《编译技术》课程设计申优文档

目录：

一、总体架构

编译器总体架构总共包括以下几个部分

1. `Compile`类，也是编译器主类，主要完成数据输入读取，以及调整数据输出方向，以及调用各模块对原文件进行分析。
2. `Sym`类，词法分析程序，对读入的原文件进行预处理，并对其进一步进行词法分析，用一个`list`存储所有单词的类别，内容，行号等信息，并将结果导入语法分析程序，从而进行进一步处理。
3. `Parsing`类，语法分析程序（**错误处理**），读入词法分析结果后对其进行语法分析，在分析的过程中同时建立符号表，并对可能存在的错误进行错误处理。
4. `ParsingTemp`类，语法分析程序（**构建语法树**），该类通过语法分析生成AST，便于下一步对AST解析生成中间代码。
5. 除此之外还有几个`package`分别存储后续处理的不同模块。
    - `Symbol_table`分别存储4个符号类别（数组类别，函数类别，变量类别，常规数类别）和2个符号表类别（函数符号表和变量符号表），前者组合起来完成符号表的搭建工作。
    - `AST`存储所有所有抽象语法树中的节点，生成中间代码将通过解析AST，遍历分析各节点完成。
    - `Optim`完成了部分代码优化和寄存器分配的工作。
    - `Midcode`和`Mipscode`分别对应了中间代码和目标代码生成和解析的相关过程。

编译器总体完成了词法分析，语法分析，语义分析和生成中间代码，代码优化，代码生成5个模块，同时在5个模块中又穿插实现了错误处理和符号表建构两个模块，从而完善编译器的功能和作用，目标语言为MIPS。

二、词法分析

1.编码前设计

词法分析程序作为编译器最先完成的部分，主要实现了两大功能，一是实现文件输入输出的功能，二是进行具体的词法分析。

> 首先**文件输入输出功能**，输入最开始使用了`String source=Files.readString(Paths.get(inputpath))`的函数将原文件一次性读入到一个字符串中，输出则采用了`out = new PrintStream(outputpath); System.setOut(out)`的方式将输出重定向至指定的文件当中。

词法分析程序主要进行了3个步骤

1. 保留字和部分符号对应编号的初始化
2. 文件预处理
3. 词法分析

文件预处理中主要是针对单行注释和多行注释进行特殊处理，将注释内容进行去除，同时避免产生错误，同时特判了在标准字符串`"..."`中间的内容，并对换行符"\r"进行了消去，在文件最后追加了"$"作为词法分析结束标志。并重新储存预处理完的字符串。

词法分析的具体程序便是根据每个单词的组成结构并进行部分预读，确认单词的所属类别，进而将其保存在一个名为`Word`的单词类中，存储在`words`的list结构中，其中比较特殊的是`identifier`和保留字类，需要判断是否存在该保留字否则即为`ident`类，当读入空格换行符等就跳过，读到`$`时词法分析程序结束。

2.编码后修改

完成词法分析程序后，在debug的过程中进行了一系列或大或小的修改。

1. 由于原来的文件读入函数对于评测机的路径读取有误，因此更改了文件读入的方式

```
    public static void readToBuffer(StringBuffer buffer, String filePath) throws
IOException {
        InputStream is = new FileInputStream(filePath);
        String line; // 用来保存每行读取的内容
        BufferedReader reader = new BufferedReader(new InputStreamReader(is));
        line = reader.readLine(); // 读取第一行
        while (line != null) { // 如果 line 为空说明读完了
            buffer.append(line); // 将读到的内容添加到 buffer 中
            buffer.append("\n"); // 添加换行符
            line = reader.readLine(); // 读取下一行
        }
        reader.close();
        is.close();
    }

        StringBuffer sb=new StringBuffer();
        try {
            Compiler.readToBuffer(sb,inputpath);
        } catch (IOException e) {
            e.printStackTrace();
        }
        String source=sb.toString();
```

使用了逐行读入的方式。

2. 在原词法分析程序中缺少读取存入行号的机制，为了之后的错误处理可以正常进行，增加了对于换行符的特殊处理，并对每一个单词存入了行号信息。

3. 在每一轮判断分支的时候未考虑首读入字符是结束符`$`的情况，因此可能导致文件错误输出，对其进行了修改。

4. 预处理程序中对于连续的多行注释处理可能存在问题，重新优化了原有的循环逻辑。

三、语法分析

1.编码前设计

语法分析程序首先编写了两个读取和获取栈顶单词的程序。

```
    private Word getWord() {
        if (index < words.size()) {
//            System.out.print(Parsing.Symbol.values()
[words.get(index).getSymnumber()]);
//            System.out.print(' ' + words.get(index).getContent() + '\n');
            return words.get(index++);
        } else {
            return new Word();
        }

    }

    private Word showWord() {
        if (index < words.size())
            return words.get(index);
        else
            return new Word();
    }
```

首先要使用递归下降分析，需要去除左递归，即对原文法中存在的左递归文法进行改写。

而具体的语法分析程序主要是**根据文法进行递归下降**，对于每一种可能的分支进行判断，并进入对应的分支程序中，而对不符合任何语法条件的情况进行`error()`，其中分支判断可能需要预读多至几个单词，才能正确进入分支，**对于比较复杂的情况可能需要进行大量的预读才能判断出分支路径。**

举例说明对于`stmt`中的`[exp];`和`Lval =...`的情况就需要比较精细的分析（因为exp就可能是Lval）,程序中使用的方法是一直预读至一个`Lval`结束，若其后的元素为=，则可判断。

2.编码后修改

1. 在对原文法的左递归修改时，虽然可以正确解析但是输出语法元素出现了问题，因此对所有的处理左递归文法的子程序进行了输出Bug的修改。
2. 由于递归下降思路较为清晰，完成代码编写和基本的debug之后没有其他本质性的内容修改。

四、错误处理

1.编码前设计

错误处理部分最重要的变化是增加了 **符号表**，通过符号表的组织和整理可以解决部分错误处理中提到的问题，而组织方法则用到了前文中提到的两个`package`分别存储4个符号类别（数组类别，函数类别，变量类别，常规数类别）和2个符号表类别（函数符号表和变量符号表）。同时在进入不同的作用域时及时更新符号表的层次关系。

在每次遇到数据声明的时候，就将其存入符号表中，不过在这之前需要判断是否存在**重定义错误**并对之后的每一个引用检查其是否在当前的作用域中存在，存在即满足条件，不存在则进入外层的作用域中查找，若到最外层仍未找到，则为**未定义错误**。

对于函数声明和调用也采用了类似的存表取数据的方式，进行不断的对应，不过函数声明和调用增加了形参和实参的数据存储以及新的函数类型。

错误处理的部分主要就是在之前语法分析的基础上建立关于错误的种种分析方法，下面的部分主要介绍部分错误的处理方法。

1. 非法符号，该错误在词法分析的过程中进行标记，创建了`word`的一个派生类专门存储是否存在该问题。
2. b,c类错误都在上文中提到
3. 函数参数个数不匹配，读取实参和形参的个数进行比较
4. 函数参数类型不匹配，主要是对于每一个exp表达式储存当前状态最高级的变量作为**实参维数**来和形参比较

2.编码后修改

错误处理部分经过较多次数的修改和debug，因为本身错误处理对于原本的架构逻辑影响较大，同时错误又较难定位。以下是debug过程中的一系列修改

1. 原代码对于函数参数不匹配的情况仍会进行函数参数类型的判断，容易产生re等相关问题
2. 对于不同的作用域进出判断，这里会涉及到几个问题，正常会在每个`BlockItem`的进入前后设置作用域变动，而当涉及到函数会在判断形参的时候就提前进入作用域的情况进行特判以及无`BlockItem`的情况产生矛盾，因此对这一部分进行了更精确的修改。
3. 最后一个bug较不易发现，但是很重要便是函数调用的实参可能存在多层嵌套，而对于每一层嵌套我们都需要引入一个栈来对其进行维护，从而可以保证函数类型层次判断不会出错。

五、语义分析

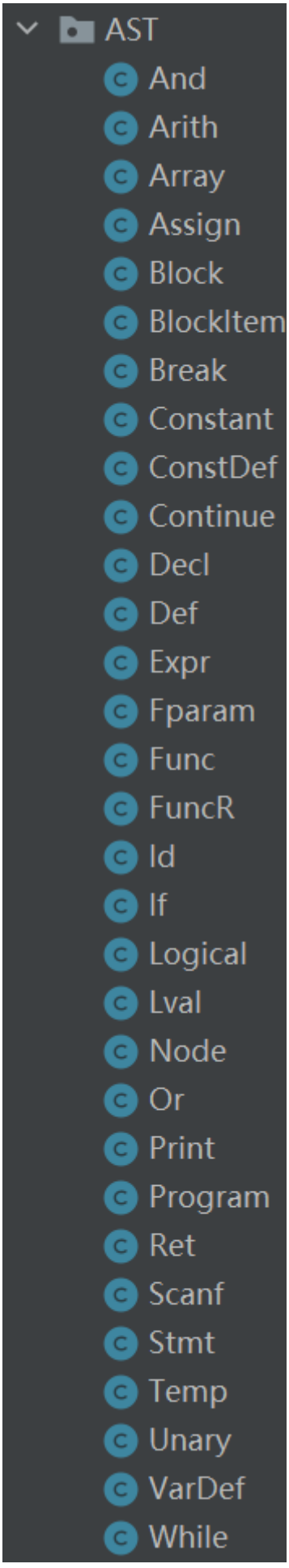
1.编码前设计

完成错误处理之后来到了无比重要的中间代码和目标代码生成环节，为了降低语法分析模块的耦合度，减少中间代码的生成难度，我选择使用语法分析生成AST，为之后的中间代码生成做好准备。

这个部分里需要根据各节点的特点选择合适的继承关系，例如整个表达式部分其实都可以继承同一个节点`Expr`，因为每个节点都有相似的特点，因此解析只要递归地驱动该节点的子节点进行解析就可以完成，而对于各`Stmt`可能就没有一个统一的解析方式，因此解析时往往需要对每个节点进行特殊设计。

建立抽象语法树的一大好处是它瞬间让整个源代码的架构变得无比清晰，同时剔除了在原文法中存在的无意义信息，后续解析也只需要对于语法树进行分析，就可以得到完整的结果。

下图为代码中AST的各节点类


AST
And
Arith
Array
Assign
Block
BlockItem
Break
Constant
ConstDef
Continue
Decl
Def
Expr
Fparam
Func
FuncR
Id
If
Logical
Lval
Node
Or
Print
Program
Ret
Scanf
Stmt
Temp
Unary
VarDef
While

2.编码后修改

编码部分主要是在语法分析中实现AST生成，对于节点进行各种设计，这一步思路较为清晰，没有太多复杂和需要修改的空间。

六、中间代码生成

1.编码前设计

这一步主要是对上一步产生的AST进行驱动解析，需要在每一个节点中完成对应的解析方法和设计中间表达式输出。我这里选择的是输出四元式序列。

```
public enum operation {
    MAIN,   //主函数进入入口
    PLUSOP, //+
    MINUOP, //-
    MULTOP, //*
    DIVOP,  // /
    MODOP,  // %
    LSSOP,  //<
    LEQOP,  //<=
    GREOP,  //>
    GEQOP,  //>=
    EQLOP,  //==
    NEQOP,  //!=
    ASSIGNOP,  //=
    GOTO,  //无条件跳转
    Jump,  //跳转标记
    BZ,    //不满足条件跳转
    BNZ,   //满足条件跳转
    PUSH,  //函数调用时参数传递
    CALL,  //函数调用
    RET,   //函数返回语句
    RETVALUE, //有返回值函数返回的结果
    SCAN,  //读
    PRINT, //写
    LABEL, //标号,区分不同作用域和标号
    CONST, //常量
    ARRAY, //数组
    VAR,   //变量
    FUNC,  //函数定义
    PARAM, //函数参数
    GETARRAY,  //取数组的值  t = a[]
    PUTARRAY,  //给数组赋值  a[] = t
    EXIT,  //退出 main最后
    DEBUG,
    SLL,   //左移
    SRA,   //右移
```

以上每一种类型都对应不同的节点。每一个四元式都要确保信息充分，并且利于被目标代码生成的模块进行解析，因此一些运算能在这部分完成为优，后续也可在代码优化中看到这个部分的重要性。

2.编码后设计

实际撰写中间代码生成的时候，还是会发现诸多的问题，这里我主要分析几个我碰到的较为麻烦的问题和解决方案。

1.作用域处理

由于在本次作业的文法中会频繁涉及到作用域的问题，而作用域又会与符号表建构和后续目标代码生成联系在一块，为了保留作用域进出的信息方便后续的解析，因此在每一个`Block`的进出我都会输出相应的`label`，识别该label即可判断作用域的情况，另外由于函数的作用域进入不是在`Block`开始，而是在参数表就开始，因此对于函数的判定还需要进行特殊处理。

2.常数计算

由于中间代码输出需要对数组进行预先计算，而对于二维数组又会涉及到第二维度的数值，因此我在中间代码生成模块完成了**常量计算**，方便对于各种数值的处理，首先对所有的常数定义，都需要写入符号表，并在符号表中保存对应符号的值，而碰到`ConstExp`的节点时，就需要驱动其`caculate`方法对各`Expr`节点进行计算，由于已验证文法的正确性，因此不需要判断是否该节点可以计算，只要对路径中的各节点计算并返回值就可以。

3.数组节点处理

数组作为其中最复杂的模块进行了比较全面和特殊的处理。由于会涉及到**数组以指针和值两种形式存在**，数组还涉及到不同的维数，以及数组结构（`putaaray`和`getarray`）的复杂性，做了以下处理。

1.计算模块对两种类型数组值进行计算，需要查表和计算两个维度的具体值。

```java
public int calculate() {
    IntergerTable table = inttable;
    ArraySymbol sym = null;
    while (table != null) {
        if (table.contains(op.getContent())) {
            sym = (ArraySymbol) table.get(op.getContent());
            break;
        }
        table = table.getOut();
    }
    int k1 = 0, k2 = 0;
    if (twoindex == null) {
        k2 = oneindex.calculate();
    } else {
        k1 = oneindex.calculate();
        k2 = twoindex.calculate();
    }
    return sym.getValue( i: k1 * sym.getLevel2() + k2);
}
```

2.由于给数组赋值的情况，输出不能用中间节点代替，故而对数组输出进行特殊化。`toString` 方法返回具体的数组形式（对于二维也需要提前计算偏移量输出）

```java
@Override
public String toString() {                    //数组被赋值等情况时，返回实际的Index
    if (twoindex == null) {
        return oneindex.reduce().toString();
    } else {
        Temp temp1 = new Temp(op);
        Temp temp2 = new Temp(op);

        IntergerTable table = inttable;
        ArraySymbol sym = null;
        while (table != null) {
            if (table.contains(op.getContent())) {
                sym = (ArraySymbol) table.get(op.getContent());
                break;
            }
            table = table.getOut();
        }

        int index2 = sym.getLevel2();

        emit(new midCode(midCode.operation.MULTOP, temp1.toString(), oneindex.reduce().toString(), String.valueOf(index2)));
        emit(new midCode(midCode.operation.PLUSOP, temp2.toString(), temp1.toString(), twoindex.reduce().toString()));
        return temp2.toString();
    }
```

3.数组正常解析输出。

这一部分主要是判断是否为指针以及是二维还是一维数组，并取出对应的维度输出，对于每一种情况都要充分的考虑和调试，才能输出正确的中间代码形式。

七、目标代码生成

目标代码生成的任务是将四元式翻译为目标代码，根据mips指令集架构的特点进行对应的分析和转化，其实就可以完成，但是对于细节和设计方案需要提前考虑清楚，例如函数栈的设计，内存分配，指针数组处理的部分都是比较重要的模块。

1.内存分配

未优化之前的版本代码生成使用全写内存的方式完成变量的引用和存储。在代码生成最开始的部分会对每个函数所需要的存储空间进行计算，具体的解析时候每进入一个新的函数都会分配充分的空间用来保存变量，同时变量的存储区分成全局变量和局部变量两种情况，**全局变量**引用是通过 `$gp` 寄存器向上的偏移量计算得到，局部变量是通过 `$fp` 寄存器向下的偏移量计算得到。因此偏移量也要通过符号表进行维护，及时保存和读取变量偏移量。

2.函数栈的设计

函数调用这一部分较为复杂，而如何正确地设计函数栈也成为了重中之重，函数调用涉及到移动 `$sp` 和 `$fp` 指针以及保护现场等操作，具体的实现如下。

```
for (int j = 0; j < pushOpstcak.size(); j++) {
    midCode mcs = pushOpstcak.get(j);
    if (mcs.x != null) {
        loadAddress(mcs.z,  regName: "$t0");
        String addr = loadValue(mcs.x,  regName: "$t1",  tableable: false);
        mipscodes.add(new Mipscode(Mipscode.operation.li,  z: "$t2",  x: "",  y: "",  imme: Integer.parseInt(mcs
        mipscodes.add(new Mipscode(Mipscode.operation.mult,  z: "$t2", addr,  y: ""));
        mipscodes.add(new Mipscode(Mipscode.operation.mflo,  z: "$t2"));
        mipscodes.add(new Mipscode(Mipscode.operation.add,  z: "$t0",  x: "$t0",  y: "$t2"));
        mipscodes.add(new Mipscode(Mipscode.operation.sw,  z: "$t0",  x: "$sp",  y: "",  imme: -4 * j));
    } else {
        String addr = loadAddress(mcs.z,  regName: "$t0");
        mipscodes.add(new Mipscode(Mipscode.operation.sw, addr,  x: "$sp",  y: "",  imme: -4 * j));
        ;
    }
}
pushOpstcak.clear();
```

```
ArrayList<String> lists = register.getReverlists();
int len = lists.size();
mipscodes.add(new Mipscode(Mipscode.operation.addi, z: "$sp", x: "$sp", y: "", imme: -4 * funclength.get(mc.z)));
mipscodes.add(new Mipscode(Mipscode.operation.sw, z: "$ra", x: "$sp", y: "", imme: 4));
mipscodes.add(new Mipscode(Mipscode.operation.sw, z: "$fp", x: "$sp", y: "", imme: 8));
for (int k = 0; k < len; k++) {
    mipscodes.add(new Mipscode(Mipscode.operation.sw, lists.get(k), x: "$sp", y: "", imme: 12 + 4 * k));
}
//  mipscodes.add(new Mipscode(Mipscode.operation.sw, "$t8", "$sp", "", 32));
//  mipscodes.add(new Mipscode(Mipscode.operation.sw, "$t9", "$sp", "", 36));
mipscodes.add(new Mipscode(Mipscode.operation.addi, z: "$fp", x: "$sp", y: "", imme: 4 * funclength.get(mc.z)));
mipscodes.add(new Mipscode(Mipscode.operation.jal, mc.z));
//  mipscodes.add(new Mipscode(Mipscode.operation.lw, "$t9", "$sp", "", 36));
//  mipscodes.add(new Mipscode(Mipscode.operation.lw, "$t8", "$sp", "", 32));
for (int k = len - 1; k >= 0; k--) {
    mipscodes.add(new Mipscode(Mipscode.operation.lw, lists.get(k), x: "$sp", y: "", imme: 12 + 4 * k));
}

mipscodes.add(new Mipscode(Mipscode.operation.lw, z: "$fp", x: "$sp", y: "", imme: 8));
mipscodes.add(new Mipscode(Mipscode.operation.lw, z: "$ra", x: "$sp", y: "", imme: 4));
mipscodes.add(new Mipscode(Mipscode.operation.addi, z: "$sp", x: "$sp", y: "", imme: 4 * funclength.get(mc.z)));
```

这一步主要完成两个工作。

1. 函数传参
2. 保存现场、函数调用和恢复现场

函数传参是在 `$sp` 指针向下的空间依次存放变量和指针地址（这种情况需要特殊处理），之后就是下移 `$sp` 指针，将当前 `$fp` 和 `$ra` 的值保存在地址空间中，再下移 `$fp` ，进行函数调用，从而实现了**保存现场**，恢复现场就相当于是上述方法的逆过程，反向读取寄存器值，恢复两个指针到原来的位置。

3.指针数组处理

由于存在数组为指针或者直接在当前函数栈内被定义两种情况，因此首先需要在符号表里对数组类型进行标记，如在函数参数里被定义的数组一定为指针类型，在给数组赋值和取数组值需要分别对两种情况处理，同时函数调用时也需要取出数组的绝对地址进行赋值，确保程序的一致性。

八、代码优化

代码优化主要实现以下几个部分的优化

1.寄存器分配

本实验中我完成了临时寄存器优化，由于考虑到在我的代码架构中，每个中间变量都只会被赋值和使用一次，因此我维护了一个寄存器池，并在每一次中间变量申请寄存器时赋给其一个空闲寄存器，而在该中间变量被再次使用时，将该寄存器进行释放，从而实现了所有的中间变量，用寄存器进行读写操作。

此外对于函数调用，由于也会涉及到当前寄存器的使用和覆盖，因此维护一个list记录当前被占用的寄存器，对所有被占用的寄存器在调用前写入内存，在返回时再重新写入寄存器，从而解决函数调用可能产生的冲突。

```java
package Optim;

import ...

public class Register {
    HashMap<String, String> maps = new HashMap<>();
    ArrayList<String> lists;
    ArrayList<String> reverlists;

    public Register() {
        List list = Arrays.asList("$t3", "$t4", "$t5", "$t6", "$t7", "$t8", "$t9");
        lists = new ArrayList<>(list);
        reverlists = new ArrayList<>();
    }

    public String findtemp(String name) {
        String reg = maps.get(name);
        lists.add(reg);
        reverlists.remove(reg);
        return reg;
    }

//    public String usetemp(String name) {
//        if (name.equals("$t3") || name.equals("$t4") || name.equals("$t5") || name.equals("$t6") || name.equals("$t7") ) {
//            lists.add(name);
//        }
//        return name;
```

2.乘除优化

1.基本优化：对于乘除2的幂次的情况，将其转化为左移，右移解决，此外对于被除数为负数的情况，需要将其加上（除数-1），再进行计算否则会出现问题。

2.除法特殊优化：对于除数为3,5,7等特殊数字，可以将其转化为乘以一个magic number，然后再进行相关的右移操作进行转化。具体实现参考了编译器是如何将特殊常数的除法转换为等效的乘

法的 - huyutian - C++博客 (cppblog.com)。此附计算表。

Table 10-1. Some Magic Numbers for W = 32

	Signed		Unsigned		
d	M(hex)	s	M(hex)	a	s
-5	99999999	1			
-3	55555555	1			
-2^k	7FFFFFFF	k-1			
1	-	-	0	1	0
2^k	80000001	k-1	2^{32-k}	0	0
3	55555556	0	AAAAAAAB	0	1
5	66666667	1	CCCCCCCD	0	2
6	2AAAAAAB	0	AAAAAAAB	0	2
7	92492493	2	24924925	1	3
9	38E38E39	1	38E38E39	0	1
10	66666667	2	CCCCCCCD	0	3
11	2E8BA2E9	1	BA2E8BA3	0	3
12	2AAAAAAB	1	AAAAAAAB	0	3
25	51EB851F	3	51EB851F	0	3
125	10624DD3	3	10624DD3	0	3
625	68DB8BAD	8	D1B71759	0	9

3.取模优化：a%b=a-a/b*b，其中除法可以用上述的转化方法进行优化。

3.常量计算

该步骤在AST生成中间代码时实现，对于每个`Expr`节点实现一个`cancaculate`方法判断当前节点能否计算成一个常数值，并提供一个value属性保存可能计算出的value值，具体遍历时先判断其能否计算，在其子节点可以计算或者其自身满足计算条件时，返回可以计算的信号，并更新自己的value值。最终输出时将所有已经转化为常量值的节点直接按值输出（常量数组也会被提前计算）

4.窥孔优化

窥孔优化指在范围较小的短序列进行局部优化，本实验完成了以下窥孔优化：

1. 对于跳转到紧随其后标签的j指令，将其删除
2. 对于跳转位置a标记紧跟的下一条语句为j跳转b语句的情况，将跳转到a的句子改为跳转到b，具体操作为维护一个Map，不断更新每一个跳转语句最终的跳转位置，直到map不再变动，然后将所有的跳转语句更新为其最终跳转位置。