# 编译原理 Lab7: 中间代码生成实验

郑子帆 1120200822

北京理工大学 计算机学院 07112002 班

日期：2023 年 5 月 30 日

**摘　　要**

本文为北京理工大学《编译原理与设计 2023》课程的 Lab7 实验报告。在本次实验我们设计了四元式，并对代码进行了四元式的生成。

## 1　实验简介 [2]

### 1.1　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。

### 1.2　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。例如下面的输入语句：

```
int main() {
    int a, b, c;
    a = 0;
    b = 1;
    c = 2;
    c = a + b + (c + 3);
    return 0;
}
```

对应的四元式输出是：

```
(=,0,,a)
(=,1,,b)
(=,2,,c)
(+,a,b,%1)
(+,c,3,%2)
(+,%1,%2,c)
```

对应的 MAPPLE IR 的输出为：

```
func &main() i32{
var %a i32
var %b i32
var %c i32
dassign %4(constval i32 0)
dassign %a(regread i32 %4)
dassign %5(constval i32 1)
dassign %b(regread i32 %5)
dassign %6(constval i32 2)
dassign %c(regread i32 %6)
dassign %7(
add i32(dread i32 %a,dread i32 %b))
dassign %8(regread i32 %7)
dassign %9(constval i32 3)
dassign %10(
add i32(dread i32 %c,regread i32 %9))
dassign %11(regread i32 %10)
dassign %12(
add i32(regread i32 %8,regread i32 %11))
dassign %13(regread i32 %12)
dassign %c(regread i32 %13)
dassign %14(constval i32 0)
return (regread i32 %14)}
```

其中% 开始的表示临时变量或者伪寄存器。

# 2 实验过程

虽然 lab8 中需要之前构建好的符号表，但是对于此实验我们并不需要使用符号表，故此实验中没有沿用 lab6 中的符号表，并计划在 lab8 中再按具体需求对此实验的代码进行改动和重构。

## 2.1 四元式设计

在本实验中，对于中间代码的输出，我们没有采用 LLVM IR 和 MAPLE IR，而是选择了 Example 中已经给好的四元式。由 *Quat.java* 文件内容可以看到，四元式由 <op, opnd1, opnd2, res> 构成，例如，对于 a = b + c，生成的四元式应该为 <+, b, c, a>。注意，在构建 Quat 对象时的顺序为 <op, res, opnd1, opnd2>，这与输出的顺序不同。

首先，我们需要基于 AST 抽象语法树 [1] 对每种 AST 节点进行讨论和设计。我们先考虑有关定义、声明的语句。注：下面设计的四元式按照 (op, opnd1, opnd2, res) 的顺序。

1. ASTVariableDeclarator，用于声明变量
   - 如果没有初始赋值。四元式 (Var, 变量类型 type, , 变量名 varName)
   - 如果有初始赋值，则在上面的基础上，还需要增加 (=, 初始值 initValue, , varName)
2. ASTFunctionDeclarator，函数声明，四元式 (Func, 返回类型 returnType, 参数个数 paramsNum, 函数名 funcName)
3. ASTFunctionDefine，函数定义
   - 在 body statement 之前定义四元式 (Proc, 返回类型 returnType, 参数个数 paramsNum, 函数名 funcName)
   - 在 body statement 之后定义四元式 (Endp, 返回类型 returnType, 参数个数 paramsNum, 函数名 funcName)
4. ASTArrayDeclarator，用于声明数组，设计四元式 (Arr, 数组类型 type, , 数组名 arrName)
5. ASTParamsDeclarator，参数声明，设计四元式 (Param, 参数类型 paramType, , 参数名 paramName)

然后我们再设计有关于运算符的四元式：

1. ASTUnaryExpression，单目运算
   - 对于可直接赋值的运算符，如 ++a，–a 等，设计四元式 (<unaryOP1>, a, , a)
   - 对于不可直接赋值的运算符，如 *a，&a 等，建立临时中间变量%id，设计四元式 (<unaryOP2>, a, , %id)

2. ASTBinaryExpression，二元运算符
   - 对于赋值运算"="，如 a=b，设计四元式 (=, b, , a)
   - 对于其他二元运算符，例 a op b，建立一个临时变量%id，设计四元式 (<binaryOp>, a, b, %id)
3. ASTPostfixExpression，后缀表达式。以 a++ 为例，分为两步：
   （a）第一步，先赋值。先将 a 赋值给临时变量%id，设计四元式 (=, a, , %id)
   （b）第二步，对 a 进行运算，设计四元式 (<postOp>, a, , a)
4. ASTArrayAccess，用于访问数组地址，以二维数组举例，若要访问 a[i][j]，则我们需要先定位位置，即 (*, i, 第二维大小 sizem, %1)，然后 (+, %1, j, %1)。访问时候为 (=[], %<id>, arrName, %<id+1>)

最后，我们设计有关语句的四元式。

1. ASTGotoStatement，跳转语句，设计四元式 (Jmp, , , labelName)
2. ASTLabeledStatement，标记语句，设计四元式 (Label, , , labelName)。该语句用于设置标签，方便中间代码执行选择、循环语句时进行跳转
3. ASTFunctionCall, 调用函数
   - 先设置参数，设计四元式 (Arg, 函数名 funcName, , ArgName)
   - 最后调用函数，设计四元式 (Call, 函数名 funcName, , 返回值 returnValue)
4. ASTReturnStatement，返回语句，设计四元式 (Ret, , , returnValue)
5. ASTSelectionStatement，选择语句，分为三部分
   - 第一部分。对 if 语句设计标签四元式 (Label, , , @If<id>)，对 else 语句设计四元式 (Label, , , @Else<id>)，在最末尾设计四元式 (Label, , , @Endif<id>)
   - 第二部分。处理完 If 语句中的 Statement，得到结果 Status，判断如果 Status 为 0，则跳转到 @Else<id>，对此设计四元式 (Jnt, Status, , @Else<id>)
   - 第三部分。如果执行完 if 中的内容后，要跳过 else 语句，实际四元式 (Jmp, , , @Endif<id>)
6. ASTIterationStatement，循环语句，分为三部分
   - 第一部分。对 for 语句设计开头标签四元式 (Label, , , @For<id>)，(Label)，结尾标签四元式 (Label, , , @EndFor<id>)，并在 cond 前设计标签四元式 (Label, , , @CondFor<id>)
   - 第二部分。在 cond 后如果不满足条件则退出循环。设计四元式 (Jnt, cond, , @EndFor<id>)
   - 第三部分。在每次 step 后无条件跳转回 cond 前，设计四元式 (Jmp, , , @CondFor<id>)

对于上面的选择语句，举例如下：

```
if(a > 0){
```

```
        b ++;
    } else {
        b --;
    }
```

它对应的四元式应该是：

```
(Label, , , @If<id>)
(>, a, 0, %1)
(Jnt, %1, 0, @Else<id>)
(++, b, , b)
(Jmp, , , @Endif<id>)
(Label, , , @Else<id>)
(--, b, , b)
(Label, , , @EndIf<id>)
```

## 2.2 代码实现

在设计完四元式之后，我们可以对 Example 中的代码按照我们的设计进行扩写。

### 2.2.1 Example 代码结构简析

icgen 文件下一共有 5 个程序文件，其中 Quat 类为四元式类，TemporaryValue 类为临时变量类，需要我们进行扩充和编写的只有 ICBuilder 和 ICPrinter 类。

### 2.2.2 TagLabel 类编写

因为我们增添了语句标记 (@If<id> 等)，故新建了 TagLabel 类，它继承自 ASTNode 类。

### 2.2.3 MyICBuilder 类编写

我们仍然使用 visitor 模式对 AST 树进行遍历，并在这个过程中生成我们要想得到的四元式。

限于篇幅原因，我们仅在这里选取几个较有特点的部分进行分析，完整代码详见 src 文件夹下。

首先对于 ASTDeclaration 类，我们可以在此生成变量、数组相关定义的四元组。具体地，我们还需要判断是否有初始赋值，而初始值中又分类各类表达式，需要我们逐一判断。在这里我们选取数组定义为例，对于 int a[10][20]，有如下展平 AST 后的内容：

```
{
    "type": "Declaration",
    "specifiers": [
        {
            "type": "Token",
            "value": "int",
            "tokenId": 21
        }
    ],
    "initLists": [
        {
            "type": "InitList",
            "declarator": {
                "type": "ArrayDeclarator",
                "declarator": {
                    "type": "ArrayDeclarator",
                    "declarator":{
                        "type": "VariableDeclarator",
                        "identifier": {
                            "type": "Identifier",
                            "value": "a",
                            "tokenId": 22
                        }
                    },
                    "expr": {
                        "type": "IntegerConstant",
                        "value": 10,
                        "tokenId": 24
                    }
                },
                "expr": {
                    "type": "IntegerConstant",
                    "value": 20,
                    "tokenId": 27
                }
            },
```

```
            "exprs": [

            ]
        }
    ]
}
```

可以看到我们的数组名"a" 是被嵌套在最里面的，所以在生成四元式时，我们通过循环取出 expr 中的数组大小并插入 LinkedList 首部的方式得到数组类型 arrType。具体代码如下：

```
LinkedList<Integer> arrayMem = new LinkedList<Integer>();
// 取出数组各维上限，合成 arrType
while (true) {
    int memSize = ((ASTIntegerConstant) expr).value;
    arrayMem.addFirst(memSize);

    if (arrayDeclarator instanceof ASTArrayDeclarator) {
        expr = ((ASTArrayDeclarator) arrayDeclarator).expr;
        arrayDeclarator = ((ASTArrayDeclarator) arrayDeclarator).declarat
    } else {
        break;
    }
}
```

对于循环而言，按照我们上面设计的四元式，有如下代码：

```
public void visit(ASTIterationStatement iterationStat) throws Exception {
    // TODO Auto-generated method stub
    if (iterationStat == null) {
        return;
    }
    // 打上开始标记
    String ForBegin = "@For" + this.ForId++;
    TagLabel BeginLabel = new TagLabel(ForBegin);
    Quat quat = new Quat("Label", BeginLabel, null, null);
    quats.add(quat);
```

```
    if (iterationStat.init != null) {
        for (ASTExpression init : iterationStat.init)
            this.visit(init);
    }
    // 条件节点的 Label
    String condstr = "@CondFor" + this.ForId;
    TagLabel condLabel = new TagLabel(condstr, this.quats.size());
    Quat quat1 = new Quat("Label", condLabel, null, null);
    quats.add(quat1);

    if (iterationStat.cond != null) {
        for (ASTExpression cond : iterationStat.cond)
            this.visit(cond);
    }
    this.visit(iterationStat.stat);
    // 如果不满足，跳到结束
    String Endstr = "@EndFor" + this.ForId;
    TagLabel EndLabel = new TagLabel(Endstr);

    Quat quat2 = new Quat("Jnt", EndLabel, null, null);
    quats.add(quat2);

    if (iterationStat.step != null) {
        for (ASTExpression step : iterationStat.step)
            this.visit(step);
    }
    // 无条件跳回 cond，循环
    Quat quat3 = new Quat("Jmp", condLabel, null, null);
    quats.add(quat3);
    //结束标记
    Quat quat4 = new Quat("Label", EndLabel, null, null);
    quats.add(quat4);
}
```

### 2.2.4 MyICPrinter 类编写

首先我们需要在程序最上面导入全所有 AST 类，即 import bit.minisys.minicc.parser.ast.*。

然后补全 astStr 函数，如下：

```
private String astStr(ASTNode node) {
    if (node == null) {
        return "";
    } else if (node instanceof ASTIdentifier) {
        return ((ASTIdentifier) node).value;
    } else if (node instanceof ASTIntegerConstant) {
        return ((ASTIntegerConstant) node).value + "";
    } else if (node instanceof ASTFloatConstant) {
        return ((ASTFloatConstant) node).value + "";
    } else if (node instanceof ASTCharConstant) {
        return ((ASTCharConstant) node).value;
    } else if (node instanceof ASTStringConstant) {
        return ((ASTStringConstant) node).value;
    } else if (node instanceof TemporaryValue) {
        return ((TemporaryValue) node).name();
    } else if (node instanceof TagLabel) {
        return ((TagLabel) node).name;
    } else if (node instanceof ASTVariableDeclarator) {
        return ((ASTVariableDeclarator) node).getName();
    } else if (node instanceof ASTFunctionDeclarator) {
        return ((ASTFunctionDeclarator) node).getName();
    } else if (node instanceof ASTToken) {
        if(((ASTToken) node).value.equals("int")) return "int";
        if(((ASTToken) node).value.equals("float")) return "float";
        if(((ASTToken) node).value.equals("char")) return "char";
        return "";
    } else {
        return "";
    }
}
```

# 3 实验结果

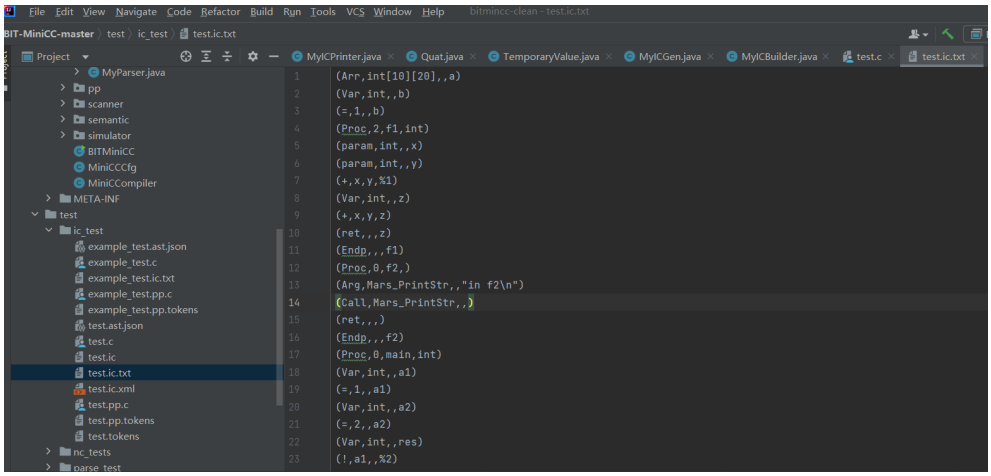
## 3.1 配置 *config.xml*

配置 *config.xml*，具体内容如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<config name="config.xml">
    <phases>
        <phase>
          <phase skip="true" type="java" path="" name="preprocess" />
          <phase skip="false" type="java" path="" name="scan" />
          <phase skip="false" type="java" path="" name="parse" />
          <phase skip="false" type="java" path="" name="semantic" />
          <phase skip="false" type="java" path="bit.minisys.minicc.icgen.
          ...
        </phase>
    </phases>
</config>
```

## 3.2 运行截图

编译 test.c 文件，结果如下。




可以看到结果和预期一致。

# 4 实验心得与体会

本次实验我对于 AST 类和 C++ 基础文法有了更深刻的掌握，并学会了如何设计四元式。

# 参考文献

[1] *Lab 5 语法分析说明及要求.pdf*. zh. 2023.

[2] *Lab 7 中间代码生成说明及要求.pdf*. zh. 2023.