编译技术设计文档

1. 参考编译器介绍

1.1 LLVM

LLVM编译器是一个功能强大的开源编译器基础设施项目，提供了一套模块化和可重用的编译器和工具链技术。

LLVM的架构可以分为三个主要部分：前端、中间表示（IR）和后端。前端负责将源代码转换为LLVM的中间表示。中间表示（IR）用于优化和代码生成，是LLVM的核心。后端将IR转换为目标机器码。LLVM的后端支持多种架构，如X86、ARM、PowerPC等。

本编译器借鉴了LLVM的前端、中端、后端设计，并主要在中端和后端做了优化。

1.2 GCC编译器

GCC编译器，全称GNU Compiler Collection（GNU编译器集合），是一个由GNU组织开发的、功能强大的开源编译器。

本编译器学习了GCC编译器的模块化设计，错误检查，以及函数内联、循环展开、常量传播等优化思路。

1.3 往届学姐学长的博客

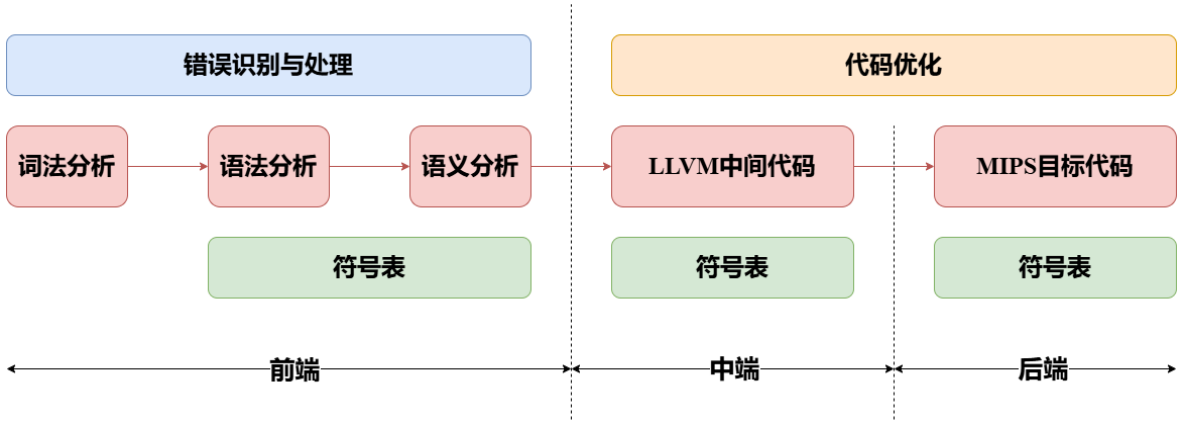
BUAA编译原理实验设计文档 | ZJYの颓圮日记（参考了符号表管理和技术文档撰写）

GitHub - KouweiLee/BUAA-2022-SysYCompiler: 2022秋季学期-北航计院-编译原理实验课设（参考了前端、中端、后端的架构）

GitHub - saltyfishyjk/BUAA-Compiler: 北航编译技术课程设计代码（2022）（参考了LLVM变量、函数、标签命名方式）

2. 编译器总体设计

本编译器按照实验作业要求，一步步实现了词法分析、语法分析、语义分析、中间代码生成以及目标代码生成。其中，选择LLVM作为中间代码，选择MIPS作为目标代码。



宏观上可以将以上五个阶段分为前端、中端、后端三部分。前端部分同步进行错误识别与处理，三部分每一部分都新建了符号表，存储各自关注的信息。

以下是对各阶段的详细设计描述：

五个阶段	主要任务
词法分析	将输入的源代码转换为词法单元（tokens）序列。需要识别源代码中的各类合法标记，涵盖关键字、标识符、常量、运算符以及分隔符等元素。同时，识别并纠正a型错误。
语法分析	通过对输入的词法单元序列进行解析，依据文法规则，解析语法成分，并自顶向下构建语法树。同时也要识别并纠正各种错误情况。
语义分析	给出具体变量的作用域序号、单词的字符 / 字符串形式以及类型名称。同时，对于存在错误的源程序，输出错误所在的行号和错误的类别码。
中间代码生成	我选择了LLVM IR 作为中间代码。通过利用前端生成的语法树，按照 LLVM 语法结构的粒度由高到低进行逐步解析，将语法树的 compUnit 结点转换为以 Module 为首的中间代码语法树，得到符合要求的中间代码。
目标代码生成	翻译 LLVM 中间代码，进一步生成 mips 目标代码

此外，为了提高目标代码的质量，在中间代码、目标代码生成阶段还进行了一些优化设计。

3. 词法分析设计

词法分析器主要是将源代码转换为词法单元（tokens），需要识别源代码中的合法标记，并处理可能存在的错误输入。

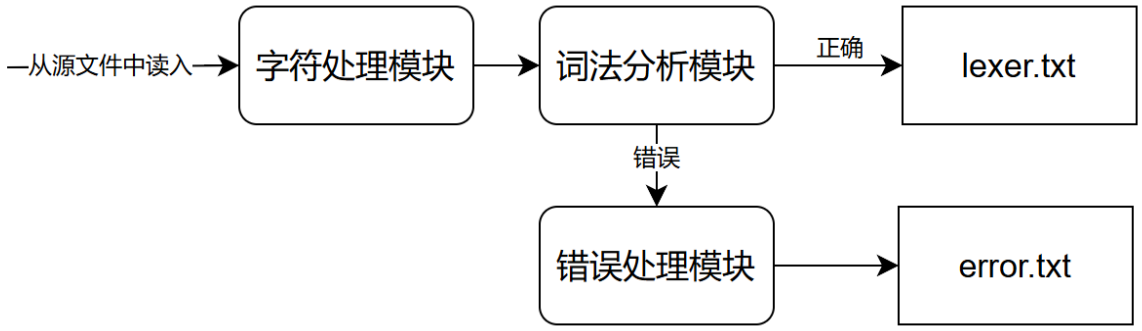
3.1 编码之前的设计

设计目标

- **支持多种输入结构**：识别单字符、双字符运算符，处理引号内的字符串、字符常量和注释。
- **处理正确的输入**：从源代码中提取并分类各种元素，形成预定义的标记类型。
- **错误处理**：能够检测非法标记，并提供详细的错误报告（包括错误所在的行号和出错类型）。

3.1.1 模块设计

为了实现上述目标，我主要设计了以下四个模块：



- **字符处理模块（`ProcessInput类`）**：负责从源文件中逐字符读取数据，初步筛选出可能构成标记的字符。
- **词法分析模块（`Token类`）**：运用预设的正则表达式，对字符流进行模式匹配，以确定具体的标记类型。
- **错误处理模块**：对不符合语言规范的字符序列，记录错误信息，包括错误类型和位置。
- **输出模块（`Lexer类`）**：将分析结果以文件的形式输出，包含合法的词法单元列表或错误日志。

3.1.2 设计细节

- **输入处理**：采用 `InputStream` 实现逐字符读取，特别关注如何有效跳过空白字符、换行符及各类注释，确保它们不会干扰正常的标记识别过程。
- **标记匹配**：利用正则表达式定义各标记类型的匹配规则，通过枚举 `Token.tokenType` 组织标记类别，借助 `Token.patterns` 存储具体模式。
- **错误处理**：识别并处理不符合规则的字符或词组，同时记录错误所在的行号和错误字符。
- **输出**：两种输出模式——正常流程下的词法单元输出和异常情况下的错误报告输出。

3.2 编码完成之后的修改

在实际编码过程中，我遇到了一些预期之外的问题，并据此对原始设计进行调整：

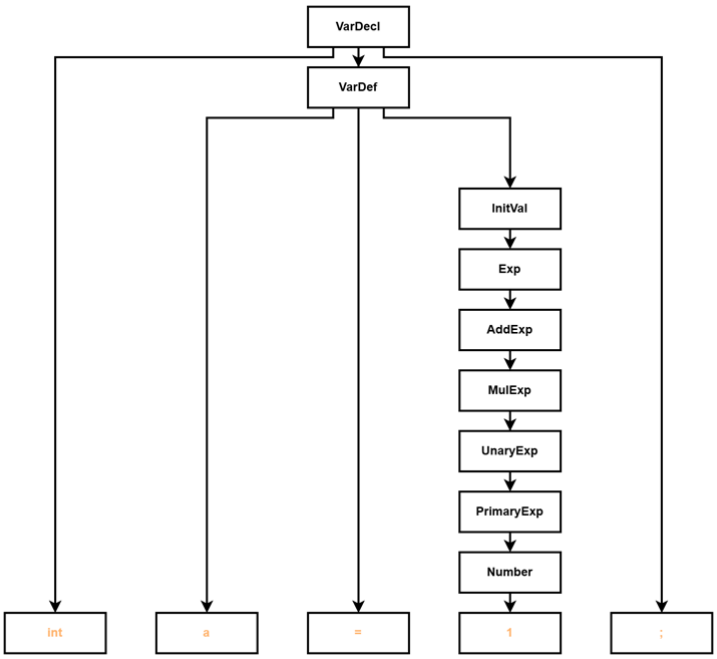
- **正则表达式的修改**：最初的设计中，字符常量的正则表达式较为简单，无法识别转义字符，导致对于`'\''`、`'\0'`无法正确识别，因此在 `Token` 类中修改了 `CHRCON` 的正则表达式，确保支持合法的转义字符。
- **完善注释处理逻辑**：原本没有处理多行注释，因此在 `ProcessInput` 类中增加了对 `/* */` 多行注释的识别逻辑。同时，保证单行注释 `//` 和多行注释均能跳过正确的字符而不被解析为词法单元。
  - 增加了对单行注释和多行注释的判断，并分别处理不同的注释格式。
- **调整正则匹配顺序**：调整了某些标记类型的匹配优先级，例如将 `patterns.put(tokenType.IDENFR, "[a-zA-Z_][a-zA-Z0-9_]` 置于最后；而将较长的操作符（如 `>=`、`<=`）的识别放在短操作符（`>`、`<`）之前，以便能正确匹配。
- **代码结构优化**：将词法分析过程中频繁使用的函数如 `matchWord()` 和 `matchToken()` 进行了代码复用的优化，减少重复代码，提高可读性。
- **注释嵌套问题**：多行注释中可能嵌套其他代码或者注释，因此在注释识别时使用了逐字符读取的方式，确保所有字符都能正确识别。

4. 语法分析设计

4.1 编码之前的设计

助教关于语法分析的讲座中，有这样一个例子：

对于`int a = 1;`，结合语法定义`变量声明 VarDecl → BType VarDef { ',' VarDef } ';'`和`变量定义 VarDef → Ident [ '[' ConstExp ']' ] | Ident [ '[' ConstExp ']' ] '=' InitVal`，可构建语法树解析如下：



因此，**语法分析实际上是一个自顶向下解析词法直至终结符为止的过程。**

本次作业的核心任务：

- 自顶向下构建语法树
- 处理语法分析中的错误情况

4.1.1 预备工具

由于构建语法树的过程中需要多次获取下一个token，或者回溯到前一个或多个token的位置，或者检查是否匹配成功，因此我们可以将该部分操作进行封装：

`public void next()`：获取下一个token，将索引值、词法类型、行号都更新为下一个token的对应信息。

`public void back(int savedPos)`：将当前token信息重置为指定索引`savedPos`对应的token。

`public boolean match(Token.tokenType type)`：检查当前token的类型是否与参数`type`相同，若匹配成功，则将对应信息`单词类别码 单词的字符/字符串形式`加入`outputList`中，以便最终输出到`parser.txt`；若匹配失败，则返回失败。

4.1.2 构建语法树

1. 创建语法树叶子结点，用于记录语法树的具体信息

   对`CompUnit`,`Decl`,`ConstDecl`,`BType`,`ConstDef`等每一种语法成分创建专门的类，在构建语法树的过程中，记录每一个叶子结点的语法信息。

2. 构建识别方法，**判断即将解析的语句具体属于哪种语法。**

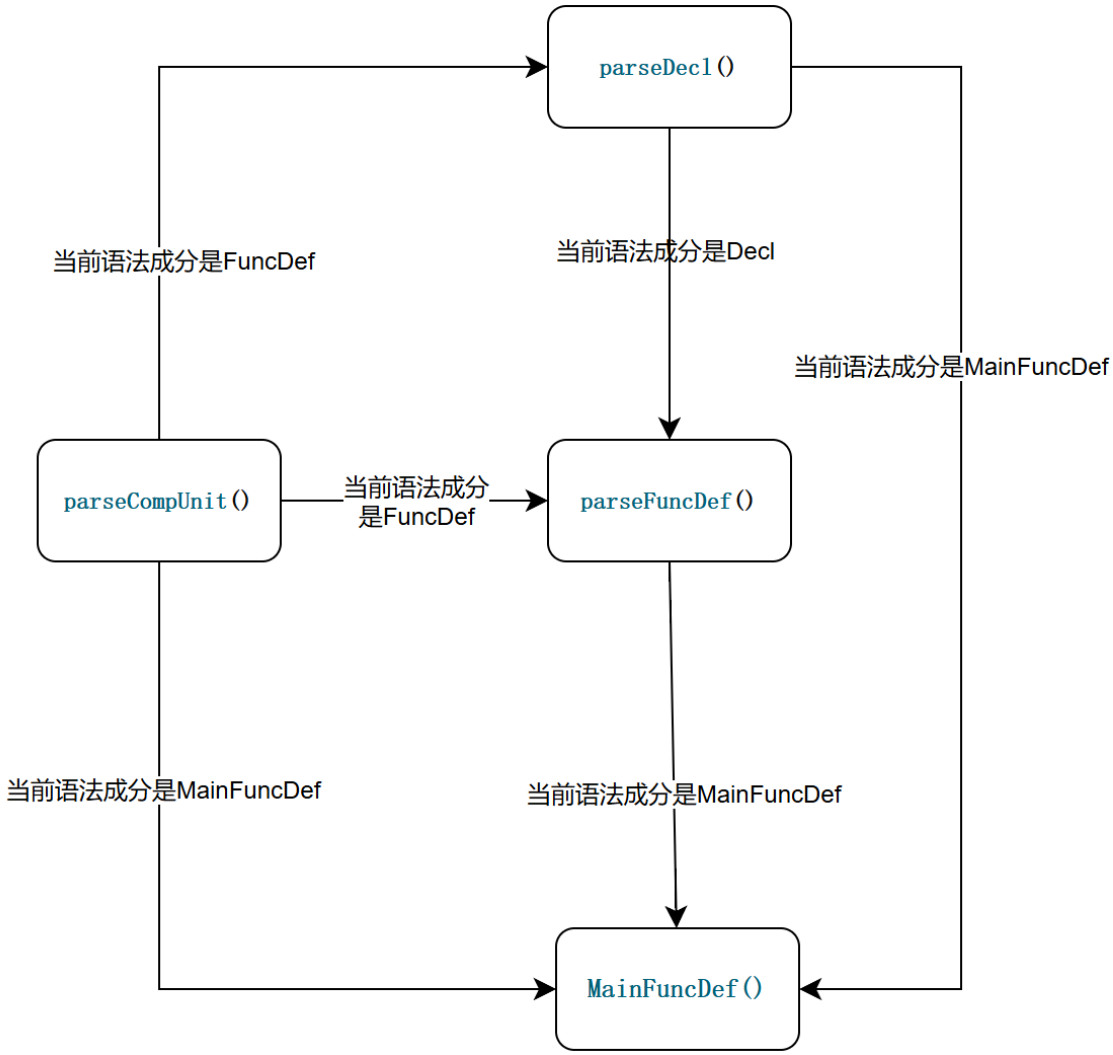
   例如：

```
// 声明 Decl → ConstDecl | VarDecl
// 常量声明 ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';'
// 变量声明 VarDecl → BType VarDef { ',' VarDef } ';'

if 当前语法为 声明 Decl：
    if currentToken == "const":
        即将解析的语法为 常量声明 ConstDecl
    else:
        即将解析的语法为 变量声明 VarDecl
```

3. 构建语法解析方法`parseXXX`(`XXX`表示某种语法成分)，**对每一种语法成分进行解析。**

   例如，对于`编译单元 CompUnit → {Decl} {FuncDef} MainFuncDef`，构造解析方法`parseCompUnit()`：



实现细节

- `符号{...}表示花括号内包含的为可重复 0 次或多次的项`

  对于`{ }`包裹的语法成分，第一步通过识别方法，确定当前语法成分是否为`{}`包裹的语法成分，若是则读取后解析，并回到第一步，否则跳过`{}`包裹的语法成分继续向后匹配。

- `符号[...]表示方括号内包含的为可选项`

  对于`[ ]`包裹的语法成分，通过识别方法，确定当前语法成分是否为`[ ]`包裹的语法成分，若是则读取后解析，然后继续向后匹配，否则跳过`{}`包裹的语法成分继续向后匹配。

4.1.3 错误处理

词法分析错误处理

由词法分析部分知，词法分析中只会检测a类错误（将`&&`写作`&`，或将`||`写作`|`）。为使该种错误不影响语法分析，因此在词法分析中做特殊处理，记录错误类型和行号，但自动纠正错误token（读到`&`仍记作`&&`，或读到`|`仍记作`||`），以便语法分析正常进行。

语法分析错误处理

该部分会出现以下错误：

错误类型	错误类别码	解释
缺少分号	i	报错行号为分号前一个非终结符所在行号。
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。
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。

在解析涉及以上错误类型的语法成分时进行检查，如果缺少以上成分，则记录行号和错误类型，并从当前token继续解析。

错误处理合并

按照行号由小到大顺序将词法分析错误和语法分析错误进行合并，并输出至`error.txt`。

4.2 编码完成之后的修改

消除左递归

在实际编码过程中，我遇到的最大的问题是关于以下文法的解析：

```
乘除模表达式 MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
加减表达式 AddExp → MulExp | AddExp ('+' | '−') MulExp
关系表达式 RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
相等性表达式 EqExp → RelExp | EqExp ('==' | '!=') RelExp
逻辑与表达式 LAndExp → EqExp | LAndExp '&&' EqExp
逻辑或表达式 LOrExp → LAndExp | LOrExp '||' LAndExp
```

可以看出，这些文法都是左递归的，我们无法使用普通的递归下降子程序对其处理，因此需要消除左递归。这里用加减表达式举例，`AddExp`一定由`MulExp`开头，后无成分或跟一个或多个`('+' | '−') MulExp`：

```
加减表达式 AddExp → MulExp | AddExp ('+' | '−') MulExp
可写作
    AddExp → MulExp AddExp'
    AddExp' → ('+' | '−') MulExp AddExp' | ϵ
```

改写后的文法可以使用递归下降进行处理。

注意：改写后，输出信息方式略有不同，需要根据开头`MulExp`后是否还有其他成分来决定输出内容。

错误处理时当前索引值的指向

一方面，我的实现会在每一个`parsexxx`完成后索引值指向该语法成分的最后一个token，而在更高一级的`parsexxx`调用低级`parsexxx`后挪动索引值，指向下一个token。

另一方面，我的实现会在解析一个语法成分之前，先通过特判查看下一个语法成分是什么，因此在查看完后，索引值要回溯到查看前的位置。

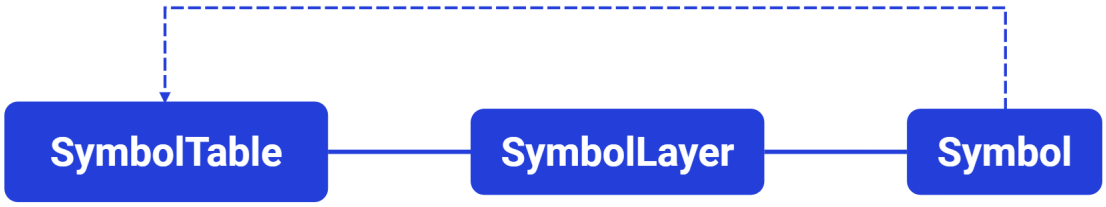
鉴于以上两方面原因，错误处理时我的索引值总会指向本该为缺失成分的token，实际上我应该指向前一个非终结符。因此要再回溯一下索引值。

5. 语义分析设计

语义分析部分将从给定的源程序中识别出定义的常量、变量、函数、形参等，并输出它们的作用域序号、单词的字符/字符串形式和类型名称。同时，对于错误的源程序，程序要能够识别出错误，输出错误所在的行号和错误的类别码。

5.1 编码之前的设计

5.1.1 符号表设计



Symbol：符号层级，存储具体符号。该部分我参考了“语义分析讲座”ppt中的内容——记录具体符号的作用域序号、符号名称、类型名称、符号类型（常量、变量、函数、形参）等。

SymbolLayer：作用域层级，存储某一作用域的信息。存储该作用域的序号、**父级**`Symbol`、所含符号。（**加粗为编码后修改**）

SymbolTable：符号表层级，存储整个程序的所有符号和作用域。以`ArrayList<Symbol>`形式存储所有的符号`Symbol`，以`Stack<SymbolLayer>`栈式存储所有的作用域`SymbolLayer`。

5.1.2 符号表构建

符号的加入

1. 识别符号：当解析器遇到一个标识符（Ident），就会调用语义分析器来识别这个标识符代表的符号类型（常量、变量、函数等）。
2. 创建符号：创建`Symbol`对象，填充其属性，包括符号名称、类型名称、符号类型等。
3. 确定作用域：获取`Stack<SymbolLayer>`栈顶部的`SymbolLayer`，新符号与其作用域相同。
4. 加入符号表和作用域：将`Symbol`对象加入到栈顶部的`SymbolLayer`中；将`Symbol`对象添加到全局的符号列表`ArrayList<Symbol>`中。

作用域的管理

1. 进入新作用域：当解析器遇到一个新的作用域（函数定义或代码块的开始），创建一个新的`SymbolLayer`对象，向其填入作用域序号属性，并将其推入作用域栈中。
2. 退出作用域：当解析器离开一个作用域（函数定义或代码块的结束），从作用域栈中弹出最顶层的`SymbolLayer`对象。
3. 作用域序号管理：全局作用域的序号为1，每进入一个新的局部作用域，序号增加1。

5.1.3 错误管理

单独建立`Operation`类进行错误处理、与词法\语法分析错误进行合并、排序、输出。

针对具体错误，做了如下处理：（**加粗为编码后修改**）

错误类别	我的实现
b	在作用域栈顶查找同名符号
c	在作用域栈中查找同名符号、**在全局符号表中查找const类型同名符号**
d	建立`FuncList`类存储函数表，记录函数形参个数和类型。比较形参个数和实参个数
e	建立`FuncList`类存储函数表，记录函数形参个数和类型。**比较形参类型和实参类型**
f	设置`voidFuncFlag`标识，记录当前函数类型，若为`void`函数且出现`exp`非空的`return`，则报错
g	函数解析至`block`结束时检查`block`中最后一个`blockItem`是否为`return`
h	对于`LVal`，在全局符号表中查找`const`类型同名符号
l	比较`%d`、`%c`**个数**和`printf`中`exps`个数
m	设置循环块`loopFlag`标识，**进入循环块后+1，退出时-1。读到**`break`**或**`continue`**时，若标识不为0，则报错**

5.2 编码完成之后的修改

1. c 类错误 “使用了未定义的标识符”，编码前未考虑作用域栈外定义的`const`常量。
2. e 类错误
   - 需要识别函数名后的参数，因此在`SymbolLayer`和`Symbol`中记录父级`Symbol`，便于查找；
   - 对形如`s[0]`的类型转换（数组转化为`Int`或`Char`）；
   - `Int`和`Char`之间可互相转化；
   - 未注意文档中 “普通常量可以作为函数参数，但是常量数组不可以”；
3. f 类错误：void返回值的函数需要考虑分支语句的return

   形如：

```
void my_print(int param) {
    if (1 == 0) {
        int a = 1;
        if (2 == 1) {
            return 1;
        }
        return (1 + 10 ) *10;
    }
    else
        return 20;

    return (param + 12) * 10;
}
int main() {
    return 0;
```

```
}
```

应对所有非空`return`报错：

```
5 f
7 f
10 f
12 f
```

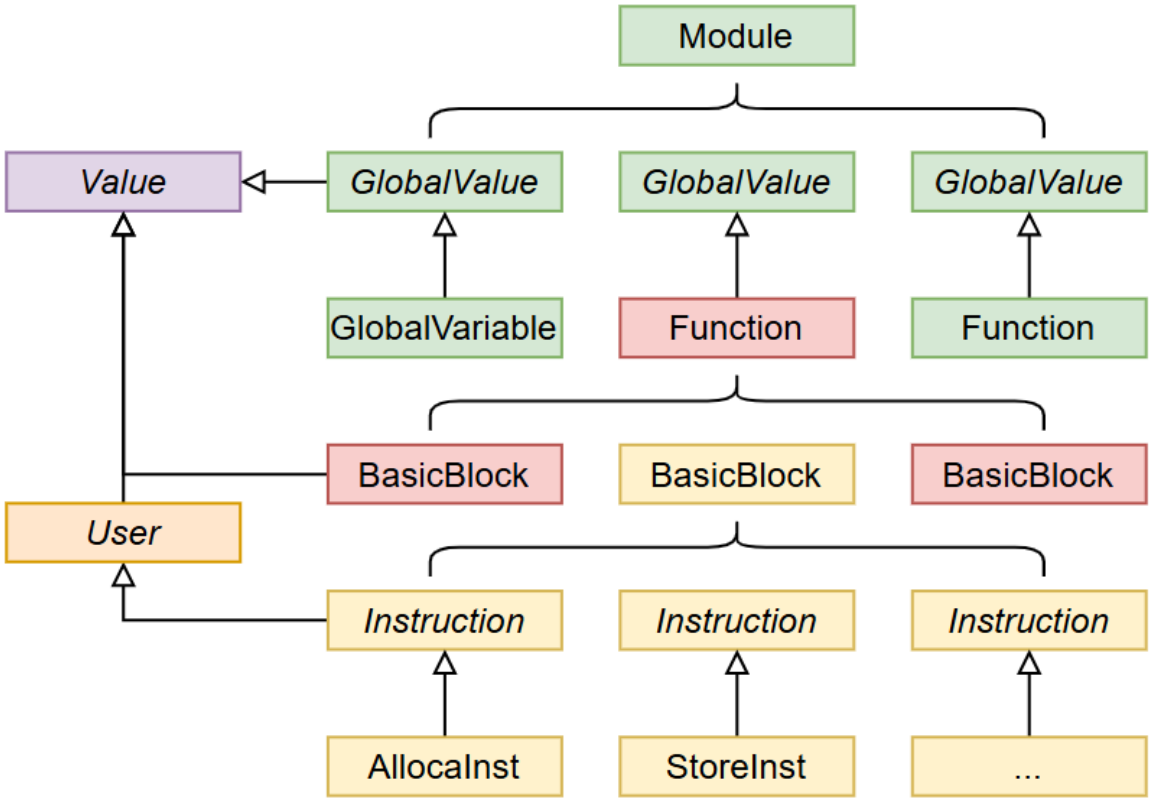
4. l 类错误：未注意 "格式字符只包含 %d 与 %c，其他 C 语言中的格式字符，如 %f 都当做普通字符原样输出"。起初错误的以`%`个数计算。
5. m 类错误：未考虑循环块嵌套问题，起初将`loopFlag`设置为`boolean`类型，进入循环块为`true`，退出为`false`，并且错误地将`if-else`作为循环块（小丑行为）。

6. 中间代码生成

我选择了 LLVM IR 作为中间代码。

6.1 编码之前的设计

按照教程，LLVM 语法结构由粒度从高到低划分为整个模块`Module`、函数`FUnction`、基本代码块`BasicBlock`、指令`Instruction`，变量/常量和符号。所有语法结构都继承`Value`。因此，可以利用前端生成的语法树，对语法树的`compUnit`结点按照该语法结构粒度由高到低解析，得到`Module`统领的中间代码语法树，以`Instruction`基本结构输出，即可得到中间代码。



6.1.1 构建符号表

由于前端部分的符号表随着语法树的生成而先建立后消亡（入栈/出栈），因此，在中端设计中我重新构建了符号表。内容基本与前端符号表相似，区别在于对每个`LLVMSymbol`，如果为标量，存储其初始值，如果为数组，存储数组大小和其初始化的值。

```
public class LLVMSymbolTable {  // 定义一个用于管理LLVM符号的符号表类

    private ArrayList<LLVMSymbol> LLVMSymbolList;   // 存储当前符号表中的所有符号

    private LLVMSymbolTable father; // 父符号表，用于支持作用域嵌套查找

    public LLVMSymbolTable(LLVMSymbolTable father) {    // 初始化符号表并设置父符号表
        this.LLVMSymbolList = new ArrayList<>(); // 初始化符号列表
        this.father = father; // 关联父符号表
    }

    public void addSymbol(LLVMSymbol LLVMSymbol) {
LLVMSymbolList.add(LLVMSymbol);}

    public LLVMSymbol search(String ident) {     // 根据标识符（ident）搜索符号
        LLVMSymbolTable LLVMSymbolTable = this;
        while (LLVMSymbolTable != null) {
            for (LLVMSymbol LLVMSymbol : LLVMSymbolTable.LLVMSymbolList) {  // 遍
历当前符号表的所有符号
                if (Objects.equals(LLVMSymbol.name, ident)) {   // 名称匹配的符号
                    return LLVMSymbol;
                }
            }
            LLVMSymbolTable = LLVMSymbolTable.father;   // 当前符号表中未找到，查找父
符号表
        }
        return null;
    }
}
```

```
public class LLVMSymbol {

    private String name;    // 当前单词所对应的字符串。
    private String irName;   // 地址名
    private IrType irType;   // symbol实值对应的irType

    private Integer type;        // 0 -> var, 1 -> array, 2 -> func
    private Integer bType;    // 0 -> int, 1 -> char, -1 -> void
    private Integer con;        // 0 -> const, 1 -> var

    // 对于普通变量/常量
    private Integer value;

    // 对于数组
    private IrType valueIrType;
    private Integer size;
    private ArrayList<Integer> arrayValue = new ArrayList<>();
    private boolean ifArrayValueZero;

    // 构建普通常量/变量symbol
    public LLVMSymbol(String name, String irName, IrType irType, Integer type,
Integer bType, Integer con) {
        ...
    }

    // 构建数组symbol
```

```
    public LLVMSymbol(String name, String irName, IrType irType, Integer type,
Integer bType, Integer con, Integer size) {
        ...
    }

    // 获取symbol维度
    public int getDimension() {return type;}

    // 设置普通常量/变量的值
    public void setValue(Integer value) {this.value = value;}

    public void setArrayValue(Integer value) {  // 设置数组的值
        this.ifArrayValueZero = false;
        this.arrayValue.add(value);
    }
}
```

6.1.2 定义参数类型

参数类型主要分为三大类：

```
public enum TypeID {
    VoidTyID,       //空返回值
    IntegerTyID,    //整数类型
    PointerTyID     //指针类型
}
```

- `void`函数的返回值设置为`VoidTyID`类型；
- `int`或`char`型函数的返回值、标量的值为`IntegerTyID`类型；
- 数组的首地址定义为`PointerTyID`类型；
- 对于`IntegerTyID`和`PointerTyID`类型，同时设置`num`表明其类型为`i32`、`i8`还是`i1`。

6.1.3 LLVM label命名

我采取了字符串命名的方式对变量以及基本块进行命名，并建立`Count`类对标号进行更新：

对于全局常量/全局变量：

```
String name = "@GlobalVariable_" + Count.getGlobalVariableCount();
```

对于普通常量/变量：

```
String name = "%LocalVariable_" + Count.getFuncInner();
```

对于函数名：

```
String name = "func_" + Count.getFuncCount();
```

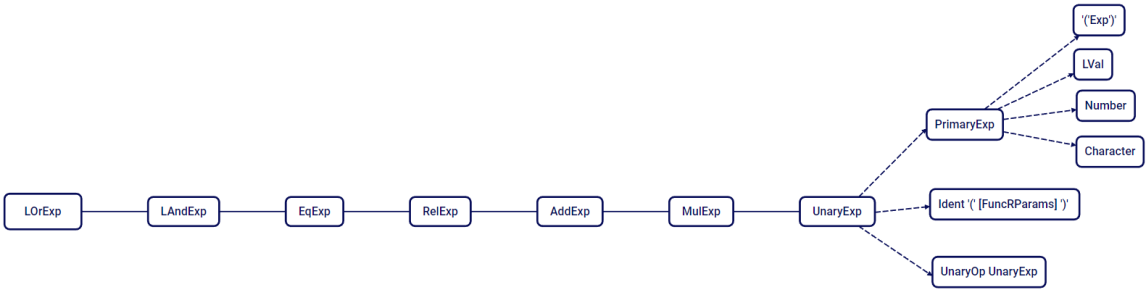
对于`If-Else`块：

```
String ifLabel = "%if_" + Count.getIfLabel();
String elseLabel = "%else_" + Count.getElseLabel();
String endLabel = "%ifElseEnd_" + Count.getIfElseEndLabel();
```

对于`For`循环语句：

```
String cond_name = "%ForCond_" + Count.getForCondLabel();
String forStmt2_name = "%ForStmt2_" + Count.getForStmt2Label();
String mainStmt_name = "%mainStmt_" + Count.getForMainStmtLabel();
String forEnd_name = "%ForEnd_" + Count.getForEndLabel();
```

6.1.4 Exp求值



6.1.4.1 判断能否直接得出值

对表达式递归，直至得到`PrimaryExp`或`Ident'('FuncRParams')'`：

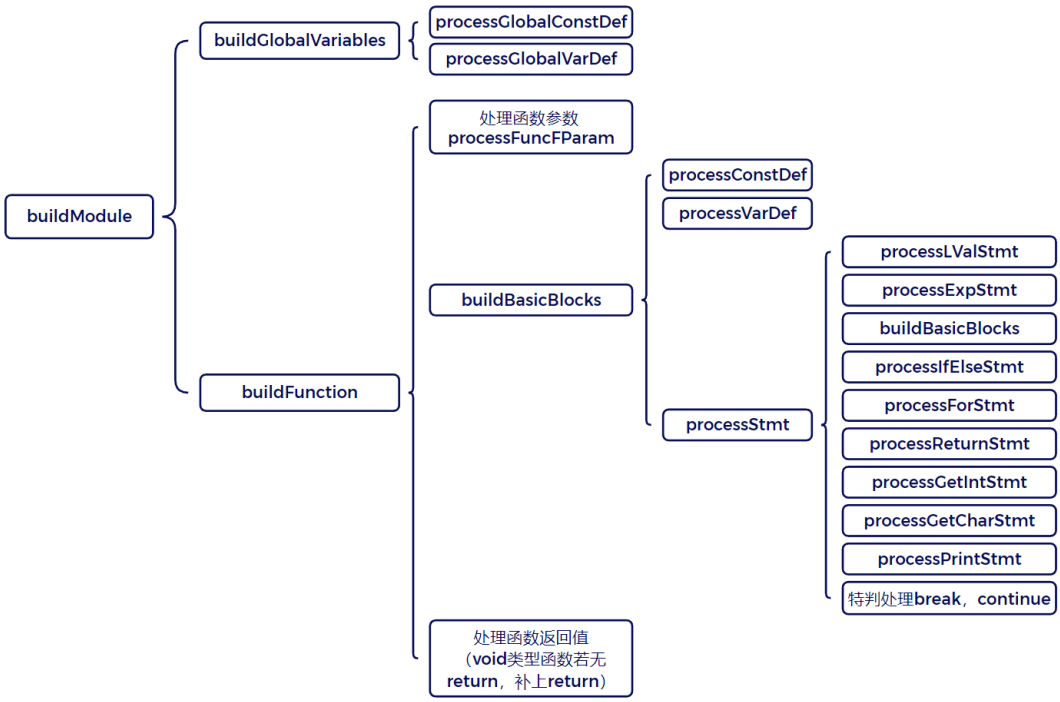
- 如果递归终点为`PrimaryExp`，且`PrimaryExp`为`LVal`，且`LVal`对应的symbol是常量，则可直接得出值；
- 如果递归终点为`PrimaryExp`，且`PrimaryExp`为`Number`，则可直接得出值；
- 如果递归终点为`PrimaryExp`，且`PrimaryExp`为`Character`，则可直接得出值。

6.1.4.2 动态计算

如果表达式不能直接得出值，则需要用计算指令动态计算。

- 递归终点：对于`LVal`，从symbol地址中加载实值；对于`Ident'('FuncRParams')'`，加载实参，调用函数。
- 中间表达式计算：中间过程中的表达式利用计算指令进行计算，需要注意类型转化（计算过程中将`char`类型转换为`int`类型进行计算）。

6.1.5 构建LLVM基本语法结构



上图为我构建LLVM基本语法结构的基本架构。以下详细说明每一部分的实现：**（其中加粗部分为编码后的修改）**

6.1.5.1 buildModule

处理全局常量/全局变量，处理函数、`main`函数，创建`new Module(globalVariables, functions, main)`。

6.1.5.2 buildGlobalVariables

分为全局变量和全局常量两种，分别使用`processGlobalVarDef`和`processGlobalConstDef`处理，得到`ArrayList<GlobalVariable>`。

- `processGlobalConstDef`
    1. 符号初始化：通过标识符（`ident`）和维度（`dimension`），生成LLVM全局变量的唯一名称（如`@GlobalVariable_1`）并创建对应的符号对象（`LLVMSymbol`）。若维度为0，则该常量是标量；若维度不为0，则该常量是数组。
    2. 类型与初始值处理：
        - 根据数据类型（如`int`或`char`）选择合适的LLVM类型（`IrType`）：`int`类型对应`i32`，`char`类型对应`i8`。
        - 如果是标量，计算初始值（通过符号表解析`ConstExp`表达式的值）并为符号对象赋值。
        - 如果是数组：
            - 若初始值是多个表达式，依次计算每个表达式的值并填充到数组中。
            - 若初始值是字符串，则按字符逐个解析，**处理转义字符**并填充数组。
            - 未提供初始值的元素以`0`填充。
    3. 符号表更新：将生成的符号对象加入符号表（`LLVMSymbolTable`）。
    4. 全局变量创建：将符号对象封装为`GlobalVariable`对象，供`Module`使用。
- `processGlobalVarDef`
    1. 符号初始化（同`processGlobalConstDef`）。
    2. 类型与初始值处理（若提供了初始值则同`processGlobalConstDef`处理，若没有提供，则全部赋值为0）

3. 符号表更新（同`processGlobalConstDef`）。
4. 全局变量创建（同`processGlobalConstDef`）。

6.1.5.3 buildFunction

根据`FuncDef`对象提取返回类型（`VoidTyID`/`IntegerTyID`）和函数名，解析完成后，将函数信息存储到符号表中。创建新的符号表，供函数内部使用。

- 处理函数参数`processFuncFParam`：如果函数定义中包含参数列表（`FuncParams`），则对每个参数调用`processFuncFParam`方法，将其转化为 LLVM 的`Argument`对象，并将参数存储到旧符号表中：

  ```
  for (FuncFParam funcFParam : funcDef.getFuncParams().getFuncFParams()) {
      argumentTIES.add(processFuncFParam(funcFParam, newTable, count));
  }
  ```

  `processFuncFParam`根据参数的维度（标量或数组）和类型（`int`或`char`），为每个参数分配唯一名称和对应的`IrType`，并将其记录到符号表中。

  同时，**存储形参的值，以备后续使用。**

- `buildBasicBlocks`（见6.1.5.4）
- **处理函数返回值**：创建`private static boolean funcReturn = false;`，用以记录当前函数是否出现`return`语句，在`buildBasicBlocks`的过程中更新该值，若该函数结束该值仍为`False`，则在输出时在函数末尾添上`return;`对应的LLVM语句。

6.1.5.4 buildBasicBlocks

根据文法定义，语句块 (Block) 内部可以包含零个或多个语句块项 (BlockItem)。语句块项进一步细分为`Decl`和`Stmt`。因此在`buildBasicBlocks`中，语句块的处理分为三种情况：

- `processConstDef`

  根据类型（`int`或`char`），为常量分配`IrType`。生成函数内变量的唯一名称并创建对应的符号对象。

  1. 标量常量的处理：从`ConstDef`中提取初始值，计算其表达式值；生成`AllocaInst`指令分配内存；使用`StoreInst`将计算的初值存储到分配的内存中。
  2. 数组常量的处理：确定数组的大小，通过`ConstExp`的计算结果获得维度信息，生成含维度的`AllocaInst`指令分配内存。如果数组有初始化值，则逐个计算初始值，否则赋0，并使用`StoreInst`指令存储到内存中。特别地，对字符数组类型，支持字符串字面量初始化，**处理转义字符。**

- `processVarDef`

  根据类型（`int`或`char`），为常量分配`IrType`。生成函数内变量的唯一名称并创建对应的符号对象。

  若标量变量/数组变量有初值，则处理同`processConstDef`，否则，仅分配内存。

- `processStmt`（见6.1.5.5）

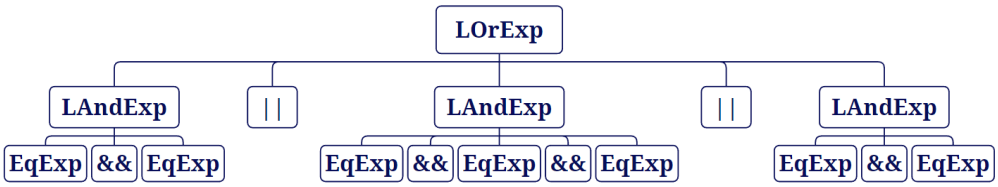
最终返回指令集合`ArratList<Instruction>`。

6.1.5.5 processStmt

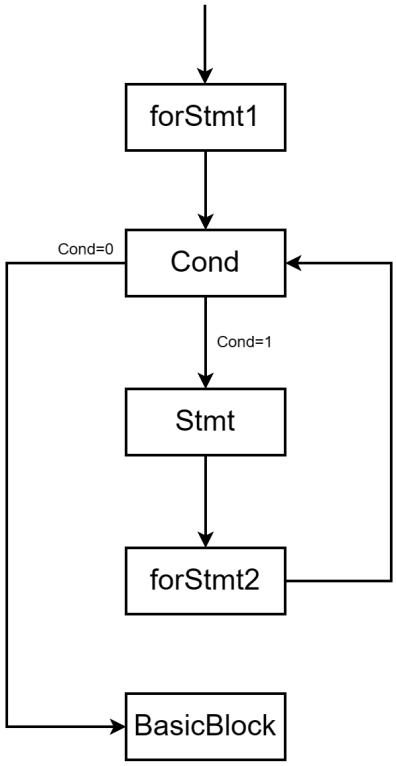
根据`Stmt`类型分别处理：

- `processLValstmt`
  1. 符号表查询：根据`stmt.lVal.getIdent()`从符号表中查找左值变量信息，获得其LLVM符号。
  2. 计算左值地址：如果`lVal`中包含数组索引表达式（`stmt.lVal.getExp()`），说明是为数组某个元素赋值，调用`locate`方法计算变量的具体地址。若无索引表达式，说明是为标量赋值，直接使用变量的LLVM地址。
  3. 计算右值：
     - 若右值表达式（`lValExp`）可直接计算（`judgeCalculate`返回`true`），则计算值并根据变量类型修正。
     - 若右值需动态计算，调用`getCalInstructions`获取对应的计算指令。
  4. 生成存储指令：`StoreInst`指令将右值结果存储到左值的地址。
  5. 返回指令列表。
- `processExpStmt`：若`stmt.exp`为空，直接返回空指令列表，否则：
  1. 若表达式可直接计算（`judgeCalculate`返回`true`），无需生成指令，直接返回空列表。
  2. 若`Exp`需要动态计算：调用`exp.getCalInstructions`生成动态计算指令，并返回指令列表。
- `buildBasicBlocks`：对`Stmt->Block`，调用`buildBasicBlocks`。
- `processIfElseStmt`
  1. 初始化标签：为`if`语句块、`else`语句块（如果存在）、整个`if-else`结构结束分别创建唯一的`ifLabel`、`elseLabel`、`endLabel`。
  2. 处理`Cond`：

     `cond`本质上是一个`LOrExp`，`LOrExp`可以分解为若干`LAndExp`的或语句，每个`LAndExp`又可以分解为若干`EqExp`的和语句。因此以`EqExp`为基本单元。

     对于`EqExp`：
     - 若为`true`，如果其不是所在的`LAndExp`中的最后一个`EqExp`，则跳转处理所在的`LAndExp`下的下一个`EqExp`；否则，`cond`为`true`。
     - 若为`false`，如果其所在的`LAndExp`不是`LOrExp`中的最后一个，则跳转处理`LOrExp`中的下一个`LAndExp`；否则，`cond`为`false`。
  3. 判断是否有`else`分支：
     - 有`else`分支：`cond`为`true`，则跳转`ifLabel`，否则跳转`elseLabel`。`ifLabel`、`elseLabel`结束均跳转`endLabel`。
     - 无`else`分支：`cond`为`true`，则跳转`ifLabel`，否则跳转`endLabel`。`ifLabel`结束跳转`endLabel`。
- `processForStmt`



1. 初始化语句：如果 `forStmt1` 存在，处理初始化语句并生成对应指令。
2. 条件表达式：如果 `forCond` 存在，生成条件判断的指令和分支。否则，默认条件始终为真。
3. 处理主循环体，生成其指令序列。
4. 增量语句：如果 `forStmt2` 存在，处理增量语句并生成指令。
5. 结束标识：生成结束标签以标识 `for` 循环的终点。

核心宗旨：上图中若某一部分缺失，则跳转到下一部分。

- `processReturnStmt`

  1. 无返回值（`void`）：如果 `return` 语句没有返回表达式（`stmt.returnExp == null`），则创建一个类型为 `VoidTyID` 的 `ReturnInst` 指令，并将其添加到 `instructions` 列表中。
  2. 带返回值：

     获取函数类型：通过 `funcName` 从 `FunctionThing.funcIrType` 获取该函数的返回类型。

     对于`stmt.returnExp`：

     - 如果表达式可以在编译时计算，创建 `ReturnInst`，使用计算得到的值和函数的返回类型，并将其添加到 `instructions` 列表中。
     - 如果表达式需要在运行时计算：通过 `exp.getCalInstructions` 获取计算表达式的指令，并将其添加到 `instructions` 列表中。然后创建 `ReturnInst`，使用函数的返回类型和计算结果的名称，并将其添加到 `instructions` 列表中。

     注意：返回值的类型需与函数类型保持一致，因此可能涉及**类型转化**。

- `processGetIntStmt`

  1. 符号表查询：根据`stmt.lVal.getIdent()`从符号表中查找左值变量信息，获得其LLVM符号。
  2. 计算左值地址：如果`lVal`中包含数组索引表达式（`stmt.lVal.getExp()`），说明是为数组某个元素赋值，调用`locate`方法计算变量的具体地址。若无索引表达式，说明是为标量赋值，直接使用变量的LLVM地址。
  3. 获取右值：创建局部变量，用于存储用户输入的整数值。该变量的类型为 32 位整数类型（`IntegerTyID`）。并创建 `InputInst` 指令，用于执行用户输入操作，将输入的值存储到

新创建的局部变量中。

4. 生成存储指令：`StoreInst`指令将右值结果存储到左值的地址。
5. 返回指令列表。

注意：右值的类型需与左值类型保持一致，因此可能涉及**类型转化。**

- `processGetChar`

  基本同`processGetIntStmt`，注意**类型转化。**

- `processPrintStmt`

  1. 处理格式化输出：对于字符串中的每个字符，检查是否为格式符（如`%d`或`%c`）。如果是格式符，处理相应的表达式：
     - 如果该表达式可以计算（`exp.judgeCalculate()`），则直接计算其值，并创建相应的`OutputInst`指令，加入到指令列表中。
     - 如果该表达式无法直接计算（例如需要计算的值复杂或无法立即计算），则生成该表达式的计算指令，并根据其类型（整数或字符）生成不同的`OutputInst`。

     注意：值的类型需与输出类型保持一致，因此可能涉及**类型转化。**
  2. **处理特殊字符**：如果遇到转义字符（如`\n`，`\t`，`\\`等），则根据字符的类型生成相应的输出指令（`OutputInst`），并加入到指令列表中。
  3. 处理剩余字符串：当遍历字符串常量中的普通字符时，会将它们加入到一个临时的字符串中，并在遇到格式符或特殊字符时，将字符串存入全局字符串常量`strStatements`中。
  4. 返回指令列表。

6.2 编码完成之后的修改

- **字符数组初始化时转义字符的处理**：起初我没有处理StringConst中的转义字符，对于`char s[15]="hello, world!\n"`，我会将`\n`作为`\`和`n`两个字符进行处理。修改后，通过特判处理了转义字符。
- `printf`**输出字符时处理转义字符**：错误原因与上一条相似，同样做了特判处理进行修改。
- **将**`printf`**部分由逐个字符输出，改为输出字符串**：在`processPrintStmt`中提取完整的字符串，将其添加至`Module`里，以全局变量的形式声明。

```
public static ArrayList<StrStatement> strStatements = new ArrayList<>();

public static middle.Class.Module buildModule(CompUnit compUnit,
LLVMSymbolTable LLVMSymbolTable) {
    ... // buildGlobalVariables
    ... // buildFunction
    return new Module(globalVariables, strStatements, functions, main);
}
```

- **处理函数形参**：函数`buildBasicBlock`之前，存储形参的值以备后续使用。

```
ArrayList<Value> instructions = new ArrayList<>();
for (Argument argument : argumentTIES) {
    LLVMSymbol LLVMSymbol = newTable.search(argument.ident);    // 找到形参对
应的symbol
    String addressName = "%LocalVariable_" + Count.getFuncInner();
    AllocaInst allocaInst = new AllocaInst(addressName, LLVMSymbol.irType);
    instructions.add(allocaInst);   // 分配空间
    StoreInst storeInst = new StoreInst(LLVMSymbol.irType,
LLVMSymbol.irName, addressName);
    instructions.add(storeInst);    // 存储形参的值
}
```

- **处理函数返回值**：创建`private static boolean funcReturn = false;`，用以记录当前函数是否出现`return`语句，在`buildBasicBlocks`的过程中更新该值，若该函数结束该值仍为`False`，则在输出时在函数末尾添上`return;`对应的LLVM语句。

```
if(!funcReturn){
    ReturnInst returnInst = new ReturnInst(new
IrType(IrType.TypeID.VoidTyID));
    res.append("\t"+returnInst.getOutput()+"\n");
}
```

- **类型转换**：

  代码生成的过程中涉及类型转化的地方较多，总结如下：

  1. 计算过程中：`char`转`int`进行计算（`i8`转`i32`）；`int`转布尔值（`i32`转`i1`）。
  2. 赋值语句（包括变量定义和`lval`赋值）：计算语句赋值给左值、`getchar`语句赋值给左值，需要`int`转`char`。
  3. `return`语句：`returnExp`的类型转化（`int`转`char`或`char`转`int`）。
  4. `printf`语句：值的类型需与输出类型保持一致（`int`转`char`或`char`转`int`）。
  5. 函数调用语句。

  主要修改是将类型转化语句聚合起来，形成统一的类型转化方法，这样在可能出现类型转化的地方调用即可，避免了代码重复冗杂。

```
public static void typeTransfer(ArrayList<Value> instructions,IrType irType)
{
    IrType irTypeTemp =
instructions.get(instructions.size()-1).getResIrType();
    if (irType.getNum()!=null &&
irTypeTemp.getNum()!=null&&!irType.getNum().equals(irTypeTemp.getNum())) {
        String transferName = "%LocalVariable_" + Count.getFuncInner();
        TransferInst transferInst = new TransferInst(transferName,
instructions.get(instructions.size() - 1).getResName(),irTypeTemp, irType);
        instructions.add(transferInst);
    }
}
```

7. 目标代码生成

根据LLVM中间代码，我进一步生成了mips。

7.1 编码之前的设计

7.1.1 存储方式

在这一阶段，我首先保证目标代码的正确性，没有考虑任何优化，采用了**栈式存储**：对每个变量名，都分配内存空间；对每条指令，申请寄存器，将要用到的变量加载到寄存器中，对寄存器进行一系列操作后，将结果保存到内存中，释放寄存器。

栈式存储的好处是，指令开始前申请寄存器，指令结束后释放寄存器，因此寄存器总是够用的，无需考虑寄存器的分配问题。缺点也显而易见，加载指令和存储指令消耗大量`cycles`，竞速优化排行预料之中会垫底。

寄存器部分仅使用`t`类寄存器（`$8-$15`，`$24-$25`），对每条指令，先申请寄存器，指令操作结束后，立刻释放寄存器。

```
public static Integer tType = 8;

public static Integer getTType() {
    while (!RegPool.regs[tType]) {  // 当前寄存器正在被用
        tType++;
        if (tType == 16) {
            tType = 24;
        } else if (tType == 26) {
            tType = 8;
        }
    }
    RegPool.regs[tType] = false;
    return tType;
}
```

存储类型部分，为了保证字节对齐、简化指令，我将所有变量都存成了`word`类型，只对其值进行类型转化。

7.1.2 构建符号表

由于从中端符号表中寻找该阶段所需符号表较为麻烦（不如新建符号表，随着目标代码生成而产生和消亡），同时该阶段我以LLVM生成的变量名作为符号名称（避免了变量重名的情况），而且此时关注的符号信息更为简单，因此我新建了符号表（至此，我的前端、中端、后端各有一张存储信息侧重点不同的符号表）。

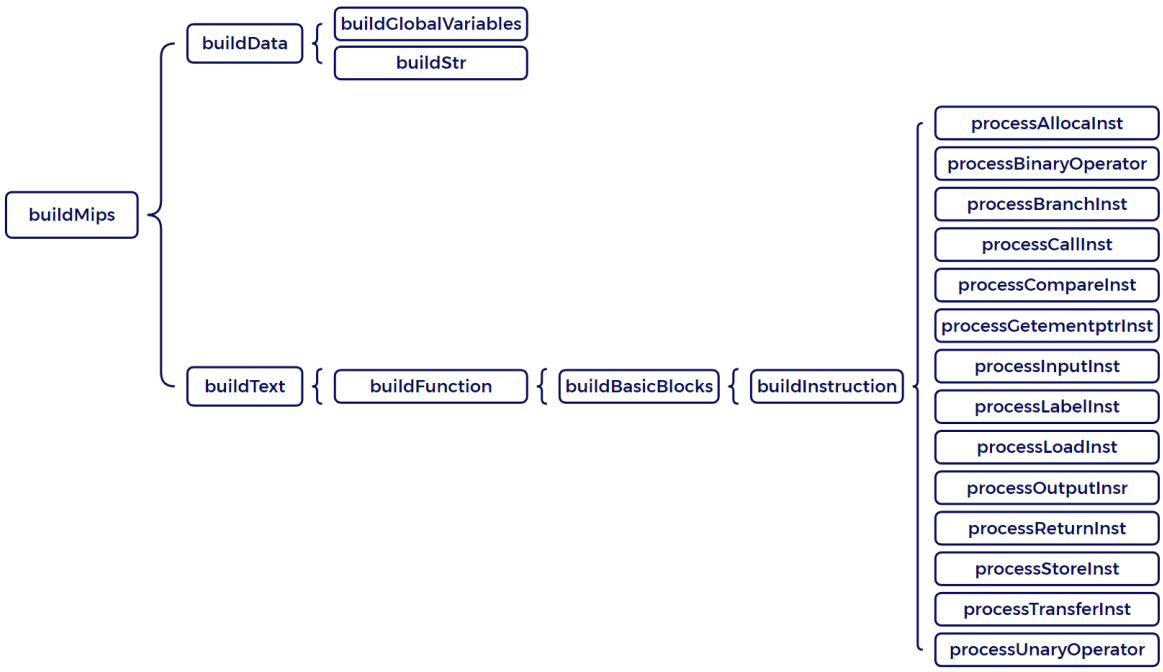
符号表类基本同6.1.1；符号存储信息如下：

```
public class MipsSymbol {
    public String irName = null;    // LLVM阶段该变量的名称
    public Integer base = 29;   // 基地址为$sp
    public Integer offset = null;   // 相对于基地址的偏移
    public Boolean isGlobal = false;    // 标明是否为全局变量/全局常量
    public Integer size = null;     // 数组大小
    public boolean isAddress = false;   // 该变量存储的是否为地址信息（为数组服务）

    public MipsSymbol(String irName) {
        this.irName = irName;
        this.offset = MipsCount.spOffset * 4;
        MipsCount.spOffset++;
    }
```

```
    public MipsSymbol(String irName, Integer size) {
        this.irName = irName;
        this.size = size;
        this.offset = MipsCount.spOffset * 4;
        MipsCount.spOffset = MipsCount.spOffset + size;
    }
}
```

7.1.3 mips基本结构



（以下加粗部分为编码后的修改）

7.1.3.1 buildData

`buildData` 是 `Data` 类中的核心方法，负责生成 `.data` 段的代码，定义全局变量和字符串常量。主要逻辑分为两部分：调用 `buildGlobalVariables` 构建全局变量部分，调用 `buildStr` 构建字符串常量部分，并最终将结果拼接为完整的 `.data` 段代码。

- `buildGlobalVariables`

  `buildGlobalVariables` 方法用于生成 `.data` 段中的全局变量定义。它逐一处理输入的全局变量列表 `globalVariables`，根据变量类型和维度生成不同的汇编代码，并更新符号表 `mipsSymbolTable`。

  1. 处理标量：如果全局变量是标量，直接生成 `.word` 指令，将变量的值写入汇编代码。并为该变量在符号表中创建对应的 `MipsSymbol` 对象，标记为全局标量（`isGlobal = true`）。
  2. 处理数组：如果全局变量是数组，根据是否初始化为零值区分处理，同样，将数组对应的符号信息添加到符号表中，标记为全局数组（`isGlobal = true`）。
     - 零值数组：使用格式 `0:n` 定义，其中 `n` 是数组的大小。
     - 非零值数组：逐一拼接数组元素的值，逗号分隔。

- `buildStr`

  `buildStr` 方法用于生成 `.data` 段中的字符串常量定义。遍历字符串语句列表 `strStatements`，为每个字符串创建对应的符号（`MipsSymbol`），并标记为全局的，添加到符号表中；构造 `.asciiz` 指令：每个字符串的指令以 `变量名: .asciiz "字符串内容"` 的形式定义，并换行分隔。

7.1.3.2 buildText

通过遍历函数列表，将每个函数的LLVM中间代码转换为对应的 MIPS 指令，生成目标代码。

- `buildFunction`

  1. 栈空间分配：根据函数参数的数量，计算需要分配的栈空间（`spOffset`）。如果参数超过 4 个，**多余的参数存储在栈上。**接着创建一个新的符号表，确保每个函数的符号表互不干扰。
  2. 处理形参：为形参申请空间：函数的形参通过 `AllocaInst` 指令分配空间，并记录是否是指针类型（`isAddress` 标志）；存储形参实值：根据目标寄存器的位置（例如 `$a0-$a3` 或堆栈偏移量），生成 `LW` 和 `SW` 指令将形参实值存储到内存中。
  3. 处理基本块：对函数的每个基本块调用 `buildBasicBlock` 方法生成指令，形成指令列表。
  4. 返回指令：如果函数没有显式的 `return` 指令，则插入一个默认的返回指令（`ReturnInst`）。
  5. 栈空间回收：**在函数开始时生成调整栈指针的指令（`ADDI`），在函数结束前保存返回地址（`SW`）。同时对某些指令如 `LW` 和 `ADDI` 进行偏移修正。**

- `buildBasicBlocks`

  为单个基本块生成 MIPS 指令：基本块中的每条指令都会通过调用 `buildInstruction` 方法生成对应的 MIPS 指令。

- `buildInstruction`（具体见7.1.3.3）

  将中间代码中的单条指令转化为 MIPS 指令：针对不同类型的指令，调用对应的处理方法生成指令列表。

7.1.3.3 process类

- `processAllocaInst`

  为每个局部变量分配一个 `MipsSymbol` 对象（获取内存地址，记录基地址和偏移量），记录变量名、是否为指针，以及分配的大小（如数组大小）。

- `processBinaryOperator`

  实现对二元操作（加法、减法、乘法、除法、去模、与、或）的处理：

  1. 寄存器资源管理：
     - 通过 `MipsCount.getTType()` 分配可用的临时寄存器。
     - 使用完成后，通过 `RegPool.regs` 标记寄存器的使用状态，避免资源冲突。
  2. 变量加载：
     - 如果操作数均为数值，说明在中间代码生成阶段已经处理过，输出错误提示并跳过后续步骤。
     - 单独处理特殊模式（一个操作数为变量名，另一个操作数为常量值）：从符号表中获取变量的符号信息（`MipsSymbol`），`LW` 获取变量，`LI` 获取常量立即数，使用立即数操作（如 `ADDI`、`ANDI` 等）生成目标寄存器值，通过存储指令 `SW` 完成存储。
     - 一般二元运算处理：如果操作数为变量名，通过符号表查找，加载对应的基地址和偏移量；如果操作数为数值，直接通过 `LI` 指令加载到寄存器中。
  3. 二元运算操作：
     - `processRegNumOperator`：处理寄存器与立即数的运算，生成立即数操作指令（如 `ADDI`、`ANDI`、`ORI` 等）。
     - `processRegRegOperator`：处理寄存器与寄存器的运算，生成对应的寄存器操作指令（如 `ADD`、`SUB` 等）。
  4. 存储操作：将计算结果存入目标寄存器，并通过符号表记录目标变量，最后通过 `SW` 指令完成存储。

- `processBranchInst`

处理分支指令（`BranchInst`），分为条件分支（`ifTrue` 和 `ifFalse`）和无条件跳转（`dest`）两种情况：

1. 条件分支：LLVM为 `br i1 cond ifTrue ifFalse`，若 `cond` 为 `true`，跳转 `ifTrue`，否则跳转 `ifFalse`。mips中用 `beq` 和 `j` 组合实现：通过 `LW` 指令加载条件变量的值到寄存器 `reg1`，使用 `LI` 指令将值 `1` 加载到寄存器 `reg2`，生成 `BEQ` 指令，比较 `reg1` 和 `reg2` 的值，若相等则跳转到 `ifTrue` 标签，生成 `J` 指令，无条件跳转到 `ifFalse` 标签。
2. 无条件跳转：直接使用 `j` 跳转指令。

- `processCallInst`

  处理函数调用指令（`CallInst`），包括参数传递、函数调用和返回值存储三个部分：

  1. 参数处理：对于前 4 个参数，将其值加载到 `$a0` 到 `$a3` 寄存器中，对于超过 4 个的参数，依次压入栈中。**调整栈指针（`$sp`）以预留存储空间。**
  2. 函数调用：使用 `JAL` 指令跳转到目标函数。若有多余参数（超过 4 个），**在返回后恢复栈指针。**
  3. 返回值处理：若函数有返回值，将 `$v0` 中的值移动到临时寄存器，并存入符号表指定的内存地址。

- `processCompareInst`（基本同`processBinaryOperator`）

  注意：由于mips只有小于指令，因此其余比较指令需要借助 `slt`、`or` 和 `xori` 做一些变换。

CompareType	原义	等价义
eq	reg1==reg2	!(reg1 < reg2 || reg2 < reg1)
ne	reg1 != reg2	reg1 < reg2 || reg2 < reg1
sgt	reg1 > reg2	reg2 < reg1
sge	reg1 >= reg2	!(reg1 < reg2)
slt	reg1 < reg2	reg1 < reg2
sle	reg1 <= reg2	!(reg2 < reg1)

- `processGetelementptrInstr`

  1. 计算基地址：根据输入符号 `LLVMSymbol` 的属性判断其存储类型（地址类型、常量类型或寄存器类型），获取基地址：
     - 地址类型（`isAddress` 为 `true`）：通过 `LW` 指令从内存加载基地址。
     - 常量类型（`isGlobal` 为 `true`）：通过 `LA` 指令加载符号对应的内存地址。
     - 寄存器类型：通过 `MOVE` 指令直接获取寄存器中的基地址。
  2. 计算目标地址：若 `Getelementptr` 的偏移量为常量：直接使用 `ADDI` 指令计算目标地址。若偏移量为变量：从符号表获取偏移量变量的存储位置，使用 `SLL` 指令将偏移值左移两位（乘以4）以考虑字节对齐。使用 `ADD` 指令将偏移值与基地址相加，得到目标地址。
  3. 更新符号表：计算得到的目标地址存储到新符号中，并添加到 `MipsSymbolTable` 中供后续指令使用。

- `processInputInst`

  1. 确定系统调用编号、发起系统调用：根据输入的 `irType` 类型，选择适当的系统调用编号并生成 `LI` 指令，并添加 `SYSCALL` 指令发起系统调用。

输入类型	系统调用号	对应指令
i32	5	`li $v0, 5`
i8	12	`li $v0, 12`

2. 存储输入结果：创建新的 `MipsSymbol` 对象，用 `SW` 指令将 `$v0` 中存储的输入结果保存到符号表中对应的变量地址。

- `processLabel`：获取标签名，将mips标签指令加入指令列表。
- `processLoadInst`

  从指定的内存地址加载数据，并将其存储到符号表中新的变量位置。

  注意：若指定内存地址为地址类型，则首先使用 `LW` 指令加载该地址变量的值（即基地址），再生成第二条 `LW` 指令加载目标值作为真实内存地址。

- `processOutputInst`

  1. 加载变量：

变量类型	输出指令
全局变量	`la $a0, hello    # 将字符串地址加载到 $a0`
普通变量	`lw $a0, offset($base)  # 加载变量 varA 的值到 $a0`
立即数	`li $a0, num            # 加载 num 的值到 $a0`

  2. 输出：

输出分类	输出指令
输出字符串	`li $v0, 4        # 设置系统调用编号为 4` `syscall         # 系统调用`
输出 `i32` 类型	`li $v0, 1              # 设置系统调用编号为 1` `syscall               # 系统调用`
输出 `i8` 类型	`li $v0, 11              # 设置系统调用编号为 11` `syscall               # 系统调用`

- `processReturnInst`：

  如果当前处理的函数是主函数，则系统调用结束程序。

  否则：

  1. 处理返回值（如果有）：
     - 立即数返回（用 `LI` 指令将立即数加载到 `$v0`）；
     - 变量返回：（在符号表中查询变量对应的信息，根据变量是否为常量生成 `LW` 指令，将变量值加载到 `$v0`）。
  2. **恢复函数调用时保存的返回地址和栈指针**：使用 `LW` 指令将保存的返回地址加载到 `$ra`，并动态调整栈的偏移量。
  3. 返回调用者：使用 `JR` 指令跳转到 `$ra`，返回调用函数。

- `processStoreInst`：

  使用`load`类指令获取待存储值，使用`SW`指令，将待存储值存入内存。

注意：如果目标变量是一个地址，则存向目标变量指向的地址。

- `processTransferInst`：

  只需考虑`i32`类型的变量转`i8`类型，取其后八位进行存储即可。

- `processUnaryOperator`：

  由于LLVM阶段的处理，此处一元操作仅有`!`操作，利用`xori`指令实现即可，基本同`processBinaryOperator`。

7.2 编码完成之后的修改

LLVM指令转化为mips指令，实质上是一个翻译指令的过程，由于在该阶段我采取了简单的栈式存储，不涉及寄存器的分配和占用，因此翻译工作较为简单，没有出现什么错误。

编码完成之后的修改主要集中在栈指针的偏移、函数声明和函数调用时参数的存储。

栈指针的偏移

对于每个函数，在处理完函数内所有基本块后，向顶层返回该函数内部占用内存的大小。

利用该值，在函数标签指令后面回填一句`ADDI addi = new ADDI(29, 29, -MipsCount.spOffset * 4)`指令，为的是在函数起始为该函数申请足够的空间、来存储函数内部变量。这样做可以避免内存冲突。

与之相应的，为了正确存储`$ra`，在申请指令后，回填一句`SW sw = new SW(31, 29, 4 * (MipsCount.spOffset - 1))`指令。

同时，由于在我的设计中，数组存储地址用的是相对地址（相对于`$sp`的地址），因此函数所有指令生成结束后，还需要对形参的读取、`$ra`的读取、`$sp`恢复偏移做修改：

```
// Text.buildFunction 中对于形参
LW lw = new LW(reg, 29, 4 * number - 16);   // 4 *number-16代表上层函数中，形参地址偏移，由于当前函数中，$sp 发生偏移，因此需要加上 MipsCount.spOffset * 4
lw.needModify = true;
```

```
// processReturnInst.processReturnInst 中对于$ra
LW lw = new LW(31, 29, -4); // 由于当前函数中，$sp 发生偏移，因此需要加上 MipsCount.spOffset * 4
lw.needModify = true;
```

```
// processReturnInst.processReturnInst 中对于$sp
ADDI addi = new ADDI(29, 29, 0);    // 运行到此处还不知道$sp的偏移量，需要处理完该函数后进行回填修改
addi.needModify = true;
```

函数所有指令生成结束后，对以上指令做修改：

```
for (mipsInstruction instruction : instructions) {
    if (instruction instanceof LW lw && lw.needModify) {
        lw.offset = lw.offset + MipsCount.spOffset * 4;
    } else if (instruction instanceof ADDI addiTemp && addiTemp.needModify) {
        addiTemp.num = MipsCount.spOffset * 4;
    }
}
```

函数调用时存储实参

宗旨：前4个参数存储在 `$a0` 至 `$a3` 寄存器中，其余参数存储在栈上，并通过栈指针（`$sp`）访问。

将前四个参数加载到 `$a0` 至 `$a3` 寄存器中。

处理第五个参数前，为前4个参数以外的参数分配足够空间：

```
Integer offset = (callInst.argumentName.size() - 4) * 4;     // 计算空间大小
ADDI addi = new ADDI(29, 29, -offset);  // 栈指针偏移，留够空间
```

自第五个参数起，利用`LW`加载实参，利用`SW sw = new SW(reg, 29, i * 4 - 16)`，将实参存入以上空间中。

如果参数个数大于四，在存储完实参后，将`$sp`指针指回原来的地方`ADDI addi = new ADDI(29, 29, offset)`。

如果该函数有返回值，将 `$v0` 中的值移动到临时寄存器，并存入符号表指定的内存地址。

函数声明时参数处理

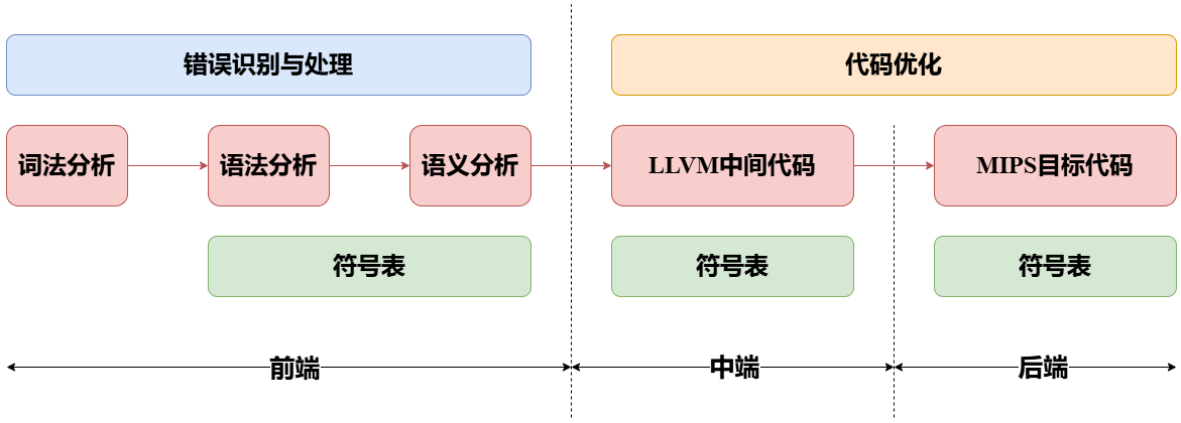
依据：前4个参数存储在 `$a0` 至 `$a3` 寄存器中，其余参数存储在栈上，并通过栈指针（`$sp`）访问。

将前四个参数从 `$a0` 至 `$a3` 寄存器加载到内存中。

第五个参数起，利用`$sp`及其偏移加载参数，并存入内存中。

```
LW lw = new LW(reg, 29, 4 * number - 16);
lw.needModify = true;   // 由于函数开始会对$sp做偏移，因此后续还要回填修改，见本节“栈指针的偏移”
SW sw = new SW(reg, mipsSymbol.base, mipsSymbol.offset);
```

8. 代码优化设计



以上是我整个编译器的组成部分，我主要是在生成LLVM的过程中和生成MIPS的过程中做了一些优化。我的优化分为两部分：一是在LLVM生成和MIPS生成过程中，内嵌完成的优化；二是在前者基础上完成的额外的“优化”。

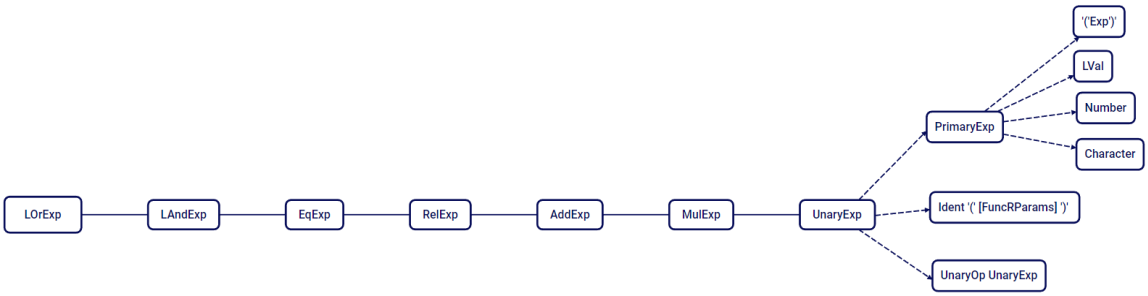
为了管理第二种优化，我设置了`Optimize`类，以便灵活开关各项优化：

```
public class Optimize {
    public static boolean optimize = true;  // 优化总开关
    public static boolean optimize1 = true && optimize; // 优化开关1
    public static boolean optimize2 = true && optimize;  // 优化开关2
    ...
}
```

8.1 LLVM 优化

8.1.1 表达式求值简化

对于`Exp`求值：



我会首先判断能否直接得出值：对表达式递归，直至得到`PrimaryExp`或`Ident'('FuncRParams')'`。

- 如果递归终点为`PrimaryExp`，且`PrimaryExp`为`LVal`，且`LVal`对应的`symbol`是常量，则可直接得出值；
- 如果递归终点为`PrimaryExp`，且`PrimaryExp`为`Number`，则可直接得出值；
- 如果递归终点为`PrimaryExp`，且`PrimaryExp`为`Character`，则可直接得出值。

如果表达式中每一部分的取值都能直接得到，则可以在编译器内部、在LLVM生成的过程中直接算出值，大大减少了中间代码、目标代码生成过程中的常数运算指令。

8.1.2 跳转指令简化

在LLVM中，对于可能产生的条件跳转指令`br i1 <cond>, label <iftrue>, label <iffalse>`，我利用短路求值和该文档第一节“表达式求值简化”中的求值规则，检查`cond`的布尔值能否直接得到，若可直接得到，如果`cond == true`，则原条件跳转指令可简化为`br label<iftrue>`，同时不生成标签`<iffalse>`引领的代码块；如果`cond == false`，则原条件跳转指令可简化为`br label<iffalse>`，同时不生成标签`<iftrue>`引领的代码块。

8.1.3 删除死代码

8.1.3.1 删除`continue`/`break`/`return`后的代码

在LLVM生成过程中，对`Stmt->continue`和`Stmt->break`、以及`return`文法做标记，若出现`contiue`、`break`、`return`，则跳过本基本块内剩下的语句，直接解析下一个基本块。

```
for (BlockItem blockItem : block.getBlockItems()) {
    ArrayList<Value> instructions = new ArrayList<>();
    boolean flag = false;
    boolean isFor = false;
    if (blockItem.getDecl() != null) {
        Decl decl = blockItem.getDecl();
        if (decl instanceof ConstDecl constDecl) {
            ...
```

```
    } else if (blockItem.getStmt() != null) {
        Stmt stmt = blockItem.getStmt();
        if (stmt.getKind() == Stmt.StmtType.LVal) ...
        else if (stmt.getKind() == Stmt.StmtType.Exp) ...
        else if (stmt.getKind() == Stmt.StmtType.Block) ...
        else if (stmt.getKind() == Stmt.StmtType.IfElse) ...
        else if (stmt.getKind() == Stmt.StmtType.For) ...
        else if (stmt.getKind() == Stmt.StmtType.Break) {
            ...
            flag = true;    // 标记是continue/break/return
        }
        else if (stmt.getKind() == Stmt.StmtType.Continue) {
            ...
            flag = true;    // 标记是continue/break/return
        }
        else if (stmt.getKind() == Stmt.StmtType.Return) {
            ...
            flag = true;    // 标记是continue/break/return
        }
        else if (stmt.getKind() == Stmt.StmtType.Getint) ...
        else if (stmt.getKind() == Stmt.StmtType.GetChar) ...
        else if (stmt.getKind() == Stmt.StmtType.Print) ...
    }
    BasicBlock basicBlock = new BasicBlock(instructions);
    basicBlocks.add(basicBlock);
    if (Optimize.optimize2 && flag) {
        // 如果是continue/break/return，则不生成该基本块内的后续代码
        break;
    }
}
```

8.1.3.2 删除未用到的函数

在LLVM生成过程中，调用函数时对函数进行标记，若某个函数从未被调用，则在LLVM指令集合中删去该函数的全部指令。

```
public static String buildText(ArrayList<Function> functions, Function main,
MipsSymbolTable mipsSymbolTable) {
    ...
    for (Function function : functions) {
        if (Optimize.optimize4 &&
FunctionThing.isCalled.contains(function.name)) {     // LLVM到MIPS只翻译会被用到的
函数
            ...
        }
    }
    ...
    return stringBuilder.toString();
}
```

8.2 MIPS优化

8.2.1 引入 $0 寄存器

在最初生成mips的过程中，除了`$sp`，我仅使用了`t`类寄存器（`$8-$15`，`$24-$25`），在优化中，我首先引入了$0寄存器，主要替换了两种情况：

情况一：对于运算指令、比较指令、填入函数实参、输出指令等一切涉及需要加载立即数 0 的地方，将加载立即数 0 到某个寄存器，再操作寄存器，简化为直接操作 $0 寄存器。

情况二：对于条件跳转指令，我原先的实现是

```
li regTemp, 1
beq reg, regTemp, iftrue
j iffalse
```

现在引入$0寄存器，修改为：

```
beq reg, 0, iffalse
j iftrue
```

8.2.2 引入寄存器池

为了减少`lw`、`sw`等内存存取指令的开销，我对局部变量引入了寄存器池，基本思路是：需要获取某个变量的值时，如果该变量在寄存器里，对该寄存器操作，如果不在，则给该变量分配一个寄存器，如果当前所有寄存器都被占用，则释放掉别的寄存器，以存储该变量的值。

具体实现中要注意一些细节：

- 根据局部性原理，当前所有寄存器都被占用、需要释放掉某个寄存器时，尽可能释放掉所有寄存器中上次使用时间最早的寄存器。
- 对于一条指令可能需要多个寄存器，如`add resultReg, reg1, reg2`，如果寄存器的申请顺序为`reg1`、`reg2`、`resultReg`，那么在申请`reg2`时，如果当前所有寄存器被占用，需要释放某个寄存器，注意不能释放`reg1`，同理，在申请`resultReg`时，需要注意不能释放`reg1`/`reg2`。
- 在函数调用前，需要将寄存器池内寄存器的值都存入内存，清空寄存器，既是为了保证当前函数中的变量存值正确，也是为了保证所调用的函数的寄存器初始时为空寄存器。
- 在进入`for`循环前将寄存器池内寄存器的值都存入内存，清空寄存器。我曾在这里出现了bug，不清空将导致以下问题：

  对于for循环（以下为伪代码，global处也可以是内存地址），这将导致下次循环的时候，$9存的并不是我设计时以为的那个值。

  ```
      lw $9 global
  for循环：
      ... # 对 $9 的操作
      sw $9 global
      lw $9 global2
      j for循环
  ```

为了将更多变量的值存入寄存器、而非存入内存中，最大化减少内存存取指令数量，我引入了`s`类寄存器（`$16-$23`），并将`s`类寄存器当作`t`类寄存器使用，共同组成寄存器池。

8.2.3 删除原地跳转指令

当生成所有mips指令后，检查指令集合，如果出现以下情况：

情况1：

```
    j label
label:
    ...
```

情况2：

```
    beq reg1,reg2,label
label:
    ...
```

情况3：

```
    beq reg1,reg2,label
    j label
    ...
```

即当前指令和下一条指令指向同一条指令，则属于原地跳转指令，删去当前指令。

8.2.4 删除冗余类型转化

由于在MIPS生成中，我将`int`类型和`char`类型变量均存储为`word`形式，因此二者在形式上完全相同，只需考虑取值问题。又因为`char`转为`int`类型，其值不变，因此在MIPS生成过程中，只需考虑`int`转`char`这种LLVM类型转化指令。

更进一步，在一个基本块中，除了`printf`、`return`、函数调用部分，其余部分的类型转化不影响最终结果，因此，考虑范围进一步缩减至`printf`、`return`、函数调用部分`int`转`char`的类型转化。

综上，将LLVM类型转化指令翻译为MIPS的过程中，只完整翻译`printf`、`return`、函数调用部分`int`转`char`的类型转化，其余类型转化只新增变量名，编译器内部对其赋值，而不生成赋值指令。

8.2.5 乘除法优化

由竞速优化计算cycles的规则可知，乘除指令权重较大，我学习教程，试图将乘除指令转化为移位运算和加减运算，实现了寄存器`reg`与立即数`num`相乘的乘除法指令优化。

8.2.5.1 乘法优化

思路如下：

- 特殊处理`num`为 0 或 1 的情况；
- 当`num`不为 0 且不为 1 时，获取`num`的绝对值`numABS`，遍历`numABS`的每一位，通过位运算`(numABS & 1)`检测当前最低位是否为 1。如果当前位为 1，表示乘数中该位对应的值需要累加到最终结果中，使用移位指令（`SLL`指令）将源寄存器`reg`的值左移相应的位数`shift`，并累加到结果寄存器`resReg`。（如果是第一次累加，直接将左移后的结果存储到目标寄存器中；否则，需要使用一个临时寄存器`regTemp`来存储移位后的中间结果，并通过加法指令（`ADD`指令）累加到目标寄存器。）注意如果`num`本身小于零，做完位运算还要做符号运算。

这里举个例子：寄存器`reg`中的值与立即数 13 相乘：

$$reg \times 13 = reg \times 0b1101 = reg \times (2^3 + 2^2 + 2^0)$$

$= reg \times 2^3 + reg \times 2^2 + reg \times 2^0 = reg << 3 + reg << 2 + reg << 0$

同时比较正常乘法和位运算乘法的指令指令权重总和，选取加权和更小的一种输出。

```
// 移位运算
ArrayList<mipsInstruction> instructions1 = new ArrayList<>();
if (num == 0) {
    LI li = new LI(resReg, 0);
    instructions1.add(li);
} else if (num == 1) {
    MOVE move = new MOVE(resReg, reg);
    instructions1.add(move);
} else {
    Integer numABS = Math.abs(num);
    int shift = 0;
    boolean lowest = true;

    while (numABS > 0) {
        if ((numABS & 1) == 1) {    // 最低位为1
            if (lowest) {
                SLL sll = new SLL(resReg, reg, shift);
                instructions1.add(sll);
                lowest = false;
            } else {
                Integer regTemp;
                if (Optimize.optimize7) {
                    regTemp = Operation.getTTYpeOptimized(instructions);
                } else {
                    regTemp = MipsCount.getTType();
                }
                SLL sll = new SLL(regTemp, reg, shift);
                ADD add = new ADD(resReg, resReg, regTemp);
                instructions1.add(sll);
                instructions1.add(add);
                RegPool.regs[regTemp] = true;
            }
        }
        numABS = numABS >> 1;
        shift++;
    }
    if (num < 0) {
        SUB subMUL = new SUB(resReg, 0, resReg);
        instructions1.add(subMUL);
    }
}
// 正常运算
ArrayList<mipsInstruction> instructions2 = new ArrayList<>();
LI li = new LI(resReg, num);
MUL mul = new MUL(resReg, resReg, reg);
instructions2.add(li);
instructions2.add(mul);
// 比较两种方法加权和
if (instructions1.size() < 4) {
    instructions.addAll(instructions1);
} else {
    instructions.addAll(instructions2);
}
```

8.2.5.2 除法优化

除法优化部分我主要参考了教程。

在 MIPS 中，除法（`DIV` 指令）涉及复杂的硬件操作，会显著增加计算时间。优化的目标是减少复杂操作，提高运行效率。

利用乘法逼近，数学上，除法可以表示为$quotient = \frac{n}{d}$

等价于：$quotient = \frac{n \cdot m}{2^N}$

其中 $m = \lceil \frac{2^N}{d} \rceil$，即通过将被除数 n 乘以常数 m，然后右移 N 位（即除以 2^N），就可以得到一个近似的商值。

进一步，根据论文，有$2^{N+l} \leq m * d \leq 2^{N+l} + 2^l$，因此我选择设$l = 1$，$m = \lceil \frac{2^{N+l}}{d} \rceil$。

实际实现中，还要特判被除数和除数的正负性，具体代码如下：

```
int numABS = Math.abs(num);

if (numABS == 1) {
    if (num == 1) {     // 如果 num 为正数 1，直接将寄存器 reg 的值移动到结果寄存器
resReg
        instructions.add(new MOVE(resReg, reg));
    } else {    // 如果 num 为负数 -1，计算结果为 -reg
        instructions.add(new SUB(resReg, 0, reg));
    }
} else if (numABS == 2) {   // num 的绝对值为 2 的情况
    Integer regTemp0 = Operation.getTTYpeOptimized(instructions); // 存储被除数的符
号位
    Integer regTemp1 = Operation.getTTYpeOptimized(instructions); // 存储被除数的绝
对值
    instructions.add(new SRL(regTemp0, reg, 31));    // 记录被除数的符号位（符号位存储
在 regTemp0 中）
    instructions.add(new MOVE(regTemp1, reg));      // 将被除数拷贝到 regTemp1 中
    instructions.add(new BEQ(regTemp0, 0, "div_start_" + MipsCount.divCount));
// 如果符号位为 0（正数），跳到 div_start
    instructions.add(new SUB(regTemp1, 0, reg));    // 符号位为 1（负数），取被除数的
绝对值
    instructions.add(new LABEL("div_start_" + MipsCount.divCount));    //
div_start 标签
    instructions.add(new SRL(resReg, regTemp1, 1));    // 计算 resReg = regTemp1
>> 1 (等效于除以 2)
    if (num < 0) {    // 如果 num 为负数，取相反数
        instructions.add(new SUB(resReg, 0, resReg));
    }
    instructions.add(new BEQ(regTemp0, 0, "div_end_" + MipsCount.divCount));
// 如果符号位为 0（正数），跳到 div_end
    instructions.add(new SUB(resReg, 0, resReg));    // 如果符号位为 1（负数），再取
相反数
    instructions.add(new LABEL("div_end_" + MipsCount.divCount));    // div_end
标签
    MipsCount.divCount++;
    // 释放寄存器
    RegPool.regs[regTemp0] = true;
    RegPool.regs[regTemp1] = true;
} else {    // num 的绝对值大于 2 的情况
    Integer regTemp0 = Operation.getTTYpeOptimized(instructions); // 存储被除数的符
号位
```

```
    Integer regTemp1 = Operation.getTTYpeOptimized(instructions); // 存储被除数的绝对值
    Integer regTemp2 = Operation.getTTYpeOptimized(instructions); // 用于存储常量 m
    // 对被除数的处理同上
    instructions.add(new SRL(regTemp0, reg, 31));
    instructions.add(new MOVE(regTemp1, reg));
    instructions.add(new BEQ(regTemp0, 0, "div_start_" + MipsCount.divCount));
    instructions.add(new SUB(regTemp1, 0, reg));
    instructions.add(new LABEL("div_start_" + MipsCount.divCount));
    // 计算常量 m
    int l = 1;
    long m = (long) (Math.pow(2, 32 + l) / numABS);
    if (!(m * numABS <= (Math.pow(2, 32 + l) + Math.pow(2, l)) && m * numABS >= Math.pow(2, 32 + l))) {
        m++;
    }
    instructions.add(new LI(regTemp2, (int) m));
    // 计算高效除法结果
    instructions.add(new MULTU(regTemp1, regTemp2)); // regTemp1 * m
    instructions.add(new MFHI(regTemp1));          // 获取高位结果
    instructions.add(new SRL(resReg, regTemp1, l)); // resReg = regTemp1 >> l
    // 对除数和被除数的处理同上
    if (num < 0) {
        instructions.add(new SUB(resReg, 0, resReg));
    }
    instructions.add(new BEQ(regTemp0, 0, "div_end_" + MipsCount.divCount));
    instructions.add(new SUB(resReg, 0, resReg));
    instructions.add(new LABEL("div_end_" + MipsCount.divCount));
    MipsCount.divCount++;
    // 释放寄存器
    RegPool.regs[regTemp0] = true;
    RegPool.regs[regTemp1] = true;
    RegPool.regs[regTemp2] = true;
}
```