代码生成实验

姓　　名：高　　月

学　　号：1120141944

班　　级：07121402

指导老师：计 卫 星

目录

一、实验目的

通过实践环节深入理解与编译实现有关的形式语言理论基本概念，掌握编译程序构造的一般原理、基本设计方法和主要实现技术，并通过运用自动机理论解决实际问题，从问题定义、分析、建立数学模型和编码的整个实践活动中逐步提高软件设计开发的能力。

二、实验内容

该实验选择 C 语言的一个子集，基于 BIT-MiniCC 构建 C 语法子集的代码生成模块，该语法分析器能够读入 XML 文件形式的语法分析树，进行寄存器分配，并生成 MIPS 或者 X86 汇编代码。

需要说明的是，生成汇编程序是该部分的基本要求；进一步地，如果生成的是 MIPS 汇编，则要求汇编代码能够在 BIT-MiniCC 集成的 MIPS 模拟器中运行（注意 config.xml 的最后一个阶段“simulating”的“skip”属性配置为“false”）；如果生成的是 X86 汇编，则要求使用 X86 汇编器生成 exe 文件并运行。

三、实验环境

本次实验系统环境等信息如下表所示：

OS	IDE	Java SDK
macOS Sierra 10.12.6	IntelliJ IDEA 17.2	1.8 update 144

四、实验步骤

4.1 选择实验所用汇编语言

考虑到 X86 汇编不便于在 macOS 上运行，以及 MIPS 指令集的简洁性，本次实验选择生成 MIPS 汇编。

MIPS 汇编本身指令复杂度较低，指令比较规整，指令类型相比 X86 汇编少得多，从而比较容易掌握， 本次实验中使用了 MIPS 指令系统中具有代表性的 R 型运算指令：`add sub mul div`，I 型运算指令 `addi` ，I型条件分支指令 `ble bne`， J 型无条件分支指令 `j` ，以及装载立即数伪指令：`li`。

4.2 配置 BIT-MiniCC 实验框架

1. 修改 comfig.xml 文件：

```
<phase skip="false" type="java" path="bit.minisys.minicc.GCodeGenerator" name="codegen" />
<phase skip="false" type="java" path="input/test.code.asm" name="simulating" />
```

2. 修改输入文件问语法分析输出文件：

```
method.invoke(c.newInstance(), pOutFile, cOutFile);
```

3. 修改代码生成输出文件后缀：

```
public static String MINICC_CODEGEN_OUTPUT_EXT = ".code.asm";//生成x86或者MIPS汇编代码
// input and output for simulator
public static String MINICC_ASSEMBLER_INPUT_EXT = ".code.asm";//目标代码
```

4.3 代码生成程序编写

4.3.1 确定程序框架

由于此次实验是建立在语法分析实验的基础之上，所以程序按照语法分析实验中所确定的 C 语言子集的文法编写。实验输入为 XML 形式语法分析树，所以根据 XML 文件中的 TagName 确定各符号或表达式的类型。

确定的代码实现框架如下：

1． 全局变量：

```
private int labelNumber;
private int registerTNumber;                // Register number $t0 ~ $t7

private Map<String, Integer> registerTable; // variable associated register
private Map<String, Integer> dataTable;     // data segment symbols

private BufferedWriter bufferedWriter;
private FileWriter fileWriter;

private String dataSegmentString;
private String codeSegmentString;
```

2． XML 文件读写：

```
private void parseXMLFile(String iXMLFile) {...}

private void saveAsmFile() {...}
```

3． 语法分析树元素的处理：

```
private void processFunction(Element functionElement) {...}
private void processFuncCodeBlock(Element codeBlock) {...}
private void processFuncStatement(Element statementElement) {...}
private void processAssignStatement(Element assignStatement) {...}
private void processCompareExpression(Element compareExpression) {...}
```

4． 寄存器的分配：

```
private String associateID2Reg(String identifier) {...}
```

5．MIPS 汇编代码的生成：

```
private void generateValue2Reg(String reg, String value) {...}
private void generateReg2Reg(String regRT, String regRS) {...}
private void generateADDI(String regRT, String regRS, String value) {...}
private void generateADD(String regRD, String regRS, String regRT) {...}
private void generateSUB(String regRD, String regRS, String regRT) {...}
private void generateMULRRI(String regRT, String regRS, String immediate) {...}
private void generateMULRRR(String regRD, String regRS, String regRT) {...}
private void generateDIVRRI(String regRT, String regRS, String immediate) {...}
private void generateDIVRRR(String regRD, String regRS, String regRT) {...}
```

6．run 函数：

```
init(oFile);

generateDataSegmentHeader();
generateTextSegmentHeader();

parseXMLFile(iFile);
saveAsmFile();
```

4.3.2 程序关键部分实现说明

1. 文件读写：

实验中将数据段、代码段的语句分别临时写在字符串变量中，语法分析树处理完毕后再使用 `BufferedWriter` 一次性写入数据段、代码段语句。

2. 语法分析树的处理：

实验中使用 DOM Parser 处理 XML 文件，根据 C 语言子集文法及语法树结构获取相应 TagName，进而生成相应的语句或进一步调用处理函数。如函数定义的处理函数具体实现如下：

```
private void processFunction(Element functionElement) {
    NodeList childNodes = functionElement.getChildNodes();
    // get function name
    for (int i = 0; i < childNodes.getLength(); i++) {
        if (childNodes.item(i).getNodeType() == Node.ELEMENT_NODE ) {
            Element child = (Element) childNodes.item(i);
            if (child.getTagName().equals("identifier")) {
                codeSegmentString += child.getTextContent()+":\n";
            }
        }
    }
    NodeList funcCodeBlock = functionElement.getElementsByTagName("CodeBlock");
    if (funcCodeBlock.item( index: 0).getNodeType() == Node.ELEMENT_NODE) {
        processFuncCodeBlock((Element) funcCodeBlock.item( index: 0));
    }

}
```

3. 标号生成算法思想：

由于实验中仅涉及 if 分支语句，while 循环语句，故先定义一个全局变量标号序号初始化为零。if 条件转型的比较表达式选择条件的反判断条件（如 == 使用 bne 不等于），不满足条件时转移到当前标号序号出，if 代码块结束后就在生成的代码创建此标号，再将全局变量自增 1，为后续标号做准备。当遇到 while 代码块时先创建标号（创建后全局变量标号序号仍自增 1），再转化比较表达式（与 if 类似），while 代码末尾处再创建一个无条件转移汇编语句此时标号为全局变量标号序号减 1（创建后不自增），再创建一个 while 条件不满足的转移的标号。

代码实现如下：

（1）汇编语句生成函数：

```
private void generateBranch(String operator, String comparer1, String comparer2) {
    String branchSmt = "\t"+operator+"\t"+comparer1+", "+comparer2+", "+"L"+labelNumber+"\n";
    codeSegmentString += branchSmt;
}

private void generateLabel() {
    System.out.println("createdLabel: "+labelNumber);
    String label = "L"+labelNumber+":\n";
    labelNumber++;
    codeSegmentString += label;
}

private void generateJump() {
    String jumpSmt = "\tj\t"+"L"+(labelNumber-1)+"\n";
    codeSegmentString += jumpSmt;
}
```

（2） if 代码块处理详细实现：

```
else if (firstChild != null && firstChild.getNodeName().equals("BranchStatement")) {
    Element branchStatement = (Element) firstChild;
    Element compareExpression = (Element) branchStatement.
            getElementsByTagName("CompareExpression").item( index: 0);
    processCompareExpression(compareExpression);
    Element codeBlockStatement = (Element) statementElement.
            getElementsByTagName("CodeBlock").item( index: 0);
    NodeList assignStatementList = codeBlockStatement.
            getElementsByTagName("AssignStatement");
    for (int i = 0; i < assignStatementList.getLength(); i++) {
        processAssignStatement((Element) assignStatementList.item(i));
    }

    // create label
    generateLabel();
}
```

（3） while 代码块处理详细实现：

```
else if (firstChild != null && firstChild.getNodeName().equals("LoopStatement")) {
    Element loopStatement = (Element) firstChild;
    Element compareExpression = (Element) loopStatement.
            getElementsByTagName("CompareExpression").item( index: 0);
    // compare expression label
    generateLabel();
    processCompareExpression(compareExpression);

    Element codeBlockStatement = (Element) statementElement.
            getElementsByTagName("CodeBlock").item( index: 0);
    NodeList assignStatementList = codeBlockStatement.
            getElementsByTagName("AssignStatement");
    for (int i = 0; i < assignStatementList.getLength(); i++) {
        processAssignStatement((Element) assignStatementList.item(i));
    }

    generateJump();
    // exit loop label
    generateLabel();
}
```

注：汇编分支指令生成函数再比较表达式处理函数中调用

（4） 汇编运算指令生成函数：

此类指令比较容易，主要涉及 1~3 个寄存器或立即数的操作，其中 addi 和 add 实现如下：

```
private void generateADDI(String regRT, String regRS, String value) {
    String addiSmt = "\taddi\t"+regRT+", "+regRS+", "+value+"\n";
    codeSegmentString += addiSmt;
}

private void generateADD(String regRD, String regRS, String regRT) {
    String addSmt = "\tadd\t"+regRD+", "+regRS+", "+regRT+"\n";
    codeSegmentString += addSmt;
}
```

（5） 寄存器的分配：

当 XML 元素处理函数处理到标识符时，调用 associateID2Reg() 函数获取标识符对应的寄存器。此函数先查看该标识符是否已经分配寄存器，若未分配则分配一个寄存器，并把分配的寄存器编号及标识符放入 registerTable (Map<String, Integer>) 中。具体实现如下：

```
private String associateID2Reg(String identifier) {
    String register;
    if (!registerTable.containsKey(identifier)) {
        registerTable.put(identifier,registerTNumber);
        register = "$t"+(registerTNumber-8);
        registerTNumber++;
    } else {
        register = "$t"+(registerTable.get(identifier)-8);
    }
    return register;
}
```

五、实验结果

5.1 编译的 C 语言程序

编译的 C 语言中使用了多个变量、赋值语句、分支语句已经循环语句，如下所示：

```c
int main ( ) {
    int a = 10 ;
    int b = 100 ;
    int c;
    c = b / a ;
    if ( c == 10 ) {
        temp = b * 2 ;
        a = temp + b ;
    }

    a = a + 2;

    while( a > b ) {
        a = a - 10;
    }
    d = 100 ;
    return 0;
}
```

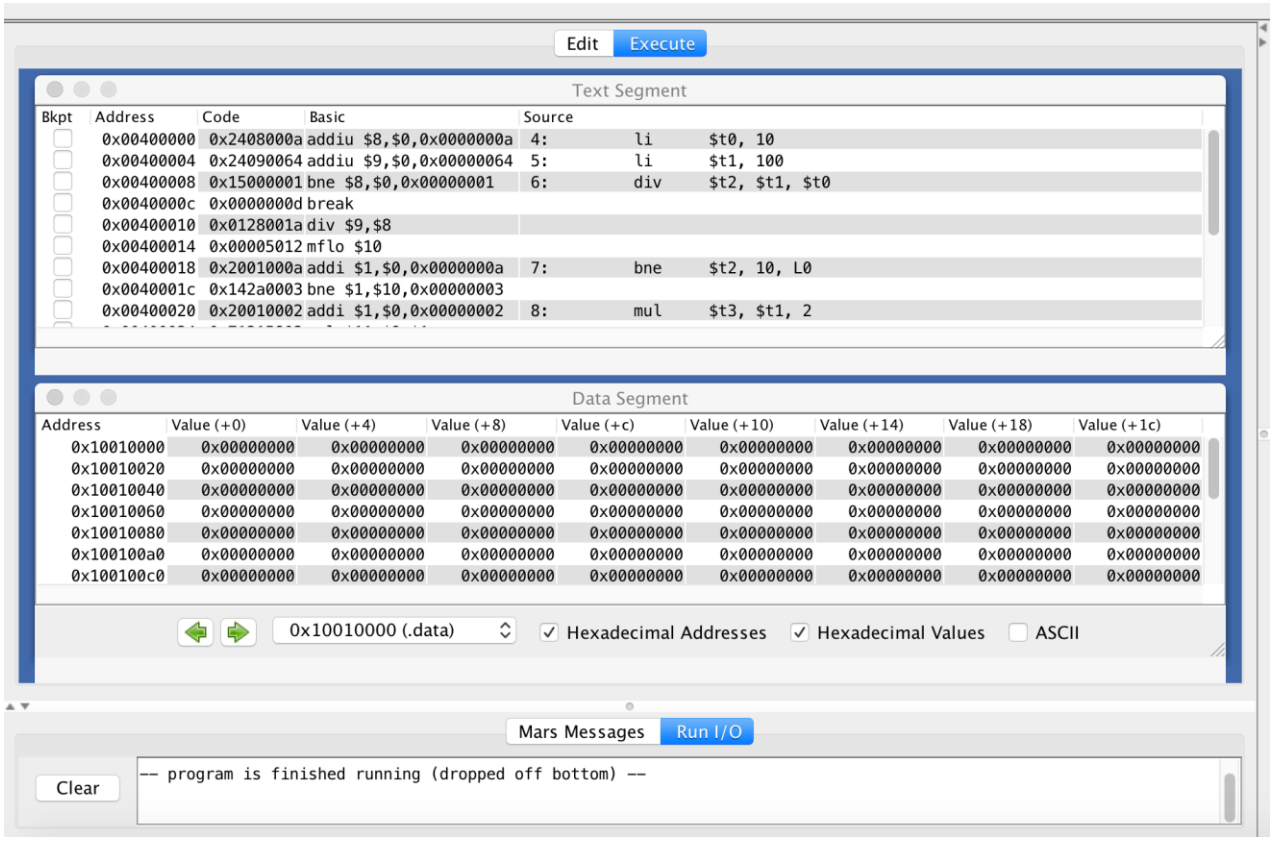
5.2 编译后的 MIPS 汇编程序

```
1          .data
2          .text
3   main:
4          li      $t0, 10
5          li      $t1, 100
6          div     $t2, $t1, $t0
7          bne     $t2, 10, L0
8          mul     $t3, $t1, 2
9          add     $t0, $t3, $t1
10  L0:
11         addi    $t0, $t0, 2
12  L1:
13         ble     $t0, $t1, L2
14         addi    $t0, $t0, -10
15         j       L1
16  L2:
17         li      $t4, 100
```

从结果中可以看出 MIPS 指令系统的简洁性。

5.3 MIPS 程序在 Mars 中运行结果

代码段和数据段如下图所示：



运行后寄存器组如下图所示：

六、实验心得

本次实验相比前几次实验难度适中，难点在于对 MPS （X86）指令系统的理解，及如何高效通过语法分析树生成对应的汇编代码。

通过本次实验我深刻的理解了 MIPS 指令系统和 X86 指令系统的优缺点，这两个指令系统分别代表了指令系统发展的两个趋势 RISC 和 CISC，很难说孰优孰劣。在工业中 Intel 处理器底层的硬件实现基本用了 MIPS 的设计思想，而 X86 却在个人计算机等领域广泛应用。

本次实验中所实现的代码生成程序在功能上仍需完善，比如未考虑复杂表达式的生成、以及函数的调用等问题，不过基本 MIPS 指令比如简单的加减乘除运算指令，分支循环指令等都能准确无误的生成对应汇编代码。

暑期中由于准备升学相关事务，此次实验完成得比较仓促，仅完成了基本功能，远不能作为一个可以实用的编译器。但通过一学期的编译器原理及设计课程的学习，对编译器从理论到实践都有了深刻的认识。顺利完成这几次实验也让我掌握了编译程序构造的一般原理、基本设计方法和主要实现技术。