写在前面

本文档在作者任务逐步完成过程中撰写，虽然文档本身**包含所要求的内容**，但并没有显示的章节标题指明，为保证原本清晰的结构，作者在每一章开头用加粗字体指明所要求内容对应的章节。

参考编译器设计（LLVM）

参考对学长的请教，无论最后是否进行竞速排名，生成中间代码时最好选择生成llvm_ir，因此这里选择参考 LLVM 这一编译器框架系统

总体结构

- **前端**：负责词法分析、语法分析、符号表建立和错误处理。
- **中端**：负责生成中间代码LLVM以及中间代码优化。
- **后端**：负责生成目标代码MIPS以及后端代码优化

接口设计

遵循高内聚、低耦合的原则

具体功能在每一子模块中实现，子模块提供顶层接口，主类Compiler仅保留调用子模块的接口。

文件组织

- `llvm/cmake`：CMake配置文件。
- `llvm/examples`：示例代码。
- `llvm/include`：公共头文件，包括LLVM-specific头文件和不同部分的LLVM子目录。
- `llvm/lib`：大部分源文件，按库组织，便于代码共享。
- `llvm/bindings`：为非C/C++语言提供的LLVM基础设施绑定。
- `llvm/projects`：不是LLVM严格部分的项目，但随LLVM一起发布，也是创建自己的LLVM-based项目的目录。
- `llvm/test`：测试代码。
- `llvm/tools`：LLVM工具。
- `llvm/utils`：辅助工具

编译器总体设计

总体结构

作者希望该编译器能进行明显的模块化，对编译过程的每一步骤，通过一个模块实现。

由于编译器流程是线性的，所以每完成一个模块，我们只需要输出下一模块所需要的输入即可

接口设计

这里借鉴llvm编译器高内聚、低耦合的原则

对于每一模块，构造main方法，作为顶层接口

文件组织

这里对参考编译器的文件进行简化，以模块的角度进行文件的划分。

对每一模块，单独构建文件夹，大致为

- Compiler.java：主类，调用模块顶层接口
- lex：词法分析模块
- parser：语法分析模块
- symbol：语义分析模块
- llvm：生成中间代码模块
- optimize：代码优化模块
- mips：生成mips模块

1 文法解读

本部分较为简单编码前设计和编码完成之后的设计一致

这一部分任务主要撰写测试样例，在构建样例的过程中，重点在于可以详细预读和理解文法的细节，才能够构建出具有充足覆盖度的样例

在这一部分，只需要详细阅读文法规则，设计语句遍历所有情况即可

2 词法分析

本任务**编码前的设计**参考“2.2.1 一次错误的尝试”，**修改后的内容**参考其余部分

2.1 任务分析

这一部分任务，主要将读到的代码内容划分为token，再判断每一个token的词法成分并输出

2.2 token划分

2.2.1 错误的尝试

思路：逐个字符读入，依据总结规律确定token划分

以空格为分隔符，对保留字，字符常量、数字常量、注释、操作符做特殊处理，其余情况看作ident进行处理

问题：①总结工作过于繁琐②存在“二义性情况”

2.2.1 思路矫正

我这里采用的token划分方式较为简单，但存在潜在风险

我构建出如下的匹配模式：

```
String patternString =
            "(\\b(main|const|int|char|break|continue|if|else|for|return|void|while)\\b)|"
+ // 关键字
            "(\\b(getint|getchar|printf)\\b)|"+//函数
            "(\\b[a-zA-Z_][a-zA-Z0-9_]*\\b)|" + // 标识符
            "(\\b\\d+\\b)|" + // 常量
            "(\".*?\")|" +
            "(//)|"+//单行注释
            "(==|!=|<=|>=|<|>|&&|\\|\\||=|\\+|-|/\\*|\\*/|/\\*|\\*/|\\*|/|%|\\{|\\}|\\
(|\\)|\\[|\\]|;|,|&|!|\\|)|" + // 操作符
            "(\'(\\\\.|[^\\\']|)\')|"+
            "(\\s+)"; // 空白字符
```

并利用内置函数对所读文本进行token匹配

```
Pattern pattern = Pattern.compile(patternString);
Matcher matcher = pattern.matcher(code);
```

经过如下简单处理后，即课输出一个完整的字符串数组用于存储所有token

```
while (matcher.find()) {
            if(matcher.group().equals("/*")){
                zhuflag=true;
            }
            if(matcher.group().equals("*/")){
                zhuflag=false;
                continue;
            }
            if(matcher.group().equals("//")&&!zhuflag) break;
            if (!matcher.group().matches("\\s+")&&!zhuflag) { // 忽略空白字符
                tokens.add(matcher.group());
            }
        }
```

上述代码，即构成了我的tokenize函数

2.3 词法成分分析

我把所有词法分析分为两类，关键字和其他其他成分(**Ident**,**IntConst**,**StringConst**,**CharConst**)

对于关键字，我建立了如下字典

```
private static final Map<String, String> words = new HashMap<>();
static {
        words.put("main", "MAINTK");
        words.put("const", "CONSTTK");
        words.put("int", "INTTK");
        words.put("char", "CHARTK");
        words.put("break", "BREAKTK");
        words.put("continue", "CONTINUETK");
```

```
        words.put("if", "IFTK");
        words.put("else", "ELSETK");
        words.put("for", "FORTK");
        words.put("getint", "GETINTTK");
        words.put("getchar", "GETCHARTK");
        words.put("printf", "PRINTFTK");
        words.put("return", "RETURNTK");
        words.put("void", "VOIDTK");
        words.put(";", "SEMICN");
        words.put("!", "NOT");
        words.put("*", "MULT");
        words.put(",", "COMMA");
        words.put("&&", "AND");
        words.put("||", "OR");
        words.put("/", "DIV");
        words.put("%", "MOD");
        words.put("(", "LPARENT");
        words.put(")", "RPARENT");
        words.put("[", "LBRACK");
        words.put("]", "RBRACK");
        words.put("{", "LBRACE");
        words.put("}", "RBRACE");
        words.put("<", "LSS");
        words.put("<=", "LEQ");
        words.put(">", "GRE");
        words.put(">=", "GEQ");
        words.put("==", "EQL");
        words.put("!=", "NEQ");
        words.put("+", "PLUS");
        words.put("-", "MINU");
        words.put("=", "ASSIGN");
    }
```

用于判断其语法成分

对于其他成分，我则为每一成分写了一个判断函数，以正则匹配为原理，判断一个token是否属于该成分

以Ident为例：

```
    private static boolean isIdentifier(String token) {
        return token.matches("[a-zA-Z_][a-zA-Z0-9_]*");
    }
```

2.4 代码架构

结合上述两个部分，首先对读入文件进行逐行token划分，再遍历所有token，逐个判断其词法成分，即完成了这部分任务

3 语法分析

本任务**编码前的设计**参考“3.2 一次错误的尝试”，**修改后的内容**参考“3.3 思路矫正”

3.1 任务分析

这一部分任务，主要根据词法分析结果，将阅读到的程序内容，按照题设语法规则，构建出一课语法树，并按照要求输出语法树存储的信息

3.2 一次错误的尝试

由于题目阅读不清和自己的错误判断，我的第一次尝试，采用了一种有问题的思路。由于我认为其问题值得在日后借鉴，所以首先记录这种错误的思路。

在这种思路下，我并未直接构建语法树，而希望通过一系列函数去模拟语法树的构建，建立一套具有普适性的语法成分判断体系。

然而很快，我就发现了这种思路的问题：

第一，不能构建出语法树，这在之后的代码中需要使用

第二，这种“普适性的语法成分判断体系”并不存在或难以建立，语法成分判断依旧需要通过“预读”的方式，通过建立理论课中提到过的“first”串来判断语法成分

3.3 思路矫正

通过回顾理论课知识，我的代码思路最终得到了矫正，整体思路如下：

- 对token进行依次遍历，依照语法，将代码构建为一个语法树
- 对于语法成分判断问题，采用“预读”的方式实现“first”串，依次对语法成分进行选择，在题设语法中，除去特殊情况，大部分判断可以被预读的2个token实现
- 后序遍历语法树，即可正确地按要求输出结果

3.4 代码架构

代码主要可以分为以下几个部分：

- tokenize函数，用于将输入文本划分为token
- 词法分析判断函数，函数名为“is+词法成分”
- 主函数，主要是词法分析的部分，添加调用语法分析入口函数
- 语法分析的“建树”函数，函数名为”parser+语法成分“，主要是一系列相互调用的函数，用于构建语法树
- 语法分析的判断函数，函数名为“is+语法成分”，主要用于语法成分判断
- 与树结构本身相关的函数，主要有addNode,insertNode,deletNode和postTraversal四个函数，用于操作树结构本身

除此之外，为方便搭建语法树，我还定义了一个树节点的结构，定义源码存储在“dataStructure”文件夹下，定义了树节点的结果，主要定义了三种节点：ENode(错误节点)，NNode(非终结符节点，用于存语法成分)，TNode（终结符节点，用于存token）

期中考试

烂！！！！！！

问题一

debug思路有问题，不够清晰，导致debug方向错误，浪费大量时间做无用功。这或许是因为考试紧张导致的。

此能力日积月累形成，无法速成，多写代码！！！！

问题二

代码结构不够清晰，存在冗余代码，需要整理

4 语义分析

这里完全按照教程思路

4.1 符号表结构设计

4.1.1 一次错误的尝试

这里由于没有读清题目，将题目中的层次编号理解为理论课内容中提到的层次号混淆，最初设计将符号表以理论课中提到的栈式符号表的方式组织

在完成这部分之后，发现输出内容与题目要求不符，故更改了思路

4.1.2 思路纠正

依靠树结构组织基本块间的关系，每一个树节点对应一个基本块

用队列组织每个基本块内的符号表

```
Codeium: Refactor | Explain
public class STTQue {

    public ArrayList<Element> que;
    public int front;
    public int rear;
    public int level; // 队列的层级
    public Element ret;

    Codeium: Refactor | Explain
    public static class Element {
        public int level;
        public String name;
        public String type;
        public String kind;

        public Element(int level, String name, String type,String kind) {
            this.level = level;
            this.name = name;
            this.type = type;
            this.kind=kind;
        }
    }
    public STTQue(int level) {
        this.que = new ArrayList<>();
        this.front = -1;
        this.rear = 0;
        this.level=level;
        this.ret = null;
    }
```

4.2 错误处理

单独构建checkX类，组织所有的错误检查方法，检查到错误后输出对应的错误信息

5 中间代码生成（LLVMIR）

本部分自己没有产生合理的思路，按照课程网站中给出教程实现，中间没有对思路进行调整

5.1 文件结构

本次任务的代码主体在llvm文件夹下，**代码顶层接口为`llvm.ir.Module.main`**

llvm文件结构如下：

```
--llvm
    |----AddTreeNode.java（特殊结构AddTree的结点定义）
    |____ir
```

```
        |----NameAllocator.java（Value编号）
        |----StrNameAllocator.java（printf用字符串的名字编号）
        |----Moudle.java（llvm的顶层模块）
        |____value
               |----AddExp.java（处理语法树中的AddExp结点）
               |----BasicBlock.java（处理Block）
               |----FormatString.java（处理printf定义的全局字符串）
               |----Function.java（处理函数定义，包括main函数）
               |----FunctionParam.java（处理函数参数）
               |----GlobalArray.java（处理全局数组，继承自GlobalValue）
               |----GlobalValue.java（处理全局变量）
               |----GlobalVar.java（处理全局非数组变量，继承自GlobalValue）
               |----ImmediateValue.java（处理立即数，包括数字常量和字符常量）
               |----InitVal.java（处理变量定义的初值）
               |----Label.java（处理标签）
               |----StringConst.java（处理字符串常量）
               |----Use.java
               |----User.java
               |----Value.java（一切皆Value，所有类的父类）
               |----inst（LLVM的每种指令一个类）
               |       |----AddInst.java
               |       |----AllocaInst.java
               |       ……
               |____Type
                       |----Type.java（父类）
                       |----ReturnType.java（返回值类型）
                       |____VarType.java（变量类型）
```

5.2 体系结构

- Module
  - 全局量（Global Value）
    - 常量
    - 变量
  - 函数定义（Function，包括其他函数和main函数）
    - Block（BasicBlock）
      - 各类指令

因此，全局量和函数定义在顶层类Moudle中处理，函数定义中只处理BasicBlock，各类指令在BasicBlock中处理。

5.3 具体实现

由上可知，顶层module模块中只需要实现对全局变量和函数定义的处理即可。因此在module中设置两个ArrayList属性，分别记录全局变量和函数定义

```
public static ArrayList<GlobalValue> globalValues = new ArrayList<>();
public static ArrayList<Function> functions = new ArrayList<>();
```

5.3.1 全局变量

根据sysy语法，为module添加正确的GlobalValue实例即可

5.3.2 函数定义

与上面类似的，根据sysy语法，为module添加正确的Function实例即可，这里不同前面，需要对参数进行分析，主要过程如下：

```
ReturnType retType=new ReturnType(symbol.getASTNodeContent(parent, new int[] {0,0}));
String funcName=symbol.getASTNodeContent(parent, new int[] {1,0});
int paraNum=(parent.children.size()==5)?0:symbol.getASTNode(parent, new int[]
{3}).children.size()/2+1;
ArrayList<VarType> paraTypes=new ArrayList<VarType>();
if (paraNum==0) paraTypes=null;
String[] paraNames=new String[paraNum];
for(int i=0;i<paraNum;i++){
  VarType paraType=new VarType(symbol.getASTNodeContent(parent, new int[] {3,2*i,0,0}));
  paraNames[i]=symbol.getASTNodeContent(parent, new int[] {3,2*i,1,0});
  paraTypes.add(paraType);
// symbolStack.pushStack(1,symbol.getASTNodeContent(parent, new int[]
{3,2*i,0,0})+"Para",paraName,paraType);
}
Function newFunction=createFunction(retType,funcName, paraTypes);
symbolStack.pushStack(0,retType+"Func",funcName,newFunction);
```

同时，我们需要继续处理函数定义的Block内部的代码：

这里在参数处理方面，第一版采用了较蠢的处理方法，即在block的一开始，先对所有参数进行Alloca和Store，这实际上是一个很轻松就可以优化的点。

另外，之所以在这里才将参数推进栈式符号表，是为了保证在符号表中先推入函数再推入对应参数。

```
BasicBlock
newbasicblock=functions.get(functions.size()-1).createBasicBlock(newFunction,parent.child
ren.get(parent.children.size()-1),1,null);
//TODO：可优化，这样时间消耗比较大
for(int i=0;i<paraNum;i++){
  Value ptr=newbasicblock.createAllocaInst(paraTypes.get(i));
  newbasicblock.createStoreInst(newFunction.params.get(i), ptr, paraTypes.get(i));
  symbolStack.pushStack(1,paraTypes.get(i).type,paraNames[i],ptr);
}
newbasicblock.orderAST(parent.children.get(parent.children.size()-1));
symbolStack.rmCurLevel(1);
```

5.3.3 局部变量定义

```
Module->basicBlock->AddExp->xxInst
```

需要添加的指令：`alloca+处理AddExp+store`

5.3.4 赋值语句(不包含函数调用)

需要添加的指令：`load+处理AddExp+store`

5.3.5 函数调用

库函数

需要添加的指令：`getint/getchar:左值相关指令+call`

`printf:全局formatstring+处理AddExp{+putint}{+putch}`

自定义函数

需要添加的指令：`[左值相关指令+][处理AddExp+]call`

5.3.6 返回

需要添加的指令：`处理AddExp+return`

5.3.7 数组

数组声明

由于本代码在初始设计时分开处理全局变量和局部变量的声明，所以这里分别涉及两处：Moudle.java中（全局），LVal中（局部数组）

由于有关变量名的处理与普通变量几乎相同，所以下面主要阐述对初值的处理。

- 在对全局的处理中

  我们把对初值的处理全权交给Initial类，具体方法与下面对局部变量的处理相同，因此对于数组和变量，我们只需要调用createGlobalValue方法即可，区别只是唇乳InitVal参数是否为null

- 在局部处理中

  初值可以被分为`字符串常量`和`一般定义（即{exp,exp}）`，两种情况分开处理即可

数组使用

程序对数组的使用主要分为以下几种情况进行处理

	作用域内定义数组名	作用域内定义数组元素	作用域内函数数组名	作用域内参数数组元素	全局数组名	全局数组元素
LVal	/	形如： %4 = getelementptr inbounds [15 x i32], [15 x i32]* %2, i32 0, i32 %3 store i32 %5, i32* %4	/	形如： %2 = alloca i32 *store i32* %0, i32* *%2* *%5 = load i32,* i32* *%2* *%6 = getelementptr inbounds i32, i32* %5, i32 %4 store i32 %7, i32* %6	/	形如： %3 = getelementptr inbounds [15 x i32], [15 x i32]* @arr, i32 0, i32 %2 store i32 %4, i32* %3
AddExp	形如： (只能在函数调用中出现) %3 = getelementptr inbounds [15 x i32], [15 x i32]* %2, i32 0, i32 0 %4 = call i32 @func(i32* %3)	形如： %3 = getelementptr inbounds [15 x i32], [15 x i32]* %2, i32 0, i32 2 %4 = load i32, i32* %3 store i32 %4, i32* %1	形如： %2 = alloca i32 *store i32* %0, i32* *%2* *%3 = load i32,* i32* *%2* *%4 = call i32 @func2(i32* %3)*	形如： %2 = alloca i32 *store i32* %0, i32* *%2* *%5 = load i32,* i32* *%2* *%6 = getelementptr inbounds i32, i32* %5, i32 %4 %7 = load i32, i32* %6	形如： (只能在函数调用中出现) %2 = getelementptr inbounds [15 x i32], [15 x i32]* @arr, i32 0, i32 0 %3 = call i32 @func(i32* %2)	形如： %3 = load i32, ptr getelementptr inbounds ([15 x i32], ptr @arr, i64 0, i64 2), align 4 store i32 %3, ptr %2, align 4

分别在遇到对应情况时插入对应指令即可

5.3.8 if语句

if主要有如下结构：Cond,Stmt1,Stmt2,nextBasicBlock

这里我们需要解决的任务如下：

- 处理Cond中的LOrExp（包括短路求值）
- 插入对应cmp指令和Br指令
- 需要输出label时输出label

解决方式如下：

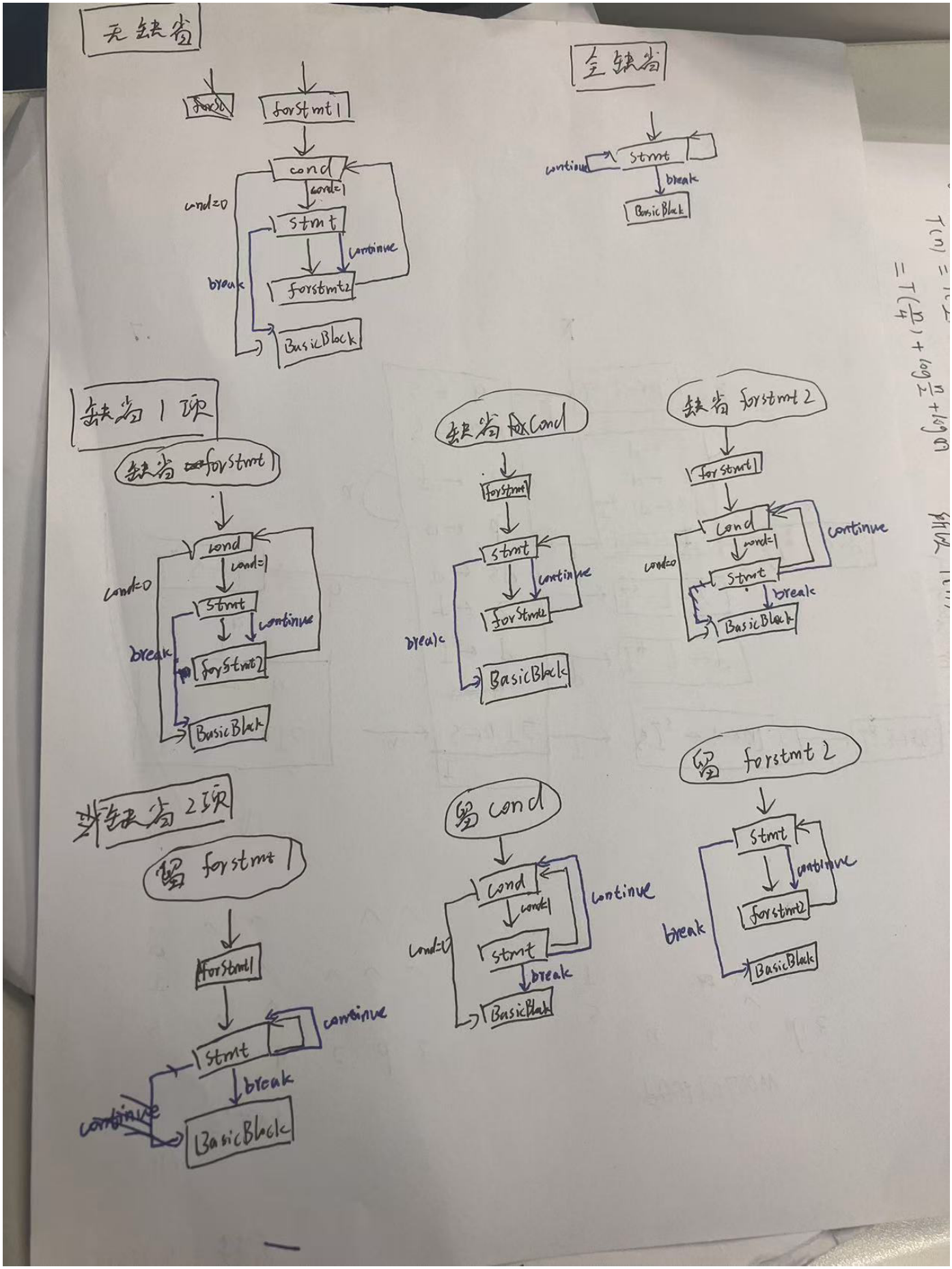
- LOrExp选择类似对AddExp的处理

  建立一个新的树称为CondTree，为了方便label输出以及跳转功能，我们令每一个CondTreeNode都有一个基本块，对应他的nowBasicBlock属性，同时，赋予trueBasicBlock和falseBasicBlock分别是该Node条件为真和为假时跳转的基本块

- 指令输出只需要在对应应当输出的位置添加createCmpInst和createBrInst方法的盗用即可
- 为BasicBlock添加label属性，对于需要输出label的基本块，为他的label赋值，否则为null

5.3.9 for语句

为缕清思路，首先列出所有缺省情况的流程图



设控制流入口基本块为entranceBasicBlock，每个子基本块(上述每个方框的基本块是子基本块，其中Cond可能为多个基本块)存在一个nextBasicBlock

汇总不难看出，

```
对于缺省：

- 缺省forStmt1对控制流无影响，因此我们可以把forStmt划在控制流之外，即forStmt1不可以作为
entranceBasicBlock
- 缺省Cond只影响entranceBasicBlock，若不缺省，则entranceBasicBlock为Cond，否则为Stmt
- 缺省forStmt2只影响Stmt的nextBasicBlock，若不缺省，则Stmt的nextBasicBlock为forStmt2，否则为
entranceBasicBlock

对于break和continue的作用

- break永远直接跳到BasicBlock
- continue让Stmt直接跳到它的nextBasicBlock
```

5.4 对AddExp的处理

由于代码中涉及广泛的对AddExp处理的需求，所以我建立了一个AddExp类，专门用于处理AddExp

处理思路如下：

- 把语法树转为特定结构AddTree，结点定义如下：

```
public class AddTreeNode {
    public String value; // 节点存储的字符串
    public List<AddTreeNode> children; // 子节点列表
    public Value exp;
    public String type;

    public AddTreeNode(String value) {
        this.value = value;
        this.children = new ArrayList<>();
    }

    // 添加子节点的方法
    public void addChild(AddTreeNode child) {
        children.add(child);
    }
}
```

在处理过程中，会面临四种情况：立即数、变量、函数调用前三者的运算式

立即数直接求值（对立即数间的运算，直接算出结果），变量遍历符号表，函数调用遍历函数表

最终得到一个三叉树，每个父节点保证有三个子节点，其中中间的为操作符，两侧的为操作数，操作数可能为以上四种的任意一种，具体解释如下：

1.立即数：value为立即数的值，exp为对应value

2.变量：value为变量名，exp为Load指令对应value

3.函数调用：value为func，exp为call指令对应的value

4.运算式：value初始为tmpp，在遍历AddTree的过程中，根据子节点情况生成

同时，所有结点的数据会被分为：int、char、intimm、charimm用于

- 遍历AddTree

  后序遍历，根据操作符、操作数进行指定输出即可

  每次遍历通过`flashType()`函数刷新AddExp的type属性：

```
public void flashType(AddTreeNode parent){
    type=parent.type;
}
```

需要注意的是，一旦涉及字符运算，就可以将字符立即数转为整形，如果是字符变量，进行类型转换

这里处理的逻辑，只要进入运算，先把所有字符常量（charImm）转为整形常量（intImm），如果左右两个节点类型都为intImm，

则直接计算得到父结点value，否则父结点一定为运算式，进行类型判断、计算指令添加即可：

```
AddTreeNode left,right;
left=parent.children.get(0);
right=parent.children.get(2);
if(left.type.equals("charImm")){
  left.value=String.valueOf((int)(left.value.charAt(1)));
  left.exp=new Value(left.value);
  left.type="intImm";
}
if(right.type.equals("charImm")){
  right.value=String.valueOf((int)(right.value.charAt(1)));
  right.exp=new Value(right.value);
  right.type="intImm";
}
System.out.println(left.value+" "+right.value);
if(left.type.equals("intImm")&&right.type.equals("intImm")){
  switch(parent.children.get(1).value){
    case "+":
      value=new
Value(String.valueOf(Integer.valueOf(left.value)+Integer.valueOf(right.value)));
      parent.exp=value;
      parent.value=value.name;
      parent.type="intImm";
      break;
    //...
  }
}
else{
  switch(parent.children.get(1).value){
    case "+":
```

```
      value=basicBlock.createAddInst((left.type.equals("char"))?
basicBlock.createZextInst(left.exp):left.exp, (right.type.equals("char"))?
basicBlock.createZextInst(right.exp):right.exp);
      parent.exp=value;
      parent.type="int";
      break;
    //...
  }
}
```

5.5 类型转换问题

5.5.1 转换逻辑

变量类型存在：int、char、intImm、charImm四种可能

以下为例：

```
if(((VarType)tmpType).type.equals("int")&&tmpAddExp.type.equals("char"))
from=createZextInst(from);
else if(((VarType)tmpType).type.equals("char")&&tmpAddExp.type.equals("int"))
from=createTruncInst(from);
else if(((VarType)tmpType).type.equals("int")&&tmpAddExp.type.equals("charImm")){
  from.name=String.valueOf((int)(from.name.charAt(1)));
}
```

只存在以上三种类型转换可能，其中字符立即数和整形立即数间可以相互转换，不必进行立即数

5.5.2 可能出现的场景

变量声明、AddExp处理、赋值语句、Printf、Return