编译设计文档

21373293　王雨帆

整体架构

此编译器是基于 SysY 语言的编译器，总体架构分为前端-终端-后端，其中后端包含了窥孔优化和乘除优化。

```
- BackEnd
    - assembly(包含各种mips指令)
        - AluAsm
        - Asciiz
        - Assembly（均是由Assembly extends 而来）
        - BranchAsm
        - CmpAsm
        - CommentAsm
        - GlobalAsm(只有GlobalAsm加入dataSegment)
        - ...
    - Optimize
        - MulDivOptimize
        - PeepholeOptimize
    - AssemblyTable（包含所有 mips 指令）
    - MipsBuilder
    - Register
- FrontEnd
    - Lexer
        - Input
        - Lexer
        - Token
        - TokenStream
        - TokenType
    - Parser
        - Node
            - AddExp
            - Block
            - BlockItem
            - CompUnit
            - Cond
            - ...
        - Parser
    - Symbol
        - ConstSymbol
        - FuncSymbol
        - VarSymbol
        - Symbol
        - SymbolType
        - SymbolTable(符号表)
        - SymbolStack（栈式符号表）
        - ErrorMessage
        - ErrorType
```

```
- LLVM
    - Instruction
        - AddInstruction
        - AllocaInstruction
        - AndInstruction
        - ...
    - BasicBlock
    - Constant
    - Function
    - GlobalVar
    - Init
    - Instr
    - IRBuilder
    - LLVMType
    - LocalVAr
    - Module
    - Param
    - Use
    - User
    - Value
- tools
    - Printer
    - RegAllocator
- Compiler
```

1 前端

1.1 词法分析器 Lexer

Lexer 将输入的字节流解析成一个一个的 token，并封装在 ArrayList 中。

```
        //词法分析
        Lexer lexer = new Lexer(source);
        TokenStream tokenStream = new TokenStream(lexer.getTokenArrayList());
```

1.1.1 编码前设计

由于刚开始实验的时候不知道后续还有错误处理，在 `getToken` 函数里写了很多如果输入错误的判断，并创造 `TokenType.ERROR`，如今看来是不必要的，因为错误处理的词法部分并不涉及这些，属于是当时想多了。其他编码前设计与下文一致

1.1.2 具体实现细节

运行

通过调用 `getTokenArrayList` 函数，如果 `ArrayList` 为空，则使用 `while` 循环不断调用 `getCurToken` 并填入 `ArrayList`。

`read` 封装 `InputStream.read()`，读取下一个字符。

空白符

- `' '` 或 `\t`
- 换行 `\r\n` 或 `\n`

解析 token

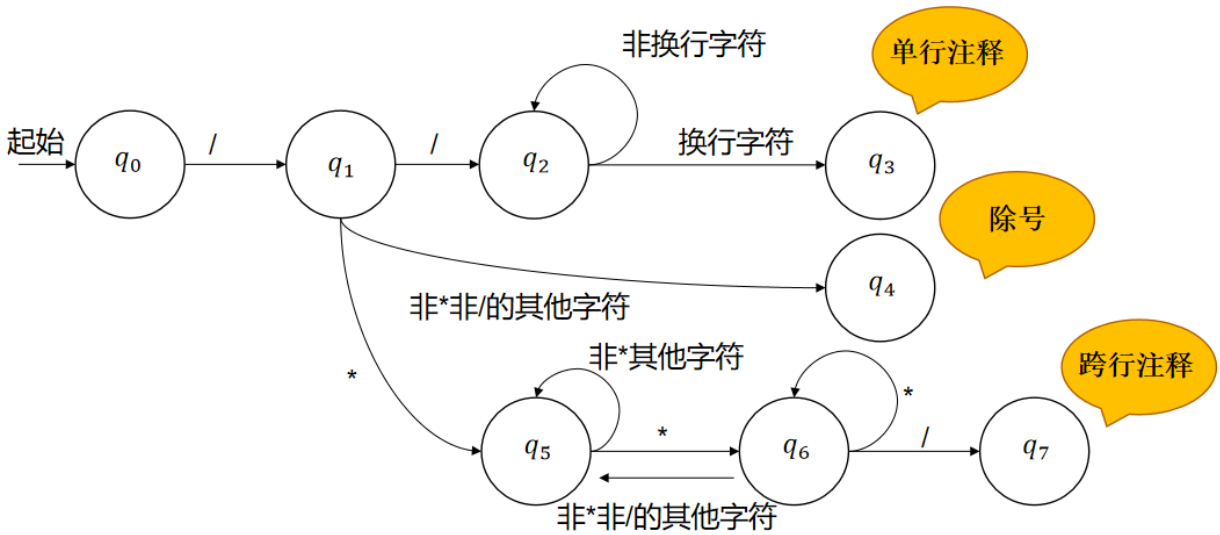
主要分成以下几个类

- skip 掉空白符
- EOF 结束标志
- 符号类
- 数字类
- string类
- ident类（标识符）
- 注释和除法类

注释和除法

这里使用 FSM

状态转移图跟PPT一致




```
        // DIV & note
        else if (curChar == '/') {
            sb.append(curChar);
            read();
            //div (q4)
            if (curChar != '/' && curChar != '*') {
                return new Token(TokenType.DIV,"/",curLine);
            }
            //only one line note (q2)
            else if (curChar == '/') {
```

```
                read();
                while (!isNewLine()) {
                    read();
                }
                //q3
                read();
                curLine++;
                return getCurToken();
            }
            //multi lines note (q5)
            //FSM
            else {
                read();
                int state = 5;
                while (state == 5 || state == 6) {
                    state = FSM(state);
                }
                if (state == 7) {
                    return getCurToken();
                } else {
                    return new Token(TokenType.ERROR,"/**/",curLine);
                }
            }

            private int FSM(int state) throws IOException {
        switch (state) {
            case 5:
                if (curChar == '*') {
                    read();
                    return 6;
                } else {
                    if (isNewLine()) {
                        curLine++;
                    }
                    read();
                    return 5;
                }
            case 6:
                if (curChar == '/') {
                    read();
                    return 7;
                } else if (curChar == '*') {
                    read();
                    return 6;
                } else {
                    if (isNewLine()) {
                        curLine++;
                    }
                    read();
                    return 5;
                }
            case 7:
```

```
                read();
                return 7;
            default:
                System.out.println("error in /**/");
                return -1;
        }
    }
```

1.1.3 反思

在 `getToken` 函数里写了很多如果输入错误的判断，并创造 `TokenType.ERROR`，如今看来是不必要的，因为错误处理的词法部分并不涉及这些，属于是当时想多了。

1.2 语法分析器 Parser

Parser 将得到的 Token 解析出语法成分，并构建语法树，最后后序遍历输出语法树。

```
        //语法分析
        Parser parser = new Parser(tokenStream);
        Node root = parser.parseCompUnit();
        //打印流
        PrintStream ps = new PrintStream("output.txt");
        System.setOut(ps);
        //后序遍历输出语法树
        visitNode(root);
```

1.2.1 编码前设计

主要思想还是递归下降，但是刚开始写的时候没学明白 FIRST 集这些理论知识，有些判断（比如 `[Exp] ;` `LVal = ...`）的判断写法并不适合后续的错误处理，还是因为没有严格按照 FIRST 集来判断，其他的设计与下文一致。

1.2.2 统一约定

- 默认读取到将要解析的 node 的第一个 token
- 调用完 parse 方法，则对应的 node 已经读完并得到
- TokenNode 需要手动 read 读完
- 获得 endLine 不能用 curToken 否则会出现 null pointer

1.2.3 实现细节

辅助方法

- `TokenStream.watch()` 帮助预读或者回读。

Node

- Node 非终结符

```
    private int startLine;
    private int endLine;
    private SyntaxType SyntaxType;
    private ArrayList<Node> children;
```

- TokenNode 终结符

  子节点集合为null，遍历到它时只输出 TokenType 和 内容。

```
public class TokenNode extends Node{
    private Token token;

TokenNode constNode = new
TokenNode(startLine,startLine,SyntaxType.TOKEN,null,curToken);
```

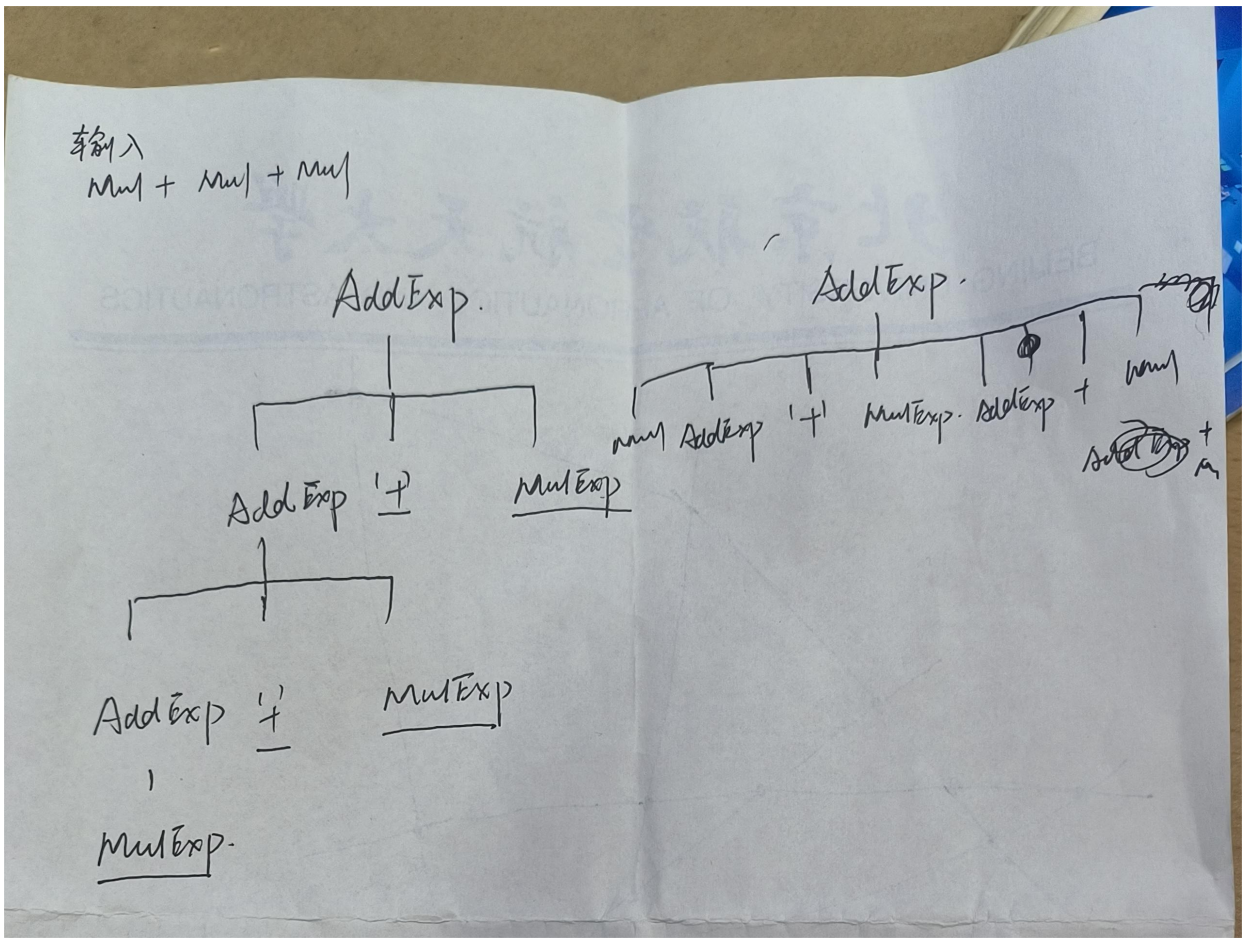
修改左递归文法

```
AddExp → MulExp | AddExp ('+' | '−') MulExp
AddExp -> MulExp {'+' MulExp | '-' MulExp}

MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
MulExp -> UnaryExp {'*' UnaryExp | '/' UnaryExp | '%' UnaryExp}

LOrExp → LAndExp | LOrExp '||' LAndExp
LOrExp -> LAndExp {'||' LAndExp}

RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
RelExp -> AddExp {'<' AddExp | '>' AddExp | '<=' AddExp | '>=' AddExp}

EqExp → RelExp | EqExp ('==' | '!=') RelExp
EqExp -> RelExp {'==' RelExp | '!=' RelExp}

LAndExp → EqExp | LAndExp '&&' EqExp
LAndExp -> EqExp {'&&' EqExp}
```

虽然修改文法可以消除左递归，但是会对语法树的建立以及语法成分的输出产生影响。因为我是后续遍历语法树输出语法成分，所以我在建立语法树时，**添加了一些 children 为 null 的非终结符作为叶子节点**，起到矫正语法输出的作用，一个例子如下。

```
public Node parseAddExp() {
    int startLine = curToken.getLine();
    ArrayList<Node> children = new ArrayList<>();
    //parse MulExp
    Node node = parseMulExp();
```

```
    children.add(node);
    //parse '+' or '-'
    while (curToken.getType() == TokenType.PLUS || curToken.getType() ==
TokenType.MINU) {
        //防止文法改变导致语法树输出改变
        node = new
Node(startLine,tokenStream.watch(-1).getLine(),SyntaxType.ADD_EXP,null);
        children.add(node);
        //parse '+' or '-'
        TokenNode pmNode = new
TokenNode(curToken.getLine(),curToken.getLine(),SyntaxType.TOKEN,null,curToken);
        children.add(pmNode);
        read();
        //parse MulExp
        node = parseMulExp();
        children.add(node);
    }
```



1.2.4 反思

- 递归下降的方法，通过 FIRST 集来判断该 parse 什么。
- 有些判断（比如 `[Exp] ;` `LVal = ...`）的判断写法并不适合后续的错误处理，还是因为没有严格按照 FIRST 集来判断，用了一些 trick 投机了。
- 很容易多读或者少读 token 导致后续出错，一定要写好注释，知道每一步在干什么。

1.3 错误处理

错误主要分为语法错误和语义错误，笔者的错误处理集成在 Lexer 和 Parser 里。

错误类型	错误类别码	解释	对应文法及出错符号 (... 表示省略该条规则后续部分)
非法符号	a	格式字符串中出现非法字符报错行号为 **<FormatString>** 所在行数。	<FormatString> → '"'{<Char>}'"'
名字重定义	b	函数名或者变量名在**当前作用域**下重复定义。注意，变量一定是同一级作用域下才会判定出错，不同级作用域下，内层会覆盖外层定义。报错行号为 **<Ident>** 所在行数。	<ConstDef>→<Ident> ... <VarDef>→<Ident> ... <Ident> ... <FuncDef>→<FuncType><Ident> ... <FuncFParam> → <BType> <Ident> ...
未定义的名字	c	使用了未定义的标识符报错行号为 **<Ident>** 所在行数。	<LVal>→<Ident> ... <UnaryExp>→<Ident> ...
函数参数个数不匹配	d	函数调用语句中，参数个数与函数定义中的参数个数不匹配。报错行号为函数调用语句的**函数名**所在行数。	<UnaryExp>→<Ident>'('[<FuncRParams>]')'
函数参数类型不匹配	e	函数调用语句中，参数类型与函数定义中对应位置的参数类型不匹配。报错行号为函数调用语句的**函数名**所在行数。	<UnaryExp>→<Ident>'('[<FuncRParams>]')'
无返回值的函数存在不匹配的return语句	f	报错行号为 **'return'** 所在行号。	<Stmt>→'return' {'['<Exp>']'}';'
有返回值的函数缺少return语句	g	只需要考虑函数末尾是否存在return语句，**无需考虑数据流**。报错行号为函数**结尾的'}'** 所在行号。	<FuncDef> → <FuncType> <Ident> '('[<FuncFParams>] ')' <Block> <MainFuncDef> → 'int' 'main' '(' ')' <Block>
不能改变常量的值	h	<LVal>为常量时，不能对其修改。报错行号为 **<LVal>** 所在行号。	<Stmt>→<LVal>'=' <Exp>';' <Stmt>→<LVal>'=' 'getint' '(' ')' ';'
缺少分号	i	报错行号为分号**前一个非终结符**所在行号。	<Stmt>,<ConstDecl>及<VarDecl>中的';'
缺少右小括号')'	j	报错行号为右小括号**前一个非终结符**所在行号。	函数调用(<UnaryExp>)、函数定义(<FuncDef>)及<Stmt>中的')'
缺少右中括号']'	k	报错行号为右中括号**前一个非终结符**所在行号。	数组定义(<ConstDef>,<VarDef>,<FuncFParam>)和使用(<LVal>)中的']'
printf中格式字符与表达式个数不匹配	l	报错行号为 **'printf'** 所在行号。	<Stmt> →'printf''('<FormatString>{,<Exp>}')'';'
在非循环块中使用break和continue语句	m	报错行号为 **'break'** 与 **'continue'** 所在行号。	<Stmt>→'break'';' <Stmt>→'continue'';'

1.3.1 编码前设计

在错误处理部分，笔者编码前后设计并未出现变化，详见下文叙述。

1.3.2 语法错误

对于词法错误 a，笔者在 Lexer 中进行处理，全部集成在对 FormatString 的分析中。

```
else if (curChar == '\\') {
                    sb.append(curChar);
                    read();
                    if (curChar != 'n') {
                        ErrorMessage errorMessage = new ErrorMessage(curLine,
ErrorType.a);
                        printer.addErrorMessage(errorMessage);
                        //把错误的字符读掉保存
                        //sb.append(curChar);
                        //read();
```

```
                    if (curChar != '"') {
                        sb.append(curChar);
                        read();
                    }
                }
            }
```

1.3.3 语义错误

对于语义错误，笔者在 Parser 里进行处理，在这之前需要先建好符号表，从而处理重定义、参数匹配等问题。

符号表

- 符号`ConstSymbol` `FuncSymbol` `VarSymbol` 均由 `Symbol` extend 而来：

```
//Symbol
public class Symbol {
    String name;
    SymbolType type;(其实不必要，因为后续都是extends而来)
    // 用于记录相应的 llvm
    Value ir;
    //在 ConstSymbol 和 VarSymbol 里被重写
    ArrayList<Integer> dims;

//ConstSymbol
public class ConstSymbol extends Symbol{
    //有几个'[]'
    private int dim;
    //被压成一维以前的维数，0维变量的size是0
    private ArrayList<Integer> dims;

public class VarSymbol extends Symbol{
    //有几个'[]'
    private int dim;
    //被压成一维以前的维数
    private ArrayList<Integer> dims;

public class FuncSymbol extends Symbol{
    private String returnType;//'void' or 'int'
    private int paramNum;
    private ArrayList<Integer> ParamDims;
```

- 每个模块的符号表其实就是一个 `hashmap`

```
public class SymbolTable {
    private HashMap<String,Symbol> symbolTable;
```

- 各模块的符号表组成一个**符号表栈**

```
public class TableStack {
    private Stack<SymbolTable> stack;
    private int depth;
    private int cur;

    private FuncSymbol curFunc;
```

- 符号表管理
  - 进入新模块的时候 `enterBlock` 建立一张新表并压入栈中。此处需要注意：**被调用函数仍属于上一层符号表**，要做好层次管理。

```
public Node parseCompUnit() {
        //...
        tableStack.enterBlock();

public Node parseFuncDef() {
        //...
        //建立新的符号表
        tableStack.enterBlock();

 public Node parseMainFuncDef() {
        //...
        //进入
        tableStack.enterBlock();

        //block
        //这种block是不必进行returnType检查的(g)
        //但是要新建符号表！！！
        if (curToken.getType() == TokenType.LBRACE) {
            //新建符号表
            tableStack.enterBlock();
            Node node = parseBlock();
```

  - 同理，在 parse 完以上语法成分时，调用 `leaveBlock` 弹出符号表。

return 检查

笔者的处理是设置一个全局的 `funcReturnCheck` 开关，注意：parse `Stmt` 中的 `block` 之前要将开关关掉，不必进行 `return` 检查，但是要保存以前的现场，以便调用完 `parseBlock` 之后恢复（其实这里用栈处理最好）。

```
//FuncDef
//parse block 可以returnTypeCheck
        funcReturnCheck = true;
        node = parseBlock();
        funcReturnCheck = false;
```

```
//MainFuncDef
//parse Block 可以 returnTypeCheck
        funcReturnCheck = true;
        Node node = parseBlock();
        funcReturnCheck = false;

//Stmt
//这种block是不必进行returnType检查的(g)
        //但是要新建符号表！！！
        if (curToken.getType() == TokenType.LBRACE) {
            boolean temp = funcReturnCheck;
            funcReturnCheck = false;
            //新建符号表
            tableStack.enterBlock();
            Node node = parseBlock();
            children.add(node);
            //离开block
            tableStack.leaveBlock();
            //恢复原先权限
            funcReturnCheck = temp;
        }
```

break continue 检查

这里笔者也是设置全局的一个参数 `forDepth` 记录当前的嵌套深度。

```
if (forDepth <= 0) {
                ErrorMessage errorMessage = new
ErrorMessage(curToken.getLine(),ErrorType.m);
                printer.addErrorMessage(errorMessage);
            }
```

2 中端——LLVM中间代码生成

主体思想是遍历语法树的每一个 `node` 调用其 `genIR` 方法，生成相应的 LLVM。

2.1 编码前设计

编码前设计与下文大体无异，总控程序与各指令对应的类。不同点在于：

- 编码前设计了 `LocalVar` 类，对后续生成 mips 造成不小的影响，最后虽然没删去，也是基本弃用了。
- 编码前未设计 `Constant` 类，想以 `name` 为常数的 `value` 来表示常数，为后续生成 mips 造成不少麻烦，在生成 mips 的时候还是加了 `Constant` 类。

2.2 IRBuilder

LLVM 生成的总控程序。

- 生成变量名、函数名

```
public String genGlobalVarName() {
        String name = GLOBAL_VAR_PREFIX + globalVarCnt;
        globalVarCnt++;
        return name;
    }
```

- 维护中间代码结构

```
//用于维持 IR 的结构
    //module
    public void addGlobalVar(GlobalVar globalVar) {
        curModule.addGlobalVal(globalVar);
    }
    public void addFunction(Function function) {
        curModule.addFunction(function);
        setCurFunction(function);
        lastInstr = null;
    }
    //function
    public void addBasicBlock(BasicBlock basicBlock) {
        curFunction.addBasicBlocks(basicBlock);
    }
    public void addParam(Param param) {
        curFunction.addParam(param);
    }
    //BasicBlock
    public void addInstr(Instr instr) {
        curBasicBlock.addInstr(instr);
        this.lastInstr = instr;
    }
```

- 短路求值中的 block 关系维护

```
public BasicBlock getLoopFollowBlock() {
        return loopFollowBlock;
    }

    public BasicBlock getLoopNextBlock() {
        return loopNextBlock;
    }

    public void setLoopNextBlock(BasicBlock loopNextBlock) {
        this.loopNextBlock = loopNextBlock;
    }
```

```
    public void setLoopFollowBlock(BasicBlock loopFollowBlock) {
        this.loopFollowBlock = loopFollowBlock;
    }
```

2.3 node.genIR()

语法树的每个节点都有吧 `genIR` 方法，用于生成 LLVM ，并调用 `IRBuilder.add` 将生成的 LLVM 加入相应位置。

2.4 数组处理

笔者的处理是在生成 LLVM 之前便将其全部压成一维，但是仍要保留其之前的各维度信息，以便后续调用。

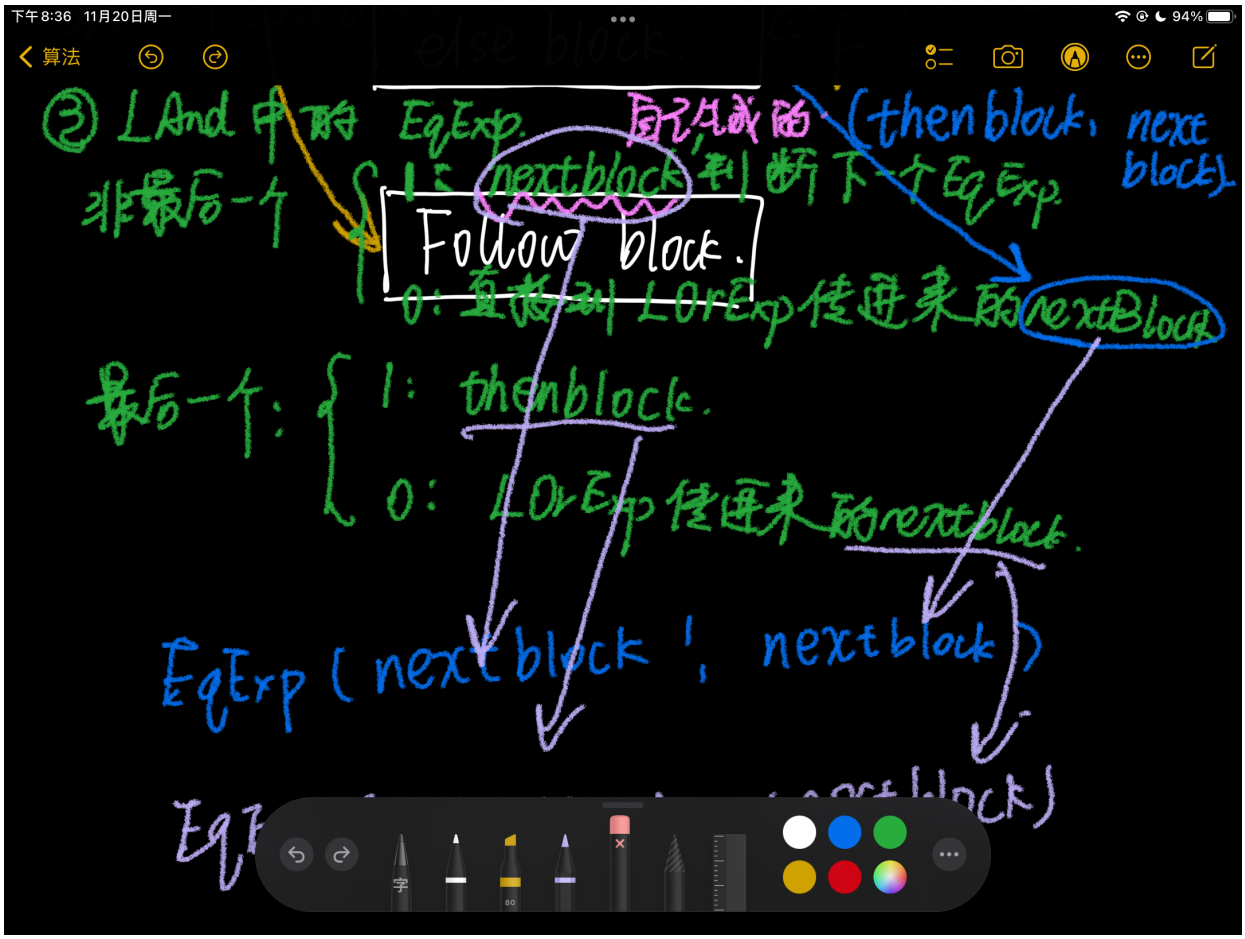
```
//ConstDef
int dim = 1;
int i;
ArrayList<Integer> initDims = new ArrayList<>();
for (i = 1; i < getChildren().size(); i++) {
    if (getChildren().get(i) instanceof TokenNode) {
          if (((TokenNode)getChildren().get(i)).getToken().getType() ==
TokenType.ASSIGN) {
                break;
             }
          }
          if (getChildren().get(i) instanceof ConstExp) {
                int subDim = ((ConstExp)getChildren().get(i)).calculate();
                initDims.add(subDim);
                dim = dim*subDim;
             }
          }
          i++;
          String targetType = "[" + String.valueOf(dim) + " x " + "i32]";


//LVal 调用
//现在各维度
ArrayList<Value> rParamDims  = new ArrayList<>();
for (Node node : getChildren()) {
     if (node instanceof Exp) {
         rParamDims.add(node.genIR());
         }
      }
      //得到压缩到一维的坐标
      int i,j;
      Value offset = new Value(LLVMType.INT32,Integer.toString(0));
      for (i = 0; i < rParamDims.size(); i++) {
           Value subOffset = rParamDims.get(i);
           for (j = i+1; j < dims.size(); j++) {
                Constant constant = new
Constant(LLVMType.INT32,dims.get(j).toString());
```

```
                MulInstruction mulInstr = new
MulInstruction(LLVMType.INSTR,IRBuilder.getInstance().genLocalVarName(),IRBuilder.ge
tInstance().getCurBasicBlock(),subOffset,constant);
                IRBuilder.getInstance().addInstr(mulInstr);
                subOffset = mulInstr;
            }
            AddInstruction addInstr = new
AddInstruction(LLVMType.INSTR,IRBuilder.getInstance().genLocalVarName(),IRBuilder.ge
tInstance().getCurBasicBlock(),offset,subOffset);
            IRBuilder.getInstance().addInstr(addInstr);
            offset = addInstr;
        }
```

2.5 短路求值

短路求值其实最重要的是梳理清楚各个 `Block` 之间的逻辑关系，这个我梳理了比较久。

以 `LAnd` 为例，先放草图吧：）



还需注意：最终结果的**位宽问题**。比如在笔者的设计中，规定 `EqExp.genIR()` 返回的一定是一个 `i1` 的 `value` 。

```
//只有一个RelExp，返回的可能是 i32 也可能是 i1，取决于 RelExp 中的 AddExp 数量
//如果是 i32 ，需要和 0 作比较
if (getChildren().size() == 1) {
    //只有一个AddExp 返回 i32
    if (getChildren().get(0).getChildren().size() == 1) {
       Value operand2 = new Constant(LLVMType.INT32,"0");
       IcmpInstruction icmpInstr = new IcmpInstruction(LLVMType.INSTR,
IRBuilder.getInstance().genLocalVarName(),basicBlock,operand1,operand2,IcmpOp.NE);
       IRBuilder.getInstance().addInstr(icmpInstr);
       operand1 = icmpInstr;
       //返回 i1 时无需多余处理
     }
     return operand1;
}
```

3 后端

3.1 Mips 生成

`mips` 是在 LLVM 的基础上生成的，因此对每个 LLVM 对象写一个 `toAssembly` 方法，然后遍历所有 LLVM 指令生成 mips 即可。

3.1.1 编码前设计

笔者认为基于 LLVM 生成 MIPS 其实是一个比较定式机械的过程。编码前架构设计与下文一致，一个总控程序，以及各 mips 指令对应的类。

其实在写 mips 生成之前，还是想直接分配寄存器的，但是后续了解到 Mem2Reg 优化，于是先全部存在内存中了，再后来就没时间写 Mem2Reg 优化了，这算一项未完成的设计吧。

3.1.2 MipsBuilder

生成 mips 的总控程序。

- **存储管理**
  - 内存管理

    `offsetMap` 负责管理 value 变量与 offset 的对应，其内存位置即为 `$sp + offset`。

    每次进入一个新的函数定义，都要新建一个表。

    ```
    public void enterFunc(Function function) {
            this.curFunc = function;
            this.curOffset = 0;
            this.offsetMap = new HashMap<>();
            this.var2Reg = function.getVar2Reg();
        }
    ```

  - 寄存器管理

由于最后没时间写寄存器分配了，所以这些接口都没用上，主要思想是建立一个 `var2Reg` 的对应表，记录各 value 与寄存器的对应关系。

- **后端代码结构管理**

  `addDataAsm()` 将指令加入数据段，`addTextAsm()` 将指令加入代码段。

3.1.3 AssemblyTable

- 分为数据段和代码段，存储所有的 mips 指令。只有 `GlobalAsm` 会加入数据段。
- 优化开关 `openPeepholeOptimize()` 和 `openMulDivOptimize()`。

3.1.4 Instr.toAssembly()

- `Instr.toAssembly()` 中注释掉了 `CommentAsm` 方便后续优化。

```
public void toAssembly() {
        //new CommentAsm("\n# " + this.toString());
 }
```

- 未启动寄存器分配，只用了 `$k0` `$k1` 两个寄存器，所有数据存在栈上。以下以 `AddExp` 为例展示计算与存储操作：

```
//直接在 new 的时候就加入了mipsBuilder
    public void toAssembly() {
        //注释
        super.toAssembly();
        //使用 k0 k1
        Register reg1 = Register.K0;
        Register reg2 = Register.K1;
        //TODO 先全部放在栈上
        Register targetReg =
MipsBuilder.getInstance().getRegByValue(this.getName());
        if (targetReg == null) {
                targetReg = Register.K0;
        }
        //先将operand1保存到k0
        //常数
        String op1Name = operand1.getName();
        int flag1 = 0;
        int i = 0;
        for (i = 0; i < op1Name.length(); i++) {
            if (!Character.isDigit(op1Name.charAt(i))) {
                flag1 = 1;
                break;
            }
        }
        String op2Name = operand2.getName();
        int flag2 = 0;
        for (i = 0; i < op2Name.length(); i++) {
            if (!Character.isDigit(op2Name.charAt(i))) {
                flag2 = 1;
```

```
                break;
            }
        }
        if (operand1 instanceof Constant || flag1 == 0) {
            new LiAsm(reg1,Integer.parseInt(operand1.getName()));
        }
        //如果已经有对应的reg
        else if (MipsBuilder.getInstance().getRegByValue(operand1.getName()) !=
null) {
            reg1 = MipsBuilder.getInstance().getRegByValue(operand1.getName());
        }
        //没有对应的add，在栈上寻找，没有则新开辟
        else {
            int offset1 =
MipsBuilder.getInstance().getOffsetOfValue(operand1.getName());
            //如果没有 offset 是MAXVALUE
            if (offset1 == Integer.MAX_VALUE) {
                MipsBuilder.getInstance().subCurOffset(4);
                offset1 = MipsBuilder.getInstance().getCurOffset();

 MipsBuilder.getInstance().addOffsetOfValue(operand1.getName(),offset1);
            }
            new MemAsm(MemAsm.Op.LW,reg1,Register.SP,offset1);
        }
        //operand2做相同操作
        //常数
        if (operand2 instanceof Constant || flag2 == 0) {
            new LiAsm(reg2,Integer.parseInt(operand2.getName()));
        }
        else if (MipsBuilder.getInstance().getRegByValue(operand2.getName()) !=
null) {
            reg2 = MipsBuilder.getInstance().getRegByValue(operand2.getName());
        }
        else {
            int offset2 =
MipsBuilder.getInstance().getOffsetOfValue(operand2.getName());
            if (offset2 == Integer.MAX_VALUE) {
                MipsBuilder.getInstance().subCurOffset(4);
                offset2 = MipsBuilder.getInstance().getCurOffset();

 MipsBuilder.getInstance().addOffsetOfValue(operand2.getName(),offset2);
            }
            new MemAsm(MemAsm.Op.LW,reg2,Register.SP,offset2);
        }
        //计算
        new AluAsm(AluAsm.Op.ADDU,targetReg,reg1,reg2);
        //如果target本身没有寄存器,存到栈上
        if (MipsBuilder.getInstance().getRegByValue(this.getName()) == null) {
            MipsBuilder.getInstance().subCurOffset(4);
            int curOffset = MipsBuilder.getInstance().getCurOffset();

 MipsBuilder.getInstance().addOffsetOfValue(this.getName(),curOffset);
```

```
            new MemAsm(MemAsm.Op.SW,targetReg,Register.SP,curOffset);
        }
    }
```

3.2 优化

由于确实期末时间紧，笔者只在后端做了简单的**窥孔优化**和**乘除法优化**，并写好了寄存器分配的部分接口，但考虑到后续时间不够完善和 debug，并未真正启动寄存器分配。

3.2.1 窥孔优化

- 优化连续对同一地址的 `sw` `lw`

```
public ArrayList<Assembly> delLwAfterSw(ArrayList<Assembly> textSegment) {
        ArrayList<Assembly> instructions = new ArrayList<>();
        int i;
        int length = textSegment.size();
        for (i = 0; i < length; i++) {
            Assembly cur = textSegment.get(i);
            if (i == length - 1) {
                instructions.add(cur);
                break;
            }
            Assembly next = textSegment.get(i + 1);
            if (cur instanceof MemAsm && next instanceof MemAsm &&
Objects.equals(((MemAsm) cur).getMemAddr(), ((MemAsm) next).getMemAddr())
                    && ((MemAsm) cur).getOp() == MemAsm.Op.SW && ((MemAsm)
next).getOp() == MemAsm.Op.LW) {
                i++;
                instructions.add(cur);
                if (((MemAsm) cur).getReg() != ((MemAsm) next).getReg()) {
                    MoveAsm moveAsm = new MoveAsm(((MemAsm) next).getReg(),
((MemAsm) cur).getReg());
                    instructions.add(moveAsm);
                }
            } else {
                instructions.add(cur);
            }
        }
        return instructions;
    }
```

- 优化对同一个寄存器的 `move`

```
public ArrayList<Assembly> moveSameDst(ArrayList<Assembly> textSegment) {
        ArrayList<Assembly> instructions = new ArrayList<>();
        int i;
```

```
        int length = textSegment.size();
        for (i = 0; i < length; i++) {
            Assembly cur = textSegment.get(i);
            if (cur instanceof MoveAsm && (((MoveAsm) cur).getDst() ==
((MoveAsm) cur).getSrc())) {
                //
            }
            else {
                instructions.add(cur);
            }
        }
        return instructions;
    }
```

- 优化连续两条像同一个寄存器 `move` （这里需要保证第二条指令的 src 不是第一条的 dst）

```
public ArrayList<Assembly> moveOverlap(ArrayList<Assembly> textSegment) {
        ArrayList<Assembly> instructions = new ArrayList<>();
        int i;
        int length = textSegment.size();
        for (i = 0; i < length; i++) {
            Assembly cur = textSegment.get(i);
            if (i == length - 1) {
                instructions.add(cur);
                break;
            }
            Assembly next = textSegment.get(i + 1);
            if (cur instanceof MoveAsm && next instanceof MoveAsm && (((MoveAsm)
cur).getDst() == ((MoveAsm) next).getDst()) && (!(((MoveAsm) next).getSrc() !=
((MoveAsm) cur).getDst()))) {
                //skip cur
            }
            else {
                instructions.add(cur);
            }
        }
        return instructions;
    }
```

- 优化加减 0 的 alu 指令
  - 加立即数 0

```
if (op == AluAsm.Op.ADDI) {
    if (((AluAsm) cur).getImm() == 0) {
       //如果rs与rd不相等
       if (((AluAsm) cur).getRs() != ((AluAsm) cur).getRd()) {
           MoveAsm moveAsm = new MoveAsm(((AluAsm) cur).getRd(),((AluAsm)
cur).getRs());
           instructions.add(moveAsm);
       }
       //如果相等 skip cur
    }
    else {
        instructions.add(cur);
    }
}
```

- 加减的寄存器在 alu 指令之前一条正好是 `li $r0 0`

```
if (((AluAsm) cur).getRt() == Register.ZERO ||
    (pre instanceof LiAsm && (((LiAsm) pre).getImm() == 0) && ((AluAsm)
cur).getRt() == ((LiAsm) pre).getRd())) {
    if (((AluAsm) cur).getRs() != ((AluAsm) cur).getRd()) {
        MoveAsm moveAsm = new MoveAsm(((AluAsm) cur).getRd(),((AluAsm)
cur).getRs());
        instructions.add(moveAsm);
    }
    // li r2,0也可以去掉
    if (pre instanceof LiAsm && (((LiAsm) pre).getImm() == 0) && ((AluAsm)
cur).getRt() == ((LiAsm) pre).getRd()) {
       instructions.remove(pre);
       }
}
```

3.2.2 乘除法优化

- 优化乘 2 的幂次

```
//2的幂次
int powerFlag = 0;
int j;
for (j = 1; j < 32; j++) {
    int power = (int) Math.pow(2,j);
    //imm = 2^j
    if (imm == power) {
        powerFlag = 1;
        ShiftAsm shiftAsm = new ShiftAsm(ShiftAsm.Op.SLL, ((MDAsm)
cur).getTarget(), ((MDAsm) cur).getRs(),j);
        instructions.add(shiftAsm);
        //skip li
        instructions.remove(pre);
```

```
            //skip mflo
            i++;
            break;
        }
    }
```

- 优化乘普通常数

```
//乘其他常数
if (powerFlag == 0) {
    if (imm == 0) {
        MoveAsm moveAsm = new MoveAsm(((MDAsm) cur).getTarget(),Register.ZERO);
        instructions.add(moveAsm);
        //skip mflo
        i++;
    }
    else if (imm == 1) {
        //skip li
        instructions.remove(pre);
        //skip cur & mflo
        i++;
    }
     //imm <= 5优化才有意义
    else if (imm <= 5) {
        int k;
        for (k = 0; k < imm-1; k++) {
            AluAsm aluAsm;
            if (((MDAsm) cur).getTarget() != ((MDAsm) cur).getRs()) {
                if (k == 0) {
                    aluAsm = new AluAsm(AluAsm.Op.ADDU,((MDAsm)
cur).getTarget(),((MDAsm) cur).getRs(),((MDAsm) cur).getRs());
                }
                else {
                    aluAsm = new AluAsm(AluAsm.Op.ADDU,((MDAsm)
cur).getTarget(),((MDAsm) cur).getTarget(),((MDAsm) cur).getRs());
                }
            }
            else {
                 if (k == 0) {
                    aluAsm = new AluAsm(AluAsm.Op.ADDU,Register.T2,((MDAsm)
cur).getRs(),((MDAsm) cur).getRs());
                }
                else if (k < imm - 2){
                    aluAsm = new AluAsm(AluAsm.Op.ADDU,Register.T2,Register.T2,
((MDAsm) cur).getRs());
                }
                else {
                    aluAsm = new AluAsm(AluAsm.Op.ADDU,((MDAsm)
cur).getTarget(),Register.T2,((MDAsm) cur).getRs());
                }
```

```
            }
            instructions.add(aluAsm);
        }
        //skip li
        instructions.remove(pre);
        //skip mflo
        i++;
```

- 优化除以 2 的幂次

```
//2的幂次
int powerFlag = 0;
int j;
for (j = 1; j < 32; j++) {
     int power = (int) Math.pow(2,j);
     //imm = 2^j
     if (imm == power) {
         powerFlag = 1;
         ShiftAsm shiftAsm = new ShiftAsm(ShiftAsm.Op.SRL, ((MDAsm)
cur).getTarget(), ((MDAsm) cur).getRs(),j);
         instructions.add(shiftAsm);
         //skip li
         instructions.remove(pre);
         //skip mflo
         i++;
         break;
         }
 }
```