法：**

以 S′→·S，#属于初始项目集中，把'#'号作为向前搜索符，表示活前缀为 γ(若 γ 是有关 S 产生式的某一右部)要归约成 S 时，必须面临输入符为'#'号才行。我们对初始项目 S′→·S，# 求闭包后再用转换函数逐步求出整个文法的 LR(1)项目集族。具体构造步骤如下：

a)假定 I 是一个项目集， I 的任何项目都属于 CLOSURE(I)。

b) 若有项目 A→α·Bβ,a 属于 CLOSURE(I)，B→γ 是文法中的产生式，β∈V*，b∈FIRST(β,a)， 则 B→·γ,b 也属于 CLOSURE(I)中。

c) 重复 b)直到 CLOSURE(I)不再增大为止

了解了项目集 DFA 的生成算法就要考虑如何在代码层面进行实现。老实说，项目集 DFA 生成模块是该项目的一个“瓶颈”。LR（1）分析法的时间复杂度非常膨胀，这是由于在计算过程中不断地加入项目集导致的。普遍而言，判断是否为空的时间复杂度为$O(n)$，计算其 First 集的时间复杂度为$O(n^2)$，计算 FOLLOW 集的时间复杂度为$O(n^3)$。（其中遍历语句复杂度为一次项，定位非终结符后跟的符号为二次项，合并符号为三次项）。随着语法文法的拓展，项目集的个数随之快速增长，使用上述方法构建的语法分析器的运行时间也会快速增加。经过多次测试，项目集 DFA 生成这一步骤基本上需要 10s 左右才可完成，这也是本项目最大的不足之处。

言归正传，程序将直接调用函数“make_project_set”函数来进行项目集的生成，这里一共创造了三个结构体来进行存储，分别是“DFA”结构体、“PROJECT”结构体、“PROJECT_SET”结构体。值得一提的是，这里的“DFA”结构体基本和词法分析器中一致，只不过其每个结点从存储非终结符集合变成了存储一个项目集。

对于某一个具体的项目集，程序运行逻辑为，首先将文法读入到一个初始的“PROJECT”结构体中，然后调用“project_closure”函数求该项目的闭包，之后一直调用“GO”函数来更新项目集，直至项目集不再扩大，然后将计算完成的项目集存入到“DFA”结构体中。像这样循环往复的进行计算，最终即可计算出所有的项目集，同时也将项目集制作成了 DFA 结构。

项目集 DFA 生成完毕后，还将调用“debug_project_set”函数来对生成好的

项目集进行输出处理，以更好地显示程序处理的中间过程。

3.2.4 Action Goto 表生成模块

项目集 DFA 生成完毕后，就是要根据项目集 DFA 来生成相应的 Action Goto 表。这里创造了结构体“ACTION_GOTO”来对 action_goto 表进行存储。程序会直接调用函数“make_action_goto”来计算 Action Goto 表，然而该函数实际上只有一条语句，即使用“ACTION_GOTO”结构体的成员函数“run”。而 run 函数将根据“DFA”结构体中的各个项目集来计算其具体的 Action 动作（包括 MOVE_IN=1,STATUTE=2,ACC=3,ERR=-1）、当 Action 动作为移进时转入的状态以及 Goto 的状态号。

同样地，在 Action Goto 表生成完毕后，还将调用“debug_action_goto”函数来对生成好的 Action Goto 表进行输出处理，以更好地显示程序处理的中间过程。

3.2.5 语法分析及结果输出模块

最后就是一步便是根据生成的 Action Goto 表进行 LR(1)分析，当然除了 Action Goto 表,LR(1)分析还需要一个符号栈和一个状态栈,这里创造了“SYMBOL”结构体来辅助生成符号栈，而状态栈则用普通的 int 数组即可。**具体的 LR(1)分析算法如下：**

a）遇到 Si:表示“移进”，并转向下一个状态

- 状态栈：i 入栈
- 符号栈：当前输入符号入栈

b）遇到 ri：表示使用第 i 个产生式“规约”（产生式的右部符号个数为 k）

- 状态栈：弹出 k 个状态，栈顶状态与所用产生式的左部查 GOTO 表并将相应的状态入栈
- 符号栈：弹出 k 个符号，并将所用产生式的左部入栈

c）遇到 acc：表示接受该输入串

d）遇到 err：表示出错

程序在进行 LR(1)分析时将调用函数“solve_token”。该函数则是基于以算法对 token 序列从左到右、从上至下进行分析，若遇到“ERR”则调用函数“post_error”来打印相关的出错信息；若遇到“ACC”则调用函数“post_succ”，来打印成功信息。

3.3 代码设计

3.3.1 结构体

1.TOKEN 结构体

该结构体主要用于存储程序输入的词法分析器输出的 token 三元式。具体的成员变量及函数**如下表所示：**

（1）成员变量：

类型	变量名	含义
int	line	所在行数

char[]	type	token 类
char[]	val	token 具体值

（2）成员函数：

函数名	输入	返回值	功能描述
set_token	(int,char[],char[])	void	设置 token

2.GRAMMAR 结构体

该结构体主要用于存储程序输入的 2°型文法。具体的成员变量及函数如下表所示：

（1）成员变量：

类型	变量名	含义
char[]	line	文法左部
char[][]	type	文法右部
bool[]	val	是否为终结符
int	cnt	文法长度
int	id	文法标记

（2）成员函数：

函数名	输入	返回值	功能描述
init	none	void	初始化
push_left	(char[],int)	void	设置文法左部
push_right	char[]	void	设置文法右部
push_right	char	void	设置文法右部，单独 char 则说明时终结符
cal_right	none	int	获取文法右部长度
operator <	const GRAMMAR	bool	小于号重载
write	none	void	文法输出

3.SYMBOL 结构体

该结构体为符号结构体，主要用于程序进行语法分析时建立符号栈。具体的成员变量及函数如下表所示：

（1）成员变量：

类型	变量名	含义
char[]	x	符号内容
bool	is	符号是单一字符还是一串字符

（2）成员函数：

函数名	输入	返回值	功能描述
init	char[]	void	当符号是一串字符时，初始化符号栈内容
init	char	void	当符号是单一字符时，初始化符号栈内容

4. NONTERMINAL_FIRST 结构体

该结构体主要用于存储非终结符的 first 集合。其结构十分简单，只有两个成员变量而无成员函数，具体**如下表所示：**

类型	变量名	含义
set<char>	fir	存储首符号集
bool	is_cal	是否已经计算过

5. PROJECT 结构体

该结构体为项目结构体，主要用于存储项目变量以便于项目集的构建。具体的成员变量及函数**如下表所示：**

（1）成员变量：

类型	变量名	含义
char[]	left	项目集左部
char[][]	val1	右部已规约部分
bool[]	is_terminator1	右部已规约部分是否有终结符
int	cnt1	右部已规约部分长度
char[][]	val2	右部待规约部分
bool[]	is_terminator2	右部待规约部分是否有终结符
int	cnt2	右部待规约部分长度
char	prospect	向前搜索符
int	id	标识

（2）成员函数：

函数名	输入	返回值	功能描述
init	none	void	初始化
operator <	const PROJECT	bool	小于号重载
_set	(GRAMMAR,int, char,int)	void	设置项目，通过文法设置
_set	(char[],char[][], bool[],int,char[][], bool[],int,char,int)	void	设置项目，直接参数设置
change_prospect	char	void	更改向前搜索符
write	none	void	项目输出

6. PROJECT_SET 结构体

该结构体为项目集结构体，主要和前面介绍的 PROJECT 结构体配合进行项目集构建。具体的成员变量及函数**如下表所示：**

（1）成员变量：

类型	变量名	含义
set<PROJECT>	projects	项目集

（2）成员函数：

函数名	输入	返回值	功能描述
init	none	void	初始化

insert_project	PROJECT	void	项目集中插入项目
insert_project	set<PROJECT>	void	项目集中插入多个项目
cnt	none	int	获取项目集个数
operator <	const PROJECT_SET	bool	小于号重载
write	none	void	项目集输出

7.Edge 结构体

该结构体为辅助性结构体，其主要用于存储前向边，是用于构建 dfa 有向图的。其没有成员函数，只有四个成员变量**如下表所示：**

类型	变量名	含义
int	to、nxt	目标结点，及下一个目标结点的地址
char[]	val	有向边的值
bool	is	标记 val 是单一字符还是一串字符

8.DFA 结构体

该结构体其实与词法分析器中的 DFA 基本相同，只不过每结点内容变成了项目集。具体的成员变量及函数**如下表所示：**

（1）成员变量：

类型	变量名	含义
head[]	head	头结点
int	n	顶点数
int	cnt	边数
Edge[]	E	有向边集合
map<int,PROJECT_SET>	name	存储结点名称
PROJECT_SET []	project_set	项目集
map<PROJECT_SET,int>	mp	创造键值对，节点名称与数字绑定

（2）成员函数：

函数名	输入	返回值	功能描述
init	none	void	变量初始化
add	(int,int,char[],bool)	void	加边
add_edge	(PROJECT_SET, PROJECT_SET, char[],bool)	bool	存储结点名称并添加绑定信息，调用 add 函数加边

9.ACTION_GOTO 结构体

该结构体主要用于存储由项目集 DFA 生成的 Action Goto 表。具体的成员变量及函数**如下表所示：**

（1）成员变量：

类型	变量名	含义

int	state_num	状态总个数
int[][]	action	action 动作为 MOVE_IN 时，转入的状态号
int[][]	action_type	action 动作，MOVE_IN=1，STATUTE=2，ACC=3，ERR=-1
int[][]	gt	goto 状态号
bool[]	vis	dfs 遍历时标记已访问

（2）成员函数：

函数名	输入	返回值	功能描述
dfs	int	void	dfs 遍历，寻找移进得情况
run	none	void	制作 action-goto 表

3.3.2 函数

该语法分析器所用的主要函数及其功能描述**如下表所示：**

函数名	输入	返回值	功能描述
insert_nonterminal	char[]	void	插入非终结符
insert_terminal	char	void	插入终结符
cal_all_grammar	none	void	文法统一存储
cal_empty	none	void	计算每个非终结符是否能推出空
cal_first	int	set<char>	计算某一非终结符的 first 集
cal_ nonterminal_first	none	void	计算所有非终结符的 first 集
get_first	(char[][],bool[],int)	set<char>	获取 first 集
project_closure	PROJECT	set<PROJECT>	获取某一项目集的闭包
read_parser_grammar	none	void	读入文法
read_lexer_out	none	void	读入词法分析器输出的三元组 token 和非终结符的 first 集
GO	(PROJECT_SET, char[],bool)	PROJECT_SET	计算项目集
make_project_set	none	void	制作项目集 DFA
debug_project_set	none	void	输出项目集
make_action_goto	none	void	制作 action-goto 表
debug_action_goto	none	void	输出 action-goto 表
write_line	int	void	根据行数输出这一行的源代码

post_error	(TOKEN,int,int)	void	输出错误信息
post_succ	none	void	无语法错误打印成功
solve_token	none	void	对读入的 token 进行分析

3.4 功能测试

我们将使用五组测试样例来测试该语法分析器的功能，（这里的输入将直接给出词法分析的源代码，以便于展示，实际上的真正输入应是词法分析器所输出的三元式 token。且输出实际上除了语法分析结果，还有项目集和 Action Goto 表，但由于这两个文件内容过多在此也不再展示）详细说明如下：

例 1：for 循环语句识别

输入：

```
int main(){
int a=1;
for(int i=0;i<10;i++){
a=a+1;
}
}
```

输出：

```
YES
```

第一组测试说明，该语法分析器能够识一般的 for 循环语句。

例 2：while 循环语句识别

输入：

```
int main(){
int a=1;
while(a<10){
a++;
}
}
```

输出：

```
YES
```

第二组测试说明，该语法分析器能够识一般的 while 循环语句。

例 3：if 语句识别

输入：

```
int main(){
int a=1;
if(a<10){
int b=1;
}
}
```

输出：

```
YES
```

第三组测试说明，该语法分析器能够识一般的 if 语句。

例 4：嵌套语句识别（以 while-if 嵌套为例）

输入：

```
int main(){
int a=1;
while(a<10){
if(a==2){break;}
a=a+1;
}
}
```

输出：

```
YES
```

第四组测试说明，该语法分析器能够识嵌套语句。

例 5：语法报错（这里以缺少分隔符为例）

输入：

```
int main(){
int a=1;
int b=2
a=a+1;
}
```

输出：

```
NO
----line information----
line 3: int b = 2
line 4: a = a + 1 ;
line 5: }
-------------------------
error line: 4
error value: a
----expected symbol-----
;
-------------------------
```

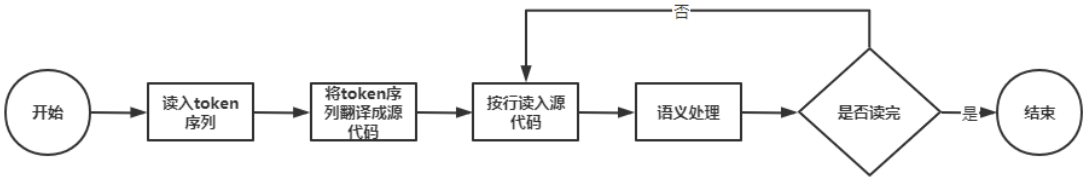
第五组测试说明，该语法分析器能够识语法错误，并给出错误行数、错误位置以及大概错误原因。

第四章 语义分析器

4.1 总体设计思路

由于本人水平有限，加之对语义分析这一步骤的流程理解不够深刻，所以该项目的语义分析器部分较为潦草。其既支持源代码的直接输入也支持任务 1 词法分析器输出的三元式 token 的输入，但并不支持文法的输入。也就是说该语义分析器的属性文法“写死”在了程序当中。

虽然没能完全实现任务 3 所要求的的所有功能，但是该语义分析器任然能够识别出简单的赋值语句、算术表达式、for 语句、while 语句以及 if 语句。该语义分析器的分析逻辑为先处理三元式 token 将其翻译成源代码形式，然后读入源代码进行语义分析并输出四元式。**下图为语义分析器的运行流程图：**



4.2 系统设计

在本项目中，词法分析器基本上包括三个模块：**一是 token 输入及翻译模块；二是源代码输入模块；三是语义分析及输出模块。**下面将对这三个模块进行介绍。

4.2.1 token 输入及翻译模块

该模块还可以分为两个小模块：**一是 token 输入模块；二是 token 翻译模块。**

这里的 token 输入模块沿用了语法分析器的 token 输入模块。程序只需调用函数“read_lexer_out”，即可完成 token 的读入工作，这里同样使用了“TOKEN”结构体来存储读入的 token 三元式，以方便后续的工作。

读入完 token 三元式序列后边进行 token 三元式的翻译工作。这里我编写了两个函数来完成翻译工作，首先是函数“write_line”，该函数将按照行号输出这一行的源代码，为了保证格式的规整，这里还会进行 token 类型的判断，若 token 类型为“type”（即 int、float、double 之类的单词）将会在单词后输入空格，其余的单词则不会。然后函数“out_source”将调用函数“write_line”输出所有行的源代码到指定文件。

4.2.2 源代码输入模块

该模块工作十分简单，就是将前一模块输出的源代码读入，每次读一行，在读的时候将判断该行代码的类型，这里我简单的分成了“if 类型”、“for 类型”、“while 类型”、“other 类型”一共四个类型，根据不同的行类型转入不同的处

理函数。

4.2.3 语义分析及输出模块

前面已经提到该模块一共分为四个小模块，分别是：**“if 类型”语句处理模块、“for 类型”语句处理模块、“while 类型”语句处理模块、“other 类型”语句处理模块。**

首先是最简单的“other 类型”语句处理模块。这一模块处理的主要是一些赋值语句和算术表达式。程序直接调用函数“solve_other”函数进行处理，该函数会以“;”为分界进行处理。首先判断其是赋值语句还是算术表达式。若是赋值语句将去除其前置的类型声明，然后调用函数“solve_expression”函数来处理一般表达式。若是算术表达式则直接调用“solve_expression”来进行处理。输出相关的四元式。

其次是“if 类型”语句处理模块。这一模块处理的主要是 if 表达式，程序直接调用函数“solve_if”函数进行处理。由于 if 语句涉及 goto 行数的计算，所以函数将先计算 if 语句内嵌代码所需要占用的函数，以此来计算 if 语句失效所要跳转的行数。对于“{}”内的内嵌语句则直接调用上述“solve_other”函数进行处理。

其次是“while 类型”语句处理模块。这一模块处理的主要是 while 表达式，程序直接调用函数“solve_while”函数进行处理。其实 while 语句的处理和 if 语句十分相似，只不过在 while 块的出口语句的前语句需要加入一条 goto 语句跳转回 while 条件判断行。对于“{}”内的内嵌语句同样调用上述“solve_other”函数进行处理。

最后是“for 类型”语句处理模块。这一模块处理的主要是 while 表达式，程序直接调用函数“solve_for”函数进行处理。相比较 if 类型语句的处理，除了需要加入一条 goto 语句跳转回 for 条件判断行之外，还需在 for 语句条件判断行之前加入 for 语句内部的赋值语句的翻译。其次对于 for 语句中变量的自增或自减也需要处理，所有 for 语句的翻译是最为复杂、繁琐的。而对于“{}”内的内嵌语句同样调用上述“solve_other”函数进行处理。

4.3 代码设计

4.3.1 结构体

1.TOKEN 结构体

该结构体主要用于存储程序输入的词法分析器输出的 token 三元式。具体的成员变量及函数**如下表所示：**

（1）成员变量：

类型	变量名	含义
int	line	所在行数
char[]	type	token 类
char[]	val	token 具体值

（2）成员函数：

函数名	输入	返回值	功能描述
set_token	(int,char[],char[])	void	设置 token

4.3.2 函数

该语义分析器所用的主要函数及其功能描述**如下表所示：**

函数名	输入	返回值	功能描述
read_lexer_out	none	void	读入词法分析器输出的三元组 token
write_line	int	void	根据行数输出这一行的源代码
out_source	none	void	token 翻译，输出源代码
get_num	string	int	获取需要增加的行数
solve_expression	string	void	分析一般赋值语句
solve_other	string	void	分析赋值语句、声明语句以及算术表达式
solve_if	string	void	分析 if 语句
solve_while	string	void	分析 while 语句
solve_for	string	void	分析 for 语句
read_source_analyze	none	void	一边读入、一边分析、一边输出

4.4 功能测试

我们将使用三组测试样例来测试该语义分析器的功能，（这里的输入将直接给出源代码，以便于展示。前面也已经说过实际上该语义分析器不仅可以根据源代码进行分析也可以根据三元组 token 进行分析）详细说明如下：

例 1：if 语句的分析

输入：

```
int a=1;
if(a<10){a=a+1;}
int b=2;
```

输出：

```
100:a=1
101:if not a<10 goto 104
102:temp0=a+1
103:a=temp0
104:b=2
105:
```

第一组测试说明，该语法分析器能够识一般的 if 语句，并输出其四元式代码。

例 2：while 语句的分析

输入：

```
int a=1;
while(a<10){a=a+1;}
int b=2;
```

输出：

```
100:a=1
101:if not a<10 goto 105
102:temp0=a+1
103:a=temp0
104:goto 101
105:b=2
106:
```

第二组测试说明，该语法分析器能够识一般的 while 语句，并输出其四元式代码。

例 3：for 语句的分析

输入：

```
int a=1;
int b=2;
for(int i=1;i<10;i++){a=a+1;}
```

输出：

```
100:a=1
101:b=2
102:i=1
103:if not i<10 goto 109
104:temp0=a+1
105:a=temp0
106:temp1=i+1
107:i=temp1
108:goto 103
109:
```

第三组测试说明，该语法分析器能够识一般的 for 语句，并输出其四元式代码。

第五章 图形化界面设计

5.1 总体设计思路

基于前面所做的词法分析器、语法分析器、语义分析器，我使用 Qt creator 对它们进行图形化界面包装，以提高用户的使用体验。

由于 Qt creator 的底层语言是 C++,而上述的词法分析器、语法分析器、语

义分析器都是基于 C++语言编写的，所以在代码移植方面并没有什么较大的困难。需要更改的地方主要就是一些文件输入输出逻辑的转换，以及 QString、string、char*类型之间的相互转换。

5.2 系统设计

由于是对上述三个分析器的图形化包装，所以该部分自然分为了三个模块，即：**词法分析器模块、语法分析器模块、语义分析器模块。**

5.2.1 词法分析器模块

首先是词法分析器模块，该模块即是对第二章中所介绍的词法分析器进行图形化界面包装，所以代码的内核部分在此便不再赘述。下面将着重介绍在输入输出以及结果显示方面与源程序的不同之处。

在输入方面，该词法分析器提供了一个输入框，用户可以直接在输入框中输入源代码。而文法的输入则是提供文件对话框来读取外部文件，因为考虑到文法比较复杂繁琐，让用户来输入显得不太合理。输入完成后，用户可以点击“开始分析”按钮，系统将通过 connect 设置按钮点击事件调用相关函数，来进行词法分析。最后，在输出方面，词法分析器提供了一个输出框，三元式 token 结果将直接输入到输出框中，输入框与输出框的配合使得词法分析过程变得直观明了。此外，系统还提供了结果保存功能，点击保存按钮同样会弹出文件对话框，用户可以以 txt 文件形式保存输出结果。

5.2.2 语法分析器模块

其次是语法分析器模块，该模块即是对第三章中所介绍的语法分析器进行图形化界面包装，所以代码的内核部分在此便不再赘述。下面将着重介绍在输入输出以及结果显示方面与源程序的不同之处。

其实这里的移植是存在一定问题的，本来我是打算和词法分析器一样，直接将代码内核写入 Qt 源码当中，通过 connect 设置按钮点击事件调用相关函数，但是同样的代码在正常的 C++环境中运行无误，到 Qt 中就会报内存不足错误。最后的解决方案是将语法分析器的 exe 文件放入到项目文件夹中，然后设置“打开驱动”按钮，该按钮的功能是打开语法分析器 exe 文件所在的文件夹（以下简称为“文件夹”），然后用户再手动双击 exe 文件，进行分析。

这里 token 三元式和文法读入的处理也比较特殊，系统同样提供文件对话框来读取外部文件，不过在读取完外部文件后，经过简单的处理，系统会将文件保存在上述文件夹中，以供文法分析使用。同样地，输出文件也会被输出到文件夹中，这时点击“读入结果”，系统会将文件夹中的输出文件读入到界面设置的三个输出栏中，这三个输出栏分别用于输出分析结果、ACTION-GOTO 表以及项目集表。此外，语法分析器同样提供了保存功能，这里提供了三个保存按钮，分别为“保存 A-G 表”、保存项目集、保存结果。点击相应保存按钮就可保存相应的结果。

5.2.3 语义分析器模块

最后是语义分析器模块，该模块即是对第四章中所介绍的语义分析器进行图

形化界面包装，所以代码的内核部分在此便不再赘述。下面将着重介绍在输入输出以及结果显示方面与源程序的不同之处。

这里的移植包装和词法分析器基本相同，都是直接将代码写 Qt 源码当中。在输入方面，该语义分析器提供了一个输入框，用户可以直接在输入框中输入源代码。此外，用户还可以选择读入 token 三元式来进行分析。点击“token 读入”按钮，会弹出文件对话框。读入 token 文件后，系统会先将 token 三元式翻译成源代码形式输入至输入框。输入完成后，用户可以点击“开始分析”按钮，系统将通过 connect 设置按钮点击事件调用相关函数，来进行语义分析。最后，在输出方面，该语义分析器和词法分析器一样提供了一个输出框来输出结果，同时也可以点击“保存结果”按钮对输出框内容进行保存。

5.3 功能测试

此处的功能测试不再给出具体例子，而是展示具体的图形化界面并介绍每个分析器的具体使用方法和流程。

5.3.1 界面展示

1. 主界面

2. 词法分析器界面

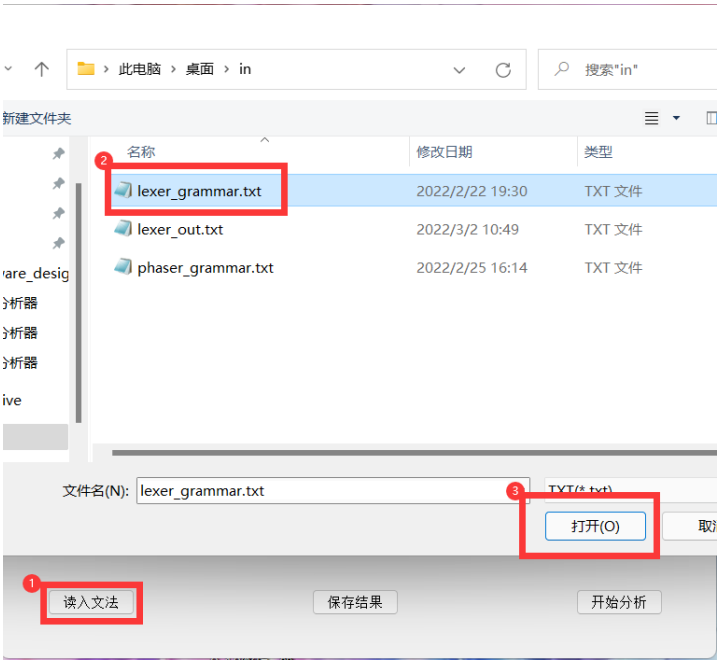
3.语法分析器界面

4.语义分析器界面

5.3.2 词法分析器功能测试

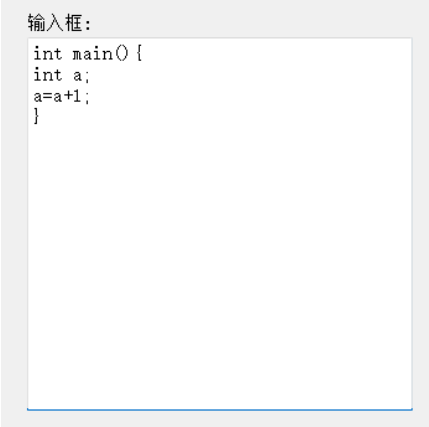
Step1：文法读入

如下图，点击“读入文法”，然后再文件对话框中选择相应文件并点击“打开”即可。



Step2：源代码的输入

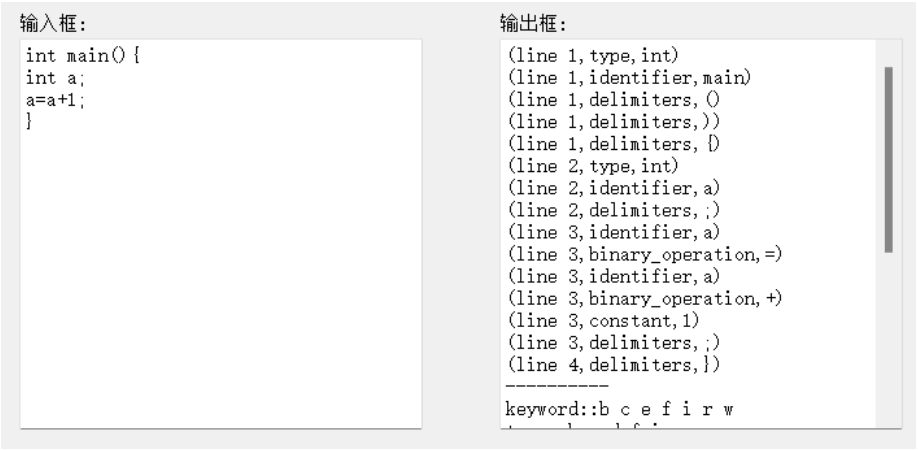
如下图，在输入框中输入想要分析的源代码即可。



Step3：开始分析

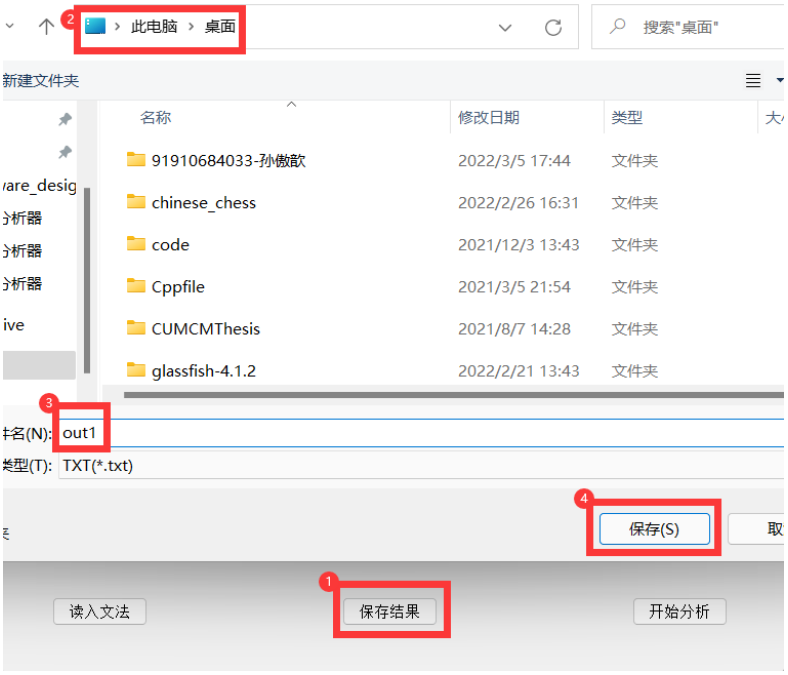
如下图，点击右下角的“开始分析”按钮即可开始进行词法分析。

如下图，分析结果将写入到输出框中。



Step4：结果保存

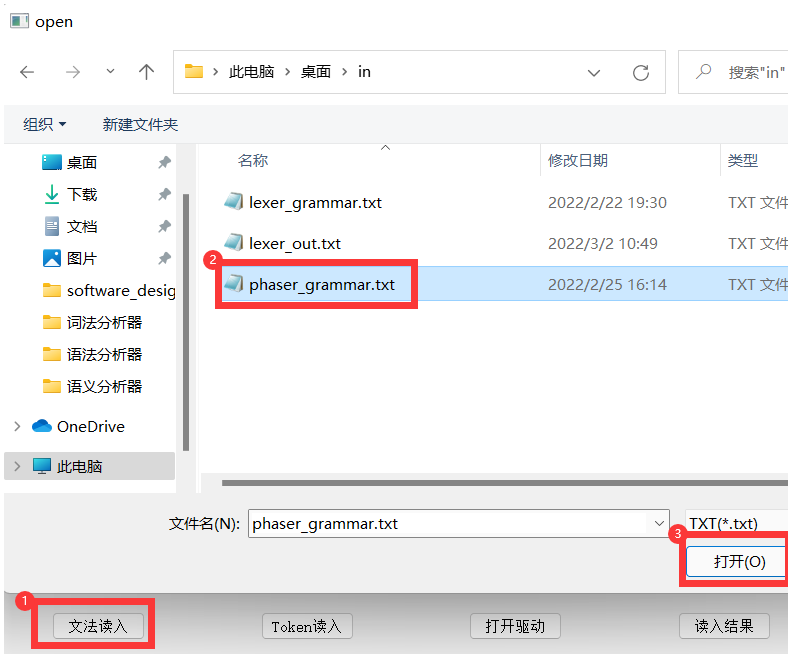
如下图，点击“保存结果”按钮，选择保存路径，输入文件名，点击保存即可。



5.3.3 语法分析器功能测试

Step1：文法读入

如下图，点击“文法读入”，然后在文件对话框中选择相应文件并点击“打开”即可。

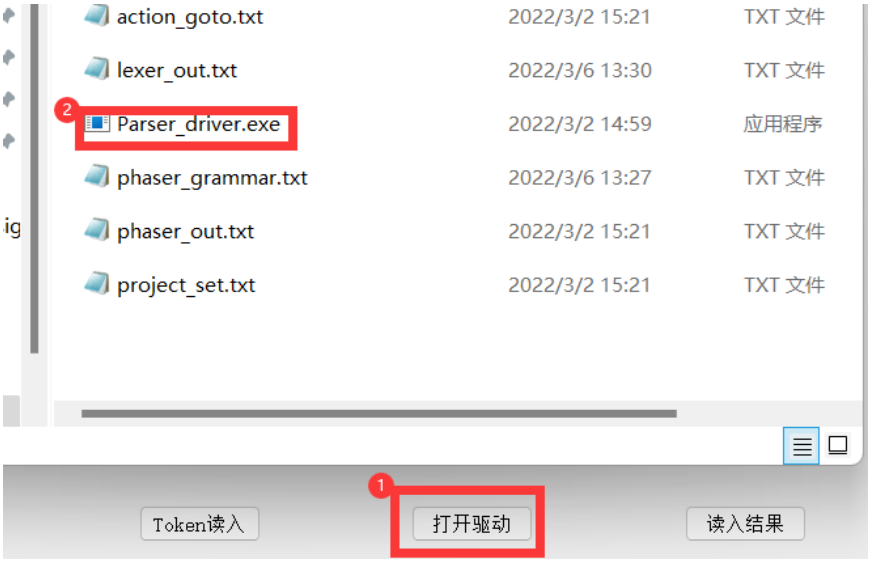


Step2：Token 读入

和**文法读入步骤**完全一致，在此便不再赘述。

Step3：打开驱动程序，进行分析

如下图，点击“打开驱动”按钮，然后双击“Parser_driver.exe”文件即可开始分析。

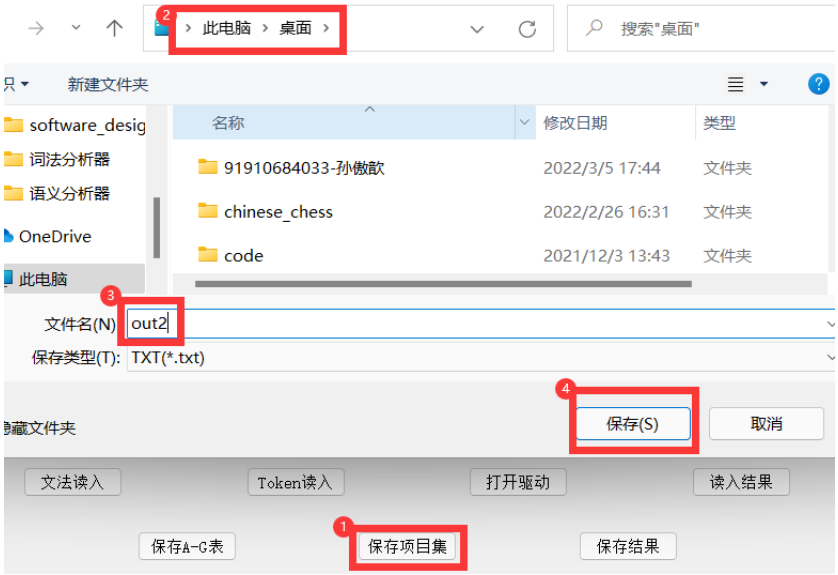


Step4:读入结果

如下图，分析完成后，点击“读入结果”按钮，程序运行结果将显示于三个输出框内。

Step5：结果保存

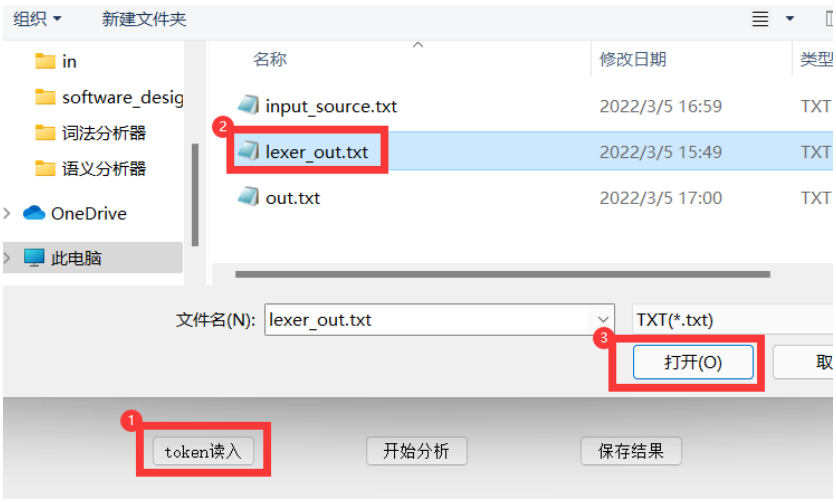
如下图，点击任意一个结果保存按钮（这里以保存项目集为例），选择保存路径，输入文件名，点击保存即可。



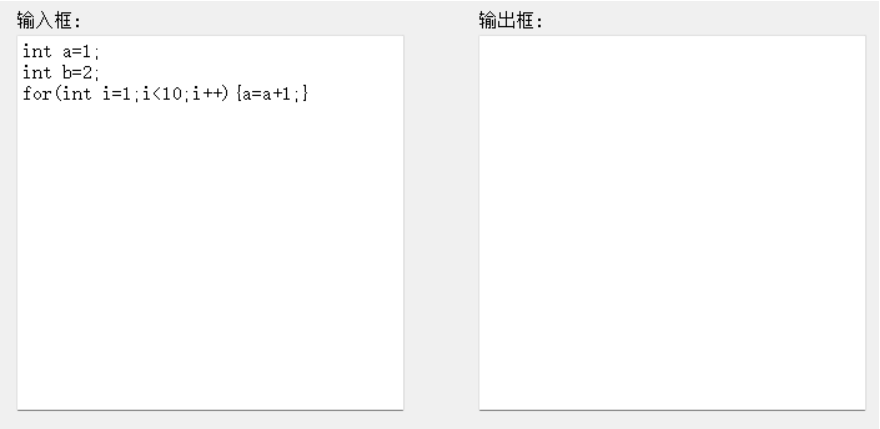
5.3.4 语义分析器功能测试

Step1：token 读入及翻译

如下图，点击“token 读入”，然后在文件对话框中选择相应文件并点击“打开”即可。



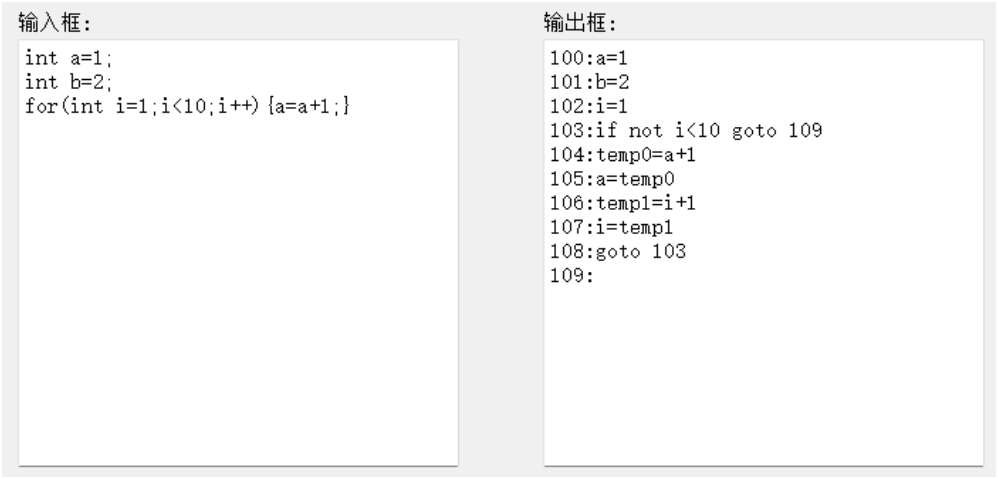
如下图，读入 token 后，会将翻译后的源代码写入至输入框。当然也可以不读如 token 文件，直接在输入框进行编辑。



Step2：开始分析

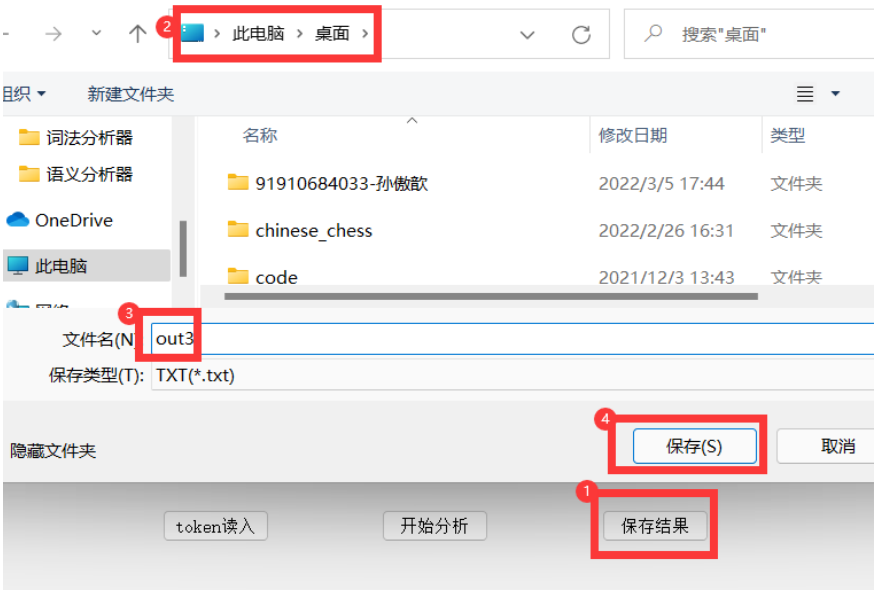
如下图，点击右下角的“开始分析”按钮即可开始进行词法分析。

如下图，分析结果将写入到输出框中。



Step3：结果保存

如下图，点击“保存结果”按钮，选择保存路径，输入文件名，点击保存即可。



第六章 心得总结

本次课设中，总共完成了词法分析、语法分析、语义分析这三个模块的编写。并使用 Qt 对这三个模块进行了图形化界面包装，代码量总计约 2500 行。像以前的《C++课程设计》以及《软件课程设计(I)》都是以小组的形式来进行的，而本次课程设计是个人项目。也就是说从项目构思到代码编写再到最后报告的撰写都需要一个人来完成，这对我来说是一次不小的挑战。

虽说是次挑战，但总的来说，我觉得我是很好地完成了这次课设任务的。本次课设中我综合运用了上学期《编译原理》课程中的各种知识，将理论知识例如 NFA 的构建、NFA 确定化为 DFA、项目集的构建、Action Goto 表的构建等等与实

际编码相结合。此外，本次课程设计选择使用 C++编写，实际上本人对于 python、C#、java 等编程语言也有些许了解，但都没有 C++掌握的牢靠。而且在《C++课程设计》中曾系统地学习过了 Qt 的图形化界面编程，所以用 Qt 进行图形化包装也比较方便。

其实本次课程设计的代码中并没有使用什么高深的算法，其思路都比较“简单”、“暴力”，说到底最困难的还是究竟如何实现。就比如 NFA 确定化为 DFA，可能我们在手算时，在草稿纸上画上几笔，简单计算一下，结果就出来了。但真的通过代码实现就没有想象中那么简单了。所以，为了让代码更加简单易懂、更具有结构性，也为了在编写具体算法时调用各种数据更加便利，我在编程时使用了大量的结构体，基本上每一步都是各种结构体之间的相互转换与计算。这种结构性编程的思想，也确实地减轻了编码和调试时的各种困难。

总问言之，本次课程设计让我受益匪浅。首先，本次课设提高了我独自思考问题并解决问题的能力，从项目构思到编码思想再到最后报告撰写都由一人完成，虽然很辛苦，但也确实地感受到了解决难题的快乐。其次，本次课设还帮助我重温了以前学过的知识，首先自然是上学期的《编译原理》所学的知识，老实说几个月过去了，课程所学的知识忘了不少，正好借此机会进行了复习。其次，大一时所学的 Qt 图形化界面编程的相关知识也得到了进一步巩固与提高。最后，本次课程设计让我明白了理论知识与实际操作、编码相结合的重要性，只会空读书，只是理论知识丰富是完全不够的，应该多进行编程实践，提高自己的编程能力与解决实际问题的能力，这样才算是一个合格的大学生。

在最后，感谢老师在本次课程设计中的指导与所做的努力，祝老师身体健康，科研顺利。

附　录

附录 1：词法分析器文法

<keyword> -> r<D>
<D> -> e<E>
<E> -> t<F>
<F> -> u<G>
<G> -> r<H>
<H> -> n
<keyword> -> f<I>
<I> -> o<J>
<J> -> r
<keyword> -> w<L>
<L> -> h<M>
<M> -> i<N>
<N> -> l<O>
<O> -> e
<keyword> -> b<P>
<P> -> r<Q>

<Q> -> e<R>
<R> -> a<S>
<S> -> k
<keyword> -> c<T>
<T> -> o<U>
<U> -> n<V>
<V> -> t<W>
<W> -> i<X>
<X> -> n<Y>
<Y> -> u<Z>
<Z> -> e
<keyword> -> i<a>
<a> -> f
<keyword> -> e<b>
<b> -> l<c>
<c> -> s<d>
<d> -> e

<type> -> i<A>
<A> -> n<B>
<B> -> t
<type> -> f<C>
<C> -> l<D>
<D> -> o<E>
<E> -> a<F>
<F> -> t
<type> -> d<G>
<G> -> o<H>
<H> -> u<I>
<I> -> b<J>
<J> -> l<K>
<K> -> e
<type> -> c<L>
<L> -> h<M>
<M> -> a<N>
<N> -> r
<type> -> b<O>
<O> -> o<P>
<P> -> o<Q>
<Q> -> l
<type> -> v<R>
<R> -> o<S>
<S> -> i<T>
<T> -> d
<type> -> s<U>

<U> -> t<V>
<V> -> r<W>
<W> -> i<X>
<X> -> n<Y>
<Y> -> g

<binary_operation> -> +<A>
<binary_operation> -> -<A>
<binary_operation> -> *<A>
<binary_operation> -> /<A>
<binary_operation> -> %<A>
<binary_operation> -> &<B>
<binary_operation> -> |<C>
<binary_operation> -> ^<A>
<binary_operation> -> <<A>
<binary_operation> -> ><A>
<binary_operation> -> =<A>
<binary_operation> -> !<D>
<A> -> @
<A> -> =
<B> -> @
<B> -> =
<B> -> &
<C> -> @
<C> -> =
<C> -> |
<D> -> =

<unary_operation> -> +<A>
<unary_operation> -> -<B>
<unary_operation> -> !
<unary_operation> -> ~
<A> -> +
<A> -> @
<B> -> -
<B> -> @

<qualifier> -> c<A>
<A> -> o<B>
<B> -> n<C>
<C> -> s<D>
<D> -> t

<constant> -> T<A>
<A> -> r<B>
<B> -> u<C>

<C> -> e
<constant> -> F<D>
<D> -> a<E>
<E> -> l<F>
<F> -> s<G>
<G> -> e
<constant> -> N<H>
<H> -> o<I>
<I> -> n<J>
<J> -> e

<delimiters> -> (
<delimiters> ->)
<delimiters> -> {
<delimiters> -> }
<delimiters> -> [
<delimiters> ->]
<delimiters> -> ,
<delimiters> -> ;

<identifier> -> _<A>
<identifier> -> ^c<A>
<identifier> -> ^C<A>
<A> -> _<A>
<A> -> ^c<A>
<A> -> ^C<A>
<A> -> ^d<A>
<A> -> @

<constant> -> 0<B>
<constant> -> 1<B>
<constant> -> 2<B>
<constant> -> 3<B>
<constant> -> 4<B>
<constant> -> 5<B>
<constant> -> 6<B>
<constant> -> 7<B>
<constant> -> 8<B>
<constant> -> 9<B>
<constant> -> -<A>
<constant> -> $<H>
<A> -> ^d<B>
<B> -> ^d<B>
<B> -> .<C>
<C> -> ^d<E>
<E> -> ^d<E>
<B> -> e<D>

<B> -> E<D>
<B> -> @
<E> -> e<D>
<E> -> E<D>
<E> -> @
<F> -> ^d<F>
<F> -> @
<D> -> +<G>
<D> -> -<G>
<G> -> ^d<F>
<H> -> ^d<H>
<H> -> +<H>
<H> -> -<H>
<H> -> i

<string> -> "<A>
<A> -> <^d<A>
<A> -> ^C<A>
<A> -> ^c<A>
<A> -> ^s<A>
<A> -> \<B>
<B> -> ^s<A>
<B> -> \<A>
<B> -> "<A>
<A> -> "
<string> -> '<C>
<C> -> ^d<C>
<C> -> ^C<C>
<C> -> ^c<C>
<C> -> ^S<C>
<C> -> \<D>
<D> -> ^S<C>
<D> -> \<C>
<D> -> '<C>
<C> -> '

<error> -> 0<A>
<error> -> 1<A>
<error> -> 2<A>
<error> -> 3<A>
<error> -> 4<A>
<error> -> 5<A>
<error> -> 6<A>
<error> -> 7<A>
<error> -> 8<A>

<error> -> 9<A>
<A> -> ^c<B>
<A>-> ^d<B>
<A> -> ^C<B>
<B> -> ^c<B>
<B>-> ^d<B>
<B> -> ^C<B>
<B>-> @

附录 2：语法分析器文法

<start> -> <program>
<program> -> <function><program>
<program> -> @
<function> -> <type><identifier>(<parameter>){<statements>}
<parameter> -> <qualifier><type><identifier>
<parameter> -> <type><identifier>
<parameter> -> <qualifier><type><identifier>,<parameter>
<parameter> -> <type><identifier>,<parameter>
<parameter> -> @
<statements> -> <statement><statements>
<statements> -> @
<statement> -> <return_statement>
<statement> -> <for_statement>
<statement> -> <while_statement>
<statement> -> <if_statement>
<statement> -> <assignment_statement>
<statement> -> <type><identifier>;
<statement> -> <type><identifier>=<constant>;
<statement> -> break;
<statement> -> continue;
<return_statement> -> return<item>;
<item> -> <string>
<item> -> <constant>
<item> -> <identifier>
<for_statement> -> for(<for_condition>){<statements>}
<for_statement> -> for(<for_condition>)
<while_statement> -> while(<condition>){<statements>}
<while_statement> -> while(<condition>)
<if_statement> -> if(<condition>){<statements>}
<if_statement> -> if(<condition>)
<for_condition> -> <type><identifier>=<constant>;<condition>;<identifier>++
<condition> -> <item><predicate><item>
<condition> -> <boolean_expression><concatenation_condition>

<boolean_expression> -> <constant>
<predicate> -> <
<predicate> -> <=
<predicate> -> ==
<predicate> -> >=
<predicate> -> >
<predicate> -> !=
<arithmetic_expression> -> (<arithmetic_expression>)
<arithmetic_expression> -> <item><operator><arithmetic_expression>
<arithmetic_expression> -> <item>
<operator> -> +
<operator> -> -
<operator> -> *
<operator> -> /
<operator> -> %
<operator> -> &
<operator> -> |
<operator> -> ^
<assignment_statement> -> <identifier><assignment><arithmetic_expression>;
<assignment> -> =
<assignment> -> <operator>=