

# 编译原理与设计实验报告

姓名/学号: 鲍吴迪/1120173588

班级: 07111704

## 实验名称

中间代码生成实验

## 实验目的

- (1) 了解编译器中间代码表示形式和方法;
- (2) 掌握中间代码生成的相关技术和方法, 设计并实现针对某种中间代码 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, 中间代码生成模块与 其他模块之间的交互过程。

## 实验内容

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。

## 实验环境

Macbook pro  
2.6Hz 双核 Intel Core i5 内存 8GB  
Mac OS Catalina 10.15.1 (clang/llvm 编译器)  
IntelliJ IDEA 2018.3.1  
JDK 1.8.0

# 实验设计与实现

本次实验选取的语句集有函数定义、函数调用、选择语句、循环语句、返回语句，数组定义与调用，二元表达式，一元表达式，布尔表达式，后缀表达式。下文将依次说明选取语句集用四元式表达的具体实现。

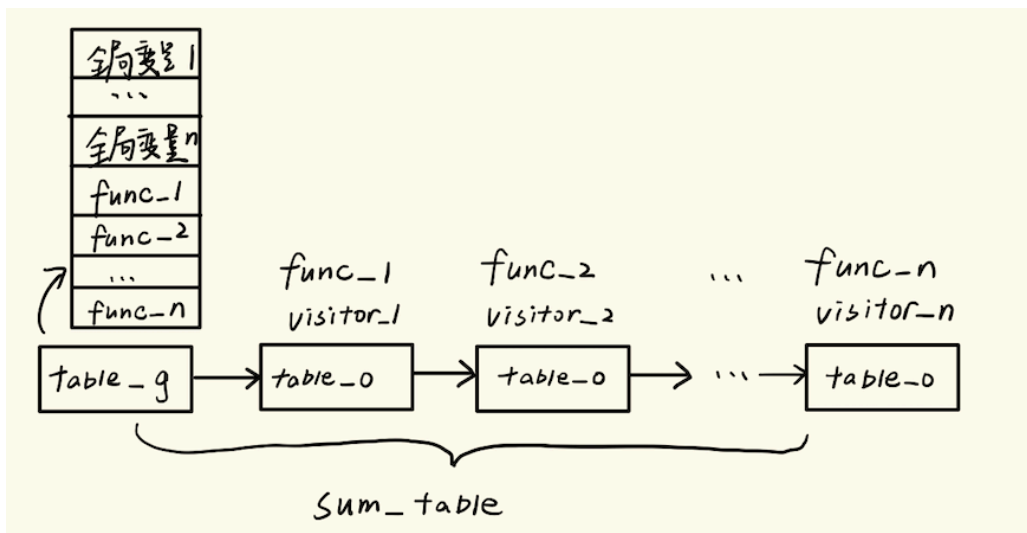
## 1. 符号表的建立

采用列表实现符号表的组织与建立，首先创建 TableItem 类，作为符号表中的项：

```
public class TableItem {
    public String Name;
    public String Type;
    public String Kind;
    public int Scope_entry;
    public int Scope_exit;
    public ArrayList<String> params=new ArrayList<>();
    public ArrayList<TableItem> sub_table;
}
```

- name 为变量名/函数名等，
- type 为数据类型（如 int），
- kind 为声明的种类（如 VariableDeclarator/FunctionDefine），
- scope\_entry/scope\_exit 为其作用域开始/结束位置，
- params 仅当该项为函数声明时有作用，列表里包括该函数的每个变量类型，如 int f (int a, int b)；则 params 为[int, int]，
- sub\_table 标识函数内部是否有封闭的子作用域，若存在，则为子作用域建立新的 table，并加入父作用域的 sub\_table 属性中。

符号表为 ArrayList<TableItem>, 每个 visitor 有一个独自の符号表 table\_0, 用 sum\_table（类型为 ArrayList< ArrayList<TableItem>>）组织所有的符号表，其中 sum\_table[0] 为全局符号表，sum\_table[1...n] 为每个函数的符号表。



## 2. 基本表达式和语句的四元式表示

一元表达式：

1. !res;
2. ~res;

四元式表达为( !, res, \_, %1), ( ~, res, \_, %1)其中%1 为自动生成的寄存器号。

```
public void visit(ASTUnaryExpression unaryExpression) throws Exception {
    String op = unaryExpression.op.value;
    ASTNode res = new TemporaryValue(++tmpId);
    ASTNode opnd1 = null;
    if (unaryExpression.expr instanceof ASTIdentifier) {
        ASTIdentifier id = (ASTIdentifier) unaryExpression.expr;
        opnd1 = id;
    }
    ASTNode opnd2 = null;
```

二元表达式：

1. a+b;
2. b=res;

四元式表达为 (+, a, b, %1), (=, res, \_, b)

判断二元表达式符号，确定是赋值表达式 (op.equals("=")、算数表达式 (op.equals("+") || op.equals("-"))、布尔表达式 (op.equals("<") || op.equals(">") || op.equals(">=") || op.equals("<=") || op.equals("=="))

```

ASTBinaryExpression value = (ASTBinaryExpression) binaryExpression.expr2;
op = value.op.value;
visit(value.expr1);
opnd1 = map.get(value.expr1);
visit(value.expr2);
opnd2 = map.get(value.expr2);

```

后缀表达式：

```

1. i++;
2.

```

同一元表达式，四元式为 (++, i, \_, %1)

返回语句：

```

3. return;
4. return a+b;

```

四元式为 (return, \_, \_, \_) 和 (+, a, b, %1) (return, %1, \_, \_) 判断 return 语句中的表达式类型（二元表达式、标识符、常量、无表达式），再进行相应实现。如二元表达式：

```

if (returnStat.expr.get(i) instanceof ASTBinaryExpression) {
    ASTBinaryExpression be = (ASTBinaryExpression) returnStat.expr.get(i);
    visit(be);
    ASTNode res = null;
    ASTNode opnd1 = quats.get(quats.size() - 1).getRes();
    ASTNode opnd2 = null;
    Quat quat = new Quat(cursor++, op: "return", res, opnd1, opnd2);
    quats.add(quat);
}

```

### 3. 选择语句的四元式实现

根据教材上给出的选择语句四元式格式：

```
if (a-b)<5 then x=a * b
```

按上述语义处理子程序将可翻译成如下四元式序列：

```

10 (-, a, b, T1)
20 (<, T1, 5, T2)
30 (jT, _, _, 50)
40 (j, _, _, 70)
50 (*, a, b, T3)
60 (=, T3, _, x)
70 ...

```

首先判断 if 语句的条件类型（标识符/布尔表达式），并为其添加相应的四元式。

```

for(int i=0;i<selectionStat.cond.size();i++){
    if(selectionStat.cond.get(i).getType().equals("Identifier")){//条件是一个字符
        visit((ASTIdentifier)selectionStat.cond.get(i));
    }else if(selectionStat.cond.get(i) instanceof ASTBinaryExpression){
        visit((ASTBinaryExpression)selectionStat.cond.get(i));
    }
}

```

再根据逻辑添加转移语句

```

res = new CursorValue( id: cursor+2);
Quat quat1 = new Quat(cursor++, op: "Jt", res, opnd1, opnd2);
quats.add(quat1);

res = new CursorValue( id: cursor+2);
Quat quat2 = new Quat(cursor++, op: "J", res, opnd1, opnd2);
quats.add(quat2);

```

最后分别为 then 和 else 的执行语句添加四元式。

对于语句

```

1. if(a<b){
2.     a=1;
3. }else{
4.     b=1;
5. }

```

输出如下：

```

7:(<,a,b,%3)
8:(Jt,,,10)
9:(J,,,11)
10:(=,1,,a)
11:(=,1,,b)

```

## 4. 循环语句的四元式实现

根据教材上循环语句的逻辑结构：

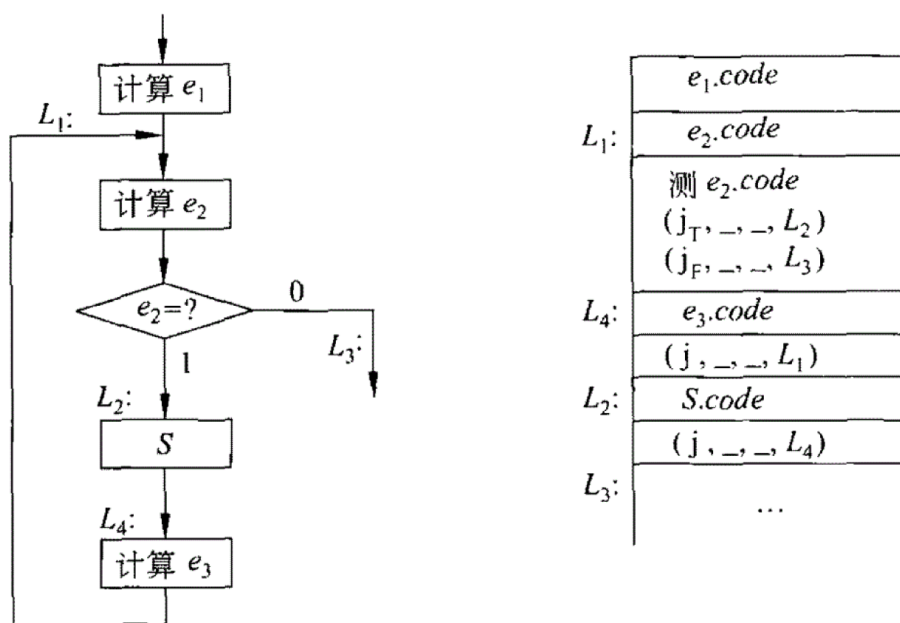


图 6-14 循环语句的目标代码结构

根据流程图先添加  $\text{init}(e_1)$  与  $\text{cond}(e_2)$  的四元式，为了便于后续重复计算  $e_2$  及以后的步骤，在访问  $e_1$  后添加一个  $L_1$  记录当前  $\text{cursor}$  的值（ $\text{cursor}$  为当前目标代码行序号），后续可以直接通过访问  $L_1$  跳转到相应的行。

```
for(int i=0;i<iterationStat.init.size();i++){
    visit(iterationStat.init.get(i));
}
Integer L1 = cursor;
for(int i=0;i<iterationStat.cond.size();i++){
    visit(iterationStat.cond.get(i));
}
```

测表达式  $e_2$  的值，为 1 则  $j_t$  跳转到  $L_1$ ，为 0 则  $j_f$  跳转到  $L_3$ ，计算循环体  $S$  和  $e_3$  的语句个数，再跳转到相应的行

```
res = new CursorValue( id: cursor+3+iterationStat.step.size());
Quat quat1 = new Quat(cursor++, op: "Jt", res, opnd1, opnd2);
quats.add(quat1);

if(iterationStat.stat instanceof ASTCompoundStatement){//循环主体部分不止一个语句
    ASTCompoundStatement cs = (ASTCompoundStatement)iterationStat.stat;
    res = new CursorValue( id: cursor+3+iterationStat.step.size()+cs.blockItems.size());
}else{
    res = new CursorValue( id: cursor+4+iterationStat.step.size());
}

Quat quat2 = new Quat(cursor++, op: "Jf", res, opnd1, opnd2);
quats.add(quat2);
```

最后访问  $e_3$  和  $S$  中的语句，并添加相应跳转，由于  $S$  需要跳转到  $L_4$ ，所以在计算  $e_3$  前添加  $L_4$  行数的临时变量，便于后续跳转。

```

Integer L4 = cursor;
for(int i=0;i<iterationStat.step.size();i++){
    visit(iterationStat.step.get(i));
}
res = new CursorValue(L1);
Quat quat3 = new Quat(cursor++, op: "J", res, opnd1, opnd2);
quats.add(quat3);

visit(iterationStat.stat);
res = new CursorValue(L4);
Quat quat4 = new Quat(cursor++, op: "J", res, opnd1, opnd2);
quats.add(quat4);

```

测试语句如下：

```

1. for(i=0;i<b;i++){
2.     a=a+i;
3. }

```

输出结果：

```

12  12: (=, 0, , i)
13  13: (<, i, b, %4)
14  14: (Jt, , , 18)
15  15: (Jf, , , 20)
16  16: (++ , i, , i)
17  17: (J, , , 13)
18  18: (+, a, i, a)
19  19: (J, , , 16)
20

```

## 5. 数组元素的定义与表示

为数组元素建立内情向量表，为 List<DopeVector>形式，每个 DopeVector 是一个内情向量，用于表示一个数组。内情向量类 DopeVector 结构如下：

```

public class DopeVector {
    public String name;           //数组名称
    public String type;           //数组类型
    public Integer dim;           //数组维数
    public ArrayList<Up_Low> bound = new ArrayList<>(); //数组每一维的上下限组成的链表
    public Integer address;       //数组首地址
}

```

其中 bound 表示每一维的上下限(用 up\_low 类表示)，up\_low 类结构如下：

```

public class Up_Low { //表示数组每一维的上限和下限
    public Integer upper;
    public Integer lower;
}

```

如 int array[2][3]，其属性为：

```

name=array
type=int
dim=2

```

bound=<0, 1><0, 2>

数组首地址暂时未使用。

在 ExampleICBuilder 中维护一个数组内情向量表 ArrayList<DopeVector> table\_vec, 声明数组时则将相应的信息组成内情向量, 添加到 table\_vec 中。

由于 json 中对多维数组的表示是嵌套的, 如 array[10][20] 的 json 表示:

```
1. "initLists": [{
2.     "type": "InitList",
3.     "declarator": {
4.         "type": "ArrayDeclarator",
5.         "declarator": {
6.             "type": "ArrayDeclarator",
7.             "declarator": {
8.                 "type": "VariableDeclarator",
9.                 "identifier": {
10.                    "type": "Identifier",
11.                    "value": "array",
12.                    "tokenId": 28
13.                }
14.            },
15.            "expr": {
16.                "type": "IntegerConstant",
17.                "value": 10,
18.                "tokenId": 30
19.            }
20.        },
21.        "expr": {
22.            "type": "IntegerConstant",
23.            "value": 20,
24.            "tokenId": 33
25.        }
26.    },
27. }
```

因此, 在逐层访问 declarator 时, 并不知道数组的维数, 为了准确表示数组维数, 首先建立内情向量 vector, 令 vector.dim=0, 在访问内层嵌套的 declarator 时判断类型。

如果是 ASTVariableDeclarator, 则已经访问到最后一层, 则为建立的内情向量 vector 添加 name 属性, 并令 vector.dim+1, 然后加入 table\_vec 中

如果是 ASTArrayDeclarator, 则说明内层嵌套数组, 将 vector.dim+1, 继续访问下一层 ArrayDeclarator。



```
vector.dim = 0;

if(arrayDeclarator.declarator instanceof ASTVariableDeclarator){
    this.table_vec.add(vector);
    ASTVariableDeclarator vd = (ASTVariableDeclarator) arrayDeclarator.declarator;
    this.table_vec.get(table_vec.size()-1).name = vd.identifier.value;
    this.table_vec.get(table_vec.size()-1).dim += 1;
    visit((ASTVariableDeclarator)arrayDeclarator.declarator);
}else if(arrayDeclarator.declarator instanceof ASTArrayDeclarator){
    visit((ASTArrayDeclarator)arrayDeclarator.declarator);
    this.table_vec.get(table_vec.size()-1).dim+=1;
}
```

然后访问 arrayDeclarator.expr 属性，为内情向量添加每一维的上下限。

```
if(arrayDeclarator.expr instanceof ASTIntegerConstant){
    ASTIntegerConstant ic = (ASTIntegerConstant) arrayDeclarator.expr;
    Up_Low ul = new Up_Low();
    ul.lower = 0;
    ul.upper = ic.value-1;
    table_vec.get(table_vec.size()-1).bound.add(ul);
}
```

取测试代码如下：

```
1. int arr[5];
2. int array[10][20];
```

建立好的 table\_vec 如下图所示：

```
▼ f table_vec = {ArrayList@3067} size = 2
  ▼ 0 = {DopeVector@3069}
    ▶ f name = "arr"
    ▶ f type = "int"
    ▶ f dim = {Integer@3073} 1
    ▶ f bound = {ArrayList@3074} size = 1
      f address = null
  ▼ 1 = {DopeVector@3070}
    ▶ f name = "array"
    ▶ f type = "int"
    ▶ f dim = {Integer@3077} 2
    ▶ f bound = {ArrayList@3078} size = 2
      f address = null
```

## 6. 函数的定义与调用

函数定义时，将函数及其参数信息加入符号表，实现方法为新建一个 ExampleICBuilder 对象维护一个新的符号表，在返回时将新对象的符号表与当前对象符号表整合，具体实现与上一节语义分析实验相同，此处不再赘述。

在输出的 .ic 文件中标明函数参数及作用域信息：

1. 在 quat 类中添加一个 scope 属性用于输出函数信息。

```
private String scope=null;//指明函数体
```

并为其添加构造函数

```
public Quat(Integer cursor,String scope){  
    this.cursor = cursor;  
    this.scope = scope;  
}
```

与一般四元式相区别，输出时判断其 scope 属性是否为 null，是则输出普通四元式，若 scope 属性不为 null，则输出函数作用域信息

```
for (Quat quat : quats) {  
    if(quat.getScope()!=null){  
        Integer cursor = quat.getCursor();  
        String scope = quat.getScope();  
        sb.append(cursor.toString()+":"+scope+"\n");  
    }  
}
```

2. 在函数定义时新建一个 quat 对象，为其添加 scope 属性  
包括函数说明 func&、函数名称、函数参数类型、函数参数名称等信息。

```
String scope_Str = "func &"+this.table_0.get(0).Name+"(";  
//访问函数参数信息，并添加到要输出的字符串中  
for(int i=0;i<this.table_0.get(0).params.size();i++){  
    scope_Str+=this.table_0.get(0).params.get(i);  
    int temp_para = i+1;//在最后一个表中找参数名称  
    scope_Str+=" ";  
    scope_Str+=table_0.get(temp_para).Name;  
  
    if(i!=this.table_0.get(0).params.size()-1){  
        scope_Str+=",";  
    }  
}  
scope_Str+=")";  
Quat quat_scope = new Quat(cursor++,scope_Str);  
quats.add(quat_scope);
```

对于测试代码：

```
1. int add(int a,int b){  
2.     return a+b;  
3. }  
4. int main(){}
```

则输出的四元式如下所示：

```
1:func &add(int a,int b):  
2:    (+,a,b,%1)  
3:    (return,%1,,)  
4:func &main():
```

包含了函数的名称和参数信息

3. 函数调用的四元式表示：

对于函数调用，则直接从符号表中找出相应函数的入口地址（符号表项的 scope\_entry 属性指明了该函数的入口）

```
ASTIdentifier funid = new ASTIdentifier();
if(funcCall.funcname instanceof ASTIdentifier){
    funid = (ASTIdentifier)funcCall.funcname;
}
for(int i=0;i<table_g.get(0).size();i++){
    if(table_g.get(0).get(i).Name.equals(funid.value)){
        res = new CursorValue(table_g.get(0).get(i).Scope_entry);
        break;
    }
}
Quat quat1 = new Quat(cursor++, op: "J", res, opnd1, opnd2);
```

对于测试代码：

```
1. int add(int a,int b){
2.     return a+b;
3. }
4. int main() {
5.     int a, b, c;
6.     a = 0;
7.     b = 1;
8.     c = 2;
9.     add(b,c);
10. }
```

输出的四元式结果如下：

```
1:func &add(int a,int b):
2:    (+,a,b,%1)
3:    (return,%1,,)
4:func &main():
5:    (=,0,,a)
6:    (=,1,,b)
7:    (=,2,,c)
8:    (J,,1)
```

可以看出，在调用函数 add (b, c) 时，直接 jump 到了 add 函数入口地址 1 处（此时调用时的参数假设入栈，执行函数时出栈）

## 7. 整体测试与总结

测试完整代码 example\_test.c 为：

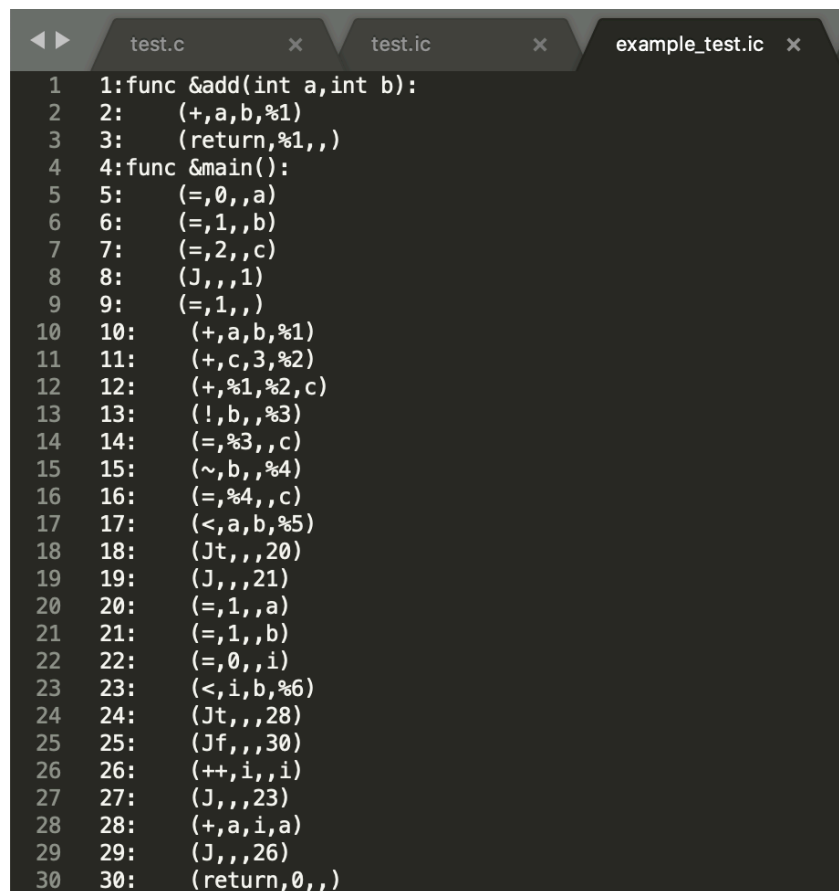
```
1. int add(int a,int b){
2.     return a+b;
3. }
4. int main() {
5.     int a, b, c;
```

```

6.    a = 0;
7.    b = 1;
8.    c = 2;
9.    add(b,c);
10.   int arr[5];
11.   int array[10][20];
12.   array[0][1]=1;
13.   c = a + b + (c + 3);
14.   c = !b;
15.   c = ~b;
16.   if(a<b){
17.       a=1;
18.   }else{
19.       b=1;
20.   }
21.   for(i=0;i<b;i++){
22.       a=a+i;
23.   }
24.   return 0;
25. }

```

输出结果为:



```

1  1:func &add(int a,int b):
2  2:    (+,a,b,%1)
3  3:    (return,%1,,)
4  4:func &main():
5  5:    (=,0,,a)
6  6:    (=,1,,b)
7  7:    (=,2,,c)
8  8:    (J,,1)
9  9:    (=,1,,)
10 10:    (+,a,b,%1)
11 11:    (+,c,3,%2)
12 12:    (+,%1,%2,c)
13 13:    (!,b,,%3)
14 14:    (=,%3,,c)
15 15:    (~,b,,%4)
16 16:    (=,%4,,c)
17 17:    (<,a,b,%5)
18 18:    (Jt,,20)
19 19:    (J,,21)
20 20:    (=,1,,a)
21 21:    (=,1,,b)
22 22:    (=,0,,i)
23 23:    (<,i,b,%6)
24 24:    (Jt,,28)
25 25:    (Jf,,30)
26 26:    (++,i,,i)
27 27:    (J,,23)
28 28:    (+,a,i,a)
29 29:    (J,,26)
30 30:    (return,0,,)

```

整体支持的语句有函数定义、函数调用、选择语句、循环语句、返回语句，数组定义与调用，二元表达式，一元表达式，布尔表达式，后缀表达式。

## 实验心得体会

总体来说，用四元式实现中间代码难度并不高，对于循环选择等语句的四元式表达，教材上都有明确的定义。

主要问题在于如何用四元式表达函数的信息等都要自己定义，与上次的语义分析实验相比，添加了数组定义的实现，新建立了内情向量表用于表示数组。

有一个卡了比较久的小 bug 是由于代码中大多是表都用 `arraylist` 实现，但定义 `arraylist` 的时候没有初始化或初始化为 `null`，导致代码运行时报了很多空指针错误。

经过本次实验，对中间代码生成的具体步骤有了实践，加深了对四元式的理解，学习了多维数组在中间代码中的具体表示，收获颇丰。