

# 哈尔滨工业大学

## 编译原理

## 大作业

姓名：	张志路
学号：	1160300909
班号：	1603106
学院：	计算机学院

2019 年 5 月

# 目 录

## 第一章 词法分析

1	词法分析需求分析.....	5
1.1	实验目的.....	5
1.2	具体要求.....	5
2	词法分析文法设计.....	6
2.1	给出各类单词的词法规则描述.....	6
2.2	各类单词的转换图.....	6
3	词法分析系统设计.....	9
3.1	系统概要设计.....	9
3.1.1	系统框架图.....	9
3.1.2	数据流图.....	9
3.1.3	功能模块图.....	10
3.2	系统详细设计.....	10
3.2.1	核心数据结构的设计.....	10
3.2.2	主要功能函数说明.....	11
3.2.3	程序核心部分的程序流程图.....	12
4	词法分析系统实现及结果分析.....	12
4.1	系统实现过程中遇到的问题.....	12
4.2	词法分析结果.....	15
4.3	词法错误报告.....	17
4.4	分析实验结果.....	17

## 第二章 语法分析

1	语法分析需求分析.....	19
2	语法分析文法设计.....	19
3	语法分析系统设计.....	21
3.1	系统概要设计.....	21
3.1.1	系统框架图.....	21
3.1.2	数据流图.....	21
3.1.3	功能模块图.....	22
3.2	系统详细设计.....	22

3.2.1	核心数据结构的设计.....	22
3.2.2	主要功能函数说明.....	24
3.2.3	程序核心部分的程序流程图.....	27
4	语法分析系统实现及结果分析.....	27
4.1	系统实现过程中遇到的问题.....	27
4.2	输出句法分析器的分析表.....	28
4.3	针对一测试程序输出其句法分析结果.....	28
4.4	错误报告.....	30
4.5	分析实验结果.....	30

### 第三章 语义分析和中间代码生成

1	语义分析需求分析.....	32
2	语义分析文法设计.....	32
3	语义分析系统设计.....	36
3.1	系统概要设计.....	36
3.1.1	系统框架图.....	36
3.1.2	数据流图.....	36
3.1.3	功能模块图.....	37
3.2	系统详细设计.....	37
3.2.1	核心数据结构的设计.....	37
3.2.2	主要功能函数说明.....	40
3.2.3	程序核心部分的程序流程图.....	46
4	语义分析系统实现及结果分析.....	47
4.1	系统实现过程中遇到的问题.....	47
4.2	针对一测试程序输出其语义分析结果.....	47
4.3	语义分析后的符号表.....	51
4.4	语义错误报告.....	51
4.5	分析实验结果.....	52

### 第四章 代码优化

1	代码优化需求分析.....	53
2	优化过程.....	53
2.1	常见的代码优化手段.....	53

2.2	基本块内的局部优化.....	53
2.3	循环优化.....	54
3	代码优化系统设计.....	54
3.1	系统逻辑.....	54
3.2	系统具体实现.....	55
4	基本块划分结果.....	57

## 第五章 目标代码生成

1	代码生成需求分析.....	58
2	常用四元式的翻译方法.....	58
3	代码生成系统设计.....	59
3.1	系统逻辑.....	59
3.2	系统具体实现.....	60
4	代码生成结果分析.....	60
4.1	代码生成结果.....	60
4.2	分析实验结果.....	60

## 第六章 完整编译系统的设计

1	需求分析.....	61
2	系统设计.....	61
3	系统结果.....	61
3.1	测试样例.....	61
3.2	系统结果.....	63

# 第一章 词法分析

## 1 词法分析需求分析

### 1.1 实验目的

设计实现类高级语言的词法分析器，基本功能为识别以下几类单词：

(1) 标识符

由大小写字母、数字以及下划线组成，但必须以字母或者下划线开头。

(2) 关键字

①类型关键字：整型、浮点型、布尔型、记录型；

②分支结构中的 if 和 else；

③循环结构中的 do 和 while。

(3) 运算符

①算术运算符；

②关系运算符；

③逻辑运算。

(4) 界符

①用于赋值语句的界符，如 “=”；

②用于句子结尾的界符，如 “；”。

(5) 常数

无符号整数和浮点数等。

(6) 注释

/\*.....\*/形式。

### 1.2 具体要求

(1) 要求基于 DFA 技术设计词法分析器。

(2) 系统的输入形式：要求能够通过文件导入测试用，测试用例要涵盖“实验内容”中列出的各类单词，并包含各种单词拼写错误。

(3) 系统的输出分为两部分：一部分是打印输出词法分析器的符号表，另一部分是打印输出源程序对应的 token 序列。

(4) 在基本要求的基础上，本程序又实现了识别单个字符(char)和字符串(String)的算法。

## 2 词法分析文法设计

### 2.1 给出各类单词的词法规则描述

令  $\text{letter\_} = [\text{A-Za-z\_}]$ ,  $\text{digit} = [0-9]$ 。

(1) 标识符

$\text{letter\_}(\text{letter\_}\text{digit})^*$

(2) 关键字

$\text{char} \mid \text{long} \mid \text{short} \mid \text{float} \mid \text{double} \mid \text{const} \mid \text{Boolean} \mid \text{void} \mid \text{null} \mid \text{false} \mid \text{true} \mid \text{enum} \mid$   
 $\text{int} \mid \text{do} \mid \text{while} \mid \text{if} \mid \text{else} \mid \text{for} \mid \text{then} \mid \text{break} \mid \text{continue} \mid \text{class} \mid \text{static} \mid \text{final} \mid \text{extends} \mid \text{new}$   
 $\mid \text{return} \mid \text{signed} \mid \text{struct} \mid \text{union} \mid \text{unsigned} \mid \text{goto} \mid \text{switch} \mid \text{case} \mid \text{default} \mid \text{auto} \mid \text{extern} \mid$   
 $\text{register} \mid \text{sizeof} \mid \text{typedef} \mid \text{volatile} \mid \text{String}$

(3) 运算符

$< \mid > \mid <= \mid >= \mid == \mid != \mid + \mid - \mid * \mid / \mid = \mid ++ \mid -- \mid << \mid >> \mid || \mid \&\& \mid \& \mid || \mid ! \mid ^ \mid \% \mid$   
 $+= \mid -= \mid *= \mid /=$

(4) 界符

$( \mid ) \mid \{ \mid \} \mid ; \mid [ \mid ] \mid , \mid =$

(5) 无符号常数

$\text{digit digit}^* (\backslash.\text{digit digit}^*)? ([\text{Ee}][+-]? \text{digit digit}^*)?$

(6) 注释

$/* */$  型注释:  $\wedge^*(\backslash.\backslash\text{r}\backslash\text{n})^*\backslash*/$

$//$  型注释:  $//[\backslash\text{s}\backslash\text{S}]^*\backslash\text{n}$

(7) 单引号内的字符 (char)

$\backslash(\backslash.)\backslash'$

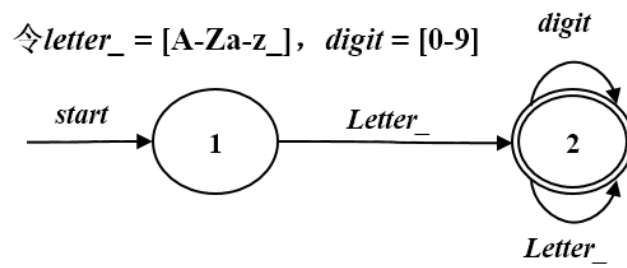
(8) 双引号内的字符串 (String)

$\backslash"(.*)\backslash"$

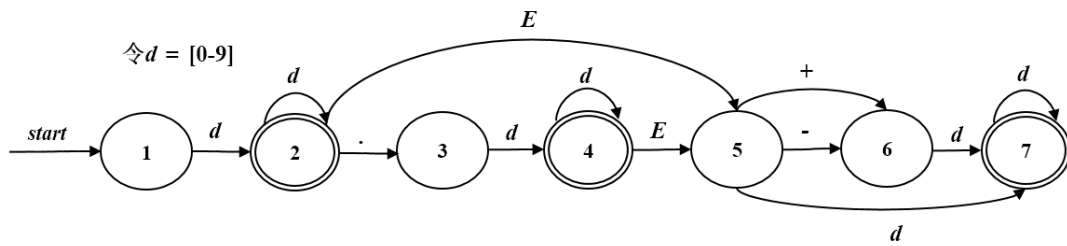
### 2.2 各类单词的转换图

关键字和界符的判断比较简单, DFA 不再给出。

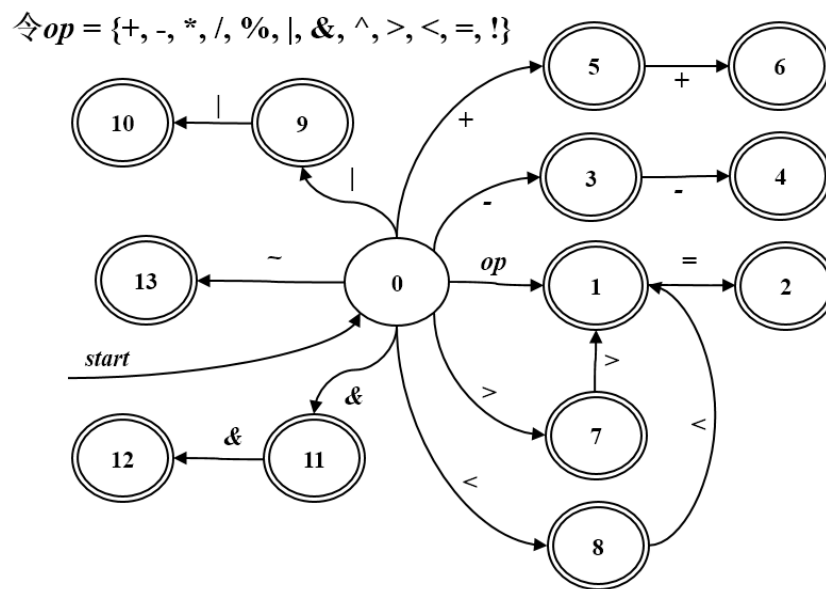
(1) 标识符



(2) 无符号常数

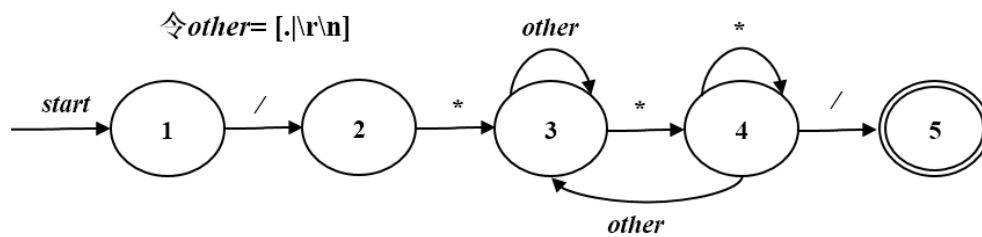


(3) 运算符

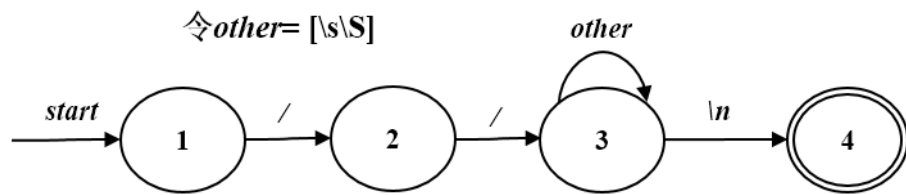


(4) 注释

/\* \*/型注释

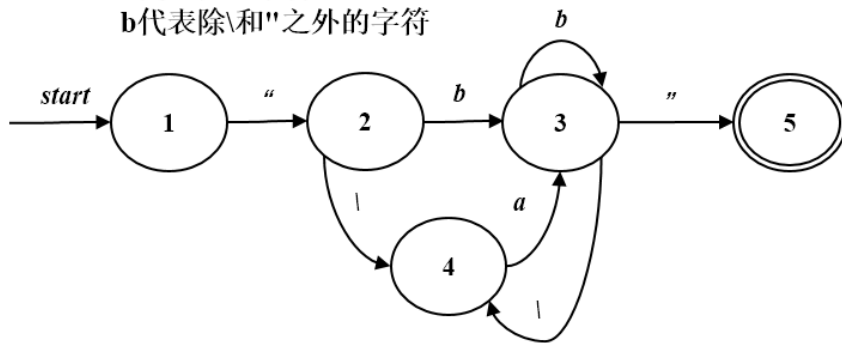


//型注释



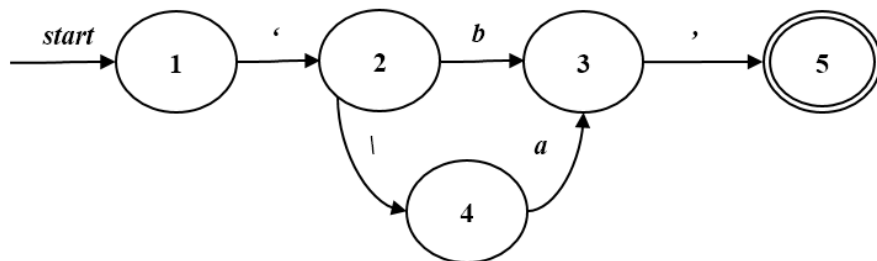
(5) 字符串

a 代表任意字符  
b 代表除 \ 和 " 之外的字符



(6) 单个字符

a 代表任意字符  
b 代表除 \ 和 ' 之外的字符

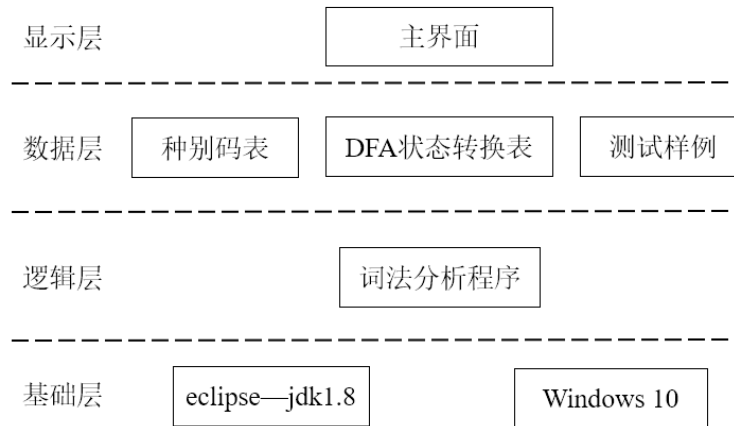




### 3 词法分析系统设计

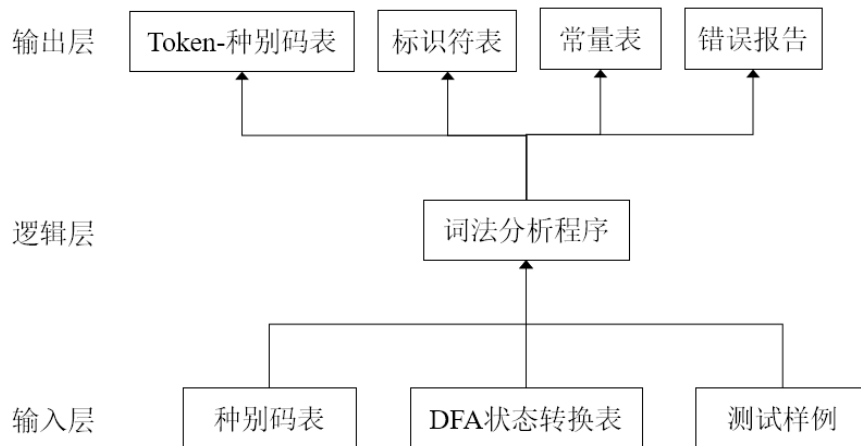
#### 3.1 系统概要设计

##### 3.1.1 系统框架图



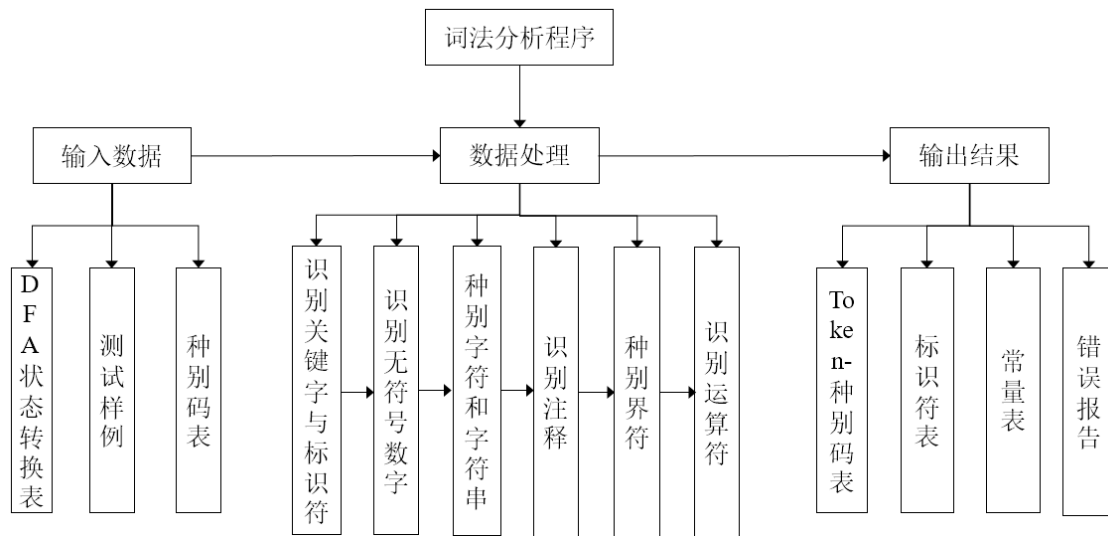
系统分为基础层、逻辑层、数据层和显示层，基础层即在 Windows10、jdk1.8 环境下利用 java 语言进行编程，逻辑层即词法分析程序，数据层包含 DFA 转换表、测试样例和设计的种别码表，显示层显示一个主界面。

##### 3.1.2 数据流图



词法分析程序通过读入 DFA 转换表和设计的种别码表，对测试样例进行分析，从而对结果进行显示，给出 Token-种别码表、标识符表、常量表和错误报告。

### 3.1.3 功能模块图



如上图所示，词法分析程序的功能主要为读入数据、分析数据和给出结果。分析数据的过程即对各类单词进行识别。

## 3.2 系统详细设计

### 3.2.1 核心数据结构的设计

```

private String text; // 读入的测试样例文本
private JTable jtable1; // 实为Object[][]数组，存储识别信息，即“行数-Token-种别码-单词类别”
private JTable jtable2; // 实为Object[][]数组，存储错误报告，即“行数-错误内容-错误信息”
private JTable jtable3; // Object[][]数组，存储的是标识符表，即“标识符-标识符位置”
private JTable jtable4; // Object[][]数组，存储的是常量表，即“常量-常量位置”

/**
 * 构造函数
 * @param text String型，读入的测试样例文本
 * @param jtable1 实为Object[][]数组，存储识别信息，即“行数-Token-种别码”
 * @param jtable2 实为Object[][]数组，存储错误报告，即“行数-错误内容-错误信息”
 * @param jtable3 实为Object[][]数组，存储的是标识符表，即“标识符-标识符位置”
 * @param jtable4 实为Object[][]数组，存储的是常量表，即“常量-常量位置”
 */
public Lexical(String text, JTable jtable1, JTable jtable2, JTable jtable3, JTable jtable4)
{
    this.text = text;
    this.jtable1 = jtable1;
    this.jtable2 = jtable2;
    this.jtable3 = jtable3;
    this.jtable4 = jtable4;
}
  
```

### 3.2.2 主要功能函数说明

#### (1) lex 函数

```
/**
 * 核心函数
 * 根据已经构成的DFA状态转换表
 * 按行分析数据，识别相应信息
 */
public void lex()
```

该函数是程序的核心函数，其程序流程图如 3.2.3 所示，该函数识别各类单词，并输出相应信息和错误报告。

#### (2) is\_string\_state 函数

```
/**
 * 字符串DFA状态匹配函数
 * @param ch 当前字符
 * @param key 状态表中的字符
 * @return 匹配成功返回true，否则返回false
 */
public static Boolean is_string_state(char ch, char key)
```

#### (3) is\_char\_state 函数

```
/**
 * 字符DFA状态匹配函数
 * @param ch 当前字符
 * @param key 状态表中的字符
 * @return 匹配成功返回true，否则返回false
 */
public static Boolean is_char_state(char ch, char key)
```

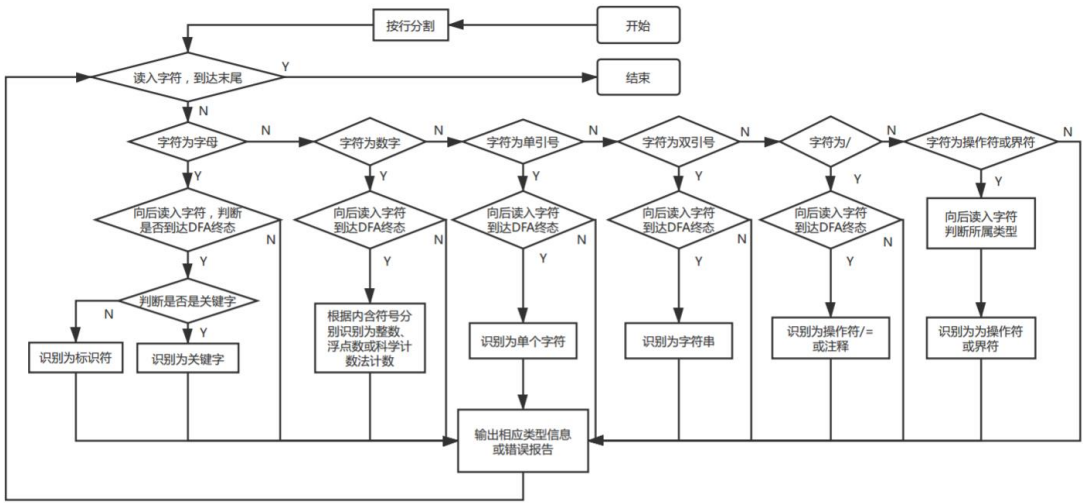
#### (4) is\_digit\_state 函数

```
/**
 * 数字DFA状态匹配函数
 * @param ch 当前字符
 * @param test 状态表中的字符
 * @return 匹配成功返回true，否则返回false
 */
public static int is_digit_state(char ch, char test)
```

#### (5) is\_note\_state 函数

```
/**
 * 注释DFA状态匹配函数
 * @param ch 当前字符
 * @param test 状态表中的字符
 * @return 匹配成功返回true，否则返回false
 */
public static Boolean is_note_state(char ch, char nD, int s)
```

3.2.3 程序核心部分的程序流程图



4 词法分析系统实现及结果分析

4.1 系统实现过程中遇到的问题

(1) 种别码编码问题

编码的清晰与否将会直接影响后续程序的实现，经过权衡，本系统对标识符、关键字、无符号数、注释等大类编码为 1~7；关键字一字一码，以 101 开头；运算符一符一码，以 201 开头；界符一符一码，以 301 开头。之所以编码不连续是为了方便后续对其进行扩展。

具体编码如下。

单词类型	具体类型	种别码
标识符	—	1
无符号数	整型常量	2
	浮点型常量	3
	科学计数法	4
字符常量	—	5
字符串常量	—	6
注释	—	7
	auto	101
	double	102
	int	103
	struct	104
	break	105
	else	106
	long	107

关键字	switch	108
	case	109
	enum	110
	register	111
	typedef	112
	char	113
	extern	114
	return	115
	union	116
	const	117
	float	118
	short	119
	unsigned	120
	continue	121
	for	122
	signed	123
	void	124
	default	125
	goto	126
	sizeof	127
	volatile	128
	do	129
	if	130
	while	131
	static	132
	String	133
	+	201
	-	202
	*	203
	/	204
	%	205
	++	206
	--	207
	<	208
	<=	209
	>	210
	>=	211
	==	212
	!=	213
	&&	214

运算符		215
	!	216
	~	217
	&	218
		219
	^	220
	>>	221
	<<	222
	+=	223
	-=	224
	*=	225
	/=	226
	%=	227
	&=	228
	^=	229
	=	230
	<<=	231
	>>=	232
界符	,	301
	;	302
	[	303
	]	304
	{	305
	}	306
	(	307
	)	308
	=	309

## (2) 常量与标识符的记录问题

常量与标识符在编码时都是以大类编码的，不同单词之间未进行区分。为了后续便于语法语义分析，本系统构造了两个 `HashMap`，分别存储标识符和常量的位置信息，其格式均为“单词—位置”形式。

## (3) DFA 设计与实现问题

关于 DFA 的状态转换表，本系统直接在程序中定义，进而构造相应的转换程序进行状态之间的转移。

## (4) 无符号数识别问题

无符号数对其进行了整形、浮点型和科学计数法的区分。在识别科学计数法的无符号数时，要注意 E 的大小写不影响其表达，例如  $1.2E-1=1.2e-1$ ；同时加号

的存在与否也不影响表达，例如  $1.2E+1=1.2E1$ 。

#### (5) 注释识别问题

在进行多行注释识别时，要注意换行符的处理问题，该问题可以通过对 DFA 状态转换表的设计来解决。

#### (6) 错误报告问题

在进行错误的识别时，本着“就近”的原则确定其错误类型。例如在进行无符号数的识别时，如果出现“12m”类似的形式，则认为这是一个无符号数错误，而不是标识符错误。

## 4.2 词法分析结果

测试样例如下。

```
1. while(num != 100)
2. {
3.     num++;
4. }
5.
6. /*test*/
7. //test
8. float a = 1.22;
9. float b = 2.32e+1;
10. float c = 22.3E-1;
11. int d = 22E1;
12. s[1] = 'a';
13. char s = '\t';
14. char s = '\';
15. String t = "ZhangZhilu";
16. /*Zhang Zhilu
17. 1160300909
18. */
19.
20. /* Sort function */
21. void bubbleSort(int *arr, int n)
22. {
23.     for (int i = 0; i < n-1; i++)
24.         for (int j = 0; j < n-i-1; j++)
25.             {
26.                 if (arr[j] > arr[j+1])
27.                     {
28.                         int temp = arr[j];
```

```

29.         arr[j] = arr[j+1];
30.         arr[j+1] = temp;
31.     }
32. }
33. }
34.
35. int main()
36. {
37.     int arr[] = { 10,6,5,2,3,8,7,4,9,1 };
38.     bubbleSort(arr, n);
39.     return 0;
40. }
41.
42.
43. /*Error*/
44. int 1_qqq = 3;
45. float e = 2.2.2;
46. float t = 1..1;
47. char s = '\t;
48. String ff = "zhangzhilu;
49. /*zhang
50. zhi
51. lu

```

注：1~43 行为无词法错误程序，44~51 行均有词法错误。

输出的部分“行号—Token—类别—种别码”表如下。

词法分析程序			
行号	Token	类别	种别码
1	while	关键字	131
1	(	界符	307
1	num	标识符	1
1	!=	运算符	213
1	100	整型常量	2
1	)	界符	308
2	{	界符	305
3	num	标识符	1
3	++	运算符	206
3	:	界符	302
4	}	界符	306
6	/*test*/	注释	7
7	//test	注释	7
8	float	关键字	118
8	a	标识符	1
8	=	界符	309
8	1.22	浮点型常量	3
8	:	界符	302
9	float	关键字	118
9	b	标识符	1
9	=	界符	309
9	2.32e+1	科学计数法	4
9	:	界符	302
10	float	关键字	118
10	c	标识符	1
10	=	界符	309
10	22.3E-1	科学计数法	4
10	:	界符	302



输出的部分“标识符—位置”和“常量—位置”表如下。

标识符	位置	常量	位置
num	0	100	0
a	1	1.22	1
b	2	2.32e+1	2
c	3	22.3E-1	3
d	4	22E1	4
s	5	'a'	5
t	6	'\t'	6
bubble...	7	"Zhang..."	7
arr	8	0	8
n	9	10	9
i	10	6	10
j	11	5	11
temp	12	2	12
main	13	3	13
e	14	o	14
ff	15		15

### 4.3 词法错误报告

针对以下程序的错误报告如下。

```

43  /*Error*/
44  int 1_qqq = 3;
45  float e = 2.2.2;
46  float t = 1..1;
47  char s = '\t;
48  String ff = "zhangzhilu;
49  /*zhang
50  zhi
51  lu

```

错误行号	Token	详细说明
44	1 qqq	无符号数不合规范
45	2.2.2	无符号数不合规范
46	1..1	无符号数不合规范
47	'\t	字符常量引号未封闭
48	"zhangzhilu;	字符串常量引号未封闭
52	/*zhangzhilu	注释未封闭

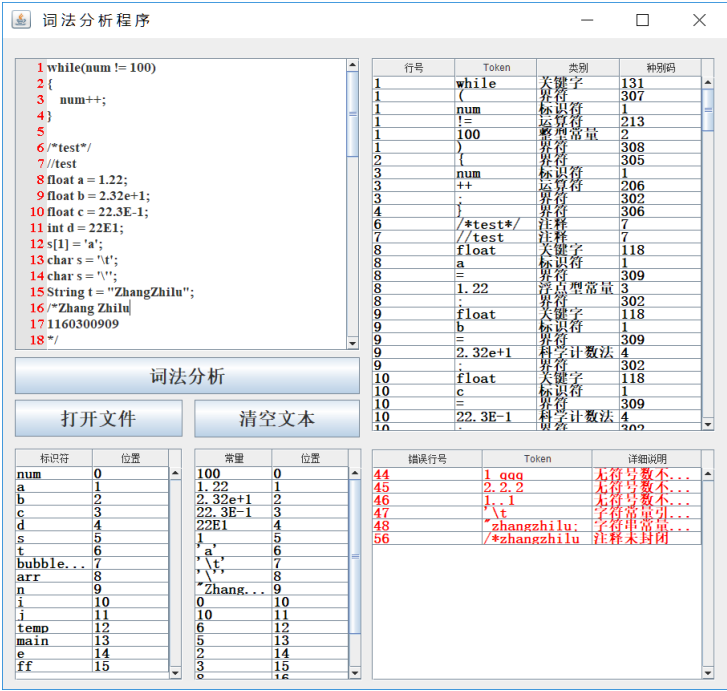
### 4.4 分析实验结果

经过反复地测试与实验，从最后的实验结果来看，该系统对各类单词均进行了细致且正确的识别，例如无符号数“22.3E-1”、字符“'\t'”、运算符“++”和各类注释。

同时也对一些常见的错误进行了识别，例如错误的无符号数“2.2.2”、错误的字符“\t”和注释未封闭的问题。

结果表明，系统的结果与目标基本一致，系统目标达成。

程序运行的主界面如下图所示。



## 第二章 基于 LR(1)方法的语法分析

### 1 语法分析需求分析

要求：采用至少一种句法分析技术（SLR(1)、LR(1)）对类高级语言中的基本语句进行句法分析。阐述句法分析系统所要完成的功能。

在词法分析器的基础上设计实现类高级语言的语法分析器，基本功能如下：

1. 能识别以下几类语句：
  - (1) 声明语句（变量声明）
  - (2) 表达式及赋值语句（简单赋值）
  - (3) 分支语句：if\_then\_else
  - (4) 循环语句：do\_while
2. 要求编写自动计算 CLOSURE(I)和 GOTO 函数的程序，并自动生成 LR 分析表。
3. 具备简单语法错误处理能力，能准确给出错误所在位置，并采用可行的错误恢复策略。输出的错误提示信息格式如下：Error at Line [行号]: [说明文字]
4. 系统的输入形式：要求可以通过文件导入文法和测试用例,测试用例要涵盖“实验内容”第 1 条中列出的各种类型的语句，并设置一些语法错误。
5. 系统的输出分为两部分：一部分是打印输出语法分析器的 LR 分析表。另一部分是打印输出语法分析结果，既输出归约时的产生式序列。

### 2 语法分析文法设计

要求：给出如下语言成分的文法描述。

在本文法在实验指导书给定的文法基础上做了相应的改进，消除了二义性。

1. 全局定义
$$P' \rightarrow P$$
$$P \rightarrow D$$
$$P \rightarrow S$$
$$S \rightarrow S S$$
2. 声明语句（变量声明）
$$D \rightarrow D D \mid \text{proc id ; } D S \mid T \text{ id ;}$$
$$T \rightarrow X C \mid \text{record } D \text{ end}$$
$$X \rightarrow \text{integer} \mid \text{real}$$
$$C \rightarrow [\text{ num } ] C \mid \epsilon$$

## 3. 表达式及赋值语句

$$S \rightarrow id = E ; \mid L = E ;$$
$$E \rightarrow E + E1 \mid E1$$
$$E1 \rightarrow E1 * E2 \mid E2$$
$$E2 \rightarrow ( E ) \mid - E \mid id \mid num \mid L$$
$$L \rightarrow id [ E ] \mid L [ E ]$$

## 4. 分支语句 “if\_then\_else” 和循环语句 “do\_while”

$$S \rightarrow S1 \mid S2$$
$$S1 \rightarrow \text{if } B \text{ then } S1 \text{ else } S1 \mid \text{while } B \text{ do } S0$$
$$S2 \rightarrow \text{if } B \text{ then } S1 \text{ else } S2 \mid \text{if } B \text{ then } S0$$
$$S0 \rightarrow \text{begin } S3 \text{ end}$$
$$S1 \rightarrow \text{begin } S3 \text{ end}$$
$$S2 \rightarrow \text{begin } S3 \text{ end}$$
$$S3 \rightarrow S3 ; S \mid S$$

## 5. 布尔表达式

$$B \rightarrow B \text{ or } B1 \mid B1$$
$$B1 \rightarrow B1 \text{ and } B2 \mid B2$$
$$B2 \rightarrow \text{not } B \mid ( B ) \mid E \text{ R } E \mid \text{true} \mid \text{false}$$
$$R \rightarrow < \mid \leq \mid = \mid \neq \mid > \mid \geq$$

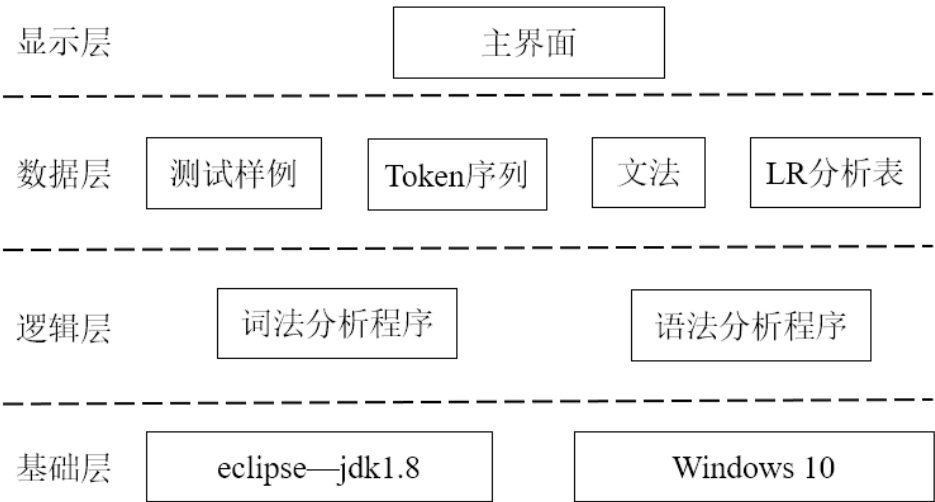
## 6. 过程调用

$$S \rightarrow \text{call } id ( EL )$$
$$EL \rightarrow EL , E$$
$$EL \rightarrow E$$

### 3 语法分析系统设计

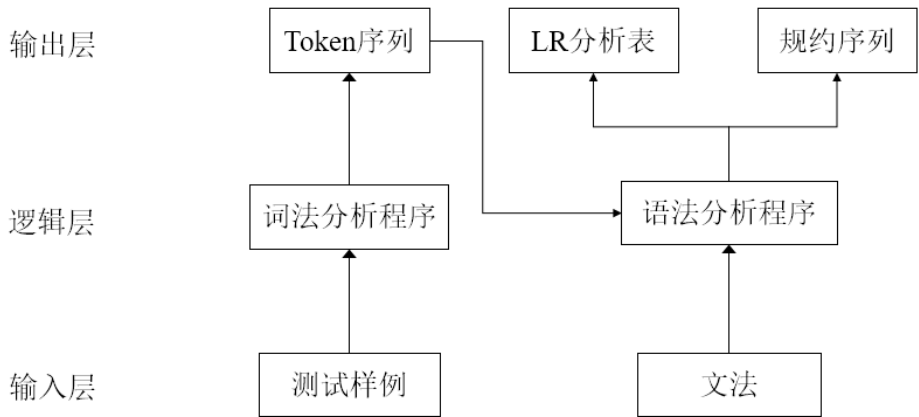
#### 3.1 系统概要设计

##### 3.1.1 系统框架图



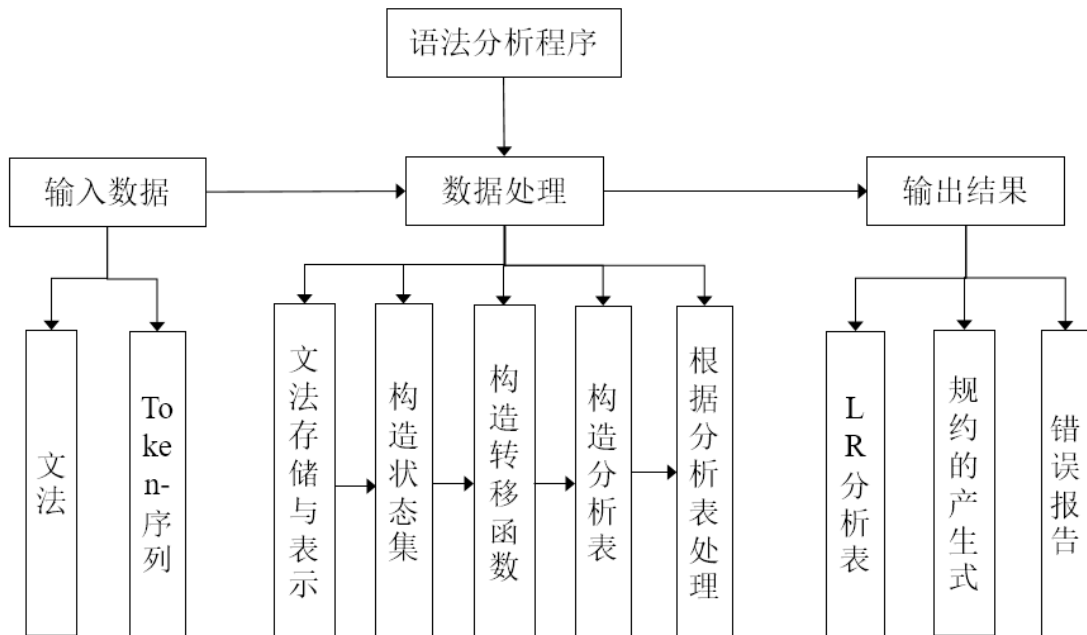
系统分为基础层、逻辑层、数据层和显示层，基础层即在 Windows10、jdk1.8 环境下利用 java 语言进行编程，逻辑层即词法分析程序和语法分析程序，数据层包含测试样例、Token 序列、文法和 LR 分析表，显示层显示一个主界面。

##### 3.1.2 数据流图



语法分析程序通过读入文法，生成 LR 分析表，然后读入对测试样例的词法分析结果，根据分析表对其进行 LR(1)分析，从而生成规约的产生式序列和错误报告。

### 3.1.3 功能模块图



如上图所示，语法分析程序的功能主要为读入数据、分析数据和给出结果。对于数据处理模块，首先要设计合理的数据结构，对文法进行存储和表示；然后根据文法生成状态集和转移函数，进而构造分析表；最后对词法分析得到的Token序列利用分析表进行移进规约操作，直到分析完成。

## 3.2 系统详细设计

### 3.2.1 核心数据结构的设计

#### a. Production 类

```

public String left; // 产生式左部
public ArrayList<String> list = new ArrayList<String>(); // 产生式右部

/**
 * 存储产生式，此时表示类似于“A->BCD”的形式
 * @param s 产生式字符串
 */
public Production(String s)

```

Production 类对一个普通的产生式进行表示，此时表示的是类似于“A → BCD”的形式。

#### b. ProductionState 类

```

public Production d; // 产生式
public String lr; // 后继符
public int index; // 后继符位置

/**
 * DFA状态集中每一个状态
 * 此时表示类似于“A->BC.D, a”的形式
 * @param d 产生式
 * @param lr 后继符
 * @param index 后继符位置
 */
public ProductionState(Production d,String lr,int index)

```

ProductionState 类对一个产生式状态进行表示, 此时表示类似于“A → BC.D, a”的形式。

### c. DFAState 类

```

// 项目集编号,即DFA状态号
public int id;
// DFA中的状态集列表,每个元素表示一个产生式状态
public ArrayList<ProductionState> set = new ArrayList<ProductionState>();

/**
 * 一个DFA状态
 * @param id 项目集编号,即DFA状态号
 */
public DFAState(int id)

```

DFAState 类用来表示 LR(1)分析的一个 DFA 状态, 即一个项目集闭包, id 为项目集编号。

### d. DFAStateSet 类

```

// 所有项目集列表, 每个元素为一个DFA状态
public ArrayList<DFAState> states = new ArrayList<DFAState>();

```

DFAState 类用来表示 LR(1)分析法的所有 DFA 状态, 即所有项目集合。

### e. GrammarProc 类

```

public static String emp = "ε"; // 空串
public static String end = "#"; // 结束符
public static TreeSet<String> VN = new TreeSet<String>(); // 非终结符集
public static TreeSet<String> VT = new TreeSet<String>(); // 终结符集
public static ArrayList<Production> F = new ArrayList<Production>(); // 产生式集
// 每个符号的first集
public static HashMap<String,TreeSet<String>> firstMap = new HashMap<String,TreeSet<String>>();

```

GrammarProc 类用来读入并存储类似于“A → BC|DE”形式的产生式, 根据前述的数据结构进行存储, 产生终结符、非终结符以及每个符号的 FIRST 集。

### f. AnalyzeTable 类

AnalyzeTable 类用来构造一个分析表, 此类构造 DFA 状态集和转移函数, 进而构造 LR(1)分析表。

主要数据结构如下图所示。

```

public static String error = "--"; // 错误符号
public static String acc = "acc"; // ACC, 接收成功符号
public DFAStateSet dfa; // 所有DFA状态
public int stateNum; // DFA状态数

public int actionLength; // Action表列数
public int gotoLength; // GoTo表列数
private String[] actionCol; // Action表列名数组
private String[] gotoCol; // GoTo表列名数组
private String[][] actionTable; // Action表, 二维数组
private int[][] gotoTable; // GoTo表, 二维数组

// 当第x号DFA状态,输入S符号时,转移到第y号DFA状态,则:
private ArrayList<Integer> gotoStart = new ArrayList<Integer>(); // 存储第x号DFA状态
private ArrayList<Integer> gotoEnd = new ArrayList<Integer>(); // 存储第y号DFA状态
private ArrayList<String> gotoPath = new ArrayList<String>(); // 存储S符号

/**
 * 构造分析表
 */
public AnalyzeTable()

```

### g. SyntaxParser 类

SyntaxParser 类根据构造的语法分析表进行 LR(1)语法分析。

```

private Lexical lex; // 词法分析器
private ArrayList<Token> tokenList = new ArrayList<Token>(); // 从词法分析器获得的所有token
private int length; // tokenList的长度
private int index; // 语法分析进行到的位置

private AnalyzeTable table; //构造的语法分析表
private Stack<Integer> stateStack; //用于存储相应的DFA状态号
private static StringBuffer result = new StringBuffer(); // 保存规约结果
private static StringBuffer error = new StringBuffer(); // 保存错误报告结果

```

### 3.2.2 主要功能函数说明

#### a. 求 FIRST 集

不断应用下列规则，直到没有新的终结符或  $\epsilon$  可以被加入到任何 FIRST 集合中为止。

- 1) 如果  $X$  是一个终结符，那么  $\text{FIRST}(X) = \{X\}$ 。
- 2) 如果  $X$  是一个非终结符，且  $X \rightarrow Y_1 \dots Y_k \in P$  ( $k \geq 1$ )，那么，
  - a) 如果对于某个  $i$ ， $a$  在  $\text{FIRST}(Y_i)$  中，且  $\epsilon$  在所有的  $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_{i-1})$  中，就把  $a$  加入到  $\text{FIRST}(X)$  中。
  - b) 如果对于所有的  $j=1, 2, \dots, k$ ， $\epsilon$  在  $\text{FIRST}(Y_j)$  中，那么将  $\epsilon$  加入到  $\text{FIRST}(X)$  中。
- 3) 如果  $X \rightarrow \epsilon \in P$ ，那么将  $\epsilon$  加入到  $\text{FIRST}(X)$  中。

#### b. 求项目集闭包

根据下列式子构造项目集闭包。

$\text{CLOSURE}(I) = I \cup \{[B \rightarrow \cdot \gamma, b] \mid [A \rightarrow \alpha \cdot B \beta, a] \in \text{CLOSURE}(I), B \rightarrow \gamma \in P, b \in \text{FIRST}(\beta a)\}$



伪代码如下所示。

```

1. CLOSURE(I)
2. {
3.     repeat
4.         for (I 中的每个项  $[A \rightarrow \alpha.B\beta, a]$ )
5.             for ( $G'$  的每个产生式  $B \rightarrow \gamma$ )
6.                 for ( $FIRST(\beta a)$  中的每个符号  $b$ )
7.                     将  $[B \rightarrow \gamma, b]$  加入到集合  $I$  中;
8.     until 不能向  $I$  中加入更多的项;
9. until  $I$  ;
10. }
```

### c. 求 GOTO 函数

根据下列式子构造 GOTO 函数。

$GOTO(I, X) = CLOSURE(\{ [A \rightarrow \alpha X.\beta, a] \mid [A \rightarrow \alpha.X\beta, a] \in I \})$

伪代码如下所示。

```

1. GOTO(I, X)
2. {
3.     将  $J$  初始化为空集;
4.     for(I 中的每个项  $[A \rightarrow \alpha.X\beta, a]$ )
5.         将项  $[A \rightarrow \alpha X.\beta, a]$  加入到集合  $J$  中;
6.     return CLOSURE(J);
7. }
```

### d. 为文法 $G'$ 构造 LR(1)项集族

伪代码如下所示。

```

1. items( $G'$ )
2. {
3.     将  $C$  初始化为  $\{CLOSURE(\{[S' \rightarrow .S, \#]\})\}$ ;
4.     repeat
5.         for( $C$  中的每个项集  $I$ )
6.             for(每个文法符号  $X$ )
7.                 if( $GOTO(I, X)$  非空且不在  $C$  中)
8.                     将  $GOTO(I, X)$  加入  $C$  中;
9.     until 不再有新的项集加入到  $C$  中;
10. }
```

### e. 构造 LR(1)语法分析表

伪代码如下所示。

1. 构造  $G'$  的规范 LR(1) 项集族  $C=\{I, I, \dots, I\}$
2. 根据  $I$  构造得到状态  $i$ 。状态  $i$  的语法分析动作按照下面的方法决定：
3. **If**  $[A \rightarrow \alpha \cdot a\beta, b] \in I$  and  $GOTO(I, a) = I$
4.      $ACTION[i, a] = sj$
5. **If**  $[A \rightarrow \alpha \cdot B\beta, b] \in I$  and  $GOTO(I, B) = I$
6.      $GOTO[i, B] = j$
7. **If**  $[A \rightarrow \alpha \cdot, a] \in I$  and  $A \neq S'$
8.      $ACTION[i, a] = rj$  ( $j$  是产生式  $A \rightarrow \alpha$  的编号)
9. **If**  $[S' \rightarrow S \cdot, \#] \in I$
10.      $ACTION[i, \#] = acc;$
11. 没有定义的所有条目都设置为“error”

#### f. 错误处理

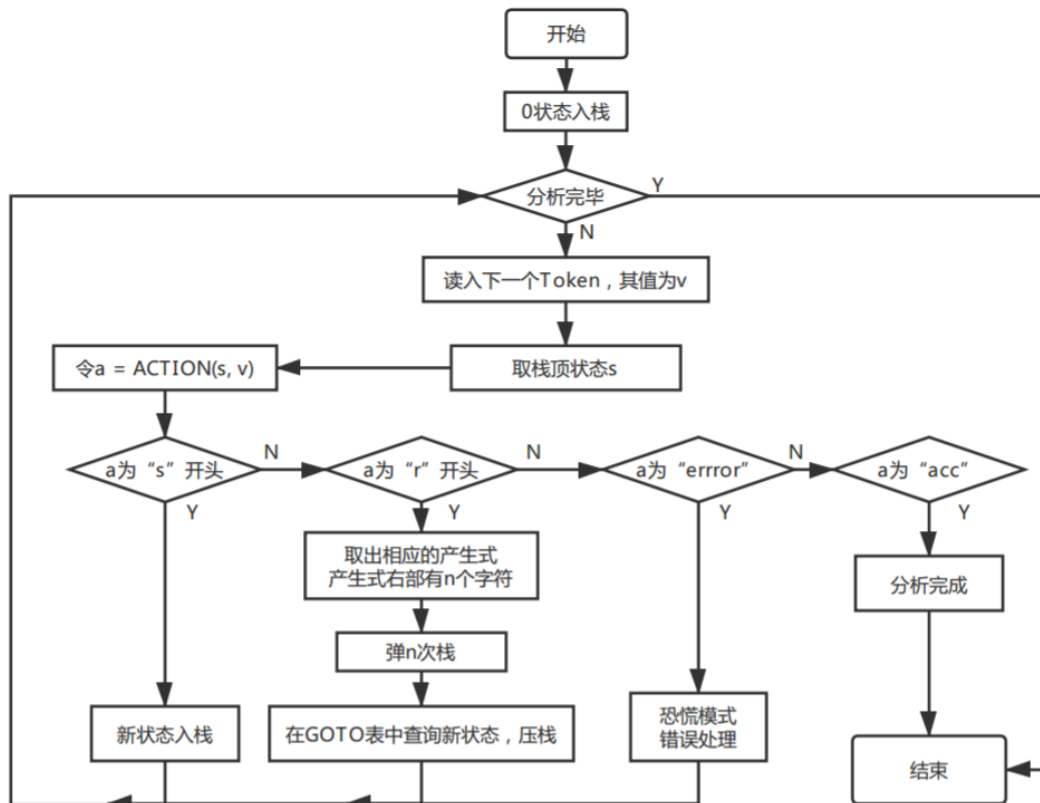
当栈顶状态为  $s$ ，当前输入符号为  $a$ ，查表得  $Action[s, a] = error$ ，这表明语法出错了。

错误处理伪代码如下。

1. 向栈内搜索非终结符  $A$ ，它的归约前状态为  $S$ ，将  $A$  前面的符号全部弹出栈。
2. 不断地读入输入符号，直到读到  $a \in Follow(A)$  集为止，
3. 将  $goto[s, A]$  压入栈

整个过程就相当于完成了一次归约，只不过归约的不是句柄。这样处理完后，就相当于从一个新的句子成分开始分析。

### 3.2.3 程序核心部分的程序流程图



## 4 语法分析系统实现及结果分析

### 4.1 系统实现过程中遇到的问题

#### 1 文法二义性问题

实验指导书给出的文法是具有二义性的, 主要体现在表达式符号的优先级和 if else 结构上, 因此对这两部分进行改进。

此外, 原文法不能确定 while 和 if else 语句的控制范围, 因此对这两部分加上 begin、end 符号以确定控制范围。

改进后的文法已经在第二部分给出, 此处不再赘述。

#### 2 空符号的处理问题

在计算 FIRST 集和规约时, 要对有空 ( $\epsilon$ ) 符号的产生式特殊处理, 这部分很容易产生 bug, 需要不断调试。

#### 3 错误处理问题

错误恢复包括恐慌模式错误恢复和短语层次错误恢复, 本程序采用的是恐慌模式的错误恢复

## 4.2 输出句法分析器的分析表

语法分析表较大，这里仅仅展示小部分。

	!=	(	)	*	+	,	-	/	<	<=	=	==	>	>=	[
0	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
1	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
2	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
3	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
4	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
5	--	--	--	--	--	--	--	s6	--	--	--	--	--	--	--
6	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
7	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
8	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
9	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
10	--	--	--	--	--	--	--	--	--	--	s11	--	--	--	s83
11	--	s18	--	--	--	--	s67	--	--	--	--	--	--	--	--
12	--	--	--	--	s14	--	--	s13	--	--	--	--	--	--	--
13	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
14	--	s18	--	--	--	--	s67	--	--	--	--	--	--	--	--
15	--	--	--	s16	r15	--	--	r15	--	--	--	--	--	--	--
16	--	s18	--	--	--	--	s67	--	--	--	--	--	--	--	--
17	--	--	--	r17	r17	--	--	r17	--	--	--	--	--	--	--
18	--	s25	--	--	--	--	s30	--	--	--	--	--	--	--	--
19	--	--	s20	--	--	s21	--	--	--	--	--	--	--	--	--
20	--	--	--	r19	r19	--	--	r19	--	--	--	--	--	--	--
21	--	s25	--	--	--	--	s30	--	--	--	--	--	--	--	--
22	--	--	r15	s23	r15	--	--	--	--	--	--	--	--	--	--
23	--	s25	--	--	--	--	s30	--	--	--	--	--	--	--	--
24	--	--	r17	r17	r17	--	--	--	--	--	--	--	--	--	--
25	--	s25	--	--	--	--	s30	--	--	--	--	--	--	--	--
26	--	--	s27	--	--	s21	--	--	--	--	--	--	--	--	--
27	--	--	r19	r19	r19	--	--	--	--	--	--	--	--	--	--
28	--	--	r16	s23	r16	--	--	--	--	--	--	--	--	--	--

## 4.3 针对一测试程序输出其句法分析结果

测试样例如下。

```

1.  proc maxminavg;
2.      integer[3] a;
3.      integer max;
4.      integer min;
5.      integer sum;
6.      integer i;
7.      real avg;
8.
9.      a[1]=1;
10.     a[2]=2;
11.     a[3]=3;
12.     sum=0;
13.     max=a[1];
14.     min=a[1];
15.     i=1;
16.
17.     while true do
18.         begin
19.             a=a+1;
20.         end
21.
22.     if a[i]>max then

```

```
23.   begin
24.       max=a[i];
25.   end
26.
27.   if a[i]>max then
28.       begin
29.           max=a[i];
30.       end
31.   else
32.       begin
33.           a=1;
34.       end
35.
36.   while true do
37.       begin
38.
39.           if i<=3 then
40.               begin
41.                   sum=sum+a[i];
42.               end
43.
44.           if a[i]>max then
45.               begin
46.                   max=a[i];
47.               end
48.
49.           if a[i]<min then
50.               begin
51.                   min=a[i];
52.               end
53.           else
54.               begin
55.                   call fuction(b,c,d)
56.               end
57.
58.           i=i+1;
59.           avg=sum*1;
60.       end
61.
62.       a[1]]>=1;;
63.
64.       if i<=3 then
```

```
65.      begin
66.          a=a+1;
67.      end
68.
```

部分语法分析结果如下。

产生式
X -> integer
C -> ε
C -> [ num ] C
T -> X C
D -> T id ;
X -> integer
C -> ε
T -> X C
D -> T id ;
X -> integer
C -> ε
T -> X C
D -> T id ;
X -> integer
C -> ε
T -> X C
D -> T id ;
X -> integer
C -> ε
T -> X C
D -> T id ;

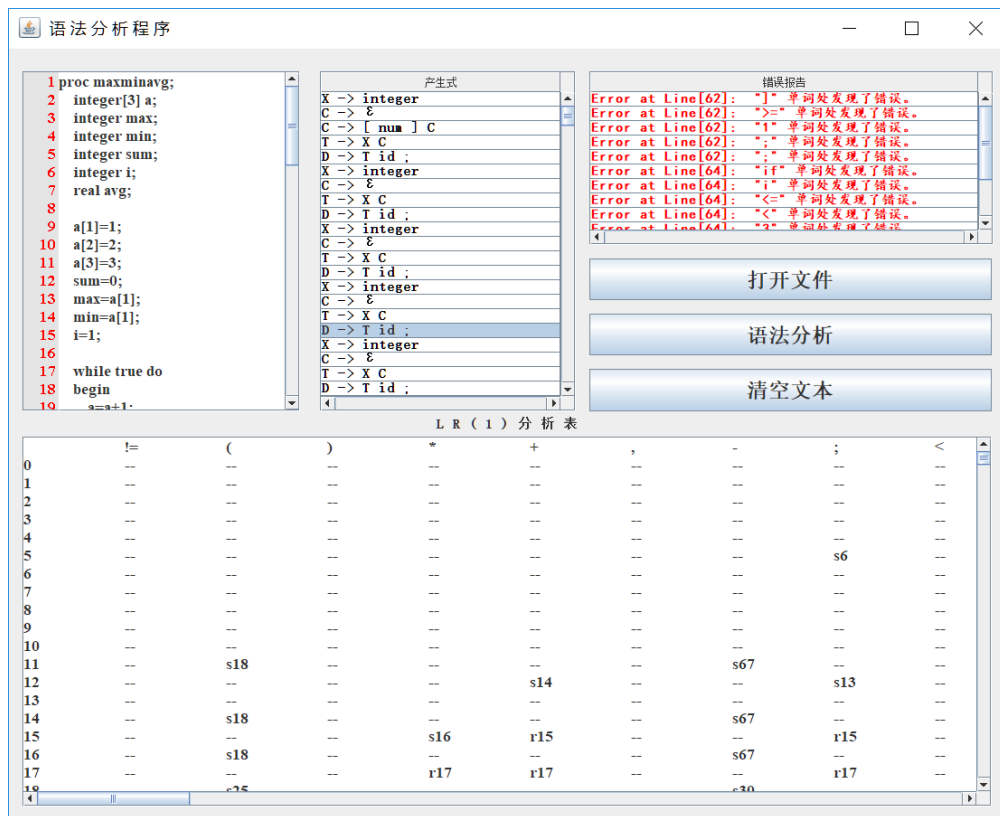
4.4 错误报告

错误报告如下。

错误报告
Error at Line[62]: "]" 单词处发现了错误。
Error at Line[62]: ">=" 单词处发现了错误。
Error at Line[62]: "1" 单词处发现了错误。
Error at Line[62]: ";" 单词处发现了错误。
Error at Line[62]: ";" 单词处发现了错误。
Error at Line[64]: "if" 单词处发现了错误。
Error at Line[64]: "i" 单词处发现了错误。
Error at Line[64]: "<=" 单词处发现了错误。
Error at Line[64]: "<" 单词处发现了错误。
Error at Line[64]: "3" 单词处发现了错误。
Error at Line[64]: "then" 单词处发现了错误。
Error at Line[65]: "begin" 单词处发现了错误。
Error at Line[66]: "a" 单词处发现了错误。
Error at Line[67]: "end" 单词处发现了错误。

4.5 分析实验结果

整体效果如下所示。



经过反复地测试与实验，从最后的实验结果来看，规约的产生式以及次序均正确，例如 ifelse 语句、while 与 ifelse 的互相嵌套、call 语句、record 语句和多维数组等。

同时也对一些常见的错误进行了识别与恢复。例如最常见的“;”后又跟了一个“;”，系统自动忽略后一个分号；操作符不规范，如出现“a>=>b”语句，系统对整个语句忽略掉。

结果表明，系统的结果与目标基本一致，系统目标达成。

## 第三章 语义分析和中间代码生成

### 1 语义分析需求分析

在语法分析器的基础上设计实现类高级语言的语义分析器，**基本功能**如下：

1. 能分析以下几类语句，并生成中间代码（三地址指令和四元式形式）：
  - (1) 声明语句（变量声明）
  - (2) 表达式及赋值语句分支语句：if\_then\_else
  - (3) 循环语句：do\_while
2. 具备语义错误处理能力，包括变量或函数重复声明、变量或函数引用前未声明、运算符和运算分量之间的类型不匹配（如整型变量与数组变量相加减）等错误，能准确给出错误所在位置，并采用可行的错误恢复策略。输出的错误提示信息格式如下：  
Error at Line [行号]: [说明文字]
3. 系统的输入形式：要求能够通过文件导入测试用例。测试用例要涵盖第 1 条中列出的各种类型的语句，以及第 2 条中列出的各种类型的错误。
4. 系统的输出分为两部分：一部分是打印输出符号表。另一部分是打印输出三地址指令或四元式序列。

除此之外，可以实现一些**额外功能**，例如自动类型转换，识别其它类型语义错误，如过程返回类型与声明类型不匹配；过程调用时实参与形参数目或类型不匹配；对非数组型变量使用数组访问操作符“[...]”；对普通变量使用过程调用操作符“call”；数组访问操作符“[...]”中出现非整数等。

### 2 语义分析文法设计

注：验收的时候，由于当时时间比较紧，我没有做嵌套过程中声明语句翻译的实现，之后又补上了这一部分。因此总的来说，该程序可以处理嵌套声明语句、record 语句、数组、算术表达式、布尔表达式、分支“if\_then\_else”语句、循环语句“do\_while”、过程调用语句“call”和类型转换，同时也可以处理各类错误。

下面给出所有文法及其语义动作。

#### 1. 全局定义

- (1)  $P' \rightarrow P$
- (2)  $P \rightarrow \text{proc id ; M0 begin D S end \{addwidth(top(tblptr),top(offset)); pop(tblptr); pop(offset)\}}$



(3)  $P \rightarrow S$

(4)  $S \rightarrow S M S \{ \text{backpatch}(S1.\text{nextlist}, M.\text{quad}); S.\text{nextlist} = S2.\text{nextlist}; \}$

(5)  $M0 \rightarrow \varepsilon \{ t = \text{mktable}(\text{nil}); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}) \}$

(6)  $M \rightarrow \varepsilon \{ M.\text{quad} = \text{nextquad} \}$

## 2. 声明语句（变量、数组、函数、记录声明）

(7)  $D \rightarrow D D$

(8)  $D \rightarrow \text{proc id; } N1 \text{ begin } D \text{ S end } \{ t = \text{top}(\text{tblptr}); \text{addwidth}(t, \text{top}(\text{offset}));$   
 $\text{pop}(\text{tblptr}); \text{pop}(\text{offset}); \text{enterproc}(\text{top}(\text{tblptr}), \text{id.name}, t) \}$

(9)  $D \rightarrow T \text{ id ; } \{ \text{enter}(\text{top}(\text{tblptr}), \text{id.name}, T.\text{type}, \text{top}(\text{offset})); \text{top}(\text{offset}) = \text{top}(\text{offset})$   
 $+ T.\text{width} \}$

(10)  $T \rightarrow X C \{ T.\text{typ} = C.\text{type}; T.\text{width} = C.\text{width}; \}$

(11)  $T \rightarrow \text{record } N2 \text{ D end } \{ T.\text{type} = \text{record}(\text{top}(\text{tblptr})); T.\text{width} = \text{top}(\text{offset});$   
 $\text{pop}(\text{tblptr}); \text{pop}(\text{offset}) \}$

(12)  $X \rightarrow \text{integer} \{ X.\text{type} = \text{integer}; X.\text{width} = 4; t = X.\text{type}; w = X.\text{width}; \}$

(13)  $X \rightarrow \text{real} \{ X.\text{type} = \text{real}; X.\text{width} = 8; t = X.\text{type}; w = X.\text{width}; \}$

(14)  $C \rightarrow [ \text{num} ] C \{ C.\text{type} = \text{array}(\text{num.val}, C1.\text{type}); C.\text{width} = \text{num.val} * C1.\text{width}; \}$

(15)  $C \rightarrow \varepsilon \{ C.\text{type} = t; C.\text{width} = w; \}$

(16)  $N1 \rightarrow \varepsilon \{ t = \text{mktable}(\text{top}(\text{tblptr})); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}) \}$

(17)  $N2 \rightarrow \varepsilon \{ t = \text{mktable}(\text{nil}); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}) \}$

## 3. 表达式及赋值语句（算术表达式、数组）

(18)  $S \rightarrow \text{id} = E ; \{ p = \text{lookup}(\text{id.lexeme}); \text{if } p == \text{nil} \text{ then error; } \text{gencode}(p = 'E.\text{addr});$   
 $S.\text{nextlist} = \text{null}; \}$

(19)  $S \rightarrow L = E ; \{ \text{gencode}(L.\text{array}['L.\text{offset}'] = 'E.\text{addr}); S.\text{nextlist} = \text{null}; \}$

(20)  $E \rightarrow E + E1 \{ E.\text{addr} = \text{newtemp}(); \text{gencode}(E.\text{addr} = 'E.\text{addr}' + 'E1.\text{addr}); \}$

(21)  $E \rightarrow E1 \{ E.\text{addr} = E1.\text{addr} \}$

(22)  $E1 \rightarrow E1 * E2 \{ E1.\text{addr} = \text{newtemp}(); \text{gencode}(E1.\text{addr} = 'E1.\text{addr}' * 'E2.\text{addr}); \}$

(23)  $E1 \rightarrow E2 \{ E1.\text{addr} = E2.\text{addr} \}$

(24)  $E2 \rightarrow ( E ) \{ E2.\text{addr} = E.\text{addr} \}$

(25)  $E2 \rightarrow - E \{ E2.\text{addr} = \text{newtemp}(); \text{gencode}(E2.\text{addr} = \text{"uminus"}E.\text{addr}); \}$

(26)  $E2 \rightarrow \text{id} \{ E2.\text{addr} = \text{lookup}(\text{id.lexeme}); \text{if } E2.\text{addr} == \text{null} \text{ then error}; \}$

(27)  $E2 \rightarrow \text{num} \{ E2.\text{addr} = \text{lookup}(\text{num.lexeme}); \text{if } E2.\text{addr} == \text{null} \text{ then error} \}$

(28)  $E2 \rightarrow L \{ E2.\text{addr} = \text{newtemp}(); \text{gencode}(E2.\text{addr} = 'L.\text{array}['L.\text{offset}']); \}$

(29)  $L \rightarrow \text{id} [ E ] \{ L.\text{array} = \text{lookup}(\text{id.lexeme}); \text{if } L.\text{array} == \text{nil} \text{ then error}; L.\text{type} =$

L.array.type.elem; L.offset = newtemp();  
 gencode(L.offset='E.addr'\*L.type.width);}

- (30)  $L \rightarrow L [ E ]$  {L.array = L1.array; L.type = L1.type.elem; t = newtemp();  
 gencode(t='E.addr'\*L.type.width); L.offset = newtemp();  
 gencode(L.offset='L1.offset'+t);}

#### 4. 布尔表达式语句

- (31)  $B \rightarrow B \text{ or } M B1$  {backpatch(B1.falselist,M.quad); B.truelist =  
 merge(B1.truelist,B2.truelist); B.falselist = B2.falselist}  
 (32)  $B \rightarrow B1$  {B.truelist = B1.truelist; B.falselist = B1.falselist}  
 (33)  $B1 \rightarrow B1 \text{ and } M B2$  {backpatch(B1.truelist,M.quad); B.truelist = B2.truelist;  
 B.falselist = merge(B1.falselist,B2.falselist)}  
 (34)  $B1 \rightarrow B2$  {B.truelist = B1.truelist; B.falselist = B1.falselist}  
 (35)  $B2 \rightarrow \text{not } B$  {B.truelist = B1.falselist; B.falselist = B1.truelist}  
 (36)  $B2 \rightarrow ( B )$  {B.truelist = B1.truelist; B.falselist = B1.falselist}  
 (37)  $B2 \rightarrow E R E$  {B.truelist = makelist(nextquad); B.falselist =  
 makelist(nextquad+1); gencode('if E1.addr relop.opE1.addr 'goto -');  
 gencode('goto -')}  
 (38)  $B2 \rightarrow \text{true}$  {B.truelist = makelist(nextquad); gencode('goto -')}  
 (39)  $B2 \rightarrow \text{false}$  {B.falselist = makelist(nextquad); gencode('goto -')}  
 (40)  $R \rightarrow < | <= | > | >= | == | !=$  {B.name = op}

#### 5. 分支语句 “if\_then\_else” 和循环语句 “do\_while”

- (41)  $S \rightarrow S1$  {S.nextlist = L.nextlist}  
 (42)  $S \rightarrow S2$  {S.nextlist = L.nextlist}  
 (43)  $S1 \rightarrow \text{if } B \text{ then } M S1 N \text{ else } M S1$  {backpatch(B.truelist,M1.quad);  
 backpatch(B.falselist,M2.quad); S.nextlist =  
 merge(S1.nextlist,merge(N.nextlist,S2.nextlist))}  
 (44)  $S1 \rightarrow \text{while } M B \text{ do } M S0$  {backpatch(S1.nextlist,M1.quad);  
 backpatch(B.truelist,M2.quad);S.nextlist = B.falselist; gencode('goto'M1.quad)}  
 (45)  $S2 \rightarrow \text{if } B \text{ then } M S1 N \text{ else } M S2$  {backpatch(B.truelist,M1.quad);  
 backpatch(B.falselist,M2.quad); S.nextlist =  
 merge(S1.nextlist,merge(N.nextlist,S2.nextlist))}  
 (46)  $S2 \rightarrow \text{if } B \text{ then } M S0$  {backpatch(B.truelist,M.quad); S.nextlist =  
 merge(B.falselist,S1.nextlist)}

- (47)  $S_0 \rightarrow \text{begin } S_3 \text{ end } \{S.\text{nextlist} = L.\text{nextlist}\}$   
 (48)  $S_1 \rightarrow \text{begin } S_3 \text{ end } \{S.\text{nextlist} = L.\text{nextlist}\}$   
 (49)  $S_2 \rightarrow \text{begin } S_3 \text{ end } \{S.\text{nextlist} = L.\text{nextlist}\}$   
 (50)  $S_3 \rightarrow S_3 ; M S \{ \text{backpatch}(L_1.\text{nextlist}, M.\text{quad}); L.\text{nextlist} = S.\text{nextlist} \}$   
 (51)  $S_3 \rightarrow S \{S.\text{nextlist} = L.\text{nextlist}\}$   
 (52)  $N \rightarrow \varepsilon \{N.\text{nextlist} = \text{makelist}(\text{nextquad}); \text{gencode}(\text{'goto -'})\}$

## 6. 过程调用语句call

- (53)  $S \rightarrow \text{call id} ( EL ) ; \{n=0; \text{for queue 中的每个 } t \text{ do } \{ \text{gencode}(\text{'param' } t); n = n+1 \}$   
      $\text{gencode}(\text{'call' id.addr, 'n'}); S.\text{nextlist} = \text{null}; \}$   
 (54)  $EL \rightarrow EL, E \{ \text{将 } E.\text{addr} \text{ 添加到 queue 的队尾} \}$   
 (55)  $EL \rightarrow E \{ \text{初始化 queue, 然后将 } E.\text{addr} \text{ 加入到 queue 的队尾} \}$

## 7. 下面再另行给出类型转换语句的语义动作

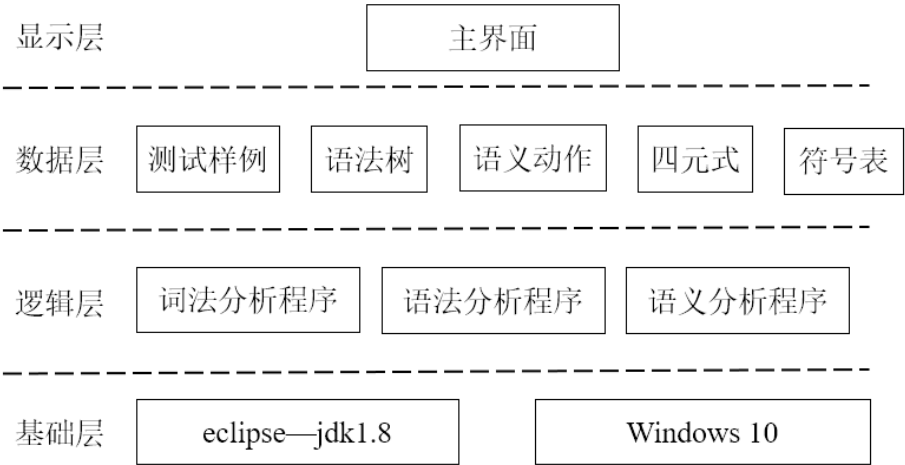
```

E → E1 + E2
{
    E.addr = newtemp
    if E1.type = integer and E2.type = integer then begin
        gencode(E.addr=E1.addr int+E2.addr);
        E.type = integer
    end
    else if E1.type = integer and E2.type = real then begin
        u = newtemp;
        gencode(u=inttoreal E1.addr);
        gencode(E.addr=u real+ E2.addr);
        E.type = real
    end
}
  
```

### 3 语义分析系统设计

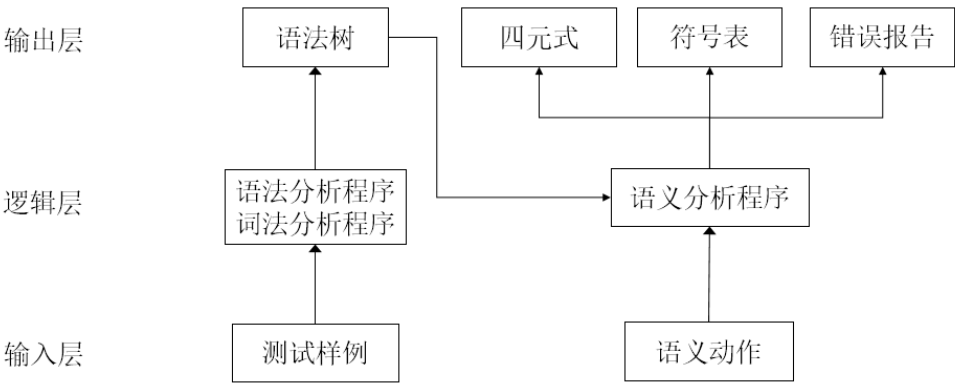
#### 3.1 系统概要设计

##### 3.1.1 系统框架图



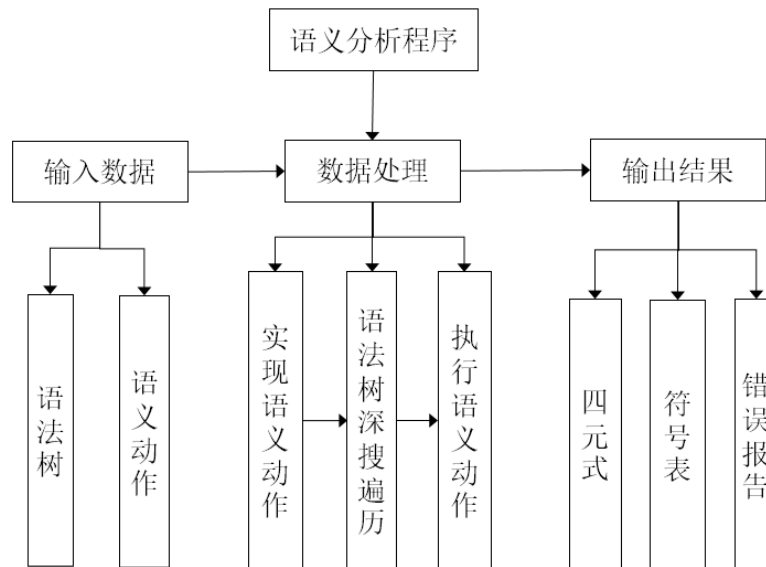
系统分为基础层、逻辑层、数据层和显示层，基础层即在 Windows10、jdk1.8 环境下利用 java 语言进行编程，逻辑层即词法分析程序、语法分析程序和语义分析程序，数据层包含测试样例、语法树、语义动作、四元式和符号表，显示层显示一个主界面。

##### 3.1.2 数据流图



首先对测试样例进行词法分析得到 Token 序列，然后再进行语法分析得到一颗语法树，最后根据语义动作进行语义分析，得到四元式、符号表和错误报告。

### 3.1.3 功能模块图



如上图所示，语义分析程序的功能主要为读入数据、分析数据和给出结果。对于数据处理模块，首先要实现各个语义动作，进而对语法树进行深搜遍历，并执行相应节点的语义动作，直到分析完成。

## 3.2 系统详细设计

### 3.2.1 核心数据结构的设计

#### g. TreeNode 类

该类用于构造语法树的每一个节点，具体的数据结构如下所示。

```

public class TreeNode
{
    private int id; // 节点编号，用以构造一个邻接表时，链接节点
    private String symbol; // 符号类型
    private String value; // 符号值
    private int line; // 所在行数

    /**
     * 语法树每一个节点的构造函数
     * @param id 节点编号，用以构造一个邻接表时，链接节点
     * @param symbol 符号类型
     * @param value 符号值
     * @param line 所在行数
     */
    public TreeNode(int id, String symbol, String value, int line)
  
```

**h. Tree 类**

语法树以邻接表形式存储，该类用于构造邻接表的每一行，具体的数据结构如下所示。

```
public class Tree
{
    private TreeNode father; // 父节点
    private ArrayList<TreeNode> children; // 孩子列表

    /**
     * 语法树的构造函数
     * @param father 父节点
     * @param children 孩子列表
     */
    public Tree(TreeNode father, ArrayList<TreeNode> children)
```

**i. Symbol 类**

该类用于构造符号表中的每一个元素，具体的数据结构如下所示。

```
public class Symbol
{
    private String name; // 符号名
    private String type; // 符号类型
    private int offset; // 偏移量

    /**
     * 符号表中每一个符号的构造函数
     * @param name 符号名
     * @param type 符号类型
     * @param offset 偏移量
     */
    public Symbol(String name, String type, int offset)
```

**j. FourAddr 类**

该类用于构造四元式列表中的每一个元素，具体的数据结构如下所示。

```
public class FourAddr
{
    private String op; // 操作符
    private String param1; // 参数一
    private String param2; // 参数二
    private String toaddr; // 地址

    /**
     * 四元式构造函数
     * @param op 操作符
     * @param param1 参数一
     * @param param2 参数二
     * @param toaddr 地址
     */
    public FourAddr(String op, String param1, String param2, String toaddr)
```

### k. Array 类

该类用于构造一个多维数组，具体的数据结构如下所示。

```
public class Array
{
    // 以"array(2,array(2,integer))"为例
    private int length; // 长度: 2
    private Array type; // 数组类型: array(2,integer)
    private String baseType; // 基本类型: integer
}
```

### l. Properties 类

该类表示语法树上每一个节点的属性，各个属性是根据语义动作得来的，在实际实现过程中，某些属性可能为 null，但这并不妨碍程序的运行。

具体的数据结构如下所示。

```
public class Properties
{
    private String name; // 变量或者函数的name
    private String type; // 节点类型
    private String offset; // 数组类型的属性
    private int width; // 类型大小属性

    private Array array; // 数组类型属性

    private String addr; // 表达式类型的属性

    private int quad; // 回填用到的属性,指令位置
    private List<Integer> truelist = new ArrayList<Integer>(); // 列表
    private List<Integer> falselist = new ArrayList<Integer>(); // 列表
    private List<Integer> nextlist = new ArrayList<Integer>(); // 列表
}
```

### m. Smantic

该类是程序的核心部分，依据前面所述的数据结构实现了所有语义动作，并对语法树深搜遍历，用到的数据结构如下所示。

```
public class Smantic
{
    private static ArrayList<Tree> tree = new ArrayList<Tree>(); // 语法树
    static List<Properties> tree_pro; // 语法树节点属性

    static List<Stack<Symbol>> table = new ArrayList<Stack<Symbol>>(); // 符号表
    static List<Integer> tablesizesize = new ArrayList<Integer>(); // 记录各个符号表大小

    static List<String> three_addr = new ArrayList<String>(); // 三地址指令序列
    static List<FourAddr> four_addr = new ArrayList<FourAddr>(); // 四元式指令序列
    static List<String> errors = new ArrayList<String>(); // 错误报告序列

    static String t; // 类型
    static int w; // 大小
    static int offset; // 偏移量
    static int temp_cnt = 0; // 新建变量计数
    static int nextquad = 1; // 指令位置
}
```

```

static List<String> queue = new ArrayList<String>(); // 过程调用参数队列
static Stack<Integer> tblptr = new Stack<Integer>(); // 符号表指针栈
static Stack<Integer> off = new Stack<Integer>(); // 符号表偏移大小栈

static int nodeSize; // 语法树上的节点数
static int treeSize; // 语法树大小
static int initial = nextquad; // 记录第一条指令的位置
public Smantic(String filename, List<Stack<Symbol>> table,
    List<String> three_addr, List<FourAddr> four_addr, List<String> errors)

```

### 3.2.2 主要功能函数说明

#### a. 首先给出实现语义动作要用到的一些函数

```

/**
 * 向符号表中增加元素
 * @param i 第i个符号表
 * @param name 元素名字
 * @param type 元素类型
 * @param offset 偏移量
 */
private static void enter(int i, String name, String type, int offset)
{
    if(table.size()==0)
    {
        table.add(new Stack<Symbol>());
    }
    Symbol s = new Symbol(name,type,offset);
    table.get(i).push(s);
}

/**
 * 查找符号表，查看变量是否存在
 * @param s 名字
 * @return 该名字在符号表中的位置
 */
private static int[] lookup(String s)
{
    int[] a = new int[2];
    for (int i=0; i<table.size(); i++)
    {
        for (int j=0; j<table.get(i).size(); j++)
        {
            if(table.get(i).get(j).getName().equals(s))
            {
                a[0] = i;
                a[1] = j;
                return a;
            }
        }
    }
    a[0] = -1;
    a[1] = -1;
    return a;
}

```



```
/**
 * 新建一个变量
 * @return 新建变量名
 */
private static String newtemp()
{
    return "t" + (++temp_cnt);
}

/**
 * 回填地址
 * @param list 需要回填的指令序列
 * @param quad 回填的地址
 */
private static void backpatch(List<Integer> list, int quad)
{
    for(int i=0; i<list.size(); i++)
    {
        int x = list.get(i) - initial;
        three_addr.set(x, three_addr.get(x)+quad);
        four_addr.get(x).setToaddr(String.valueOf(quad));
    }
}

/**
 * 合并列表
 * @param a 列表
 * @param b 列表
 * @return a与b合并后的列表
 */
private static List<Integer> merge(List<Integer> a, List<Integer> b)
{
    List<Integer> a1 = a;
    a1.addAll(b);
    return a1;
}

/**
 * 返回下一条指令地址
 * @return 下一条指令地址
 */
private static int nextquad()
{
    return three_addr.size() + nextquad;
}

/**
 * 新建包含i的列表并返回
 * @param i
 * @return 列表
 */
private static List<Integer> makelist(int i)
{
    List<Integer> a1 = new ArrayList<Integer>();
    a1.add(i);
    return a1;
}
```

```

/**
 * 新增一个符号表
 */
public static void mktable()
{
    table.add(new Stack<Symbol>());
}

```

## b. 语义动作函数

当前述的各个模块构建好之后，语义动作函数的实现就已经变得非常简单，只要把语义动作转换成程序语言即可。

由于语义动作较多，而且实际上只要了解了几个代表性语义动作的实现方法，其余实现方法也就能举一反三，因此在此仅给出几个有代表性语义动作的实现方法，其余的不再赘述。

首先看一下变量声明部分。

```

// S -> S1 M S2 {backpatch(S1.nextlist,M.quad); S.nextlist=S2.nextlist;}
public static void semantic_3(Tree tree)
{
    int S = tree.getFather().getId(); // S
    int S1 = tree.getChildren().get(0).getId(); // S1
    int M = tree.getChildren().get(1).getId(); // M
    int S2 = tree.getChildren().get(2).getId(); // S2

    backpatch(tree_pro.get(S1).getNext(), tree_pro.get(M).getQuad());

    Properties a1 = new Properties();
    a1.setNext(tree_pro.get(S2).getNext());
    tree_pro.set(S,a1);
}

// X -> integer {X.type=integer; X.width=4;}{t=X.type; w=X.width;}
public static void semantic_8(Tree tree)
{
    int X = tree.getFather().getId(); // X
    t = "integer";
    w = 4;

    Properties a1 = new Properties();
    a1.setType("integer");
    a1.setWidth(4);
    tree_pro.set(X,a1);
}

```

```
// D -> T id ; {enter(top(tblptr),id.name,T.type,top(offset));
//                top(offset) = top(offset)+T.width}
public static void semantic_5(Tree tree)
{
    int T = tree.getChildren().get(0).getId(); // T
    String id = tree.getChildren().get(1).getValue(); // id

    int[] i = lookup(id);
    if (i[0] == -1)
    {
        enter(tblptr.peek(), id, tree_pro.get(T).getType(), off.peek());
        int s = off.pop();
        off.push(s + tree_pro.get(T).getWidth());
        offset = offset + tree_pro.get(T).getWidth();
    }
    else
    {
        String s = "Error at Line [" + tree.getChildren().get(1).getLine() +
            "]:\t[" + "变量" + id + "重复声明]" ;
        errors.add(s);
    }
}
```

算术表达式部分如下。

```
// E -> E1 * E2 {E.addr=newtemp(); gencode(E.addr='E1.addr'*E2.addr);}
public static void semantic_16(Tree tree)
{
    int E = tree.getFather().getId(); // E
    int E1 = tree.getChildren().get(0).getId(); // E1
    int E2 = tree.getChildren().get(2).getId(); // E2
    String newtemp = newtemp();

    Properties a1 = new Properties();
    a1.setAddr(newtemp);
    tree_pro.set(E,a1);

    String code = newtemp + " = " + tree_pro.get(E1).getAddr() +
        "*" + tree_pro.get(E2).getAddr();
    three_addr.add(code);
    four_addr.add(new FourAddr("=",tree_pro.get(E1).getAddr(),
        tree_pro.get(E2).getAddr(),newtemp));
}
```

布尔表达式部分如下。

```

// B -> B1 or M B2 {backpatch(B1.falselist,M.quad);
//                  B.truelist=merge(B1.truelist,B2.truelist);
//                  B.falselist=B2.falselist}
public static void semantic_25(Tree tree)
{
    int B = tree.getFather().getId(); // B
    int B1 = tree.getChildren().get(0).getId(); // B1
    int M = tree.getChildren().get(2).getId(); // M
    int B2 = tree.getChildren().get(3).getId(); // B2

    backpatch(tree_pro.get(B1).getFalse(),tree_pro.get(M).getQuad());

    Properties a1 = new Properties();
    a1.setTrue(merge(tree_pro.get(B1).getTrue(),tree_pro.get(B2).getTrue()));
    a1.setFalse(tree_pro.get(B2).getFalse());
    tree_pro.set(B,a1);
}

```

控制流语句部分如下。

```

// S -> if B then M1 S1 N else M2 S2
// {backpatch(B.truelist, M1.quad); backpatch(B.falselist,M2.quad);
// S.nextlist=merge(S1.nextlist,merge(N.nextlist, S2.nextlist))}
public static void semantic_42_44(Tree tree)
{
    int S = tree.getFather().getId(); // S
    int B = tree.getChildren().get(1).getId(); // B
    int M1 = tree.getChildren().get(3).getId(); // M1
    int S1 = tree.getChildren().get(4).getId(); // S1
    int N = tree.getChildren().get(5).getId(); // N
    int M2 = tree.getChildren().get(7).getId(); // M2
    int S2 = tree.getChildren().get(8).getId(); // S2

    backpatch(tree_pro.get(B).getTrue(), tree_pro.get(M1).getQuad());
    backpatch(tree_pro.get(B).getFalse(), tree_pro.get(M2).getQuad());
    Properties a1 = new Properties();
    a1.setNext(merge(tree_pro.get(S1).getNext(),
        merge(tree_pro.get(N).getNext(), tree_pro.get(S2).getNext())));
    tree_pro.set(S,a1);
}

```

函数调用部分如下。

```

// S -> call id ( EL )
// {n=0; for queue中的每个t do {gencode('param't); n=n+1}
//   gencode('call'id.addr','n');}
public static void semantic_54(Tree tree)
{
    int S = tree.getFather().getId(); // S
    String id = tree.getChildren().get(1).getValue(); // id
    int[] index = Lookup(id);

    if (!table.get(index[0]).get(index[1]).getType().equals("函数"))
    {
        String s = "Error at Line [" + tree.getChildren().get(0).getLine()
            + "]:\t[" + id + "不是函数,不能用于call语句]" ;
        errors.add(s);
        Properties a1 = new Properties();
        a1.setNext(new ArrayList<Integer>());
        tree_pro.set(S,a1);
        return;
    }

    int size = queue.size();
    for (int i=0; i<size; i++)
    {
        String code = "param " + queue.get(i);
        three_addr.add(code);
        four_addr.add(new FourAddr("param","-","-",queue.get(i)));
    }
    String code = "call " + id + " " + size;
    three_addr.add(code);
    four_addr.add(new FourAddr("call",String.valueOf(size),"-",id));

    Properties a1 = new Properties();
    a1.setNext(new ArrayList<Integer>());
    tree_pro.set(S,a1);
}

```

函数嵌套部分如下。

```

// D -> proc id; N1 D S
// {t=top(tblptr); addwidth(t, top(offset));
//   pop(tblptr); pop(offset); enterproc(top(tblptr), id.name,t)}
public static void semantic_57(Tree tree)
{
    String id = tree.getChildren().get(1).getValue();
    int t = tblptr.peak();
    //tablesize.add(off.peak());
    tblptr.pop();
    off.pop();

    enter(tblptr.peak(), id, "函数", t);
}

```

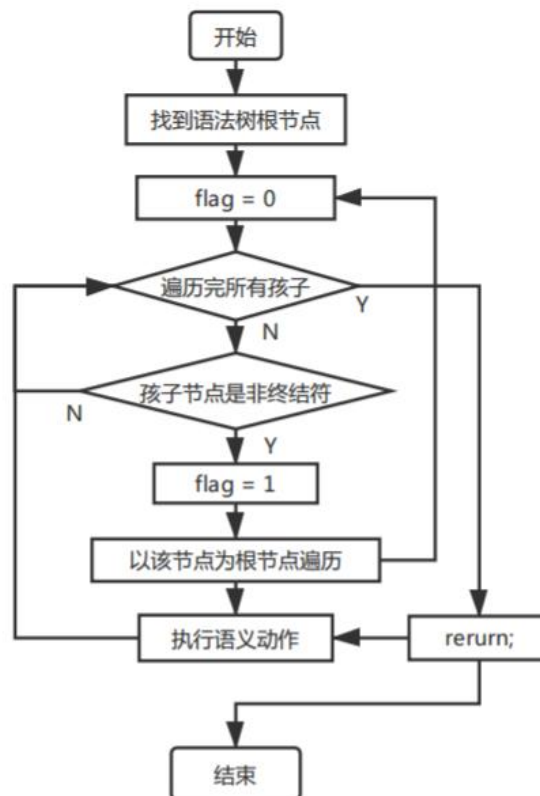
### c. 深度优先搜索

采用深度优先搜索遍历语法树，对每个节点执行相应语义动作，直到分析完成。此时，语义分析也就完成了。

深度优先搜索采用递归实现，代码如下。

```
/**
 * 深搜遍历语法树
 * @param tree 语法树根节点
 */
public static void dfs(Tree tree)
{
    int flag = 0;
    for(int i=0; i<tree.getChildren().size(); i++)
    {
        TreeNode tn = tree.getChildren().get(i);
        if (!util.endPoint(tn)) // 非终结符
        {
            flag = 1;
            // 找到邻接表的下一节点
            Tree f = findTreeNode(tn.getId());
            dfs(f); // 递归遍历孩子节点
            findSemantic(f); // 查找相应的语义动作函数
        }
    }
    if (flag == 0)
    {
        return;
    }
}
```

### 3.2.3 程序核心部分的程序流程图



## 4 语义分析系统实现及结果分析

### 4.1 系统实现过程中遇到的问题

#### (1) 语法树

语义分析实际上也可以不建语法树，直接按产生式序列遍历即可。但从做实验的角度，考虑到语法分析的结果本应输出一个语法树，又考虑到尽量保持语法和语义之间的高内聚、低耦合特性，因此决定用语法树来做。此外，语法树也让程序看起来更清晰，更易扩展。

#### (2) 回填

对于分支语句和循环语句，如果想一趟扫描就完成语法分析，就需要回填技术。当生成一个跳转指令时，暂时不指定该跳转指令的目标标号，这样的指令都被放入由跳转指令组成的列表中，同一个列表中的所有跳转指令具有相同的目标标号。等到能够确定正确的目标标号时，才去填充这些指令的目标标号。

#### (3) 嵌套过程中声明语句的翻译

我认为这部分可能是语义动作中比较难的一部分。实现时，我在原来的基础上将符号表外层又加了一层 `List<>` 型，用以表示不同的符号表，这样不同符号表之间用其在 `List` 中的位置来区分即可。

#### (4) 错误处理

对于大部分错误，处理策略是忽略掉该错误，继续进行。但是有的错误如果忽略掉，可能会影响后续的处理，导致后面抛 `NullPointerException`。因此将这些错误节点的属性自定义一个合理的值，以防止后续处理出错。

### 4.2 针对一测试程序输出其语义分析结果

测试样例如下。

```
1. proc fuction1;
2.   begin
3.     integer f11;
4.     real f12;
5.
6.     proc fuction2;
7.       begin
8.         integer[7][6] arr;
9.         integer m;
10.        integer n;
11.        integer a;
```

```
12.      integer b;
13.      integer c;
14.      integer d;
15.      real e;
16.      record real re1; integer re2; end r1; // 记录
17.
18.      integer x;
19.      integer y;
20.      integer z;
21.      integer z; // Duplicate definition
22.
23.
24.      while a<b do
25.          begin
26.              if c<d then
27.                  begin
28.                      x=y+z;
29.                  end
30.              else
31.                  begin
32.                      x=y*z;
33.                  end
34.              end
35.
36.      while a<b do
37.          begin
38.              if c<5 then
39.                  begin
40.                      while x>y do
41.                          begin
42.                              z=x+1;
43.                          end
44.                      end
45.                  else
46.                      begin
47.                          x=y;
48.                      end
49.                  end
50.
51.
52.      arr[3][5] = 2;
53.      m=(m+n)*9;
```



```

54.
55.
56.      e = e + a; // real = real + int , Type conversion
57.
58.      call a(1,2+1,a*b); //Common variable with call
59.      e1 = 7; // e1 Undefined
60.      a = e2; // e2 Undefined
61.      a[0] = 1; // Non-array using array operators
62.      e3[9] = 1; // e3 Undefined
63.      a = a + arr; // int = int + array
64.      // Integer variables are added to array variables
65.
66.      end
67.      f11 = 1;
68.      call fuction2(1,2+1,a*b);
69. end

```

四元式和三地址指令如下。

1	(j<, a, b, 3)	if a<b goto 3
2	(j, -, -, 11)	goto 11
3	(j<, c, d, 5)	if c<d goto 5
4	(j, -, -, 8)	goto 8
5	(+, y, z, t1)	t1 = y+z
6	(=, t1, -, x)	x = t1
7	(j, -, -, 1)	goto 1
8	(*, y, z, t2)	t2 = y*z
9	(=, t2, -, x)	x = t2
10	(j, -, -, 1)	goto 1
11	(j<, a, b, 13)	if a<b goto 13
12	(j, -, -, 23)	goto 23
13	(j<, c, 5, 15)	if c<5 goto 15
14	(j, -, -, 21)	goto 21
15	(j>, x, y, 17)	if x>y goto 17
16	(j, -, -, 11)	goto 11
17	(+, x, 1, t3)	t3 = x+1
18	(=, t3, -, z)	z = t3
19	(j, -, -, 15)	goto 15

20	(j, -, -, 11)	goto 11
21	(=, y, -, x)	x = y
22	(j, -, -, 11)	goto 11
23	(*, 3, 6, t4)	t4 = 3*6
24	(*, 5, 4, t5)	t5 = 5*4
25	(+, t4, t5, t6)	t6 = t4+t5
26	(=, 2, -, arr[t6])	arr[t6] = 2
27	(+, m, n, t7)	t7 = m+n
28	(*, t7, 9, t8)	t8 = t7*9
29	(=, t8, -, m)	m = t8
30	(=, intTOreal, -, t9)	t9 = intTOreal a
31	(+, e, t9, t10)	t10 = e+t9
32	(=, t10, -, e)	e = t10
33	(+, 2, 1, t11)	t11 = 2+1
34	(*, a, b, t12)	t12 = a*b
35	(=, 7, -, e1)	e1 = 7
36	(=, e2, -, a)	a = e2
37	(=, 0, -, t13)	t13 = 0
38	(=, 1, -, a[t13])	a[t13] = 1
39	(=, 4, -, t14)	t14 = 4
40	(=, 1, -, e3[t14])	e3[t14] = 1
41	(=, 7, -, t15)	t15 = 7
42	(+, a, t15, t16)	t16 = a+t15
43	(=, t16, -, a)	a = t16
44	(=, 1, -, f11)	f11 = 1
45	(+, 2, 1, t17)	t17 = 2+1
46	(*, a, b, t18)	t18 = a*b
47	(param, -, -, 1)	param 1
48	(param, -, -, t17)	param t17
49	(param, -, -, t18)	param t18
50	(call, 3, -, fuction2)	call fuction2

从三地址和四元式指令中可以看出，程序对控制流语句、多维数组、各种表达式语句、过程调用语句和类型转换都能处理得很好。

### 4.3 语义分析后的符号表

针对此测试样例语义分析后的符号表如下所示。

表号	Name	Type	Offset
0	f11	integer	0
0	f12	real	4
0	fuction2	函数	1 (表号)
1	arr	array(7,array(6,integer))	0
1	m	integer	168
1	n	integer	172
1	a	integer	176
1	b	integer	180
1	c	integer	184
1	d	integer	188
1	e	real	192
1	r1	record	200
1	x	integer	212
1	y	integer	216
1	z	integer	220
1	e1	integer	248
1	e2	integer	252
2	re1	real	0
2	re2	integer	8

从符号表中可以看出，程序对嵌套声明语句、多维数组、基本类型和 record 语句都能处理得很好。

例如，上面表中第 0 个表的“fuction2”表示函数名，其“offset”为 1，表示其链接到第 1 个符号表。第 1 个表的“arr”表示数组“array(7,array(6,integer))”，其大小为  $4 \times 6 \times 7 = 168$ ，表中也成功计算出来。第 1 个表的“r1”表示 record 类型，其有两个基本类型变量“re1”和“re2”，存储在第 2 个表之中。

### 4.4 语义错误报告

针对此测试样例语义分析后语义错误报告如下所示。

Error at Line [21]: [变量 z 重复声明]

Error at Line [58]: [a 不是函数,不能用于 call 语句]

Error at Line [59]: [变量 e1 引用前未声明]

Error at Line [60]: [变量 e2 引用前未声明]

Error at Line [61]: [非数组变量 a 访问数组]

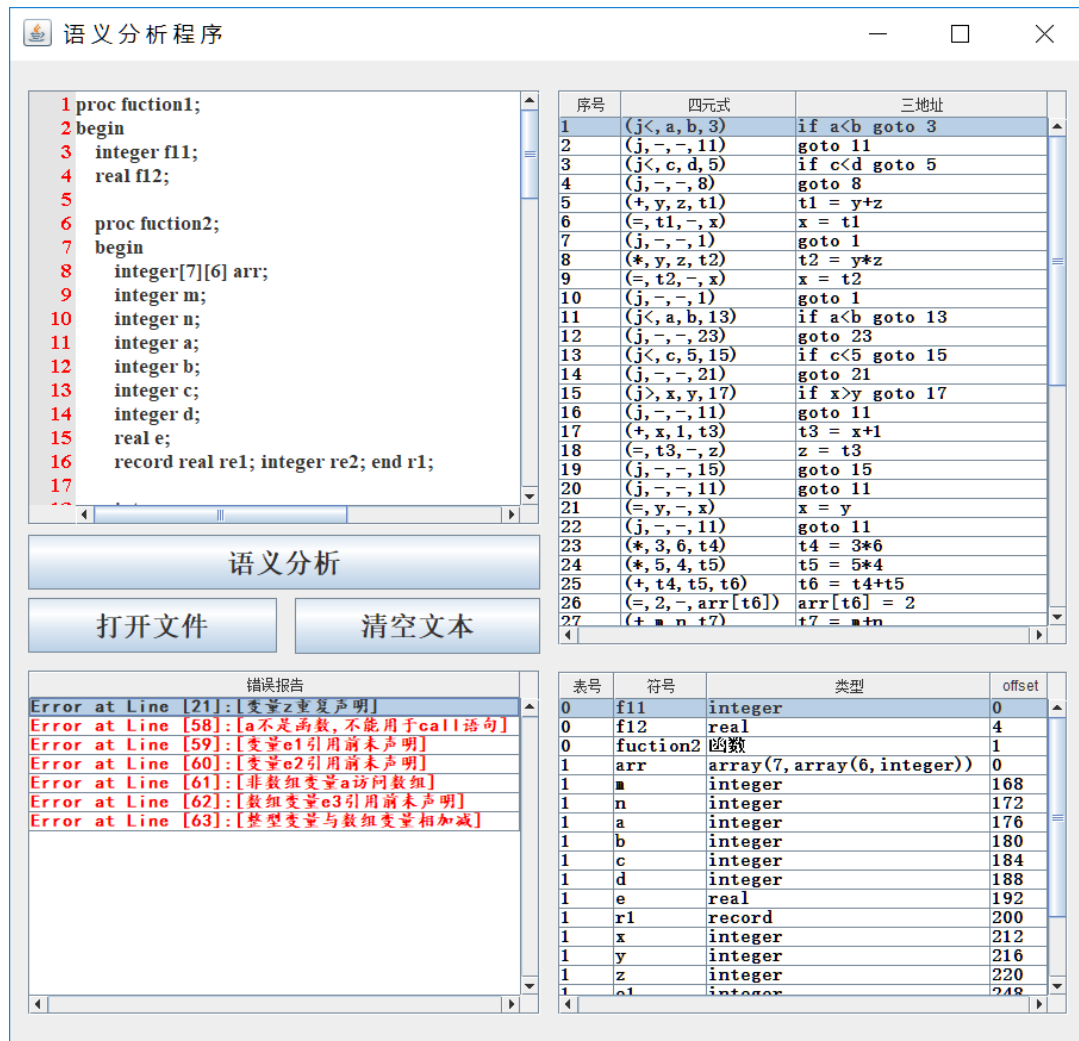
Error at Line [62]: [数组变量 e3 引用前未声明]

Error at Line [63]: [整型变量与数组变量相加减]

从错误报告可以看出，程序可以完成对变量重复声明、变量引用前未声明、运算符和运算分量之间的类型不匹配（如整型变量与数组变量相加减）、非数组型变量使用数组访问操作符“[...]”、普通变量使用过程调用操作符“call”几类错误都处理得很好。

## 4.5 分析实验结果

主界面如下所示。



经过反复地测试与实验，从最后的实验结果来看，四元式、三地址指令以及符号表都能正确显示，同时也对一些常见的错误进行了处理。具体分析已经在4.2-4.4部分陈述，这里不再赘述。

总体而言，系统的结果与目标基本一致，系统目标达成。

## 第四章 代码优化

### 1 代码优化需求分析

代码优化就是为了提高目标程序的效率，对程序进行等价变换，亦即在保持功能不变的前提下，对源程序进行合理的变换，使得目标代码具有更高的时间效率和/或空间效率。空间效率和时间效率有时是一对矛盾，有时不能兼顾。

优化的原则：等价原则，经过优化后不应该改变程序运行的结果。等效原则，使优化后所产生的目标代码运行时间较短，占用的储存空间较小。合算原则，应尽可能以较低的代价取得较好的优化效果。

本部分要求设计一个代码优化器。

### 2 优化过程

通常只对中间代码进行优化，包括控制流分析、数据流分析和代码变换三部分。

以程序的基本块为基础，基本块内的优化叫局部优化，跨基本块的优化为全局优化，循环优化是针对循环进行的优化，是全局优化的一部分。

公共子表达式的删除、复制传播、无用代码删除、代码外提、强度削弱和归纳变量删除等都是常用的针对局部或者全局的代码优化方法。

#### 2.1 常见的代码优化手段

常见的代码优化技术有：删除多余运算，代码外提，强度削弱，合并已知量，复制传播，删除无用赋值等。

#### 2.2 基本块内的局部优化

##### 1. 基本块的划分

入口语句的定义如下：

- (1) 程序的第一个语句；或者，
- (2) 条件转移语句或无条件转移语句的转移目标语句；
- (3) 紧跟在条件转移语句后面的语句。

有了入口语句的概念之后，就可以给出划分中间代码（四元式程序）为基本块的算法，其步骤如下：

- (1) 求出四元式程序中各个基本块的入口语句。
- (2) 对每一入口语句，构造其所属的基本块。它是由该入口语句到下一入口语句（不包括下一入口语句），或到一转移语句（包括该转移语句），或到一停语

句（包括该停语句）之间的语句序列组成的。

- (3) 凡未被纳入某一基本块的语句、都是程序中控制流程无法到达的语句，因而也是不会被执行到的语句，可以把它们删除。

## 2. 基本块的优化手段

由于基本块内的逻辑清晰，故而要做的优化手段都是较为直接浅层次的。目前基本块内的常见的块内优化手段有：

- (1) 删除公共子表达式
- (2) 删除无用代码
- (3) 重新命名临时变量（一般是用来应对创建过多临时变量的，如  $t2 = t1 + 3$  如果后续并没有对  $t1$  的引用，则可以  $t1 = t1 + 3$  来节省一个临时变量的创建）
- (4) 交换语句顺序
- (5) 在结果不变的前提下，更换代数操作（如  $x = y^2$  是需要根重载指数函数的，这是挺耗时的操作，故而可以用强度更低的  $x = y * y$  来代替）

## 3. DAG 应用于基本块的优化

在 DAG 图中，通过节点间的连线和层次关系来表示表示式或运算的归属关系。

图的叶结点：即无后继的结点，以一标识符（变量名）或常数作为标记，表示这个结点代表该变量或常数的值。如果叶结点用来代表某变量  $A$  的地址，则用  $\text{addr}(A)$  作为这个结点的标记。

图的内部结点：即有后继的结点，以一运算符作为标记，表示这个结点代表应用该运算符对其后继结点所代表的值进行运算的结果。

通过 DAG 图可以发现诸多的优化信息，如重复定义、无用定义等，则根据上图的 DAG 图可以构建最后的优化代码序列

### 2.3 循环优化

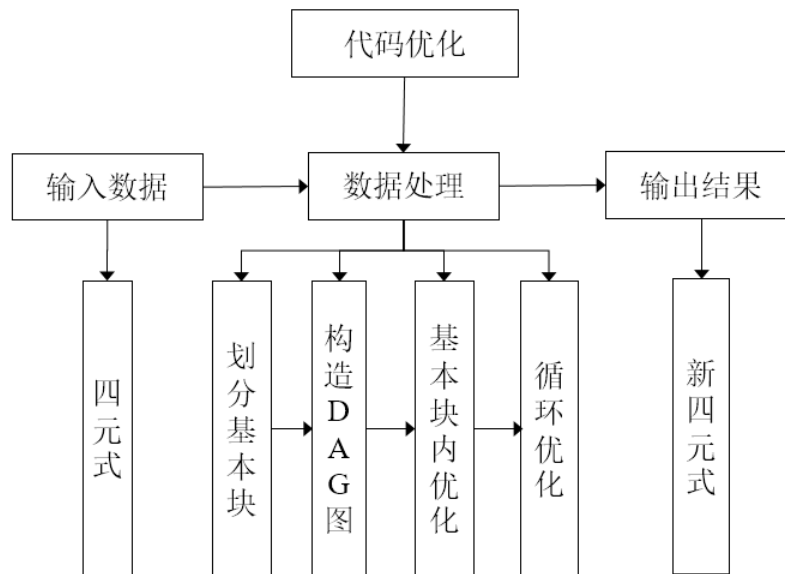
根据上面基本块的定义，我们将诸多基本块组装在一起，构建成程序循环图。

循环块最主要的特点是只有一个数据流和控制流入口，而出口可能有多个。循环优化的主要手段有：循环次数无关性代码外提、删除归纳变量和运算强度削弱。

## 3 代码优化系统设计

### 3.1 系统逻辑

测试样例经过词法、语法和语义分析后，得到四元式序列，代码优化器需要将得到的四元式进行优化。



如上图所示，代码优化分为输入数据、数据处理和输出结果三个模块。在数据处理阶段，首先要划分基本块，构造出 DAG 图，对基本块内的四元式运用各种优化方法进行优化，然后再进行循环优化。

### 3.2 系统具体实现

限于时间关系，只实现了基本块的划分和公共子表达式的删除。

1. 对于基本块的划分，首先求出基本块的起始点列表。

```

/**
 * 求出基本块的起始点列表
 * @param list 四元式序列
 * @return 基本块的起始点列表
 */
public static List<Integer> indexBlock(List<FourAddr> list)
{
    Set<Integer> index = new HashSet<Integer>();
    index.add(0);

    for(int i = 0; i < list.size(); i++)
    {
        String s = list.get(i).getOp();
        if (s.contains("j"))
        {
            int n = Integer.valueOf(list.get(i).getToaddr());
            index.add(n-1);
            index.add(i+1);
        }
    }

    List<Integer> result = new ArrayList<Integer>(index);
    Collections.sort(result);
    return result;
}

```

然后再根据列表进行划分。

```
/**
 * 划分结果
 * @param list 四元式序列
 * @return 划分结果
 */
public static List<List<FourAddr>> basicBlock(List<FourAddr> list)
{
    List<List<FourAddr>> result = new ArrayList<List<FourAddr>>();
    List<FourAddr> block = new ArrayList<FourAddr>();
    List<Integer> index = indexBlock(list);
    //System.out.println(index);
    int num = 1;
    int n = index.get(num);

    for(int i = 0; i<list.size(); i++)
    {
        block.add(list.get(i));
        if (n-1 == i)
        {
            result.add(block);
            block = new ArrayList<FourAddr>();
            num++;
            if (num < index.size())
            {
                n = index.get(num);
            }
            else
            {
                n = list.size();
            }
        }
    }
    return result;
    //return re;
}
```

2. 对于基本块内公共子表达式的删除，代码如下。

```
/**
 * 删除公共子表达式
 * @param result 删除公共子表达式后的基本块
 * @return 删除公共子表达式后的基本块
 */
public static List<List<FourAddr>> sub_expression(List<List<FourAddr>> result)
{
    for (int i=0; i<result.size(); i++) // 每个基本块
    {
        for (int j=0; j<result.get(i).size(); j++) // 基本块内每个四元式
        {
            FourAddr f = result.get(i).get(j);
            boolean flag = false;

            Set<String> par = new HashSet<String>(); // 公共子表达式左部集合
```



```

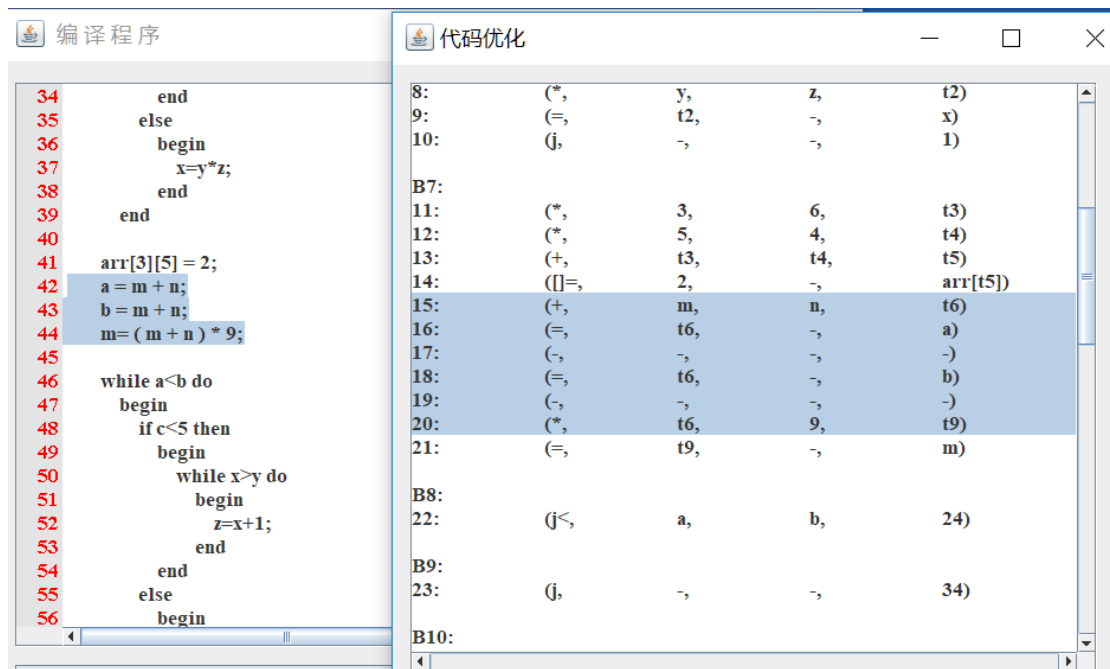
for (int k=j+1; k<result.get(i).size(); k++) // 向后查找
{
    FourAddr f2 = result.get(i).get(k);
    if (f2.getToaddr().equals(f.getParam1())
        || f2.getToaddr().equals(f.getParam2()))
    {
        // 后面有引用
        break;
    }
    if (flag == true)
    {
        // 检查后面是否有对公共子表达式左部的引用。如有，改变参数。
        if (par.contains(f2.getParam1()))
        {
            result.get(i).get(k).setParam1(f.getToaddr());
        }
        if (par.contains(f2.getParam2()))
        {
            result.get(i).get(k).setParam2(f.getToaddr());
        }
        if (par.contains(f2.getToaddr()))
        {
            result.get(i).get(k).setToaddr(f.getToaddr());
        }
    }

    if (f2.getParam1().equals(f.getParam1())
        && f2.getParam2().equals(f.getParam2())
        && f2.getOp().equals(f.getOp()))
    {
        flag = true;
        par.add(f2.getToaddr());
        result.get(i).get(k).setOp("-");
        result.get(i).get(k).setParam1("-");
        result.get(i).get(k).setParam2("-");
        result.get(i).get(k).setToaddr("-");
    }
}
}
return result;
}

```

## 4 代码优化结果

部分结果如下所示。



可以看到对于左面公共子表达式“ $m+n$ ”，代码优化系统成功进行了删除。

## 第五章 目标代码生成

### 1 代码生成需求分析

代码生成是编译的最后一个阶段，由代码生成器完成。其任务是把中间代码转换为等价的、具有较高质量的目标代码，以充分利用目标机器的资源。

当然，代码生成器本身也必须具有较高的运行效率。目标代码可以是绝对地址的机器代码，或相对地址的机器代码，也可以是汇编代码。

本部分要求设计一个代码生成器

### 2 常用四元式的翻译方法

(1) (program, prg\_id, , )

```
{MAIN SEGMENT: ASSUME
  CS:MAIN, DS:MAIN, ES:MAIN}
```

(2) (ret, , , p)

```
{JMP  p'}
```

(3) (+, A, B, T)

```
{MOV  AX, A
```

- ADD AX, B

MOV T, AX}
- (4) (-, A, \_, T)

{MOV AX, A

NEG AX

MOV T, AX}
- (5) ([]=, A, , T1[T2])

{MOV SI, T2

MOV AX, A

MOV [SI]+T1, AX}
- (6) (JNZ, A, , P)

{MOV AX, A

JNZ P'}
- (7) (J , , , P)

{JMP P'}
- (8) (J<, A, B , P)

{MOV AX, A

CMP AX, B

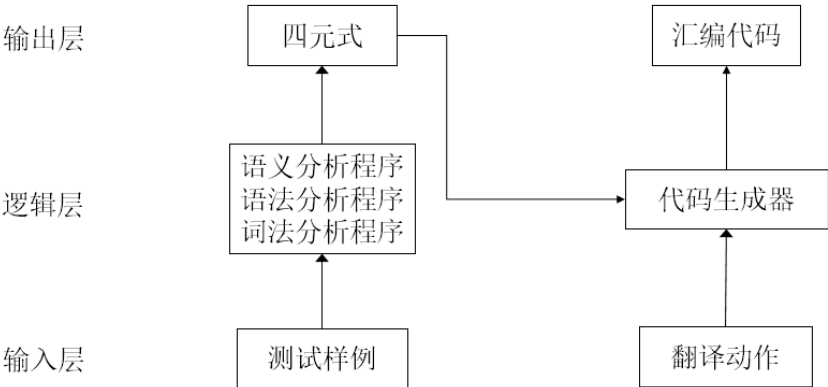
JL P'}

3 代码生成系统设计

3.1 系统逻辑

代码生成器与前几次实验相比，比较简单。其与整个系统的逻辑图如下所示。

测试样例经过词法、语法和语义分析后，得到四元式序列，代码生成器根据四元式与汇编代码对应的翻译动作将四元式翻译为汇编代码。



## 3.2 系统具体实现

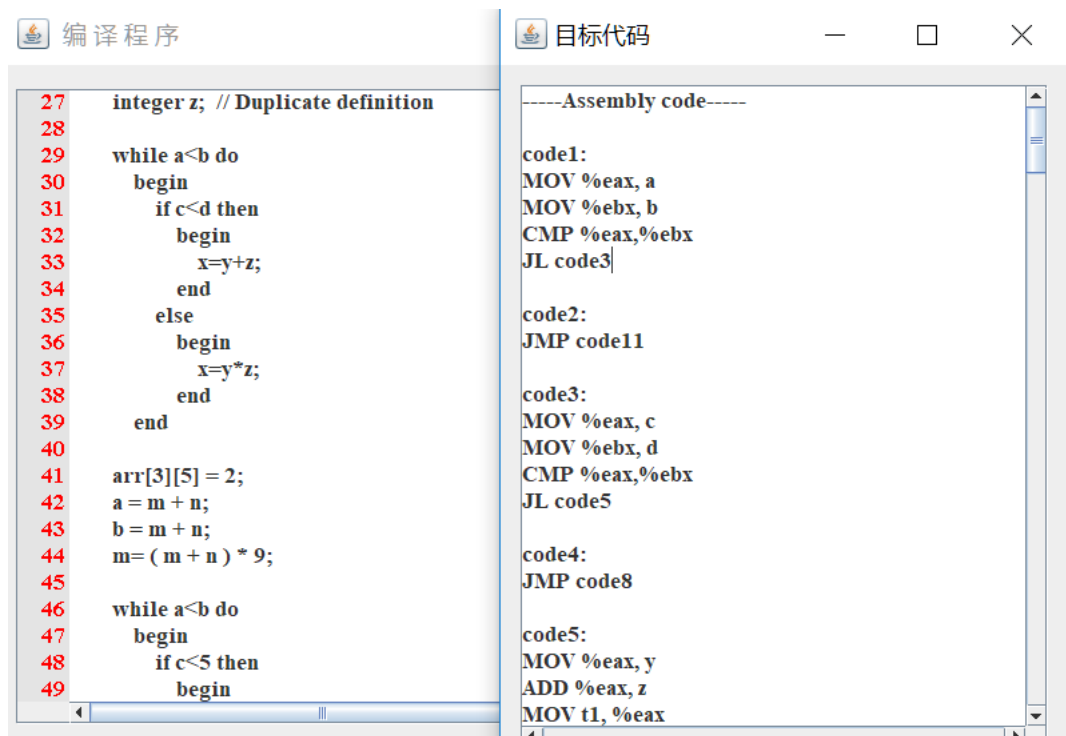
系统实现的核心就是实现四元式到汇编代码的转换，函数定义如下，详见源代码。

```
/**
 * 四元式到汇编代码的转换函数
 * @param sk1 四元式
 * @return 汇编代码
 */
public static String asm(FourAddr sk1)
```

## 4 代码生成结果分析

### 4.1 代码生成结果

部分结果如下所示。



### 4.2 分析实验结果

实验结果中，赋值表达式、数组、条件转移以、无条件转移以及转移后的目标指令均能正确显示，系统目标达成。

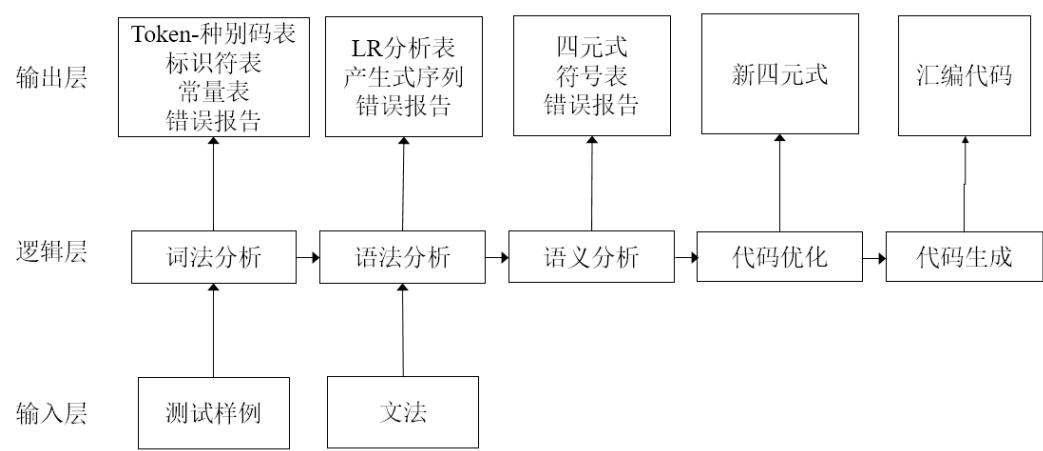
# 第六章 完整编译系统的设计

## 1 需求分析

根据前述模块，设计完整的编译系统。

## 2 系统设计

系统最终的数据流图如下，系统通过文件读入测试样例和文法，依次进行经过词法、语法、语义分析以及代码优化和目标代码生成，输出所需要的信息。



## 3 系统结果

### 3.1 测试样例

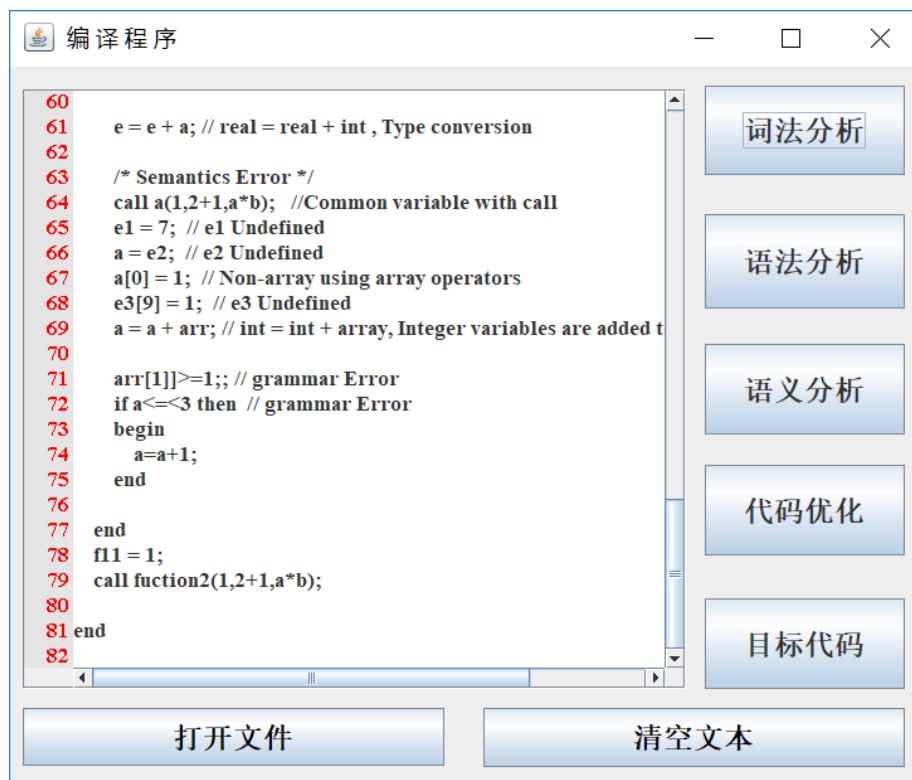
```
1. proc fuction1;
2.  begin
3.
4.     integer 1_qqq; //Lexical Error
5.     real eee;
6.     real ttt;
7.     real s;
8.
9.     integer f11;
10.    real f12;
11.
12.    proc fuction2;
13.    begin
14.        integer[7][6] arr;
15.        integer m;
```

```
16.      integer n;
17.      integer a;
18.      integer b;
19.      integer c;
20.      integer d;
21.      real e;
22.      record real re1; integer re2; end r1;
23.
24.      integer x;
25.      integer y;
26.      integer z;
27.      integer z; // Duplicate definition
28.
29.      while a<b do
30.          begin
31.              if c<d then
32.                  begin
33.                      x=y+z;
34.                  end
35.              else
36.                  begin
37.                      x=y*z;
38.                  end
39.              end
40.
41.          arr[3][5] = 2;
42.          a = m + n;
43.          b = m + n;
44.          m= ( m + n ) * 9;
45.
46.      while a<b do
47.          begin
48.              if c<5 then
49.                  begin
50.                      while x>y do
51.                          begin
52.                              z=x+1;
53.                          end
54.                      end
55.              else
56.                  begin
57.                      x=y;
```

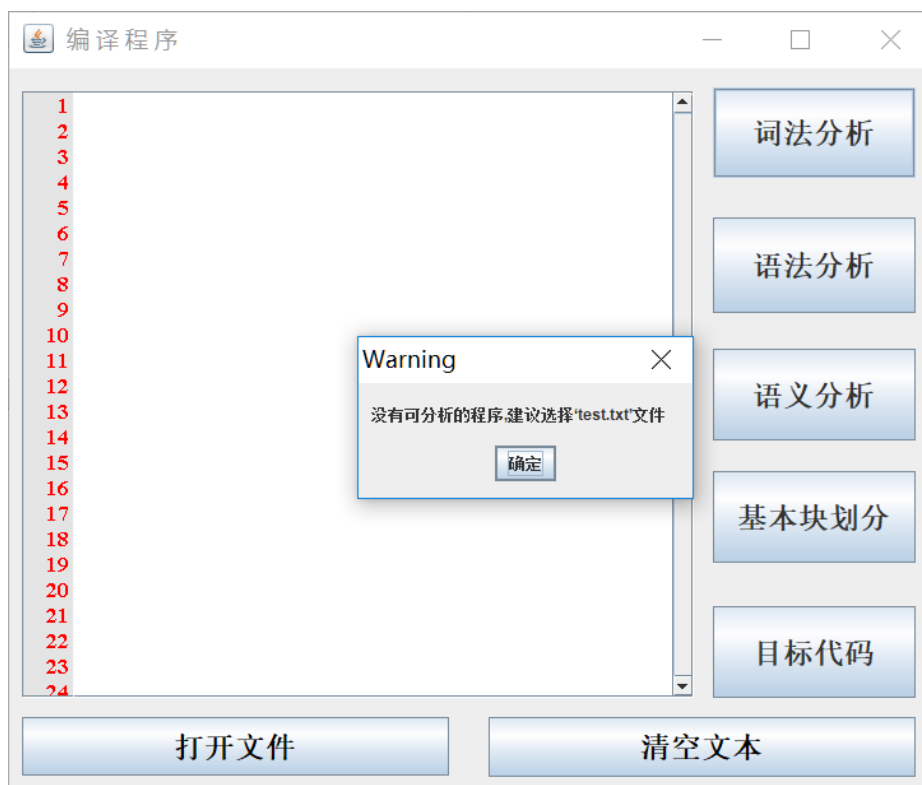
```
58.         end
59.     end
60.
61.     e = e + a; // real = real + int , Type conversion
62.
63.     /* Semantics Error */
64.     call a(1,2+1,a*b); //Common variable with call
65.     e1 = 7; // e1 Undefined
66.     a = e2; // e2 Undefined
67.     a[0] = 1; // Non-array using array operators
68.     e3[9] = 1; // e3 Undefined
69.     a = a + arr; // int = int + array, Integer variables are added to ar
ray variables
70.
71.     arr[1]]>=1;; // grammar Error
72.     if a<=<3 then // grammar Error
73.     begin
74.         a=a+1;
75.     end
76.
77. end
78. f11 = 1;
79. call fuction2(1,2+1,a*b);
80.
81. end
```

## 3.2 系统结果

主界面如下。



没有选择文件就进行分析时提示如下。





词法分析程序

行号

Token

类别

种别码

1

proc

关键字

101

1

fuction1

标识符

1

1

:

界符

302

2

begin

关键字

113

4

integer

关键字

103

4

:

界符

302

4

//Lexica...

注释

7

5

real

关键字

104

5

eee

标识符

1

5

:

界符

302

6

real

关键字

104

6

:

界符

302

6

ttt

标识符

1

6

:

界符

302

7

real

关键字

104

7

:

界符

302

7

s

标识符

1

7

:

界符

302

9

integer

关键字

103

9

:

界符

302

9

fll

标识符

1

9

:

界符

302

10

real

关键字

104

10

:

界符

302

10

f12

标识符

1

10

:

界符

302

12

proc

关键字

101

12

:

界符

302

12

fuction2

标识符

1

12

:

界符

302

13

begin

关键字

113

14

integer

关键字

103

14

:

界符

302

标识符

位置

fuction1

0

eee

1

ttt

2

s

3

fll

4

f12

5

fuction2

6

arr

7

m

8

n

9

a

10

b

11

d

12

e

13

rel

14

re2

15

rl

16

x

17

y

18

z

19

e1

20

e2

21

e3

22

e3

23

错误行号

Token

详细说明

4

1 qaq

无符号数不合规规范

常量

位置

7

0

6

1

3

2

5

3

2

4

9

5

1

6

0

7

[illegible]

上述样例的语义分析结果如下。

语义分析程序		
序号	四元式	三地址
1	(j<, a, b, 3)	if a<b goto 3
2	(j, -, -, 11)	goto 11
3	(j<, c, d, 5)	if c<d goto 5
4	(j, -, -, 8)	goto 8
5	(+, y, z, t1)	t1 = y+z
6	(=, t1, -, x)	x = t1
7	(j, -, -, 1)	goto 1
8	(*, y, z, t2)	t2 = y*z
9	(=, t2, -, x)	x = t2
10	(j, -, -, 1)	goto 1
11	(*, 3, 6, t3)	t3 = 3*6
12	(*, 5, 4, t4)	t4 = 5*4
13	(+, t3, t4, t5)	t5 = t3+t4
14	([], -, 2, -, arr[t5])	arr[t5] = 2
15	(+, m, n, t6)	t6 = m+n
16	(=, t6, -, a)	a = t6
17	(+, m, n, t7)	t7 = m+n
18	(=, t7, -, b)	b = t7
19	(+, m, n, t8)	t8 = m+n
20	(*, t8, 9, t9)	t9 = t8*9
21	(=, t9, -, a)	a = t9
22	(j<, a, b, 24)	if a<b goto 24
23	(j, -, -, 34)	goto 34
24	(j<, c, 5, 26)	if c<5 goto 26
25	(j, -, -, 32)	goto 32
26	(j>, x, y, 28)	if x>y goto 28
27	(j, -, -, 22)	goto 22

表号	符号	类型	offset
0	eee	integer	0
0	ttt	real	4
0	s	real	12
0	f11	integer	20
0	f12	real	24
0	fuction2	函数	1
1	arr	array(7, ar...	0
1	m	integer	168
1	n	integer	172
1	a	integer	176
1	b	integer	180
1	c	integer	184
1	d	integer	188
1	e	real	192
1	r1	record	200
1	x	integer	212

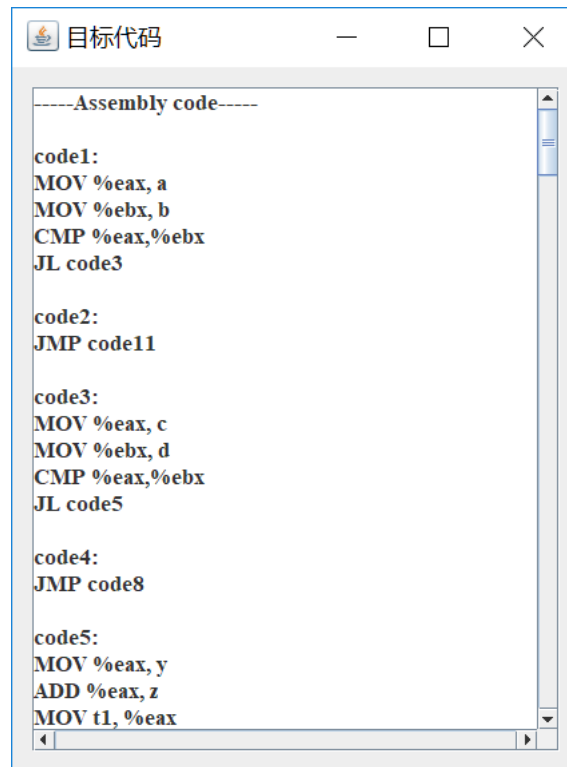
  

错误报告
Error at Line [27]: [变量z重复声明]
Error at Line [64]: [a不是函数, 不能用于call语句]
Error at Line [65]: [变量e1引用尚未声明]
Error at Line [66]: [变量e2引用尚未声明]
Error at Line [67]: [非数组变量a访问数组]
Error at Line [68]: [数组变量e3引用尚未声明]
Error at Line [69]: [整型变量与数组变量相加减]

上述样例的代码优化结果如下（以基本块结构显示）。

代码优化	
10:	(j, -, -, 1)
B7:	
11:	(*, 3, 6, t3)
12:	(*, 5, 4, t4)
13:	(+, t3, t4, t5)
14:	([], -, 2, -, arr[t5])
15:	(+, m, n, t6)
16:	(=, t6, -, a)
17:	(-, -, -, -)
18:	(=, t6, -, b)
19:	(-, -, -, -)
20:	(*, t6, 9, t9)
21:	(=, t9, -, m)
B8:	
22:	(j<, a, b, 24)
B9:	
23:	(j, -, -, 34)
B10:	
24:	(j<, c, 5, 26)

上述样例的目标代码生成结果如下。



```
-----Assembly code-----  
  
code1:  
MOV %eax, a  
MOV %ebx, b  
CMP %eax,%ebx  
JL code3  
  
code2:  
JMP code11  
  
code3:  
MOV %eax, c  
MOV %ebx, d  
CMP %eax,%ebx  
JL code5  
  
code4:  
JMP code8  
  
code5:  
MOV %eax, y  
ADD %eax, z  
MOV t1, %eax
```

## 4 总结

编译系统实验已经结束了，通过逐步完成词法分析、语法分析、语义分析和中间代码生成、代码优化、目标代码生成这五个部分，最终设计成了一个较为完整的编译系统。

其中词法分析、语法分析、语义分析和中间代码生成这三部分的完成度很高，凡是实验要求的（包括选做内容）基本都做到了；然而代码优化和目标代码生成这两个部分，限于时间关系，最终的完成度并不高，仍有待改进。

通过一连串的编译原理实验，可以说收获了很多，理论和实践水平都有一定的进步。尤其对于一些不理解的理论知识，通过实际代码的编写，最终“守得云开见月明”。

路漫漫其修远兮，吾将上下而求索。虽然说对编译系统的设计有了一些基本了解，但我深知这不过是皮毛而已，仍需要不断地努力学习，勤于实践。

2019 年 5 月 12 日