



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

编译原理实验报告

学号： 1140310606

姓名： 张茗帅

指导教师： 辛明影

哈尔滨工业大学计算机科学与技术学院

2017 年 5 月

目录

一、实验目的	3
二、实验要求:	3
三、实验环境	3
四、实验内容及实验报告格式	3
实验一 词法分析	4
实验二 LL（1）分析法	11
实验三 自顶向下语义分析	22
实验四 代码优化和汇编生成	29
实验五 总结、安装与调试	34

一、实验目的

通过实验教学，加深学生对所学的关于编译原理的理论知识的理解，增强学生对所学知识的综合应用能力，并通过实验达到对所学的知识进行验证。通过实验，使学生掌握词法分析的实现技术，深入了解语法分析的四种实现技术及具体实现方法，掌握语义分析的实现技术及具体实现方法，了解代码优化和代码生成的实现技术及具体实现方法。明确编译各阶段之间的关系。熟悉符号表的建立及在编译过程中的作用。

二、实验要求：

完成类高级语言的编译器的设计与实现

三、实验环境

硬件：我的 PC 机

程序语言：JAVA

IDE：MyEclipse

四、实验内容及实验报告格式

实验一、词法分析

实验二、LL（1）分析法

实验三、自顶向下语义分析

实验四、代码优化和汇编生成

实验五、总结、安装、调试

实验一 词法分析

1、实验目的：

通过本实验加深对词法分析程序的功能及实现方法的理解

2、实验内容：

编写一个词法分析程序，对给出的程序段进行词法分析，要求输出以文件形式存放的 TOKEN 串和符号表

3、实验方式

每位同学上机编程实现，指导教师现场指导答疑

4、实验报告内容

①实验中用到的单词的词类编码表,字符集,标识符和常数的定义,用于分析的实例

类别描述	Token	种别码
括号	(1.
括号)	2.
括号	[3.
括号]	4.
括号	{	5.
括号	}	6.
算数运算符	+	7.
算数运算符	-	8.
算数运算符	*	9.
算数运算符	/	10.
注释	#	11.
界符	;	12.
赋值运算符	=	13.
逻辑运算符	!	14.
判断运算符	>	15.
判断运算符	<	16.
单字符	'...'对应字符值 (注：当类别为标识符，此处存 对应的值，下同)	17.
字符串	"..."对应字符串值	18.
整数	Integer 对应值	19.
浮点数	Float 对应值	20.
判断运算符	!=	21.
判断运算符	==	22.
判断运算符	>=	23.

判断运算符	<=	24.
标识符	(注：当类别为标识符，此处存对应的值)	25.
关键字	void	26.
关键字	int	27.
关键字	in	28.
关键字	out	29.
关键字	string	30.
关键字	char	31.
关键字	float	32.
关键字	else	33.
关键字	if	34.
关键字	back	35.
关键字	for	36.
关键字	loop	37.
逻辑运算符	&	38.
逻辑运算符		39.

②设计符号表的逻辑结构及存贮结构

在词法分析中简单的存储结构如下（并不是真正的符号表，真正完整的符号表在语义分析阶段进行了修改）

```
private ArrayList<HashMap<String, String>> token;
```

通过使用动态数组 ArrayList 来存储每一各符号的相关内容，其中 ArrayList 中的每一项用 HashMap 的键值对来保存，“键”代表种别码，“值”代表可能对应的数值（例如当“键”为 17 代表字符时，“值”就对应其具体的字符内容）

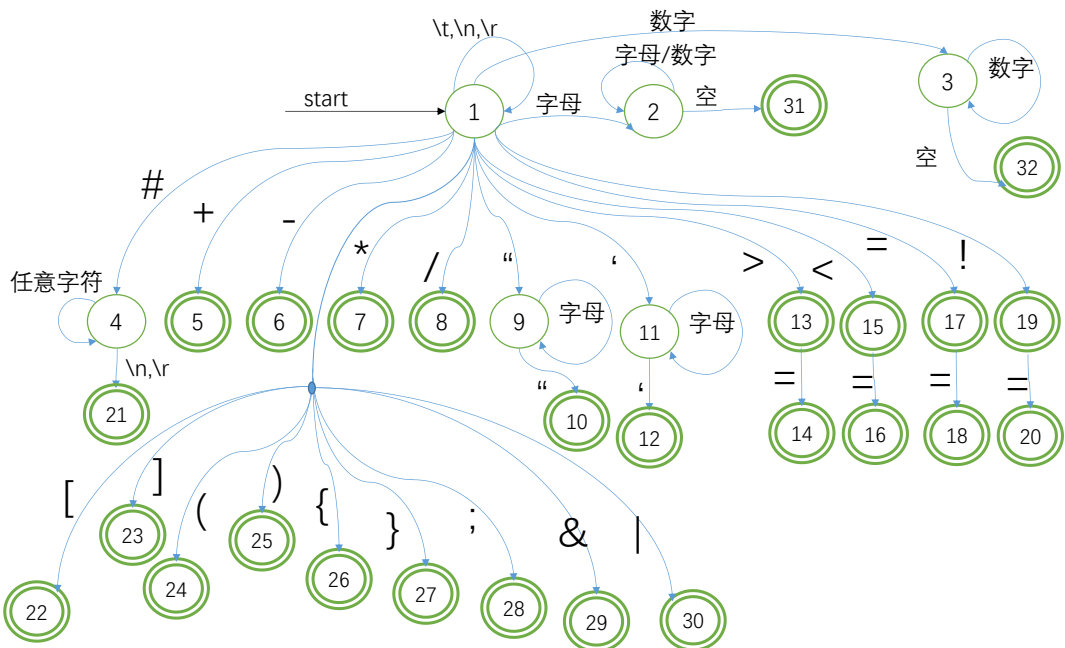
语义阶段，改正后的真正符号表如下：

```
public class Id {
    String name;
    String type;//基本类型
    int offset;//起始地址
    int length;//长度
}
```

③写出词法分析程序中主要模块的算法

分类函数的算法：

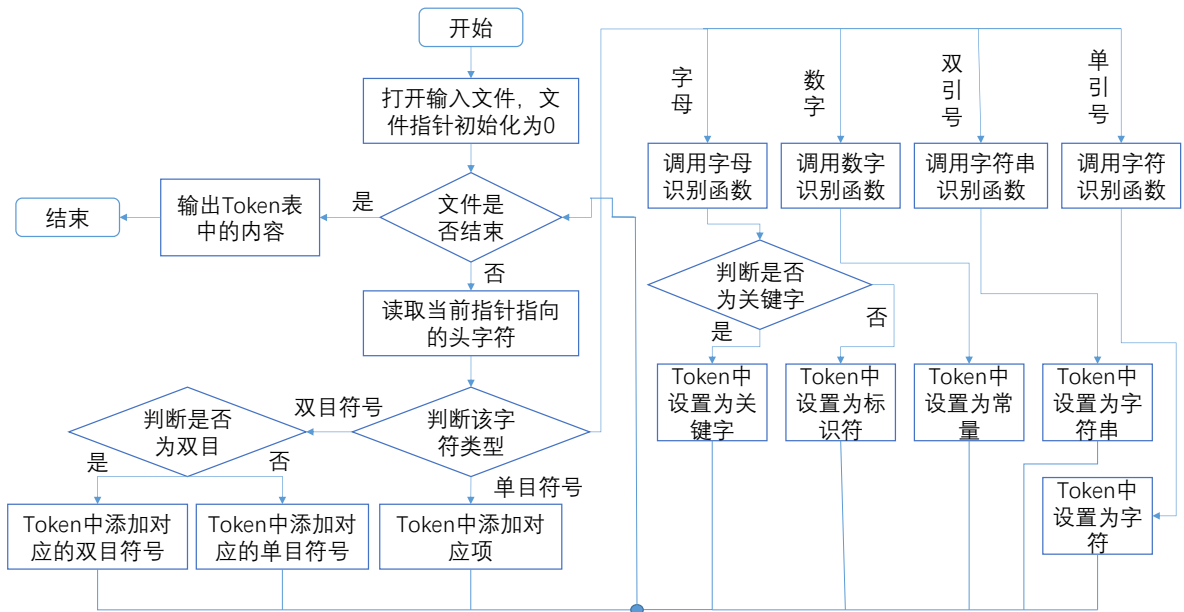
简单来说，就是通过自己设计的有穷自动机进行单词识别，这里给出识别源程序单词的有穷自动机，如下图：



该识别字符的 DFA 中各状态和其识别的单词类别对应如下：

状态号	对应识别的单词	状态号	对应识别的单词
5	+	20	!=
6	-	21	注释
7	*	22	[
8	/	23]
10	字符串	24	(
12	字符	25)
13	>	26	{
14	>=	27	}
15	<	28	;
16	<=	29	&
17	=	30	
18	==	31	标识符，关键字
19	!	32	数字常量

其具体的程序对应的流程图如下：



识别标识符的算法:

```

    if (judge_Alpha(ch) == 1) {
        s = "" + ch;
        return Deal_Alpha(i, s);
    }

public int Deal_Alpha(int index, String first) {
    int i = index;
    String s = first;
    char ch = text.charAt(++i);
    while (judge_Alpha(ch) == 1 || judge_Number(ch) == 1) {
        s = s + ch;
        ch = text.charAt(++i);
    }
    if (judge_Key(s) == 1) {
        printResult(s, "关键字", row_number, key_map.get(s));
        return i;
    } else {
        printResult(s, "标识符", row_number, var);
        return i;
    }
}

```

如上面第一个图如果识别的第一个字符为字母的话, 进入 Deal_Alpha 子程序, 从而进一步判断是关键字还是标识符。

查填符号表的算法：

查询符号表中已存在的符号：

```
for(int i=0;i<ids.size();i++){
    if(ids.get(i).getName().equals(name))
        return ids.get(i);
}
```

添加新项至符号表中（这个例子中的属性继承了其父节点的一些属性）

```
int length=Integer.parseInt(father.attribute.get("length"));
Id id=new Id(father.attribute.get("name"),father.attribute.get("type"), offset, length);

offset+=length;


int dimension=Integer.parseInt(father.attribute.get("dimension"));

for(int i=0;i<dimension;i++){
    id.arr_list.add(Integer.parseInt(father.attribute.get("arr"+i)));
}

ids.add(id);
```

④词法分析后生成的 TOKEN 文件和符号表；(可截图)

这里给出的源程序测试了每一个功能，但为了简洁，每一个具体的功能块内没有给出具体语义清晰的语句，只是为了方便地测试词法分析和语法分析的正确性，后面的语法分析的源程序也是使用的这个。

 Compiler Designed By ZhangMingshuai

File Operation

Zhang Mingshuai's Compiler

类别	Token	行号	种别码
关键字	int	1	27
标识符	main	1	25
括号	(1	1
括号)	1	2
括号	{	1	5
关键字	int	2	27
标识符	a	2	25
赋值运算...	=	2	22
整数	100	2	19
界符	;	2	12
关键字	int	3	27
标识符	h	3	25
括号	[3	3
整数	10	3	19
括号]	3	4
界符	;	3	12
关键字	string	4	30
标识符	s	4	25
赋值运算...	=	4	22

类别	Token	行号	种别码
标识符	s	4	25
赋值运算符	=	4	22
字符串	"axiba"	4	18
界符	;	4	12
关键字	float	5	32
标识符	b	5	25
赋值运算符	=	5	22
浮点数	3.4	5	20
界符	;	5	12
标识符	b	6	25
赋值运算符	=	6	22
标识符	b	6	25
算数运算符	+	6	7
浮点数	0.1	6	20
界符	;	6	12
标识符	a	7	25
赋值运算符	=	7	22
标识符	a	7	25
算数运算符	+	7	7

类别	Token	行号	种别码
标识符	b	7	25
括号	[7	3
整数	10	7	19
括号]	7	4
界符	;	7	12
关键字	if	8	34
括号	(8	1
标识符	a	8	25
判断运算符	>	8	15
整数	1	8	19
括号)	8	2
括号	{	9	5
标识符	a	10	25
赋值运算符	=	10	22
标识符	a	10	25
算数运算符	-	10	8
整数	2	10	19
界符	;	10	12
括号	}	11	6

类别	Token	行号	种别码
关键字	else	12	33
括号	{	13	5
括号	}	14	6
关键字	loop	15	37
括号	(15	1
标识符	a	15	25
赋值运算符	<	15	16
整数	2	15	19
括号)	15	2
括号	{	16	5
括号	}	17	6
关键字	for	18	36
括号	(18	1
标识符	a	18	25
赋值运算符	=	18	22
整数	1	18	19
界符	;	18	12
界符	;	18	12
标识符	a	18	25

类别	Token	行号	种别码
算数运算符	+	18	7
整数	1	18	19
括号)	18	2
括号	{	19	5
括号	}	20	6
关键字	in	21	28
括号	(21	1
标识符	a	21	25
括号)	21	2
界符	;	21	12
关键字	out	22	29
括号	(22	1
标识符	b	22	25
括号)	22	2
界符	;	22	12
关键字	back	23	35
整数	0	23	19
界符	;	23	12
括号	}	24	6

符号表如下图：

变量名称	类型	长度	内存地址
a	int	4	0
h	int[]	11	4
s	string	6	48
b	float	4	54

⑤实验中遇到的问题及解决方案

在本次实验中，整体思路还是很清晰的，自己先规定好所设计的程序语言可以识别的各类单词，然后画出对应的自动机，然后按照自动机来写程序。由于所要识别的单词种类很多，故每一部分的逻辑思考都要做的很充分，否则就会出现错误。

我在本次实验出现了两次印象比较深的错误。

第一个是在常数的识别阶段所出现的错误，由于常数可能包含浮点数，因为不能单独根据某一位不是数字就下定论识别结束（因为浮点数还有小数点“.”），所以判断常数为整数时，对识别出的不是数字的那一位要做相应的判断，当时只考虑了其后为空格、分隔符、换行、等号这些作为合法整数情况，也就是说没有考虑整数后接了+、-、*、/这些符号这些情况，之前的程序这些会被识别为错误。后来加以改正，再次判断就是正确的了。整个判别一定要仔细思考，想透所有可能情况，才能够完全正确地识别。

第二个错误是关于双目符号（例如>=、==）与单目符号（例如>、=）识别时发生的冲突问题，刚开始我的程序只要需要单目符号直接输出识别结果，因为无法识别到双目符号。后来只需对具有相同前缀的单目符号和双目符号进行特殊的判断处理，在识别出单目符号后，不要立即下结论，再向前看一个符号，如果符合双目，则输出双目符号，否则再输出单目符号。

⑥本次实验完成情况及指导教师评语

独立完成	独立未完成	未独立	指导教师签字	特殊评语
实验中用到的特色方法及设计技巧				

实验二 LL (1) 分析法

1. 实验目的:

掌握用 LL (1) 方法进行语法分析的方法

2. 实验内容:

针对定义的文法，编写一个利用 LL (1) 分析技术实现的语法分析程序，并对所给程序段进行分析，输出推导过程中所用产生式序列。

3. 实验方式

每位同学上机编程实现，指导教师现场指导答疑

4. 实验报告:

① 改造后的文法

LL1 文法必须去掉左递归和回溯（及去掉公共左因子），以及消除二义性的文法，改造后的文法如下：

整体结构（可以由多个函数构成）：

```
S -> part parts
parts -> part parts
parts -> $
```

函数声明语句（函数的返回值类型 `type` 参数 `paras` 以及主体 `part_body`）：

```
part -> type ID ( paras ) part_body
type -> int
type -> char
type -> float
type -> string
type -> void
paras -> type ID paras
paras -> $
part_body -> ;
part_body -> block
block -> { def_statements statements }
```

变量声明语句（包含数组、同时可以赋值）：

```
def_statements -> type ID def_statement def_statements
def_statements -> $
def_statement -> init ;
def_statement -> [ INT_VALUE ] init_array ;
init -> = expression
init -> $
```

init_array -> = { const consts }

init_array -> \$

表达式语句（包括赋值、计算，其中计算的项可以通过函数调用得到，见下面 factor -> ID call_part 这一处，ID 为函数名或数组名，call_part 用于传参或指明数组的索引）：

expression_statement -> expression

expression -> value operation

operation -> op value

operation -> \$

op -> >

op -> >=

op -> <

op -> <=

op -> ==

op -> !=

op -> =

value -> item value'

value' -> + item value'

value' -> - item value'

value' -> \$

item -> factor item'

item' -> * factor item'

item' -> / factor item'

item' -> \$

factor -> (value)

factor -> ID call_part

factor -> const

call_part -> [INT_VALUE]

call_part -> (para_in)

call_part -> \$

para_in -> expression para_in

para_in -> \$

consts -> const consts

consts -> \$

const -> INT_VALUE

const -> CHAR_VALUE

const -> STRING_VALUE

const -> FLOAT_VALUE

函数返回语句:

```
jump_statement -> back back_parain ;  
back_parain -> expression  
back_parain -> $
```

输入输出语句:

```
io_statement -> in ( ID ) ;  
io_statement -> out ( io_out ) ;  
io_out -> ID  
io_out -> STRING_VALUE
```

条件分支语句:

```
condition_statement -> if ( logical_expression ) block_statement result  
result -> else block_statement  
result -> $  
logical_expression -> expression logical_second  
logical_second -> lop expression logical_second  
logical_second -> $  
lop -> &  
lop -> !  
lop -> |  
block_statement -> { statements }
```

循环语句

```
loop_statement -> loop ( logical_expression ) block_statement  
loop_statement -> for ( back_parain ; back_parain ; back_parain ) block_statement  
logical_expression -> expression logical_second  
logical_second -> lop expression logical_second  
logical_second -> $  
lop -> &  
lop -> !  
lop -> |  
block_statement -> { statements }  
back_parain -> expression  
back_parain -> $
```

完整的文法如下:

```
S -> part parts  
parts -> part parts
```

```
parts -> $
part -> type ID ( paras ) part_body
type -> int
type -> char
type -> float
type -> string
type -> void
type -> $
paras -> type ID paras
paras -> $
part_body -> ;
part_body -> block
block -> { def_statements statements }
def_statements -> type ID def_statement def_statements
def_statements -> $
def_statement -> init ;
def_statement -> [ INT_VALUE ] init_array ;
init -> = expression
init -> $
init_array -> = { const consts }
init_array -> $
statements -> statement statements
statements -> $
statement -> expression
statement -> jump_statement
statement -> loop_statement
statement -> condition_statement
statement -> io_statement
io_statement -> in ( ID );
io_statement -> out ( io_out );
io_out -> ID
io_out -> STRING_VALUE
jump_statement -> back back_parain ;
loop_statement -> loop ( logical_expression ) block_statement
loop_statement -> for ( back_parain ; back_parain ; back_parain ) block_statement
condition_statement -> if ( logical_expression ) block_statement result
result -> else block_statement
result -> $
logical_expression -> expression logical_second
```

logical_second -> lop expression logical_second

logical_second -> \$

lop -> &

lop -> !

lop -> |

block_statement -> { statements }

back_parain -> expression

back_parain -> \$

expression -> value operation

operation -> op value

operation -> \$

op -> >

op -> >=

op -> <

op -> <=

op -> ==

op -> !=

op -> =

value -> item value'

value' -> + item value'

value' -> - item value'

value' -> \$

item -> factor item'

item' -> * factor item'

item' -> / factor item'

item' -> \$

factor -> (value)

factor -> ID call_part

factor -> const

call_part -> [INT_VALUE]

call_part -> (para_in)

call_part -> \$

para_in -> back_parain para_in

para_in -> \$

consts -> const consts

consts -> \$

const -> INT_VALUE

const -> CHAR_VALUE

const -> STRING_VALUE

const -> FLOAT_VALUE

注：\$代表空产生式，INT_VALUE、CHAR_VALUE、STRING_VALUE、FLOAT_VALUE 分别代表实际对应的整数型变量值、字符型变量值、字符串型变量值、浮点型变量值

② 设计 LL (1) 分析表的逻辑结构和存贮结构

逻辑结构：

主要通过一个矩阵来保存，横坐标代表终结符，纵坐标代表变元，矩阵中的每一个值代表当该变元（纵坐标值）遇到该终结符（横坐标值）应该采用哪个产生式来代替当前栈内的第一个符号，从而来找到输入串的一个最左推导。

存储结构：

我的程序主要通过 hashmap 来存储分析表的内容，如下：

```
HashMap<String, HashMap<String, Production>> analyse;
```

第一个 String（即第一个 HashMap 的键）代表变元，也就是上面所说的纵坐标，一个变元对应每一个终结符都会有一个矩阵的项，因此每一个变元（“键”）都要对应多个值，因此选择第二个 HashMap 作为“键”所对应的“值”第二个 HashMap 的“键”为终结符，“值”代表产生式（及第一个“键”变元和第二个“键”终结符所对应的产生式。）

程序中我的分析表如下：（由于分析表很大，在此只展示部分）

分析表	ID	()	int	char	float	string	void	;	{
S				S -> part pa...	S -> part pa...	S -> part pa...	S -> part pa...	S -> part pa...		
parts				parts -> par...	parts -> par...	parts -> par...	parts -> par...	parts -> par...		
part				part -> type ...	part -> type ...	part -> type ...	part -> type ...	part -> type ...		
type				type -> int	type -> char	type -> float	type -> strin...	type -> void		
paras			paras ->	paras -> ty...	paras -> ty...	paras -> ty...	paras -> ty...	paras -> ty...		
part_body									part_b...	part_bod...
block										block -> ...
def_statem...	def_statem...	def_statem...		def_statem...	def_statem...	def_statem...	def_statem...	def_statem...		
def_statem...	def_statem...	def_statem...		def_statem...	def_statem...	def_statem...	def_statem...	def_statem...	def_st...	
init									init -> \$	
init_array									init_ar...	
statements	statements...	statements...								
statement	statement -...	statement -...								
expression...	expression...	expression...								
io_statement										
io_out	io_out -> ID									
jump_state										

程序的 first 集合和 follow 集合如下：

变元&终结符	First集合	变元	follow集合
expression	(, ID, INT_VALUE, CH...	expression	;, &, !, ,), (, ID, INT_V...
logical_second	&, !, , \$	result	back, loop, for, if, in, ...
for	for	logical_second)
char	char	const	INT_VALUE, CHAR_...
parts	int, char, float, string, ...	block	int, char, float, string, ...
item'	*, /, \$	parts	#
jump_statement	back	item'	+, -, >=, <=, <, <=, ==, !=...
type	int, char, float, string, ...	type	ID
float	float	jump_statement	back, loop, for, if, in, ...
operation	>, >=, <, <=, ==, !=, =, \$	loop_statement	back, loop, for, if, in, ...
value	(, ID, INT_VALUE, CH...	operation	;, &, !, ,), (, ID, INT_V...
ID	ID	io_out)
condition_statement	if	S	#
else	else	statements	}
!	!	value	>, >=, <, <=, ==, !=, =, ...
INT_VALUE	INT_VALUE	para_in)
op	>, >=, <, <=, ==, !=, =	paras)
lop	&, !,	condition_statement	back, loop, for, if, in, ...
&	&	op	(, ID, INT_VALUE, CH...

③ 文法在计算机内的存放方法

```
//产生式类
public class Production {
    public String left;
    public String[] right;
    // 初始化select集
```

我在我的程序中用一个类来存储文件中文法的每一条记录，left 用于保存变元的左部（即->的左面），right[] 字符串数组用于保存产生式右部的每一个变元和终结符。

④ LL(1)分析法主控程序算法

```
public void begin_analyse(DefaultTableModel result) {
    stackArrayList.add("S");
    stackArrayList.add("#"); //初始化分析栈，加入#和 S
    token.add("#"); //初始化输入缓冲区，加入#
    String stack_top;
    String token_top;
    String[] rights;
    String right;
    Production tempProduction;
    int kk=0;
```

```

while (true) {
    token_top = token.get(0);
    stack_top = stackArrayList.get(0);
    if(stack_top.equals("#")){          //如果分析栈顶为#且输入缓冲区
        if(token_top.equals("#")){ //栈顶也为#, 则分析成功
            result.addRow(new String[]{" ", " ", "Sucess!"});
            System.out.println("Success!");
            break;
        }
        else{ //分析栈顶为#, 输入栈顶还有未处理的字符, 则分析失败
            result.addRow(new String[]{" ", " ", "Error!"});
            System.out.println("Error!");
            break;
        }
        //如果分析栈顶和输入栈顶都为相同的终结符则匹配消除
    }else if (terminal.contains(stack_top)){ /
        if(stack_top.equals(token_top)){
            System.out.println("匹配"+stack_top);
            result.addRow(new String[]{stack_top, token_top, "匹配替
换"+stack_top});
            token.remove(0);
            stackArrayList.remove(0);
        }else{
            System.out.println("2Error!");
        }
    }
    else{ //如果不满足以上情况, 则按照分析表替换分析栈中的产生式
        tempProduction = analyse.get(stack_top).get(token_top);
        rights = tempProduction.returnRights();
        right = "";
        for(String ss:rights){
            right = right+ss+" ";
        }
        System.out.println("替换"+stack_top+"->"+right);
        result.addRow(new String[]{stack_top, token_top, "替 换
"+stack_top+"->"+right});
        stackArrayList.remove(0);
        if(!rights[0].equals("$")){
            for(int i=rights.length-1;i>=0;i--){
                stackArrayList.add(0, rights[i]);
            }
        }
    }
}

```

```

    }
    }
    }
    kk++;
    System.out.println(" 栈 顶： "+stackArrayList.get(0)+" 输 入 流： "+token.get(0));
    }
}

```

程序得到的分析结果如下图（分析的源程序为词法分析报告中的那个截图中的源程序）：

栈顶	输入流顶端	动作
S	int	替换S->part parts
part	int	替换part->type ID (paras) part_body
type	int	替换type->int
int	int	匹配消除int
ID	ID	匹配消除ID
((匹配消除(
paras)	替换paras->\$
))	匹配消除)
part_body	{	替换part_body->block
block	{	替换block->{ def_statements statements }
{	{	匹配消除{
def_statements	int	替换def_statements->type ID def_statement def_state...
type	int	替换type->int
int	int	匹配消除int
ID	ID	匹配消除ID
def_statement	=	替换def_statement->init;
init	=	替换init->= expression

栈顶	输入流顶端	动作
jump_statement	back	替换jump_statement->back back_parain ;
back	back	匹配消除back
back_parain	INT_VALUE	替换back_parain->expression
expression	INT_VALUE	替换expression->value operation
value	INT_VALUE	替换value->item value'
item	INT_VALUE	替换item->factor item'
factor	INT_VALUE	替换factor->const
const	INT_VALUE	替换const->INT_VALUE
INT_VALUE	INT_VALUE	匹配消除INT_VALUE
item'	;	替换item'->\$
value'	;	替换value'->\$
operation	;	替换operation->\$
;	;	匹配消除;
statements	}	替换statements->\$
}	}	匹配消除}
parts	#	替换parts->\$
		Sucess!

可以看到，最后分析得到了成功

⑤ 实验中遇到的问题及解决方案

LL1 语法分析最大的难处在与语法分析表的构造，只要分析表构造正确，接下来的一切基本都没有什么问题。预测分析的总控程序相对来说也好写了很多。我在本次实验中遇到了很多困难，因为当分析失败的时候，我们需要根据程序失败的地方（点）一步一步推导到那里，从而才能知道到底是分析表中的哪个项填错了，而填错的原因是构造分析表的时候出错了呢，还是 **first** 集合或者 **follow** 集合求错了。在如此庞大的分析表中搜索错误，任务量巨大，我几乎花了 3 整天的时间进行 **bug** 调试（不包括写的时间...）。

整个实验我印象最深的错误主要有三个：

Follow 集合的构造。对于 **follow** 集合构造的第五个步骤（书中 130 页）， $A \rightarrow B\alpha$ ，如果 α 能够推出为空，则将 A 的 **follow** 集合合并到 B 的 **follow** 集合中，刚开始我把 α 当作了第一个符号，只判断了 B 之后的第一个符号是否为空，其实这样做是不对的， α 表示的是 B 后面的所有字符，即使第一个字符可能推导出空，但后面的可能不出来为空，所有整个的 α 就推导不出来为空，这样就不能够把 A 的 **follow** 集合合并到 B 的 **follow** 集合中。解决方案就是对 α 中的每一个符号从头开始判断，直到有一个符号无法推导出为空，则此时 α 无法推导出空，或者 α 中的每一个符号均能推导出空，则此时认为 α 可以推导出空，并将 A 的 **follow** 集合合并到 B 的 **follow** 集合中。同样的错误也发生在分析表添加项过程中（如对于 $A \rightarrow \alpha$ ，a 属于 $\text{first}(\alpha)$ 填 $A \rightarrow \alpha$ 到分析表中，对于这个 α ，我刚开始也是进行了错误的判断）。

分析表的存储问题。我的分析表存储结构是 **hashmap** 中又嵌入了一个 **hashmap**。即如下式子（在上面的“分析表的存储结构”有详细介绍）

```
HashMap<String, HashMap<String, Production>> analyse
```

在为每一个变元添加其对应的值（即第二个 **HashMap**）时，在完成分析表构造第一个步骤后（书中 134 页），此时我为每一个变元 **new** 了一个 **HashMap** 与其对应。而问题来了，当利用第二个步骤继续添加分析表的时候，有一些满足条件的变元再一次 **new** 了一个 **HashMap** 与其对应，而覆盖了之前的 **HashMap**（也就是第一个步骤所填写的项），那么最后得到的分析表一定是不正确的，我刚开始没有注意到这个问题，后来通过 **debug** 调试后发现了这个问题。改正措施为，在构造分析表之前就为每一个变元只 **new** 一个 **HashMap** 与其对应，当之后不管哪一个步骤要添加项的时候，只需要找到特定变元所对应的 **HashMap** 往里面添加对应的终结符和产生式即可。

无左递归文法的构造问题。在进行程序 **debug** 过程中，我发现分析

表已经更改正确了，但仍然无法得到正确分析结果，后来经过反复推敲，发现可能是文法的某一部分有问题，于是我去检查文法，发现真的存在问题，而错误的原因通常是很小的以至于不容易被发现，例如文法文件中某个产生式右部本该有分号就少了个分号，或是产生式右部每个字符之间忘记了打空格，以至于程序识别其为一个新的变元或终结符，等等。解决方案就是仔细校对文法，得到完全正确的 LL(1) 文法即可。

⑥ 本次实验完成情况及指导教师评语

独立完成	独立未完成	未独立	指导教师签字	特殊评语
实验中用到的特色方法及设计技巧				

实验三 自顶向下语义分析

1、实验目的：

加深对自顶向下语法制导翻译技术的理解与掌握

2、实验内容：

针对 LL(1) 分析法中所使用的文法，为其设计翻译方案，并利用该翻译方案，对所给程序段，进行分析，输出生成中间代码序列和符号表。

3、实验方式

每位同学上机编程实现，指导教师现场指导答疑

4、实验报告内容

① 写出主要产生式语义翻译时的目标代码结构

简单的赋值语句：如 $a = 1$; (其中“-”代表该项为空，下同)

(=, 1, -, a)

用于计算的数学表达式：如 $a = a * 1$;

(*, a, 1, a)

对于布尔型变量的判断转移：如 $\text{if}(a > 1 \text{ and } b < 2) \{ \} \text{else} \{ \}$

1: (j>, a, 1, 3)

2: (j, -, -, 6)

3: (j<, b, 2, 5)

4: (j, -, -, 6)

5: 第一个花括号中内容

6: 第二个花括号中内容

再例如： $\text{if}(a > 1 \text{ or } b < 2) \{ \} \text{else} \{ \}$

1: (j>, a, 1, 5)

2: (j, -, -, 3)

3: (j<, b, 2, 5)

4: (j, -, -, 6)

5: 第一个花括号中内容

6: 第二个花括号中内容

过程调用语句：例如调用 $a = f(1)$

1: (param, 1, -, -)

2: (call, f, -, a)

输入输出语句：例如 $\text{printf}(\text{"\%d"}, a)$

1: (param, a, -, -)

2: (param, identifier:\%d, -, -)

3: (call, printf, -, -)

含有数组的表达式

对于数组地址的计算公式为： $\text{addr} = a - c + v$

a 为数组的首地址，c 为数组不变的部分（通过数组的下标下界和数组元

素类型所占有的字节以及数组的维数和每一维的维长计算得到), v 是针对不同的索引值, 计算得到的相对于 $a-c$ (也就是数组的基地址, $a[0,0]$ 类似与这种) 的偏移量。

例如 $y=x[i, j]$;

1: $(*, i, 10, t1)$

2: $(+, t1, j, t1)$

3: $(-, A, 44, t2)$

4: $(*, 4, t1, t3)$

5: $(=[], t2, t3, t4)$

6: $(=, t4, -, y)$

注: 这里面 44 的计算就是默认该数组下标下界为 1, 上界为 10, 一共有 2 维, 且数组每一个元素占用的字节数为 4 (即可以认为是 int 整型), 则 $x[1, 1]$ 相对于 $x[0, 0]$ 的距离就是 $4*(1*10+1)$ 得到 $c=44$ 。

循环语句和条件语句

需要用到回填技术, 每写完一个带转移的语句(while、if、for)时, 需要在下一个标号出给出无条件跳转指令, 跳转的位置暂且不填, 等到之后跳转的标号确定后再填写。

举一个复杂的例子例如

```
while (a<b){
    if (c<5) {
        while (x>y) {
            z =x+1;
        }
    }
    else {
        x =y;
    }
}
```

翻译得到的四元式如下: (其中-代表该项为空)

0: $(j<, a, b, 102)$

1: $(j, -, -, 112)$

2: $(j<, c, 5, 104)$

3: $(j, -, -, 110)$

4: $(j>, x, y, 106)$

5: $(j, -, -, 100)$

6: $(+, x, 1, t1)$

7: $(=, t1, -, z)$

8: $(j, -, -, 104)$

```

9: (j,-,100)
10: (=,y,-,x)
11: (j,-,100)
12:

```

② 写出适合自顶向下分析的翻译方案

首先，在设计翻译方案之前定义了以下动作：

```

A.name=B.name
将 B 的 name 属性赋给 A 的 name
CALL (params)
表示执行 CALL 函数，params 为参数
lookup(A) 查找变量 A 的引用
gencode_param() 将参数值拼接成三地址码
gencode_assign() 赋值语句生成三地址码
lookupArray() 数组引用处理
newTemp() 新建临时变量
newlabel() 新建一个标签
label(A.name) 将 A 的 name 属性作为标签加入下一个三地址码中
assign() 生成赋值语句
A.name="value"
将 A 的 name 属性赋值为 value 字符串
A.name=NULL
将 A 的 name 属性清空

```

注：语义动作在花括号中 {}，在语法分析时同步进行语义分析，{} 中的内容可以当作是一个终结符，每当执行到此处时，执行其对应的语义动作。

函数定义：

```

function_definition ->
type_specifier {function_definition.name=type_specifier.type}
pointer declarator_for_fun
{function_definition.name=declarator_for_fun.name;
function_definition.params=declarator_for_fun.params;declareFu
nction(function_definition)}compound_statement

```

变量说明：

```

declaration_list -> declaration declaration_list'
declaration_list' -> declaration declaration_list' | $
declaration -> type_specifier
{declaration.type=type_specifier.type}
pointer init_declarator
{declaration.name=init_declaration.name} declaration'

```



```

{declareVar(declaration)}
declaration' -> , init_declarator declaration' | $
init_declarator -> declarator
{init_declarator.name=declarator.name} init_declarator'
init_declarator' -> $ | = initializer

```

赋值语句:

```

assignment_expression ->
IDENTIFIER
{assignment_expression.name=IDENTIFIER.name;ASSIGN()}
assignment_expression'
| const_expression
{assignment_expression.addr=const_addr;ASSIGN()}
assignment_expression'
assignment_expression' -> assignment_expression' | =
logical_or_expression
assignment_expression' -> > logical_or_expression | <
logical_or_expression | >= logical_or_expression | <=
logical_or_expression | == logical_or_expression | !=
logical_or_expression | && logical_or_expression | ||
logical_or_expression | + logical_or_expression | -
logical_or_expression | / logical_or_expression | *
logical_or_expression | % logical_or_expression | $

```

表达式语句:

```

const_expression ->
CONST_INT {const_expression.name=CONST_INT.name}
| CONST_FLOAT {const_expression.name=CONST_FLOAT.name}
| CONST_CHAR {const_expression.name=CONST_CHAR.name}
| CONST_STRING {const_expression.name=CONST_STRING.name}

```

过程调用语句:

```

S → call id (Elist) {n := 0;
repeat
    n:=n+1;
    从 queue 的队首取出一个实参地址 p;    gencode('param', -, -, p);
until queue 为空;
gencode('call', id.addr, n, -)}
Elist → Elist, E {将 E.addr 添加到 queue 的队尾}
Elist → E {初始化 queue, 然后将 E.addr 加入到 queue 的队尾。}
过程返回语句:
S → return E {if 需要返回结果 then gencode(':=', E.addr, -, F);

```

```
gencode('ret', -, -, -)}
```

循环语句:

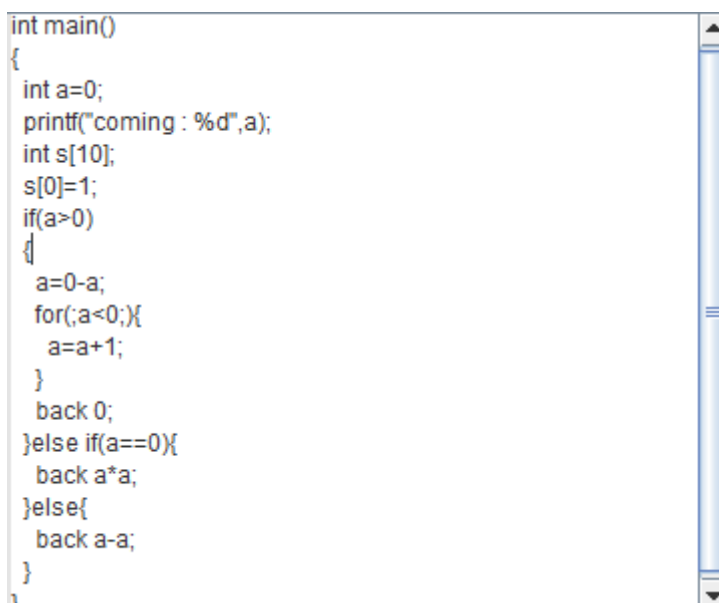
```
iteration_statement ->
  while ( {newlabel(iteration_statement.begin);
newlabel(iteration_statement.true);
newlabel(iteration_statement.next);
newlabel(iteration_statement.begin)}
expression )
statement {iteration_statement.addr=statement.addr}
{gencode_param(IF)}
| for ( expression_statement expression_statement expression )
statement
```

条件分支语句:

```
selection_statement ->
  if (
{newlabel(selection_statement.true);
newlabel(selection_statement.false)}
expression
{selection_statement.addr=expression.addr;gencode_param(IF)} )
statement selection_statement'
selection_statement' -> $
| else {LABEL(selection_statement.false)} statement
```

③ 写出翻译结果(局部)

通过程序测试得到如下结果：



```
int main()
{
    int a=0;
    printf("coming : %d",a);
    int s[10];
    s[0]=1;
    if(a>0)
    {
        a=0-a;
        for(;a<0;){
            a=a+1;
        }
        back 0;
    }else if(a==0){
        back a*a;
    }else{
        back a-a;
    }
}
```

经过语义分析得到的结果如下：

序号	四元式
0	(=, 0, -, a)
1	(param, a, -, -)
2	(param, identifier: %d, -, -)4
3	(call, printf, -, t1)
4	([]off, s, 0, t2)
5	(=, 1, -, t2)
6	(j>, a, 0, 8)
7	(j, -, -, 16)
8	(-, 0, a, t4)
9	(=, t4, -, a)
10	(j<, a, 0, 12)
11	(j, -, -, 15)
12	(+, a, 1, t6)
13	(=, t6, -, a)
14	(j, -, -, 10)
15	(return, 0, -, -)
16	(j=, a, 0, 18)
17	(j, -, -, 20)
18	(*, a, a, t8)
19	(return, t8, -, -)
20	(-, a, a, t9)
21	(return, t9, -, -)
22	END

④ 实验中遇到的问题及解决方案

语义分析，可以说是工作量最大的一次实验了，每一个步骤都需要反复推敲，考虑清楚了，否则任何一个细小的错误都会导致最终的失败，例如符号表是否填写完全正确，每一个产生式的语义动作是否完整且正确等等。

我在本次实验遇到的最大困难就是针对 LL（1）文法所设计其自顶向下的语义分析的动作，主要是 LL（1）文法不能是左递归的，而往往具有左递归的文法语义更加清楚，一旦消除了左递归，语义动作就变得不那么清晰了。我在本次实验所遇到最大的困难就是对 LL（1）文法语义动作的添加，由于在文法中计算表达式的那一部分几乎都进行了非左递归的改造，因此这个语义动作的添加步骤就变得异常的艰难。但是我还是一步一步的按照老师 ppt 上的指导，引入继承属性 R.i 来临时保存当前所计算的结果，最后再引入一个综合属性 R.s 来在结束时（通常为 R→\$时）复制 R.i 的值作为最终的表达式结果。最终在细心的操作下，成功完成了本次实验。

⑤本次实验完成情况及指导教师评语

独立完成	独立未完成	未独立	指导教师签字	特殊评语
实验中用到的特色方法及设计技巧				

实验四 代码优化和汇编生成

1、实验目的：

掌握中间代码优化的基本方法以及汇编生成的原理

2、实验内容：

针对之前语义分析得到的中间代码进行优化并且生成可执行的汇编语言程序

3、实验方式

每位同学上机编程实现，指导教师现场指导答疑

4、实验报告内容

① 代码优化实现

测试程序仍然选择在语义分析时所使用的程序，曾经的中间代码如下：

序号	四元式
0	(=, 0, -, a)
1	(param, a, -, -)
2	(param, identifier: %d, -, -)4
3	(call, printf, -, t1)
4	([]off, s, 0, t2)
5	(=, 1, -, t2)
6	(j>, a, 0, 8)
7	(j, -, -, 16)
8	(-, 0, a, t4)
9	(=, t4, -, a)
10	(j<, a, 0, 12)
11	(j, -, -, 15)
12	(+, a, 1, t6)
13	(=, t6, -, a)
14	(j, -, -, 10)
15	(return, 0, -, -)
16	(j=, a, 0, 18)
17	(j, -, -, 20)
18	(*, a, a, t8)
19	(return, t8, -, -)
20	(-, a, a, t9)
21	(return, t9, -, -)
22	END

通过代码优化的一些思想，我们可以发现在这个中间代码序列中，t4 和 t6 只是作为传递的临时值而存在，一次使用之后便不再使用了，这也就是所谓的复制传播，将其对应的等式右部均转化为同一个来源的根式，则就会发现公共子表达式，于是就可以通过删除它进行优化。

例如看序号 8 和 9，序号 8 用来把 0-a 得到的值赋给 t4，而序号 9 用来把 t4 的值赋给 a，因此首先把序号 9 更改为把 0-a 的值赋给 a，此时便有了共同子表达式 0-a，又因为 t4 在之后未被使用，则直接删除 t4，

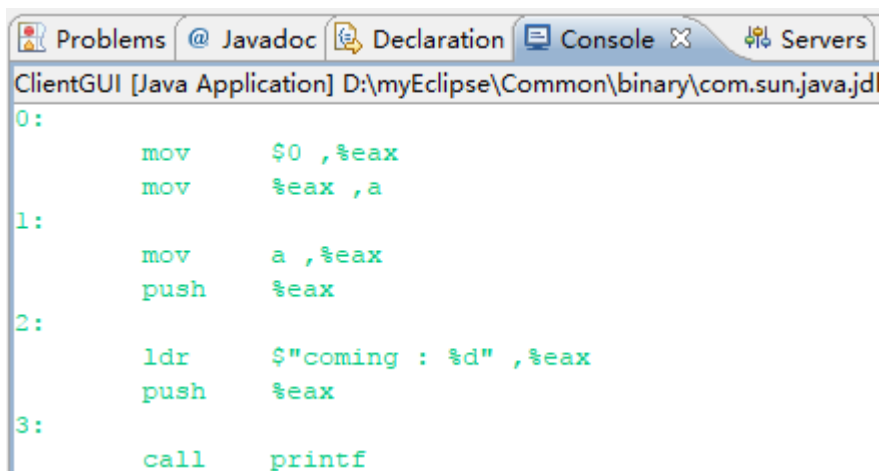
完成该部分优化。优化后得到的新四元式如下：

序号	四元式
0	(=, 0, -, a)
1	(param, a, -, -)
2	(param, identifier: %d, -, -)
3	(call, printf, -, t1)
4	([]off, s, 0, t2)
5	(=, 1, -, t2)
6	(j>, a, 0, 8)
7	(j, -, -, 14)
8	(-, 0, a, a)
9	(j<, a, 0, 11)
10	(j, -, -, 13)
11	(+, a, 1, a)
12	(j, -, -, 9)
13	(return, 0, -, -)
14	(j=, a, 0, 16)
15	(j, -, -, 18)
16	(*, a, a, t8)
17	(return, t8, -, -)
18	(-, a, a, t9)
19	(return, t9, -, -)
20	END

可以看到四元式的数量得到了减少。与此同时，我还简单做了一些其他方面的代码优化，只不过这个测试用例没有体现出来。例如一些简单的常量合并，避免以后的重复计算；循环不变量的外提；强度削弱（求平方运算转换为乘法运算，求除法运算转换为求乘法运算等等）。

② 汇编生成

根据四元式得到对应的汇编代码如下：



```

ClientGUI [Java Application] D:\myEclipse\Common\binary\com.sun.java.jdl
0:
    mov     $0 ,%eax
    mov     %eax ,a
1:
    mov     a ,%eax
    push    %eax
2:
    ldr     $"coming : %d" ,%eax
    push    %eax
3:
    call    printf
  
```

```
4:      ldr      s ,%eax
        mov     $0 ,%ebx
        add     %ebx ,%eax
        mov     %eax ,t2

5:      mov     $1 ,%eax
        mov     t2 ,%ebx
        mov     %eax ,(%ebx)

6:      mov     a ,%eax
        mov     $0 ,%ebx
        cmp     %eax ,%ebx
        mov     %ebx ,t3

7:      mov     t3 ,%eax
        cmp     $0 ,%eax
        jz      17

8:      mov     $0 ,%eax
        mov     a ,%ebx
        sub     %eax ,%ebx
        mov     %ebx ,t4

9:      mov     t4 ,%eax
        mov     %eax ,a

10:     mov     a ,%eax
        mov     $0 ,%ebx
        cmp     %ebx ,%eax
        mov     %eax ,t5

11:     mov     t5 ,%eax
        cmp     $0 ,%eax
        jz      15

12:     mov     a ,%eax
        mov     $1 ,%ebx
        add     %eax ,%ebx
        mov     %ebx ,t6

13:     mov     t6 ,%eax
        mov     %eax ,a

14:     jmp     10
```

```
15:      mov     $0 ,%eax
      ret     8

16:      jmp     24

17:      mov     a ,%eax
      mov     $0 ,%ebx
      sub     %eax ,%ebx
      mov     %ebx ,t7

18:      mov     t7 ,%eax
      cmp     $0 ,%eax
      jz      22

19:      mov     a ,%eax
      mov     a ,%ebx
      mul     %eax ,%ebx
      mov     %ebx ,t8

20:      mov     t8 ,%eax
      ret     8

21:      jmp     24

22:      mov     a ,%eax
      mov     a ,%ebx
      sub     %eax ,%ebx
      mov     %ebx ,t9

23:      mov     t9 ,%eax
      ret     8

24:
```

③ 实验中遇到的问题及解决方案

本次实验让我对代码优化的思想和方法有了深入的了解，实现了一些简单的代码优化。与此同时，根据相应的四元式生成了对应的汇编代码，在这个过程中是比较困难的，主要原因是当年学习的汇编语言指令有一些忘记了，不过重新捡起来也很快，通过网上搜索不会的汇编语言的指令就可以解决疑惑。本来对于代码优化这一块我还有一个优化想做，就是循环中归纳变量的删除，但是由于时间问题没有将这个优化完成，其他优化基本算是完成了，循环归纳变量删除思路很清晰，就是找到随着每一次循环变化有规律的变量，我们只保留其中的一个，对于其他的归纳变量进行删除，同时若其他归纳变量的值在循环体中会被用到的话，则用保留的基本归纳变量来算出其

他归纳变量的值再进行利用计算，这是一定可行的，因为每一个归纳变量都随便循环体等比例变化，知道其中的一个，就可以利用它表示其他的值。

④ 本次实验完成情况及指导教师评语

独立完成	独立未完成	未独立	指导教师签字	特殊评语
实验中用到的特色方法及设计技巧				

实验五 总结、安装与调试

1、实验目的：

深入了解编译程序的完整工作过程及各阶段之间的数据流

2、实验内容：

将前面各个过程配上输入输出部分，组装成一个完整的编译程序前端

3、实验方式

每位同学上机编程实现，指导教师现场指导答疑

3、实验报告内容

① 本次大实验完成的全部内容

(1)对于实验内容：

完成了全部实验，包括词法分析、语法分析、语义分析、代码优化、汇编生成。

注：在格物楼进行验收时并没有验收代码优化，但后来腾出时间了将代码优化部分完成)

(2)对于编译源程序：

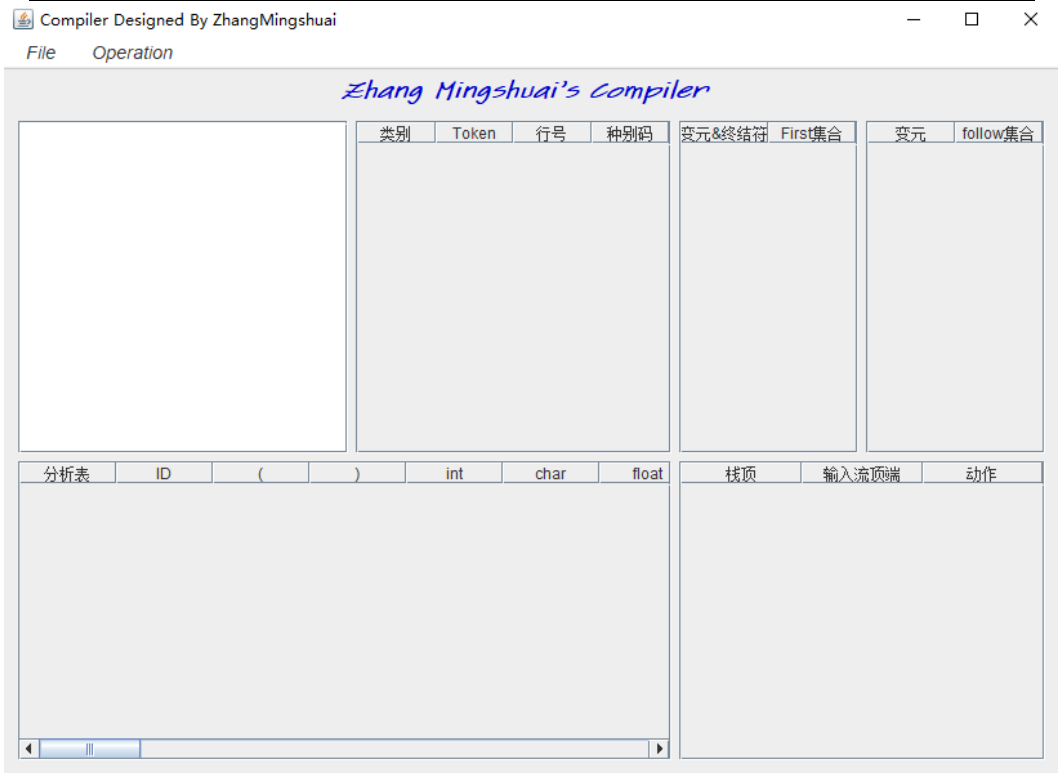
实现了老师所要求的全部内容（包括拓展），即声明语句，包括变量声明；算术表达式的运算包括加、减、乘、除；赋值语句；布尔表达式；三种典型的控制流语句（顺序，循环、分支）；一维数组的声明和引用；函数声明和函数调用语句；输入、输出语句。

注：在实验室进行语义验收时，当时由于时间紧张珊瑚了输入输出部分，后来考完试了腾出时间将该部分添加回来，具体可以在“语义”部分的报告中观察截图，可以看到输出语句 `printf`

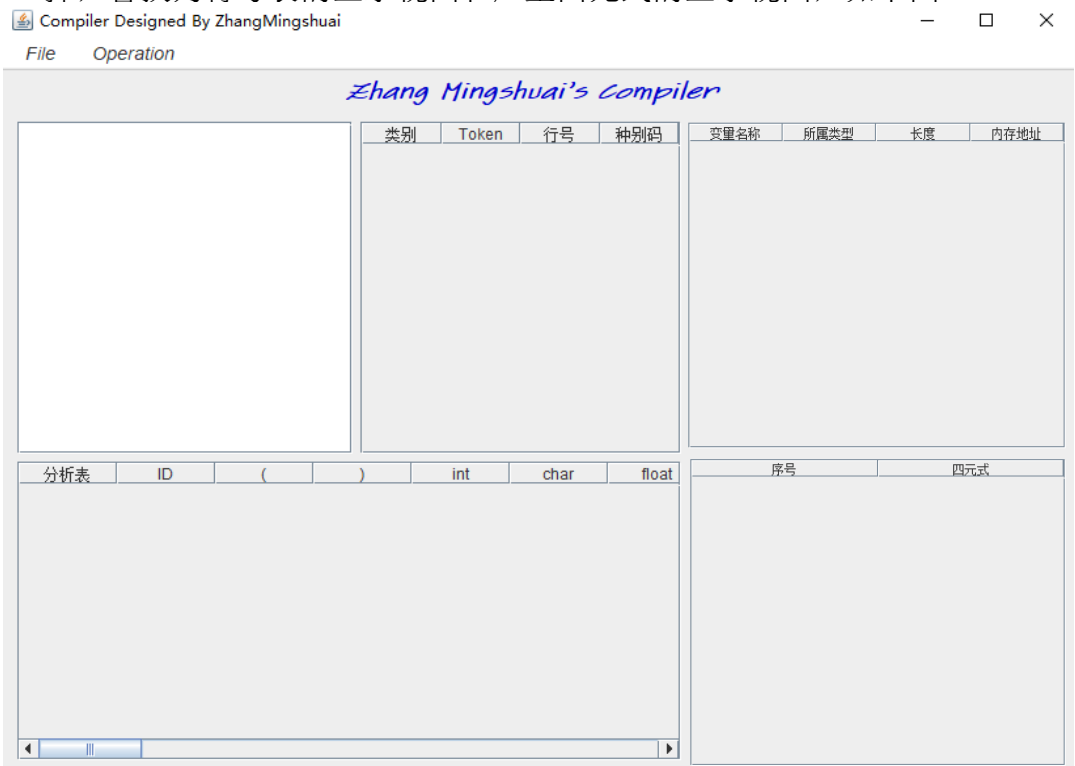
② 写出组装过程及在组装过程存在的过程

我的程序显示除了汇编生成是在命令行中显示（因为 UI 界面实在是腾不出地方来了），其他的均在图形化 GUI 界面中进行显示。我的图形化界面基于 java 中的 JFrame 框架，采用 matisse 插件进行图形绘制，然后将不同的按钮事件对应到相应的 java 类中即可。通过返回的值都要存储在 jTable 中，于是给 java 类传入 DefaultTableModel，故算法在执行时可以直接每一行每一行地将特定值填入 jTable 表中。

语法阶段的整体图形化界面如下：



语义分析中，将右面的 first 集合显示视图和 follow 集合显示视图去掉，替换为符号表的显示视图和产生四元式的显示视图，如下图：



③ 谈谈你在整个设计过程中的体会与收获

通过本次编译原理的全部实验，我感觉自己收获巨大

首先从实验本身来讨论这个问题。只有通过这个实验才能深刻地理解课堂上所学的知识内容，所谓实践是检验真理的唯一标准，对于这句话我很赞同，只有通过实践，我们对知识的掌握才能十分牢靠，因为没有完全正确的程序，是不可能得到正确的输出结果的。

其次，从课程的角度来看，我对编译原理这门课产生了浓厚的兴趣，它仿佛为揭开了高级语言的神秘面纱，让我们深入了解了高级语言是如何翻译成汇编语言的，从而让我们感觉高级语言也不再那么难了，以后若学习一门新的高级程序语言也会从心里上感到一丝轻松，毕竟原理我们都懂了还害怕什么。与此同时，本次编译原理的大实验综合了我们之前学过的好多门课程的知识，例如计组、数据结构、算法等等，不仅使我们复习了这些课程的内容，同时对这些课程与编译原理相关联的知识有了十分深入的理解和认识。

最后，从实验过程来看，不得不说，本次实验可以说是大学以来做过的一个最庞大的实验，代码量十分巨大，虽然花费了相当多的时间来做这个实验，但收获真的是很大。首先提高了编写代码的熟练性，最重要的是需要问题解决问题时的思考以及 debug 过程中的思考，我认为这些思考能够真正锻炼我们的思维，让我们获得真正的进步。

总之，我真心很感谢编译原理实验，它让我获得了思维和实践上的巨大提升。

④ 实验中遇到的问题及解决方案

在整个的实验过程中，每一部分都会需要各种各样的错误，具体我遇到了什么错误，可以参照之前各部分报告中该位置的阐述内容。但我想说的是，无论遇到什么错误，只要自己耐心的一步一步去推导，找到错误的位置，仔细研究错误的原因，最后错误一定会得到改正。

⑤ 本次实验完成情况及指导教师评语

独立完成	独立未完成	未独立	指导教师签字	特殊评语
实验中用到的特色方法及设计技巧				