实验六、语义分析和中间代码生成实验

> *学号：1120180207*
>
> *姓名：唐小娟*
>
> *班级：07111801*

1. 实验目的

（1）熟悉 C 语言的语义规则，了解编译器语义分析的主要功能，了解编译器中间代码表示形式和方法；

（2）掌握语义分析模块构造和中间代码生成的相关技术和方法，设计并实现具有一定分析功能的 C 语言语义分析模块以及针对某种中间代码的编译器模块；

（3）掌握编译器从前端到后端各个模块的工作原理，语义分析模块、中间代码生成模块与其他模块之间的交互过程。

2. 实验内容

2.1 语义分析

语义分析阶段的工作为基于语法分析获得的分析树构建符号表，并进行语义检查。如果存在非法的结果，请将结果报告给用户，其中语义检查的内容主要包括：

- 变量使用前是否进行了定义；
- 变量是否存在重复定义；
- break 语句是否在循环语句中使用；
- 函数调用的参数个数和类型是否匹配；
- 函数使用前是否进行了定义或者声明；
- 运算符两边的操作数的类型是否相容；
- 数组访问是否越界；
- goto 的目标是否存在；

由于语义分析查错内容过多，这里就以老师给的示例代码为范围，除了数组访问越界没有实现，其他都基本实现。

2.2 中间代码生成

以自行完成的语义分析阶段的抽象语法树为输入，或者以 BIT-MiniCC 的语 义分析阶段的抽象语法树为输入，针对不同的语句类型，将其翻译为中间代码序列。本实验是采用了四元式的中间代码形式。

3. 实验过程

由于这两次实验的合并，语义分析和中间代码生成我在遍历树的过程中一次性完成，在介绍实验过程中，分两项内容阐述。

利用ObjectMapper对象的方法readValue得到树的根节点，调用接口ASTVisitor进行树的遍历，并访问相应AST类的节点信息。

3.1 语义分析

在语法分析实验的基础上，构建符号表，并基于符号表和抽象语法树进行语义检查。要求对于给定的程序，输出检测到的语义错误信息并且输出信息中包含错误编号。

3.1.1 符号表构建

根据变量、函数定义、函数体等等的特点不一样，以及由于复合语句{}等引起的嵌套层次关系导致的作用域问题。我设置了4类Symbol，并且为了方便得到部分信息，我采用了Map机制。

（1）Symbol：涵盖了所有符号，属性信息如下。

```
class Symbol {

    String name;//名字
    String kind;//种类，包括变量、常量、函数定义、数组和标号等
    String type;//int、double类型

    int link;   //上一个

    List<Symbol> params; //争对函数调用里面的参数

    int paraId;
    int lbId;
    int arrId;

    //为了方便测试
    public void print()
    {
        System.out.printf("
{name:%s;kind:%s;type:%s;link:%d}\n",name,kind,type,link);
    }

}
```

（2）BodySymbol：说明该层次所包含的变量信息。

```
class BodySymbol {
    int lastPar; //最后一个形参在SymbolTabsbles的位置
    int last;   //最后一个变量在SymbolTabsbles的位置
    int pSize;  //总形式参数变量大小
    int vSize;  //总变量大小
}//函数体body
```

（3）DisplaySymbol：说明代码嵌套关系。对应的元素是嵌套代码在bodySymbols的位置（下标）

```
class DisplaySymbol{
    Stack<Integer> disTab; //层次关系
    DisplaySymbol(){
        disTab = new Stack<Integer>();
    }
}
```

（4）LabelSymbol：为了给goto语句和label语句使用的标号内容。

```
class LabelSymbol{
    int id; //唯一标识
    int ir_id;//中间代码的label值
    boolean f;//定义性出现还是使用性出现
}
```

根据上面四类符号的种类，我设置了几个对象，在遍历树的过程中，遇到声明语句需要将符号表存入到其中，进行检查。

```
    public List<Symbol> SymbolTables = new ArrayList<>();
    public DisplaySymbol displaySymbols = new DisplaySymbol();
    public List<LabelSymbol> labelSymbols = new ArrayList<>();
    public List<ArrayTable> arrayTables = new ArrayList<>();
    public List<BodySymbol> bodySymbols = new ArrayList<>();

    Map<ASTNode, Symbol> mapSb = new HashMap<>();
    Map<String,String> mapType = new HashMap<>();
    Map<Symbol,Integer> mapGotolb = new HashMap<>();
    Map<ASTArrayDeclarator,Integer> mapArrRef = new HashMap<>
();

    boolean flag = false;
    public int fIter = 0;//对循环语句标记，看是否在循环语句里面
    public int freturn = 0;
```

3.1.2 符号表检查

这里以有无重复定义和是否定义两个检查作为说明。

（1）检查有无重复定义变量

- 得到当前层次的BodySymbol对象bs，因为检查是处于最上层，直接利用displaySymbol的栈即可。
- 得到bs的最后一个变量在SymbolTables的位置，根据link关系，不断找到该层次作用域中的前一个变量，检查name是否一致。
- 由于label和goto的特殊性，变量可以在goto语句之后才声明，这里特殊处理：检查它所对应的labelSymbol中的属性f是否为false（也就是定义性出现）。

```
//有无重复定义变量
boolean checkRepeat(Symbol sb)
{
    int btable_num = displaySymbols.disTab.peek();
    //System.out.println("--"+btable_num);
    int j =0;
    int id = bodySymbols.get(btable_num).last;
    //      System.out.println(id);
    //
System.out.println(bodySymbols.get(btable_num).vSize);
    while(j<bodySymbols.get(btable_num).vSize)
    {
        Symbol temp_sb = SymbolTables.get(id);
        //temp_sb.print();

 if(sb.name.equals(temp_sb.name)&&sb.kind.equals(temp_
sb.kind)){
            if(sb.kind == "FunctionDefine")
                //如果包含，则表示重复
                printError(sb.kind+" "+sb.name+" is
defined.",2);
            else if(sb.kind == "LabeledStatement")
            {
                LabelSymbol ls =
labelSymbols.get(temp_sb.lbId);
                if(ls.f)
                {
                    //定义性出现
                    printError("Label: "+sb.name+" has
been defined",1);
                }
                else //goto、未定义
                {
                    mapGotolb.put(sb,ls.ir_id);
                    return false;
                }

            }
            else
```

```
34                  printError(sb.kind+" "+ sb.name + "
    has been declarated.",2);
35              return false;
36          }
37          j++;
38          id = temp_sb.link;
39      }
40      return true;
41  }
```

（2）检查变量是否定义

- 不断根据层次表中的层次关系从下往上得到bodysymbol
- 根据bodysymbol的last得到最后一个变量的位置，依次找到该范围的所有变量进行查找
- 如果查到了，看是否是函数调用，检查参数个数和类别是否一致；看是否是goto语句，并且要得到相应标签的中间代码标记。
- 如果没有查到，看是否是goto语句类型的标签，如果是，则不用输出错误信息。（因为可能会有标签变量在后面定义的情况）

```
1   boolean checkIn(Symbol sb) {
2       //sb.print();
3       Symbol sb_res = new Symbol();
4       Stack<Integer> temp_s = new Stack<>();
5       BodySymbol btable_res = new BodySymbol();
6       boolean res = true;
7       boolean f = false;
8       int ksize = displaySymbols.disTab.size();
9       for (int i = 0; i < ksize; i++) {
10          int btable_num = displaySymbols.disTab.pop();
11          //System.out.println(btable_num);
12          temp_s.push(btable_num);
13          int j = 0;
14          int id = bodySymbols.get(btable_num).last;
15          while (j < bodySymbols.get(btable_num).vSize) {
16              Symbol sb_ed = SymbolTables.get(id);
17              //sb_ed.print();
18              //如果变量名相等
19              if (sb.name.equals(sb_ed.name)) {
20                  f = true;
21                  sb_res = sb_ed;
22                  break;
23              }
24              id = sb_ed.link;
25              j++;
26          }
27
28          if (f) {
29              btable_res = bodySymbols.get(btable_num);
30              break;
31          }
```

```
32
33      }
34      if (f) {
35          //sb.print();
36          //找到了，看匹配
37          if (sb.kind == "FunctionCall") {
38              btable_res = bodySymbols.get(sb_res.paraId);
39              //先判断参数长度
40              //sb_res.print();
41              if (btable_res.pSize != sb.params.size()) {
42                  printError(sb.kind + ":" + sb.name + "'s" + "
    param num is not matched.", 4);
43                  res =  false;
44              }
45              else{
46                  //System.out.println(temp.param.size());
47                  //再判断参数类型是否匹配
48                  int j = 0;
49                  int id = btable_res.lastPar;
50
51                  while (j < btable_res.pSize) {
52                      Symbol temp_sb = SymbolTables.get(id);
53                      if (!checkMatch(temp_sb, sb.params.get(j)))
    {
54                          printError(sb.kind + ":" + sb.name +
    "'s" + " param type is not matched.", 4);
55                          res = false;
56                          break;
57                      }
58                      j++;
59                      id = temp_sb.link;
60                  }
61              }
62          }else if(sb.kind == "GotoStatement"){
63
     mapGotolb.put(sb,labelSymbols.get(sb_res.lbId).ir_id);
64          }
65          res = true;
66      }else if(sb.kind == "GotoStatement"){
67          //没有检查到
68          res = false;
69
70      }
71      else {
72          if (sb.kind == "FunctionCall")
73              printError(sb.kind + " " + sb.name + " is not
    declarated.", 1);
74          else
75              printError(sb.kind + " " + sb.name + " is not
    defined.", 1);
76          res =  false;
77      }
```

```
78      //再恢复displaytable
79      ksize = temp_s.size();
80      for(int i=0;i<ksize;i++)
81      {
82          displaySymbols.disTab.push(temp_s.pop());
83      }
84      mapType.put(sb.name,sb_res.type);
85      return res;
86  }
```

3.1.3 变量有无定义

由于函数和循环语句等的嵌套关系，不同作用域里可以传递变量的定义。因此在搜寻查找的时候，需要根据层次表得到每个层次里的变量去检查有无定义。如果检查到了，则停止，否则则给出相应错误提示。除此之外，对于一些特殊情况，例如label语句和goto语句，变量在出现后定义也是可以的；而函数调用需要检查参数个数和参数类型。

这里每一次使用变量之前都需要检查有无定义。这里以表达式作为例子：

```
1           visit(postfixExpression.expr);
2           Symbol sb = new Symbol();
3           sb.name = mapSb.get(postfixExpression.expr).name;
4           sb.kind = postfixExpression.expr.getType();
5           checkIn(sb);
6
7   public void visit(ASTIdentifier identifier) throws Exception {
8           map.put(identifier, identifier);
9           //放符号表
10          Symbol symbol = new Symbol();
11          symbol.name = identifier.value;
12          mapSb.put(identifier,symbol);
13      }
```

先访问相应的子表达式，如果是identifier类型，则会得到相应的名字，然后调用checkIn函数检查该使用的变量有无定义。

3.1.4 变量有无重复定义

在同一作用域里，变量不允许重复定义。因此在检查的时候只需要看当前层次的变量即可。但是label语句需要特殊考虑，对于使用性出现的，就可以不报错，得到相应的中间代码标识返回即可。

这里在每一次变量定义的时候，都需要检查是否已经被定义了。

```
1   public void visit(ASTDeclaration declaration) throws Exception
    {
2           // TODO Auto-generated method stub
3           BodySymbol bs =
    bodySymbols.get(displaySymbols.disTab.peek());
4           //System.out.println(displaySymbols.disTab.peek());
```

```
        Symbol sb = new Symbol();
        sb.type = "";
        for(int i=0;i<declaration.specifiers.size();i++)
        {
            visit(declaration.specifiers.get(i));
            sb.type +=
mapSb.get(declaration.specifiers.get(i)).type;
        }
        for(int i=0;i<declaration.initLists.size();i++)
        {
            visit(declaration.initLists.get(i));
            Symbol sb_res =
mapSb.get(declaration.initLists.get(i));
            sb_res.type = sb.type;
            sb_res.link = bs.last;
            //判断有无重复
            if(checkRepeat(sb_res)){
                //sb_res.print();
                SymbolTables.add(sb_res);
                bs.vSize++;
                bs.last = SymbolTables.size()-1;
                //System.out.println(bs.last);
            }
        }
    }
```

首先得到该声明的类型，并且要传递到接下来声明的每一个变量中，在`visit(declaration.initLists.get(i));`得到相应变量名，之后判断有无重复，如果没有重复，需要添加到总符号表（`SymbolTables.add(sb_res)`）并且修改当前层次的last和vsize值。

3.1.5 break和continue有无在循环语句中

这里采用类似括号匹配的原则，每次进入一个循环体，fIter++，出循环体，fIter--，每次访问到break语句的时候判断fIter是否大于0；同理continue语句也是如此。

```
//访问循环体内部
fIter++;
visit(iterationStat.stat);
fIter--;
if(fIter<=0)
{
    //说明break没有在循环体里面
    printError("BreakStatement:must be in a LoopStatement.",3);
}
```

3.1.6 函数调用参数是否匹配

在调用函数的时候，首先检查函数调用名是否已经被定义，之后检查参数个数是否一致，最后检查参数类型是否匹配。这里为了简便，我直接在Symbol里面添加了params属性，代表函数调用所传入的参数。这里只需要关注类型即可。

这里的参数类型检查，我放在了checkIn函数里直接检查。

```
Symbol sb = new Symbol();
sb.params = new ArrayList<>();

//System.out.println("functioncall");
for(int i=0;i<funcCall.argList.size();i++)
{
    Symbol sb_type = new Symbol();
    //System.out.println(funcCall.argList.get(i).getType());
    visit(funcCall.argList.get(i));
    if(mapSb.get(funcCall.argList.get(i))!=null)
    {
        sb_type.name = mapSb.get(funcCall.argList.get(i)).name;
        sb_type.kind = mapSb.get(funcCall.argList.get(i)).kind;
        sb_type.type = mapSb.get(funcCall.argList.get(i)).type;
        checkIn(sb_type);
        sb_type.type = mapType.get(sb_type.name);
        sb.params.add(sb_type);
    }
    res = map.get(funcCall.argList.get(i));
    Quat quat = new Quat(op,res,opnd1,opnd2);
    quats.add(quat);

}
op = "call";

visit(funcCall.funcname);

sb.name = mapSb.get(funcCall.funcname).name;
sb.kind = funcCall.getType();
//sb.params.get(0).print();
checkIn(sb);

```

在函数调用的时候还需要考虑传入的参数有无定义，如果找不到该变量的定义，返回类型就会为null。之后根据自己传入的参数，判断参数个数和参数类型是否匹配。这里不仅考虑传入常量还要考虑传入变量，并且要获得变量的类型。

3.1.7 操作数是否相容

这里操作符考虑大多数二元表达式，但是由于表达式的操作数又可以是表达式，涉及到动态分析和静态分析，所以这只考虑二元表达式的左右操作数是变量或者常量的情况。因为`<< >> & | ^`这些符号不能的操作数不能是浮点类型的。依此为例，进行说明：

- 首先取出第一个操作数，判断属于什么类型，进行相应处理。
- 之后取出第二个操作数，同样的操作
- 对操作符判断，根据操作数是否为浮点数给出错误提示。

3.1.8 goto语句

因为goto语句的特殊性，我们需要在函数体最后进行判断。根据压入的符号表以及当前层次，遍历寻找goto语句的标签，如果属性值f为true，说明已经定义了。

```
    boolean checkAllLabel(BodySymbol bs){
        //bs是函数定义的那部分
        if(labelSymbols.size()!=0)
        {
            int j = 0;
            int id = bs.last;
            while(j<bs.vSize){
                Symbol sb = SymbolTables.get(id);
                //sb.print();
                if(sb.kind == "GotoStatement")
                {
                    LabelSymbol lsb =
labelSymbols.get(sb.lbId);
                    if(!lsb.f){
                        printError("Label: "+sb.name+" is not
defined.",7);
                        return false;
                    }
                    return true;
                }

                j++;
                id = sb.link;
            }
            return true;
        }

        return true;
    }
```

3.1.9 有无以return语句结束

这里直接每次遍历到return语句的时候给freturn++，在函数结束的时候判断freturn是否大于0，等于0说明没有return语句，给出相应错误提示信息。

3.2 中间代码生成

该实验的中间代码以四元式的形式，由于声明语句主要是对符号表进行操作，所以在中间代码生成的时候只需要考虑表达式、循环语句、选择语句等等。同时由于实验中用到了较多的中间变量，用ASTNode的子类TemporaryValue表示，并且在四元式输出的时候直接调用老师提供的类ExampleICPrinter即可，并对方法astStr进行补充。

形如四元式（op,res,opnd1,opnd2），op表示操作符，res存放结果（可以是变量名，也可以是临时变量），opnd1和opnd2是左右操作数。

3.2.1 函数定义

这里采用汇编代码的思想。函数体开始的时候用proc标志字，函数体结束的时候用endp结束字。

返回语句return采用ret，返回值跟在ret的后面。

如果有参数，还需要出栈，出栈的顺序和压栈的顺序相反。

```
(pro,,xx)
...
(ret,,xx)
(endp,,xx)
```

3.2.2 表达式语句

数组

采用的四元式操作符op为[]，res为临时变量，opnd1是数组名（这里如果是多维数组的话，还可能上一层数组的临时变量），opnd2是数组里面的元素值。

```
例如：
a[1] = 2;
        ->
            ([],a,1,%1)
            (=,2,,%1)
```

二元表达式

由于操作符有多种，这里分成三类，分别是二元运算的（+，-，*，/，%）、一元运算符（++，--，sizeof，~，!，-）和赋值运算符（=，*=，/=，%=，+=...）。根据三类运算符的不同，输出的四元式格式也不一样。

- 赋值操作符：访问expr1得到res，根据expr2表达式的不同，执行不同的代码：

（1）一元表达式：opnd1是null，op是对应的操作符，opnd2是访问一元表达式操作数的结果。

（2）二元表达式：opnd1是二元表达式的expr1访问结果，opnd2是二元表达式expr2的访问结果，op是二元操作符。

（3）后缀表达式：opnd2是null，op是对应的操作符，opnd1是访问一元表达式操作数的结果。

（4）函数调用表达式：访问函数调用表达式之后得到结果赋给opnd1

- 二元操作符：创建一个临时变量保存到res中，opnd1和opnd2是访问左右两个操作数的结果。
- 一元操作符：创建一个临时变量保存到res中，opnd1和opnd2是访问左右两个操作数的结果。（这里不考虑是前缀还是后缀，通过看opnd1和opnd2哪一个为null来判断的）

后缀表达式

创建一个临时变量作为res，访问它的操作数。形式为：

```
例如：
a++;
    ->
        (++,a,,%1)
```

一元表达式

创建一个临时变量作为res，访问它的操作数。形式为：

```
例如：
++b;
    ->
        (++,,b,%1)
```

函数调用

这里为了和汇编代码适配，采用压栈参数的形式，函数调用前先压栈，之后再调用，将结果保存到临时变量里面。

```
例如：
res = f(a,b)
        ->  (push,,,a)
            (push,,,b)
            (call,f,,%1)
            (=,%1,,res)
```

3.2.3 选择语句

这里采用条件跳转语句的方式进行。jf表示如果表达式为否，进行跳转；jmp表示无条件跳转。

```
...
```

```
if(con){
    stat;
}else{
    stat;
}
...
        ->
        (expression.ir)[结果存到了%1中]
        (jf,%1,,%2)
        (statement.ir)
        (jmp,,%3)
        (label,,%2)
        (statement.ir)
        (label,,%3)
```

3.2.4 循环语句

循环语句还是采用条件跳转的方式，具体格式如下：

```
for(init;cond;step){
    stat;
}
                    ->
                        (init.ir)
                        (label,,%1)
                        (cond.ir)[结果存到了%2中]
                        (jf,%2,,%3)
                        (statement.ir)
                        (step.ir)
                        (jmp,,%1)
                        (label,,%3)
```

由于存在多重循环的情况，所以在循环标签和错误跳转标签的时候要进行压栈出栈处理。

对于特殊情况，break语句要无条件跳转到错误跳转标签处。continue语句要在栈step_num中添加标签，等到生成step中间代码的时候，判断step_num是否为空，不为空说明有continue语句，根据step_num的最上层标签可以得到step的标签值。

3.2.5 goto语句

goto语句要通过查找有无该标签定义，如果没有的话，说明是使用性出现，那么需要创建一个局部变量来代替这个标签，并且存放到该标签对应的ir_id中，以便在label语句的时候直接使用。如果有的话，直接查看标签得到的ir_id，创建对应下标的临时变量。

3.2.6 label语句

与goto语句类似，首先检查有无出现，如果已经出现了，并且是使用性出现，则直接使用对应的ir_id来创建相应下标的临时变量，否则创建一个新的临时变量。

5. 实验结果

5.1 语义分析

5.1.1 变量无定义

```
1  //int f();
2  int main()
3  {
4      int res = f();
5      int c = a+1;
6      return 0;
7  }
8  int f(){
9      return 1;
10 }
```

上面代码所示，f函数定义以及变量也没有定义，给出错误提示。

```
errors:
-------------------------------------
ES01 >> FunctionCall f is not declarated.
ES01 >> Identifier a is not defined.
-------------------------------------
```

5.1.2 变量有无重复定义

```
1  int f();
2  //int f();
3  int f(){
4      return 1;
5  }
6
7  int f(){
8      return 2;
9  }
10
11 int main()
12 {
13     int a;
```

```
14      int a;
15
16      int b;
17      {
18          int b;
19      }
20      return 0;
21  }
```

由上面代码可知，函数定义f重复，变量a定义重复，由于b在不同的作用域里，所以不作为变量重复。

```
errors:
-------------------------------------
ES02 >> FunctionDefine f is defined.
ES02 >> VariableDeclarator a has been declarated.
-------------------------------------
```

5.1.3 break是否在循环体

```
1   int main()
2   {
3       break;
4
5       for(;;){
6           break;
7       }
8
9       break;
10
11      return 0;
12  }
```

由上面代码可知，有两个break不在循环体中。

```
errors:
-----------------------------------
ES03 >> BreakStatement:must be in a LoopStatement.
ES03 >> BreakStatement:must be in a LoopStatement.
-----------------------------------
```

5.1.4 函数调用参数是否匹配

```
int f(int a,int b){
    return a+b;
}
int main(){
    float a,b;
    int res = f(1);
    res = f(1,0.2);
    res = f(a,b);
    return 0;
}
```

由上面代码可知，第一个函数调用参数数量不匹配，第二个和第三个函数调用参数类型不匹配

```
errors:
-------------------------------------
ES04 >> FunctionCall:f's param num is not matched.
ES04 >> FunctionCall:f's param type is not matched.
ES04 >> FunctionCall:f's param type is not matched.
-------------------------------------
```

5.1.5 操作数是否相容

```
int main()
{
    int a = 2;
    double b = 3.5;
    int res = a<<b;
    return 0;
}
```

由上面代码可知，"<<"操作符不能是浮点数。

```
errors:
-------------------------------------
ES05 >> BinaryExpression:(<< >> & | ^)expression's should be int.
-------------------------------------
```

5.1.6 goto语句

```
int main()
{
k1:
    goto k2;
k2:
    goto k1;

    goto end;
    return 0;
}
```

由上面代码可知，end标签未定义

```
errors:
-------------------------------------
ES07 >> Label: end is not defined.
-------------------------------------
```

5.1.7 有无以return语句结束

```
int main()
{
    int a,b,c;
}

int f(){
    return 1;
}
```

由上面代码可知，main函数没有return语句。

```
errors:
-------------------------------------
ES08 >> Function: main must have a return in the end.
-------------------------------------
```

5.2 中间代码

5.2.1 函数调用

```
int foo(int a) {
    return 0;
}
```

对应输出：

```
(proc,,,foo)
(pop,,,a)
(ret,0,,)
(endp,,,foo)
```

5.2.2 表达式

```
int a, b, c;
    a = 0;
    b = 1;
    c = 2;
    c = a + b + (c + 3);
    int a1 = 1;
    int a2 = 2;
    int res;
    // unary oper
    res = !a1;
    res = ~a1;
    // binary oper
    res = a1+a2;
    res = a1%a2;
    res = a1 << a2;
    // selfchange
    res = a1++;
    res = ++a1;
```

对应输出：

```
(=,0,,a)
(=,1,,b)
(=,2,,c)
(+,a,b,%1)
(+,c,3,%2)
(+,%1,%2,c)
(=,1,,a1)
(=,2,,a2)
(!,,a1,res)
(~,,a1,res)
(+,a1,a2,res)
(%,a1,a2,res)
(<<,a1,a2,res)
(++,a1,,res)
(++,,a1,res)
```

5.2.3 选择语句

```
if(a1 && a2){
        res = f1(a1,a2);
    }else if(!a1){
        // b is global
        res = f1(b,a2);
    }else{
        f2();
    }
```

对应输出情况如下：

```
(&&,a1,a2,%1)
(jf,%1,,%2)
(push,,,a1)
(push,,,a2)
(call,f1,,%3)
(=,%3,,res)
(jmp,,,%4)
(label,,,%2)
(!,,a1,%5)
(jf,%5,,%6)
(push,,,b)
(push,,,a2)
(call,f1,,%7)
(=,%7,,res)
(jmp,,,%8)
(label,,,%6)
(call,f2,,%9)
(label,,,%8)
(label,,,%4)
```

5.2.4 循环语句

```
// "for" control-flow
    for(int i = 0; i<a1; i++){
        res += 2;
        break;
        res += 3;
        continue;
        res += 1;
    }
```

对应输出情况：

```
(label,,,%1)
(<,i,a1,%3)
(jf,%3,,%2)
(+=,2,,res)
(jmp,,,%2)
(+=,3,,res)
(jmp,,,%4)
(+=,1,,res)
(label,,,%4)
(++,i,,%5)
(jmp,,,%1)
(label,,,%2)
(ret,0,,)
(endp,,,main)
```

5.2.5 goto和label语句

```
k1:
    goto k2;
k2:
    goto k1;

```

对应输出情况：

```
(label,,,%1)
(jmp,,,%2)
(label,,,%2)
(jmp,,,%1)
```

5.2.6 综合情况

这里为了测试基本功能有无实现，测试了最终代码BubbleSort。

```
int main(){
    int a[10];
    Mars_PrintStr("please input ten int number for bubble sort:\n");
    for (int i = 0; i < 10; i++) {
```

```
        a[i] = Mars_GetInt();
    }
    Mars_PrintStr("before bubble sort:\n");
    for (int i = 0; i < 10; i++) {
        Mars_PrintInt(a[i]);
    }
    Mars_PrintStr("\n");
    // bubble sort
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10-i-1; j++) {
            if (a[j] > a[j + 1]) {
                int tmp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = tmp;
            }
        }
    }
    Mars_PrintStr("after bubble sort:\n");
    for (int i = 0; i < 10; i++) {
        Mars_PrintInt(a[i]);
    }

    return 0;
}
```

测试最终结果:

```
(proc,,,main)
(push,,,"please input ten int number for bubble sort:\n")
(call,Mars_PrintStr,,%1)
(=,0,,i)
(label,,,%2)
(<,i,10,%4)
(jf,%4,,%3)
([],a,i,%5)
(call,Mars_GetInt,,%6)
(=,%6,,%5)
(++,i,,%7)
(jmp,,,%2)
(label,,,%3)
(push,,,"before bubble sort:\n")
(call,Mars_PrintStr,,%8)
(=,0,,i)
(label,,,%9)
(<,i,10,%11)
(jf,%11,,%10)
([],a,i,%12)
(push,,,%12)
(call,Mars_PrintInt,,%13)
(++,i,,%14)
(jmp,,,%9)
```

```
(label,,,%10)
(push,,,"\n")
(call,Mars_PrintStr,,%15)
(=,0,,i)
(label,,,%16)
(<,i,10,%18)
(jf,%18,,%17)
(=,0,,j)
(label,,,%19)
(-,10,i,%23)
(-,%23,1,%22)
(<,j,%22,%21)
(jf,%21,,%20)
([],a,j,%25)
(+,j,1,%27)
([],a,%27,%26)
(>,%25,%26,%24)
(jf,%24,,%28)
([],a,j,%29)
(=,%29,,tmp)
([],a,j,%30)
(+,j,1,%32)
([],a,%32,%31)
(=,%31,,%30)
(+,j,1,%34)
([],a,%34,%33)
(=,tmp,,%33)
(label,,,%28)
(++,j,,%35)
(jmp,,,%19)
(label,,,%20)
(++,i,,%36)
(jmp,,,%16)
(label,,,%17)
(push,,,"after bubble sort:\n")
(call,Mars_PrintStr,,%37)
(=,0,,i)
(label,,,%38)
(<,i,10,%40)
(jf,%40,,%39)
([],a,i,%41)
(push,,,%41)
(call,Mars_PrintInt,,%42)
(++,i,,%43)
(jmp,,,%38)
(label,,,%39)
(ret,0,,)
(endp,,,main)
```

经过分析，尤其双重循环的过程代码，完全符合预期情况！

6. 实验感想

编译原理的实验已经快接近尾声了，在语义分析和中间代码生成的实验过程中，采集符号表进行语义检查，到四元式的生成，这个过程主要是逻辑分析，尤其是在循环和选择语句过程中生成中间代码的跳转标签采用栈的数据结构；以及语义检查过程中关于作用域不同所引起的问题更要细心。虽然这次实验难度和代码里都不小，但是只要抓住了关键问题，然后采取适当的算法，比如break语句是否在循环体内，考虑到多重循环的问题，我采用了括号匹配类似的思想。然后在遇到bug的时候耐心的断电调试，问题都可以一步一步解决。

总之，这次试验我掌握了语义分析和中间代码生成的方法，对C语言的一些语法有了更多的认识，对代码运行背后的机理有了更为深刻的了解。

只还差生成目标代码这最后一个步骤，就可以实现自己的编译器。虽然不够全面，但是这样一个简易的编译器会让我对编程语言有一个完整全面的认识，代码能力也大幅度提高，克服困难的毅力也会逐渐增强。我期待最后一步的实现，并且一定会更加努力坚持完成最后一个步骤。

提示：代码里面的src中semantic文件里的txjVisitor是我实现语义检查的一份代码，由于考虑到后面的中间代码生成需要符号表的一些信息（比如goto的标签），并且考虑到之前写的逻辑感不强，有些混乱，因此我重新写了一份代码包含了语义检查和中间代码生成，一次性遍历树完成。存放在ExampleICBuilder里面。