compiler_文档

参考编译器

主要参考的编译器为Pascal编译器。

总体结构和接口设计

有以下几个procedure：

`nextch` 读取下一个字符

`error` 打印错误信息

`adjustscale` 处理实数

`enter` 登记符号表

`enterarray` 登记数组符号表

`enterreal` 登记实常量表

`enterblock` 登记分程序信息入分程序表

`emit` 生成中间代码PCODE

`test` 检查符号合法性

`enter` 登记符号表

`constant` 处理常量

`typ` 处理类型

`parameterlist` 处理形参

`typedeclaration` 处理自定义类型

`variabledeclaration` 处理普通变量

`procdeclaration` 处理过程和函数

`statement` 处理各种语句

文件组织

课程组所提供的pascal编译器所有代码处在同一个word文件中。

编译器总体设计

总体结构

本编译器由四个部分组成，分别为：

- frontend:含有lexer和parser，进行词法分析和语法分析。
- middleend:定义了各种Value及Value的子类，负责将AST语法树转化为llvm中间代码。
- backend:负责将llvm中间代码转化为mips汇编代码。
- error:定义了错误处理时所需要的类，实际错误处理位于frontend中的parser部分。

接口设计

- frontend:

  `Category`类定义了单词的类别。

  `Lexer`：调用`next`读下一个单词，调用`peekToken`获得当前读到的单词

  `Compiler.getTokens`：用Lexer处理输入文件，将所有识别出的单词保存到`tokens`里

  `Parser`：接受`tokens`，调用各类Parser方法如`parseCompUnit`，构造语法树。

- middleend:

  `IRModule`：单例模式，`IRModule.getModuleInstance`返回唯一的iRModule。

  `Generator`：调用`visit`方法访问语法树，为IRModule增添GlobalVar或Function，为Function增添BasicBlock，为BasicBlock增添Instruction。

  `IROptimaze`：进行中端优化，修改irModule

  各类Value：IRModule、GlobalVar、Function、BasicBlock、Instruction。

  各类Type：SureArrayType、IntegerType、PointerType等

- backend:

  `MipsGenerator`：visit中端生成的llvm Instruction，生成mips代码，如`visitModule`

  `MipsFactory`：便捷地生成mips代码，如`genBrWithLabel`和`genLw`

文件组织

总体

```
├─backend
├─frontend
|   ├─lexer_package
|   └─paser_package
|       └─stmt_package
└─middleend
    ├─refill
    ├─symbol
    ├─type
    └─value
        └─user
            └─instruction
                └─terminateInst
```

frontend

```
|   Lexer.java
|   Parser.java
|
├─lexer_package
|       Category.java
|       Token.java
|
└─paser_package
    |   AddExp.java
```

```
    |  Block.java
    |  BlockItem.java
    |  CompUnit.java
    |  Cond.java
    |  ConstDecl.java
    |  ConstDef.java
    |  ConstExp.java
    |  ConstInitVal.java
    |  Decl.java
    |  EqExp.java
    |  Exp.java
    |  ForStmt.java
    |  FuncDef.java
    |  FuncFParam.java
    |  FuncFParams.java
    |  FuncRParams.java
    |  FuncType.java
    |  Ident.java
    |  InitVal.java
    |  LAndExp.java
    |  LOrExp.java
    |  LVal.java
    |  MainFuncDef.java
    |  MulExp.java
    |  Number.java
    |  PrimaryExp.java
    |  RelExp.java
    |  Stmt.java
    |  UnaryExp.java
    |  UnaryOp.java
    |  VarDecl.java
    |  VarDef.java
    |
    └─stmt_package
            StmtBC.java
            StmtBlock.java
            StmtExp.java
            StmtFor.java
            StmtIf.java
            StmtLValExp.java
            StmtPrint.java
            StmtRead.java
            StmtReturn.java
```

middleend

```
|  Factory.java
|  Generator.java
|  IRModule.java
|  IROptimaze.java
|  Operator.java
|  Value.java
|
├─refill
|      RefillFor.java
|      RefillIf.java
|      RefillUtil.java
```

```
|
├─symbol
|      SymbolTable.java
|      TableList.java
|
├─type
|      ArrayType.java
|      FunctionType.java
|      InstType.java
|      IntegerType.java
|      PointerType.java
|      SureArrayType.java
|      Type.java
|      VoidType.java
|
└─value
    |  CondValue.java
    |  ConstValue.java
    |  GlobalVar.java
    |  ParamVarValue.java
    |  User.java
    |  VarValue.java
    |
    └─user
        |  BasicBlock.java
        |  Function.java
        |  Instruction.java
        |  LibraryFunction.java
        |
        └─instruction
            |  AllocaInst.java
            |  BinaryInst.java
            |  CallInst.java
            |  CondString.java
            |  GetPtrInstSureArray.java
            |  GetPtrNormal.java
            |  IcmpInst.java
            |  LibraryCallInst.java
            |  LoadInst.java
            |  StoreInst.java
            |  ZextInst.java
            |
            └─terminateInst
                    BrInst.java
                    RetInst.java
```

backend

```
    GlobalMips.java
    MipsFactory.java
    MipsGenerator.java
    ValuePlace.java
```

error:

```
Error.java
ReturnType.java
Symbol.java
SymbolTable.java
Type.java
```

词法分析设计

compiler：

`private static void getTokens`：

1. **初始化**：
   - `String line;`：用于存储读取的每一行文本。
   - `boolean inComment = false;`：标志当前是否处于多行注释块中。
2. **读取处理**：
   - 使用 `while` 循环逐行读取文本。
   - `if (line.equals(""))`：跳过空行。
   - `if (inComment)`：如果当前处于多行注释中，则检查该行是否包含注释结束标记 `*/`。如果找到，则截取注释结束标记之后的内容并继续处理；如果没有找到，则跳过该行。
3. **词法分析**：
   - `Lexer lexer = new Lexer(line);`：为当前行创建一个 `Lexer` 实例，用于进行词法分析。
   - 使用 `while` 循环，通过 `lexer.next()` 方法逐个提取标记，直到行的末尾。
   - `Token curToken = lexer.peekToken();`：获取当前标记，并将其添加到 `tokens` 列表中。
4. **注释处理**：
   - `if (lexer.getValue() == -2)`：如果 `Lexer` 返回 `-2`，表明遇到了多行注释的开始，设置 `inComment` 为 `true`。
   - `else`：否则，设置 `inComment` 为 `false`。

frontend：

1.
   - 主要成员变量
     - `input`: 存储整个输入的字符串。
     - `pos`: 当前解析到的位置（索引）。
     - `curToken`: 当前解析到的Token。
     - `value`: 用于存储特殊值，表明是否当前input已分析完毕。
   - 方法
     - `next()`: 词法分析的核心方法，用于读取下一个Token。
     - `getValue()`, `getIndent()`, `getNumber()`: 辅助方法，用于获取当前Token的值或解析标识符和数字。
     - `isIdentChar()`: 判断字符是否可以是标识符的一部分。
     - `match()`, `matchNonLetter()`, `matchOne()`: 用于匹配特定的字符或字符串，并更新当前Token。
     - `peekToken()`: 返回当前Token。
2. **Token 类**
   - 主要成员变量
     - `category`: 标记的类别（如关键字、标识符等）。

        - `name`: 标记的字面值。
    - **构造方法。**
3. **Category 枚举**
    - 定义了不同类型的标记，如 `INTTK`（整数标记）、`IDENFR`（标识符）、`MAINTK`（main 关键字）等。

语法分析设计

Parser 类

- `tokens`: 存储词法分析阶段生成的所有标记（Token）的列表。
- `pos`: 当前解析到的标记的位置。
- `ans`: 存储解析过程中生成的语法结构的列表。
- **构造函数**:
    - 初始化 `tokens`, `pos`, 和 `ans`。
- **getToken()**:
    - 返回当前位置的 Token。
- **next()**:
    - 将当前 Token 转换为字符串并添加到 `ans`，然后增加 `pos` 以指向下一个 Token。
- **getAns()**:
    - 返回解析生成的语法结构列表。
- **parseXXX() 方法**:
    - 每个 `parseXXX` 方法都对应于语法规则的一个部分。例如，`parseCompUnit` 解析整个程序单元，`parseExpr` 解析表达式，`parseStmt` 解析语句等。
    - 这些方法通常会读取当前的 Token，并根据其类型调用适当的其他 `parseXXX` 方法来进一步解析语法结构。
    - 解析过程中，这些方法会更新 `ans` 列表，记录解析到的语法结构。
- **特定语法结构的解析**:

  例如，`parseIfStmt` 解析 if 语句，`parseForStmt` 解析 for 循环等。

语法成分类，以`Decl`为例：

- `type`: 一个整数，用于区分声明的类型。在这个上下文中，`0` 代表常量声明（`ConstDecl`），`1` 代表变量声明（`VarDecl`）。
- `constDecl`: 一个 `ConstDecl` 类型的对象，用于存储常量声明的详细信息。`
- `varDecl`: 一个 `VarDecl` 类型的对象，用于存储变量声明的详细信息。
- **构造函数**:
    - `Decl(ConstDecl constDecl)`: 接受一个 `ConstDecl` 对象作为参数。此构造函数创建一个类型为常量声明的 `Decl` 对象，并将传入的 `ConstDecl` 对象赋值给 `constDecl` 成员变量。
    - `Decl(VarDecl varDecl)`: 接受一个 `VarDecl` 对象作为参数。此构造函数创建一个类型为变量声明的 `Decl` 对象，并将传入的 `VarDecl` 对象赋值给 `varDecl` 成员变量。

错误处理设计

Error类

- **成员变量**:
  - `code`: 一个 `String` 类型，用于存储错误代码或错误信息。
  - `line`: 一个 `int` 类型，用于指示错误发生的行号。
- **构造函数**:
  - `Error(String code, int line)`: 接受一个错误代码和一个行号作为参数。构造函数将这些值分别赋值给 `code` 和 `line` 成员变量。
- **方法**:
  - `toString()`: 重写 `Object` 类的 `toString()` 方法。此方法返回一个表示错误的字符串，通常包含错误所在的行号和错误代码。
  - `getLine()`: 返回错误发生的行号。

SymbolTable

- **成员变量**:
  - `symbols`: 一个 `ArrayList`，其中每个元素是一个 `HashMap<String, Symbol>`。这个结构实现了多层符号表，其中每个哈希表代表一个作用域层级。
- **构造函数**:
  - 初始化 `symbols` 并添加一个空的哈希表，表示全局作用域或最外层作用域。

主要方法和功能

- **buildLevel()**:
  - 向 `symbols` 添加一个新的空哈希表，用于创建一个新的作用域层级。
- **checkRepeat(Ident ident, Type type, int ax)**:
  - 检查当前作用域层级中是否已存在给定名称的符号。
  - 如果不存在，则在当前层级添加该符号并返回 `false`；如果已存在，则返回 `true`。
- **fillProcSymbolDef(Ident ident, ArrayList paramAxes, ReturnType returnType)**:
  - 用于填充函数符号的定义信息。搜索所有层级以找到给定名称的函数，并填充其参数轴和返回类型信息。
- **findProcDef(Ident ident)**:
  - 从最内层作用域开始向外搜索，查找具有给定名称的函数定义。如果找到，则返回相应的 `Symbol` 对象；否则返回 `null`。
- **MissingCVDef(Ident ident)**:
  - 检查是否缺少给定名称的非函数（变量或常量）的定义。如果找不到相应的定义，则返回 `true`。
- **clear()**:
  - 移除最内层的符号表，通常在退出当前作用域时调用。
- **getAxOfIdent(String name)** 和 **getType(String name)**:
  - 用于获取指定标识符的维数和类型。如果在所有层级中都找不到，则输出错误信息。

Parser中的检测

```
private boolean checkRepeatDef(Ident ident, Type type, int ax) {
    if (this.symbolTable.checkRepeat(ident, type, ax)) {
        //被重复定义
        this.errors.add(new Error("b", ident.getLine()));
        return true;
    }
    return false;
}
```

```
public ForStmt parseForStmt() {
    int line = this.getToken().getLine();
    LVal lVal = this.parseLVal();
    this.next();//=
    Exp exp = this.parseExp();
    ForStmt forStmt = new ForStmt(lVal, exp);
    this.ans.add("<ForStmt>");
    if (lVal.isConst()) {
        this.errors.add(new Error("h", line));
    }
    return forStmt;
}
```

```
if (this.getToken().getCategory() == Category.LBRACK) {
    int line = this.getToken().getLine();
    this.next();//[
    if (this.getToken().getCategory() == Category.RBRACK) {
        this.next();//]
    } else {
        this.errors.add(new Error("k", line));
    }
    funcFParam.addAxes();
    while (this.getToken().getCategory() == Category.LBRACK) {
        this.next();//[
        line = this.getToken().getLine();
        ConstExp exp = this.parseConstExp();
        funcFParam.appendExp(exp);
        if (this.getToken().getCategory() == Category.RBRACK) {
            this.next();//]
        } else {
            this.errors.add(new Error("k", line));
        }
        funcFParam.addAxes();
    }
}
```

```
public MainFuncDef parseMainFuncDef() {
        this.next();//int
        this.returnTypeStack.push(ReturnType.INT);
        this.next();//main
        int line = this.getToken().getLine();
        this.next();//(
//        this.next();//)
        if (this.getToken().getCategory() == Category.RPARENT) {
            this.next();//)
        } else {
//            this.foundError("j", line);
            this.errors.add(new Error("j", line));
        }
```

```
if (lastBlockItem == null || lastBlockItem.isStmtReturn() == null) {
    this.errors.add(new Error("g", mainFuncDef.getBlock().getLastLine()));
}
```

```
if (this.getToken().getCategory() == Category.RPARENT) {
    this.next();//)
} else {
    this.errors.add(new Error("j", line));
}
```

```
for (StmtReturn stmt : stmtReturns) {
    if (stmt.getType() == ReturnType.INT) {
        if (supposedReturnType == ReturnType.VOID) {
            this.errors.add(new Error("f", stmt.getLine()));
        }
    }
}
```

```
if (supposedReturnType == ReturnType.INT) {
    BlockItem lastBlockItem = funcDef.getBlock().getLastBlockItem();
    if (lastBlockItem == null || lastBlockItem.isStmtReturn() == null) {
        this.errors.add(new Error("g", funcDef.getBlock().getLastLine()));
    }
}
```

```
if (this.getToken().getCategory() == Category.SEMICN) {
    this.next();//;
} else {
    this.errors.add(new Error("i", line));
}
```

```
public StmtBC parseStmtBC() {
    Token t = this.getToken();
    this.next();
    int line = this.getToken().getLine();
    StmtBC stmt = new StmtBC(t);
    if (this.getToken().getCategory() == Category.SEMICN) {
        this.next();//;
    } else {
```

```
            this.errors.add(new Error("i", line));
        }
        if (this.forNum <= 0) {
            this.errors.add(new Error("m", t.getLine()));
        }
        return stmt;
    }
```

等。根据不同的错误插入再Parser中对不同语法成分的parse过程中。

代码生成设计

中端

Generator

- **成员变量**:
  - `compUnit`: 代表编译单元，可能是经过语法分析后得到的抽象语法树（AST）。
  - `tableList`: 符号表列表，用于在编译过程中跟踪变量和函数的定义。
  - `curFunction`: 当前正在处理的函数。
  - `factory`: 用于创建和管理IR代码中使用的各种值和指令。
  - `irModule`: 代表整个IR模块，可能包含多个函数和全局变量。
  - `isGlobal`: 标志，用于区分全局变量和局部变量。
- **构造函数**:
  - 初始化 `compUnit`, `irModule`, `tableList`, `factory` 等成员变量。

主要方法和功能

- **visitCompUnit()**: 遍历编译单元，生成IR代码。
- **visitFuncDef()**: 遍历函数定义，生成对应的IR函数。
- **visitMainFuncDef()**: 特别处理主函数的IR代码生成。
- **visitDecl()**, **visitConstDecl()**, **visitVarDecl()** 等: 遍历不同类型的声明，生成相应的IR代码。
- **visitStmt()** 和相关的方法（如 **visitStmtIf()**, **visitStmtFor()**）: 遍历不同类型的语句，生成相应的IR代码。
- **visitAddExp()**, **visitMulExp()**, **visitUnaryExp()** 等: 遍历表达式，生成相应的IR代码。

Function

- **成员变量**:
  - `basicBlocks`: 一个 `ArrayList`，存储函数中的基本块（`BasicBlock` 对象）。
  - `returnType`: 函数的返回类型（`Type` 对象），例如 `VoidType` 或 `IntegerType`。
  - `name`: 函数的名称。
  - `registerNum`: 用于为函数内的虚拟寄存器分配唯一编号。
  - `params`: 函数的参数列表，包含 `ParamVarValue` 对象。
  - `symbolTable`: 与函数相关的符号表。
  - `calledNum`: 函数被调用的次数，用于优化。
  - `optimaze`: 标记是否对函数进行优化。
- **构造函数**:
  - 初始化函数名称、返回类型、基本块列表等，并将函数添加到IR模块中。

主要方法和功能

- **alloca_store_param()**: 为函数参数分配空间并存储它们的初始值。

- **addFirstBasicBlock()**, **addBasicBlock()**: 添加新的基本块到函数中。
- **fillBlock()**: 确保基本块之间正确连接，特别是对于无条件跳转。
- **getCurBasicBlock()**: 获取当前的基本块。
- **assignRegister()**: 为函数内的操作分配新的虚拟寄存器编号。
- **getPrint()**: 生成并返回函数的字符串表示，用于IR代码输出。
- **startOptimaze()**, **optimazeBlockJump()**, **optimizeCalculation()**: 执行不同层次的优化操作。

Instruction

- `Instruction` 类是IR代码表示的核心，每个 `Instruction` 对象代表了程序中的一条指令。
- `replaceValueWithConst` 方法提供了替换指令中某个值为常数的功能，以常数传播等优化。
- `Instruction` 类是一个基类，其子类有：

```
instruction
            |  AllocaInst.java
            |  BinaryInst.java
            |  CallInst.java
            |  CondString.java
            |  GetPtrInstSureArray.java
            |  GetPtrNormal.java
            |  IcmpInst.java
            |  LibraryCallInst.java
            |  LoadInst.java
            |  StoreInst.java
            |  ZextInst.java
            |
            └─terminateInst
                    BrInst.java
                    RetInst.java
```

- 类中的 `getPrint` 方法默认不产生任何输出。被所有子类重写以生成llvm代码。
- `isTerInst` 方法判断指令是否为终结指令。

Refill

在我的代码设计中，遇到if和for的时候需要进行回填操作，我实现了一个基础的RefillUtil类进行对AST树中Cond的回填以实现短路求值，其子类RefillFor和RefillIf各自扩展了功能以实现对if和for数据流的回填。

- 成员变量
  - `refillBasicBlocks`: 用于存储 `BasicBlock` 对象的嵌套列表，每个列表代表一个逻辑层次。
  - `tempLevel`: 临时存储当前处理的逻辑层次的 `BasicBlock` 列表。
  - `realTrueBlock1` 和 `realTrueBlock2`: 分别表示 `if` 语句条件为真时跳转的基本块。
  - `realFalseBlock`: 表示 `if` 语句条件为假时跳转的基本块。
  - `endBlock`: 表示 `if` 或 `for` 语句结束后跳转的基本块。
  - `hasCond`: 标记是否存在条件表达式。

方法和功能

- **构造函数**: 一个默认构造函数，用于初始化 `RefillUtil` 实例。
- **refill()**: 核心方法，用于实现条件跳转指令的回填。这个方法根据 `refillBasicBlocks` 中存储的逻辑层次和基本块信息，生成适当的跳转指令（`BrInst`），确保控制流正确地从一个基本块跳转

到另一个基本块。

后端

MipsGenerator

- `irModule`: 存储llvm中间表示的模块。
- `datas`, `texts`, `macros`: 分别存储 `.data`、`.text` 和宏定义部分的 MIPS 代码。
- `spTable`, `basicBlockSpTable`: 分别用于存储栈指针（SP）相关信息和基本块的栈指针信息。
- `mipsFactory`: 用于生成 MIPS 指令的工厂类实例。
- `curSp`: 表示当前的栈指针位置。
- `inMain`, `debug`, `curFunction`, `curBlockNum`, `optimize`: 控制流和生成代码的状态标志。

构造器

- **构造器**(`MipsGenerator`): 初始化并开始访问模块以生成 MIPS 代码。

主要方法

- `visitModule()`: 遍历 IR 模块，初始化数据段和文本段，并遍历所有函数。
- `visitFunction(Function function)`: 处理单个函数，设置函数标签，处理函数内的基本块。
- `visitBasicBlock(BasicBlock basicBlock)`: 遍历并处理基本块中的指令。
- `visitInstruction(Instruction instruction)`: 根据指令类型调用相应的处理方法。
- `visitAllocaInst`, `visitLoadInst`, `visitStoreInst`, 等: 处理不同类型的指令，生成相应的 MIPS 指令。
- `visitAllocaInst` 根据所指向的空间大小分配4×int类型的数量的空间，即sp-=所需分配的空间大小，然后再往下分配一个大小为4的空间，以存放指针。
- `visitLoadInst` 则是先访问符号表，用lw指令先取得地址，再用一条lw指令从地址中取出数值。
- `visitStoreInst` 则是先访问符号表，用lw指令先取得地址，再用一条sw指令把需要存储的值存储到取出的地址。
- `visitBinaryInst` 则是先把操作数访问到两个寄存器上，再通过addiu、subu、mult、div等指令计算出结果，将结果存储到新的内存空间上。
- `visitGetPtr` 先通过访问符号表获得及基地址，再算出偏移与基地址相加，保存在新的内存空间。
- `visitCallInst` 先把前四个参数（如果有）放到a0-a3寄存器，多的参数存到内存，再保存ra寄存器，生成jal指令跳到函数label，再生成恢复ra，恢复内存的指令。

代码优化设计（mips）

我选择做的优化有：

1.删去死函数

在第一次生成中间代码的时候，我统计了每个函数被调用的数量。从而在优化的时候删除被调用次数为0的函数。

```
public void startOptimaze() {
        this.optimaze = true;
        if (!name.equals("main")) {
            if (this.calledNum == 0) {
                //判断为死函数
                //如果不优化是不会判断任何函数为死的
                output = false;
            }
        }
    }
```

2.基本快合并

在中间代码层面，遍历每一个BasicBlock，如果有只含一条无条件跳转到下一个BasicBlock的BrInst，将这个BasicBlock删除，并且把所有跳转到它的指令改为跳转到它的下一个BasicBlock。

```
public void optimazeBlockJump() {
        for (BasicBlock block : this.basicBlocks) {
            if (block.instructions.size() == 1 &&
                    block.instructions.get(0) instanceof BrInst) {
                //只有一条Br指令
                BrInst brInst = (BrInst) block.instructions.get(0);
                int curBlockNum = block.registerNum;
                if (brInst.type == 1 &&
                        ((BasicBlock) brInst.dest).registerNum == curBlockNum +
1) {
                    //无条件跳转到下一个BasicBlock的情况
                    block.delete = true;//优化
                    //以下:标记需要更改的跳转语句
                    Iterator<Map.Entry<BasicBlock, BasicBlock>> iterator =
replaceJumpInstructions.entrySet().iterator();
                    while (iterator.hasNext()) {
                        Map.Entry<BasicBlock, BasicBlock> entry =
iterator.next();
                        //System.out.println("Key = " + entry.getKey() + ",
Value = " + entry.getValue());
                        BasicBlock key = entry.getKey();
                        BasicBlock value = entry.getValue();
                        if (value.equals(block)) {
                            replaceJumpInstructions.put(key, ((BasicBlock)
brInst.dest));
                        }
                    }
                    replaceJumpInstructions.put(block, ((BasicBlock)
brInst.dest));
                    //done
                }
            }
        }
        for (BasicBlock block : this.basicBlocks) {
            if (block.terInst instanceof BrInst) {
                BrInst brInst = (BrInst) block.terInst;
                if (brInst.type == 1) {
                    BasicBlock dest = (BasicBlock) brInst.dest;
                    if (replaceJumpInstructions.containsKey(dest)) {
                        brInst.dest = replaceJumpInstructions.get(dest);
```

```
                    }
                } else {
                    BasicBlock ifTrue = (BasicBlock) brInst.ifTure;
                    if (replaceJumpInstructions.containsKey(ifTrue)) {
                        brInst.ifTure = replaceJumpInstructions.get(ifTrue);
                    }
                    BasicBlock ifFalse = (BasicBlock) brInst.ifFalse;
                    if (replaceJumpInstructions.containsKey(ifFalse)) {
                        brInst.ifFalse = replaceJumpInstructions.get(ifFalse);
                    }
                }
            }
        }
        replaceJumpInstructions = new HashMap<>();
    }
```

3.常量优化

我进行常量优化的基本思想是，如果一条指令的的操作数都是常数，即这条指令的结果可以被计算出来，替换为某个常数，则把以该指令结果为操作数指令中的该指令结果替换为常数。由于替换之后可能回出现新的满足果**一条指令的的操作数都是常数**的指令，因此需要迭代多次，直至没有变化为止。这个优化是跨块的。在`Function`类中实现。

```
 public void optimizeCalculation() {
        int change;
        do {
            change = replaceConstantValue();
            System.out.println("change is " + change);
        } while (change != 0);
    }

    private int replaceConstantValue() {
        int change = 0;
        HashMap<Value, ConstValue> toReplace = new HashMap<>();
        for (BasicBlock block : this.basicBlocks) {
            ArrayList<Instruction> toRemove = new ArrayList<>();
            for (Instruction inst : block.instructions) {
                if (inst instanceof BinaryInst) {
                    BinaryInst binaryInst = (BinaryInst) inst;
                    if (binaryInst.op1 instanceof ConstValue && binaryInst.op2
instanceof ConstValue) {
                        //标记这条Instruction
                        //block.instructions.remove(inst);//?直接remove
                        toRemove.add(inst);
                        int ans = Operator.cal(
                                ((ConstValue) (binaryInst.op1)).getNum(),
                                ((ConstValue) (binaryInst.op2)).getNum(),
                                binaryInst.operator
                        );
                        ConstValue newValue = new
ConstValue(String.valueOf(ans));
                        Value oldResult = binaryInst.result;
                        //要把所有的这条binaryInst删掉
                        //并把其余Instruction里所有的oldResult全部替换为newValue
                        toReplace.put(oldResult, newValue);
                        change++;
```

```
                    }
                }
            }
            for(Instruction inst:toRemove){
                block.instructions.remove(inst);
            }
        }
        for (BasicBlock block : this.basicBlocks) {
            for (Instruction inst : block.instructions) {
                for (Value key : toReplace.keySet()) {
                    inst.replaceValueWithConst(key,toReplace.get(key));
                }
            }
        }
        return change;
    }
```

4.算数优化，主要包含乘法优化

算数优化主要在后端实现，其主要目的是简化计算指令。其中最重要的乘法优化的主要思想为：

- 如果 $d=0$，那就直接不用算，直接得到结果 $p=0$
- 如果 $d=\pm 1$，那也不用算，直接令 $p=a$
- 如果 $d=2^k$，那就直接计算 $p=a<<k$
- 如果 $d=2^k+1$，那就直接计算 $p=(a<<k)+a$
- 如果 $d=2^k-1$，那就直接计算 $p=(a<<k)-a$

实现如下：

```
if (optimize) {
            if (v1 instanceof ConstValue &&
                    v2 instanceof ConstValue) {
                //都为常量
                String reg = "$t0";
                int a = Integer.parseInt(((ConstValue) v1).num);
                int b = Integer.parseInt(((ConstValue) v2).num);
                int ans = Operator.cal(a, b, op);
                mipsFactory.genLi(String.valueOf(ans), reg);
                this.minusSp();
                mipsFactory.genSw("$t0");
                this.spTable.put(result.getMipsName(), new ValuePlace(curSp));
                return;
            } else if ((v1 instanceof ConstValue || v2 instanceof ConstValue)) {
                //有一是常数
                //不一定能成功优化，只有成功优化才返回
                boolean success = false;
                int num;
                //变为t1与num做运算保存到t0
                if (v1 instanceof ConstValue) {
                    num = Integer.parseInt(((ConstValue) v1).num);
                    this.visitValueToReg("$t1", v2);
                } else {
                    num = Integer.parseInt(((ConstValue) v2).num);
                    this.visitValueToReg("$t1", v1);
                }
                //优化上述运算
                String reg = "$t0";
```

```
                if (op == Operator.add) {
                    //加
                    this.texts.add("addiu " + reg + ",$t1," + num + "\n");//可以
少一条li
                    success = true;
                } else if (op == Operator.mul) {
                    if (num == 0) {
                        //*0
                        mipsFactory.genLi("0", reg);
                        success = true;
                    } else if (num == 1) {
                        //*1
                        reg = "$t1";//直接把t1存进去
                        success = true;
                    } else if (num == -1) {
                        //* -1
                        this.texts.add("subu " + reg + ",$zero,$t1\n");
                        success = true;
                    } else if (isPowerOfTwo(num)) {
                        //2的n次方
                        int k = countPowerOfTwo(num);
                        this.texts.add("sll " + reg + "," + "$t1" + "," + k +
"\n");
                        success = true;
                    } else if (isPowerOfTwo(num + 1)) {
                        //2的n次方-1
                        int k = countPowerOfTwo(num + 1);
                        this.texts.add("sll " + reg + "," + "$t1" + "," + k +
"\n");//t0 = t1 << k
                        this.texts.add("subu " + reg + "," + reg + "," + "$t1" +
"\n");//t0 = t0 - t1;
                        success = true;
                    } else if (isPowerOfTwo(num - 1)) {
                        //2的n次方+1
                        int k = countPowerOfTwo(num - 1);
                        this.texts.add("sll " + reg + "," + "$t1" + "," + k +
"\n");//t0 = t1 << k
                        this.texts.add("addu " + reg + "," + reg + "," + "$t1" +
"\n");//t0 = t0 + t1;
                        success = true;
                    }
                } else if (op == Operator.sdiv) {
                    //todo 除法优化
                }
                if (success) {
                    this.minusSp();
                    mipsFactory.genSw(reg);
                    this.spTable.put(result.getMipsName(), new
ValuePlace(curSp));
                    return;
                }
            }
        }
```