

编译原理课程设计

学 院（系）： 计算机科学与技术学院
学 生 姓 名： 陈豪宇
学 号： 201961203
班 级： 电计 1905
联 系 方 式： 18895336428
负责工作及工作量： 100%
同 组 人： 无

大连理工大学

Dalian University of Technology

目 录

1	源语言定义.....	1
1.1	样本语言文法定义.....	1
1.2	关键字记录数组.....	2
1.3	结构体与容器.....	3
1.4	中间代码定义.....	3
1.5	单词的识别模型-有穷自动机 DFA.....	4
2	词法分析程序的实现.....	5
2.1	属性字定义.....	5
2.1.1	标识符.....	5
2.1.2	整型数字.....	5
2.1.3	浮点型数字.....	5
2.1.4	保留字.....	5
2.1.5	分界符.....	5
2.1.6	运算符.....	5
2.2	单词判断类型.....	6
2.3	单词读入程序设计.....	6
2.4	词法分析流程图.....	7
2.5	词法分析结果输出.....	7
3	LL(1)实现语法分析程序.....	9
3.1	读入文法.....	9
3.2	Get first 集.....	10
3.3	Get follow 集.....	10
3.4	Get select 集.....	11
3.5	输出预测表.....	11
3.5	语法分析过程.....	12
4	语义分析程序以及四元式生成.....	13
4.1	优先级定义.....	13
4.2	逻辑值以及数值计算.....	13
4.3	主函数部分生成四元式.....	13
4.3.1	声明语句.....	14
4.3.2	cin 语句.....	14

4.3.3 cout 语句	14
4.3.4 赋值语句	14
4.3.5 if 语句	14
4.3.5 while 语句	15
4.3.5 for 语句	15
4.4 四元式正反例结果分析	16
4.4.1 正例程序以及四元式	16
4.4.2 反例程序以及四元式	17
5 汇编语言生成	19
感 想	20
附录 语法分析阶段结果展示	21
1 该 C 子集文法的 first 集结果	21
2 该 C 子集文法的 follow 集结果	22
3 该 C 子集文法的 select 集结果	23
4 该 C 子集文法的预测分析表结果	25

1 源语言定义

该语言采用的是 C++ 语言子集的语法，本程序所识别的关键字或保留字共有 41 个，但是在翻译阶段仅仅翻译了几个常用关键词如：int、while、for、if、else、double、cin、cout、return 等，识别的操作符或运算符有 22 个，实现翻译的有常用的数值计算符号、逻辑表达式计算、赋值语句，比如：“+、-、*、/、<、>、%、=”等，识别的分界符有 8 个，；（）[]{}，实现翻译的有：，；（）{ }。

1.1 样本语言文法定义

```

<函数定义> -> <类型> <变量> ( <参数声明> ) { <函数块> }
<类型> -> type
<变量> -> <标志符> <数组下标>
<标志符> -> id
<数组下标> -> [ <因式> ] | @
<因式> -> ( <表达式> ) | <变量> | <数字>
<数字> -> number
<表达式> -> <因子> <项>
<因子> -> <因式> <因式递归>
<因式递归> -> * <因式> <因式递归> | / <因式> <因式递归> | % <因式> <因式递归> | @
<项> -> + <因子> <项> | - <因子> <项> | @
<参数声明> -> <声明> <声明闭包> | @
<声明> -> <类型> <变量> <赋初值>
<赋初值> -> = <右值> | @
<右值> -> <表达式> | { <多个数据> }
<多个数据> -> <数字> <数字闭包>
<数字闭包> -> , <数字> <数字闭包> | @
<声明闭包> -> , <声明> <声明闭包> | @
<函数块> -> <声明语句闭包> <函数块闭包>
<声明语句闭包> -> <声明语句> <声明语句闭包> | @
<声明语句> -> <声明> ;
<函数块闭包> -> <赋值函数> <函数块闭包> | <for 循环> <函数块闭包> | <条件语句> <函数块闭包> | <函数返回> <函数块闭包> | <while 循环> <函数块闭包> | @

```

图 1.1 使用文法定义 (1)

```

<赋值函数> -> <变量> <赋值或函数调用> | cin <cin 闭包> ; | cout <cout 闭包> ;
<cin 闭包> -> >> <表达式> <cin 闭包> | @
<cout 闭包> -> << <表达式> <cout 闭包> | @
<赋值或函数调用> -> = <右值> ; | ( <参数列表> ) ;
<参数列表> -> <参数> <参数闭包>
<参数闭包> -> , <参数> <参数闭包> | @
<参数> -> <标志符> | <数字>
<for 循环> -> for ( <赋值函数> <逻辑表达式> ; <后缀表达式> ) { <函数块> }
<while 循环> -> while ( <逻辑表达式> ) { <函数块> }
<逻辑表达式> -> <表达式> <逻辑运算符> <表达式>
<逻辑运算符> -> <|> | == | != | >= | <=
<后缀表达式> -> <变量> <后缀运算符>
<后缀运算符> -> ++ | --
<条件语句> -> if ( <逻辑表达式> ) { <函数块> } <否则语句>
<否则语句> -> else { <函数块> } | @
<函数返回> -> return <因式> ;

```

图 1.2 使用文法定义 (2)

其中@为空，在 for 循环语句中，由于赋值函数本身后面带有一个分号，因此没有在<赋值函数> <逻辑表达式>中间增加分号的设置，如果增加该分号的话，导致源语言定义的 for 循环的括号中的赋值语句后需要识别两个分号，否则在语法分析阶段不通过，显示为非法的 LL (1) 文法。

1.2 关键字记录数组

在 struct.h 文件中初始化时就定义了全局的数组变量：

```

string key_word[n_key] = {"auto", "enum", "restrict", "unsigned", "break", "extern", "return",
"void", "case", "float", "short", "volatile", "char", "for", "signed", "while", "const", "goto",
"sizeof", "_Bool", "continue", "if", "static", "_Complex", "default", "inline", "struct",
"_Imaginary", "do", "int", "switch", "double", "long", "typedef", "else", "register", "union",
"scanf", "printf", "cin", "cout"};
string oper[] = {"+", "-", "*", "/", "^", "<", ">", "++", "--", "==", "!=", "/", ">=", "<=",
"<<", ">>", ">>=", "<<=", "%", "&", "^", "="};
char bound[] = {';', ':', '(', ')', '[', ']', '{', '}'};

```

在词法分析的程序中，通过插叙全局的数组来确认是否标识符或是保留字，以及帮助程序判断是否是操作符、分界符。

1.3 结构体与容器

定义全局的结构体函数，如下图所示：

词结构体	int类型的id标识
	string类型的value值
	int类型的line所在行记录
数字结构体	int类型的id标识
	int类型的vi记录int的值
	double类型的vd记录double的值
	int类型的line所在行记录
单词结构体	string类型的 value值
	string类型的type类型标识
	int类型的number数字数值
	int类型的line所在行记录
四元式结构体	string op记录操作符
	string r1, r2两个操作数
	string left被赋值

图 1.3 相关结构体说明

预测表的容器 `map< pair<string, string>, string> pre_list ;`

first, follow, wenfa, select 的容器 `map<string, sets> ;`

Term, Nterm 的容器 `sets <string>; //终极符与非终极符`

now, last 的容器：词法结构体；//语义阶段的读取四元式的序号

变量名的表的容器：`vector<string> var;`

四元式表，汇编指令表的容器：`vector<Four_exp> four_exp, assemble;`

优先级数组的容器：`map<string, int> pri;`

1.4 中间代码定义

中间代码以四元式保存，四元式的格式为[运算符，操作数 1，操作数 2，运算结果]，各个运算符的具体含义见下表：

表 1.1 四元式运算符含义

运算符	含义	运算符	含义
+	加法运算	j=	条件相等跳转
-	减法运算	<	逻辑判断：小于
*	乘法运算	>	逻辑判断：大于
/	除法运算	<=	逻辑判断：小于等于
=	赋值运算	>=	逻辑判断：小于等于
cin	输入	==	逻辑判断：相等
cout	输出	&&	逻辑与
j	无条件跳转		逻辑或

1.5 单词的识别模型-有穷自动机 DFA

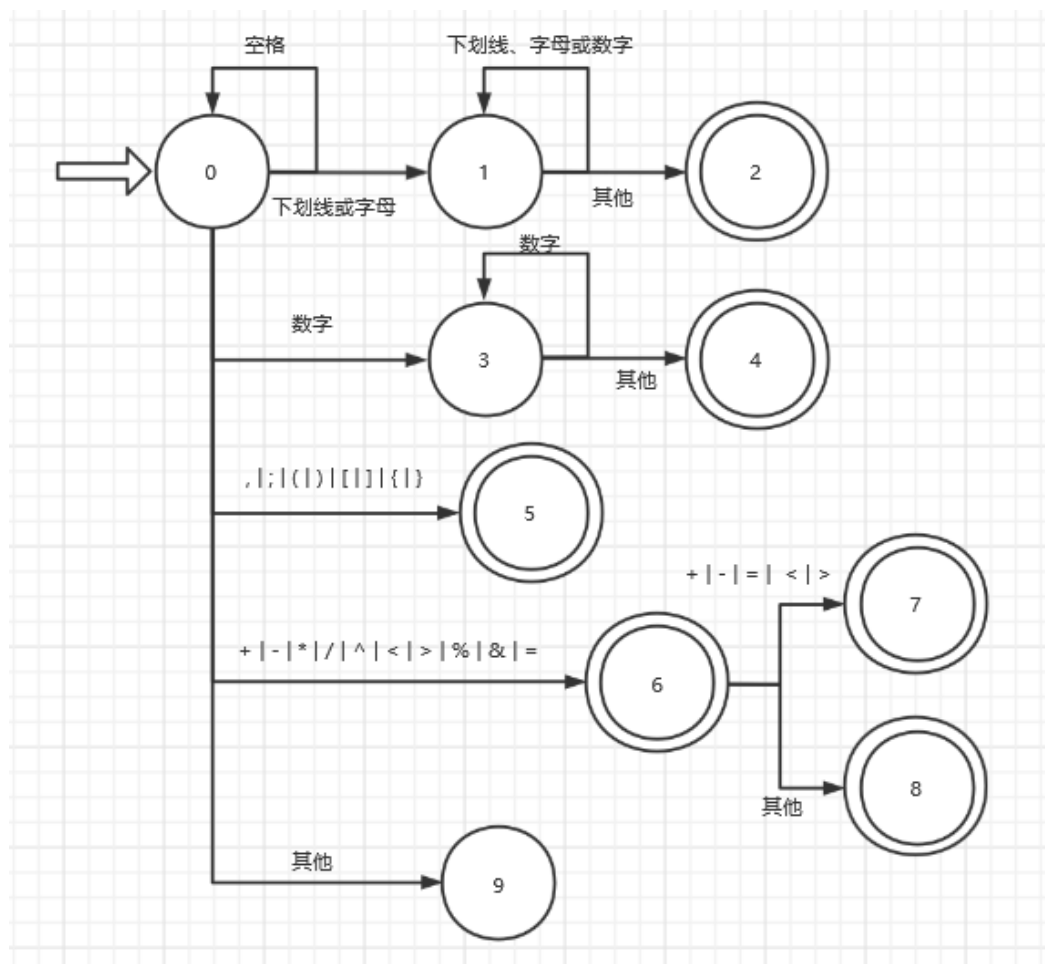


图 1.4 单词的识别模型

2 词法分析程序的实现

算法的基本任务是从字符串表示的源程序中识别出其具有独立意义的单词符号，其基本思想是根据扫描到的单词符号的第一个字符的种类，拼出相应的单词符号。

2.1 属性字定义

2.1.1 标识符

$ID \rightarrow (<字母> | <下划线>)(<字母> | <下划线> | <数字>)^*$

标识符的命名规则为：由字符、数字、下划线组成，开头是字母或者下划线，且区分大小写。

2.1.2 整型数字

$NUM \rightarrow (- | \varepsilon)(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$

整型数据赋值规则为：对首位数字是否为零不做要求，如果是负数需要加负号，对数据的范围大小未做出明确的限制。

2.1.3 浮点型数字

$NUM \rightarrow (- | \varepsilon)(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*.(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*(e(0|1|2|3|4|5|6|7|8|9)^* | \varepsilon)$

浮点型数据赋值规则为：对首位数字是否为零不做要求，如果是负数需要加负号，对数据和精度的范围大小都未做出明确的限制，此外需要注意的是加入识别科学计数法的文法，因此允许在数字中最多仅出现一次字母 e。

2.1.4 保留字

auto, enum, restrict, unsigned, break, extern, return, void, case, float, short, volatile, char, for, signed, while, const, goto, sizeof, _Bool, continue, if, static, _Complex, default, inline, struct, _Imaginary, do, int, switch, double, long, typedef, else, register, union, scanf, printf, cin, cout

2.1.5 分界符

, ; () [] { }

2.1.6 运算符

+, -, *, /, ^, <, >, ++, --, ==, !=, /=, >=, <=, <<, >>, >>=, <<=, %, &, =

2.2 单词判断类型

在 lexical_analysis.h 文件定义了 is_oper(string s)判断操作符、is_bound(char c)判断分界符、is_key(string s)判断关键字（保留字）、my_stoi(string s)强制转换 string to int、my_stof(string s)强制转换 string to double 的全局函数，其中判断操作符、分界符、关键字都是使用在查询的方式实现，即遍历该数组逐个比较单词是否相同即可判断是否操作符、分界符、关键字。

在强制类型转换的函数中，这里的类型转换并不是实现识别函数中的类型转换，而是为了在程序运行过程中更快处理的字符串和数字定义的，它为程序服务的。将 string 类型转换为 int 类型时，由于程序是从左向右读入，但是 int 类型的变量是从左向右是次幂降低的顺序，因此每读入一个数字将收到的数字乘以 10 再加上读入的个位，由此直到达到 string 的长度为止。

在强制转换 string 为 double 的数据时，需要注意本程序实现了科学计数法的识别，因此需要定位小数点和字母 e 的位置消息，针对小数点和 e 的出现情况可以分为以下两种情况：小数点出现，但是 e 没有出现，即是普通 double 类数据；小数点和 e 都出现即是科学计数法，由于前面程序运行过程已经过滤掉了科学计数法的格式错误的情况，这里只是识别正确的情况，并将其转换为正确的 double 类型的数据，之后需要识别小数点和 e 位置，小数点必须在 e 的前面，小数点后面的是小数部分，e 后面的次幂的部分，最后通过 math 库中的 pow 函数完成赋值运算。词法分析实现的过程由有穷自动机执行。

2.3 单词读入程序设计

该功能在 lexical_analysis.h 中 getword 函数中实现，采用的方法是逐行读入、逐个分析，如果读到的是空格，则自动过滤；如果是字母或者下划线开头的单词，则又可能是标识符也有可能是保留字，通过检测下一个读入的是否是数字、字母或下划线来判断结束，读完该单词之后，通过初始化建立的保留字的数组来查询，如果与保留字相同则证明读入的是保留字，否则就认为读入的是自定义的标识符；如果以数字开头，则逐个读入，直到读入不是数字、小数点或字母 e，因为这里考虑到了科学计数法的表示，并在读入的过程中判定是否存在小数点和字母 e，如果最后一位是小数点或者 e，则是不合法的输入，如果先读到 e 都读到小数点也是不合法的输入，如果没有读到小数点和 e 的话则证明就是普通的 int 类型的变量，否则排除上述的情况就是规范的科学计数法的表示形式；如果以读到逗号、分号、左右括号则证明是分界符；如果以操作符开头，则需要判断下一个读入的会不会也是操作符，因为操作符不一定是只有一位的，如自增运

算符；如果是其他情况，则单独标记为其他的类型；读到的每一个单词全部存入单词的结构体中，该结构体于 struct.h 文件中定义，其他文件引用即可。

2.4 词法分析流程图

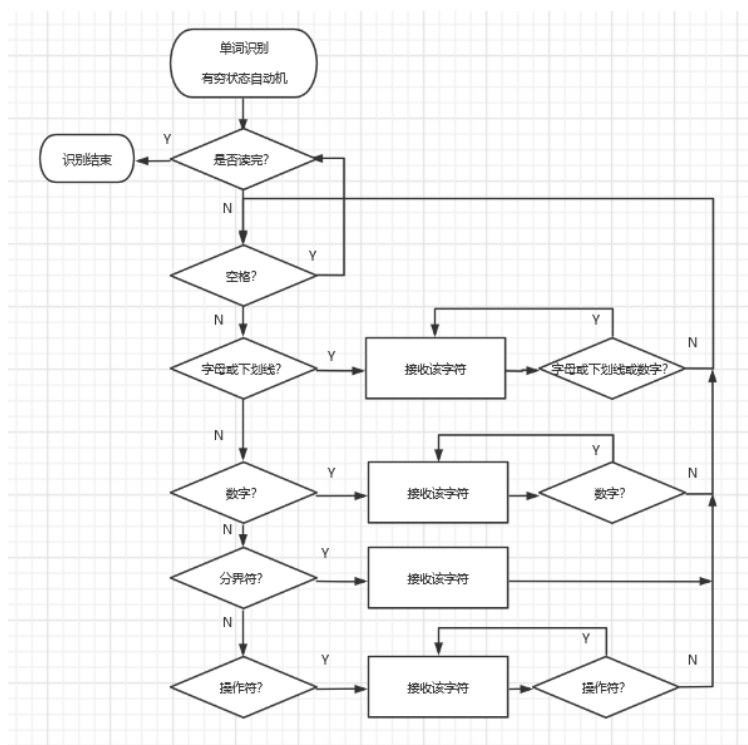


图 2.1 词法分析流程图

2.5 词法分析结果输出

将词法分析中识别的单词全部都输出到 main_table.txt 文件中，内容是识别的名字以及该单词的 id 的值，id 的设置为 1.保留字 2.标识符 3.数字 4.分界符 5.操作符 6.其他，以正确样例作为输入程序，下为正确的程序代码，为方便观看删去部分换行：

```

int main()
{
    int ans; int m; int n; int t; double f; f = 3.141e2;
    m = 1; t = 0; ans = 0; cin >> n; ans = ans + 10 + m; ans = ans + m;
    while(n > 0)
    {
        ans = ans * 10 + n % 10;
        n = n / 10;
    }
}

```

```

for(m = 10;m>0;m--)
{
    t = t + 2;
    if(t<10)
    {
        t = t + 10; f = f + 1.2;
    }
    else {
        t = t - 5;
    }
}

cout << ans << t << f;
return 0;
}

```

以下为 id_table.txt、number_table.txt、main_table.txt、watest.txt 文件的截图：

id_table.txt - 记事本	number_table.txt - 记事本	main_table.txt - 记事本	watest.txt - 记事本
文件(F) 编辑(E)	文件(F) 编辑(E) 格式(O)	文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)	文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
ans	1.int 2.double	1.保留字 2.标识符 3.数字 4.分界符 5.操作符 6.其他	value type number line
f			int type 1 1
m	2 314.1	1 int	main id 2 1
main	1 1	2 main	((4 1
n	1 0	4 ()) 4 1
t	1 0	4)	{ { 4 2
	1 10	4 {	int type 1 3
	1 0	1 int	ans id 2 3
	1 10	2 ans	; ; 4 3
	1 10	4 ;	int type 1 4
	1 10	1 int	m id 2 4
	1 10	2 m	; ; 4 4
	1 0	4 ;	int type 1 5
	1 2	1 int	n id 2 5
	1 10	2 n	; ; 4 5
	1 10	4 ;	int type 1 6
	2 1.2	1 int	t id 2 6
	1 5	2 t	; ; 4 6
	1 0	4 ;	double type 1 7
		1 double	f id 2 7
		2 f	; ; 4 7
		4 ;	f id 2 8
		2 f	= = 5 8
		5 =	3.141e2 number 3 8
		3 3.141e2	; ; 4 8
		4 ;	m id 2 9

图 2.2 词法分析结果部分展示

3 LL(1)实现语法分析程序

语法分析要处理的输入是词法分析的输出即单词序列。该部分主要用到两个栈，一个用来保存单词标志号的状态栈另一是用来保存单词本身信息的符号栈。程序从词法分析输出的单词序列中读取一个单词，检查单词是否是合法单词，如果是合法的，则取符号栈的栈顶元素，如果在当前状态下的预测分析表中没有对应的单词类型则出错并给出出错的信息，整个程序的语法分析出错，退出该程序不再继续执行后续的语义分析以及中间代码生成的阶段功能，若程序正确则持续读入直到词法分析输出结果全部遍历完。

本程序实现了输入一个自定义的文法，即可根据 LL(1) 文法的规则输出该文法的 first 集、follow 集、select 集以及预测分析表，文法可以是 C 子集或者其他正确的文法，以保证输入的程序能够完全符合输入文法的要求，本实验以 C 子集文法作为实现。

3.1 读入文法

打开 grammar.txt 文件，按照逐行读入文法，例如对于文法中的第一句：<函数定义> -> <类型> <变量> (<参数声明>) { <函数块> }，则读出的结果为 foo: <函数定义>，s: <类型> <变量> (<参数声明>) { <函数块> }，并将读入的文法保存到文法的容器中。

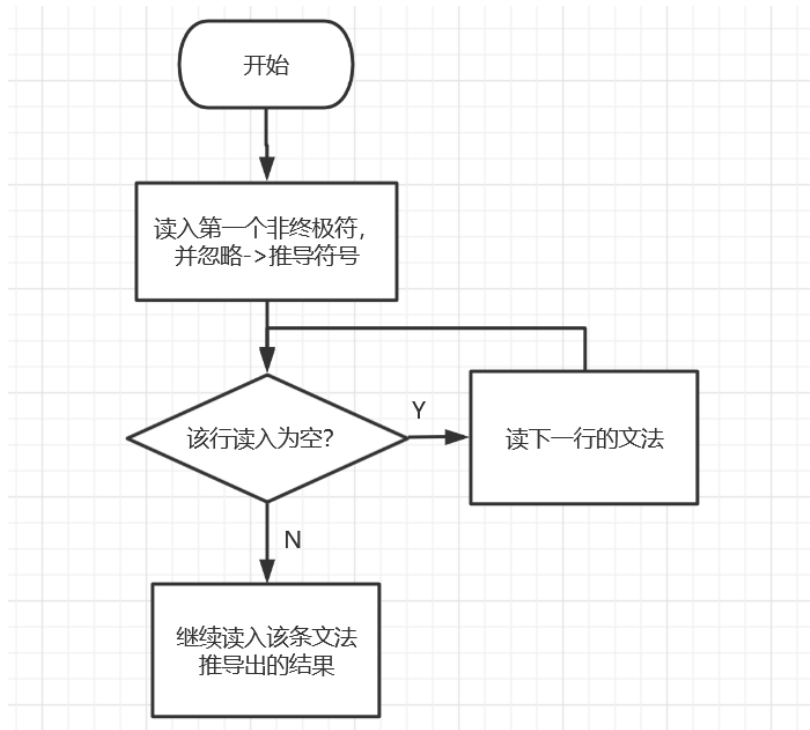


图 3.1 读入文法的流程图

3.2 Get first 集

根据读入的文法来生成 first 集，对于 $X \rightarrow Y_1 Y_2 Y_3 \dots$ ，首先根据读入文法的顺序得到第一个需要更新 first 集的非终极符 X ，再使用一个变量保存 X 的推导式的结果，逐个分析 Y_1 ，如果 Y_1 是终极符则直接将 Y_1 添加到 X 的 first 集，如果 Y_1 是非终极符，则需要将 Y_1 的 first 集添加到 X 的 first 集中，如果 Y_1 能够推导出空，则一直像后面寻找直到读到终极符或者是非终极符不能推导出空的，再将该 first 集添加到 X 的 first 集，由此遍历每一个产生式。求解 first 集的具体定义如下图：

计算文法符号 X 的 $FIRST(X)$ ，不断运用以下规则直到没有新终结符号或 ϵ 可以被加入为止：

- (1) 如果 X 是一个终结符号，那么 $FIRST(X) = X$ 。
- (2) 如果 X 是一个非终结符号，且 $X \rightarrow Y_1 Y_2 \dots Y_k$ 是一个产生式，其中 $k \geq 1$ ，那么如果对于某个 i ， a 在 $FIRST(Y_1)$ 、 $FIRST(Y_2)$... $FIRST(Y_{i-1})$ 中，就把 a 加入到 $FIRST(X)$ 中。
- (3) 如果 $X \rightarrow \epsilon$ 是一个产生式，那么将 ϵ 加入到 $FIRST(X)$ 中。

图 3.2 求解 first 集过程

根据上述文法所得到的 first 集输出结果见附录。

3.3 Get follow 集

继续根据读入的文法求解 follow 集，同样对于 $X \rightarrow Y_1 Y_2 Y_3$ 来说，将 Y_2 中的 first 集出去空全部添加到 Y_1 中的 follow 集，如果 Y_2 能够推导出空，则继续向后寻找，如果找到终极符则直接添加到 follow 集且该非终极符的 follow 集在本条文法中已经添加完成，继续向后寻找非终极符，为它们更新 follow 集。此外，寻找该条文法的推导式的末尾是否为一个非终极符或者是从后向前第一个 first 集中不含有空的非终极符，将 X 的 follow 集给添加到该非终极符的 follow 集中。

求解 follow 集的具体定义如下图：

计算文法符号 X 的 $FOLLOW(X)$ ，不断运用以下规则直到没有新终结符号可以被加入任意 FOLLOW 集合为止：

- (1) 将 $\$$ 加入到 $FOLLOW(X)$ 中，其中 S 是开始符号，而 $\$$ 是输出右端的结束标记。
- (2) 如果存在一个产生式 $S \rightarrow \alpha X \beta$ ，那么将集合 $FIRST(\beta)$ 中除 ϵ 外的所有元素加入到 $FOLLOW(X)$ 当中。
- (3) 如果存在一个产生式 $S \rightarrow \alpha X$ ，或者 $S \rightarrow \alpha X \beta$ 且 $FIRST(\beta)$ 中包含 ϵ ，那么将集合 $FOLLOW(S)$ 中的所有元素加入到集合 $FOLLOW(X)$ 中。

图 3.3 求解 follow 集过程

根据上述文法所得到的 follow 集输出结果见附录。

3.4 Get select 集

当某一非终结符的产生式中含有空产生式时，它的非空产生式右部的开始符号集两两不相交，并与在推导过程中紧跟该非终结符右部可能出现的终结符集也不相交，则仍可构造确定的自顶向下分析。通过上述求解的 first 集和 follow 集来求解 select 集，具体过程见下图：

- 一个产生式的选择符号集 SELECT。给定上下文无关文法的产生式 $A \rightarrow \alpha, A \in V_N, \alpha \in V^*$ ，若 $\alpha \xrightarrow{*} \varepsilon$ ，则 $SELECT(A \rightarrow \alpha) = FIRST(\alpha)$ 。
- 如果 $\alpha \not\xrightarrow{*} \varepsilon$ ，则 $SELECT(A \rightarrow \alpha) = (FIRST(\alpha) - \{\varepsilon\}) \cup FOLLOW(A)$ 。

因此一个上下文无关文法是 LL(1) 文法的充分必要条件是，对每个非终结符 A 的两个不同产生式， $A \rightarrow \alpha, A \rightarrow \beta$ ，满足

$$SELECT(A \rightarrow \alpha) \cap SELECT(A \rightarrow \beta) = \emptyset$$

图 3.4 求解 select 集过程

根据上述文法所得到的 select 集输出结果见附录。

3.5 输出预测表

构造分析表 M 的算法是：

- (1) 对文法 G 的每个产生式 $A \rightarrow a$ 执行求解 first 集和 follow 集；
- (2) 对每个终结符 $a \in FIRST(a)$ ，把 $A \rightarrow a$ 加至 $M[A, a]$ 中；
- (3) 若 $\varepsilon \in FIRST(a)$ ，则把任何 $b \in FOLLOW(A)$ 把 $A \rightarrow a$ 加至 $M[A, b]$ 中；
- (4) 把所有无定义的 $M[A, a]$ 标上出错标志。

根据上述文法所得到的预测分析表输出结果见附录。

3.5 语法分析过程

步骤 | 当前栈顶 | 当前串首 | 推导所用产生式

1	<函数定义>	type	<函数定义> -> <类型> <变量> (<参数声明>) { <函
2	<类型>	type	<类型> -> type
3	type	type	"type" 匹配
4	<变量>	id	<变量> -> <标志符> <数组下标>
5	<标志符>	id	<标志符> -> id
6	id	id	"id" 匹配
7	<数组下标>	(<数组下标> -> @
8	(("(" 匹配
9	<参数声明>)	<参数声明> -> @
10))	")" 匹配
11	{	{	"{" 匹配
12	<函数块>	type	<函数块> -> <声明语句闭包> <函数块闭包>
13	<声明语句闭包>	type	<声明语句闭包> -> <声明语句> <声明语句闭包>
14	<声明语句>	type	<声明语句> -> <声明>;
15	<声明>	type	<声明> -> <类型> <变量> <赋初值>
16	<类型>	type	<类型> -> type
17	type	type	"type" 匹配
18	<变量>	id	<变量> -> <标志符> <数组下标>
19	<标志符>	id	<标志符> -> id
20	id	id	"id" 匹配
21	<数组下标>	;	<数组下标> -> @
22	<赋初值>	;	<赋初值> -> @
23	;	;	";" 匹配
24	<声明语句闭包>	type	<声明语句闭包> -> <声明语句> <声明语句闭包>

图 3.5 语法分析过程 (1)

527	<表达式>	id	<表达式> -> <因子> <项>
528	<因子>	id	<因子> -> <因式> <因式递归>
529	<因式>	id	<因式> -> <变量>
530	<变量>	id	<变量> -> <标志符> <数组下标>
531	<标志符>	id	<标志符> -> id
532	id	id	"id" 匹配
533	<数组下标>	;	<数组下标> -> @
534	<因式递归>	;	<因式递归> -> @
535	<项>	;	<项> -> @
536	<cout闭包>	;	<cout闭包> -> @
537	;	;	";" 匹配
538	<函数块闭包>	return	<函数块闭包> -> <函数返回> <函数块闭包>
539	<函数返回>	return	<函数返回> -> return <因式>;
540	return	return	"return" 匹配
541	<因式>	number	<因式> -> <数字>
542	<数字>	number	<数字> -> number
543	number	number	"number" 匹配
544	;	;	";" 匹配
545	<函数块闭包>	}	<函数块闭包> -> @
546	}	}	"}" 匹配
547	#	#	接受

-->LL(1)合法句子

图 3.6 语法分析过程 (2)

4 语义分析程序以及四元式生成

4.1 优先级定义

在函数 `init_pri` 中实现优先级的定义，结果见下表：

表 4.1 优先级定义

符号	优先级	符号	优先级
(10	>=	6
++	9	<=	6
--	9	!=	5
*	8	==	5
/	8	&&	4
%	8		3
+	7	=	2
-	7)	1
>	6	@	1
<	6		

4.2 逻辑值以及数值计算

在 `Caculate` 函数中，当前需要翻译的单词序号存放在全局的函数中，因此只要给出该数值计算或者逻辑值计算的结束符号，计算出最终的结果并返回主函数则可以完成赋值计算和逻辑判断的功能。首先建立 `num` 和 `opr` 的栈，先给 `opr` 入栈一个 `@` 表示空，然后进入死循环开始完成计算。

读入当前单词的值，如果是左侧小括号则表示进入一层括号的嵌套计算，需要等待一个与之匹配的右侧小括号以及来判断是否计算结束，并在这个过程中完成处理逆波兰式。在表达式处理过程中，如果当前的单词是变量或者常量时，则进行入栈并匹配四元式。否则即是读入的运算符，则需要读入栈顶的两个数值，根据不同的运算符进行相关的计算，之后生成相应的四元式存入四元式的存储结构中。

4.3 主函数部分生成四元式

在 `body` 函数中完成大部分的四元式的生成以及检验输入程序的正确性，函数执行的过程也是一个死循环的过程，它的结束方式为识别到 `'}'` 返回逻辑真值或者未识别子集设定的关键词或 `id` 则结束循环并返回逻辑假的值。

4.3.1 声明语句

当前第一个单词属性值为"type"，并且期望着下一个单词的属性是 id 的属性，并将该变量存入变量栈中，并在最后期望得到一个分号作为该语句的结束标志。

4.3.2 cin 语句

当前第一个单词属性值为"cin"，并在在此之后一直期望着">>"、"id"，每出现一次这两个单词的属性则证明在该句中存在着一个输入，并对应生成输入格式的四元式：
`Four_exp{"cin", "_", "_", last.value}`，last 是上一个识别的 id 名。

4.3.3 cout 语句

当前第一个单词属性值为"cout"，并在在此之后一直期望着"<<"、"id"，每出现一次这两个单词的属性则证明在该句中存在着一个输出，并对应生成输入格式的四元式：
`Four_exp{"cout", "_", "_", last.value}`，last 是上一个识别的 id 名。

4.3.4 赋值语句

当前第一个单词属性值为"id"，首先在变量表中去寻找该变量名，查找方式是利用 stl 中的 find 函数，如果没有在变量名中没有找到该变量名，则迭代器位置为该容器的最后末尾位置加一，否则进行赋值的功能。首先期望一个等于符号，再去利用计算器的函数去计算需要赋值的数值大小并返回，最后将该数值赋予等于符号前面的 id 属性，并对应生成输入格式的四元式：`Four_exp{"=", tt, "_", left.value}`。

4.3.5 if 语句

当前第一个单词属性值为"if"，并且期望着下一个单词的属性是"("，进入逻辑判断语句，利用计算器函数计算逻辑值并返回判断是否需要进入该函数体，并此时生成四元式 `Four_exp{"J=", tt, "1", "(" + itos(four_exp.size()+2) + ")"}`，`Four_exp{"J", "_", "_", ""}`，第一句表示判断的逻辑值是真，则需要进入该函数体，并且函数体的位置在下面两个的四元式位置，否则的话跳到 else 的部分或者没有 else 就直接到 if 最后的}符号的四元式后面，这里先设置一个临时变量记录此时没有填好完整跳转位置的四元式的序号信息，在之后完成 if 函数体中内容后返回填写，if 的函数体部分直接使用递归程序再次执行

body 的主函数，由此该 if 中涉及到的嵌套与该 if 没有直接关系，该 if 也不能去管理内部嵌套的跳转位置，由记录序号临时变量处理。

如果存在关于这个 if 的 else 语句，则证明上一步的假出口位置已经找到，否则就直接在该位置填假出口位置即可。继续讨论存在 else 的情况，首先需要有一个跳到 else 结束的位置四元式信息，因为 if 逻辑值为真的式子已经执行完成了，所以需要跳转结束位置，因此在这个位置继续设置临时变量存放出口位置，接着使用递归程序执行 else 的函数体，执行完成后填回出口位置。

4.3.5 while 语句

当前第一个单词属性值为"while"，并且期望着下一个单词的属性是"("，利用计算器函数计算逻辑值，如果为假就直接跳出，生成四元式 `Four_exp{"J=", tt, "1", "(" + itos(four_exp.size()+2) + ")"}`，`Four_exp{"J", "_", "_", ""}`，假出口位置的填写原理同上，之后继续递归 body 函数，执行完函数体内，跳回 while 的判断语句，最后回填假出口的位置。

4.3.5 for 语句

当前第一个单词属性值为"for"，并且期望着下一个单词的属性是"("，首先进行赋值语句的功能，并在对应的函数中完成赋值语句的四元式的生成，之后给出分号作为结束位置计算逻辑表达式的值，并生成四元式 `Four_exp{"J=", tt, "1", "(" + itos(four_exp.size()+2) + ")"}`，并用临时变量记录逻辑判断的四元式位置，接着是逻辑假的四元式，`Four_exp{"J", "_", "_", ""}`，同理采用临时变量记录四元式序号，不过这个是为了回填假出口，之后计算后缀表达式的值并存入临时变量，在运行完成一次函数体内容后再赋值给该后缀表达式的变量。接着执行函数体内容，for 的函数体部分直接使用递归程序再次执行 body 的主函数，由此该 for 中涉及到的嵌套与该 for 没有直接关系，该 for 也不能去管理内部嵌套的跳转位置，由记录序号临时变量处理。

执行完函数体内，接着执行 for 的后缀表达式，之后跳回 for 的逻辑判断语句，并接上假出口，至此 for 语句编写完成。

表 4.2 单词属性判断归属语句

属性	语句	符号	优先级
type	声明语句	if	if 语句
cin	输入语句	while	while 语句
cout	输出语句	for	for 语句
id	赋值语句		

4.4 四元式正反例结果分析

输入程序同词法分析中的代码，这部分展示四元式的生成结果，程序代码以及四元式结果如下图所示：

4.4.1 正例程序以及四元式

```
int main()
{
    int ans; int m; int n; int t; double f; f = 3.141e2;
    m = 1; t = 0; ans = 0; cin >> n; ans = ans + 10 + m; ans = ans + m;
    while(n > 0)
    {
        ans = ans * 10 + n % 10;
        n = n / 10;
    }
    for(m = 10; m > 0; m--)
    {
        t = t + 2;
        if(t < 10)
        {
            t = t + 10; f = f + 1.2;
        }
        else {
            t = t - 5;
        }
    }

    cout << ans << t << f;
    return 0;}
```

(0) (=, 3.141e2, _, f)	(1) (=, 1, _, m)	(2) (=, 0, _, t)
(3) (=, 0, _, ans)	(4) (cin, _, _, n)	(5) (+, ans, 10, T0)
(6) (+, T0, m, T1)	(7) (=, T1, _, ans)	(8) (+, ans, m, T2)
(9) (=, T2, _, ans)	(10) (>, n, 0, T3)	(11) (J=, T3, 1, (13))
(12) (J, _, _, (20))	(13) (*, ans, 10, T4)	(14) (% , n, 10, T5)
(15) (+, T4, T5, T6)	(16) (=, T6, _, ans)	(17) (/ , n, 10, T7)
(18) (=, T7, _, n)	(19) (J, _, _, (10))	(20) (=, 10, _, m)
(21) (>, m, 0, T8)	(22) (J=, T8, 1, (24))	(23) (J, _, _, (39))
(24) (--, m, _, T9)	(25) (+, t, 2, T10)	(26) (=, T10, _, t)
(27) (<, t, 10, T11)	(28) (J=, T11, 1, (30))	(29) (J, _, _, (35))
(30) (+, t, 10, T12)	(31) (=, T12, _, t)	(32) (+, f, 1.2, T13)
(33) (=, T13, _, f)	(34) (J, _, _, (37))	(35) (-, t, 5, T14)
(36) (=, T14, _, t)	(37) (=, T9, _, m)	(38) (J, _, _, (21))
(39) (cout, _, _, ans)	(40) (cout, _, _, t)	(41) (cout, _, _, f)

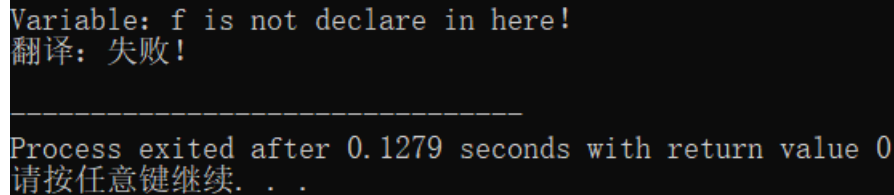
4.4.2 反例程序以及四元式

反例 1，出现变量未定义即使用，这里为 double 类型的 f 变量未定义。

```
int main()
{
    int ans; int m; int n; int t;
    f = 3.141e2; m = 1; t = 0; ans = 0;
    cin >> n;
    ans = ans + 10;  ans = ans + m;
    while(n > 0)
    {
        ans = ans * 10 + n % 10;
        n = n / 10;
    }
    for(m = 10; m > 0; m--)
    {
        t = t + 2;
        if(t < 10)
        { t = t + 10; f = f + 1.2; }
        else { t = t - 5; }
```

```
}  
cout << ans;  
return 0;}
```

报错截图：

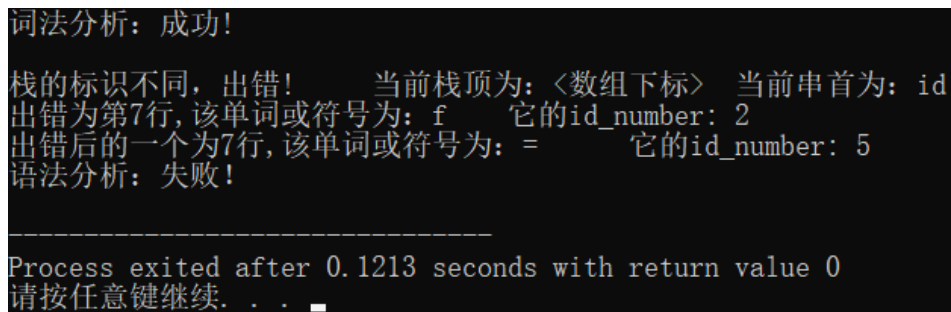


```
Variable: f is not declare in here!  
翻译：失败！  
  
-----  
Process exited after 0.1279 seconds with return value 0  
请按任意键继续. . .
```

图 4.1 反例 1 报错提示

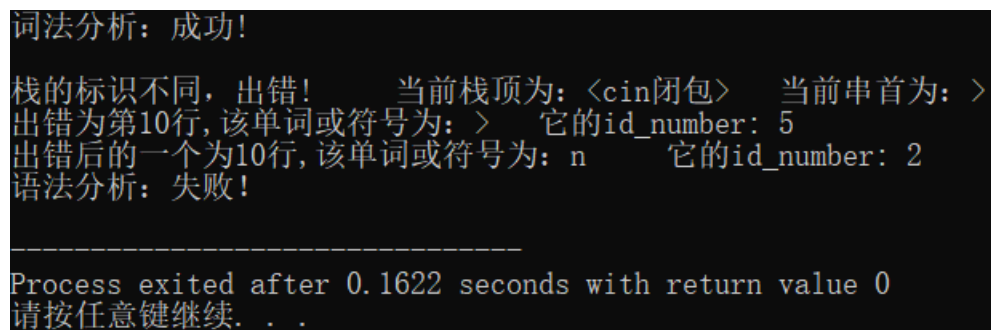
同理其余错误代码大体相同，只是错误地点不一致，因此不展示完整错误样例代码，直接展示错误的原因以及报错的界面。

反例 2，句尾没有出现分号；反例 3，cin 少一个<符号。



```
词法分析：成功！  
  
栈的标识不同，出错！    当前栈顶为：<数组下标>    当前串首为：id  
出错为第7行，该单词或符号为：f    它的id_number: 2  
出错后的一个为7行，该单词或符号为：=    它的id_number: 5  
语法分析：失败！  
  
-----  
Process exited after 0.1213 seconds with return value 0  
请按任意键继续. . .
```

图 4.2 反例 2 报错提示



```
词法分析：成功！  
  
栈的标识不同，出错！    当前栈顶为：<cin闭包>    当前串首为：>  
出错为第10行，该单词或符号为：>    它的id_number: 5  
出错后的一个为10行，该单词或符号为：n    它的id_number: 2  
语法分析：失败！  
  
-----  
Process exited after 0.1622 seconds with return value 0  
请按任意键继续. . .
```

图 4.3 反例 3 报错提示

5 汇编语言生成

生成汇编语言的原理比较简单，即直接根据四元式的操作符来对应相应的汇编语言，因此在编写过程中没有全局性的考虑汇编语言的调用库以及寄存器的使用等。

输入程序的代码同上，以下为汇编语言的生成代码：

(0) MOV f, 3.141e2	(1) MOV m, 1	(2) MOV t, 0
(3) MOV ans, 0	(4) IN n	(5) MOV R0, ans
(6) ADD R0, 10	(7) MOV T0, R0	(8) MOV R1, T0
(9) ADD R1, m	(10) MOV T1, R1	(11) MOV ans, T1
(12) MOV R2, ans	(13) ADD R2, m	(14) MOV T2, R2
(15) MOV ans, T2	(16) CMP n, 0	(17) JG (20)
(18) MOV T3, 0	(19) J (21)	(20) MOV T3, 1
(21) CMP T3, 1	(22) JZ (23)	(23) JMP (39)
(24) MOV R4, ans	(25) MUL R4, 10	(26) MOV T4, R4
(27) MOV R5, n	(28) MOD R5, 10	(29) MOV T5, R5
(30) MOV R6, T4	(31) ADD R6, T5	(32) MOV T6, R6
(33) MOV ans, T6	(34) MOV R7, n	(35) DIV R7, 10
(36) MOV T7, R7	(37) MOV n, T7	(38) JMP (10)
(39) MOV m, 10	(40) CMP m, 0	(41) JG (44)
(42) MOV T8, 0	(43) J (45)	(44) MOV T8, 1
(45) CMP T8, 1	(46) JZ (47)	(47) JMP (39)
(48) MOV R9, t	(49) ADD R9, 2	(50) MOV T10, R9
(51) MOV t, T10	(52) CMP t, 10	(53) JNG (56)
(54) MOV T11, 0	(55) J (57)	(56) MOV T11, 1
(57) CMP T11, 1	(58) JZ (60)	(59) JMP (35)
(60) MOV R11, t	(61) ADD R11, 10	(62) MOV T12, R11
(63) MOV t, T12	(64) MOV R12, f	(65) ADD R12, 1.2
(66) MOV T13, R12	(67) MOV f, T13	(68) JMP (37)
(69) MOV R13, t	(70) SUB R13, 5	(71) MOV T14, R13
(72) MOV t, T14	(73) MOV m, T9	(74) JMP (21)
(75) OUT ans	(76) OUT t	(77) OUT f

感 想

通过该课程设计,掌握了编译程序工作的基本过程及其各阶段的基本任务,熟悉了编译程序总流程框图。通过该程序算法的内容,对实验原理有更深入的理解,使我对抽象的理论知识有了更加具体的认识,自己动手写程序能够把课堂上学的知识通过自己设计的程序表示出来,加深了对课上学习的理论知识的理解。并且通过该课程设计激发了学习的积极性,全面理解了编译原理程序构造的一般原理和基本实现方法,对于编译原理考试的复习也大有裨益。

编译原理的编译过程一般包括:词法分析、语法分析、语义分析与中间代码产生、优化、目标代码生成五个阶段。通过本次设计,使我对编译原理有了进一步的了解,更加巩固了所学习的知识。我所选择的语言是子集所熟悉的 C 语言子集,这样在网上查阅相关资料时能够充分的理解并加以运用,使用 LL (1)文法的编译器自动生成文法的 first 集、follow 集、select 集以及预测分析表。语法分析的主要任务就是在词法分析的基础上,根据语言的语法规则,把单词符号串分解成各类语法单位,在语法分析阶段使用自顶向下分析的方法采用两个栈顶的匹配来实现 LL (1)文法的匹配。通过语法分析,确定整个输入串是否构成语法上正确的程序。在中间代码的生成阶段,使用首单词属性匹配的方法,根据其属性判断是 C 子集中的哪一种语句并完成相对应的翻译四元式的工作,在回填真假出口的时候使用临时变量记录四元式序号并在完成对应的功能后进行回填,有效避免了多次嵌套时真假出口次序混杂在一起的情况,并可以在理论上实现多层的嵌套。在翻译汇编语言时,根据每个四元式的操作符含义生成对应相对简单的汇编语言,能够使其描述基本的程序的思想与实现功能。

在整个程序的编写过程中,自己也遇到了若干问题,在调试程序过程中经常会出现自己未意料到的错误,因此在调试程序时需要一定的耐心,并勤于在网络寻找问题的答案为自己提供参考。比如在编写有关循环程序时,出现了未进入函数体的情况,导致了后续程序没有完成翻译,起初认为是直接使用递归程序导致的错误,通过仔细的检查发现是在计算完成相关的逻辑判断后未完成识别该句的终止符号,从而导致没有进入嵌套的函数体程序。此外,在编写 C 子集文法的过程中,还需要理解文法中对应的含义,并通过自己对 C 语言的理解调整对应的文法规则,使得最后的输入程序可以按照自己常规的理解完成编译程序的功能。

最后,非常感谢毕老师在编译课程上的指导与讲解,在课程答辩过程中,老师长时间坚持在机房参与答辩使得我们同学都能在预期的时间内完成自己的答辩,并充足展示自己的工作与成果,感谢老师在这一学期的辛苦工作。

附录 语法分析阶段结果展示

1 该 C 子集文法的 first 集结果

FIRST(<cin 闭包>): >> @
FIRST(<cout 闭包>): << @
FIRST(<for 循环>): for
FIRST(<while 循环>): while
FIRST(<变量>): id
FIRST(<标志符>): id
FIRST(<表达式>): (id number
FIRST(<参数>): id number
FIRST(<参数闭包>): , @
FIRST(<参数列表>): id number
FIRST(<参数声明>): @ type
FIRST(<多个数据>): number
FIRST(<否则语句>): @ else
FIRST(<赋初值>): = @
FIRST(<赋值函数>): cin cout id
FIRST(<赋值或函数调用>): (=
FIRST(<函数定义>): type
FIRST(<函数返回>): return
FIRST(<函数块>): @ cin cout for id if return type while
FIRST(<函数块闭包>): @ cin cout for id if return while
FIRST(<后缀表达式>): id
FIRST(<后缀运算符>): ++ --
FIRST(<类型>): type
FIRST(<逻辑表达式>): (id number
FIRST(<逻辑运算符>): != < <= == > >=
FIRST(<声明>): type
FIRST(<声明闭包>): , @
FIRST(<声明语句>): type

FIRST(<声明语句闭包>): @ type
 FIRST(<数字>): number
 FIRST(<数字闭包>): , @
 FIRST(<数组下标>): @ [
 FIRST(<条件语句>): if
 FIRST(<项>): + - @
 FIRST(<因式>): (id number
 FIRST(<因式递归>): % * / @
 FIRST(<因子>): (id number
 FIRST(<右值>): (id number {

2 该 C 子集文法的 follow 集结果

FOLLOW(<cin 闭包>): ;
 FOLLOW(<cout 闭包>): ;
 FOLLOW(<for 循环>): cin cout for id if return while }
 FOLLOW(<while 循环>): cin cout for id if return while }
 FOLLOW(<变量>): != % () * + ++ , - -- / ; < << <= = == > >= >>]
 FOLLOW(<标志符>): != % () * + ++ , - -- / ; < << <= = == > >= >> []
 FOLLOW(<表达式>): !=) , ; < << <= == > >= >>
 FOLLOW(<参数>):) ,
 FOLLOW(<参数闭包>):)
 FOLLOW(<参数列表>):)
 FOLLOW(<参数声明>):)
 FOLLOW(<多个数据>): }
 FOLLOW(<否则语句>): cin cout for id if return while }
 FOLLOW(<赋初值>):) , ;
 FOLLOW(<赋值函数>): (cin cout for id if number return while }
 FOLLOW(<赋值或函数调用>): (cin cout for id if number return while }
 FOLLOW(<函数定义>): #
 FOLLOW(<函数返回>): cin cout for id if return while }
 FOLLOW(<函数块>): }

FOLLOW(<函数块闭包>): }
 FOLLOW(<后缀表达式>):)
 FOLLOW(<后缀运算符>):)
 FOLLOW(<类型>): id
 FOLLOW(<逻辑表达式>):) ;
 FOLLOW(<逻辑运算符>): (id number
 FOLLOW(<声明>):) , ;
 FOLLOW(<声明闭包>):)
 FOLLOW(<声明语句>): cin cout for id if return type while }
 FOLLOW(<声明语句闭包>): cin cout for id if return while }
 FOLLOW(<数字>): != %) * + , - / ; < << <= == > >= >>] }
 FOLLOW(<数字闭包>): }
 FOLLOW(<数组下标>): != % () * + ++ , - -- / ; < << <= = == > >= >>]
 FOLLOW(<条件语句>): cin cout for id if return while }
 FOLLOW(<项>): !=) , ; < << <= == > >= >>
 FOLLOW(<因式>): != %) * + , - / ; < << <= == > >= >>]
 FOLLOW(<因式递归>): !=) + , - ; < << <= == > >= >>
 FOLLOW(<因子>): !=) + , - ; < << <= == > >= >>
 FOLLOW(<右值>):) , ;

3 该 C 子集文法的 select 集结果

SELECT(<赋初值> @):) , ;
 SELECT(<赋值函数> <变量> <赋值或函数调用>): id
 SELECT(<赋值函数> cin <cin 闭包> ;): cin
 SELECT(<赋值函数> cout <cout 闭包> ;): cout
 SELECT(<赋值或函数调用> (<参数列表>) ;): (
 SELECT(<赋值或函数调用> = <右值> ;): =
 SELECT(<函数定义> <类型> <变量> (<参数声明>) { <函数块> }): type
 SELECT(<函数返回> return <因式> ;): return
 SELECT(<函数块> <声明语句闭包> <函数块闭包>): cin cout for id if return
 type while }

SELECT(<函数块闭包> <for 循环> <函数块闭包>): for
 SELECT(<函数块闭包> <while 循环> <函数块闭包>): while
 SELECT(<函数块闭包> <赋值函数> <函数块闭包>): cin cout id
 SELECT(<函数块闭包> <函数返回> <函数块闭包>): return
 SELECT(<函数块闭包> <条件语句> <函数块闭包>): if
 SELECT(<函数块闭包> @): }
 SELECT(<后缀表达式> <变量> <后缀运算符>): id
 SELECT(<后缀运算符> ++): ++
 SELECT(<后缀运算符> --): --
 SELECT(<类型> type): type
 SELECT(<逻辑表达式> <表达式> <逻辑运算符> <表达式>): (id number
 SELECT(<逻辑运算符> !=): !=
 SELECT(<逻辑运算符> <): <
 SELECT(<逻辑运算符> <=): <=
 SELECT(<逻辑运算符> ==): ==
 SELECT(<逻辑运算符> >): >
 SELECT(<逻辑运算符> >=): >=
 SELECT(<声明> <类型> <变量> <赋初值>): type
 SELECT(<声明闭包> , <声明> <声明闭包>): ,
 SELECT(<声明闭包> @):)
 SELECT(<声明语句> <声明> ;): type
 SELECT(<声明语句闭包> <声明语句> <声明语句闭包>): type
 SELECT(<声明语句闭包> @): cin cout for id if return while }
 SELECT(<数字> number): number
 SELECT(<数字闭包> , <数字> <数字闭包>): ,
 SELECT(<数字闭包> @): }
 SELECT(<数组下标> @): != % () * + ++ , - -- / ; < << <= == > >= >>]
 SELECT(<数组下标> [<因式>]): [
 SELECT(<条件语句> if (<逻辑表达式>) { <函数块> } <否则语句>): if
 SELECT(<项> + <因子> <项>): +
 SELECT(<项> - <因子> <项>): -
 SELECT(<项> @): !=) , ; < << <= == > >= >>

SELECT(<因式> (<表达式>)): (
 SELECT(<因式> <变量>): id
 SELECT(<因式> <数字>): number
 SELECT(<因式递归> % <因式> <因式递归>): %
 SELECT(<因式递归> * <因式> <因式递归>): *
 SELECT(<因式递归> / <因式> <因式递归>): /
 SELECT(<因式递归> @): !=) + , - ; < << <= == > >= >>
 SELECT(<因子> <因式> <因式递归>): (id number
 SELECT(<右值> <表达式>): (id number
 SELECT(<右值> { <多个数据> }): {

4 该 C 子集文法的预测分析表结果

	!=	#	%	()	*	+	++	,	-	--	/	;	<	<<
<cin闭包>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	@	\$
<cout闭包>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	@	\$<
<for循环>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<while循环>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<变量>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<标志符>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<表达式>	\$	\$	\$	<因子>	<项>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<参数>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<参数闭包>	\$	\$	\$	\$	@	\$	\$	\$	\$	<参数>	<参数闭包>	\$	\$	\$	\$
<参数列表>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<参数声明>	\$	\$	\$	\$	@	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<多个数据>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<否则语句>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<赋初值>	\$	\$	\$	\$	@	\$	\$	\$	@	\$	\$	\$	@	\$	\$
<赋值函数>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<赋值或函数调用>	\$	\$	\$	\$	\$(<参数列表>);	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<函数定义>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<函数返回>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<函数块>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<函数块闭包>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<后缀表达式>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<后缀运算符>	\$	\$	\$	\$	\$	\$	\$	++	\$	\$	--	\$	\$	\$	\$
<类型>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<逻辑表达式>	\$	\$	\$	<表达式>	<逻辑运算符>	<表达式>	\$	\$	\$	\$	\$	\$	\$	\$	\$
<逻辑运算符>	!=	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	<
<声明>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<声明闭包>	\$	\$	\$	\$	@	\$	\$	\$	\$	<声明>	<声明闭包>	\$	\$	\$	\$
<声明语句>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<声明语句闭包>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<数字>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<数字闭包>	\$	\$	\$	\$	\$	\$	\$	\$	\$	<数字>	<数字闭包>	\$	\$	\$	\$
<数组下标>	@	\$	@	@	@	@	@	@	@	@	@	@	@	@	@
<条件语句>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<项>	@	\$	\$	\$	@	\$	+	<因子>	<项>	\$	@	-	<因子>	<项>	\$
<因式>	\$	\$	\$	\$(<表达式>)	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<因式递归>	@	\$	%	<因式>	<因式递归>	\$	\$	@*	<因式>	<因式递归>	@	\$	\$	\$	\$
<因子>	\$	\$	\$	<因式>	<因式递归>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
<右值>	\$	\$	\$	<表达式>	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$

[illegible]

以上预测分析表为部分截图展示，由于文字过长不做完整展示，详细可以见源文件中 `pre_list.txt`。