编译原理课程实验报告

实验 3：语义分析

姓名	张志路	院系	计算机学院	学号	1160300909
任课教师	辛明影		指导教师	辛明影	
实验地点	格物 208		实验时间	2019 年 4 月 28 日周日 5-6 节	
实验课表现	出勤、表现得分		实验报告得分		实验总分	
	操作结果得分					

一、需求分析

得分

要求：阐述语义分析系统所要完成的功能。

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。

5. 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。

二、文法设计

得分

要求：给出如下语言成分所对应的语义动作

注：周日检查的时候由于时间比较紧，没有做嵌套过程中声明语句翻译的实现，这几天又补上了这一部分。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、分支语句“if_then_else”、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' → P

(2) P → proc id ; M0 begin D S end {addwidth(top(tblptr),top(offset)); pop(tblptr); pop(offset)}

(3) P → S

(4) S → S M S {backpatch(S1.nextlist,M.quad); S.nextlist=S2.nextlist;}

(5) M0 → ε {t = mktable(nil); push(t,tblptr); push(0,offset)}

(6) M → ε {M.quad = nextquad}

2. 声明语句（变量、数组、函数、记录声明）

(7) D → D D

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

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

(10) T → X C {T.typ = C.type; T.width = C.width;}

(11) T → record N2 D end {T.type = record(top(tblptr)); T.width = top(offset); pop(tblptr); pop(offset)}

(12) X → integer {X.type = integer; X.width = 4; t = X.type; w = X.width;}

(13) X → real {X.type = real; X.width = 8; t= X.type; w = X.width;}

(14) C → [num] C {C.type = array(num.val,C1.type); C.width = num.val*C1.width;}

(15) C → ε {C.type = t; C.width = w;}

(16) N1 → ε {t = mktable(top(tblptr)); push(t,tblptr); push(0,offset)}

(17) N2 → ε {t = mktable(nil); push(t,tblptr); push(0,offset)}

3. 表达式及赋值语句（算术表达式、数组）

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

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

(20) E → E + E1 {E.addr = newtemp(); gencode(E.addr'='E.addr'+'E1.addr);}

(21) E → E1 {E.addr = E1.addr}

(22) E1 → E1 * E2 {E1.addr = newtemp(); gencode(E1.addr'='E1.addr'*'E2.addr);}

(23) E1 → E2 {E1.addr = E2.addr}

(24) E2 → (E) {E2.addr = E.addr}

(25) E2 → - E {E2.addr = newtemp(); gencode(E2.addr'="uminus'E.addr);}

(26) E2 → id {E2.addr = lookup(id.lexeme); if E2.addr==null then error;}

(27) E2 → num {E2.addr = lookup(num.lexeme); if E2.addr==null then error}

(28) E2 → L {E2.addr = newtemp(); gencode(E2.addr'='L.array'['L.offset']');}

(29) L → id [E] {L.array = lookup(id.lexeme); if L.array==nil then error; L.type = L.array.type.elem; L.offset = newtemp(); gencode(L.offset'='E.addr'*'L.type.width);}

(30) L → 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 → B or M B1 {backpatch(B1.falselist,M.quad); B.truelist = merge(B1.truelist,B2.truelist);
B.falselist = B2.falselist}

(32) B → B1 {B.truelist = B1.truelist; B.falselist = B1.falselist}

(33) B1 → B1 and M B2 {backpatch(B1.truelist,M.quad); B.truelist = B2.truelist; B.falselist =
merge(B1.falselist,B2.falselist)}

(34) B1 → B2 {B.truelist = B1.truelist; B.falselist = B1.falselist}

(35) B2 → not B {B.truelist = B1.falselist; B.falselist = B1.truelist}

(36) B2 → (B) {B.truelist = B1.truelist; B.falselist = B1.falselist}

(37) B2 → E R E {B.truelist = makelist(nextquad); B.falselist = makelist(nextquad+1); gencode('if'
E1.addr relop.opE1.addr 'goto –'); gencode('goto –')}

(38) B2 → true {B.truelist = makelist(nextquad); gencode('goto –')}

(39) B2 → false {B.falselist = makelist(nextquad); gencode('goto –')}

(40) R → < | <=| > | >= | ==| != {B.name = op}

5. 分支语句“if_then_else”和循环语句“do_while”

(41) S → S1 {S.nextlist = L.nextlist}

(42) S → S2 {S.nextlist = L.nextlist}

(43) S1 → if B then M S1 N 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 → while M B do M S0 {backpatch(S1.nextlist,M1.quad);
backpatch(B.truelist,M2.quad);S.nextlist = B.falselist; gencode('goto'M1.quad)}

(45) S2 → if B then M S1 N 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 → if B then M S0 {backpatch(B.truelist,M.quad); S.nextlist = merge(B.falselist,S1.nextlist)}

(47) S0 → begin S3 end {S.nextlist = L.nextlist}

(48) S1 → begin S3 end {S.nextlist = L.nextlist}

(49) S2 → begin S3 end {S.nextlist = L.nextlist}

(50) S3 → S3 ; M S {backpatch(L1.nextlist,M.quad); L.nextlist = S.nextlist}

(51) S3 → S {S.nextlist = L.nextlist}

(52) N → ε {N.nextlist = makelist(nextquad); gencode('goto –')}

6. 过程调用语句call

(53) 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); S.nextlist = null;}

(54) EL → EL ,E {将 E.addr 添加到 queue 的队尾}

(55) EL → E {初始化 queue,然后将 E.addr 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
}
```

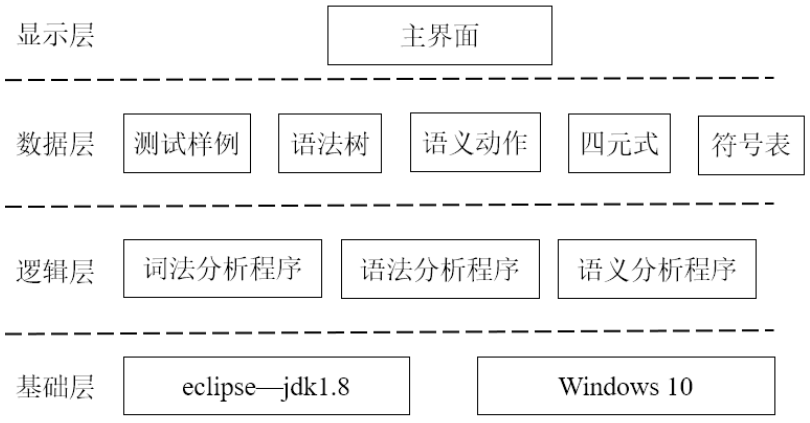
三、系统设计

得分

要求：分为系统概要设计和系统详细设计。

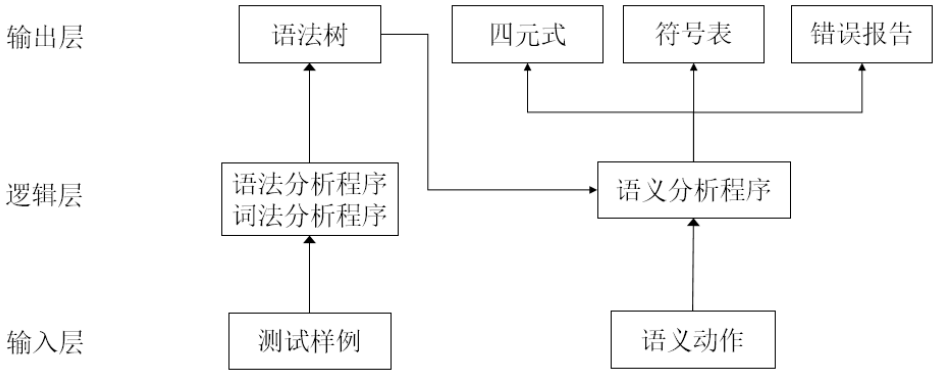
1. 系统概要设计：给出必要的系统宏观层面设计图，如系统框架图、数据流图、功能模块结构图等以及相应的文字说明。

(1) 系统框架图



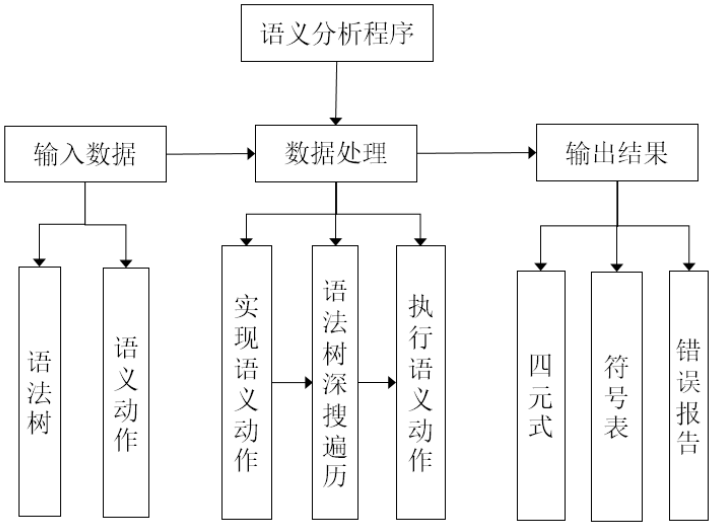
系统分为基础层、逻辑层、数据层和显示层，基础层即在 Windows10、jdk1.8 环境下利用 java 语言进行编程，逻辑层即词法分析程序、语法分析程序和语义分析程序，数据层包含测试样例、语法树、语义动作、四元式和符号表，显示层显示一个主界面。

(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) 功能模块结构图



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。

2. 系统详细设计：对如下工作进行展开描述

(1) 核心数据结构的设计

a. 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。

```java
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)
```

b. 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。

```java
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)
```

c. 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。

```java
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)
```

d. 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)
```

e. 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
```

f. 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
```

g. 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> tablesize = 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)
```

(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。

```
//  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。

```
//  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。

```
//  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。

```
//  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。

```
//  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.peek();
    //tablesize.add(off.peek());
    tblptr.pop();
    off.pop();

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

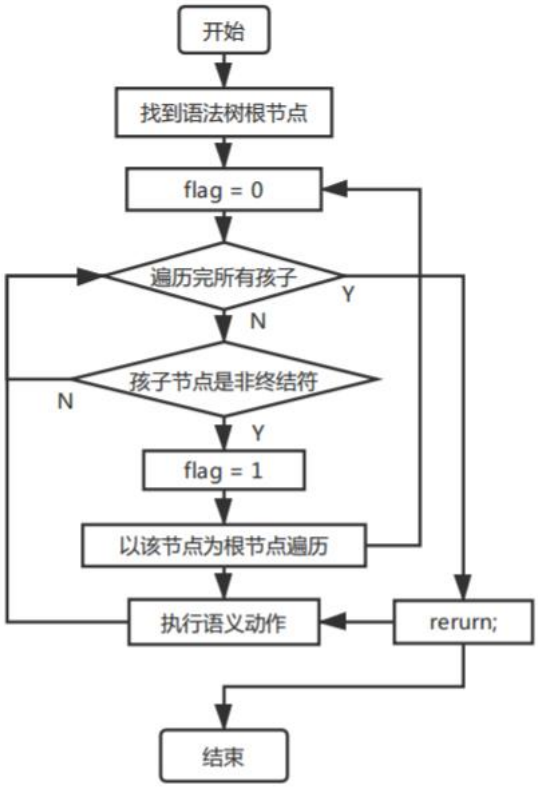
c. 深度优先搜索

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) 程序核心部分的程序流程图



四、系统实现及结果分析

得分

要求：对如下内容展开描述。

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。

2. 针对一测试程序输出其语义分析结果

测试样例如下。

```
proc fuction1;
begin
    integer f11;
    real f12;

    proc fuction2;
    begin
        integer[7][6] arr;
        integer m;
        integer n;
        integer a;
        integer b;
        integer c;
        integer d;
        real e;
        record real re1; integer re2; end r1;  // 记录

        integer x;
        integer y;
        integer z;
        integer z;  // Duplicate definition


        while a<b do
            begin
                if c<d then
                    begin
                        x=y+z;
                    end
                else
                    begin
                        x=y*z;
                    end
            end

        while a<b do
            begin
                if c<5 then
```

```
                    begin
                        while x>y do
                            begin
                                z=x+1;
                            end
                    end
                else
                    begin
                        x=y;
                    end
            end


        arr[3][5] = 2;
        m=(m+n)*9;


        e = e + a; // real = real + int , Type conversion

        call a(1,2+1,a*b);   //Common variable with call
        e1 = 7;  // e1 Undefined
        a = e2;  // e2 Undefined
        a[0] = 1;  // Non-array using array operators
        e3[9] = 1;  // e3 Undefined
        a = a + arr; // int = int + array
        // Integer variables are added to array variables

    end
    f11 = 1;
    call fuction2(1,2+1,a*b);
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	(=, intTOreala, -, 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。

3. 输出针对此测试程序经过语义分析后的符号表

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*6*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. 输出针对此测试程序对应的语义错误报告

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]

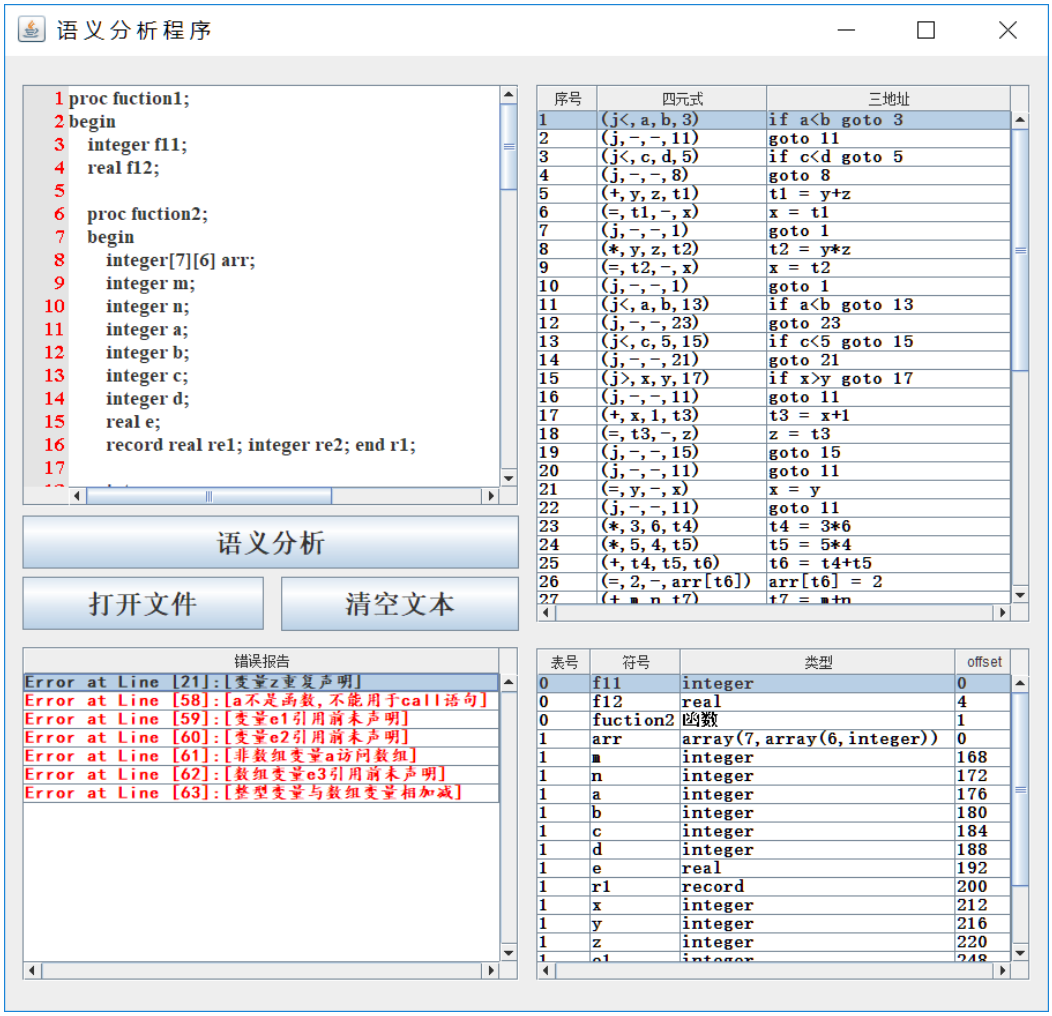
Error at Line [62]: [数组变量 e3 引用前未声明]

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。

5. 对实验结果进行分析。

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 2-4 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。

指导教师评语： 日期：