编译课系统程实验报告

实验 3：语义分析

姓名	余涛	院系	计算学部	学号	1180300829
任课教师	陈鄞		指导教师	陈鄞	
实验地点	格物 213		实验时间	2021.5.15	
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

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，并建立符号表及生成中间代码（三地址指令和四元式形式）：

声明语句（包括变量声明、数组声明、记录声明和过程声明）

表达式及赋值语句（包括数组元素的引用和赋值）

分支语句：if_then_else

循环语句：do_while

过程调用语句

（2）能够识别出测试用例中的语义错误，包括

变量（包括数组、指针、结构体）或过程未经声明就使用

变量（包括数组、指针、结构体）或过程名重复声明

运算分量类型不匹配（也包括赋值号两边的表达式类型不匹配）

操作符与操作数之间的类型不匹配

赋值号左边出现一个只有右值的表达式

数组下标不是整数

对非数组变量使用数组访问操作符

对非结构体类型变量使用“.”操作符

对非过程名使用过程调用操作符

过程调用的参数类型或数目不匹配

函数返回类型有误

能准确给出错误所在位置。输出的错误提示信息格式如下：

Semantic 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

输出三地址指令和四元式序列，格式如下图所示（以输入语句“while a<b do if c<d then x=y+z else x=y-z”为例）：

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

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

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. 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
}
```

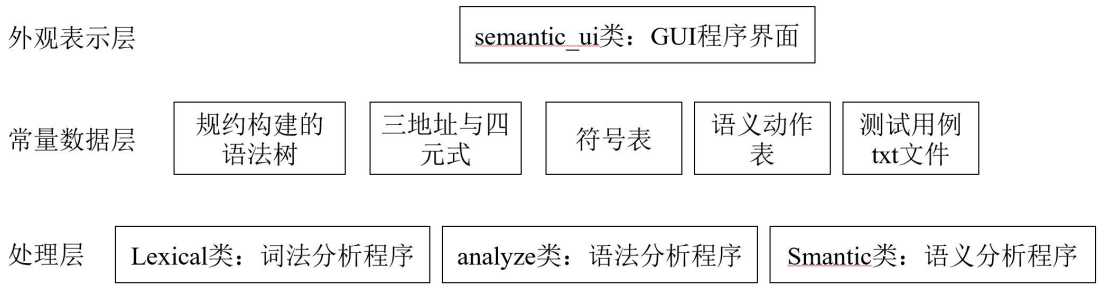
三、系统设计	得分	

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。

1.1 系统框架图：

整个系统包括三个层次：外观表示层、常量数据层、处理层。外观表示层负责输出三地址和四元式、符号表与错误信息的输出显示。常量数据层中包含五个部分：实验二构建的语法树、实验三生成的符号表、三地址和四元式、语义动作表、测试用的 txt 测试文件。处理层包含三个部分，分别是词法分析程序、语法分析程序、语义分析程序，三个程序共同共同协作最后完成语义分析。



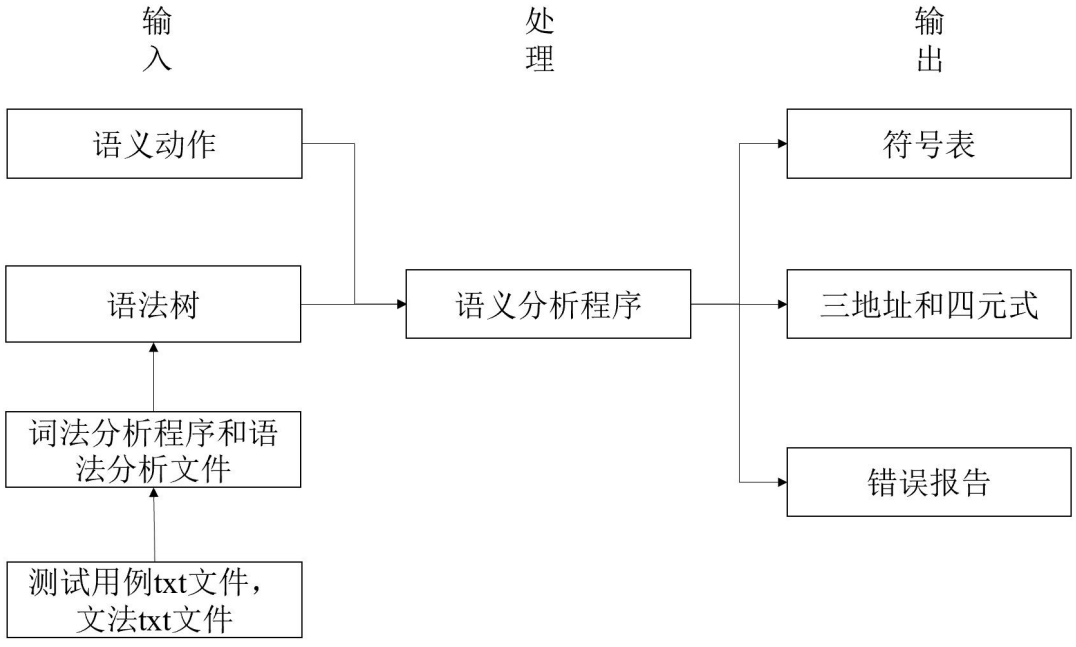
1.2 数据流图：

数据流图包括三个部分：输入、处理、输出。

输入：首先需要输入测试用例的 txt 文件，对于相应为文法，调用词法分析分析程序和语法分析程序生成语法树。

处理：使用语义分析程序来对语法树执行相应的语义动作。

输出：对语义分析后符号表、三地址和四元式、错误报告进行 GUI 界面的输出。



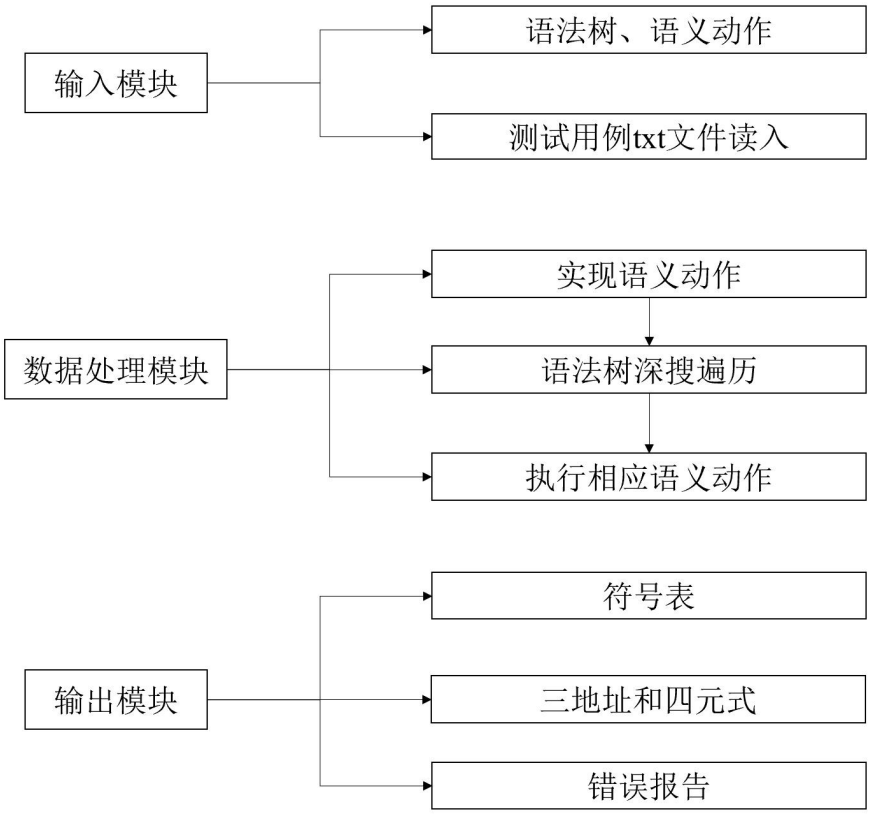
1.3 功能模块结构图：

整个系统一共包括三个模块：输入模块、数据处理模块、输出模块

输入模块：负责通过 GUI 读入测试 txt 文件中的测试用例，并调用实验一的词法分析程序进行词法分析生成 Token 序列，同时读入文法 txt 文件然后得到语法树，并且创建语义动作。

数据处理模块：根据语法树从根节点深搜遍历，通过语义分析程序执行相应的语义动作。

输出模块：负责通过 GUI 输出符号表、三地址和四元式、错误报告。



（2）系统详细设计：对如下工作进行展开描述

✓ 核心数据结构的设计

1.Treenode 类：

这个类用来构成一个树的节点，在本次实验中使用邻接表来储存树，所以只需要在树的节点中，具体数据结构如下：

```
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
```

2.Tree 类：

对于实验二的规约过程生成语法树，使用邻接表储存语法树，Tree 类用于构建邻接表的一行，字段包括父节点和其所有的孩子，具体数据结构如下：

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

3.Symbol 类：

这个类用于储存符号表里面的每一个元素，字段包括符号名、符号类型和偏移量，具体数据结构

如下：

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

4.FourAddr 类：

这个类用于储存四元式表中的每一个元素，字段包括操作符、参数一、参数二和地址，具体数据结构如下：

```
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
```

5.Array 类：

这个类用来创建一个多维数组，字段包括长度、数组类型和基本类型，具体数据结构如下：

```
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
```

6.Properties 类：

这个类用来储存语法树上的每一个节点相应的属性，通过相应的语义动作能够得到各自的属性，字段在以下的注释中给出，具体数据结构如下：

```
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<>();  // 列表
    private List<Integer> falselist = new ArrayList<>();  // 列表
    private List<Integer> nextlist = new ArrayList<>();  // 列表
```

7.Smantic 类：

使用上述提到的数据结构实现了所有的语义动作，然后对语法树进行深搜遍历，进行语义分析，储存所有的三地址指令序列、四元式指令序列、错误报告序列，字段在以下的注释中给出，具体

数据结构如下：

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

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

    static List<String> three_addr = new ArrayList<>();  // 三地址指令序列
    static List<FourAddr> four_addr = new ArrayList<>();  // 四元式指令序列
    static List<String> errors = new ArrayList<>();  // 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<>();  // 过程调用参数队列
    static Stack<Integer> tblptr = new Stack<>();  // 符号表指针栈
    static Stack<Integer> off = new Stack<>();  //  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
```

✓ 主要功能函数说明

1.实现语义动作需要用到的一些函数：

1.1 向符号表中增加元素：

```
1.  /**
2.      * 向符号表中增加元素
3.      *
4.      * @param i      第 i 个符号表
5.      * @param name   元素名字
6.      * @param type   元素类型
7.      * @param offset 偏移量
8.      */
9.     private static void enter(int i, String name, String type, int offset) {
10.        if (table.size() == 0) {
11.            table.add(new Stack<Symbol>());
12.        }
13.        Symbol s = new Symbol(name, type, offset);
```

```
        table.get(i).push(s);
    }
```

1.2 查找符号表，查看变量是否存在：

```
/**
     * 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;
    }
```

1.3 新建一个变量：

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

1.4 回填地址

```
/**
     * 新建一个变量
     *
```

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

1.5 合并列表：

```
/**
     * 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;
    }
```

1.6 返回下一条指令地址：

```
/**
     * 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;
    }
```

1.7 新建包含 i 的列表并返回：

```
/**
     * 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<>();
        a1.add(i);
```

```
        return a1;
    }
```

1.8 新增一个符号表：

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

2.语义动作函数：

当前面的所有模块构建完成之后，就可以构建相应的语义动作函数，将对应的语义动作用代码呈现即可。语义动作特别多，下面选取了几个具有代表性的语义动作进行展示

2.1 变量声明：

```
// 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);
    }
```

```
// 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);
        }
    }
```

```
// 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);
    }
```

2.2 算术表达式：

```
//  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));
    }
```

2.3 布尔表达式：

```
// 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);
}
```

2.4 控制流语句：

```
// 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);
}
```

2.5 函数调用：

```
1.  //  S -> call id ( EL )
2.    //  {n=0; for queue 中的每个 t do {gencode('param't); n=n+1}
3.    //   gencode('call'id.addr','n);}
4.    public static void semantic_54(Tree tree) {
5.       int S = tree.getFather().getId();  // S
6.       String id = tree.getChildren().get(1).getValue();  // id
7.       int[] index = lookup(id);
8.
9.       if (!table.get(index[0]).get(index[1]).getType().equals("函数")) {
10.          String s = "Error at Line [" + tree.getChildren().get(0).getLine()
11.                + "]:\t[" + id + "不是函数,不能用于 call 语句]";
12.          errors.add(s);
13.          Properties a1 = new Properties();
14.          a1.setNext(new ArrayList<Integer>());
15.          tree_pro.set(S, a1);
16.          return;
17.       }
18.
19.       int size = queue.size();
20.       for (int i = 0; i < size; i++) {
21.          String code = "param " + queue.get(i);
22.          three_addr.add(code);
23.          four_addr.add(new FourAddr("param", "-", "-", queue.get(i)));
24.       }
25.       String code = "call " + id + " " + size;
26.       three_addr.add(code);
27.       four_addr.add(new FourAddr("call", String.valueOf(size), "-", id));
28.
29.       Properties a1 = new Properties();
30.       a1.setNext(new ArrayList<Integer>());
31.       tree_pro.set(S, a1);
32.    }
```

2.6 函数嵌套：

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

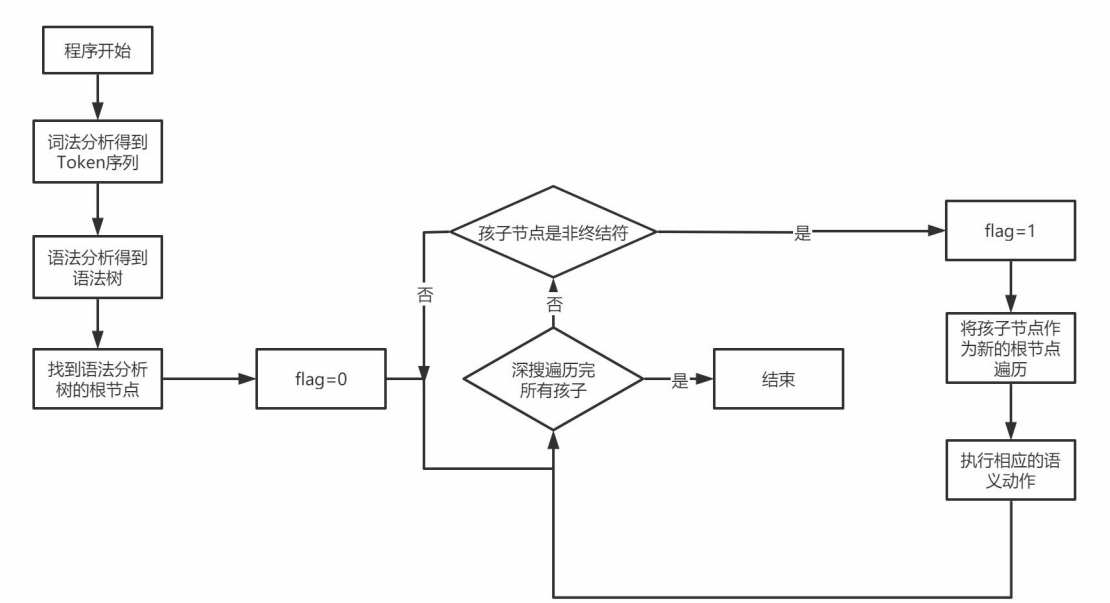
```
        tblptr.pop();
        off.pop();

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

3.深度优先搜索：

对于规约过程中建立的语法树，进行深搜遍历，对搜索到的每个节点执行对应的语义动作，深搜结束，语义分析完成，深搜代码如下：

```
/**
     * 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;
        }
    }
```

✓ 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；

（1.1）回填

为了实现对于分支语句和循环语句的一趟扫描就完成语法分析，需要使用回填技术。每当生成一个跳转指令，不指定该跳转指令的目标标号，将所有这样的指令全都放入跳转指令组成的列表中，在同一个列表中的所有的跳转指令都具有相同的目标标号。当能够确定正确的目标标号时，然后才去填充这些指令的目标标号。

（1.2）语法树

由于在实验二建立了语法树，所以可以重新使用深搜来遍历语法树，每从叶节点往上规约出一个产生式，就让产生式执行对应的语法动作。这样会导致程序效率变低，但却巧妙地利用上了实验二产生的语法树的内容。

（1.3）错误处理

对于遇到错误的情况，选择忽略掉该错误，继续执行。部分实验指导书中给出的错误：“赋值号左边出现一个只有右值的表达式”，“数组下标不是整数”属于语法错误，在实验二中已经给出了。并且有的错误忽略掉的话会抛出异常，所以对于错误节点选择定义一个值来防止异常影响后续的处理。

（2） 针对一测试程序输出其语义分析结果；

测试用例如下：

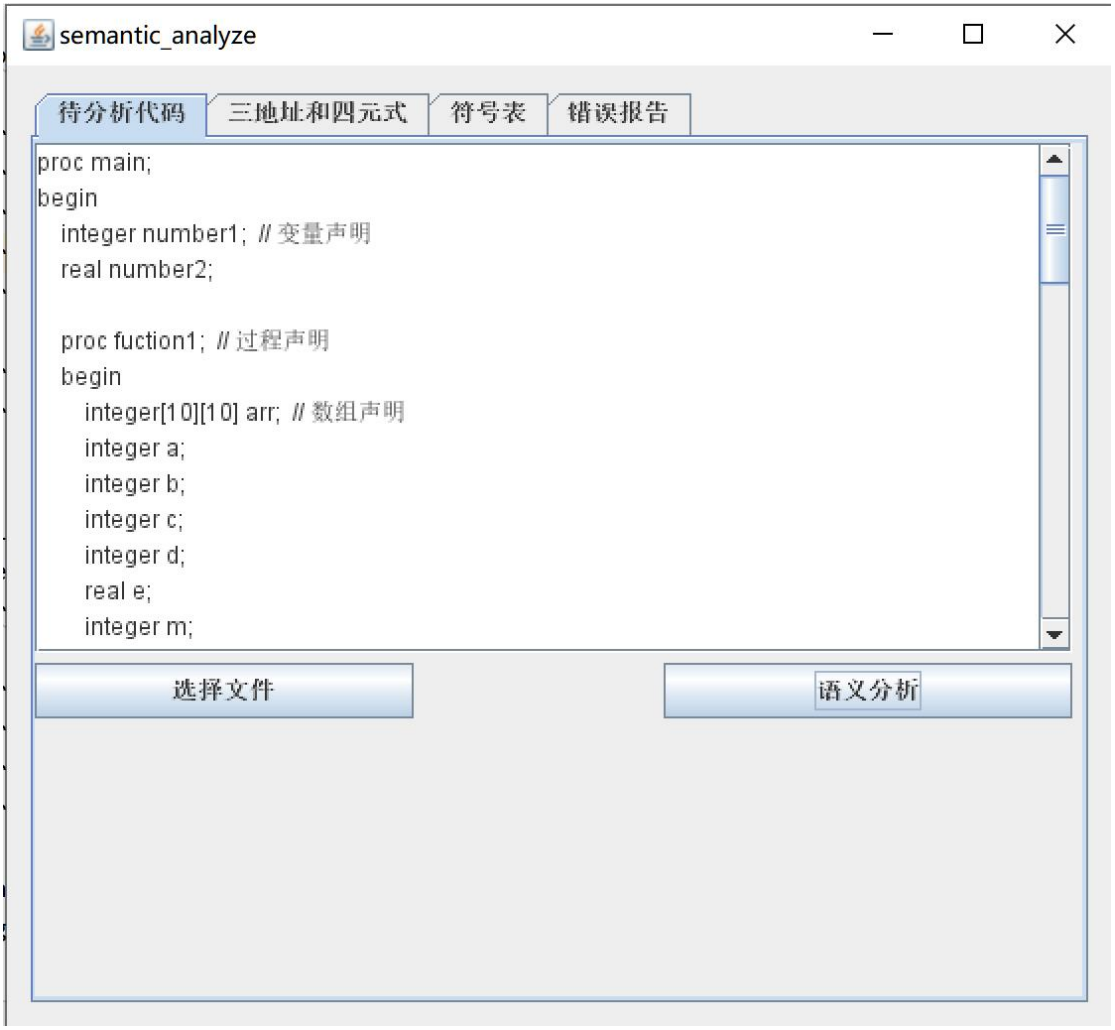
```
1.  proc main;
2.  begin
3.      integer number1;  // 变量声明
4.      real number2;
```

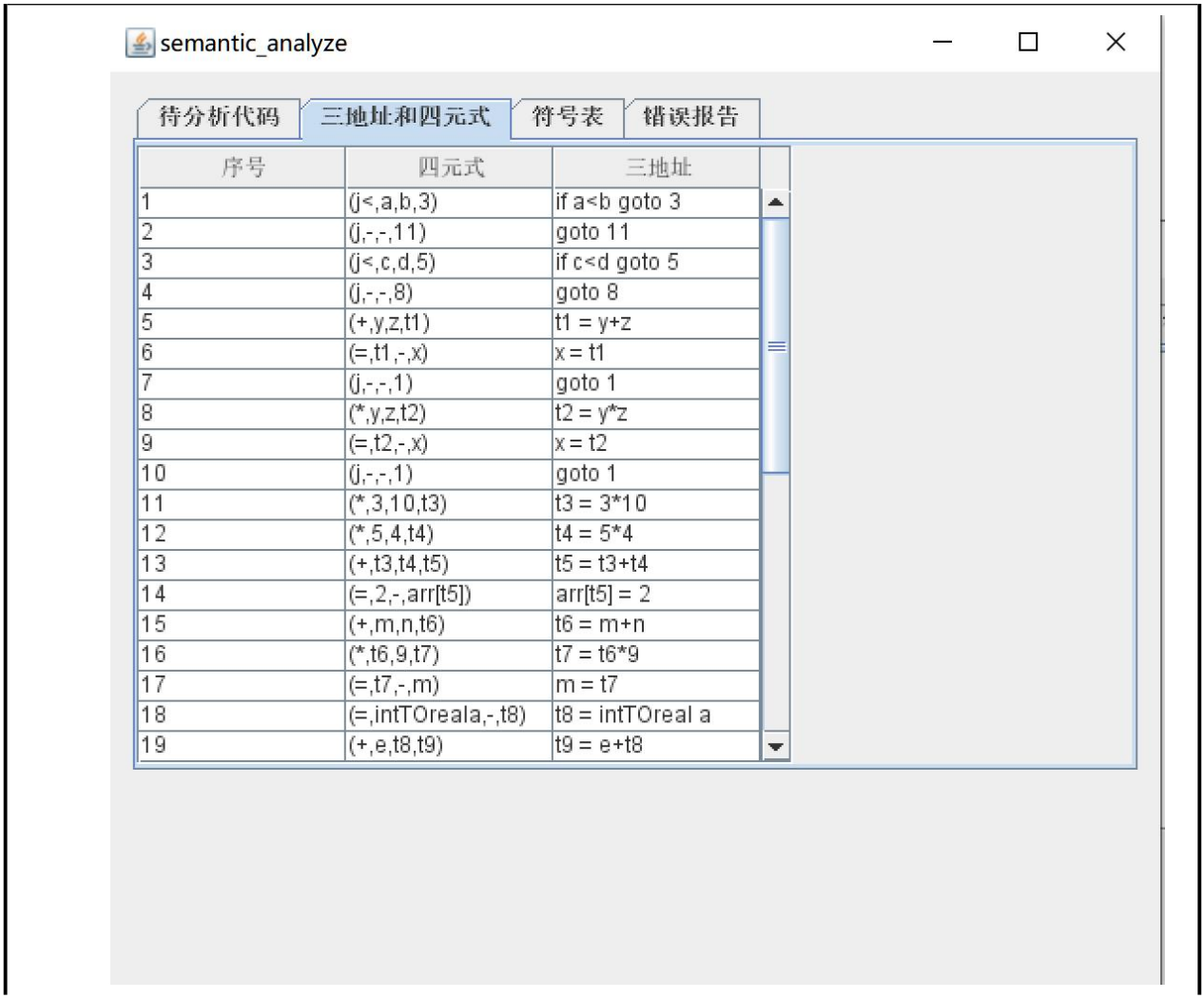
```
5.
6.      proc fuction1;  // 过程声明
7.      begin
8.          integer[10][10] arr;  // 数组声明
9.          integer a;
10.         integer b;
11.         integer c;
12.         integer d;
13.         real e;
14.         integer m;
15.         integer n;
16.
17.         record
18.           real myrecord1;
19.           integer myrecord2;
20.         end myrecord;  // 记录声明，类似于结构体
21.
22.         integer x;
23.         integer y;
24.         integer z;
25.         integer z;  // 变量重复声明
26.
27.         while a < b do  // 循环语句和表达式
28.           begin
29.             if c < d then  // 分支语句和表达式
30.               begin
31.                 x = y + z;
32.               end
33.             else
34.               begin
35.                 x = y * z;
36.               end
37.           end
38.         arr[3][5] = 2;  // 数组赋值
39.         m = ( m + n ) * 9;  // 表达式及赋值语句
40.         e = e + a; // real = real + int , 类型转换
41.
42.         call a(1, 2+1, a*b);   // 对非过程名使用过程调用操作符
43.         error_number1 = 7;  // error_number1 变量未经声明就使用
44.         a = error_number2;  // error_number2 变量未经声明就使用
45.         a[0] = 1;  // a 非数组变量使用数组访问操作符
46.         error_number3[9] = 1;  // error_number3 数组变量未经声明就使用
47.         a = a + arr; // int = int + array, 运算分量类型不匹配
```

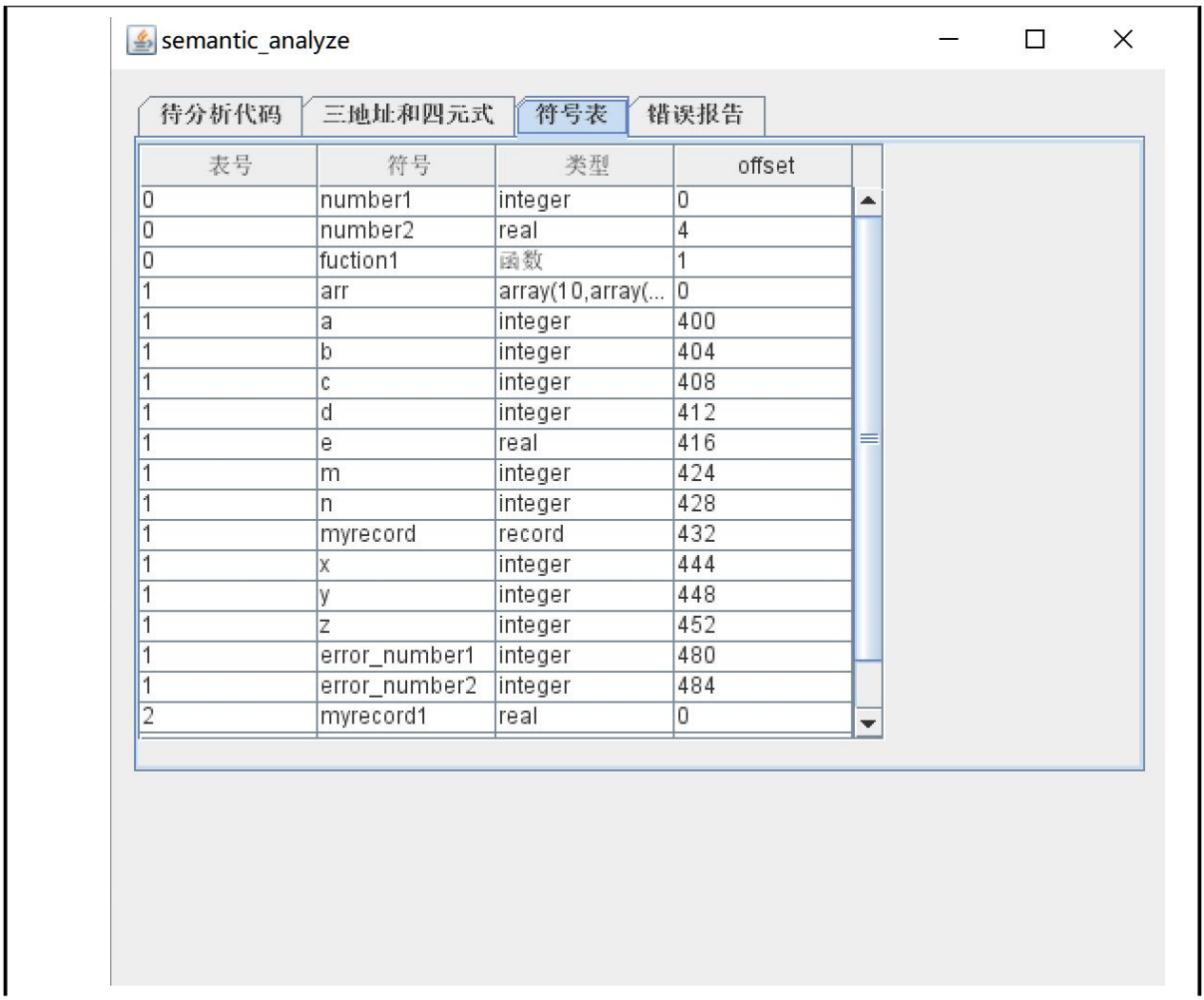
```
48.     end
49.
50.     number1 = 1;
51.     call fuction1(1, 2+1, a*b);  // 过程调用
52. end
53.
```

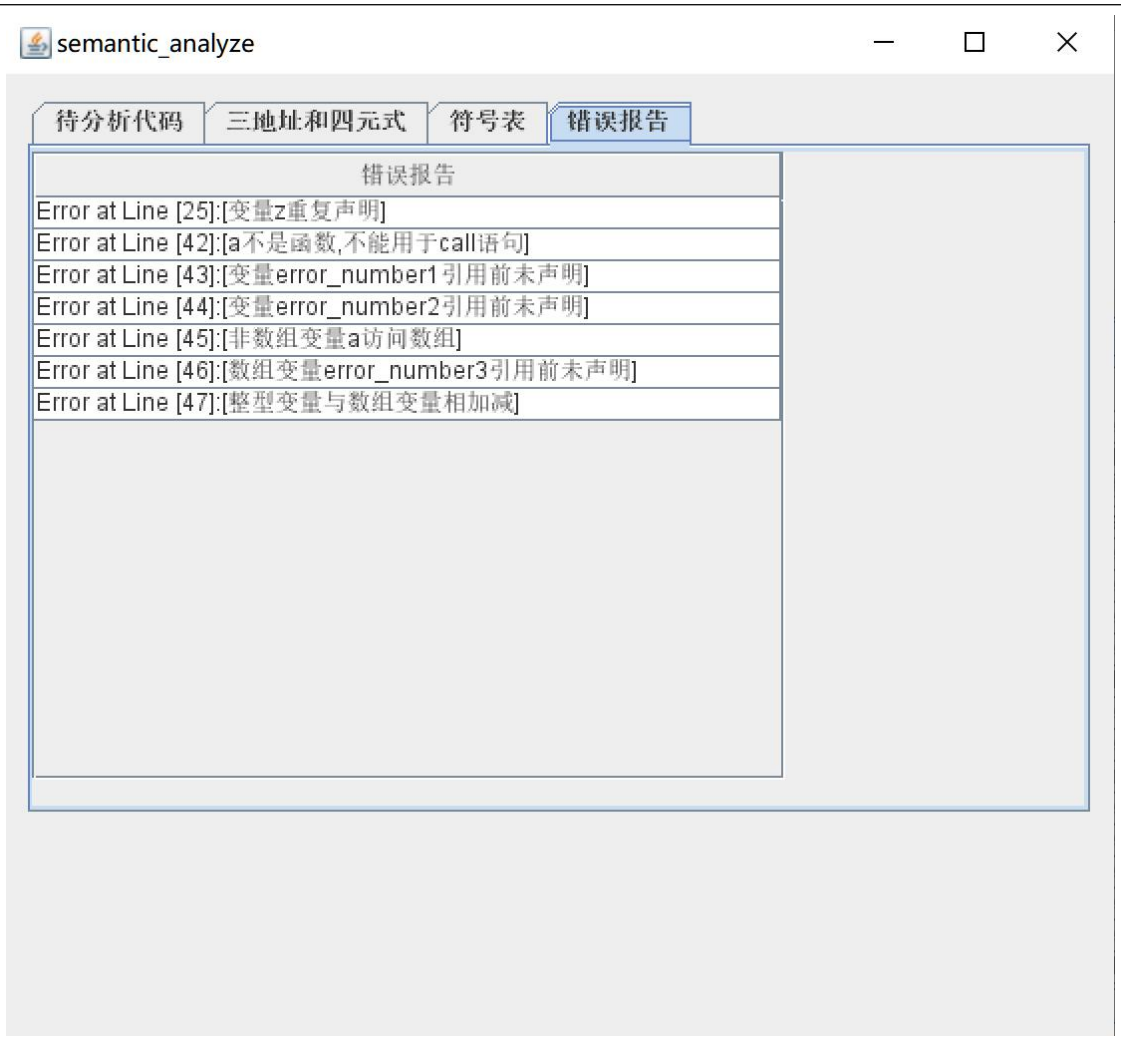
该测试用例涵盖了实验指导书中给出的各种需要分析的语句及语法错误

执行结果如下：



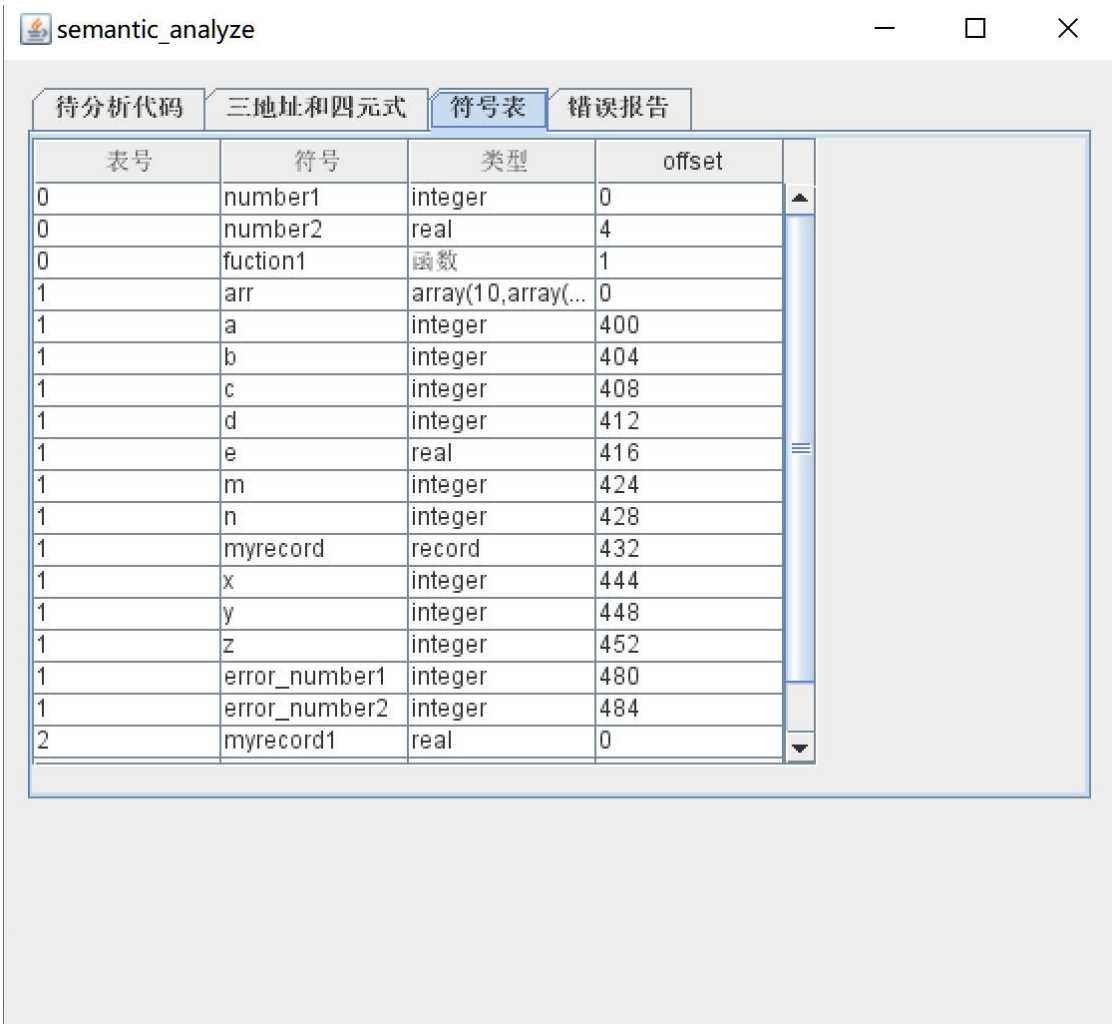

semantic_analyze
待分析代码
三地址和四元式
符号表
错误报告
序号
四元式
三地址
1 (j<,a,b,3) if a<b goto 3
2 (j,-,-,11) goto 11
3 (j<,c,d,5) if c<d goto 5
4 (j,-,-,8) goto 8
5 (+,y,z,t1) t1 = y+z
6 (=,t1,-,x) x = t1
7 (j,-,-,1) goto 1
8 (*,y,z,t2) t2 = y*z
9 (=,t2,-,x) x = t2
10 (j,-,-,1) goto 1
11 (*,3,10,t3) t3 = 3*10
12 (*,5,4,t4) 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 (*,t6,9,t7) t7 = t6*9
17 (=,t7,-,m) m = t7
18 (=,intTOreala,-,t8) t8 = intTOreal a
19 (+,e,t8,t9) t9 = e+t8


semantic_analyze
待分析代码
三地址和四元式
符号表
错误报告
表号
符号
类型
offset
0 number1 integer 0
0 number2 real 4
0 fuction1 函数 1
1 arr array(10,array(... 0
1 a integer 400
1 b integer 404
1 c integer 408
1 d integer 412
1 e real 416
1 m integer 424
1 n integer 428
1 myrecord record 432
1 x integer 444
1 y integer 448
1 z integer 452
1 error_number1 integer 480
1 error_number2 integer 484
2 myrecord1 real 0



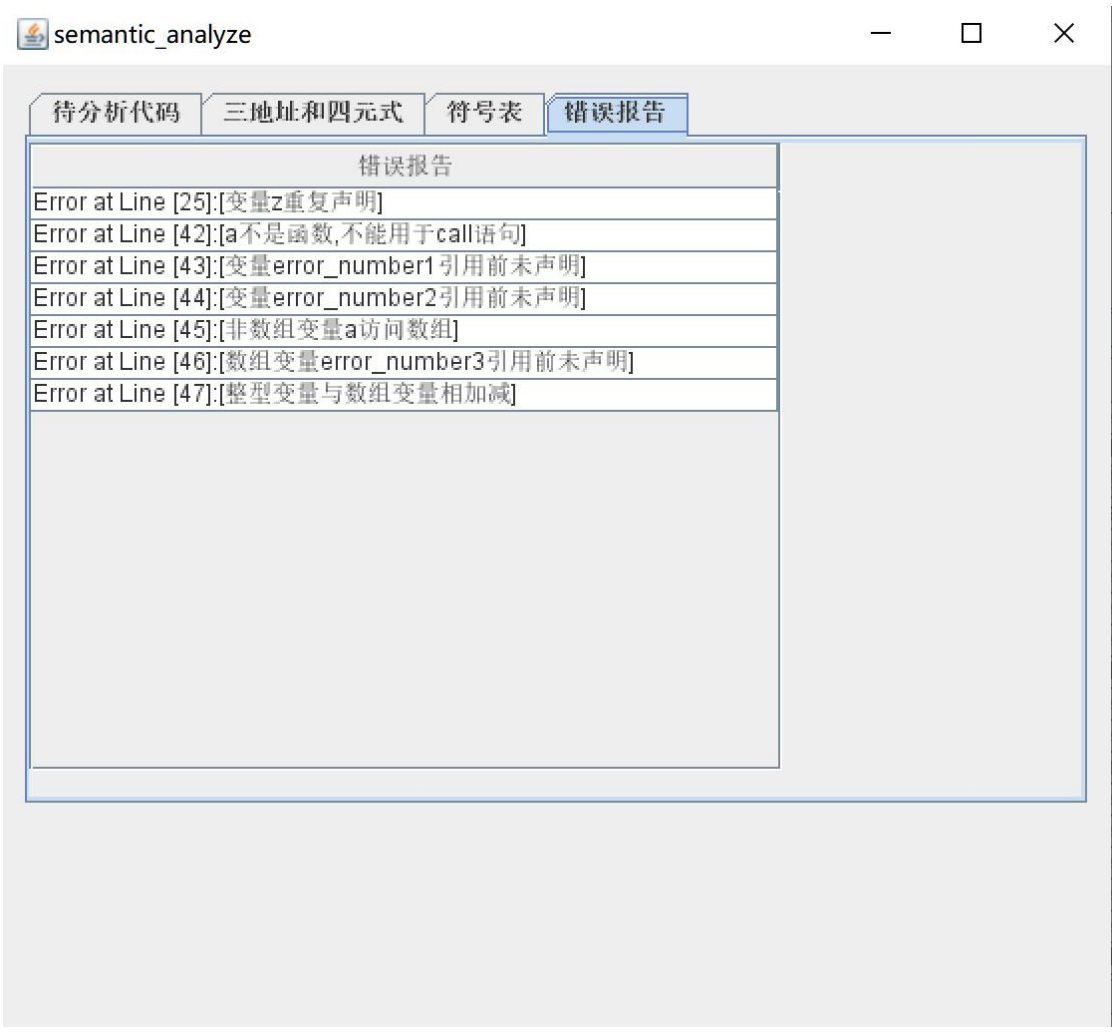
（3） 输出针对此测试程序经过语义分析后的符号表；

符号表如下：



（4） 输出针对此测试程序对应的语义错误报告；

错误报告如下：



（5） 对实验结果进行分析。

通过对三地址和四元式、符号表、错误报告进行分析，能够发现实验结果涵盖了实验指导书的要求：

三地址和四元式、符号表分析后，识别了如下语句：

声明语句（包括变量声明、数组声明、记录声明和过程声明）

表达式及赋值语句（包括数组元素的引用和赋值）

分支语句：if_then_else

循环语句：do_while

过程调用语句

错误报告分析后，识别了以下错误：

变量（包括数组、指针、结构体）或过程未经声明就使用

变量（包括数组、指针、结构体）或过程名重复声明

运算分量类型不匹配（也包括赋值号两边的表达式类型不匹配）

操作符与操作数之间的类型不匹配

赋值号左边出现一个只有右值的表达式（语法分析的错误）

数组下标不是整数（语法分析的错误）

对非数组变量使用数组访问操作符

对非结构体类型变量使用“.”操作符

对非过程名使用过程调用操作符

过程调用的参数类型或数目不匹配

函数返回类型有误

所以此语义分析程序基本完成了实验指导书的要求。

注：其中的测试样例需先用已编写的词法分析程序进行处理。

指导教师评语：

日期：