

编译原理课程设计

学 院（系）： 计算机科学与技术
学 生 姓 名： 范瀚文
学 号： 201981303
班 级： 电计 1905
联 系 方 式： 19969302360
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	C-Minus 语言描述	1
1.2	C-Minus 语言文法定义	1
1.3	C-Minus 单词识别有穷自动机 DFA.....	2
2	词法分析程序的实现.....	3
2.1	C-Minus 词法符号分类	3
2.2	词法分析类.....	3
2.2.1	词法分析类的定义.....	3
2.2.2	词法分析函数的实现.....	4
2.2.3	词法分析结果的输出.....	5
2.3	符号表管理.....	6
2.3.1	符号表存储结构.....	6
2.3.2	标识符和保留字区分方法.....	7
3	语法分析和语义分析程序的实现.....	7
3.1	语法分析和语义分析类的定义.....	8
3.2	语法分析和语义分析函数的实现.....	8
3.2.1	void Program()函数.....	9
3.2.2	void Define()函数	9
3.2.3	void Start()函数.....	9
3.2.4	void Numgiven()函数	10
3.2.5	void IFCHECK()函数	11
3.2.6	void WhileCheck()函数	11
3.2.7	void FORCHECK()函数	11
3.2.8	void Printout()函数和 void Scanfin()函数	12
3.2.9	string Equal()函数	12
3.2.10	余下的判断函数.....	13
3.3	中间代码的定义.....	13
3.3.1	中间代码-四元式.....	13
3.3.2	拉链回填的实现.....	14
3.4	文法特色与难点	15

4	解释程序的实现.....	16
4.1	解释程序函数的实现.....	16
4.2	顺序执行预期结果.....	16
4.3	调试执行预期结果.....	16
4.4	错误类型.....	17
5	测试用例.....	18
5.1	测试用例 1 正确样例.....	18
5.2	测试用例 2（词法分析错误）.....	19
5.3	测试用例 3（语法分析错误）.....	19
6	感 想.....	20
附录 A	词法分析程序.....	21
1	词法分析类的声明.....	21
2	四元式详细解释.....	22
3	四元式拉链回填实现.....	23
3.1	While-do 函数拉链回填实现和展示.....	23
3.2	if-then-else 函数拉链回填实现核心.....	24
3.3	for 函数拉链回填实现核心.....	26
附录 B	解释程序.....	28
1	四元式的解释翻译结果.....	28
附录 C	全部测试样例.....	29
1	正确用例结果展示.....	29
2	错误用例结果展示.....	31

1 源语言定义

1.1 C-Minus 语言描述

C-Minus 语言是 C 语言的一个子集。完全由我从 C 语言进行删减而得到，它比 C 语言简单，并作了一些限制。C-Minus 的程序结构比较完全，相应的选择，不但支持常量，变量（int, char, string）及过程声明，并且也具有分支选择结构，循环嵌套机构和输入输出结构也一应俱全。

1.2 C-Minus 语言文法定义

C-Minus 语言文法的 EBNF 表示：

```

<程序> ::= <头文件定义> { <分程序> }
<头文件定义> ::= #include <iostream> int main()
<分程序> ::= <变量定义> <执行语句>
<变量定义> ::= <变量类型> <标识符>; { <变量定义> }
<标识符> ::= <字母> { <字母> | <数字> }
<执行语句> ::= <输入语句> | <输出语句> | <赋值语句> | <条件语句> | <While 语句> | <For 语句> | { <执行语句> }
<赋值语句> ::= <标识符> <赋值运算符> <表达式>;
<While 语句> ::= while ( <条件语句> ) do <执行语句>
<For 语句> ::= for ( [ <赋值语句> ] <条件语句>; <赋值语句> ) <执行语句>
<条件语句> ::= if ( <条件> ) then <执行语句> [ else <执行语句> ]
<条件语句> ::= <逻辑或表达式> { || <逻辑或表达式> }
<逻辑或表达式> ::= <逻辑与表达式> { && <逻辑与表达式> }
<逻辑与表达式> ::= <表达式> <关系运算符> <表达式>
<表达式> ::= <按位或表达式> { | <按位或表达式> }
<按位或表达式> ::= <按位与表达式> { & <按位与表达式> }
<按位与表达式> ::= <位移表达式> { <位移运算符> <位移表达式> }
<位移表达式> ::= <乘除表达式> { <加减运算符> <乘除表达式> }
<乘除表达式> ::= <运算式> { <乘除运算符> <运算式> }
<运算式> ::= ( <运算式> ) | <标识符> | <整数>
<乘除运算符> ::= * | /
<加减运算符> ::= + | -

```

〈位移运算符〉 ::= >> | <<

〈关系运算符〉 ::= == | != | < | <= | > | >=

〈赋值运算符〉 ::= = | += | -= | *= | /=

〈输入语句〉 ::= scanf(〈标识符〉)

〈输出语句〉 ::= printf(〈标识符〉)

〈变量类型〉 ::= int | char | string

〈字母〉 ::= a|b|...|X|Y|Z

〈数字〉 ::= 0|1|2|...|8|9

〈整数〉 ::= [-]〈数字〉

其中 加粗的 {}, |, || 表示终极符, 未加粗的表示非终极符

1.3 C-Minus 单词识别有穷自动机 DFA

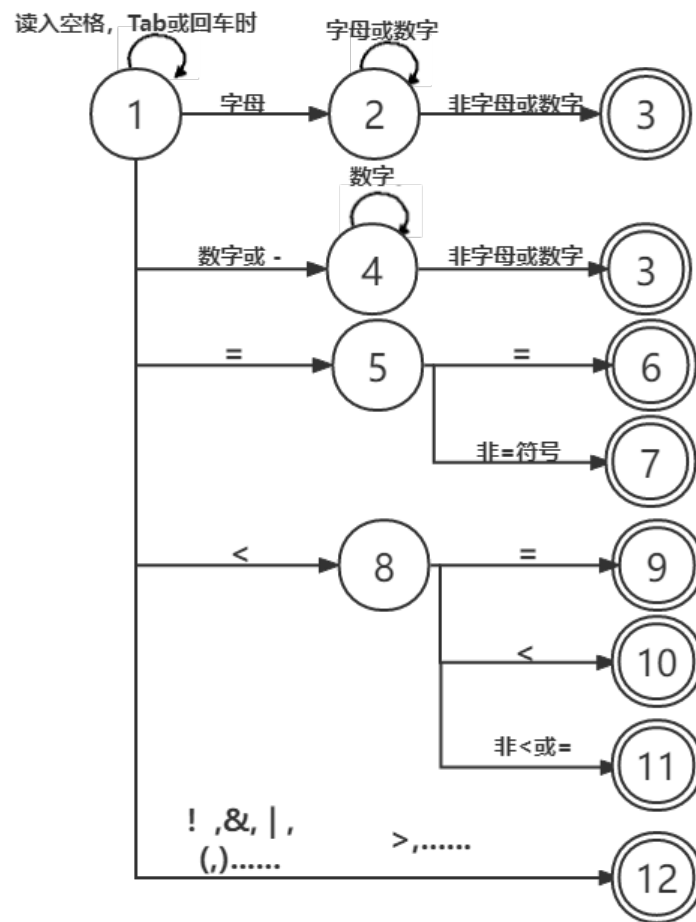


图 1.1 单词识别的有穷自动机

2 词法分析程序的实现

2.1 C-Minus 词法符号分类

C-Minus 编译系统可以看作是 C 语言的一个子集，所以 C-Minus 中所有的字符，字符串的类型均可以在 C 中找到对应的。C-Minus 的所有符号可以大体分为三类：保留字，操作字和 ID 或 NUM，具体的 C-Minus 系统中的所欲类型定义如下表格所示。

表 2.1 C-Minus 编译系统中所有的字符，字符串类型

类别	字符与字符串
保留字	int, char, string, for, while, if, then, else, scanf, printf, include, iostream, main
逻辑运算符	&&,
算数运算符	+, -, *, /
关系运算符	==, !=, >=, <=, >, <
位操作运算符	>>, <<, , &
赋值运算符	=, +=, -=, *=, /=
标识符	整型变量名, 字符型变量名, 字符串变量名, 常数名, 保留字名, 符号名
字符型变量	C, s 等单字节变量
字符串变量	Abc, cde 等字符串变量
常数	10, -25 等整数
其他符号	;, #

2.2 词法分析类

2.2.1 词法分析类的定义

词法分析类主要定义了词法分析的具体函数和相关的数据储存结构，实现了文件读入程序，词法分析，此法分析结果展示和储存的功能。具体的类的完整定义见下图 2.1。

```

class WordAnalyze {
public:
    WordAnalyze(istream& file);
    void wordsdivide();//词法分析
    void output(ofstream &file); //输出词法分析结果
    void outeach(ofstream& file); //输出保留字表 ID 和sign
    map<pair<string, string>, pair<int, int>> ERRORLIST; //错误位置和情况
private:
    istream& sourceFile;          // 源程序文件输入流
    friend class GrammerAnalyze;
    map<string, string> keep;//保留字表
    vector < pair< pair< string, string >,pair<int,int> > > Words; //词法分析结果
    set< string > IDs; //ID表
    set< string > Signs;//字符表
};

```

图 2.1 词法分析类的定义

2.2.2 词法分析函数的实现

词法分析实现的主要函数是 wordsdivide() 函数，在函数内实现了对文件中读入的程序数据的分析。词法分析的流程按照图 1.1 的所示的 C-Minus 文法识别有穷自动机 DFA 流程实现。具体实现采用流水线流程，实时对于读入的字符进行判断和分类，并进行相关的判断。由于整个词法分析函数较长，在这里只会对几个典型的操作进行解释，全部的函数将会在附录给出。

首先对于此函数中使用的宏定义和变量进行解释。pair<int,int>类型的 position 将会标识此时或得到的词法符号的位置。GET 将会从文件中读取一个字符，PEEK 将会获取文件中下一个字符，但并不会读取，ADD(c)将会将字符 c 添加到读取的字符串尾部。

```

pair<int, int> position(1,1);
#define GET      ch = sourceFile.get()
#define PEEK     ch = sourceFile.peek()
#define ADD(c)   string_get.append(1, c)

```

图 2.2 词法分析函数使用的宏定义

将读入的字符进行分类，读入的是空白符，字母，数字，合法符号和非法符号时有不同的处理逻辑。使用 while 循环来不断获得符合 DFA 的字符，当读入的是空白符，字母或数字时，继续向后读取，直到读取到的字符不在符合 DFA 要求，符合条件的词将会和其属性和位置一起插入到词法分析表中。当读入的是符号时，会将符号和符号对应的属性及位置插入到词法分析表中。读取到非法符号时，则会将其位置和具体符号记录在错误表中。

数字，字母和空格的处理大体相似，在这里展示数字型的读取流程和符号类型中 <<, <=, < 的读取和错误符号的判别流程。具体操作实现见图 2.3，其中左侧的是符号读取流程，右上角的是数字判别流程，右下角的是非法字符判别流程。全部程序实现见附带程序。

```

case '<':
    string_get.clear();
    ADD(ch);
    PEEK;
    switch (ch)
    {
    case '=':
        ADD(ch);
        GET;
        PUSH(string_get, "<=");
        Signs.insert("<=");
        position.second++;
        break;
    case '<':
        ADD(ch);
        GET;
        PUSH(string_get, "<<");
        Signs.insert("<<");
        position.second++;
        break;
    default:
        PUSH(string_get, "<");
        Signs.insert("<");
        break;
    }

    if (isdigit(ch))
    {
        temp = ch - '0';
        for (GET; isdigit(ch); GET)
        {
            temp = temp * 10 + ch - '0';
            position.second++;
        }
        PUSH(to_string(temp), "NUM");
    }

    default:
        string_get.clear();
        ADD(ch);
        newitem = make_pair(string_get, "不合法字符");
        ERRORLIST.insert({ newitem, position });
        break;

```

图 2.3 词法分析函数实现（片段）

2.2.3 词法分析结果的输出

词法分析的结果将会被保存在 result 文件夹下的，以源文件名及 _Wordresule 和 _Worddivid 结尾的两个 txt 类型文件中，并且也会在控制台进行同时输出。其中，_Wordresule 中保存的是按输入顺序的词法分析结果，_Worddivid 中保存的是词法分析归类为符号表的结果。并且，当词法分析出现错误时，将会直接输出错误位置，不会产生上述两个词法分析文件。

词法分析结果的两个文件分别由 outeach(ofstream& file) 和 output(ofstream & file) 完成向文件和控制台的输出。output 函数执行的是对词法分析的顺序输出，outeach 函数执行的是对符号表输出，正确执行展示见下图 2.4 所示。其中左侧为按输

入顺序的词法分析，右侧为分类的词法分析结果。错误执行结果在 4.3 错误类型中详细说明。

字符值:#	符号类别:BEGIN	ID表;	
字符值:include	符号类别:INCLUDE	字符值:a	符号类别:ID
字符值:<	符号类别:<	符号表;	
字符值:iostream	符号类别:IOSTREAM	符号值:#	符号类别:Sign
字符值:>	符号类别:>	符号值;;	符号类别:Sign
字符值:{	符号类别:{	符号值:<	符号类别:Sign
字符值:int	符号类别:INT	符号值:>	符号类别:Sign
字符值:a	符号类别:ID	符号值{	符号类别:Sign
字符值;;	符号类别;;	符号值}	符号类别:Sign
字符值}	符号类别}	保留字表;	
字符值:#	符号类别:FINISH	符号值:char	符号类别:保留字
		符号值:do	符号类别:保留字
		符号值:else	符号类别:保留字
		符号值:for	符号类别:保留字
		符号值:if	符号类别:保留字
		符号值:include	符号类别:保留字
		符号值:int	符号类别:保留字
		符号值:iostream	符号类别:保留字
		符号值:main	符号类别:保留字
		符号值:printf	符号类别:保留字
		符号值:scanf	符号类别:保留字
		符号值:string	符号类别:保留字

图 2.4 词法分析正确结果展示

2.3 符号表管理

2.3.1 符号表存储结构

符号表的存储使用一个 STL 中的 Vector 容器实现，该队列中分别存储了具体的符号，符号种类，符号位置(行，列)信息，具体定义如下：

```
vector < pair< pair< string, string >, pair<int, int> > > Words;
```

同时，还有 ID 表，字符表和保留字表，分别存储对应的各类符号，具体定义如下：

```
map<string, string> keep; //保留字表
```

```
set< string > IDs; //ID表
```

```
set< string > Signs; //字符表
```

同时，在进行类的初始化时将会把保留字插入到 keep 中，为后续的查找提供便利。具体插入的实现为：

```
WordAnalyze::WordAnalyze(ifstream& file) : sourceFile(file) {
    keep.insert({ "int", "INT" });
```

```

keep.insert({ "while", "WHILE" });
keep.insert({ "do", "DO" });
keep.insert({ "else", "ELSE" });
keep.insert({ "if", "IF" });
keep.insert({ "then", "THEN" });
keep.insert({ "scanf", "SCANF" });
keep.insert({ "printf", "PRINTF" });
keep.insert({ "include", "INCLUDE" });
keep.insert({ "iostream", "IOSTREAM" });
keep.insert({ "for", "FOR" });
keep.insert({ "char", "CHAR" });
keep.insert({ "string", "STRING" });
keep.insert({ "main", "MAIN" });
//插入保留字
}

```

插入的保留字有 int, char, string, if, then, else, while, do, scanf, printf, include, iostream, for 和 main, 总计 14 个保留字.

2.3.2 标识符和保留字区分方法

由于保留字全部都是由字母构成, 所以在实现时只需要做到保留字和 ID 类型变量的区分。由于保留字全部储存在 map 类型的存储结构中, 所以在进行区分时, 只需要查询此时获得的字符串是否能在 map 中找到, 如果能找到就认为属性是对应的的保留字, 反之则认为是 ID 变量。具体实现见下图 2.5.

```

map<string, string>::iterator it = keep.find(string_get);
if (it == keep.end())
{
    // 不在保留字表中, 说明是标识符
    PUSH(string_get, "ID");
    IDs.insert(string_get); // 将获得的ID放在ID表里
}
else
{
    // 在保留字表中
    PUSH(it->first, it->second);
}

```

图 2.5 标识符和保留字的区分方法

3 语法分析和语义分析程序的实现

语法分析程序使用递归下降分析方法, 按照 1.1 中描述的 C-Minus 语言的 ENBF 文法进行递归分析, 并在进行语法分析的同时完成语义分析, 生成对应的四元式。

3.1 语法分析和语义分析类的定义

语法分析和语义分析类主要包含了语法分析各个非终极符的递归查询函数，并在函数内实现四元式的实时生成，并将四元式进行输出的函数。具体定义见 附录 A 语法分析程序 中的详细描述。

3.2 语法分析和语义分析函数的实现

下面将会介绍语法分析函数中每个函数具体对应的文法语句以及每个函数的具体实现框图。

表 3.2 语法分析函数名及对应的文法

函数名	对应文法
Program()	$\langle \text{程序} \rangle ::= \langle \text{头文件定义} \rangle \{ \langle \text{分程序} \rangle \}$ $\langle \text{头文件定义} \rangle ::= \#include \langle \text{iostream} \rangle \text{ int main}()$ $\langle \text{分程序} \rangle ::= \langle \text{变量定义} \rangle \langle \text{执行语句} \rangle$
Define()	$\langle \text{变量定义} \rangle ::= \langle \text{变量类型} \rangle \langle \text{标识符} \rangle ; \{ \langle \text{变量定义} \rangle \}$
Strat()	$\langle \text{执行语句} \rangle ::= \langle \text{输入语句} \rangle \langle \text{输出语句} \rangle \langle \text{赋值语句} \rangle \langle \text{条件语句} \rangle \langle \text{While 语句} \rangle \langle \text{For 语句} \rangle \{ \langle \text{执行语句} \rangle \}$
Numgiven()	$\langle \text{赋值语句} \rangle ::= \langle \text{标识符} \rangle \langle \text{赋值运算符} \rangle \langle \text{表达式} \rangle ;$
IFCHECK()	$\langle \text{条件语句} \rangle ::= \text{if}(\langle \text{条件} \rangle) \text{ then } \langle \text{执行语句} \rangle [\text{else } \langle \text{执行语句} \rangle]$
WhileCheck()	$\langle \text{While 语句} \rangle ::= \text{while}(\langle \text{条件语句} \rangle) \text{ do } \langle \text{执行语句} \rangle$
FORCHECK()	$\langle \text{For 语句} \rangle ::= \text{for}([\langle \text{赋值语句} \rangle] \langle \text{条件语句} \rangle ; \langle \text{赋值语句} \rangle) \langle \text{执行语句} \rangle$
Printout()	$\langle \text{输出语句} \rangle ::= \text{printf}(\langle \text{标识符} \rangle)$
Scanfin()	$\langle \text{输入语句} \rangle ::= \text{scanf}(\langle \text{标识符} \rangle)$
And()	$\langle \text{按位或表达式} \rangle ::= \langle \text{按位与表达式} \rangle \{ \& \langle \text{按位与表达式} \rangle \}$
OR()	$\langle \text{表达式} \rangle ::= \langle \text{按位或表达式} \rangle \{ \langle \text{按位或表达式} \rangle \}$
Relopcheck()	$\langle \text{逻辑与表达式} \rangle ::= \langle \text{表达式} \rangle \langle \text{关系运算符} \rangle \langle \text{表达式} \rangle$
Bool()	$\langle \text{条件语句} \rangle ::= \langle \text{逻辑或表达式} \rangle \{ \langle \text{逻辑或表达式} \rangle \}$
Drift()	$\langle \text{按位与表达式} \rangle ::= \langle \text{位移表达式} \rangle \{ \langle \text{位移运算符} \rangle \langle \text{位移表达式} \rangle \}$
DoubleAnd()	$\langle \text{逻辑或表达式} \rangle ::= \langle \text{逻辑与表达式} \rangle \{ \&\& \langle \text{逻辑与表达式} \rangle \}$
ADDCheck()	$\langle \text{赋值语句} \rangle ::= \langle \text{标识符} \rangle \langle \text{赋值运算符} \rangle \langle \text{表达式} \rangle ;$
MulCheck()	$\langle \text{乘除表达式} \rangle ::= \langle \text{运算式} \rangle \{ \langle \text{乘除运算符} \rangle \langle \text{运算式} \rangle \}$
Equal()	$\langle \text{运算式} \rangle ::= (\langle \text{运算式} \rangle) \langle \text{标识符} \rangle \langle \text{整数} \rangle$

下面将会依次介绍每个函数的具体实现表达式语句的逻辑框图，并且会给部分函数的框架，全部函数实现的代码见附带程序。

3.2.1 void Program() 函数

Program() 函数没有返回值，具体实现了<程序> ::= <头文件定义> {<分程序>}
<头文件定义> ::= #include<iostream> int main(), <分程序> ::= <变量定义> <执行语句>这几条语句的功能，全部运行完之后整个函数及处理完毕。处理的逻辑框图和函数框架如下图 3.2。

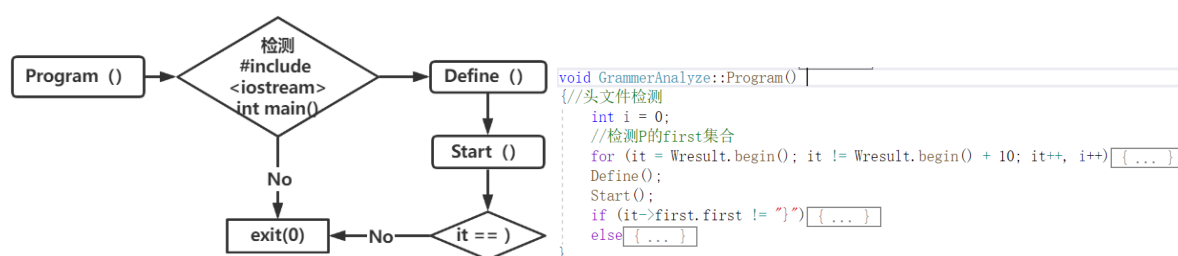


图 3.2 Program() 函数的逻辑框图与框架

3.2.2 void Define() 函数

Define() 函数没有返回值，具体实现了<变量定义> ::= <变量类型> <标识符>; {<变量定义>}这几条语句的功能，全部运行结束之后将会完成对于整个函数中所有变量的定义，函数的处理的逻辑框图和函数框架如下图 3.3。

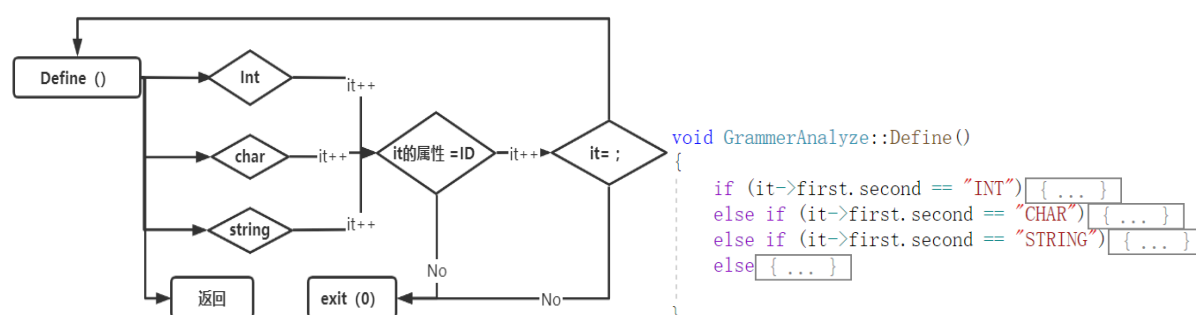


图 3.3 Define() 函数的逻辑框图和框架

3.2.3 void Start() 函数

Start() 函数没有返回值，具体实现了<执行语句> ::= <输入语句> | <输出语句> | <赋值语句> | <条件语句> | <While 语句> | <For 语句> | {<执行语句>}这条语句的功能，全部运

行结束之后将会完成整个程序相关执行指令，且会进行相关的递归调用。函数的处理的逻辑框图和函数框架如下图 3.4。

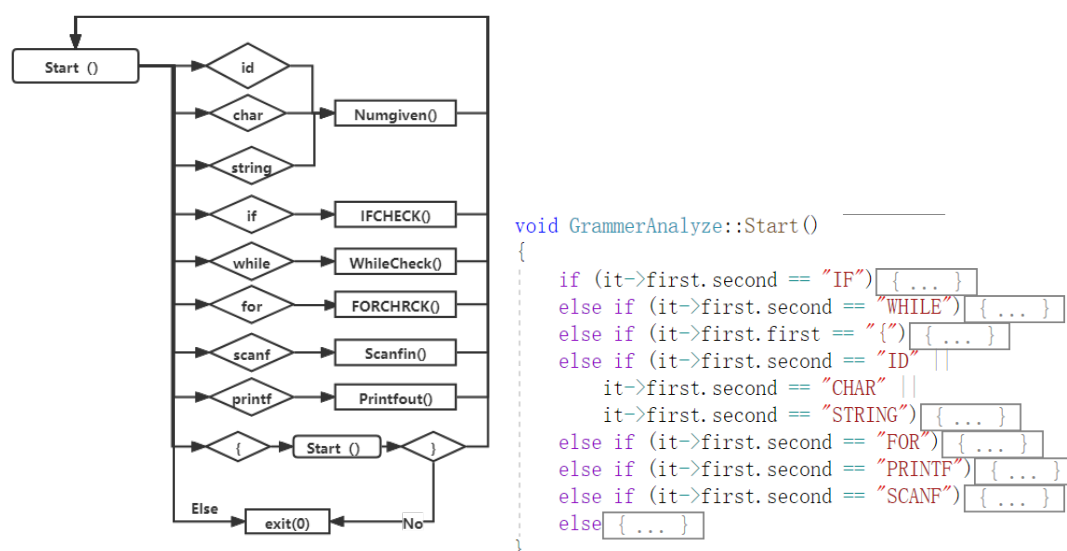


图 3.4 Start() 函数的逻辑框图和框架

3.2.4 void Numgiven() 函数

Numgiven() 函数没有返回值，具体实现了 <赋值语句>::=<标识符> <赋值运算符> <表达式>；这条语句的功能。全部运行完成之后将会形成一条赋值语句，具体的函数的处理的逻辑框图和函数框架如下图 3.5。

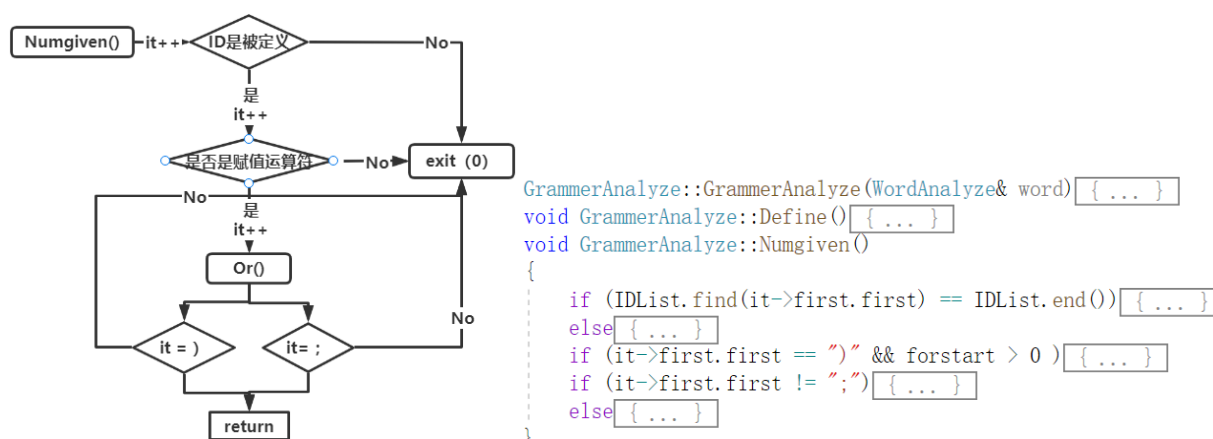


图 3.5 Numgiven() 函数框图和框架

3.2.5 void IFCHECK() 函数

IFCHECK() 函数没有返回值，具体实现了<条件语句>::= if(<条件>) then <执行语句> [else <执行语句>]这条语句的功能。全部运行完成之后将会形成一个条件语句块，具体的函数的处理的逻辑框图如下图 3.6。

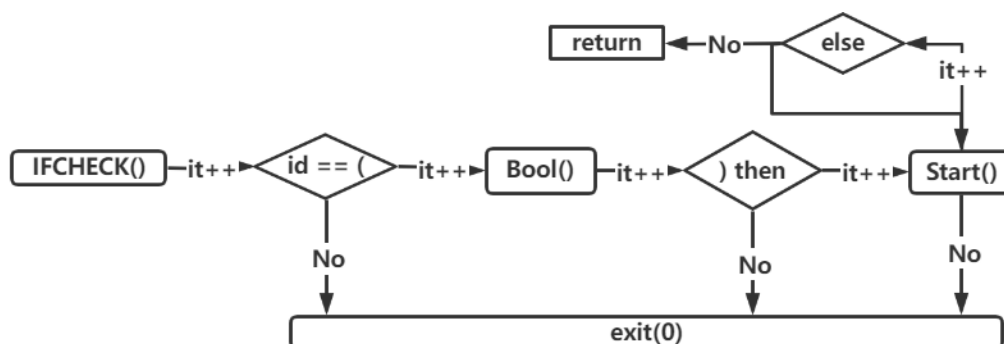


图 3.6 IFCHECK() 函数逻辑框图

3.2.6 void WhileCheck() 函数

WhileCheck() 函数没有返回值，具体实现了<While 语句>::= while(<条件语句>) do<执行语句>这条语句的功能。全部运行完成之后将会形成一个条件语句块，具体的函数的处理的逻辑框图如下图 3.7。

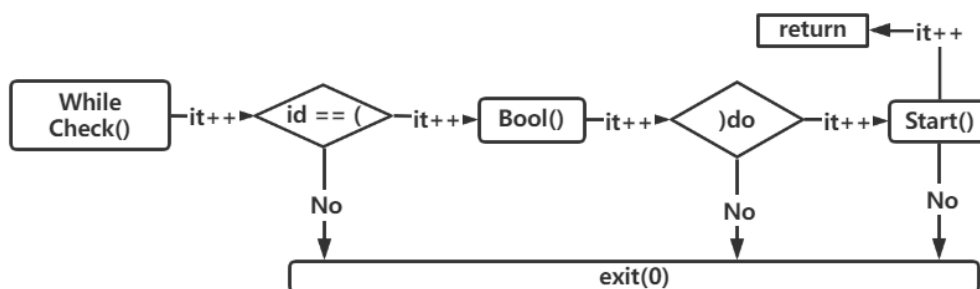


图 3.7 WhileCheck() 函数逻辑框图

3.2.7 void FORCHECK() 函数

FORCHECK() 函数没有返回值，具体实现了<For 语句>::= for([<赋值语句>] <条件语句> ; <赋值语句>) <执行语句>这条语句的功能。全部运行完成之后将会形成一个条件语句块，具体的函数的处理的逻辑框图如下图 3.8。

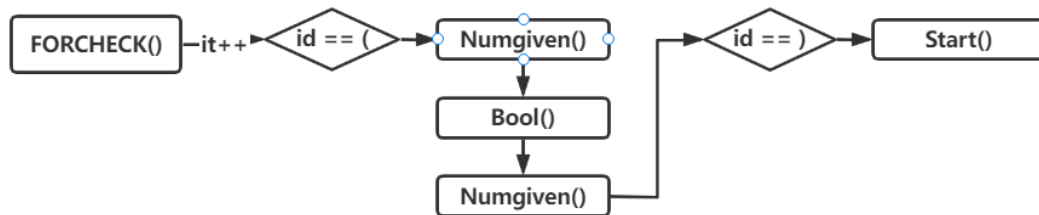


图 3.8 FORCHECK() 函数的逻辑框图

3.2.8 void Printout() 函数和 void Scanfin() 函数

Scanfin() 函数和 Printout() 函数没有返回值，且两个函数之间逻辑结构相似，组成成分一致，所以在这里共同展示。他们具体实现了<输出语句> ::= printf(<标识符>) 和<输入语句> ::= scanf(<标识符>) 语句所指示的输入输出功能。下面将对 Scanfin() 函数进行展示。Scanfin() 具体的逻辑框图和函数框架见下图 3.9

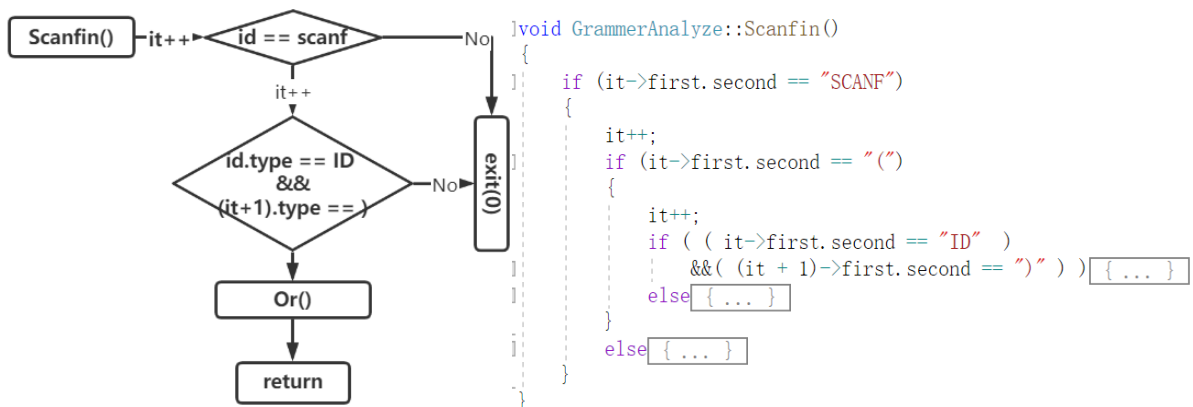


图 3.9 Sanfin() 函数的逻辑框图和函数框架

3.2.9 string Equal() 函数

Equal() 函数会返回字符串类型声明的临时表达式。函数实现了<运算式> ::= (<运算式>) | <标识符> | <整数> 语句所指示的功能。是整个文法达到 ID 和 MUN 类型终极符的唯一语句，具体的逻辑框图和函数框架见下图 3.10

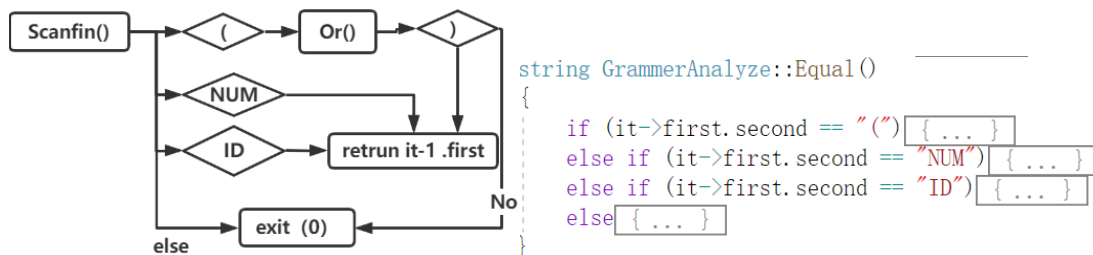


图 3.10 Equal() 函数的逻辑框图和函数框架

3.2.10 余下的判断函数

余下的判断函数均为string类型，分别有AND()、OR()、Bool()、Drift()、DoubleAnd()、Relopcheck()和ADDCheck()总计7个函数，分别实现了

〈按位或表达式〉::= 〈按位与表达式〉 { & 〈按位与表达式〉}

〈表达式〉::= 〈按位或表达式〉 { | 〈按位或表达式〉}

〈条件语句〉::= 〈逻辑或表达式〉 { || 〈逻辑或表达式〉 }

〈按位与表达式〉::= 〈位移表达式〉 { 〈位移运算符〉 〈位移表达式〉}

〈逻辑或表达式〉::= 〈逻辑与表达式〉 { && 〈逻辑与表达式〉 }

〈乘除表达式〉::= 〈运算式〉{〈乘除运算符〉 〈运算式〉}

〈逻辑与表达式〉::= 〈表达式〉〈关系运算符〉〈表达式〉

总计七条相关语句的实现。每次均会在被访问的语句之后搜索中间的终极符，在满足条件之后，会继续向后搜索访问文法访问的函数。直到无法访问到相关终结符后进行返回。下面将以string Relopcheck()作为示例，接下来将会展示Relopcheck()函数的逻辑框图和框架，见下图4.11. 其余函数的逻辑与其十分相似。

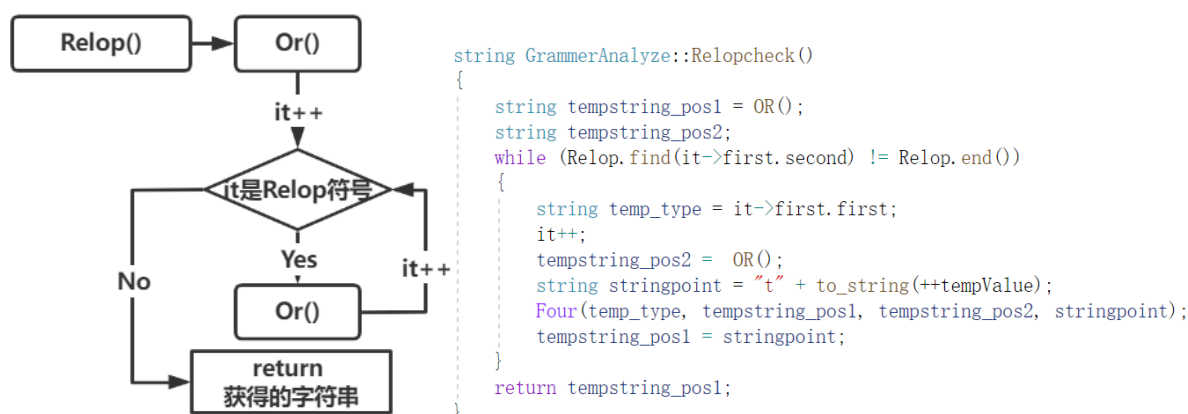


图 3.11 Relop() 函数的逻辑框图和函数实现

3.3 中间代码的定义

3.3.1 中间代码-四元式

中间代码的形式是四元式，四元式储存类型是 Foursentence 的结构体中，Foursentence 结构体定义如下。

```

struct Foursentence
{
    string Type;
    string Number_1;
    string Number_2;
    string Name;
}

```



```
int Posi;
};
```

其中 Type 标识了四元式的类型, Number_1 和 Number_2 分别是操作数 1 和操作数 2, Name 则用来标识临时表达式的标号, Posi 则记录了四元式的行号。具体的四元式组成为 (Type, Number_1, Number_2, Name), 下面将会简略介绍不同四元式的解释。全部四元式的详细解释将在附录 B 中给出。

表 3.1 四元式的解释

四元式	产生状态
(INT , ID , _ , _)	声明一个整型变量 ID
(CHAR , ID , _ , _)	声明一个字符型变量 ID
(STR , ID , _ , _)	声明一个字符串型变量 ID
(IN , ID , _ , _)	输入 ID 的值
(OUT , ID , _ , _)	输出 ID 的值
(<双目运算符>, Number1, Number2, temp)	将 Number1 和 Number2 执行<双目运算符>的运算结果储存在临时语句 temp 中 (包括 &, &&, , , ,, +, -, *, /<<, >> 运算符)
(= , Number1 , Number2 , _)	将 Number2 的值赋给 Number1
(Relop, Number1, Number2, temp)	判断 Number1 Relop Number2 的真假, 并将对应结果保留在 temp 中 (Relop 包含 >, >=, <, <=, ==, !=)
(JNZ , temp , _ , Posi)	当 temp 表达式为真的时候, 跳转到 Posi 位置
(JEZ , temp , _ , Posi)	当 temp 表达式为假的时候, 跳转到 Posi 位置
(J , _ , _ , Posi)	无条件跳转到 Posi 位置
(FINISH , _ , _ , _)	结束程序

3.3.2 拉链回填的实现

由于布尔表达式在运算产生四元式时, 需要进行跳转的位置暂时无法确定, 这时需要使用拉链回填的方式将后面获得的布尔表达式的真出口和假出口重新填回到布尔表达式对应位置。下面将会展示 if-then-else, while-do 语句和 for 语句的拉链回填实现方式。具体实现逻辑见下表 3.2 至 3.4. 实现相关代码的展示见附件 A

表 3.2 while-do 语句的拉链回填实现

while <condition> do <statement>	
Line1	<condition>
Line2	(JNZ , Number1, , Line4)

Line3	(JEZ , Number1, , Line6)
Line4	<statement>
Line5	(J , , , Line1)
Line6	

表 3.3 if-then-else 语句的拉链回填实现

if <condition> then <statement 1> else <statement 2>	
Line1	<condition>
Line2	(JNZ , Number1, , Line4)
Line3	(JEZ , Number1, , Line6)
Line4	<statement 1>
Line5	(J , , , Line7)
Line6	<statement 2>
Line7	

表 3.3 for 语句的拉链回填实现

for (<statement 1> <condition> <statement 2>) <statement 3>	
Line1	<statement 1>
Line2	<condition>
Line3	(JNZ , Number1, , Line7)
Line4	(JEZ , Number1, , Line9)
Line5	<statement2>
Line6	(J , , , Line2)
Line7	<statement3>
Line8	(J , , , Line5)
Line9	

上述表格展示了 while-do, if-then-else 和 for 语句拉链回填实现的逻辑方法。在具体实现中,使用了栈作为储存结构来储存需要被记录的地址,每次回填时则访问栈顶并弹出,被弹出的便是此时需要回填的地址。以这种方式实现了循环语句和条件语句的嵌套使用的四元式输出。具体位置保存在栈 stack <string> outpos 中。

3.4 文法特色与难点

本文法的特色便是在实验要求的基础上额外实现了位运算指令,包括了按位与(&),按位或(|),左移(<<),右移(>>)四种运算。实现的函数有And(),Or()和Drift()函

数，特色也包含了实现的解释程序，解释程序的具体描述则在第4点中描述，在这里不做详细描述。文法的难点实现在拉链会回填的地址选择和重填，在3.3中已经做出了详细的描述，在这里不再描述。

4 解释程序的实现

解释程序是由我编写的，针对不同的中间代码进行翻译和执行并进行展示的函数（`Explain.cpp` 和 `Explain.h`）。实现了顺序执行和调试执行两个大分类的功能，调试执行又分为从头调试和选择调试两种。在词法分析，语法和语义分析正确执行结束之后，将会自动执行解释程序，并产生如下图 4.1 所示的选择语句，需要用户做出选择，在获得合法输入之后会为用户执行对应的程序，反之则会提醒用户，在用户输入正确后才会执行。（错误输入样例见下图 4.2）




图 4.1 解释程序执行展示

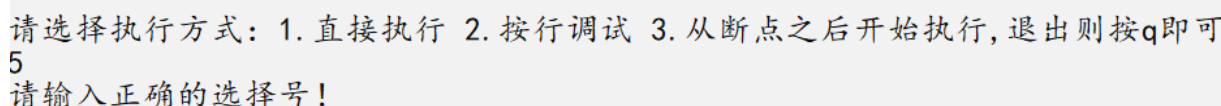


图 4.2 解释程序输入错误提示

4.1 解释程序函数的实现

解释程序采用的图灵机的方法，由于四元式储存在一个 `vector` 类型变量中，从头至尾顺序读取当前四元式，并根据四元式，执行对应的操作，直到四元式被全部读取完毕即完成了解释程序的功能。

4.2 顺序执行预期结果

顺序执行即按顺序实现四元式的翻译，并且对于程序中的输入输出语句提供对应的输入输出展示。具体的顺序执行结果可以查看 5.1 正确测试样例中的展示。

4.3 调试执行预期结果

调试执行和输出执行类似，但是调试执行提供了两个功能的选择，按行调试和从断点之后开始调试。其中，从断点开始调试需要输入断点所指代的四元式行号才能执行。调试执行会额外输出此时正在执行的四元式和该四元式所指代的语句，以及所有临时变量的值。具体调试执行结果见下图 4.3。四元式在调试执行时的翻译表见附录 B

```
当前执行语句: 184 (JEZ ,t39 ,_ ,191 )    此四元式代指的C语句为: if(t39 ==false) goto191
各表达式执行后结果为
a  5
b  2
b2b 0
i  0
j  3
c  s
```

图 3.3 调试执行输出示例

4.4 错误类型

错误类型可以大体上可以分为语法分析错误，词法分析错误和执行错误，分别对应了错误被发现的位置。词法分析的错误会在执行完词法分析后弹出，语法分析和执行错误则会在被发现的时刻立即弹出并结束程序执行。整个程序可以发现 16 类，30 种不同错误，每种错误均会提示发现的位置的行、列号和对应的错误描述。大体的错误分类在下表 4.1 中给出，具体每个的错误运行结果展示将会在测试用例和附录 C 中进行展示。

表 4.1 错误表

出错编号	错误原因
1	未找到输入的文件
2	输入了词法允许以外的符号
3	没有函数必须的开始符号
4	没有定义变量
5	没有执行语句
6	语句缺少;
7	语句缺少匹配的)或}
8	语句缺少(或{
9	if 语句缺少 then
10	while 语句缺少 do
11	sacnf, printf 执行了多个元素输入输出
12	对于双目运算符只提供了一个操作数
13	使用的 ID 未被定义
14	同一个 ID 被重定义
15	在错误的位置定义变量
16	除法表达式除 0

5 测试用例

5.1 测试用例 1 正确样例

下面将展示正确的程序和执行结果包括了_Wordresult 和_Worddivid 中的词法分析和_Grammerresult 中的四元式结果，图 5.1 是结果的部分展示，全部展示见附件 C

test.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
#include <iostream>
int main()
{
    int a; int b2b; int i; int j; int b;
    char c;
    string str;
    scanf(a)
    scanf(c)
    scanf(str)
    printf(a)
    printf(c)
    printf(str)
    for(i = -3 ; i<0 ; i +=1)
    {
        j = 1;
        for(; j <= 1 && j != 2 ; j += 2)
        {
            scanf(b)
            while(b2b != 0 && a != 0)
            do{
                a = (a/a) - (a+a) * (a-a) - a;
                b2b= b2b << 1;
            }
        }
    }
}
```

test.txt_Wordresule.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
字符值:# 符号类别:BEGIN
字符值:include符号类别:INCLUDE
字符值:< 符号类别:<
字符值:iostream符号类别:IOSTREAM
字符值:> 符号类别:>
字符值:int 符号类别:INT
字符值:main 符号类别:MAIN
字符值:( 符号类别:(
字符值:) 符号类别:)
字符值:{ 符号类别:{
字符值:int 符号类别:INT
字符值:a 符号类别:ID
字符值;; 符号类别;;
字符值:int 符号类别:INT
字符值:b2b 符号类别:ID
```

ID表:

字符值:a	符号类别:ID
字符值:b	符号类别:ID
字符值:b2b	符号类别:ID
字符值:c	符号类别:ID
字符值:i	符号类别:ID
字符值:j	符号类别:ID
字符值:str	符号类别:ID

符号表:

符号值:!=	符号类别:Sign
符号值:#	符号类别:Sign
符号值:&	符号类别:Sign
符号值:&&	符号类别:Sign
符号值:(符号类别:Sign
符号值:)	符号类别:Sign
符号值>*	符号类别:Sign
符号值:*=	符号类别:Sign
符号值:+	符号类别:Sign
符号值:+=	符号类别:Sign
符号值:-	符号类别:Sign
符号值:--	符号类别:Sign

保留字表:

符号值:char	符号类别:保留字
符号值:do	符号类别:保留字
符号值:else	符号类别:保留字
符号值:for	符号类别:保留字
符号值:if	符号类别:保留字
符号值:include	符号类别:保留字
符号值:int	符号类别:保留字
符号值:iostream	符号类别:保留字
符号值:main	符号类别:保留字
符号值:printf	符号类别:保留字
符号值:scanf	符号类别:保留字
符号值:string	符号类别:保留字
符号值:then	符号类别:保留字
符号值:while	符号类别:保留字

test.txt_Grammerresult.txt

文件(F) 编辑(E) 格式(O) 查看(V)

```
125 (JNZ ,t6 ,130)
126 (JEZ ,t6 ,151)
100 (INT ,a ,127)
101 (INT ,b2b ,128)
102 (INT ,i ,129)
103 (INT ,j ,130)
104 (INT ,b ,131)
105 (CHAR ,c ,132)
106 (STR ,str ,133)
107 (IN ,a ,134)
108 (IN ,c ,135)
109 (IN ,str ,136)
110 (OUT ,a ,137)
111 (OUT ,c ,138)
112 (OUT ,str ,139)
113 (= ,i ,140)
114 (< ,i ,141)
115 (JNZ ,t1 ,142)
116 (JEZ ,t1 ,143)
117 (+ ,i ,144)
118 (= ,i ,145)
119 (J ,146)
120 (= ,j ,147)
121 (<= ,j ,148)
122 (!= ,j ,149)
123 (& ,t3 ,t4 ,t5)
124 (!= ,t5 ,0 ,t6)
```

```
请选择执行方式: 1. 直接执行 2. 按行调试 3. 从断点之后开始执行, 退出则按q即可
1
请输入变量a:1
请输入变量c:sss
请输入变量str:sss

a的值为1
c的值为s

str的值为sss
请输入变量b:1

a的值为1
b2b的值为0
请输入变量b:0

a的值为1
b2b的值为0
```

图 5.1 正确执行的词法、语法和解释执行输出结果

5.2 测试用例 2（词法分析错误）

下面将展示词法分析错误的程序和执行结果，见图 5.2

```
2.txt - 记事本
文件(F) 编辑(E) 格式(O) 查:
#include <iostream>
int main()
{
%
}
```

请输入测试文件名称
2.txt
不合法字符 % 位置: 4行1列
词法分析有错误, 请检查代码对应位置

图 5.2 词法分析错误执行

5.3 测试用例 3（语法分析错误）

下面将展示词法分析错误的程序和执行结果，见图 5.3.具体展示的是词法分析错误类别 4，其余的错误执行的程序和结果将会在附录 D 中给出。

```
4.txt - 记事本
文件(F) 编辑(E) 格式(O) 查:
#include<iostream>
int main()
{

}

Please Check where ID was define 错误位置5行1列
```

图 5.3 语法分析错误执行

6 感想

通过这次编译原理课程设计，我的感触颇深。从零开始按照自己学过的编译原理知识实现一个特定语言的编译器的制作。通过制作编译器的流程，我对编译器执行的流程有了一些了解，对上课讲过的词法分析 DFA 实现、在语法分析时如何将已有的文法改造成满足递归子程序的文法的方法（消除文法的二义性，进一步进行文法变换，改造文法为 LL(1) 文法，如消除左递归、提取左公共因子，角替换等方法）、自顶向下语法分析、递归子程序实现、语法制导翻译、目标四元式的翻译执行和临时变量的存储分配等有了较深刻的体会。

本次实验实现的 C-Minus 语言编译器使用 C++ 语言编写，编译环境为 VS2019，由于本次实验是由我自己一个人自己实现的，没有任何参考。所以在一开始的做的时候也遇到了一些困难，像在一开始实现词法分析时没有做好整体规划，词法分析结果的储存结构从 map 到 vector 到 `vector<pair, pair>`，一点一点的添加。增加了自己很多重复的工作，而这些工作本来是可以开始便被解决的。这也提醒了我在进行程序设计的时候应该做好整体设计，自顶向下的提前做好每一个模块的大体设计。这样从总体上设计的方法才能更好的解决问题，也可以让我整体的对于项目每个模块实现的状态和预期结果有了一个完整的掌握。这也是我从本次实验中学到的。

本次实验也有一些小小的遗憾，比如没能完成整体的从中间代码四元式到 x86 汇编语言的转化，只能使自己生成的中间代码在自己写好的解释程序上进行解释执行和调试，没能做到在所有程序上进行执行的展示，没有能够为整体的执行程序写出一套配套的 UI 出来，没有能够使得语法分析程序可以一次性找出全部的与语法错误。欠缺的时间使这些一切本可能实现的功能没有能够最终展现出来。我也会在后续时间充足的时候将这些功能附上。

本次实验中，总计 2500 多行代码是我上大学至今 100% 手写的最长的一份工程文件，尽管他仍有不足，但是仍让我收获良多。同样，在本次实验中自己亲手实践的编译原理程序也让我对于编译原理这样一门计算机偏向底层交互的重要课程理解更加深刻，希望在后续的课程中，如果可能的话可以适当增加编译原理实验的学分，这样时间充分的情况下，我相信会有更好的结果实现。

附录 A 词法分析程序

1 词法分析类的声明

```

class GrammerAnalyze
{
public:
    GrammerAnalyze(WordAnalyze&);
    bool grammeranalyze(); //语法分析函数
    void outputfour(ofstream& file);
private:
    WordAnalyze& wordanalyze; //语法分析要使用词法分析
    vector < pair< pair< string, string >, pair<int, int> > >
        ::iterator it = wordanalyze.Words.begin();
    void Program(); //检测开头文件
    void Define();
    void Start();
    void Numgiven();
    void IFCHECK();
    void WhileCheck();
    void FORCHECK();
    void Printout();
    void Scanfin();
    string Check();
    string AND();
    string OR();
    string Bool();
    string Weiyi();
    string DoubleAnd();
    string Relopcheck();
    string ADDCheck();
    string MulCheck();
    string Equal();
    set<string> Relop;
    set<string> DEQU;
    map<string, string> IDList; //声明的变量 <名称 <, 值>>
    bool jumpsymbol = false; //使用此标记表明是否需要跳转
    struct Foursentence
    {
        string Type;
        string Number_1;
        string Number_2;
        string Name;
        int Posi;
    };
    friend class Explain;
public:
    vector < Foursentence > FourLine; //类型, 运算符1, 运算符2 代称
    vector < Foursentence > ::iterator start = FourLine.begin();
};

```

图 1 词法分析类的定义

2 四元式详细解释

表 1 全部四元式及对应解释

四元式	翻译结果
(INT , ID , _ , _)	声明一个整型变量 ID
(CHAR , ID , _ , _)	声明一个字符型变量 ID
(STR , ID , _ , _)	声明一个字符串型变量 ID
(IN , ID , _ , _)	输入 ID 的值
(OUT , ID , _ , _)	输出 ID 的值
(&, Number1, Number2, temp)	将 Number1 和 Number2 按位与, 并将运算结果保存在 temp 中
(&&, Number1, Number2, temp)	将 Number1 和 Number2 逻辑, 并将运算结果保存在 temp 中
(, Number1, Number2, temp)	将 Number1 和 Number2 按位或, 并将运算结果保存在 temp 中
(, Number1, Number2, temp)	将 Number1 和 Number2 逻辑或, 并将运算结果保存在 temp 中
(+, Number1, Number2, temp)	将 Number1 和 Number2 加和, 并将运算结果保存在 temp 中
(-, Number1, Number2, temp)	将 Number1 减去 Number2, 并将运算结果保存在 temp 中
(*, Number1, Number2, temp)	将 Number1 乘上 Number2, 并将运算结果保存在 temp 中
(/, Number1, Number2, temp)	将 Number1 除以 Number2, 并将运算结果保存在 temp 中
(<<, Number1, Number2, temp)	将 Number1 逻辑左移 Number2 位, 并将结果保存在 temp 中
(>>, Number1, Number2 , temp)	将 Number1 逻辑右移 Number2 位, 并将结果保存在 temp 中
(= , Number1 , Number2 , _)	将 Number2 的值赋给 Number1
(==, Number1, Number2, temp)	判断 Number1 的值是否等于 Number2, 将判断结果储存在 temp 中, 真时 temp 值为 1, 反之 temp 的值 0
(>=, Number1, Number2, temp)	判断 Number1 的值是否大于等于 Number2, 将判断结果储存在 temp 中, 真时 temp 值为 1, 反之 temp 的值 0
(<=, Number1, Number2, temp)	判断 Number1 的值是否小于等于 Number2, 将判断结果储存在 temp 中, 真时 temp 值为 1, 反之 temp 的值 0
(<, Number1, Number2, temp)	判断 Number1 的值是否小于 Number2, 将判断结果储存在 temp 中, 真时 temp 值为 1, 反之 temp 的值 0

表 1 全部四元式及对应解释（续）

(>, Number1, Number2, temp)	判断 Number1 的值是否大于 Number2, 将判断结果储存在 temp 中, 真时 temp 值为 1, 反之 temp 的值 0
(!=, Number1, Number2, temp)	判断 Number1 的值是否不等于 Number2, 将判断结果储存在 temp 中, 真时 temp 值为 1, 反之 temp 的值 0
(JNZ, temp, _, Posi)	当 temp 表达式为真的时候, 跳转到 Posi 位置
(JEZ, temp, _, Posi)	当 temp 表达式为假的时候, 跳转到 Posi 位置
(J, _, _, Posi)	无条件跳转到 Posi 位置
(FINISH, _, _, _)	结束程序

3 四元式拉链回填实现

布尔表达式的翻译是由一组条件真跳和无条件跳代码组成。待填地址采用回填、拉链方法实现。本次实现的布尔表达式拉链回填是对整个表达式进行的, 并没有针对或语句完成每条判否的跳出, 特此说明。在下面将会描述 while-do 语句, if-then-else 语句和 for 语句的拉链回填实现核心逻辑, 并结合四元式进行解析。

同时, 在插入四元式时使用的宏定义 Four 和 FouTemp 的具体描述如下图 2

```
#define Four(s1, s2, s3, s4) Foursentence line;\
line.Type = s1;\
line.Number_1 = s2;\
line.Number_2 = s3;\
line.Name = s4;\
line.Posi = Line++;\
FourLine.push_back(line);

#define FouTemp(s1, s2, s3, s4) Foursentence line_temp;\
line_temp.Type = s1;\
line_temp.Number_1 = s2;\
line_temp.Number_2 = s3;\
line_temp.Name = s4;\
line_temp.Posi = Line++;\
FourLine.push_back(line_temp);
```

图 2 Four 和 FouTemp 的声明

3.1 While-do 函数拉链回填实现和展示

下面将会介绍 while-do 拉链回填的核心代码和结果展示, 代码展示见下方, 核心结果展示见图 3 所示。核心代码为:

```
//通过while-do头判断, 开始生成四元式
int jumpback = Line;//jumpback记录返回时跳转的位置, 后续无条件跳转使用
```

```

outposi.push(to_string(jumpback)); //加入地址栈，为递归调用使用
string temp_while = Bool(); //布尔表达式语句生成
int trueout, falseout;
Four("JNZ", temp_while, "_", ""); //真出口语句生成，此时跳转位置待填
trueout = Line;
FoueTemp("JEZ", temp_while, "_", ""); //假出口语句生成，此时跳转位置待填
falseout = Line;
//继续判断while-do语句do部分组成
FourLine[trueout - 101].Name = to_string(Line); //判断条件为真时会跳此位置执行，将行号填入真地址跳转位置

//while-do语句的执行语句
Foursentence Jtemp; //生成一个四元式，用于实现无条件跳转语句
Jtemp.Type = "J";
Jtemp.Number_1 = Jtemp.Number_2 = "_";
Jtemp.Name = outposi.top(); //无条件跳转的位置从位置栈顶获得
outposi.pop();
Jtemp.Pos = Line++;
FourLine.push_back(Jtemp); //将无条件跳转语句插入四元式
FourLine[falseout - 101].Name = to_string(Line); //while-do语句完成，此时的位置就是错误时的跳转出口

```

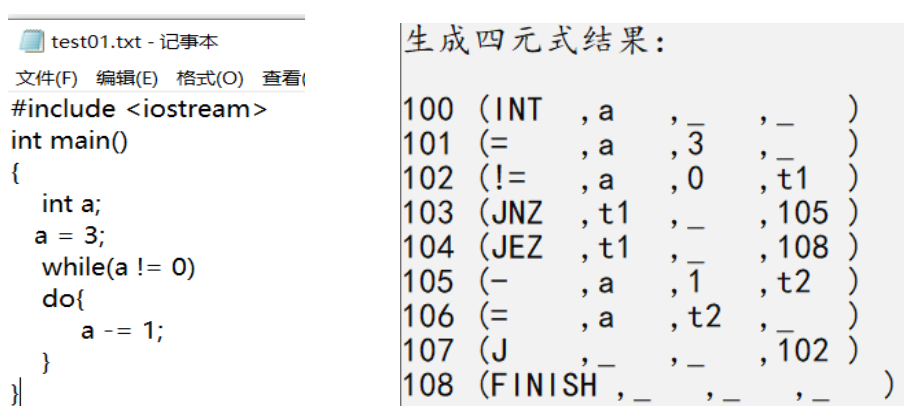


图 3 while-do 语句结果展示

上图中实现的便是简单的一个while-do循环语句，其中103和104四元式分别指向do语句执行位置和结束位置，107条语句的为无条件跳转语句，跳转地址为102，即为布尔表达式执行位置。

3.2 if-then-else 函数拉链回填实现核心

下面将会介绍 if-then-else 拉链回填的核心代码和结果展示，代码展示见下方，核心结果展示见图 4 所示。核心代码为：

```

//通过if-then-else头判断，开始生成四元式
string temp_if = Bool(); //布尔表达式语句生成
int trueout, falseout;
Four("JNZ", temp_if, "_", ""); //真出口语句生成，此时跳转位置待填
trueout = Line;
FoueTemp("JEZ", temp_if, "_", ""); //假出口语句生成，此时跳转位置待填
falseout = Line;
//继续判断if-then-else语句then部分组成
FourLine[trueout - 101].Name = to_string(Line); //then语句执行的位置就是
//真出口，填入到真出口的四元式对应位置

//then语句的执行语句
Foursentence Jtemp; //生成一个四元式，用于实现无条件跳转语句
Jtemp.Type = "J";
Jtemp.Number_1 = Jtemp.Number_2 = "_";
Jtemp.Name = ""; //此时不知道无条件跳转到哪里，暂时填入空
Jtemp.Posi = Line++;
int jump_posi = Line;
FourLine.push_back(Jtemp);
//else语句判断组成
FourLine[falseout - 101].Name = to_string(Line); //else语句的执行位置就是
//假出口，找到假出口四元式的对应位置

//else执行语句
FourLine[jump_posi - 101].Name = to_string(Line); //else语句执行结束的位置就是函数执行结束then语句之后需要跳
//转到的位置，将此位置填入无条件跳转语句

```

<pre> test01.txt - 记事本 文件(F) 编辑(E) 格式(O) 查看(V) #include <iostream> int main() { int a; a = 3; if(a == 3) then a = 4; else a = 3; } </pre>	<p>生成四元式结果：</p> <pre> 100 (INT , a , _ , _) 101 (= , a , 3 , _) 102 (== , a , 3 , t1) 103 (JNZ , t1 , _ , 105) 104 (JEZ , t1 , _ , 107) 105 (= , a , 4 , _) 106 (J , _ , _ , 108) 107 (= , a , 3 , _) 108 (FINISH , _ , _ , _) </pre>
---	--

图 4 if-then-else 语句结果展示

上图中实现的便是简单的一个if-then-else循环语句，其中103和104四元式分别指向then语句执行和else语句执行位置，在语句判断完之后获得。105语句即为无条件语句，跳转地址为108，即为else语句执行结束之后的位置。

3.3 for 函数拉链回填实现核心

下面将会介绍 if-then-else 拉链回填的核心代码和结果展示，代码展示见下方，核心结果展示见图 5 所示。核心代码为：

```
//通过for语句头判断，开始生成四元式
Numgiven(); //执行for语句组成的第一条语句 变量声明
string jumpback = to_string(Line); //记录此时位置，方便后续无条件跳转使用
    outposi.push(jumpback); //压入栈中，实现for语句的执行顺序和循环嵌套
string temp_if = Bool(); //布尔表达式语句执行
int trueout, falseout;
Four("JNZ", temp_if, "_", ""); //真出口语句生成，此时跳转位置待填
trueout = Line;
FoueTemp("JEZ", temp_if, "_", ""); //假出口语句生成，此时跳转位置待填
falseout = Line;
//for语句构成成分判断 ";"
string numback = to_string(Line); //记录此时位置，方便后续无条件跳转使用
Numgiven(); //后续的也是一个变量声明语句
Foursentence Jtemp; //完成变量定义之后也是一个无条件跳转
Jtemp.Type = "J";
Jtemp.Number_1 = Jtemp.Number_2 = "_";
Jtemp.Name = outposi.top(); //此时的跳转语句跳转到for语句判断
outposi.pop();
Jtemp.Posi = Line++;
FourLine.push_back(Jtemp);
outposi.push(numback); //此时将变量赋值之前的语句压入栈
FourLine[trueout - 101].Name = to_string(Line); //之后的位置将开始for语句
                                                    的执行，这里的位置就是真出口

//for执行语句
Foursentence Jtemp;
Jtemp.Type = "J";
Jtemp.Number_1 = Jtemp.Number_2 = "_";
Jtemp.Name = outposi.top(); //这里的无条件跳转在for语句执行完成之后，跳转
                                                    到最后的变量赋值位置，如i++的执行位置

outposi.pop();
Jtemp.Posi = Line++;
FourLine.push_back(Jtemp);
FourLine[falseout - 101].Name = to_string(Line); //在所有语句执行结束之后，
                                                    我们就获得了假出口的位置
```

test01.txt - 记事本		生成四元式结果:	
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)		100	(INT , a , _ , _)
#include <iostream>		101	(= , a , 3 , _)
int main()		102	(!= , a , 9 , t1)
{		103	(JNZ , t1 , _ , 108)
int a;		104	(JEZ , t1 , _ , 111)
for(a = 3 ; a !=9 ; a+=2)		105	(+ , a , 2 , t2)
a -= 1 ;		106	(= , a , t2 , _)
}		107	(J , _ , _ , 102)
		108	(- , a , 1 , t3)
		109	(= , a , t3 , _)
		110	(J , _ , _ , 105)
		111	(FINISH , _ , _ , _)

图 5 for 语句执行结果展示

上图中实现的便是简单的一个for循环语句，由于for循环的构成特点和执行顺序，总计有两条有条件跳转语句和两条无条件跳转语句组成其中103和104四元式分别指向for语句执行语句和for语句结束后的语句。107语句为无条件跳转语句，指向的位置为102，即为布尔表达式位置，110语句为无条件跳转语句，指向的位置为105，即为for语句最后的表达式（i++等语句）执行位置。

以上所有语句的顺序执行等完全按照正常的C语言执行顺序即可。

附录 B 解释程序

1 四元式的解释翻译结果

表 1 四元式解释执行翻译表

四元式	翻译结果
(INT , ID , _ , _)	int ID;
(CHAR , ID , _ , _)	char ID;
(STR , ID , _ , _)	string ID;
(IN , ID , _ , _)	scanf(ID)
(OUT , ID , _ , _)	printf(ID)
(&, Number1, Number2, temp)	temp = Number1 & Number2
(&&, Number1, Number2, temp)	temp = Number1 && Number2
(, Number1, Number2, temp)	temp = Number1 Number2
(, Number1, Number2, temp)	temp = Number1 Number2
(+, Number1, Number2, temp)	temp = Number1 + Number2
(-, Number1, Number2, temp)	temp = Number1 - Number2
(*, Number1, Number2, temp)	temp = Number1 * Number2
(/, Number1, Number2, temp)	temp = Number1 / Number2
(<<, Number1, Number2, temp)	temp = Number1 << Number2
(>>, Number1, Number2, temp)	temp = Number1 >> Number2
(= , Number1 , Number2 , _)	Number1 = Number2
(==, Number1, Number2, temp)	temp = Number1 == Number2 ? 1:0
(>=, Number1, Number2, temp)	temp = Number1 >= Number2 ? 1:0
(<=, Number1, Number2, temp)	temp = Number1 <= Number2 ? 1:0
(<, Number1, Number2, temp)	temp = Number1 < Number2 ? 1:0
(>, Number1, Number2, temp)	temp = Number1 > Number2 ? 1:0
(<!=, Number1, Number2, temp)	temp = Number1 != Number2 ? 1:0
(JNZ , temp , _ , Posi)	if(temp == true) goto Posi
(JEZ , temp , _ , Posi)	if(temp != true) goto Posi
(J , _ , _ , Posi)	goto Posi
(FINISH , _ , _ , _)	

附录 C 全部测试样例

1 正确用例结果展示

正确测试样例为储存在 test 文件夹下的 test.txt 文件，下面是源文件及执行结果展示。

```

#include <iostream>
int main()
{
    int a; int b2b; int i; int j; int b;
    char c;
    string str;
    scanf(a)
    scanf(c)
    scanf(str)
    printf(a)
    printf(c)
    printf(str)
    for(i = -3 ; i<0 ; i +=1)
    {
        j = 1 ;
        for(; j <= 1 && j != 2 ; j += 2)
        {
            scanf(b)
            while(b2b != 0 && a != 0)
            do{
                a = (a/a) - (a+a) * (a-a) - a ;
                b2b= b2b << 1 ;
            }
            printf(a)
            a = a[1];
        }
        printf(b2b)
    }
}
printf(a)
printf(b)
printf(c)
b = b& 1;
b = 2;
while(b != 0)
do{
    scanf(a)
    if(a != 0)
    then
    {
        if(a == c || a>= b || a< b || (a+b)> (b - 4 )/b * 3 && a != b)
        then a += 1 * 2 - b/1 ;
        else a -= ( 1 + 1 ) - ( b * 1 );
    }
    a *= 0;
    b /= 2;
}
else
{
    for(i = 1 ; i<= 4 ; i+=1)
    a += ( a * b ) - ( b >> 2 );
    a -= ( a + 1 ) << 1;
    printf(a)
}
printf(a)
printf(b)
printf(c)
printf(str)

```

图 1.1 test.txt 源文件

字符值:char 符号类别:CHAR	字符值: 符号类别:)	字符值j 符号类别:ID	字符值do 符号类别:DO	字符值:<< 符号类别:<<	字符值: 符号类别:)
字符值:c 符号类别:ID	字符值:printf符号类别:PRINTF	字符值:<= 符号类别:<=	字符值:{ 符号类别:{	字符值:1 符号类别:NUM	字符值:printf符号类别:PRINTF
字符值: 符号类别:;	字符值{(符号类别:(字符值:1 符号类别:NUM	字符值:a 符号类别:ID	字符值: 符号类别:;	字符值:(符号类别:(
字符值:string符号类别:STRING	字符值:str 符号类别:ID	字符值:&& 符号类别:&&	字符值:= 符号类别:=	字符值: 符号类别:;	字符值:c 符号类别:ID
字符值:str 符号类别:ID	字符值: 符号类别:)	字符值j 符号类别:ID	字符值:(符号类别:(字符值:printf符号类别:PRINTF	字符值: 符号类别:)
字符值: 符号类别:;	字符值:for 符号类别:FOR	字符值!= 符号类别:!=	字符值:a 符号类别:ID	字符值:(符号类别:(字符值:b 符号类别:ID
字符值:scanf 符号类别:SCANF	字符值{(符号类别:(字符值:2 符号类别:NUM	字符值:/ 符号类别:/	字符值:a 符号类别:ID	字符值:= 符号类别:=
字符值{(符号类别:(字符值i 符号类别:ID	字符值: 符号类别:;	字符值:a 符号类别:ID	字符值: 符号类别:)	字符值:b 符号类别:ID
字符值:a 符号类别:ID	字符值:= 符号类别:=	字符值j 符号类别:ID	字符值: 符号类别:)	字符值:a 符号类别:ID	字符值:& 符号类别:&
字符值: 符号类别:)	字符值:-3 符号类别:NUM	字符值+= 符号类别:+=	字符值:- 符号类别:-	字符值:= 符号类别:=	字符值:1 符号类别:NUM
字符值:scanf 符号类别:SCANF	字符值: 符号类别:;	字符值:2 符号类别:NUM	字符值:(符号类别:(字符值:a 符号类别:ID	字符值: 符号类别:;
字符值{(符号类别:(字符值i 符号类别:ID	字符值: 符号类别:)	字符值:a 符号类别:ID	字符值:] 符号类别:]	字符值:b 符号类别:ID
字符值:c 符号类别:ID	字符值:< 符号类别:<	字符值{(符号类别:(字符值:+ 符号类别:+	字符值:1 符号类别:NUM	字符值:= 符号类别:=
字符值:scanf 符号类别:SCANF	字符值:0 符号类别:NUM	字符值:scanf 符号类别:SCANF	字符值:a 符号类别:ID	字符值: 符号类别:;	字符值:2 符号类别:NUM
字符值: 符号类别:)	字符值: 符号类别:;	字符值:(符号类别:(字符值: 符号类别:)	字符值: 符号类别:)	字符值: 符号类别:;
字符值:str 符号类别:ID	字符值i 符号类别:ID	字符值b 符号类别:ID	字符值:* 符号类别:*	字符值:printf符号类别:PRINTF	字符值:while 符号类别:WHILE
字符值: 符号类别:)	字符值+= 符号类别:+=	字符值: 符号类别:)	字符值:(符号类别:(字符值:(符号类别:(字符值:(符号类别:(
字符值:printf符号类别:PRINTF	字符值:1 符号类别:NUM	字符值:while 符号类别:WHILE	字符值:a 符号类别:ID	字符值:b2b 符号类别:ID	字符值:b 符号类别:ID
字符值{(符号类别:(字符值: 符号类别:)	字符值:(符号类别:(字符值:- 符号类别:-	字符值: 符号类别:)	字符值:= 符号类别:=
字符值:a 符号类别:ID	字符值j 符号类别:ID	字符值:b2b 符号类别:ID	字符值:a 符号类别:ID	字符值: 符号类别:)	字符值:0 符号类别:NUM
字符值: 符号类别:)	字符值:= 符号类别:=	字符值!= 符号类别:!=	字符值:- 符号类别:-	字符值:printf符号类别:PRINTF	字符值:do 符号类别:DO
字符值:printf符号类别:PRINTF	字符值:1 符号类别:NUM	字符值:&& 符号类别:&&	字符值:a 符号类别:ID	字符值:a 符号类别:ID	字符值:(符号类别:(
字符值:(符号类别:(字符值: 符号类别:;	字符值:a 符号类别:ID	字符值: 符号类别:;	字符值: 符号类别:)	字符值:scanf 符号类别:SCANF
字符值:c 符号类别:ID	字符值for 符号类别:FOR	字符值!= 符号类别:!=	字符值:b2b 符号类别:ID	字符值:printf符号类别:PRINTF	字符值:(符号类别:(

图 1.2 词法分析结果_wordsresult

ID表;		
字符值:a 符号类别:ID	符号值:< 符号类别:Sign	
字符值:b 符号类别:ID	符号值:<< 符号类别:Sign	
字符值:b2b 符号类别:ID	符号值:<= 符号类别:Sign	
字符值:c 符号类别:ID	符号值:= 符号类别:Sign	
字符值:i 符号类别:ID	符号值:== 符号类别:Sign	
字符值:j 符号类别:ID	符号值:> 符号类别:Sign	
字符值:str 符号类别:ID	符号值:>= 符号类别:Sign	
	符号值:>> 符号类别:Sign	
符号表;	符号值{(符号类别:Sign	
符号值:!= 符号类别:Sign	符号值 符号类别:Sign	
符号值:# 符号类别:Sign	符号值 符号类别:Sign	
符号值:& 符号类别:Sign	符号值) 符号类别:Sign	
符号值:&& 符号类别:Sign		
符号值:(符号类别:Sign	保留字表;	
符号值:) 符号类别:Sign	符号值:char 符号类别:保留字	
符号值:* 符号类别:Sign	符号值:do 符号类别:保留字	
符号值:*= 符号类别:Sign	符号值:else 符号类别:保留字	
符号值:+ 符号类别:Sign	符号值:for 符号类别:保留字	
符号值:+= 符号类别:Sign	符号值:if 符号类别:保留字	
符号值:- 符号类别:Sign	符号值:include符号类别:保留字	
符号值:=- 符号类别:Sign	符号值:int 符号类别:保留字	
符号值:/ 符号类别:Sign	符号值:iostream符号类别:保留字	
符号值:/= 符号类别:Sign	符号值:main 符号类别:保留字	
符号值;; 符号类别:Sign	符号值:printf符号类别:保留字	
	符号值:scanf 符号类别:保留字	

图 1.3 词法分析结果_worddivide

```

100 (INT ,a ,_ ,_ ) 125 (JNZ ,t6 ,_ ,130 ) 150 (J ,_ ,_ ,127 ) 175 (/ ,t31 ,b ,t32 ) 200 (J ,_ ,_ ,219 )
101 (INT ,b2b ,_ ,_ ) 126 (JEZ ,t6 ,_ ,151 ) 151 (OUT ,b2b ,_ ,_ ) 176 (* ,t32 ,3 ,t33 ) 201 (= ,i ,1 ,_ )
102 (INT ,i ,_ ,_ ) 127 (+ ,j ,2 ,t7 ) 152 (J ,_ ,_ ,117 ) 177 (> ,t30 ,t33 ,t34 ) 202 (<= ,i ,4 ,t50 )
103 (INT ,j ,_ ,_ ) 128 (= ,j ,t7 ,_ ) 153 (OUT ,a ,_ ,_ ) 178 (!= ,a ,b ,t35 ) 203 (JNZ ,t50 ,_ ,208 )
104 (INT ,b ,_ ,_ ) 129 (J ,_ ,_ ,121 ) 154 (OUT ,b ,_ ,_ ) 179 (& ,t34 ,t35 ,t36 ) 204 (JEZ ,t50 ,_ ,219 )
105 (CHAR ,c ,_ ,_ ) 130 (IN ,b ,_ ,_ ) 155 (OUT ,c ,_ ,_ ) 180 (!= ,t36 ,0 ,t37 ) 205 (+ ,i ,1 ,t51 )
106 (STR ,str ,_ ,_ ) 131 (!= ,b2b ,0 ,t8 ) 156 (& ,b ,1 ,t20 ) 181 (| ,t29 ,t37 ,t38 ) 206 (= ,i ,t51 ,_ )
107 (IN ,a ,_ ,_ ) 132 (!= ,a ,0 ,t9 ) 157 (= ,b ,t20 ,_ ) 182 (!= ,t38 ,0 ,t39 ) 207 (J ,_ ,_ ,202 )
108 (IN ,c ,_ ,_ ) 133 (& ,t8 ,t9 ,t10 ) 158 (= ,b ,2 ,_ ) 183 (JNZ ,t39 ,_ ,185 ) 208 (* ,a ,b ,t53 )
109 (IN ,str ,_ ,_ ) 134 (!= ,t10 ,0 ,t11 ) 159 (!= ,b ,0 ,t21 ) 184 (JEZ ,t39 ,_ ,191 ) 209 (>> ,b ,2 ,t54 )
110 (OUT ,a ,_ ,_ ) 135 (JNZ ,t11 ,_ ,137 ) 160 (JNZ ,t21 ,_ ,162 ) 185 (* ,1 ,2 ,t41 ) 210 (- ,t53 ,t54 ,t55 )
111 (OUT ,c ,_ ,_ ) 136 (JEZ ,t11 ,147 ) 161 (JEZ ,t21 ,_ ,222 ) 186 (/ ,b ,1 ,t42 ) 211 (+ ,a ,t55 ,t52 )
112 (OUT ,str ,_ ,_ ) 137 (/ ,a ,a ,t12 ) 162 (IN ,a ,_ ,_ ) 187 (- ,t41 ,t42 ,t43 ) 212 (= ,a ,t52 ,_ )
113 (= ,i ,-3 ,_ ) 138 (+ ,a ,a ,t13 ) 163 (!= ,a ,0 ,t22 ) 188 (+ ,a ,t43 ,t40 ) 213 (+ ,a ,1 ,t57 )
114 (< ,i ,0 ,t1 ) 139 (- ,a ,a ,t14 ) 164 (JNZ ,t22 ,_ ,166 ) 189 (= ,a ,t40 ,_ ) 214 (<< ,t57 ,1 ,t58 )
115 (JNZ ,t1 ,_ ,120 ) 140 (* ,t13 ,t14 ,t15 ) 165 (JEZ ,t22 ,_ ,201 ) 190 (J ,_ ,_ ,200 ) 215 (- ,a ,t58 ,t56 )
116 (JEZ ,t1 ,_ ,153 ) 141 (- ,t12 ,t15 ,t16 ) 166 (== ,a ,c ,t23 ) 191 (+ ,1 ,1 ,t45 ) 216 (= ,a ,t56 ,_ )
117 (+ ,i ,1 ,t2 ) 142 (- ,t16 ,a ,t17 ) 167 (>= ,a ,b ,t24 ) 192 (* ,b ,1 ,t46 ) 217 (OUT ,a ,_ ,_ )
118 (= ,i ,t2 ,_ ) 143 (= ,a ,t17 ,_ ) 168 (| ,t23 ,t24 ,t25 ) 193 (- ,t45 ,t46 ,t47 ) 218 (J ,_ ,_ ,205 )
119 (J ,_ ,_ ,114 ) 144 (<< ,b2b ,1 ,t18 ) 169 (!= ,t25 ,0 ,t26 ) 194 (- ,a ,t47 ,t44 ) 219 (- ,b ,1 ,t59 )
120 (= ,j ,1 ,_ ) 145 (= ,b2b ,t18 ,_ ) 170 (< ,a ,b ,t27 ) 195 (= ,a ,t44 ,_ ) 220 (= ,b ,t59 ,_ )
121 (<= ,j ,1 ,t3 ) 146 (J ,_ ,_ ,131 ) 171 (| ,t26 ,t27 ,t28 ) 196 (* ,a ,0 ,t48 ) 221 (J ,_ ,_ ,159 )
122 (!= ,j ,2 ,t4 ) 147 (OUT ,a ,_ ,_ ) 172 (!= ,t28 ,0 ,t29 ) 197 (= ,a ,t48 ,_ ) 222 (OUT ,a ,_ ,_ )
123 (& ,t3 ,t4 ,t5 ) 148 (| ,a ,1 ,t19 ) 173 (+ ,a ,b ,t30 ) 198 (/ ,b ,2 ,t49 ) 223 (OUT ,b ,_ ,_ )
124 (!= ,t5 ,0 ,t6 ) 149 (= ,a ,t19 ,_ ) 174 (- ,b ,4 ,t31 ) 199 (= ,b ,t49 ,_ ) 224 (OUT ,c ,_ ,_ )

```

图 1.4 test.txt 语法分析结果

请选择执行方式：1. 直接执行 2. 按行调试 3. 从断点之后开始执行，退出则按q即可

1

```

请输入变量a:1      b2b的值为0      a的值为-14
请输入变量c:sss    请输入变量b:-1    a的值为40
请输入变量str:sss  a的值为1      请输入变量a:1
a的值为1          b2b的值为0      a的值为2
c的值为s          a的值为1      b的值为0
str的值为sss      b的值为-1    c的值为s
请输入变量b:5     c的值为s      str的值为sss
a的值为1          请输入变量a:0    执行结束
b2b的值为0        a的值为-2
请输入变量b:0     a的值为4
a的值为1

```

图 1.5 解释执行结果

2 错误用例结果展示

下面将按照表 4.1 的错误表分别展示不同错误情况下，程序会返回的结果。下面的图 2.1 至图 2.16 分别展示了 16 类错误的输出结果和对应的源文件情况。

```
请输入测试文件名称
error1
无法打开文件 error1
请检查文件名称和位置是否正确
```

图 2.1 文件位置或名称错误

```
2.txt - 记事本
文件(F) 编辑(E) 格式(O) 查
#include <iostream>
int main()
{
%
}
```

```
请输入测试文件名称
2.txt
不合法字符 % 位置: 4行1列
```

图 2.2 输入了词法允许以外的符号

```
3.txt - 记事本
文件(F) 编辑(E) 格式(O)
#include<cstdio>
int main()
{
}
```

```
Expect for 'iostream' to start
```

图 2.3 没有函数必要的开始符号

```
4.txt - 记事本
文件(F) 编辑(E) 格式(O)
#include<iostream>
int main()
{
}
```

```
Please Check where ID was define 错误位置5行1列
```

图 2.4 没有定义变量

```
5.txt - 记事本
文件(F) 编辑(E) 格式(O) i
#include<iostream>
int main()
{
    int a;
}
```

```
缺少执行语句! 错误位置:5行1列
```

图 2.5 缺少执行语句

```
6.txt - 记事本
文件(F) 编辑(E) 格式(O)
#include<iostream>
int main()
{
    int a;
    a = 3
}
```

缺少 ';' 错误位置:6行1列

图 2.6 语句缺少;

```
7.txt - 记事本
文件(F) 编辑(E) 格式(O)
#include<iostream>
int main()
{
    int a;
    a = 3;
```

缺少}作为结尾位置 5行9列

图 2.7 缺少匹配的)或}

```
8.txt - 记事本
文件(F) 编辑(E) 格式(O)
#include<iostream>
int main()
{
    int a;
    while a == 3)
}
```

缺少(错误位置:5行10列

图 2.8 语句缺少(或{

```
9.txt - 记事本
文件(F) 编辑(E) 格式(O)
#include<iostream>
int main()
{int a;
if(a = 3)
a = 4;
}
```

缺少Then 错误位置:5行1列

图 2.9 if 语句缺少 then

```
10.txt - 记事本
文件(F) 编辑(E) 格式(O)
#include<iostream>
int main()
{
    int a;
    while(a == 1) a-=1;
}
```

缺少Do 错误位置:5行15列

图 2.10 while 语句缺少 do

11.txt - 记事本

文件(F) 编辑(E) 格式(O) 窗口(W) 帮助(H)

#include<iostream>

int main()

{

int a;int b;

scanf(a b)

}

scanf语句每次只能赋值一个变量! 错误位置:5行10列

图 2.11 scanf, printf 语句每次只能操作一个变量

12.txt - 记事本

文件(F) 编辑(E) 格式(O) 窗口(W) 帮助(H)

#include<iostream>

int main()

{

int a; int b;

a = a+ ;

}

不合规的语句; 位置 5行10列

图 2.12 对于双目运算符只提供了一个操作数

13.txt - 记事本

文件(F) 编辑(E) 格式(O) 窗口(W) 帮助(H)

#include<iostream>

int main()

{

int b;

a = 3;

}

未被定义的变量a 错误位置:5行5列

图 2.13 使用了未定义变量

14.txt - 记事本

文件(F) 编辑(E) 格式(O) 窗口(W) 帮助(H)

#include<iostream>

int main()

{

int a;int a;

}

重定义元素! 错误位置4行12列

图 2.14 重定义变量

15.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

#include<iostream>

int main()

{

int a;

for(int b = 3 ; b < 4 ; b += 1)

}

请在最开始声明变量! 错误位置5行10列

图 2.15 在错误位置声明变量

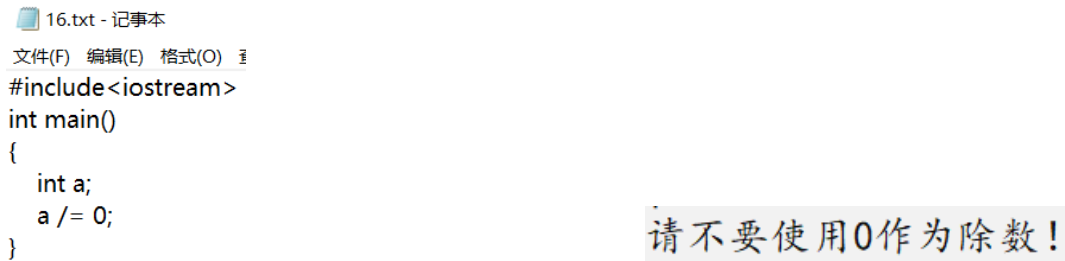


图 2.16 使用 0 作为除数

以上展示的便是全部的 16 类错误的展示简单程序和出现错误时的提示显示。具体的 30 种结果不在这里做详细展示，每一种错误都有对应的错误描述和相应的错误出现位置。可以利用错误提示和错误位置展示及时进行修改。但是需要申明的一点是，再进行分析时，每次只能分析出最开始的错误，无法将全部错误一次性找出，这也是我们需要改善的。