同濟大學
TONGJI UNIVERSITY

Tongji University

Department of Computer Science and Technology

类 C 语言中间代码生成器

Author:
陈开煦 何征昊 黎可杰

An Assignment submitted for the Tongji University:

CS100395 编译原理

2023 年 1 月 7 日

目录

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、语法分析和中间代码生成，输出分析结果、分析过程、语法树和中间代码。

1.2 程序任务输入

程序输入: 一套 ANSI YACC C99 语言的文法规则和一个 C 语言代码的文本文件。

文法支持的 token 表如下

token	类型	值
AUTO	保留字	auto
BREAK	保留字	break
CASE	保留字	case
CHAR	保留字	char
CONST	保留字	const
CONTINUE	保留字	continue
DEFAULT	保留字	default
DO	保留字	do
DOUBLE	保留字	double
ELSE	保留字	else
ENUM	保留字	enum
EXTERN	保留字	extern
FLOAT	保留字	float
FOR	保留字	for
GOTO	保留字	goto
IF	保留字	if
INT	保留字	int
LONG	保留字	long
RETURN	保留字	return
SHORT	保留字	short
SIGNED	保留字	signed
SIZEOF	保留字	sizeod

STATIC	保留字	static
STRUCT	保留字	struct
SWITCH	保留字	switch
TYPEDEF	保留字	typedef
UNION	保留字	union
UNSIGNED	保留字	unsigned
VOID	保留字	void
VOLATILE	保留字	volatile
WHILE	保留字	while
IDENTIFIER	标识符	由字母，下划线和数字 (0-9) 组成
CONSTANT	数字、常量字符、字符串	支持十六进制、八进制和科学计数法
...	运算符	ELLIPSIS
»=	运算符	RIGHT_ASSIGN
«=	运算符	LEFT_ASSIGN
+=	运算符	ADD_ASSIGN
-=	运算符	SUB_ASSIGN
*=	运算符	MUL_ASSIGN
/=	运算符	DIV_ASSIGN
%=	运算符	MOD_ASSIGN
&=	运算符	AND_ASSIGN
^=	运算符	XOR_ASSIGN
|=	运算符	OR_ASSIGN
»	运算符	RIGHT_OP
«	运算符	LEFT_OP
++	运算符	INC_OP
–	运算符	DEC_OP
->	运算符	PTR_OP
&&	运算符	AND_OP
||	运算符	OR_OP
<=	运算符	LE_OP
>=	运算符	GE_OP
==	运算符	EQ_OP
!=	运算符	NE_OP
;	运算符	;
,	运算符	,
:	运算符	:
=	运算符	=

.	运算符	.
&	运算符	&
!	运算符	!
~	运算符	~
-	运算符	-
+	运算符	+
*	运算符	*
/	运算符	/
%	运算符	%
<	运算符	<
>	运算符	>
^	运算符	^
|	运算符	|
?	运算符	?
[	界符	[
]	界符	]
;	界符	;
(	界符	(
)	界符	)
{	界符	{
}	界符	}

文法的语法部分共包含 229 个产生式，涉及个 68 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 *ANSI_YACC_C*99*.txt* 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, 在语法分析进行规约的同时进行语法制导的中间代码生成。假如词法分析成功，显示分析成功的 token 串；假如语法分析和静态语义检查成功，我们需要显示移进规约过程和对应的语法树，并在可执行文件根目录下输出写有中间代码的文件。

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、静态语义检查和前端 UI 设计四个部分。接下来，我们使用 ANSI Yacc C99 的文法，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，语法分析和语义分析的部分，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 *Visual Studio* 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:

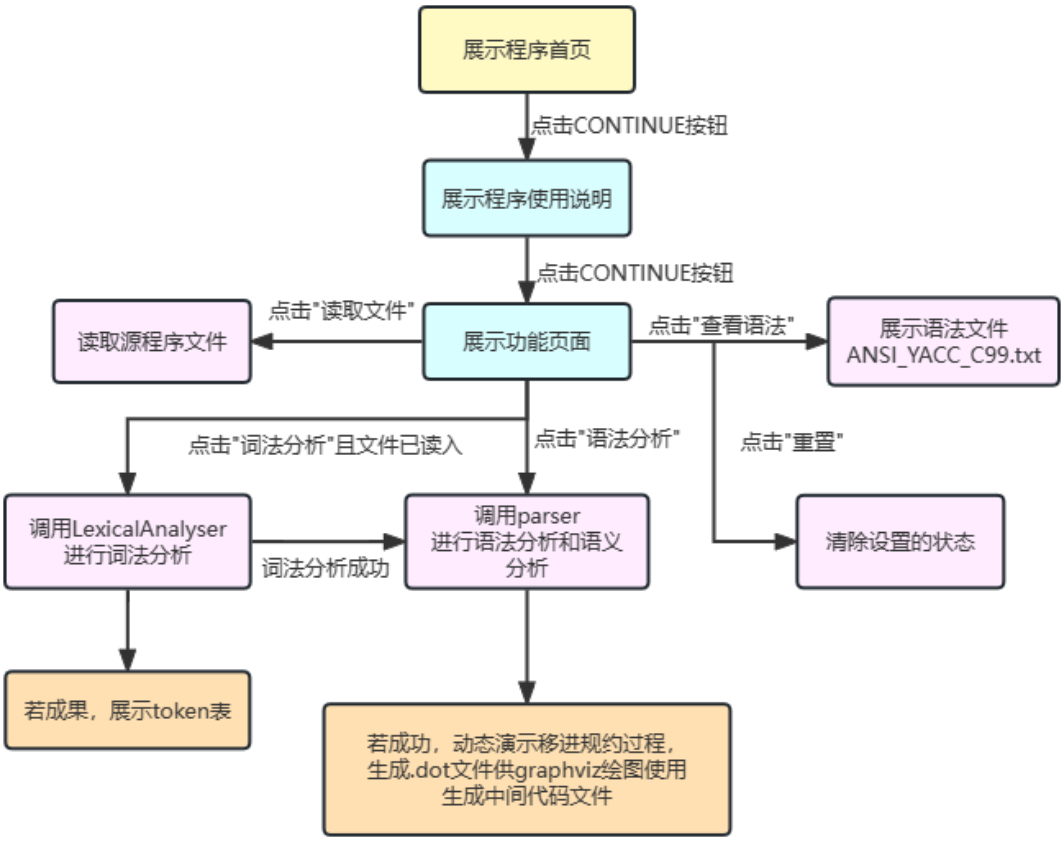


图 1: 主程序流程图

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和语义分析的步骤。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、语义分析、语义分析以及前端 UI 设计四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，我们会给出这四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、调用七类识别 Token 的 DFA 进行 Token 的识别，将识别得到的 Token 串与可能有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 与运算符 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，把报错信息通过 *std* :: *tuple* 返回。

词法分析程序总体框图如下:

图 2: 词法分析程序总体框图

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、常量字符和常量字符串；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，在数字可接受之后还读入其他字符）。

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*,*Constchar* 和 *literalstring* 并以此生成对应七个类，分别为:

ReservedWordAnalyzer, *IdentifierAnalyzer*, *DigitAnalyzer*, *DelimiterAnalyzer* 以及 *OperatorAnalyzer*,*const_char_DFA* 和 *literal_string_DFA*，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。

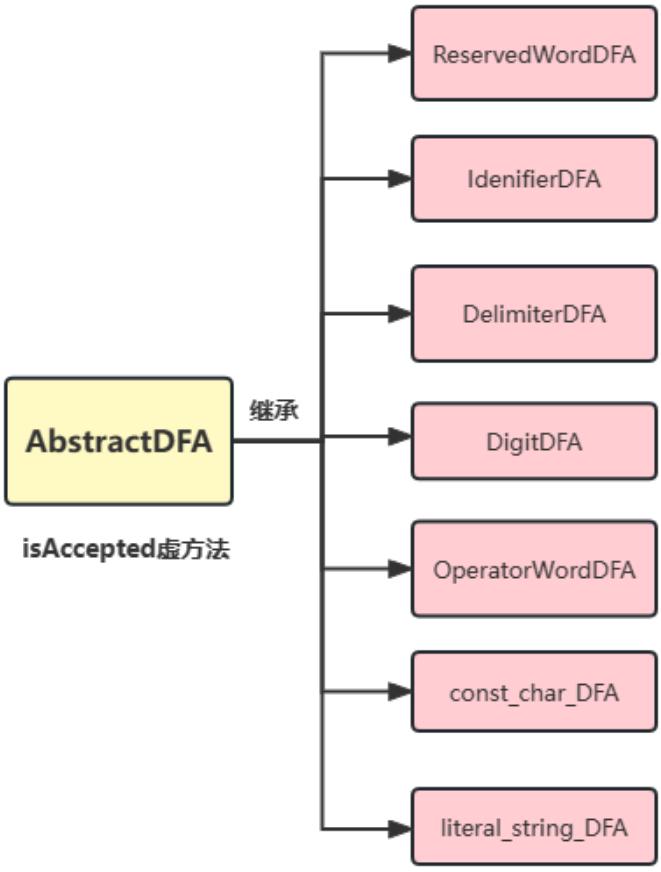


图 3: 识别各 token 的 DFA 关系

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,const_char_dfa,literal_string_dfa 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.tmp 中。
2. 预处理错误信息以 tuple 的形式返回。
3. 预处理后的词法分析错误信息以 tuple 的形式返回，tuple 中包含行、列和一些具体的报错提示。

2.3.6 模块间的调用关系

具体见下图。

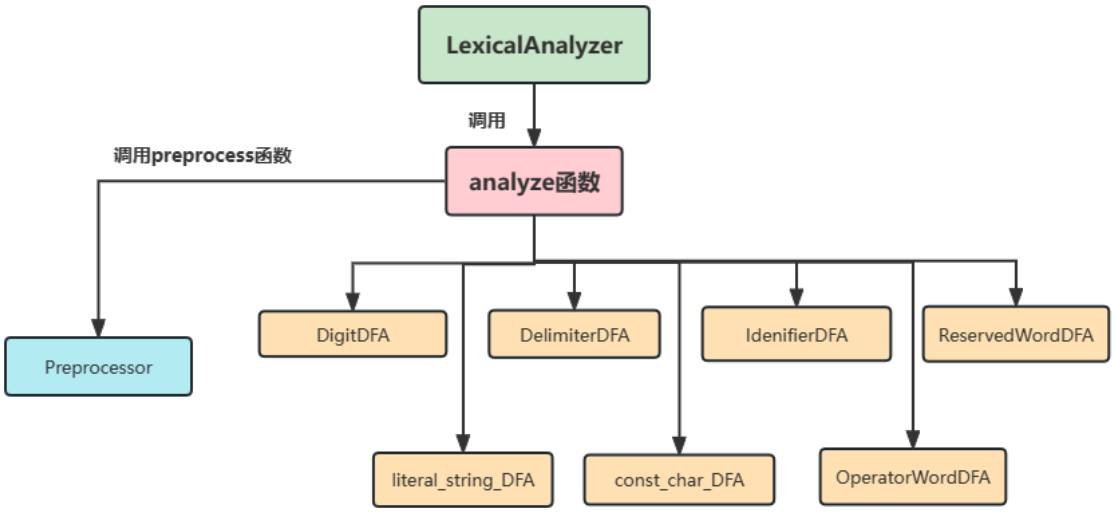


图 4: 词法分析器模块间的调用关系

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 ANSI_YACC_C99.txt, 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 $[A \to \alpha.B\beta, a]$ 属于 CLOSURE(I),$B \to \xi$ 是一个产生式，那么对于 $FIRST(\beta a)$ 中的每个终结符 b, 如果 $B \to .\xi, b$ 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 ε , 则把 ε 加入到 FIRST 集合中。
   (b) 假如 A 右侧的第一个符号为非 ε 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 ε, 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 ε, 则把该终结符的 FIRST 集中除了 ε 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 ε 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 ε, 则将 ε 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

$$start \rightarrow .translation\_unit\ ,\#$$

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 1820 个状态，转换关系十分复杂，因此我们不在报告中对于该 DFA 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。

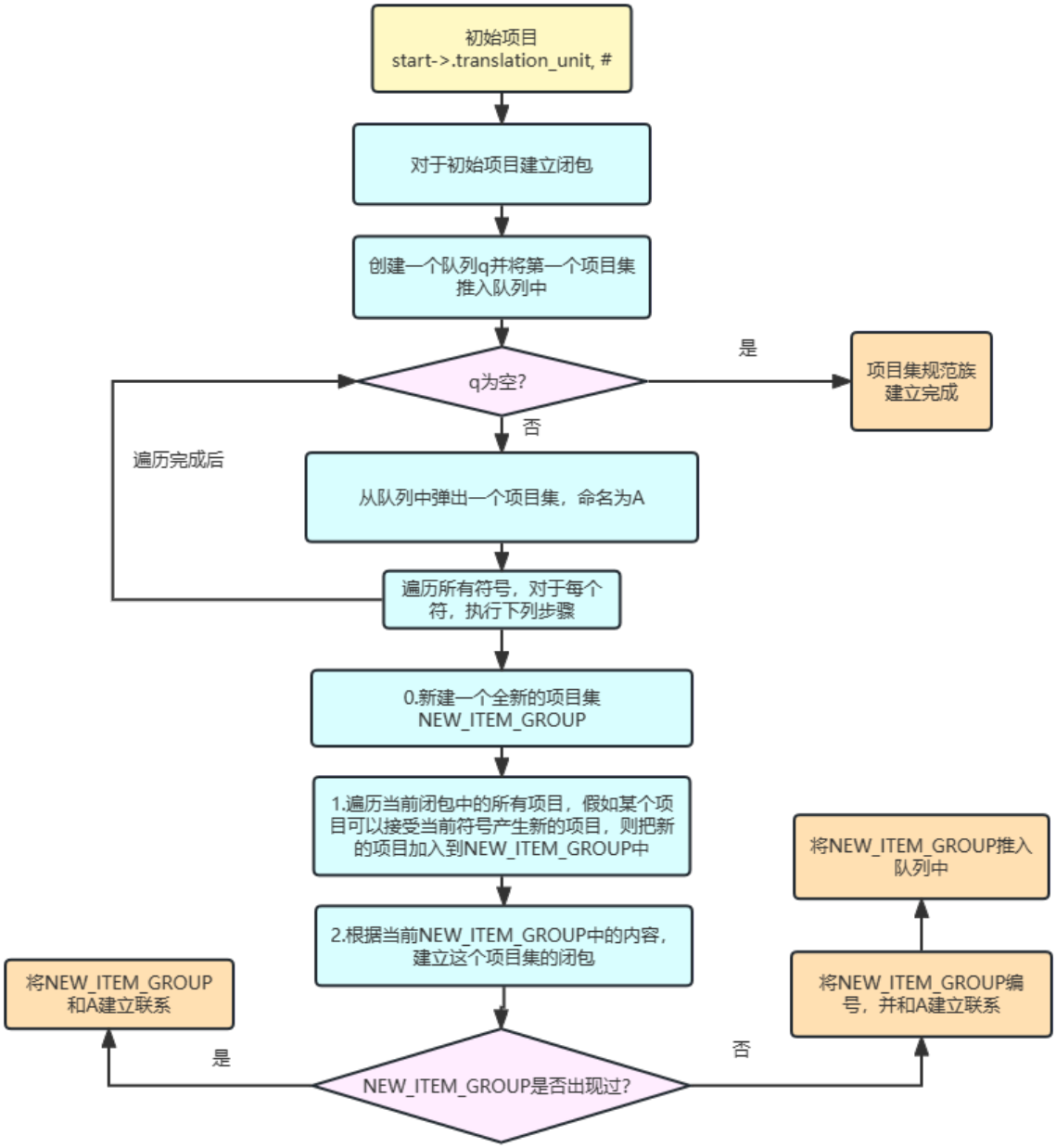


图 5: 项目集 DFA 构造流程图

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-归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一些细节将在下一节 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 $A \to \alpha.a\beta, b$ 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 $A \to .\alpha, a$ 的格式且满足 $GO(I_k, a) = I_j$, 那么置 $ACTION[K, a] = r_j$。

    其中 $A \to \alpha$ 为文法的第 j 条产生式。
  - 若项目为 $start \to translation\_unit.$,#, 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 $GOTO[k, S] = j$。

在建立项目集规范族以及 ACTION-GOTO 的时候，我们发现，由于我们使用的这套文法十分全面和细致，共有 229 条产生式，导致项目集规范族及其相互转换的 DFA 十分庞大，共有 1820 个状态。因此为了加快构建 ACTION-GOTO 表的速度，优化用户体验，我们采用了缓存机制，将 ACTIOON-GOTO 表的信息输入到 cache 文件中，程序只有当无法检测到 cache 时才会从头开始构建 ACTION-GOTO 表。

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：

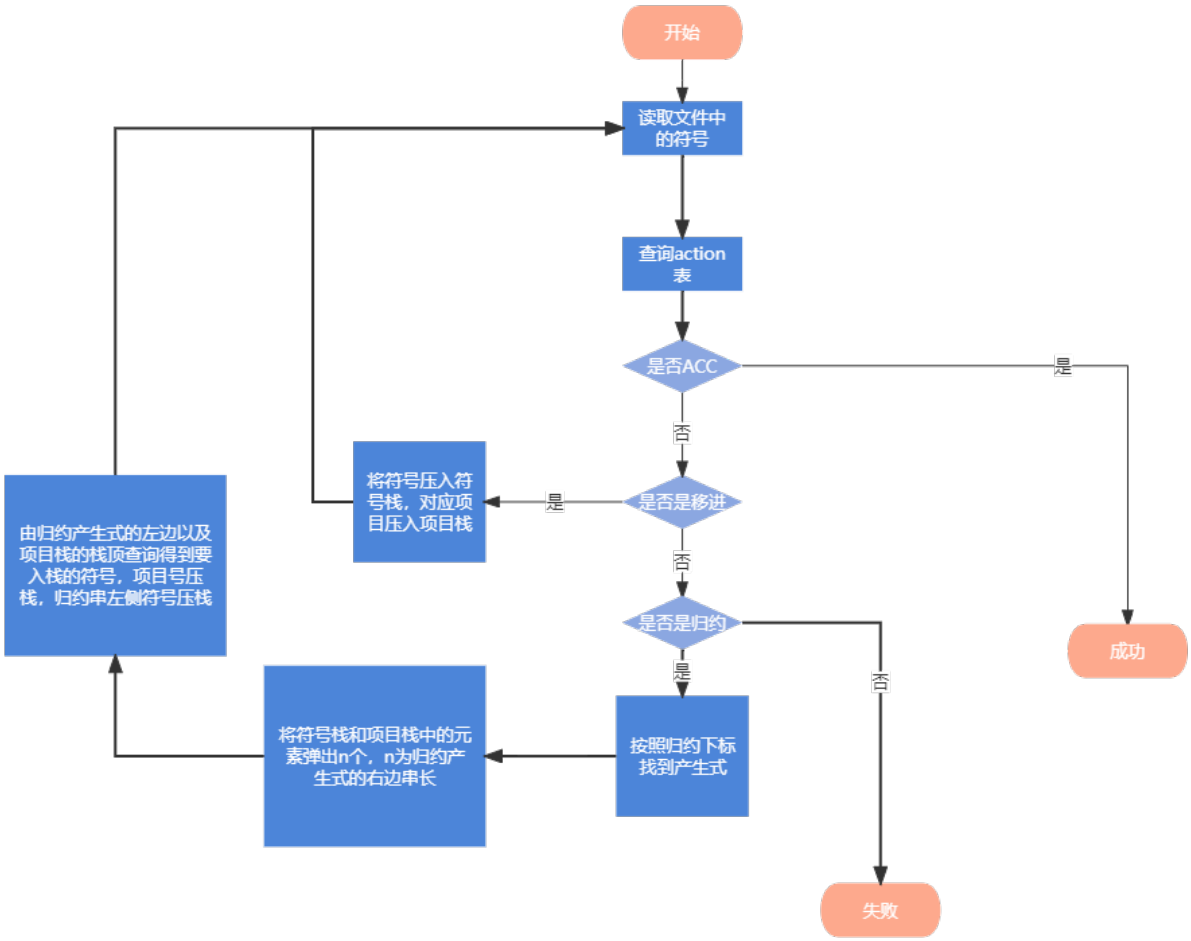


图 6: 移进规约过程流程图

2.5 语义分析 (生成中间代码)

2.5.1 具体流程总纲

采用一遍扫描与回填技术，采用四元式形式。例如，约定

- 四元式 (jnz, a, -, p) 表示 if a goto p
- 四元式 (jrop, x, y, p) 表示 if x rop y goto p
- 四元式 (j, -, -, p) 表示 goto p

整个语义分析包含如下几个部分，如下图所示，限于时间原因，我们并没有完整地实现过程调用的全过程，因此在这次的报告中暂时不做说明，接下来，我们会对说明语句、赋值语句、布尔表达式以及控制语句进行逐一说明。

2.5.2 说明语句

说明语句：定义变量常量的语句，可能会在定义的时候赋初值，也可能仅仅是声明。

说明语句的翻译是不会产生四元式的，也就是说，说明语句只有语义动作，不产生中间代码。

说明语句主要的标识符：int bool double short usigned int 等；

说明语句在初始化的时候既可以用常量初始化，也可以用变量或者表达式来初始化，对于几种情况对应的不同的产生式进行语义动作的解释说明。

首先在本文法中，用到的说明语句的产生式主要有：（只列出最主要的，因为一条赋值语句涉及到的产生式在本文法中非常的多，这一点从生成的语法树可以看到）：

init_declarator_list –> init_declarator

init_declarator_list –> init_declarator_list , init_declarator

init_declarator –> declarator

init_declarator –> declarator = initializer

其中后两条条产生式的主要功能就是实现声明和实现声明时赋初值。

2.5.3 赋值语句

赋值语句需要做的事有两件，一件是计算左值，然后将计算得到的值存在临时变量中，第二件事是把这个值赋给左值，如果右边不是算式，那么可以省掉第

一步，直接赋值。

赋值语句涉及到的产生式非常多，但是有很多在性质上是重复的，这里就按照上述的两种情况分别介绍这两种情况中对应的一些具体的例子。

首先涉及到的最主要的产生式如下：

assignment_expression –> conditional_expression

assignment_expression –> unary_expression assignment_operator assignment_expression

第一条产生式主要是为了给不用计算直接赋值的情况，例如:a=b;

第二条产生式主要是为了给计算后赋值的情况，例如:a=b+1;

这里需要注意的点有赋值表达式右边的变量都必须在符号表中登记过，也就是必须要先声明过才可以引用。

2.5.4 布尔表达式

布尔表达式：用布尔运算符把布尔量、关系表达式联结起来的式子。

- 布尔运算符: and , or , not;
- 关系运算符:$<, \leq, =, \neq\ >, \geq$.

使用的 utility function 有 makelist,emit,backpatch,merge 四个以及全局变量 nextquad，具体如下:

- 变量 nextquad: 它指向下一条将要产生但尚未形成的四元式的地址 (标号)。nextquad 的初值为 1，每当执行一次 emit 之后，nextquad 将自动增 1。
- 函数 makelist(i)：它将创建一个仅含 i 的新链表，其中 i 是四元式数组的一个下标 (标号)；函数返回指向这个链的指针。
- 函数 merge(p1,p2): 把以 p1 和 p2 为链首的两条链合并为一，作为函数值，回送合并后的链首。
- 过程 backpatch(p, t): 其功能是完成“回填”，把 p 所链接的每个四元式的第四区段都填为 t。

```
1 void makelist(int quad, std::set<int>* dst);
2 void emit(string type, string a, string b, string addr);
3 int backpatch(std::set<int>list, string addr);
4 void merge(std::set<int>* list1, std::set<int>* list2, std::set<int>* dst);
```

本文具体对 M,N 的处理进行了一定的改进，控制语句同理。

具体地, 将 N 的 nextlist 等信息存入 else 里，使其在移进的时候就进行 N 的语义动作。

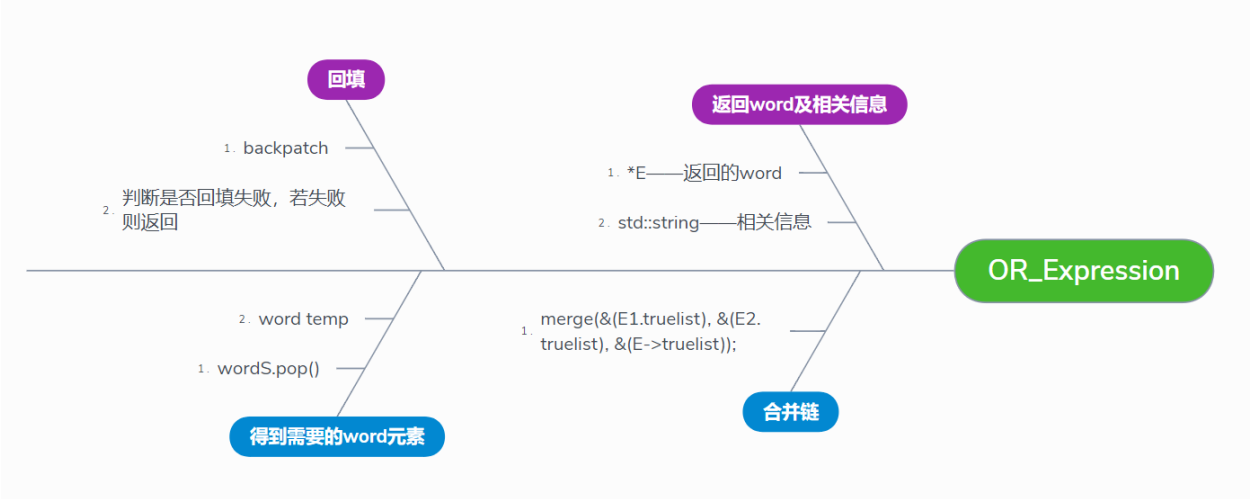
其中，各 bool 表达式对应的产生式如下:

- E->E1 or E2:
- E->E1 and E2:
- E->not E1:
- E->id1 relop id2
- E->id

将其对应到本文采用的文法，则分别为

- 58: logical_or_expression->logical_or_expression OR_OP logical_and_expression
- 56: logical_and_expression->logical_and_expression AND_OP inclusive_or_expression
- 20: unary_expression->unary_operator cast_expression
- 4: primary_expression->(expression)
- 42: relational_expression->relational_expression < shift_expression
- 43: relational_expression ->relational_expression > shift_expression
- 44: relational_expression ->relational_expression LE_OP shift_expression
- 45: relational_expression-> relational_expression GE_OP shift_expression
- 47: equality_expression->equality_expression EQ_OP relational_expression
- 48: equality_expression->equality_expression NE_OP relational_expression

下图为 OR_expression 的流程，其余布尔表达式类似，就不赘述。



2.5.5 控制语句

使用的 utility function 同布尔表达式。

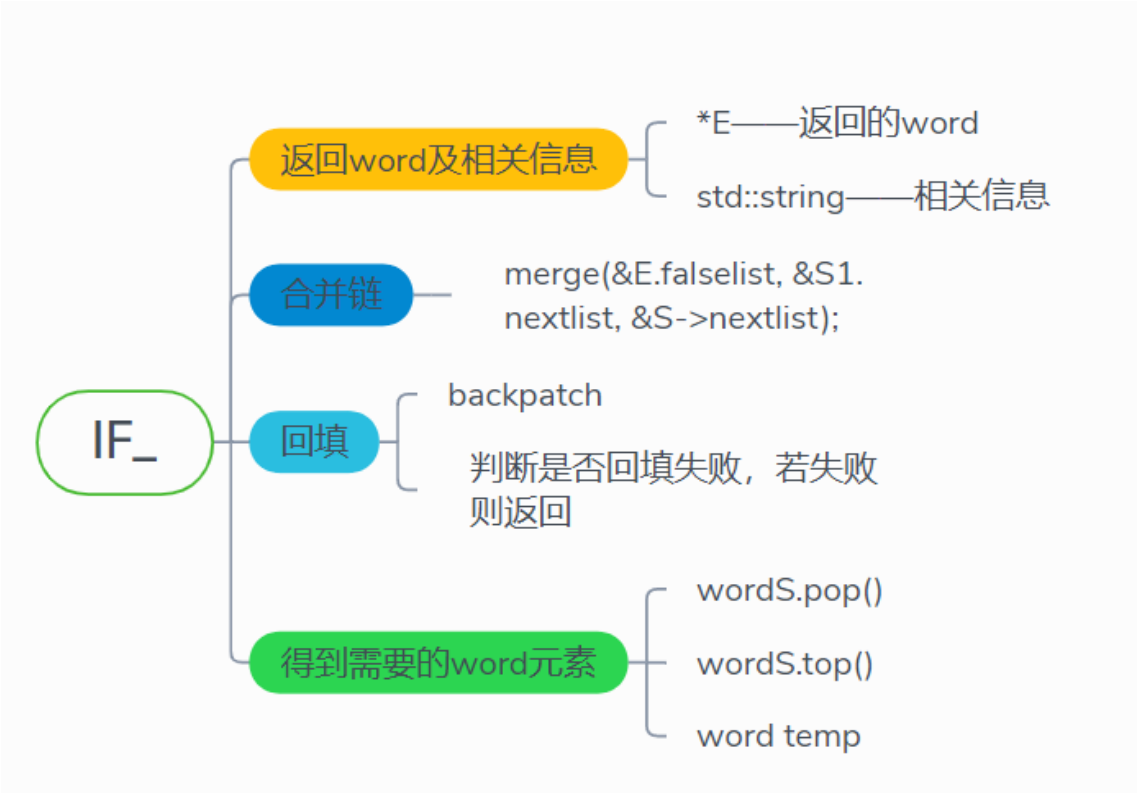
其中，各控制语句对应的产生式如下:

- S->if E then S1
- S->if E then S1 else S2
- S->while E do S1
- L->L1;S

将其对应到本文采用的文法，则分别为

- 207: selection_statement-> IF (expression) statement
- 208: selection_statement-> IF (expression) statement ELSE statement
- 210: iteration_statement-> WHILE (expression) statement
- 206: expression_statement -> expression ;
- 205 : expression_statement ->;

下图为 IF_ 的流程，其余控制语句类似，就不赘述。



2.6 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”，“语法分析和静态语义分析”，“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。
- “语法分析和静态语义分析”: 对经过词法分析的 Token 串进行语法分析，在规约过程中进行静态语义检查并生成中间代码。UI 上会显示是否分析成功，如果分析成功，则会显示分析栈和 token 串中首符号的动态变化过程，并在根目录下生成中间代码文件。
- “查看语法”：显示语法文件 *ANSI_YACC_C*99*.txt* 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。

2.7 四个模块之间的数据交互

四个模块之间数据交互示意图如下所示。

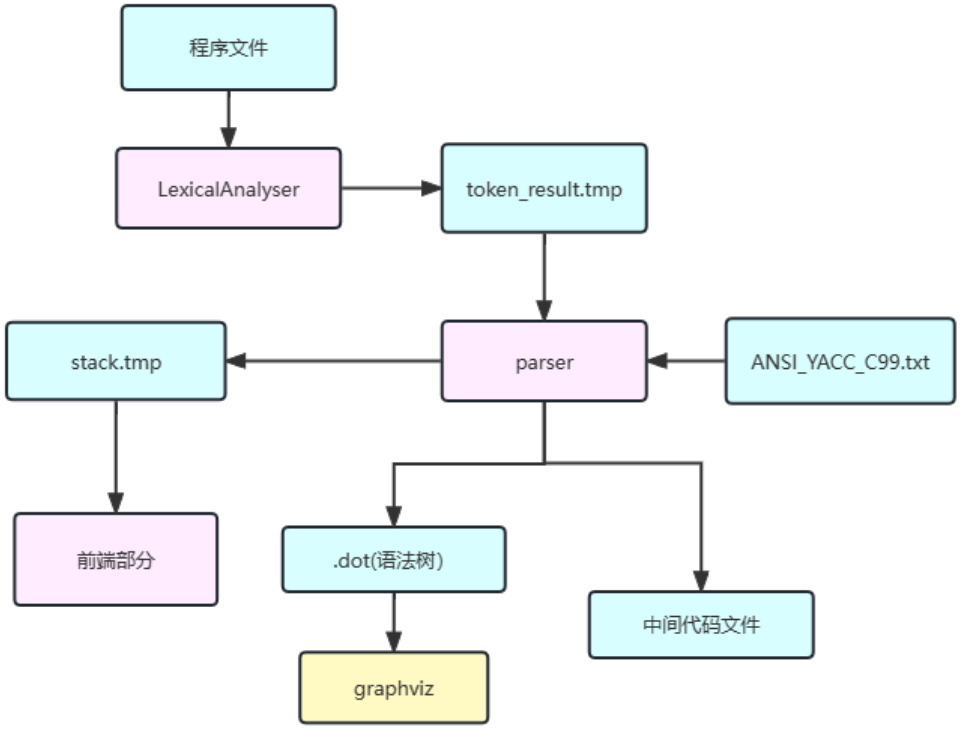


图 7: 数据传输示意图

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 以及在各个类里的具体实现**。具体函数声明如下:

```
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，比如"int", "double", "char", "float", "break", "continue",” while", "if", "else", "for", "return",” auto" 等。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，由于只需要满足该符号由字符、数字和下划线组成，且首字符不是数字，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，下列代码演示了我们读取到'-' 这个运算符后，对于'-','–','-=' 和'->' 四个运算符的识别过程。

```
if (ch == '-')
{
    char ch1 = str[pos + 1];
    end = pos + 1;
    if (ch1 == '-')
    {
        token_content = "--";
        token_type = "DEC_OP";
    }
    else if (ch1 == '=')
    {
        token_content = "-=";
        token_type = "SUB_ASSIGN";

    }
    else if (ch1 == '>')
    {
        token_content = "->";
        token_type = "PTR_OP";
    }
    else
```

```
22      {
23          end = pos;
24          token_content = token_type = "-";
25      }
26  } //perfix= –,-=,-,->
```

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 $std::string\ Delimiter[DELIMITER\_NUM]$ 表, 判断是否为保留字。

数字识别 DFA DigitAnalyzer：

对于数字的识别，我们在参考了 ANSI Yacc C99 的语法文件，将所有可能出现的数字常量总结为几种正则文法。

首先,我们需要介绍这些数字常量正则文法中出现的一些符号:IS,H,D,L,P,E,FS。它们的含义如下:

- D: [0-9]
- L: [a-zA-Z]
- H: [a-fA-F0-9]
- E: ([Ee][+-]?D+)
- P: ([Pp][+-]?D+)
- FS:(f|F|l|L)
- IS: ((u|U)|(u|U)?(l|L|ll|LL)|(l|L|ll|LL)(u|U))

所有可能出现的数字常量的正则文法如下，包含了十进制、十六进制、八进制、科学计数法等情况

1. 十六进制数: 0[xX]H+IS?
2. 十六进制数: 0[xX]H+PFS?
3. 十六进制数: 0[xX]H*"."H+PFS?
4. 十六进制数: 0[xX]H+"."H*PFS
5. 八进制数: 0[0-7]*IS?
6. 十进制数: [1-9]D*IS?

7. 十进制数:　D+EFS?

8. 十进制数:　D*"."D+E?FS?

9. 十进制数:　D+"."D*E?FS?

由于数字常量的情况较多，我们在识别过程中进行了较为复杂的条件判断和状态转换，判断思路如下图所示。

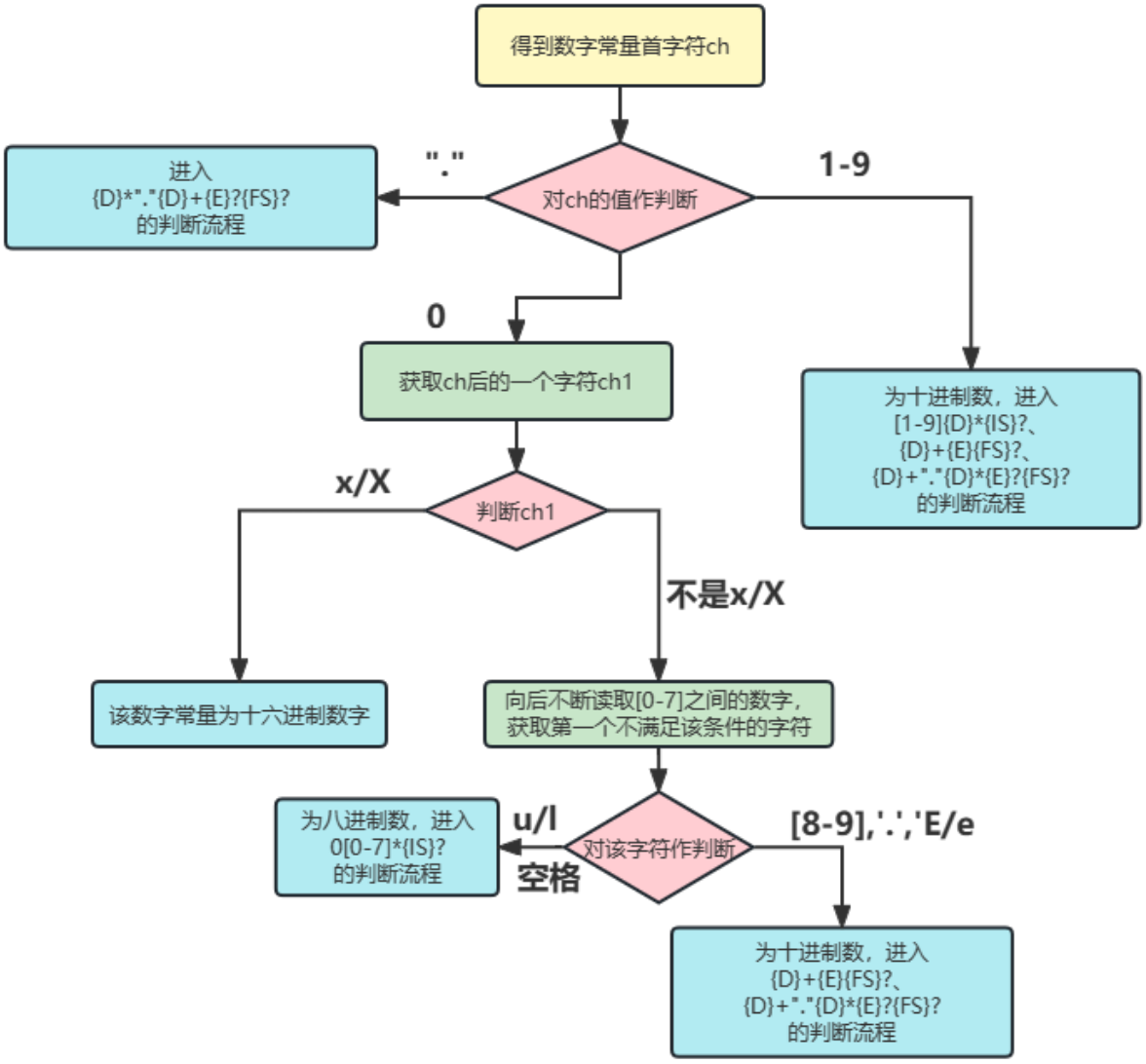


图 8: 数值识别条件判断和状态转换图

常量字符识别 const_char_DFA：当我们在缓冲区中读取到"'" 单引号的时候，我们进入常量字符的识别中，我们参考了 ANSI Yacc C 的文法，使用 C++11 标准的 *std* :: *regex* 的 *search* 方法进行提取和识别。这里需要注意的是当我们在出现一个单引号后读取到换行，则不将其视作一个常量字符，进行报错处理。

常量字符串识别 literal_string_DFA：当我们在缓冲区读取到""" 双引号时，我

们会进入常量字符串的识别中，在这个部分我们同样使用 C++ 语言 *std* :: *regex* 的 *search* 方法进行提取和识别，在这里不再赘述。

还有**部分工具函数**, 具体函数声明如下:

```
bool isLetter(char letter);
bool isDigit(char digit);
bool isDelimiter(char ch);
bool isOperator(char ch);
bool isOct(char ch);
bool isHex(char ch);
```

它们分别用于判断当前字符 (串) 是否为字母、十进制数字、界符、运算符、八进制数和十六进制数并将返回结果转为相应信号量传递给外部。

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

```
void Parser::read_grammer_Yacc(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:

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

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

```
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

```
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 集的过程中可能需要求解其他符号的 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。

```
class _item_   // point and prospect symbols are included
{
public:
        const generator base; //产生式
        unsigned int index; //圆点的位置
        std::string prospect_symbol; //展望
};
```

而一个项目集闭包的定义是这样的:

```
int id; //项目集编号
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。

- ```
  void construct_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。

- ```
  void Parser::get_sequence_first(const std::vector<std::string>& seq,
      std::vector<std::string>& re);
  //the seq is the input sequence
  //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。

- ```
  void state_group_go(const _item_group_& scr, _item_group_& dst, std::
      string input);
  //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

```
void get_state_group_list();
```

在这个函数中，我们首先利用项目 $start \rightarrow .translation\_unit,\#$ 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，我们就会遍历所有符号，并对于每个符号 input 调用 *state_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:

```
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我们首选的数据结构。

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

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

建立 ACTION-GOTO 表对应的函数为

```
void get_LR1_table();
```

在概要设计 一节中提到，我们为了加速 ACTION-GOTO 表的构建速度以优化用户体验，将预建立好的表存入了缓存 cache 中，因此在实现中我们同样实现了一个读取缓存的函数 *read_cache*, 函数声明如下。

```
int Parser::read_cache()
```

当无法读取缓存文件时，该函数会返回 0，此时我们才会重新开始构造 ACTION-GOTO 分析表。

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:

```
std::tuple<bool,std::string,int,int> parser::check(std::string path);
```

这里的函数参数就是词法分析的结果，返回值是一个包含分析结果，报错信息的元组。

3.3 语义分析

3.3.1 说明语句

说明语句由于不产生四元式，也就是不会有中间代码生成。说明语句主要的工作就是登记变量，将声明的变量登记到一张表里去，因此，对于声明语句的最主要的难点就在于对于符号表的设计。

这里设计两个结构体，一个结构体是 oneWord：

```
class oneWord
{
public:
        std::string type;
        std::string name;
        std::string value;

        int offset;
        oneWord(std::string type, std::string name, std::string value, int
            offset)
        {
                this->name = name;
                this->type = type;
                this->offset = offset;
                this->value = value;
        }
};
```

这个结构体存储的主要就是一个符号表中的一个项目的基本信息，如：变量名、变量类型、偏移量、以及变量的值。

然后再用另一个类将这个单个变量表项封装成一张表 sym_tbl：

```
class sym_tbl
{
        std::string tblName;
        std::string nextTblName;
        sym_tbl* next_table;
        std::vector<oneWord> syms;
public:
        sym_tbl(std::string tblName);
        void addsys(oneWord x);
        std::string newTemp(std::string type, std::string value);
```

```
11          oneWord lookup(std::string name);
12  };
```

这个符号表中加入了一个指向下一个符号表的指针：

```
1  sym_tbl* next_table;
```

这个指针的作用就是指向这个符号表所在的作用域的子作用域的符号表，方便做同名变量的作用域判断。

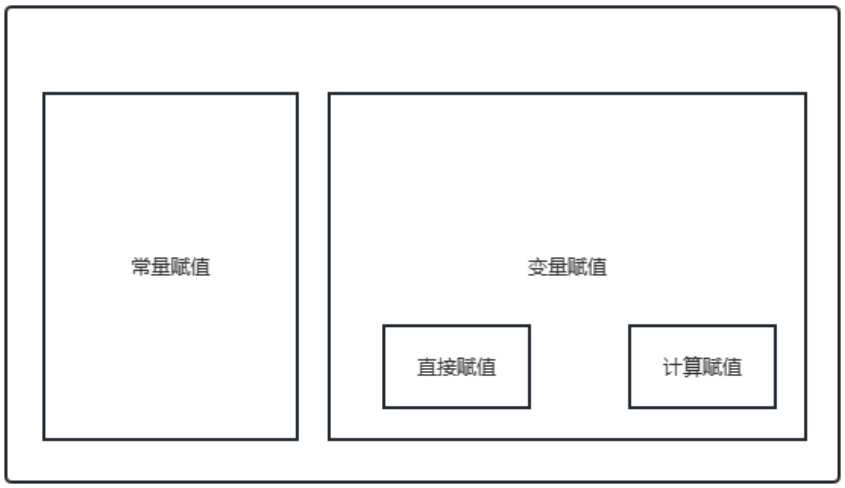
这里的符号表之后是用栈来存储，因为函数调用也是一个先调用先返回的一个模式，而相应的符号表也是嵌套使用的，因此整体的表结构还需要在 parser 中借助栈式存储管理。

3.3.2 赋值语句

赋值语句主要分成两部分，主要分为运算部分和赋值部分。

运算部分主要分为变量运算 + 变量常量运算

赋值部分的逻辑如下



下面分别介绍三类赋值

3.3.3 常量赋值

常量赋值分为三步：首先是查找等号右边的常量，确定其值

第二步是将等号左边的变量的属性根据右边来修改

第三步就可以直接写出四元式，用 emit 函数发射到文件中。

具体流程如下图所示。

3.3.4 变量直接赋值

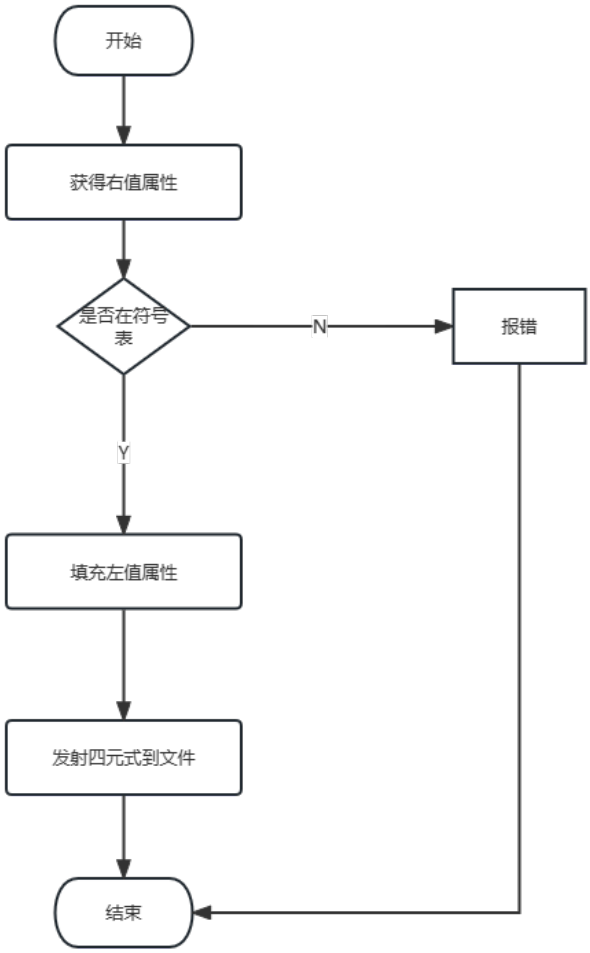
变量赋值直接赋值分为三步

第一步是到符号表中查找右边的变量是否登记在了符号表中，若查到则继续，查不到报错

第二步是将等号左边的变量的属性根据右边来修改

第三步就可以直接写出四元式，用 emit 函数输出到文件中

具体流程如下图所示。



3.3.5 变量运算赋值

变量赋值直接赋值分为两阶段

一阶段:

第一步是到符号表中查找右边的变量是否登记在了符号表中，若查到则继续，查不到报错

第二步是用右值做计算，发出计算的四元式，结果存在临时变量

第三步临时变量登记到符号表

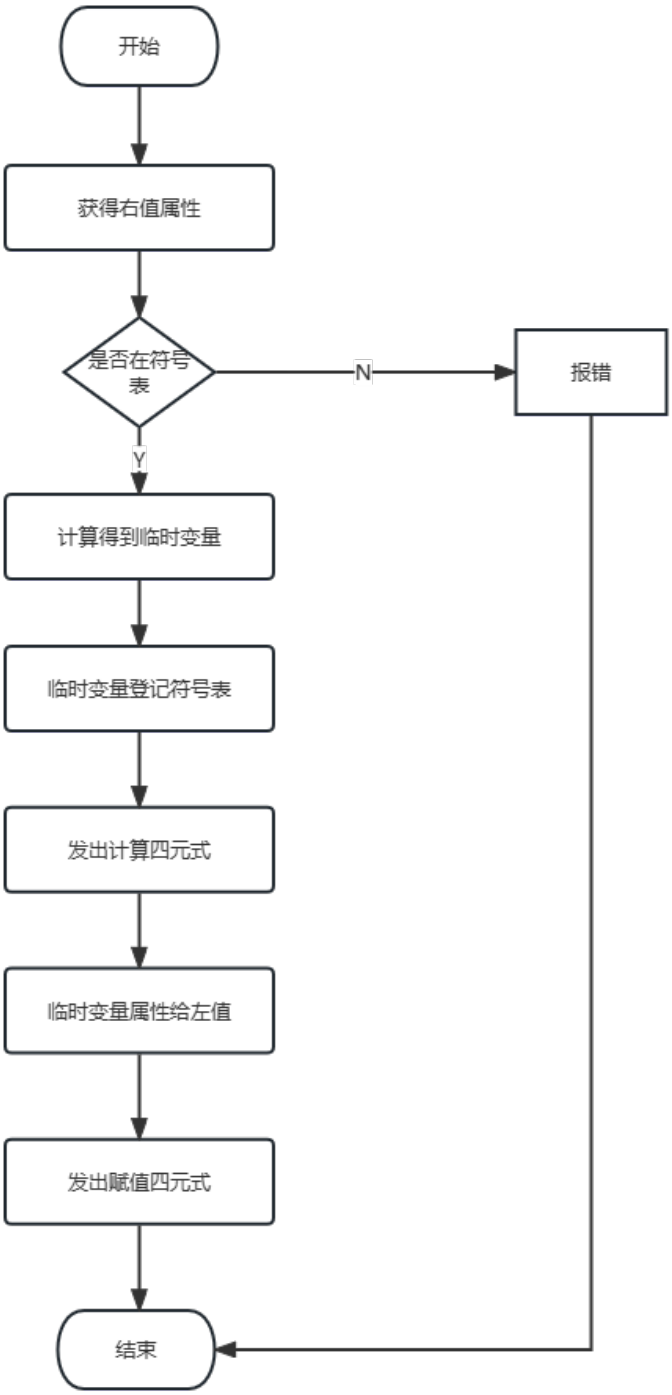
第四步发出计算的四元式

二阶段:

第一步是将等号左边的变量的属性根据临时变量修改

第二步就可以直接写出四元式，用 emit 函数发射到文件中

具体流程如下图所示



这里需要解释的是，这种变量运算赋值的表达式在本语法里面不是一句产生式产生的，而是由两句产生的：

分别是 (以加法为例)：

additive_expression –> additive_expression + multiplicative_expression

assignment_expression–>unary_expression assignment_operator assignment_expression

第一条产生式对应的是一阶段，第二条产生式对应的是二阶段。

综上，变量赋值语句主要就是在声明语句的基础上，查找符号表，判断运算是否存在先引用后定义的情况，同时发出四元式即可。

3.3.6 布尔表达式

语义分析过程中布尔表达式使用的函数

函数名	功能
OR_expression	完成 AND 语义对应语义动作
AND_expression	完成 OR 语义对应语义动作
NOT_expression	完成 NOT 语义对应语义动作
Identifier_expression	完成 Identifier 语义对应语义动作
Relop_expression	完成 Relop 语义对应语义动作

OR_expression logical_or_expression->logical_or_expression OR_OP logical_and_expression

- **具体函数描述与功能**

  主要用于完成 OR 语义对应语义动作。

  具体地，将需要地 logical_or_expression(E1) 与 logical_and_expression(E2) 分别从 wordS 的暂存栈中弹出，用于生成 logical_or_expression(E) 的属性，即将 E2.quad 回填 E1.falselist 最后一个参数 addr，并且将 E1.truelist 与 E2.truelist 合并到 E.truelist 中，并令 E.falselist 为 E2.falselist, 最后返回 E 的信息以及相关语义动作的报错信息 (可能为 None)。

- **采用的对应的数据结构部分**

OR_expression 的数据结构	功能
std::set<int> truelist	记录该 word 的真出口链
std::set<int> falselist	记录该 word 的假出口链
std::string quad	记录该 word 的 quad(四元式编号)

- **具体函数声明**

  std::tuple<word, std::string> OR_expression();

AND_expression logical_and_expression->logical_and_expression AND_OP inclusive_or_expression

- **具体函数描述与功能**

  主要用于完成 AND 语义对应语义动作。

  具体地，将需要地 logical_and_expression(E1) 与 inclusive_or_expression(E2) 分别从 wordS 的暂存栈中弹出，用于生成 logical_and_expression(E) 的属性，即将

E2.quad 回填 E1.truelist 最后一个参数 addr，并且将 E1.falselist 与 E2.falselist 合并到 E.falselist 中，并令 E.truelist 为 E2.truelist, 最后返回 E 的信息以及相关语义动作的报错信息 (可能为 None)。

- **采用的对应的数据结构部分**

AND_expression 的数据结构	功能
std::set<int> truelist	记录该 word 的真出口链
std::set<int> falselist	记录该 word 的假出口链
std::string quad	记录该 word 的 quad(四元式编号)

- **具体函数声明**

std::tuple<word, std::string> AND_expression();

NOT_expression unary_expression->unary_operator cast_expression

- **具体函数描述与功能**

主要用于完成 NOT 语义对应语义动作。

具体地，将需要地 cast_expression(E1) 从 wordS 的暂存栈中弹出，用于生成 unary_expression(E) 的属性，即将 E.truelist 等于 E1.falselist，并令 E.falselist 为 E1.truelist, 最后返回 E 的信息以及相关语义动作的报错信息 (可能为 None)。

- **采用的对应的数据结构部分**

NOT_expression 的数据结构	功能
std::set<int> truelist	记录该 word 的真出口链
std::set<int> falselist	记录该 word 的假出口链

- **具体函数声明**

std::tuple<word, std::string> NOT_expression();

Identifier_expression primary_expression->(expression)

- **具体函数描述与功能**

主要用于完成 Identifier 语义对应语义动作。

具体地，将需要地 expression(id) 从 wordS 的暂存栈中弹出，用于生成 primary_expression(E) 的属性，即分别用 nextquad 以及 nextquad+1 生成 E.truelist 和 E.falselist, 然后生成两条跳转的四元式，最后返回 E 的信息以及相关语义动作的报错信息 (可能为 None)。

- **采用的对应的数据结构部分**

Identifier_expression 的数据结构	功能
std::string nextquad	记录全局的下一个四元式编号
std::set<int> truelist	记录该 word 的真出口链
std::set<int> falselist	记录该 word 的假出口链

- **具体函数声明**

  std::tuple<word, std::string> Identifier_expression();

Relop_expression relational_expression->relational_expression < shift_expression

relational_expression ->relational_expression > shift_expression

relational_expression ->relational_expression LE_OP shift_expression

relational_expression-> relational_expression GE_OP shift_expression

equality_expression->equality_expression EQ_OP relational_expression

equality_expression->equality_expression NE_OP relational_expression

- **具体函数描述与功能**

  主要用于完成 Relop 语义对应语义动作。

  具体地，将需要地 id1,relop,id2 分别从 wordS 的暂存栈中弹出，用于生成 E 的属性，即即分别用 nextquad 以及 nextquad+1 生成 E.truelist 和 E.falselist, 然后生成两条跳转的四元式，最后返回 E 的信息以及相关语义动作的报错信息 (可能为 None)。

- **采用的对应的数据结构部分**

Relop_expression 的数据结构	功能
std::string nextquad	记录全局的下一个四元式编号
std::set<int> truelist	记录该 word 的真出口链
std::set<int> falselist	记录该 word 的假出口链

- **具体函数声明**

  std::tuple<word, std::string> Relop_expression();

3.3.7 控制语句

语义分析过程中控制语句使用的函数

函数名	功能
if_	完成 if 控制语句对应语义动作
if_else	完成 if_else 控制语句对应语义动作
while_	完成 while 控制语句对应语义动作
end_sentence	完成; 控制语句对应语义动作
end_sentence_2	完成; 控制语句对应语义动作

if_ selection_statement-> IF (expression) statement

- **具体函数描述与功能**

  主要用于完成 if 控制语句对应语义动作。

  具体地，将需要地 expression(E) 与 statement(S1) 分别从 wordS 的暂存栈中弹出，用于生成 selection_statement(S) 的属性，即将 S1.quad 回填 E.truelist 最后一个参数 addr，并且将 E.falselist 与 S1.nextlist 合并到 S.nextlist 中, 最后返回 S 的信息以及相关语义动作的报错信息 (可能为 None)。

- **具体函数声明**

  std::tuple<word, std::string> if_();

if_else selection_statement-> IF (expression) statement ELSE statement

- **具体函数描述与功能**

  主要用于完成 if_else 控制语句对应语义动作。

  具体地,将需要地 expression(E),else,statement(S1) 与 statement(S2) 分别从 wordS 的暂存栈中弹出，用于生成 selection_statement(S) 的属性，即将 S1.quad 回填 E.truelist 最后一个参数 addr，S2.quad 回填 E.falselist 最后一个参数 addr，并且将 S1.nextlist,S2.nextlist 与 N(else).nextlist 合并到 S.nextlist 中, 最后返回 S 的信息以及相关语义动作的报错信息 (可能为 None)。

- **具体函数声明**

  std::tuple<word, std::string> if_else();

while_ iteration_statement-> WHILE (expression) statement

- **具体函数描述与功能**

  主要用于完成 while 控制语句对应语义动作。

  具体地，将需要地 expression(E) 与 statement(S1) 分别从 wordS 的暂存栈中弹出，用于生成 iteration_statement(S) 的属性，即将 E.quad 回填 S1.nextlist 最后

一个参数 addr，将 S1.quad 回填 E.truelist 最后一个参数 addr，并且令 S.nextlist 等于 E.falselist, 并生成一条跳转四元式，最后返回 S 的信息以及相关语义动作的报错信息 (可能为 None)。

- **具体函数声明**

  std::tuple<word, std::string> while_();

end_sentence expression_statement -> expression ;

- **具体函数描述与功能**

  主要用于完成; 控制语句 1 对应语义动作。

  具体地，将需要地 expression(L1) 从 wordS 的暂存栈中弹出，用于生成 expression_statement(L) 的属性，即将 nextquad 回填 L1.nextlist 最后一个参数 addr，并且令 L.nextlist 等于 L1.nextlist, 最后返回 L 的信息以及相关语义动作的报错信息 (可能为 None)。

- **具体函数声明**

  std::tuple<word, std::string> end_sentence();

end_sentence_2 expression_statement ->;

- **具体函数描述与功能**

  主要用于完成; 控制语句 2 对应语义动作。

  具体地，生成 expression_statement(L) 的属性，即用 nextquad 生成 L.quad 并用 nextquad 生成 L.nextlist, 最后返回 L 的信息以及相关语义动作的报错信息 (可能为 None)。

- **具体函数声明**

  std::tuple<word, std::string> end_sentence_2();

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 在这个部分，我们将对 3 组数据进行测试，测试内容涵盖注释语句、声明语句、赋值语句、while 循环语句以及 if、if-else 控制语句。

第一组测试代码：

```
nt main(){
    double a=0;
    while(1){
    if(a<5)
        ;
    else{
        if(a==2)
            a+=2;
        else{
            a--;
            int b=2;
            int c=b;
            if(c<=a)
                ;
        }
     }
    a++;
   }
   return 0;
}
```

语法分析和语义分析通过，生成的四元式如下，经检查无误。

```
0 ( =, 0, -, a )
1 ( j<, relational_expression , shift_expression , 3 )
2 ( j , None, None, 13 )
3 ( j , None, None, INIT_STATE )
4 ( j==, equality_expression , relational_expression , 6 )
5 ( j , None, None, 13 )
6 ( +=, 2, -, a )
7 ( j , None, None, INIT_STATE )
8 ( -, a, 1, T1 )
9 ( :=, T1, None, a )
```

```
10 ( =, 2, -, b )
11 ( =, b, -, c )
12 ( j<=, relational_expression, shift_expression, 14 )
13 ( j, None, None, INIT_STATE )
14 ( +, a, 1, T2 )
15 ( :=, T2, None, a )
16 ( j, None, None, 0 )
```

第二组测试代码如下:

```
int fun()
{
    int i=0;
    while(i<5)
        i++;

    int c=i+1.0;
    unsigned out;
    if(c<0x3f)
       out=c;
    else if(c<0x13f)
        out=1+c;
    else
        out=2+c;

    while(out>0)
        out--;
    return out;
```

词法、语法和语义检查通过，生成的四元式如下，经检查无误。

```
0 ( =, 0, -, i )
1 ( j<, relational_expression, shift_expression, 4 )
2 ( j, None, None, INIT_STATE )
3 ( +, i, 1, T1 )
4 ( :=, T1, None, i )
5 ( j, None, None, 1 )
6 ( +, i, 1.0, T2 )
7 ( =, T2, -, c )
8 ( j<, relational_expression, shift_expression, 10 )
9 ( j, None, None, 19 )
10 ( =, c, -, out )
11 ( j, None, None, INIT_STATE )
12 ( j<, relational_expression, shift_expression, 15 )
13 ( j, None, None, 18 )
14 ( +, 1, c, T3 )
15 ( =, T3, -, out )
16 ( j, None, None, INIT_STATE )
17 ( +, 2, c, T4 )
18 ( =, T4, -, out )
```

```
19 ( j>, relational_expression , shift_expression , 22 )
20 ( j , None , None , INIT_STATE )
21 ( -, out , 1 , T5 )
22 ( :=, T5 , None , out )
23 ( j , None , None , 19 )
```

第三组测试代码如下:

```
int main()
{
    int a=0;
    int i=0;
    double d=.3f;
    unsigned c=a+2;
    if(c<0)
    {
        while(i<0xa)
        {
            c--;
            i--;
        }
    }
    else
    {
        i=0;
        if(c<10)
            while(i<0xca)
                i++;
        else
            i=2;
    }
}
```

词法、语法和语义检查通过，生成的四元式如下，经检查无误。

```
0 ( =, 0, -, a )
1 ( =, 0, -, i )
2 ( =, .3f, -, d )
3 ( +, a, 2, T1 )
4 ( =, T1, -, c )
5 ( j<, relational_expression , shift_expression , 13 )
6 ( j , None , None , 24 )
7 ( j<, relational_expression , shift_expression , 12 )
8 ( j , None , None , INIT_STATE )
9 ( -, c , 1 , T2 )
10 ( :=, T2, None , c )
11 ( -, i , 1 , T3 )
12 ( :=, T3, None , i )
13 ( j , None , None , 7 )
14 ( j , None , None , INIT_STATE )
```

```
15 ( =, 0, -, i )
16 ( j<, relational_expression , shift_expression , 22 )
17 ( j , None , None , 24 )
18 ( j<, relational_expression , shift_expression , 21 )
19 ( j , None , None , INIT_STATE )
20 ( +, i , 1 , T4 )
21 ( :=, T4 , None , i )
22 ( j , None , None , 18 )
23 ( j , None , None , INIT_STATE )
24 ( =, 2 , -, i )
```

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、语法分析器以及中间代码检查器的报错能力：

首先是未闭合的注释：

```
int main()
{
    /****
    comment
    int a;
    return 0;
}
```

测试结果如下，显示因注释未闭合导致词法分析失败。

```
The comment is not closed. (8,2)
```

接着测试数字常量的错误定义

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

测试结果如下，词法分析失败。

```
LEXICAL ERROR::Text beyond expectations after
decimals
 (5,14)
```

接着我们测试常量字符定义出错的情况

```
char ch='\r ;
int a;
```

```
LEXICAL ERROR::the const character can't be
identidied
 (3, 12)
```

测试结果如上图，词法分析失败。

这是常量字符串出错的情况，同样会在词法分析部分出错

```
char ch[7]="compilation test !\' ' ;
int a;
```

测试结果如下，词法分析失败。

```
LEXICAL ERROR::the literal string is not closed
 (3, 15)
```

接着测试一些我们写代码时经常会出现的语法错误:

首先是缺少";"，会在语法分析出错

```
int main()
{
  ...
 return 0 //这里缺少分号
}
```

测试结果如下，语法分析失败。

```
SYNTAX ERROR::syntax error
 (12, 15)
```

接下来是括号不匹配的问题，同样会在语法分析出错

```
int fun()
{
    ...
    return 0;
//这里缺少大括号
```

测试结果如下，语法分析失败。

```
SYNTAX ERROR::syntax error
 (15,1)
```

我们还对一些常见的语法正确但语义错误的情况进行了处理，比如引用未定义变量的问题，这里给出两个例子。

```
int main(){
    double a=0;
    while(1){
    else{
        if(a==2)
            a+=2;
        else{
            a--;
        //用未出现变量定值
            int c=b;
            if(c<=a)
                ;
        }
     }
    a++;
   }
   return 0;
}
```

程序会显示出如下的报错信息:

```
SEMANTIC ERROR::use undeclared variable
 (11,12)
```

当我们用一个表达式为变量定值时，若该表达式中出现未声明变量时，程序同样甄别出该语义错误，比如下面这段代码。

```
int main(){
    double a=0;
    while(1){
    if(a<5)
        ;
    else{
        if(a==2)
```

```
            a+=2;
        else{
            a--;
        //用含 b 的表达式给 c 赋值，但 b 未定义
            int c=b+2;
            if(c<=a)
                ;
        }
      }
     a++;
    }
    return 0;
}
```

程序同样会出现如下的报错信息。

```
SEMANTIC ERROR::use undeclared variable
 (11,14)
```

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

数字的识别采用纯 DFA 的方式进行识别，其中只涉及到条件判断，因此所以平均时间复杂度为 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)。

常量字符和字符串的 DFA

这个部分的 DFA 使用 *std* :: *regex* 构建，只识别字符串开头处的子串，经过查询资料和一定数目的试验，可以得出平均为 $O(n)$ 的时间复杂度，其中 n 是符合条件的子串长度。

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 ε 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 ε，那就把 ε 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 β，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).

接着是**语义分析的复杂性**：语义分析主要部分在于调用的若干语义动作的函数，主要有以下几个部分:

1. 说明语句
2. 赋值语句
3. 布尔表达式
4. 控制语句

对于赋值语句的复杂性分析，首先赋值语句分为两种，一种是 a=1 直接赋值常量，另一种是 a=b+1, 对于第二种也有可能是 a=b+c 或者 a=b，

对于第一种，直接赋值即可，传递一下属性值，复杂度为 O(1)；

对于上述后三种情况都需要查表，而查表首先需要从表栈中将表弹出，然后遍历一遍栈顶的符号表，时间复杂度是 O(n)；

对于布尔表达式的语义动作，主要过程就是弹出栈顶的 word 元素，然后进行 makelist，回填等一系列操作，最终返回生成的新的 word 以及相关信息，其中主要有 makelist，pop 以及 backpatch。

- makelist: 由于初始生成 list 的时候都是用一个元素，所以复杂度为 O(1)。
- pop: 由于采用的是 STL 的 stack 结构，所以每次 pop 得到需要的 word 元素的时间复杂度为 O(1)。
- backpatch: 将 addr 地址回填至 set<int>list 中的各个四元式的最后一个元素，由于需要遍历 set 数组，所以时间复杂度为 O(n), 其中 n 为 set 中元素的个数。

对于控制语句的语义动作，主要使用的函数与时间复杂度与布尔表达式的类似，就不再赘述。

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。

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

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。

3. 当输入不符合词法规则的源程序文件后，程序会陷入死循环。

   解决方法: 经过检查发现，当调用 DFA 产生错误信息后，词法分析环节中读取字符的循环体没有及时中断，在补充了对这种情况的判断后解决了这个问题。

4. 词法分析过程中无法识别诸如 $.2e3f$ 这样以小数点开头的数字常量。

   解决方法: 经过检查发现，当词法分析器读取到'.' 符号后，会将其视作是运算符'.' 或者'...'(ELLIPSIS) 的前缀从而进入到识别运算符的 DFA 中，不会将其视作数字进行识别。因此我们在读取到'.' 后，会首先展望其后面的一个字符，若为数字字符 [0-9] 将将其视作数字常量进行识别，若不是才将其视作运算符，从而解决了这个问题，进行特殊判断的代码如下。

```
else if (ch == '.')      //'.' and ellipse and '.3'
{
    if (start + 1 < out_str.size() && isDigit(out_str[start + 1]))
        enter_dfa = DIGIT_DFA;
    else
        enter_dfa = OPERATOR_DFA;
}
```

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 falsse，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 ε 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。

4.2.3 语义分析

在完成语义分析的过程中主要遇到了以下一些主要问题:

1. 采用函数指针数组时，统一初始化后发现程序报错。

   解决方法: 查找后发现是因为函数指针指向的是类的函数，但由于无法明确指向的是哪个对象的函数因而报错，最后，将这部分函数改成全局函数，并将原来存储的数组形式由 std::tuple<word, std::string>(Parser::*p[229])() 调整为 std::tuple<word, std::string>(*p[229])()，最终解决问题。

2. 一开始在某些回填的时候发现一部分四元式的 addr 部分为空字符串，即回填内容为空。

   解决方法: 在处理语句；即 end_sentence 的时候未将单独的 expression_statement ->; 配上相应的语义动作，使得相关属性可以向上传递，从而解决问题。

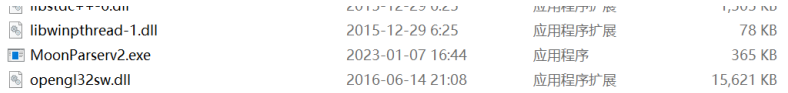
3. 一开始没有区分 a=b, 和 a=b+1 两种类型的赋值语句，导致出错。

   解决方法: 打印出语法树，找到对应的产生式，然后在用类似的方法对产生式实现语义动作，从而解决问题。

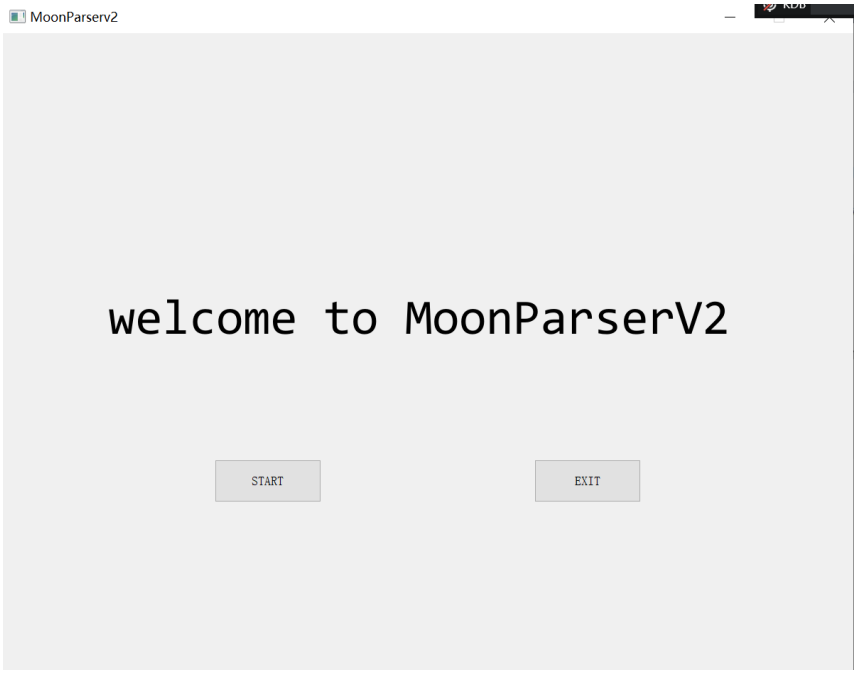
5 程序使用说明

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。

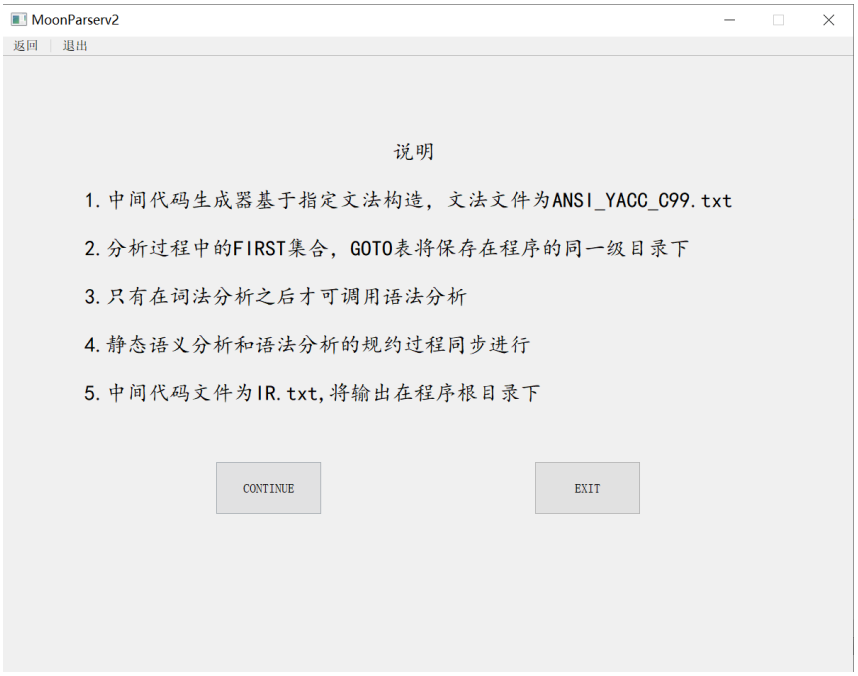
1. 首先进入 *holistic_program* 文件夹，其中的 *MoonParserv*2.*exe* 即为可执行文件。

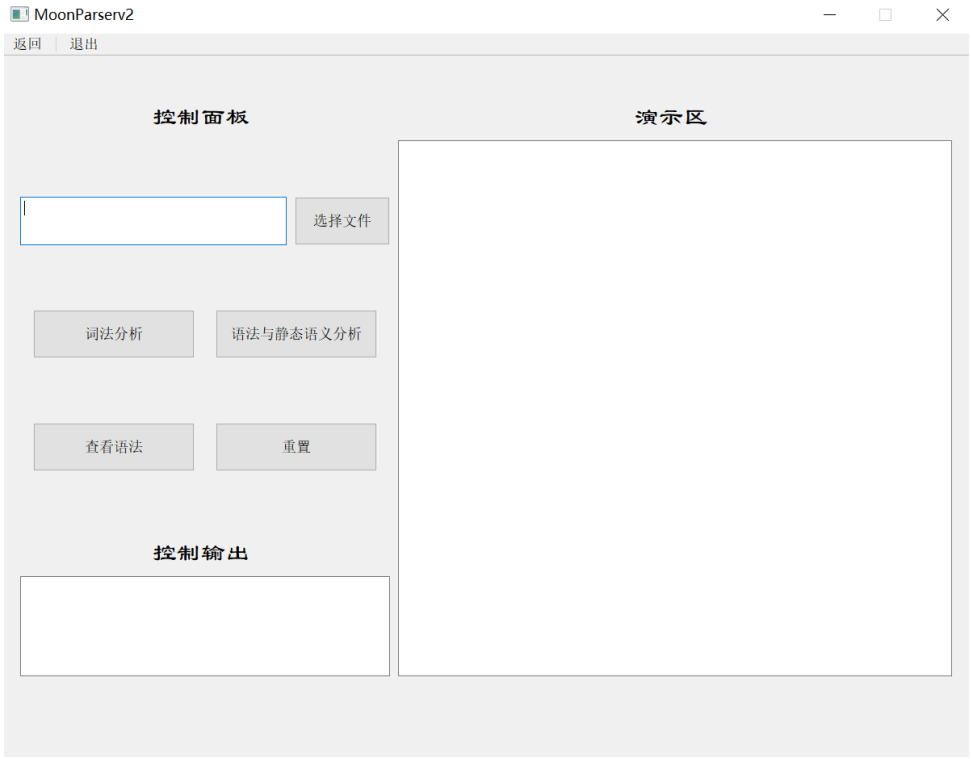


2. 点击进入，即可进入程序欢迎界面。点击左下方的 *START* 按钮即可进入规则说明页面。

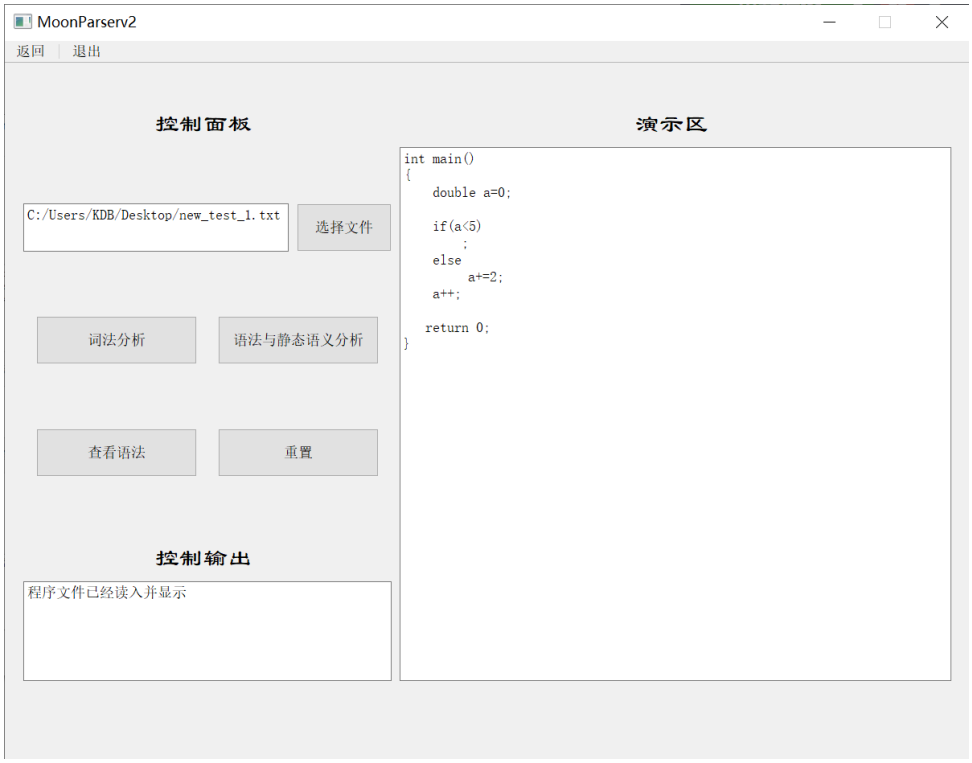


3. 查看规则后点击 *CONTINUE* 页面即可进入功能演示界面。

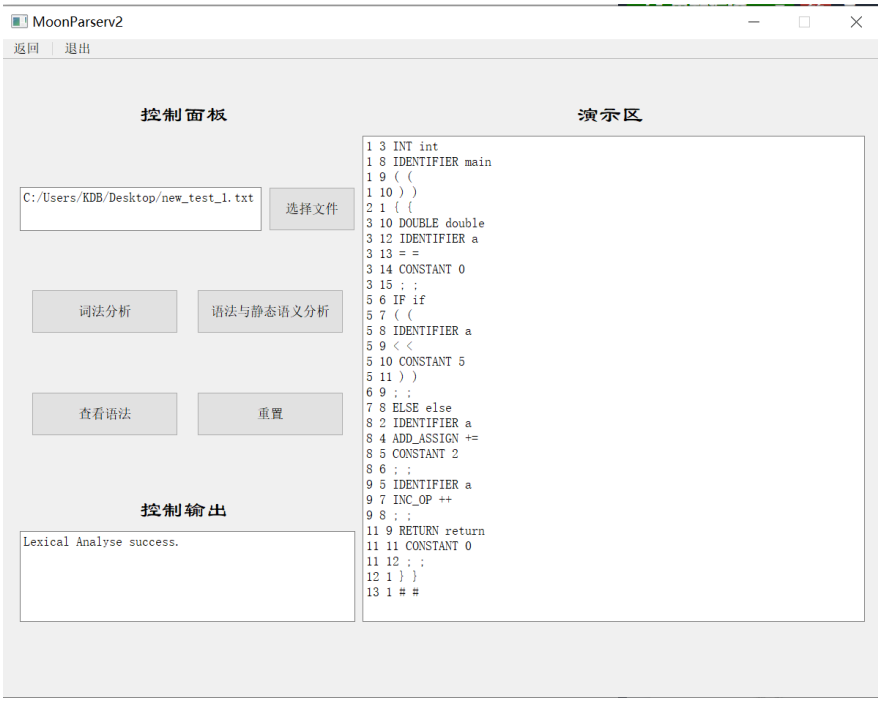




4. 在功能演示界面中点击“选择文件”按钮，会弹出文件对话框，确认之后，文件中的内容将会展示在界面右侧。主要注意的是，文件的路径中不能含有中文字符。



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, 如果源程序代码无法通过词法分析，则会在同样的位置显示具体的报错信息和出错位置（具体到行、列）。

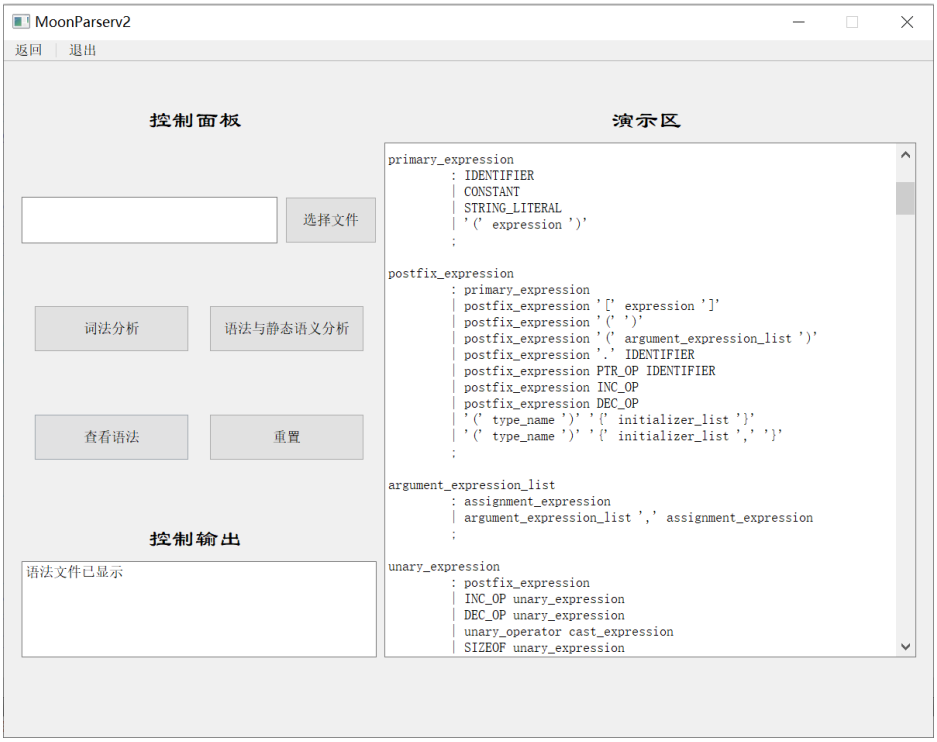


6. 在完成词法分析后，用户可以点击“语法分析”进行语法分析，若 token 串可以成功通过语法分析，则会在左下方的控制输出中打印" syntax analysis success. the IR file is output as *IR.txt*", 并于右侧面板中动态展示移进规约的全过程。

生成的 *IR.txt* 的四元式形式如下所示:

```
0 ( =, 0, -, a )
1 ( j<, relational_expression , shift_expression , 3 )
2 ( j , None , None , 4 )
3 ( j , None , None , INIT_STATE )
4 ( +=, 2, -, a )
5 ( +, a, 1, T1 )
6 ( :=, T1, None , a )
```

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。

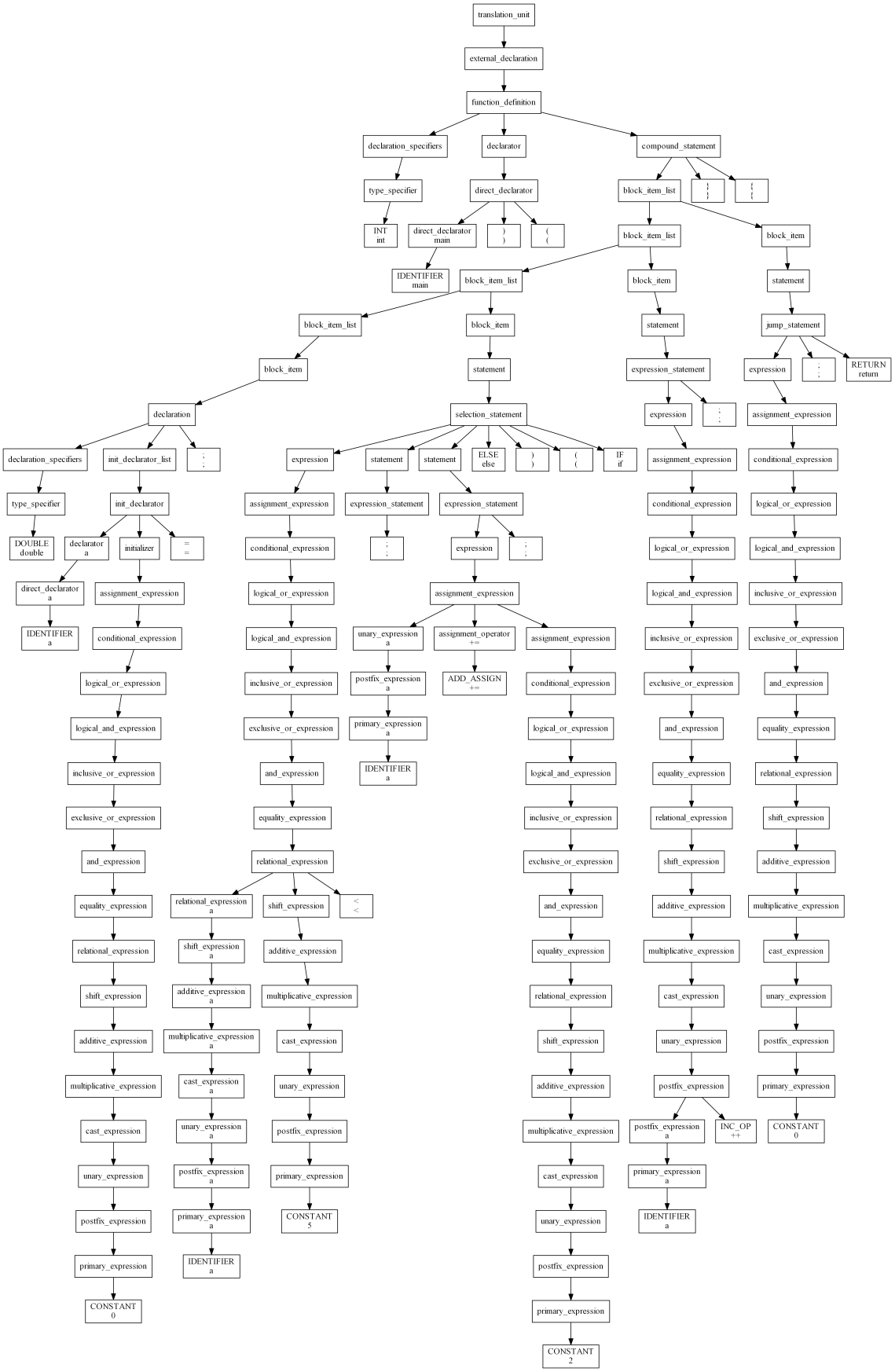


在成功完成一次语法分析和语义分析后，可执行文件同级别目录下会出现 *IR.txt* 和 *grammer_tree.dot* 两个文件,其中 *IR.txt* 是生成的中间代码文件,*grammer_tree.dot* 中存储的是语法树的信息，将该 dot 文件输入到 graphviz 中即可生成对应的图片。具体方法如下。

- 在命令行中将当前目录切换为 *Graphviz* 下。(如果已经将 *Graphviz* 文件夹目录加入到环境变量中则无需此步骤)
- 通过输入

$$dot\ -T png\ path(grammer\_tree.dot)\ -o\ path(grammer\_tree.png)$$

命令，即可将路径为 *path*(*grammer_tree.dot* 的 dot 文件转换为 png 文件输出到 *path*(*grammer_tree.png*) 目录下, 画出的语法树 grammer_tree.png 如下图。


translation_unit
external_declaration
function_definition
declaration_specifiers
type_specifier
INT
int
declarator
direct_declarator
direct_declarator
main
IDENTIFIER
main
(
(
)
)
compound_statement
{
{
block_item_list
block_item_list
block_item
statement
jump_statement
RETURN
return
expression
assignment_expression
conditional_expression
logical_or_expression
logical_and_expression
inclusive_or_expression
exclusive_or_expression
and_expression
equality_expression
relational_expression
shift_expression
additive_expression
multiplicative_expression
cast_expression
unary_expression
postfix_expression
primary_expression
CONSTANT
0
;
;
}
}
block_item_list
block_item
statement
expression_statement
expression
assignment_expression
conditional_expression
logical_or_expression
logical_and_expression
inclusive_or_expression
exclusive_or_expression
and_expression
equality_expression
relational_expression
shift_expression
additive_expression
multiplicative_expression
cast_expression
unary_expression
postfix_expression
postfix_expression
INC_OP
++
primary_expression
IDENTIFIER
a
;
;
block_item_list
block_item
declaration
declaration_specifiers
type_specifier
DOUBLE
double
init_declarator_list
init_declarator
declarator
a
direct_declarator
a
IDENTIFIER
a
initializer
=
=
assignment_expression
conditional_expression
logical_or_expression
logical_and_expression
inclusive_or_expression
exclusive_or_expression
and_expression
equality_expression
relational_expression
relational_expression
a
shift_expression
a
additive_expression
a
multiplicative_expression
a
cast_expression
a
unary_expression
a
postfix_expression
a
primary_expression
a
IDENTIFIER
a
shift_expression
<
<
additive_expression
multiplicative_expression
cast_expression
unary_expression
postfix_expression
primary_expression
CONSTANT
5
;
;
block_item
statement
selection_statement
expression
assignment_expression
conditional_expression
logical_or_expression
logical_and_expression
inclusive_or_expression
exclusive_or_expression
and_expression
equality_expression
relational_expression
shift_expression
additive_expression
multiplicative_expression
cast_expression
unary_expression
postfix_expression
primary_expression
CONSTANT
2
statement
statement
ELSE
else
)
)
(
(
IF
if
expression_statement
expression_statement
;
;
expression
;
;
unary_expression
a
assignment_operator
+=
ADD_ASSIGN
+=
assignment_expression
postfix_expression
a
primary_expression
a
IDENTIFIER
a
conditional_expression
logical_or_expression
logical_and_expression
inclusive_or_expression
exclusive_or_expression
and_expression
equality_expression
relational_expression
shift_expression
additive_expression
multiplicative_expression
cast_expression
unary_expression
postfix_expression
primary_expression
CONSTANT
0

由于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/MoonParserv2/images

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、语法分析和语义分析的相关知识，接下来，我们使用了 ANSI Yacc C99 的文法，并以此为基础完成相关的分析过程，以四元式作为中间代码的形式输出。

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，可以完整地完成词法分析、语法分析的过程，并对赋值、说明语句、布尔表达式以及控制语句等生成四元式中间代码，并输出 *.dot* 文件实现语法分析树的绘制。
- 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便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、语法树等信息，并可以在根目录下获取中间代码文件。当源程序出现错误时，程序会将错误定位到行、列以方便用户进行修改。
- 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/MoonParserv2

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了变得清晰明了。

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。由于在合作开发的过程中需要将不同同学开发的模块进行合并，我们对于模块测试的重要性也有了更多认识，有时一个模块中的一个小细节出错就会使整个程序崩溃。

在中间代码生成的过程中，我们尝试使用一些 C++ 语言的新特性，诸如结构化绑定，初始化列表等等，拓展了我们的技术能力。

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、严谨和复杂，希望可以在未来的学习中能够对编译原理和编译技术有更深刻地了解和认识，使计算机科学家和工程师们的智慧得到发扬。

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.

[2] Jutta Degener. ANSI C Yacc grammar [EB/OL]. (2012) [2023-1-5].https://www.quut.com/c/ANSI-C-grammar-y-1999.html

[3] Jutta Degener. ANSI C grammar, Lex specification [EB/OL]. (2012) [2023-1-5].https://www.quut.com/c/ANSI-C-grammar-l-1999.html

[4] 李磊. C 编译器中间代码生成及其后端的设计与实现. Diss. 电子科技大学, 2016.

[5] 孙梓森, and 原庆能. "C plus 编译器设计." 广西工学院学报 (2005).

[6] 陈辉, and 郭艳玲. " 用 LEX 构造数控编译器词法分析程序的研究." 机电工程技术 35.2(2006):3.