参考编译器

参考`kira-rs`，为一用`rust`编写的`SysY`编译器。

其使用`lalrpop`通过约定规则规定语法结构，自动生成词法分析、语法分析程序。

其使用`Koopa-IR`作为中间表示，并为`Koopa-IR`编写了一个`rust`库。

文件结构分为两个部分，分别处理生成ir，生成risc-v，其文件结构如下：

```
.
├── ast.rs
├── codegen
|   ├── builder.rs
|   ├── func.rs
|   ├── gen.rs
|   ├── info.rs
|   ├── mod.rs
|   └── values.rs
├── irgen
|   ├── eval.rs
|   ├── func.rs
|   ├── gen.rs
|   ├── mod.rs
|   ├── scopes.rs
|   └── values.rs
├── main.rs
└── sysy.lalrpop
```

在`main.rs`中调用各个部分的方法，完成编译。首先调用自动生成的`sysy::CompUnitParser`获得`comp_unit`为`ast`根节点，后用`irgen::generate_program(&comp_unit)`得到`program`存储`Koopa-IR`结构，最后用`codegen::generate_asm(&program, &output)`获得最终`risc-v`代码。

编译器总体设计

编译器分为三部分，分别用于生成AST、生成IR、生成mips，故分为三个主要文件夹`astGen`，`irGen`，`mipsGen`，每个模块接口设计如下：

- astGen：输入代码，返回生成的ast的根节点
- irGen：输入一个合法ast的根节点，返回生成的ir结构
- mipsGen：输入ir结构，输出mips代码

除了三个主要文件夹外，为了存储信息、简化代码额外开若干个文件夹如`token`，`ast`，`utils`等。

词法分析设计

编码前设计

总体思路

输入输出使用`Java`自带的支持文件读写的库函数，并同一写在一个文件内，统一提供文件读写服务。

在`Lexer`中存储4个变量，分别为`i`存储当前遍历到的字符下标，`atLine`存储当前遍历到的行，`lineComment`存储当前是否在行注释中，`blockComment`存储当前是否在块注释中，用一个`while`语句块遍历输入字符串，对当前遍历到的字符进行判断并执行添加`Token`和移动下标操作。

同时注意到`Lexer`是纯方法，不需要占内存空间，故为方便直接将所有变量、函数设置为`static`。

注释的处理

在`while`块中首先处理注释。

若当前**在**行注释、列注释中，判断当前是否满足注释结束条件，若满足则将对应标记修改，并继续遍历字符串。

若当前**不在**行注释、列注释中，首先处理注释判断，判断当前是否满足进入注释条件，若满足则修改对应标记，并继续遍历字符串。

单字符可辨认Token的处理

在处理完注释后首先进行单字符可辨认Token的处理，用一个`HashMap`存储`字符<->Token类型`的键值对，用`containsKey`方法判断当前字符是否是单字符可辨认Token，若是则进行添加Token操作。

`HashMap`的声明如下：

```
public static HashMap<Character, TokenType> single = new HashMap<>() {{
    put('+', TokenType.PLUS);
    put('-', TokenType.MINU);
    put('*', TokenType.MULT);
    put('%', TokenType.MOD);
    put(';', TokenType.SEMICN);
    put(',', TokenType.COMMA);
    put('(', TokenType.LPARENT);
    put(')', TokenType.RPARENT);
    put('[', TokenType.LBRACK);
    put(']', TokenType.RBRACK);
    put('{', TokenType.LBRACE);
    put('}', TokenType.RBRACE);
}};
```

双字符可辨认Toekn的处理

有些Token必须要2个字符才能判断是哪个，如`<`和`<=`，采用预读的思路，一次读入当前字符`c`以及下一个字符`nc`，通过判断`c`和`nc`识别是哪个`Token`并进行相应处理。

双字符可辨认token有

```
<  <=
>  >=
=  ==
!  !=
|  ||
&  &&
```

在处理的`|`和`&`时候特殊判断一下，若存在错误则向`ErrHandle`发送错误。

字符常数的处理

若当前字符是`'`则进入字符常数的处理过程。

若`nc=\`则说明是转义字符，将后4位放入Tokens，否则将后3位放入Tokens。

字符串常数的处理

若当前字符是`"`则进入字符串常数的处理过程。

用`while`继续扫描字符串，同时使用`slash`存储上一个字符是否是转义字符，若是则跳过当前字符。当遇到`"`时结束扫描，并将对应区间字符串加入Tokens。

IDENT的处理

若当前字符是`_`或字母，则继续扫描，直到扫描到非IDENT字符结束，并将对应区间字符串加入Tokens。

同时判断一下IDENT是否为保留字，使用`HashMap`存储`保留字<->TokenType`的对应表，用`containsKey`判断是否是保留字并相应处理即可。

`HashMap`的声明如下：

```
public static HashMap<String, TokenType> reserveWords = new HashMap<>() {{
    put("main", TokenType.MAINTK);
    put("const", TokenType.CONSTTK);
    put("int", TokenType.INTTK);
    put("char", TokenType.CHARTK);
    put("break", TokenType.BREAKTK);
    put("continue", TokenType.CONTINUETK);
    put("if", TokenType.IFTK);
    put("else", TokenType.ELSETK);
    put("for", TokenType.FORTK);
    put("getint", TokenType.GETINTTK);
    put("getchar", TokenType.GETCHARTK);
    put("printf", TokenType.PRINTFTK);
    put("return", TokenType.RETURNTK);
    put("void", TokenType.VOIDTK);
}};
```

常数的处理

若当前字符是数字，则继续扫描，直到扫描到非数字字符结束，并将对应区间字符串加入Tokens。

编码完成之后的修改

暂无

语法分析设计

编码前设计

AST结构设计

由于语言语法成分较多，如果对每一个语法成分单独设计一个类，会导致类的数量过多，不利于编码和维护，于是我为AST设计了一个统一的类`ASTNode`，同时存储一个`ASTNodeType`枚举类，用于区分不同的语法成分。注意到AST的叶子节点肯定是一个Token，故在`ASTNodeType`中增加了一个`TOKEN`类型，用于识别叶子节点。

```
public class AstNode {
    public AstType valueType;
    public int line;
    public int lstLine;
    public Token token;
    public ArrayList<AstNode> sons = new ArrayList<>();
}
public enum AstType {
    CompUnit, Decl, ConstDecl, BType, ConstDef, ConstInitVal,
    VarDecl, VarDef, InitVal, FuncDef, MainFuncDef, FuncType,
    FuncFParams, FuncFParam, Block, BlockItem, Stmt, ForStmt,
    Exp, Cond, LVal, PrimaryExp, Number, Character,
    UnaryExp, UnaryOp, FuncRParams, MulExp, AddExp, RelExp,
    EqExp, LAndExp, LOrExp, ConstExp, Token
}
```

Parser设计

Parser设计的总体思路是递归调用对应语法成分的Parse函数，继而构建出AST。由于相似内容较多，故不一一展开，仅阐述不同类型的Parse函数的设计思路。

不需要预读来确定结构的语法成分

对于有确定可分割FIRST集合的语法成分，如`Decl`。对于这类语法的Parse，只需要根据FIRST集合判断当前语法成分应该如何递归，然后递归对应的Parse函数即可。

```
private static AstNode Decl() {
    AstNode node = new AstNode(AstType.Decl, now.line);
    if (now.valueType == TokenType.CONSTTK) {
        node.add(ConstDecl());
    } else {
        node.add(VarDecl());
    }
    return node;
}
```

需要预读来确定结构的语法成分

需要预读来确定结构的语法成分，如`CompUnit`。对于这类语法的Parse，需要根据预读的Token来判断当前语法成分应该如何递归，然后递归对应的Parse函数即可。

`CompUnit`由三部分组成，分别为`Decl`，`FuncDef`，`MainFuncDef`，考察其FIRST集合，分别为`const|int|char`，`void|int|char`，`int`，无法直接判断，故需要预读。注意到`FuncDef`和`MainFuncDef`在之后会有`(`，且`MainFuncDef`会出现`main`关键字，故可以根据这些性质判断分支。

```
private static AstNode CompUnit() {
    AstNode node = new AstNode(AstType.CompUnit, now.line);
    while (now.valueType == TokenType.CONSTTK ||
            ((now.valueType == TokenType.INTTK || now.valueType ==
TokenType.CHARTK) &&
                peek(1).valueType == TokenType.IDENFR &&
                peek(2).valueType != singleType('(')
            )) {
        node.add(Decl());
    }
    while ((now.valueType == TokenType.VOIDTK ||
            now.valueType == TokenType.INTTK ||
            now.valueType == TokenType.CHARTK) &&
            peek(1).valueType == TokenType.IDENFR) {
        node.add(FuncDef());
    }
    node.add(MainFuncDef());
    return node;
}
```

需要改变语法来消除左递归的语法成分

对于存在左递归的语法乘法，如`MulExp`，改变其语法为如下：

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

于是就可以将左递归消除，同时为了建出正确的AST，先用一个`ArrayList`存储下所有的`UnaryExp`，然后再构建AST。

```
private static AstNode form(ArrayList<AstNode> list, AstType valueType) {
    AstNode node = new AstNode(valueType, now.line);
    node.add(list.get(0));
    for (int i = 1; i < list.size(); i += 2) {
        AstNode newNode = new AstNode(valueType, now.line);
        newNode.add(node);
        newNode.add(list.get(i));
        newNode.add(list.get(i + 1));
        node = newNode;
    }
    return node;
}
private static AstNode MulExp() {
    ArrayList<AstNode> list = new ArrayList<>();
    list.add(UnaryExp());
    while (now.valueType == singleType('*') || now.valueType ==
singleType('/') || now.valueType == singleType('%')) {
        list.add(new AstNode(now));
        next();
        list.add(UnaryExp());
    }
    return form(list, AstType.MulExp);
}
```

复杂语法成分的处理

对于`Stmt`语法成分，首先我分析了其FIRST集合，归纳整理如下：

```
语句 Stmt →
  'if' '(' Cond ')' Stmt [ 'else' Stmt ] // j                __if
| 'for' '(' [ForStmt] ';' [Cond] ';' [ForStmt] ')' Stmt      __for
| 'break' ';'                                                __break
| 'continue' ';' // i                                        __continue
| Block                                                      __{
| 'printf''('StringConst {', 'Exp}')'';' // i j              __printf
| 'return' [Exp] ';' // i                                    __return
| [Exp] ';' // i
| LVal '=' 'getint''('')'';' // i j
| LVal '=' 'getchar''('')'';' // i j
| LVal '=' Exp ';' // i                                      __Ident

( Ident IntConst CharConst + - ! { if for break return printf ;
```

在右侧的是该分支的FIRST，可以据此判断出一些分支结构，于是问题就变成了如何判断当前是`Exp`还是`LVal`。经过一点分析，可以发现`Exp`会和`LVal`的FIRST集合产生交集的两种情况是`LVal`或`Ident '(' [FuncRParams] ')'`，可以通过判断`(`将第二种情况排除掉，那么现在必然可以Parse出一个`LVal`，同时可以发现`Exp`中不会出现`=`，于是可以通过判断`=`来判断当前应该是`LVal`还是`Exp`。

由于每个`LVal`最多只会读2遍，故在此回溯不会影响总时间复杂度。同时在预读的时候，将`ErrHandler`关闭，防止加入不该加入的错误。

```
private static AstNode Stmt() {
    ......
    } else if (now.valueType == TokenType.IDENFR && peek(1).valueType != singleType('(')) {
        // Exp begins with Ident in two cases:  Exp -> LVal, Ident '(' [FuncRParams] ')'
        // so if it is not the second case, we can always parse a LVal
        int lstIndex = index;
        ErrHandler.close();
        LVal(); // parse a LVal
        ErrHandler.open();
        Token nextToken = now;
        index = lstIndex;
        now = tokens.get(index);
        if (nextToken.valueType == singleType('=')) {
            // it is actually LVal, aad it into node
            node.add(LVal()); // rerun to make ErrHandler work
            pushToken(node, now); // =
            if (now.valueType == TokenType.GETINTTK || now.valueType == TokenType.GETCHARTK) {
                pushToken(node, now); // getInt / getChar
                pushToken(node, now); // (
                pushToken(node, now, singleType(')'), new ErrInfo(ErrType.j, node.lstLine));
                pushToken(node, now, singleType(';'), new ErrInfo(ErrType.i, node.lstLine));
            } else {
```

```
                node.add(Exp());
                pushToken(node, now, singleType(';'), new ErrInfo(ErrType.i,
node.lstLine));
            }
        } else {
            // it is not LVal but Exp, go back and parse Exp
            // one LVal can be parsed at most two times so it is still O(n)
            node.add(Exp());
            pushToken(node, now, singleType(';'), new ErrInfo(ErrType.i,
node.lstLine));
        }
    } else {
    ......
}
```

编码完成之后的修改

暂无

错误处理设计

使用一个单独的类`ErrHandle`用于接收全流程中识别到的错误，定义`ErrHandle.addError`方法接收错误，用`ArrayList`存储错误，同时支持`ErrHandle.close`和`ErrHandle.open`方法，用于在某些情况下关闭/重新打开错误处理。

同时开一个错误类型`ErrType`类，以及错误信息`ErrInfo`类，结构化存储信息。

```
public class ErrHandler {
    private static ArrayList<ErrInfo> errors = new ArrayList<ErrInfo>();
    private static boolean isOpen = true;

    public static void close() {
        isOpen = false;
    }

    public static void open() {
        isOpen = true;
    }

    public static void addError(ErrInfo error) {
        if (isOpen) {
            errors.add(error);
        }
    }
    ......
}
```

代码生成设计

中间代码生成设计

我使用LLVM作为中间代码，在ir文件夹下实现了ir数据结构，在irGen文件夹的IrGen.java文件中实现了从AST生成LLVM的代码。

中间代码数据结构部分

参考教程中给出的继承结构搭建llvm数据结构，不完全一样，主要继承关系如下：

- Value类为初始类
- GlobalVariable类继承Value
- Function类继承Value
- BasicBlock类继承Value
- User类继承Value
- Instr类继承User
- 各种操作类继承Instr
- 使用Program类代表一个LLVM模块

之后根据程序需要往数据结构中填入变量

中间代码生成

`IrGen.java`中包含生成LLVM所需的所有变量

符号表管理

使用`ArrayList<HashMap<String, Value>>`作为符号表类型，并提供函数作为统一接口

```
private static void enter() {
    scopes.add(new HashMap<>());
}
private static void exit() {
    scopes.remove(scopes.size() - 1);
    if (is_global()) cur_func = null;
}
public static boolean is_global() {
    return scopes.size() == 1;
}
private static void new_value(String name, Value value) {
    scopes.get(scopes.size() - 1).put(name, value);
}
private static Value get_value(String name) {
    for (int i = scopes.size() - 1; i > 0; i--) {
        if (scopes.get(i).containsKey(name)) {
            return scopes.get(i).get(name);
        }
    }
    return scopes.get(0).get(name);
}
```

将instr插入Functino的机制

在instr的构造方法中调用IrGen.java中的new_instr函数，使得instr在创建时可以自动插入Function中，同时创建`isIrGen`变量控制该机制的启动与否。

```
// Instr.java
public Instr(Type type, String name, Value... operands) {
    // name = %id
    super(type, name, operands);
    new_instr(this); // auto add to current basic block
}
// IrGen.java
public static void new_instr(Instr instr) {
    if (!isIrGen) return;
    cur_bb.instrs.add(instr);
    instr.parentBB = cur_bb;
}
```

其他同类需求使用相同方式实现，如

- Function插入全局函数表中
- globalVariable插入全局变量表中
- constString插入Program中
- param插入Function中
- basicblock插入Function中

局部变量名字生成

在`Function.java`中创建一个静态变量`var_cnt`表示当前变量个数，保证所有生成的变量不重名，并提供静态方法作为创建局部变量名字的接口

```
// Function.java
private static int var_cnt = 0;
public String new_var() {
    return "%a" + var_cnt++;
}
```

在需要创建局部变量时只需调用该函数即可。

其他同类需求使用相同方法实现，如：

- 基本块名字生成
- 字符常量名字生成
- 全局变量名字生成

常量表达式求值优化

以`AddExp`为例，当两个操作数都是`ConstInt`类型时，其值是可以确定的，判断该种情况并根据规则求值即可

```
// IrGen.java
public static Value AddExp(AstNode ast) {
    ArrayList<AstNode> sons = flatten(ast, AstType.AddExp);
    Value op1 = MulExp(sons.get(0));
    for (int i = 1; i < sons.size(); i += 2) {
```

```
        Value op2 = MulExp(sons.get(i + 1));
        BinaryOperator.Op op =
            sons.get(i).token.type == TokenType.PLUS ? BinaryOperator.Op.ADD
:
        BinaryOperator.Op.SUB;
        if (op1 instanceof ConstInt && op2 instanceof ConstInt) {
            switch (op) {
                case ADD -> op1 = new ConstInt(((ConstInt) op1).value +
((ConstInt) op2).value);
                case SUB -> op1 = new ConstInt(((ConstInt) op1).value -
((ConstInt) op2).value);
            }
        } else {
            op1 = castTo(op1, INT_TYPE);
            op2 = castTo(op2, INT_TYPE);
            op1 = new BinaryOperator(cur_func.new_var(), op, op1, op2);
        }
    }
    return op1;
}
```

在求其他表达式时也使用同样方法进行优化。这样写还可以自然地处理`ConstExp`。

MIPS代码生成设计

我并没有搭建mips数据结构，而采用了直接输出的方式生成mips代码。

在`mipsGen`文件夹`MipsInfo.java`中储存所有需要的信息，以及提供常用方法的接口。在每个ir数据结构类中实现对应转mips的方法，最后只需调用`Program`中的方法即可生成mips。

Load/Store统一管理

在`MipsInfo.java`中提供load/store接口，统一管理内存操作，方便使用且减少错误概率。

```
// MipsInfo.java
public static Regs loadValue(Value value, Regs reg) {
    if (value instanceof ConstInt) {
        writeln(String.format("    li $%s, %d", reg, ((ConstInt)
value).value));
    } else if (value2reg.containsKey(value.name)) {
        reg = value2reg.get(value.name);
    } else {
        if (!value2offset.containsKey(value.name)) {
            alloc(value.type);
            value2offset.put(value.name, cur_offset);
        }
        load(value.type, reg, value2offset.get(value.name), Regs.sp);
    }
    return reg;
}
public static void storeValue(Value value, Regs reg) {
    if (value2reg.containsKey(value.name)) {
        move(value2reg.get(value.name), reg);
    } else {
        if (!value2offset.containsKey(value.name)) {
```

```
            alloc(value.type);
            value2offset.put(value.name, cur_offset);
        }
        store(value.type, reg, value2offset.get(value.name), Regs.sp);
    }
}
public static void load(Type type, Regs target_reg, int offset, Regs
pointer_reg) {
    if (type.getByte() == 4) {
        writeln(String.format("    lw $%s, %d($%s)", target_reg, offset,
pointer_reg));
    } else {
        writeln(String.format("    lb $%s, %d($%s)", target_reg, offset,
pointer_reg));
    }
}

public static void store(Type type, Regs target_reg, int offset, Regs
pointer_reg) {
    if (type.getByte() == 4) {
        writeln(String.format("    sw $%s, %d($%s)", target_reg, offset,
pointer_reg));
    } else {
        writeln(String.format("    sb $%s, %d($%s)", target_reg, offset,
pointer_reg));
    }
}
```

mips代码的输出

使用`Utils/IO.java`中的`setOut`和`writeln`方法实现统一的输出。

```
// IO.java
public static void setOut(String filename) {
    outputFile = Paths.get(filename);
    try {
        Files.write(outputFile, "".getBytes());
    } catch (java.io.IOException e) {
        e.printStackTrace();
    }
}
public static void writeln(String content) {
    try {
        Files.write(outputFile, (content + '\n').getBytes(),
StandardOpenOption.APPEND);
    } catch (java.io.IOException e) {
        e.printStackTrace();
    }
}
// Compiler.java
IO.setOut(Config.mipsOutputFile);
program.to_mips();
```

编码完成之后的修改

暂无

代码优化设计

我主要实现了以下优化：

- Mem2Reg
- 死代码删除
- RemovePhi
- 图着色分配寄存器
- 乘除法优化
- 窥孔优化
- 指令选择

Mem2Reg

这部分优化主要分为以下几个步骤：

- 简单清理基本块
- 构建CFG
- 求出支配关系
- 求出支配边界
- 插入Phi指令
- 变量重命名

简单清理基本块

清除基本块中第一条跳转指令之后的代码，清理不可达基本块，便于后续处理

```
for (Function function : program.functions) {
    for (BasicBlock bb : function.bbs) {
        ArrayList<Instr> newInstrs = new ArrayList<>();
        for (Instr instr : bb.instrs) {
            newInstrs.add(instr);
            if (instr.isJump()) break;
        }
        bb.instrs = newInstrs;
    }
    ArrayDeque<BasicBlock> queue = new ArrayDeque<>();
    ArrayList<BasicBlock> vis = new ArrayList<>();
    queue.add(function.getEntry());
    while (!queue.isEmpty()) {
        BasicBlock x = queue.poll();
        if (vis.contains(x)) continue;
        vis.add(x);
        Instr last = x.getLastInstr();
        if (last instanceof Branch branch) {
            queue.add(branch.getThenBB());
            queue.add(branch.getElseBB());
        } else if (last instanceof Jump jump) {
            queue.add(jump.getDestBB());
        }
    }
```

```
    function.bbs = vis;
}
```

构建CFG

根据每个基本块最后的跳转指令的目的基本块加边，构建得到控制流图

```
for (Function function : program.functions) {
    for (BasicBlock bb : function.bbs) {
        CFG.put(bb, new HashSet<>());
        rCFG.put(bb, new HashSet<>());
    }
}
for (Function function : program.functions) {
    ArrayDeque<BasicBlock> queue = new ArrayDeque<>();
    ArrayList<BasicBlock> vis = new ArrayList<>();
    queue.add(function.getEntry());
    while (!queue.isEmpty()) {
        BasicBlock x = queue.poll();
        if (vis.contains(x)) continue;
        vis.add(x);
        Instr last = x.getLastInstr();
        if (last instanceof Branch branch) {
            CFGadd(x, branch.getThenBB());
            CFGadd(x, branch.getElseBB());
            queue.add(branch.getThenBB());
            queue.add(branch.getElseBB());
        } else if (last instanceof Jump jump) {
            CFGadd(x, jump.getDestBB());
            queue.add(jump.getDestBB());
        }
    }
}
```

求出支配关系

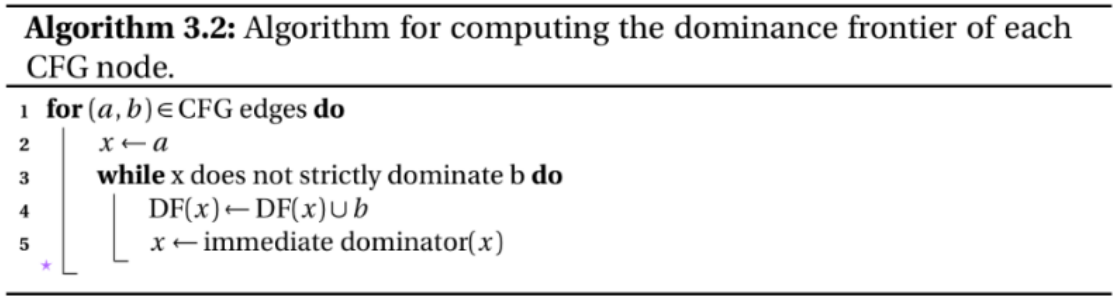
求一个节点时把该节点删除，后从entry开始dfs，能够到达的节点均不被当前点支配，由此求出支配关系

```
for (Function function : program.functions) {
    for (BasicBlock bb : function.bbs) {
        domG.put(bb, new HashSet<>(function.bbs));
        getDomingNodes(function.getEntry(), bb, domG.get(bb));
    }
}
private static void getDomingNodes(BasicBlock now, BasicBlock ban,
HashSet<BasicBlock> doming) {
    if (now == ban) return;
    doming.remove(now);
    for (BasicBlock next : CFG.get(now)) {
        if (doming.contains(next)) {
            getDomingNodes(next, ban, doming);
        }
    }
```

```
}
```

求出支配边界

根据教程中给出的算法求出每个点的支配边界




```java
for (BasicBlock a : CFG.keySet()) {
    for (BasicBlock b : CFG.get(a)) {
        BasicBlock x = a;
        while (!domT.get(x).contains(b) || x.equals(b)) {
            DF.get(x).add(b);
            x = domTfa.get(x);
        }
    }
}
```

插入Phi指令

根据求出的支配边界为每个点对应的支配边界上插入Phi指令

```java
for (Allocate allocate : allocates) {
    HashSet<BasicBlock> F = new HashSet<>();
    HashSet<BasicBlock> W = new HashSet<>(defb.get(allocate));
    while (!W.isEmpty()) {
        BasicBlock x = W.iterator().next();
        W.remove(x);
        for (BasicBlock y : DF.get(x)) {
            if (!F.contains(y)) {
                Phi phi = new Phi(allocate.allocType, function.new_var(),
rCFG.getOrDefault(y, new HashSet<>()));
                phi.parentBB = y;
                y.instrs.add(0, phi);
                phi2allocate.put(phi, allocate);
                F.add(y);
                if (!defb.get(allocate).contains(y)) {
                    W.add(y);
                }
            }
        }
    }
}
```

变量重命名

插入Phi后需要对变量进行重命名，以免破坏SSA形式，在支配树上遍历一遍并做相应操作即可

```
private static void dfs(BasicBlock curBB, HashMap<Phi, Allocate> p2a,
HashMap<Allocate, Value> a2v) {
    vis.add(curBB);
    HashMap<Allocate, Value> cur_a2v = new HashMap<>(a2v);
    ArrayList<Instr> newInstrs = new ArrayList<>();
    for (Instr instr : curBB.instrs) {
        if (instr instanceof Load load) {
            if (load.getPtr() instanceof Allocate allocate &&
cur_a2v.containsKey(allocate)) {
                updateUser(load, cur_a2v.get(allocate));
            } else {
                newInstrs.add(instr);
            }
        } else if (instr instanceof Store store) {
            if (store.getPtr() instanceof Allocate allocate &&
cur_a2v.containsKey(allocate)) {
                cur_a2v.put(allocate, store.getVal());
            } else {
                newInstrs.add(instr);
            }
        } else if (instr instanceof Phi phi) {
            if (p2a.containsKey(phi)) {
                cur_a2v.put(p2a.get(phi), phi);
            }
            newInstrs.add(instr);
        } else if (instr instanceof Allocate) {
            if (!allocates.contains(instr)) {
                newInstrs.add(instr);
            }
        } else {
            newInstrs.add(instr);
        }
    }
    curBB.instrs = newInstrs;
    for (BasicBlock nextBB : CFG.get(curBB)) {
        for (Instr instr : nextBB.instrs) {
            if (instr instanceof Phi phi && p2a.containsKey(phi)) {
                phi.addOperand(curBB, cur_a2v.get(p2a.get(phi)));
            }
        }
    }
    for (BasicBlock nextBB : domT.get(curBB)) {
        assert !vis.contains(nextBB);
        if (vis.contains(nextBB)) continue;
        dfs(nextBB, p2a, cur_a2v);
    }
}
```

死代码删除

我实现了一个简单版本的死代码删除。

判定一个指令是有用的的条件是其有副作用（跳转，会改变内存，调用库函数），于是得到如下判据

```
private static boolean isUseful(Instr instr) {
    return instr instanceof Branch || instr instanceof Jump ||
        instr instanceof Return || instr instanceof Call ||
        instr instanceof IOInstr || instr instanceof Store;
}
```

根据该判据判断所有指令，并以所有有用的指令为起点进行dfs，根据def-use关系标记所有需要的代码，后将不需要的代码删除

```
public static void run(Program program) {
    for (Function function : program.functions) {
        vis.clear();
        for (BasicBlock bb : function.bbs) {
            for (Instr instr : bb.instrs) {
                if (isUseful(instr)) {
                    dfs(instr);
                }
            }
        }
        for (BasicBlock bb : function.bbs) {
            bb.instrs.removeIf(instr -> !vis.contains(instr));
        }
    }
}
private static void dfs(Instr instr) {
    if (vis.contains(instr)) return;
    vis.add(instr);
    for (Value operand : instr.operands) {
        if (operand instanceof Instr) {
            dfs((Instr) operand);
        }
    }
}
```

RemovePhi

我使用的方法和教程中介绍的方法并不完全一样。

首先预处理所有需要删除的Phi指令以及需要添加的Move指令及对应基本块，记录完成后就可以把基本块中的所有Phi消除

```
HashSet<Phi> phis = new HashSet<>();
HashMap<BasicBlock, HashSet<Move>> movesToAdd = new HashMap<>();
for (BasicBlock bb : function.bbs) {
    for (Instr instr : bb.instrs) {
        if (instr instanceof Phi phi) {
            phis.add(phi);
        }
```

```
    }
}
function.bbs.forEach(bb -> movesToAdd.put(bb, new HashSet<>()));
for (Phi phi : phis) {
    for (BasicBlock bb : phi.preBBs.keySet()) {
        movesToAdd.get(bb).add(new Move(phi, phi.preBBs.get(bb)));
    }
}
for (BasicBlock bb : function.bbs) {
    bb.instrs = new ArrayList<>(bb.instrs.stream().filter(instr -> !(instr
instanceof Phi)).toList());
}
```

遍历所有需要添加move指令的基本块，单独处理每个基本块。对于一个基本块，首先建立move之间的赋值图，每次取出不会再使用的一个变量进行赋值，若没有符合条件的变量则创造一个新变量消除环路，使过程能够继续进行

```
for (BasicBlock bb : movesToAdd.keySet()) {
    HashSet<Move> moves = movesToAdd.get(bb);
    HashMap<Value, Integer> out = new HashMap<>();
    moves.forEach(move -> {
        if (out.containsKey(move.getSource())) {
            out.put(move.getSource(), out.get(move.getSource()) + 1);
        } else {
            out.put(move.getSource(), 1);
        }
        if (!out.containsKey(move.getTarget())) {
            out.put(move.getTarget(), 0);
        }
    });
    while (!moves.isEmpty()) {
        HashSet<Move> toRemove = new HashSet<>();
        for (Move move : moves) {
            if (out.get(move.getTarget()) == 0) {
                bb.instrs.add(bb.instrs.size() - 1, move);
                toRemove.add(move);
                out.put(move.getSource(), out.get(move.getSource()) - 1);
            }
            if (move.getTarget().equals(move)) {
                toRemove.add(move);
                out.put(move.getSource(), out.get(move.getSource()) - 1);
            }
        }
        moves.removeAll(toRemove);
        if (toRemove.isEmpty()) {
            Move curMove = moves.iterator().next();
            Phi tempReg = new Phi(curMove.getTarget().type,
function.new_var(), new HashSet<>());
            Move newMove = new Move(tempReg, curMove.getSource());
            bb.instrs.add(bb.instrs.size() - 1, newMove);
            out.put(curMove.getTarget(), 0);
            for (Move move : moves) {
                if (move.getSource().equals(curMove.getTarget())) {
                    move.replaceSource(tempReg);
                }
```

```
            }
        }
    }
}
```

图着色寄存器分配

使用图着色算法为变量分配寄存器，为了实现简单，对除了全局变量及数组的所有变量分配寄存器，具体的判断依据为

```
private static boolean canAssignReg(Value value) {
    if (value instanceof Instr instr) {
        return !(instr instanceof Return || instr instanceof Branch ||
                 instr instanceof Jump || instr instanceof Store ||
                 (instr instanceof Call && instr.type.equals(VOID_TYPE)) ||
                 instr instanceof IOInstr.PutString ||
                 instr instanceof IOInstr.PutChar ||
                 instr instanceof IOInstr.PutInt);
    } else {
        return value instanceof FuncParam;
    }
}
```

可用寄存器集合设置为`gp, fp, t0~t9, s0~s7`

```
private static ArrayList<Regs> getRegs() {
    return new ArrayList<>(){{
        add(Regs.gp); add(Regs.fp);
        add(Regs.t0); add(Regs.t1); add(Regs.t2); add(Regs.t3); add(Regs.t4);
add(Regs.t5); add(Regs.t6); add(Regs.t7); add(Regs.t8); add(Regs.t9);
        add(Regs.s0); add(Regs.s1); add(Regs.s2); add(Regs.s3); add(Regs.s4);
add(Regs.s5); add(Regs.s6); add(Regs.s7);
    }};
}
```

活跃变量分析

以指令为单位进行活跃变量分析，初始对每条指令分析其def和use集合，后使用迭代算法求出每条指令的in和out集合

```
public static void activeAnalyse(Function function) {
    MipsInfo.act_flag = true;
    for (BasicBlock bb : function.bbs) {
        for (Instr instr : bb.instrs) {
            Iuses.put(instr, getUse(instr));
            Idefs.put(instr, getDef(instr));
        }
    }
    boolean changed = true;
    while (changed) {
        changed = false;
        for (BasicBlock bb : function.bbs) {
            for (int i = bb.instrs.size() - 1; i >= 0; i--) {
```

```
                Instr instr = bb.instrs.get(i);
                HashSet<String> new_act_out = new HashSet<>();
                if (instr instanceof Branch branch) {

new_act_out.addAll(branch.getElseBB().instrs.get(0).act_in);

new_act_out.addAll(branch.getThenBB().instrs.get(0).act_in);
                } else if (instr instanceof Jump jump) {

new_act_out.addAll(jump.getDestBB().instrs.get(0).act_in);
                } else if (instr instanceof Return) {

                } else {
                    new_act_out.addAll(bb.instrs.get(i + 1).act_in);
                }
                HashSet<String> new_act_in = new HashSet<>(new_act_out);
                new_act_in.removeAll(Idefs.get(instr));
                new_act_in.addAll(Iuses.get(instr));
                if (!new_act_in.equals(instr.act_in) ||
                    !new_act_out.equals(instr.act_out)) {
                    changed = true;
                }
                instr.act_in = new_act_in;
                instr.act_out = new_act_out;
            }
        }
    }
}
private static HashSet<String> getUse(Instr instr) {
    HashSet<String> res = new HashSet<>();
    for (Value value : instr.operands) {
        if (value instanceof Instr || value instanceof FuncParam) {
            res.add(value.name);
        }
    }
    return res;
}
private static HashSet<String> getDef(Instr instr) {
    HashSet<String> res = new HashSet<>();
    if (canAssignReg(instr)) res.add(instr.name);
    return res;
}
```

构建冲突图

判定两个变量冲突的标准是：

- 同一个out集合中互相冲突
- 同一个in集合中互相冲突
- 某一变量与定义指令的out集合中所有变量冲突

由此得到以下代码

```
for (BasicBlock bb : function.bbs) {
    for (Instr instr : bb.instrs) {
```

```
        for (String x : instr.act_out) {
            for (String y : instr.act_out) {
                if (x.equals(y)) continue;
                G.get(x).add(y);
                G.get(y).add(x);
            }
        }
        for (String x : instr.act_in) {
            for (String y : instr.act_in) {
                if (x.equals(y)) continue;
                G.get(x).add(y);
                G.get(y).add(x);
            }
        }
        for (String x : Idefs.get(instr)) {
            for (String y : instr.act_out) {
                if (x.equals(y)) continue;
                G.get(x).add(y);
                G.get(y).add(x);
            }
        }
    }
}
```

分配过程

我实现了一个简易的分配策略，主要分为4个过程：

- 简化小度数节点
- 尝试合并move指令两端变量
- 排除大度数节点
- 由栈得到最终分配

简化小度数节点

对于冲突图中度数小于寄存器个数的节点直接移除，并放入栈中，表示可以分配

```
while (true) {
    boolean flg = false;
    ArrayList<String> iter = new ArrayList<>(G.keySet());
    for (String value : iter) {
        if (G.get(value).size() < regs.size()) {
            changed = true;
            flg = true;
            stack.add(value);
            stackG.add(new HashSet<>(G.get(value)));
            remove(value);
        }
    }
    if (!flg) break;
}
```

尝试合并move指令两端变量

对于一个move指令两端变量，若合并后节点度数小于寄存器个数则合并，同时存储合并变量信息，否则不合并

```
while (true) {
    boolean flg = false;
    ArrayList<Pair<String, String>> iter = new ArrayList<>(RegAlloc.moves);
    for (Pair<String, String> move : iter) {
        String x = move.first;
        String y = move.second;
        HashSet<String> union = new HashSet<>(G.get(x));
        union.addAll(G.get(y));
        if (union.size() < regs.size()) {
            changed = true;
            flg = true;
            same.get(x).addAll(same.get(y));
            for (String z : G.get(y)) {
                G.get(z).remove(y);
                G.get(z).add(x);
                G.get(x).add(z);
            }
            remove(y);
        }
    }
    if (!flg) break;
}
```

排除大度数节点

当无法继续删除节点时排除大度数节点，为进一步分配寄存器创造空间

```
if (!G.isEmpty()) {
    String spill = null;
    for (String value : G.keySet()) {
        if (spill == null || G.get(value).size() > G.get(spill).size()) {
            spill = value;
        }
    }
    spilled.add(spill);
    stack.add(spill);
    stackG.add(new HashSet<>(G.get(spill)));
    remove(spill);
}
```

其中使用了封装好的`remove`方法

```
private static void remove(String x) {
    for (String y : G.get(x)) {
        G.get(y).remove(x);
    }
    G.remove(x);
    HashSet<Pair<String, String>> del = new HashSet<>();
    for (Pair<String, String> move : moves) {
        if (move.first.equals(x) || move.second.equals(x)) {
            del.add(move);
        }
    }
    moves.removeAll(del);
}
```

分配寄存器

反向遍历得到的栈，每次根据冲突边为变量分配寄存器，同时注意到被排除的大度数节点可能仍然可以分配寄存器，该部分在其余变量分配完毕后处理

```
for (int i = stack.size() - 1; i >= 0; i--) {
    String value = stack.get(i);
    HashSet<String> adj = stackG.get(i);
    if (spilled.contains(value) || function.value2reg.containsKey(value))
continue;
    ArrayList<Regs> validRegs = new ArrayList<>(regs);
    for (String v : adj) {
        if (function.value2reg.containsKey(v)) {
            validRegs.remove(function.value2reg.get(v));
        }
    }
    function.value2reg.put(value, validRegs.get(0));
}
for (int i = stack.size() - 1; i >= 0; i--) {
    String value = stack.get(i);
    HashSet<String> adj = stackG.get(i);
    if (!spilled.contains(value) || function.value2reg.containsKey(value))
continue;
    ArrayList<Regs> validRegs = new ArrayList<>(regs);
    for (String v : adj) {
        if (function.value2reg.containsKey(v)) {
            validRegs.remove(function.value2reg.get(v));
        }
    }
    if (!validRegs.isEmpty()) {
        function.value2reg.put(value, validRegs.get(0));
    }
}
```

为在move合并过程中合并的变量分配寄存器

```
for (String value : same.keySet()) {
    if (function.value2reg.containsKey(value)) {
        for (String v : same.get(value)) {
            function.value2reg.put(v, function.value2reg.get(value));
        }
    }
}
```

乘除法优化

由于我没有实现mips数据结构，故我选择直接在输出时进行乘除法优化。

乘法优化只优化一个数为常数且为2的幂次的情况，这种情况下将乘法优化为左移操作降低运算强度。

在除法优化方面，根据《Division by Invariant Integers using Multiplication》论文中给出的方法实现除法优化

Inputs: **sword** d and n, with d constant and $d \neq 0$.
udword m;
int ℓ, sh_{post};
$(m, sh_{\text{post}}, \ell) = \text{CHOOSE\_MULTIPLIER}(|d|, N-1)$;
if $|d| = 1$ **then**
  Issue $q = d$;
else if $|d| = 2^{\ell}$ **then**
  Issue $q = \text{SRA}(n + \text{SRL}(\text{SRA}(n, \ell-1), N-\ell), \ell)$;
else if $m < 2^{N-1}$ **then**
  Issue $q = \text{SRA}(\text{MULSH}(m, n), sh_{\text{post}}) - \text{XSIGN}(n)$;
else
  Issue $q = \text{SRA}(n + \text{MULSH}(m - 2^N, n), sh_{\text{post}}) - \text{XSIGN}(n)$;
  Cmt. Caution — $m - 2^N$ is negative.
end if

if $d < 0$ **then**
  Issue $q = -q$;
end if

procedure CHOOSE_MULTIPLIER(**uword** d, **int** $prec$);
Cmt. d – Constant divisor to invert. $1 \le d < 2^N$.
Cmt. $prec$ – Number of bits of precision needed, $1 \le prec \le N$.
Cmt. Finds m, sh_{post}, ℓ such that:
Cmt. $2^{\ell-1} < d \le 2^{\ell}$.
Cmt. $0 \le sh_{\text{post}} \le \ell$. If $sh_{\text{post}} > 0$, then $N + sh_{\text{post}} \le \ell + prec$.
Cmt. $2^{N+sh_{\text{post}}} < m * d \le 2^{N+sh_{\text{post}}} * (1 + 2^{-prec})$.
Cmt. Corollary. If $d \le 2^{prec}$, then $m < 2^{N+sh_{\text{post}}} * (1 + 2^{1-\ell})/d \le 2^{N+sh_{\text{post}}-\ell+1}$.
Cmt. Hence m fits in $\max(prec, N-\ell)+1$ bits (unsigned).
Cmt.
int $\ell = \lceil \log_2 d \rceil$, $sh_{\text{post}} = \ell$;
udword $m_{\text{low}} = \lfloor 2^{N+\ell}/d \rfloor$, $m_{\text{high}} = \lfloor (2^{N+\ell} + 2^{N+\ell-prec})/d \rfloor$;
Cmt. To avoid numerator overflow, compute m_{low} as $2^N + (m_{\text{low}} - 2^N)$.
Cmt. Likewise for m_{high}. Compare m' in Figure 4.1.
Invariant. $m_{\text{low}} = \lfloor 2^{N+sh_{\text{post}}}/d \rfloor < m_{\text{high}} = \lfloor 2^{N+sh_{\text{post}}} * (1 + 2^{-prec})/d \rfloor$.
while $\lfloor m_{\text{low}}/2 \rfloor < \lfloor m_{\text{high}}/2 \rfloor$ **and** $sh_{\text{post}} > 0$ **do**
$m_{\text{low}} = \lfloor m_{\text{low}}/2 \rfloor$; $m_{\text{high}} = \lfloor m_{\text{high}}/2 \rfloor$; $sh_{\text{post}} = sh_{\text{post}} - 1$;
end while; /* Reduce to lowest terms. */
return (m_{high}, sh_{post}, ℓ); /* Three outputs. */
end CHOOSE_MULTIPLIER;

```
1  private static void do_div(Regs target, Regs src, int d) {
2      MagicNumber res = choose_multiplier(Math.abs(d), N - 1);
3      if (Math.abs(d) == 1) {
4          move(target, src);
5      } else if (Math.abs(d) == (1L << res.l)) {
6          writeln(String.format("    sra $%s, $%s, %d", Regs.k1, src, res.l -
   1));
7          writeln(String.format("    srl $%s, $%s, %d", Regs.k1, Regs.k1, N -
   res.l));
8          writeln(String.format("    addu $%s, $%s, $%s", target, src,
   Regs.k1));
9          writeln(String.format("    sra $%s, $%s, %d", target, target,
   res.l));
10     } else {
11         if (res.M < (1L << (N - 1))) {
12             writeln(String.format("    li $%s, %d", Regs.k1, res.M));
13             writeln(String.format("    mult $%s, $%s", src, Regs.k1));
14             writeln(String.format("    mfhi $%s", Regs.k1));
15             writeln(String.format("    sra $%s, $%s, %d", Regs.k1, Regs.k1,
   res.sh_post));
16         } else {
17             writeln(String.format("    li $%s, %d", Regs.k1, res.M - (1L <<
   N)));
18             writeln(String.format("    mult $%s, $%s", src, Regs.k1));
19             writeln(String.format("    mfhi $%s", Regs.k1));
20             writeln(String.format("    addu $%s, $%s, $%s", Regs.k1,
   Regs.k1, src));
21             writeln(String.format("    sra $%s, $%s, %d", Regs.k1, Regs.k1,
   res.sh_post));
22         }
23         writeln(String.format("    srl $%s, $%s, %d", Regs.v1, src, N - 1));
24         writeln(String.format("    addu $%s, $%s, $%s", target, Regs.k1,
   Regs.v1));
```

```
    }
    if (d < 0) {
        writeln(String.format("    subu $%s, $0, $%s", target, target));
    }
}
```

窥孔优化

通过观察我发现生成出来的代码中具有大量

```
sw $t0, 0($sp)
lw $t0, 0($sp)
```

结构，而这种情况下把lw命令去掉不会影响程序正确性，故我在`IO.java`中特判了这一类情况，实现了优化

```
public static String last = "";
public static void writeln(String content) {
    if (Config.taskType == Config.TaskType.MIPS
        && ((last.startsWith("    sw") && content.startsWith("    lw"))
            || (last.startsWith("    sb") && content.startsWith("    lb")))
        && last.substring(6).equals(content.substring(6))) {
        return;
    }
    if (!content.startsWith("    #")) {
        last = content;
    }
    try {
        Files.write(outputFile, (content + '\n').getBytes(),
StandardOpenOption.APPEND);
    } catch (java.io.IOException e) {
        e.printStackTrace();
    }
}
```

指令选择

指令选择方面我主要做了以下几种：

- subiu改为addiu
- sle换为slt与xori
- sge换为sgt与xori

编码完成之后的修改

暂无