北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY

本科生《编译原理》课程实践报告

题　　目： 语义分析实验

学　　院： 徐特立学院

专业名称： 计算机科学与技术

姓　　名： 陈照欣-1120191086

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。

实验步骤

中间代码使用四元式表示，使用了 lab6 中实现的 SymboTable

1. 数组定义

op	opnd1	opnd2	res
arr	数组类型+维度	范围	数组名

在符号表中取出数组的维数列表 limit，并获取数组访问索引表达式列表 index，需要进行计算以获取真正的数组索引。

```
if(node.declarator instanceof ASTArrayDeclarator) {
  LinkedList limit = new LinkedList();
  ASTDeclarator declarator =
((ASTArrayDeclarator)node.declarator).declarator;
  ASTExpression expr = ((ASTArrayDeclarator)node.declarator).expr;
  while(true) {
    int limit0 =  ((ASTIntegerConstant)expr).value;
    limit.addFirst(limit0);
    if(declarator instanceof ASTArrayDeclarator) {
      expr = ((ASTArrayDeclarator)declarator).expr;
      declarator = ((ASTArrayDeclarator)declarator).declarator;
    }else {
      break;
    }
  }
    if(declaration.scope == globaltable) {
    globaltable.addvar(name, specifier, limit);
    String sl = declaration.specifiers.get(0).value;
    for(int k=0 ; k<limit.size() ; k++) {
      int limit0 = (int)limit.get(k);
      sl += "<" + limit0 + ">";
    }
    Quat quat0 = new Quat("arr",declarator,new
```

```
ControlLabel(sl),localscope);
      quats.add(quat0);
    }
    else {
      localtable.addvar(name, specifier, limit);
      String sl = declaration.specifiers.get(0).value;
      for(int k=0 ; k<limit.size() ; k++) {
        int limit0 = (int)limit.get(k);
        sl += "<" + limit0 + ">";
      }
      Quat quat0 = new Quat("arr",declarator,new
ControlLabel(sl),localscope);
      quats.add(quat0);
    }
}
```

2. 变量声明

op	opnd1	opnd2	res
var	变量类型	范围	变量名

若变量声明为一个二元表达式，则将中间表达式拆分成两部分。第一部分表示变量的声明，格式如上所示；第二部分表示变量值的初始化部分。

1. 若等式右边为二元表达式，则完整的四元式为

op	opnd1	opnd2	res
var	变量类型	范围	变量名 1
+;-;*;/;%	变量名 2	变量名 3	变量名 1

2. 若等式右边为函数调用，则先声明传递的参数，后使用 call 声明调用函数

op	opnd1	opnd2	res
var	变量类型	范围	变量名 1
arg			变量名 2
call	函数名		临时变量名
=	返回值类型	范围	变量名 1

3. 若等式右边为前缀表达式，则先对前缀表达式进行运算，再将值赋给变量

op	opnd1	opnd2	res
var	变量类型	范围	变量名 1
++;--		变量名 2	变量名 2
=	变量名 2		变量名 1

4. 若等式右边为后缀表达式，则先用临时变量保存表达式中变量的值，对后缀表达式进行运算后将临时变量值赋给等号左侧变量

op	opnd1	opnd2	res
var	变量类型	范围	变量名 1
=	变量名 2		临时变量名
++;--	变量名 2		变量名
=	临时变量名		变量名 1

3. 二元表达式

与变量声明类似，省略

```
String op = "=";
ASTNode res = node.declarator;
ASTNode opnd1 = null;
ASTNode opnd2 = localscope;
ASTExpression expr = node.exprs.get(0);
if (expr instanceof ASTIdentifier) {
   opnd1 = expr;
}else if(expr instanceof ASTIntegerConstant) {
   opnd1 = expr;
}else if(expr instanceof ASTFloatConstant) {
   opnd1 = expr;
}else if(expr instanceof ASTCharConstant) {
   opnd1 = expr;
}else if(expr instanceof ASTStringConstant) {
   opnd1 = expr;
}else if(expr instanceof ASTBinaryExpression) {
   ASTBinaryExpression value = (ASTBinaryExpression)expr;
   op = value.op.value;
   visit(value.expr1);
   opnd1 = map.get(value.expr1);
   visit(value.expr2);
   opnd2 = map.get(value.expr2);
}else if(expr instanceof ASTUnaryExpression) {
   visit(expr);
   opnd1 = map.get(expr);
}else if(expr instanceof ASTPostfixExpression) {
   visit(expr);
   opnd1 = map.get(expr);
}else if(expr instanceof ASTFunctionCall) {
   visit(expr);
   opnd1 = map.get(expr);
}
Quat quat = new Quat(op, res, opnd1, opnd2);
quats.add(quat);
map.put(node, res);
```

4. 选择表达式

选择结构分为三个部分，分别是 Condition、Then、Else，并且由于可能出现的 `else if`，需要两个标签`@IfFalse`、`@IfEnd`

op	opnd1	opnd2	res
jf	临时变量名		`@IfFalseL0`
j			`@IfEndL1`
label			`@IfFalseL0`
label			`@IfEndL1`

5. 循环表达式

循环结构分为 4 个部分 Init、Check、Body、Step，其中 Check 和 Body 中的 break 和 continue 语句导致控制流的转移，故设置三个标签分别记录为`@IterationCheck`，`@IterationEnd`，`@IterationNext`。标签可以使用 ContrlLabel 类临时生成，故最终一个循环的四元式序列可以参考如下。

op	opnd1	opnd2	res
label			`@IterationCheckL0`
jf	临时变量名		`@IterationEndL1`
j			`@IterationEndL1`
j			`@IterationNextL2`
label			`@IterationNextL2`
j			`@IterationCheckL0`
label			`@IterationEndL1`

实验结果

1. 外部定义

输入:

```
int a[10][20];

int b = 1;
```

结果:

(arr,int<10><20>,global,a)

(var,int,global,b)

(=,1,global,b)

2. 函数定义与函数调用

输入;

```
int f1(int x,int y){

    int z = x + y;

    return z;

}

void f2(){

    Mars_PrintStr("in f2\n");

    return;
```

```
}
```

结果：

(Func_Type,int,,f1)
(param,int,f1,x)
(param,int,f1,y)
(var,int,f1,z)
(+,x,y,z)
(ret,,,z)
(Func_End,,,f1)
(Func_Type,void,,f2)
(arg,,,"in f2\n")
(call,Mars_PrintStr,,)
(ret,,,)
(Func_End,,,f2)

3. 单元式与二元式

输入：

```
res = !a1;

res = ~a1;

res = a1+a2;

res = a1%a2;

res = a1 << a2;

res = a1++;

res = ++a1;
```

结果：

(var,int,main,a1)
(=,1,main,a1)
(var,int,main,a2)
(=,2,main,a2)
(var,int,main,res)
(!,a1,,%1)
(=,%1,,res)
(~,a1,,%2)
(=,%2,,res)
(+,a1,a2,res)
(%,a1,a2,res)
(<<,a1,a2,res)

(=,a1,,%3)
(++,a1,,a1)
(=,%3,,res)
(++,,a1,a1)
(=,a1,,res)

4. 选择表达式

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

(&&,a1,a2,%4)
(jf,%4,,@IfFalseL0)
(Scope_Beg,main,,SubScope@0)
(var,int,SubScope@0,%5)
(arg,,,a1)
(arg,,,a2)
(call,f1,,%5)
(=,int,f1,res)
(j,,,@IfEndL1)
(Scope_End,main,,SubScope@0)
(label,,,@IfFalseL0)
(!,a1,,%6)
(jf,%6,,@IfFalseL2)
(Scope_Beg,main,,SubScope@1)
(var,int,SubScope@1,%7)
(arg,,,b)
(arg,,,a2)
(call,f1,,%7)
(=,int,f1,res)
(j,,,@IfEndL3)
(Scope_End,main,,SubScope@1)
(label,,,@IfFalseL2)

(Scope_Beg,main,,SubScope@2)
(call,f2,,)
(Scope_End,main,,SubScope@2)
(label,,,@IfEndL3)
(label,,,@IfEndL1)

5. 循环语句

```
for(int i = 0; i<a1; i++){
        break;
        continue;
        res += 1;
    }
```

(Scope_Beg,main,,SubScope@3)
(var,int,SubScope@3,i)
(var,int,SubScope@3,%8)
(var,int,SubScope@3,%9)
(var,int,SubScope@3,i)
(=,0,SubScope@3,i)
(label,,,@IterationCheckL4)
(<,i,a1,%8)
(jf,%8,,@IterationEndL5)
(j,,,@IterationEndL5)
(j,,,@IterationNextL6)
(+,res,1,res)
(label,,,@IterationNextL6)
(=,i,,%9)
(++,i,,i)
(j,,,@IterationCheckL4)
(label,,,@IterationEndL5)
(Scope_End,main,,SubScope@3)

实验总结

在词法分析，语法分析，语义分析一系列实验的基础之上，将 C 源代码翻译成中间代码，在这个实验的过程中也认识到了中间代码的表示形式和生成中间代码的原理和技巧，掌握了对简单赋值语句，循环语句，选择语句等语句的翻译过程，对编译器的编译原理有了更加深刻的理解和认识，同时在过程中也锻炼了代码能力。

中间代码生成是整个编译器前端最后一个部分。一种中间语言可以根据需求在后续的实验中翻译成不同型号的目标机的目标代码。可以在中间代码上进行与机器无关的优化，能够提高目标代码的质量。