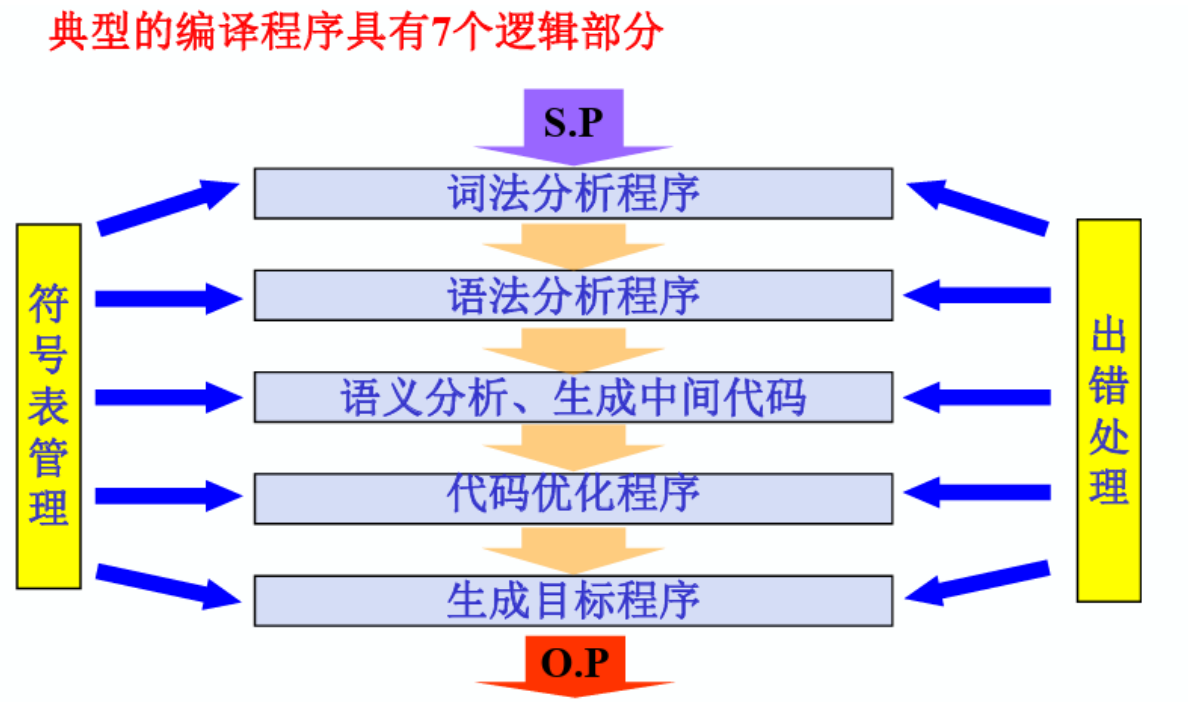
编译设计文档

author:21373405 周靖宇

总览

总体结构

我的编译器主要分为五个部分：词法分析，语法分析，语义解释与中间代码生成，代码优化和目标代码生成。基本按照了理论课程的要求，符号表技术和错误处理技术贯穿了我的每个阶段，最终完成了`llvm`作为中间代码，`MIPS`作为目标代码的编译器。



文件结构

```
├── BackEnd
│   └── MIPS
│       ├── Assembly
│       ├── MipsController.java
│       ├── PopMarco.java
│       ├── PushMarco.java
│       └── Register.java
├── Compiler.java
├── Config
├── Enums
├── FrontEnd
│   ├── ErrorManager
│   ├── Lexer
│   ├── Nodes
│   ├── Parser
│   ├── Symbol
│   └── SyntaxError.java
├── MidEnd
│   ├── ActAnalysis.java
```

```
|   ├── CFGBuilder.java
|   ├── De_SSA.java
|   ├── DeadCodeDeletion.java
|   ├── FuncInline.java
|   ├── GVN_GCM.java
|   ├── GlobalForInline.java
|   ├── MergeBlock.java
|   ├── Optimizer.java
|   ├── RegAllocator.java
|   ├── RegAllocatorForSSA.java
|   ├── RegDispatcher.java
|   └── SSABuilder.java
└── llvm_ir
    ├── IRController.java
    ├── Module.java
    ├── Use.java
    ├── User.java
    ├── Value.java
    ├── Values
    |   ├── BasicBlock.java
    |   ├── ConstBool.java
    |   ├── ConstInteger.java
    |   ├── Function.java
    |   ├── GlobalVar.java
    |   ├── InlinedFunc.java
    |   ├── Instruction
    |   └── llvmType
```

上述为笔者的编译器架构，其中`FrontEnd`主要包括了前端的词法分析和语法分析器，`Config`包括了编译器的具体选项，`llvm_ir`主要包括了中间代码生成的主要组件，包括了`IRController，Value`等文件，`MidEnd`主要包括了中端优化的各种优化，由`Optimizer`进行统一调控。`BackEnd`主要包括了生成`MIPS`所需要的依赖。

接口设计

和理论课需要的结构类似，词法分析输出一个`token`序列，语法分析输出一颗抽象语法树，中间代码生成器通过对语法树进行前序遍历进行文法翻译，文法翻译得到一颗中间代码树，优化器对其进行优化，最终通过后端遍历中间代码树进行目标代码生成。

词法分析

词法分析是编译器遇到的第一个任务，其难度并不高，关键点在于对不同种类`token`的识别优先级。

我的词法分析基本的思想就是根据每个词法`token`的优先级来不断识别`token`直到文段结束。

核心就是对`Token`的识别。

```java
public class Token {

    private final String value;

    private tokenType type;

    private final int line;
```

```
    public Token(String token, tokenType type, int line) {
        this.value = token;
        this.type = type;
        this.line = line;
    }
}
```

我的`token`的识别优先级大约是**跳过空格 > 注释> " > ident > Integer > 固定标识符**，其中固定标识符按照最远匹配法则进行匹配。大概的代码如下：

```
    public Token nextToken() {
        Pattern patternIdentHead = Pattern.compile("[a-zA-Z_]");
        String symbolsHead = "!|&+-*/%<>=;,()[]{}";
        skipSpace();
        while (hasNext()) {
            if (nextString(2).equals("//") || nextString(2).equals("/*")) {
                skipComment();
            } else if (source.charAt(pos) == '"') {
                return recognizeFormatString();
            } else if (patternIdentHead.matcher(nextString(1)).matches()) {
                return recognizeIdent();
            } else if (isIntCon(nextString(1))) {
                return recognizeIntCon();
            } else if (symbolsHead.contains(source.substring(pos, pos + 1))) {
                return recognizeSymbol();
            } else skipSpace();
        }
        return null;
    }
```

这样就可以将字符转化为一个个固定的`token`，以`tokenstram`的形式参与到语法分析中。

我的语法分析即没有用到高级的`LEX`知识，也没有自己去画`DFA`或者`NFA`归根到底还是因为自己在写着一部分的时候仍是知识浅陌，把这玩意当作了一个简单的字符串游戏，没有很好的体现状态转移的思想。。

修改

基本上一气呵成，无太大修改

语法分析

我的`Parser`的功能是通过递归子程序法解析构建语法树。思路清晰，难点在于将语法成分抽象为抽象语法树。

递归子程序法

我的递归子程序法采用了代价最高，最没有技巧性的递归下降方法，没有使用超前扫描，而是通过回溯的方式进行试错，具体的流程大概是：

1. 递归调用子程序进行解析，如果成分不符合文法需求，则将其赋为`ILLEGAL`类型
2. 如果解析出来的结点类型为`ILLEGAL`则进行回溯
3. 如果最终可以得到期望的子结点类型，则构建一个相应类型的抽象语法树结点。

主要的核心方法模式见下：

```
private void Parse_XXX() {
    ArrayList<Node> children = new ArrayList<>();
    Node n = Parse_xxx();
    if (n.getType() != SyntaxVarType.ILLEGAL) children.add(n);//移进
    else unread(n.getSize()); //回溯
    //构建XXX结点
}
```

消除左递归

左递归问题是`buaasysy`文法最大的问题之一，对于左递归问题，我采用的方法是最常用的方法拆除左递归，然后重新构建新的子节点。

比如对于文法

```
AddExp -> AddExp (+ | -) MulExp | MulExp
//将其改为
AddExp -> {MulExp}(+ | -) MulExp
```

为了满足题目的要求，我会把解析出来的前n-1个`MulExp`合并成`n-1`个`AddExp`

即$MulExp_1 \rightarrow AddExp_1, AddExp_1 + MulExp_2 \rightarrow AddExp_2$，以此类推。

这样更多是针对题目的妥协。。不好地方是显而易见的，语法树的深度会大大增加，对于之后的优化也并不方便。但是好处是方便生成中间代码。

`Stmt`分流

`Stmt`有太多种了，去这样混淆各个语法成分百害而无一利，我在`Stmt`的基础上进行了进一步分流：

```
AssignStmt,BlockStmt,BrakStmt,ContinueStmt,ExpStmt,ForLoopStmt,ForStmt,GetintStmt
,IfStmt,PrintfStmt,ReturnStmt
```

在输出上仍然是按照`Stmt`输出，在之后的生成代码上降低了工作量。

FIRST集合冲突

在消除了左递归之后，我发现对于`LVal`和`Exp`两种结点的`FIRST`集合仍然有冲突，我的选择是优先级选择。

首先解析`LVal`使用更深的读取，若读取到了`=`则其语法成分为`AssignStmt`，不成立时再进行对`Exp`的解读。

结点结构&生成抽象语法树

我的结点主要分为两种，语法成分结点和词法成分节点，词法成分节点结点在语法树种只以子节点的形式存在。通过工厂模式实现生成抽象语法树。具体实现见下

```
public class NodeGenerator {
    public static Node generateNode(SyntaxVarType type, ArrayList<Node> children)
{
        return switch (type) {
```

```
            case Block -> new Block(type, children);
            case VarDecl -> new VarDecl(type, children);
            case ConstDecl -> new ConstDecl(type, children);
            case FuncDef -> new FuncDef(type, children);
            case MainFuncDef -> new MainFuncDef(type, children);
            case FuncFParams -> new FuncFParams(type, children);
            case FuncFParam -> new FuncFParam(type, children);
            case FuncRParams -> new FuncRParams(type, children);
            case FuncType -> new FuncType(type, children);
            case VarDef -> new VarDef(type, children);
            case ConstDef -> new ConstDef(type, children);
            case ConstInitVal -> new ConstInitVal(type, children);
            case Stmt -> new Stmt(type, children);
            case LVal -> new LVal(type, children);
            case Cond -> new Cond(type, children);
            case ForStmt -> new ForStmt(type, children);
            case AddExp -> new AddExp(type, children);
            case LOrExp -> new LOrExp(type, children);
            case PrimaryExp -> new PrimaryExp(type, children);
            case Number -> new Number(type, children);
            case IntConst -> new IntConst(type, children);
            case UnaryExp -> new UnaryExp(type, children);
            case UnaryOp -> new UnaryOp(type, children);
            case MulExp -> new MulExp(type, children);
            case RelExp -> new RelExp(type, children);
            case EqExp -> new EqExp(type, children);
            case LAndExp -> new LAndExp(type, children);
            case CompUnit -> new CompUnit(type, children);
            case Decl -> new Decl(type, children);
            case BlockItem -> new BlockItem(type, children);
            case Exp -> new Exp(type, children);
            case BType -> new BType(type, children);
            case InitVal -> new InitVal(type, children);
            case ConstExp -> new ConstExp(type, children);
            case ILLEGAL -> new Node(type, children);
            case AssignStmt -> new AssignStmt(type, children);
            case ExpStmt -> new ExpStmt(type, children);
            case IfStmt -> new IfStmt(type, children);
            case ForLoopStmt -> new ForLoopStmt(type, children);
            case BreakStmt -> new BreakStmt(type, children);
            case ContinueStmt -> new ContinueStmt(type, children);
            case ReturnStmt -> new ReturnStmt(type, children);
            case PrintfStmt -> new PrintfStmt(type, children);
            case BlockStmt -> new BlockStmt(type, children);
            case GetIntStmt -> new GetIntStmt(type, children);
            default -> null;
        };
    }

    public static TokenNode generateToken(Token token, int pos) {
        return new TokenNode(SyntaxVarType.TOKEN, token, pos);
    }
}
```

修改

在语法分析之后，我对我的整个项目进行了修改，分离了`Config`软件包，用以存储不同的编译选项，生成不同的输出，提高了编译器的可读性与可使用性。

错误处理

错误处理主要分为两种语法错误和语义错误。

对于语法错误，我们的处理是保留改语法成分，并不按照`ILLEGAL`类型的结点存下来。这是比较简单的部分。

对于语义错误，比较复杂，其在语法处理的过程中没有出现问题，为了解析这个问题，我们需要引入符号表技术。而对语法成分的检查则是通过对抽象语法树的前序遍历解决。

符号表技术

我的符号表按照单例模式进行调用。

采用`Map`数据结构完成从函数到符号表的映射。对于每一个变量或函数，构建对应的符号，插入到符号表中。各个函数的符号表按照栈的形式存放在符号表中。

```
public class SymbolManager {

    private static final SymbolManager instance = new SymbolManager(); // 采取单例
模式
    private final Stack<SymbolTable> symbolTableStack;

    private final HashMap<String, SymbolTable> funcMap;
}
```

```
public class SymbolTable {

    private final HashSet<Symbol> symbols;

    private final FuncSymbol funcSymbol;

    private final HashMap<String, Symbol> symbolMap;
}
```

具体寻找错误

我的错误处理是通过对抽象语法树的前序遍历实现的，在需要进行检查的部分，进行具体的处理。

```
public class Node {

    public void checkError() {
        for (Node child : children) child.checkError();
    }
}
```

只需要在对应的结点进行方法重写即可。

修改

无

中间代码生成(`llvm`)

`llvm`作为一种成熟的中间代码，大大降低了我们生成目标代码的难度，也加深了我对基本快等基本概念的理解，是编译过程中最精华的部分之一。

`llvm`设计框架

`llvm`的翻译和编译理论中的属性翻译文法有一定的相像之处，需要同时自上而下和自下而上同时进行。而`llvm`中有一切皆`value`的思想，我也沿用了这一思想，设计了$value \leftarrow user \leftarrow instr$的结构，使用`IRController`的单例模式作为全局代码生成的辅助工具，通过遍历语法树来生成`llvm`

```
public class Node {
        public Value genLLVMir() {
        for (Node child : children) {
            child.genLLVMir();
        }
        return null;
    }
}
```

实现方法和属性文法翻译中的继承属性和综合属性有所相似，通过遍历子节点构建综合属性，子节点之间调用前驱的属性进行继承属性的文法翻译。生成结点只会会对符号表进行查表，将对应的`llvmValue`插入到符号表中。

新的符号表技术

在遍历语法树的过程中，我将生成的`llvm`插入到符号表对应的`Symbol`中，使得只需要调用符号表就可以得到相应的`VALUE`

构建use边

在生成`llvm`的过程中，可以直接构建`use`关系方便之后的优化。

短路求值

短路求值是`llvm`生成中较为困难的一个问题，我的解决方法是在进行判断之前就提前生成`thenBlock`,`elseBlock`,`nextBlock`，在处理`LOrExp`和`LAndExp`中递归处理生成对应的语句，可以参考Compilify

除此之外还有一种处理思路

```
if (a && b) {
    stmt
}

..
if (a) {
    if (b) {
        stmt
    }
}
if (a || b) {

}
//可以处理为
if (a) {
    stmt
} else {
    if (b) {
        stmt
    }
}
```

但是思路较为冗余，生成很多冗余代码，不多阐述。

GEP

`GEP`指令是`llvm`中最复杂的指令之一，对于这个指令的理解，我的大概理解是

```
GEP a i1 i2 = *(a+i1)+i2
GEP a i1 = a+i1
```

正确理解gep的含义之后，对于数组的操作也变得简单起来

由于在`llvm`中`alloca`指令得到的结果是一个指向某块内存的指针。我们需要做的就是分类讨论。

若数组不是函数参数，则我们可以拿到的是存取这个数组的指针`p`

1. 若其为一维数组p[i1]

   ```
   p1 = GEP p 0 i1;
   p2 = LOAD p1;
   ```

   p2即为所求
2. 若其为二维数组`p[i1][j1]`

```
p1 = GEP p 0 i1;
p2 = GEP p1 0 j1;
p3 = LOAD p2;
```

p3即为所求

若数组是参数

1. 若其为一维数组，则传入的参数为`int*`，假设我们需要求的是`p[i1]`

```
p1 = LOAD p;//相当于GEP p 0 0
p2 = GEP p1 i1;// *(p)+i1
p3 = LOAD p2; // *(*(p) + i1)
```

2. 若其为二维数组，传入参数类型为`int []*`,假设我们需要的是`p[i][j]`

```
p1 = LOAD p;//相当于GEP p 0 0
p2 = GEP p1 i1 j1;// *(*(p)+i1) + j1
p3 = LOAD p2; // *(*(*(p)+i1) + j1)
```

修改

这部分的修改其实很多，我至少经历了一到两次的重构

在设计的初期，我认为只需要将对应`value`的名字传入就可以完成。事实上，如果只是为了简单的生成`llvm`的话，这种方法也是完全可行的，但是我把一件非常重要的事情搞错了，就是`llvmValue`之间有着非常强大的`use`关系链，如果我只将名字作为生成的指标的话会导致`value`之间缺乏联系，在之后的`MIPS`和优化中都会非常麻烦，因此我对这一部分，即使在通过了代码生成2之后仍然进行了大规模的重构。同时为了保证相隔很远的对于同一个`value`的使用是正确的，我对符号表也进行了更新，插入了对应的`value`。

第二次重构源于我对`llvm`的不理解，在这次重构中，我对我的`value`的类型进行了大改，在初期，我甚至认为`alloca`指令返回的类型为`int`，这导致我在`GEP`指令的生成中出现了巨量的特判，在仔细进行了`code review`之后，我决定对类型部分进行大刀阔斧的重构，最终也是将类型修改的非常自然，之后的代码生成等等也是比较容易。

代码生成优化

在代码生成中，我做了一些显而易见的简单优化，

1. 在对`llvm`进行计算的时候，如果计算的两个`value`均为常数，直接返回一个新的常数为他们的计算结果。如果计算两旁的值为`const`直接将其值取出来即可。
2. 在进行判断时，我也对比较计算进行了类似的操作，可以简化比较结果的获得。

中端优化

中端优化我做了`Mem2Reg`，`GVN`，死代码删除，函数内联。

构建CFG

CFG的重要性不必多言，他站在了全局的视角去对编译器的资源进行了分配。下面是我对其的主要实现。

删除不可达块

不可达块的做法非常简单，不断进行DFS算法，将所有可到达块保存起来，其余的块即为不可到达块。

确定支配块

确定支配块的思路很简单，就是*拦路虎*思想，对于一个块，我从入口处进行DFS算法进行不断推进，如果遇到该块，我马上进行截断，然后继续不断进行，此时不可到达的块即为该块的支配块。

确定支配边界和直接支配关系

参考了教程的做法，不过多赘述。

Mem2Reg

据助教学长所言，这个优化可以说是所有中端优化之母，有了他就可以生成SSA式的中间代码，在做各种优化的时候都可以事半功倍。在做了这个优化之后我也发现了其特别之处：对于整数型变量而言，他取消了alloc和store，采用了并行赋值的操作，降低了io的次数，将代码转化为了严格的SSA形式，对之后的优化有着奠基的作用。

实现方法与困难

实现方法就是对着教程的实现方法学习（

教程的解读对于简单的SSA实现已经是足够，完全足以实现一个简单的Mem2Reg

phi的插入

在Mem2Reg中的第一个困难就是无法理解教程中的算法，尤其是对phi指令的插入，为此我也专门去找了助教，助教告诉了我一定的实现方法，但是我完全无法理解，主要问题还是对支配的关系理解不够深刻，在答疑结束之后，我细细对我之前的一些测试样例进行画图，构建出了支配关系图，理解了插入phi函数的意义。

> 节点n的支配边界时CFG中刚刚号不被n支配的节点的集合，形式化的定义是：
>
> $$DF(x) = \{x | n支配x的前驱节点，n不严格支配x\}$$
>
> 就是说，在此之间n对节点都是路径支配的，但是在这个集合中出现了其他的可能性，这些其实很好想的一点就是这些可能性并不是凭空产生的，所以说一个节点x，他时一个支配边界，那他必然是不止一个节点的支配边界，对于在这些块中定义的变量，如果这个节点x也对这个节点进行了定义，那么这个节点必然也是join node

phi指令应该针对的是某个变量有可能存在不同的到达定义的基本块，因为在支配链中，任何的到达定义都是可以预见的，只有在支配边界的地方，由于对该块不再支配，此时可能有另一个支配链的末尾指向该基本块，此时就需要判断是从哪个支配链的到达定义，因此教程中的算法就显得相当自然，就应该插入到某个支配链末尾的后继基本块处，这是因为此时我们不再能确定他的到达定义。

变量重命名

变量重命名也是`Mem2Reg`中一个比较有趣的点，其实这个也是相当自然的，我的操作是对每一个Value进行重命名的时候对他开一个栈，对其的支配树进行前序遍历，在遇到新的到达定义的时候就需要把他压倒栈中，当离开这个基本块的时候就把压入这个栈的到达定义取出来，这样就可以简单的完成重命名的操作，而支配树加前序遍历的操作在之后的`GVN`中也是相当常见而且自然。

后续

在完成了上面的几步之后，我意外的非常轻松的实现了`SSA`，几乎没有出现`bug`，非常轻松的完成了这个优化。

感想

对于`Mem2Reg`，我的最大的感觉就是图论，`Mem2Reg`是一个与图论相当强相关的优化，在进行了`Mem2Reg`之后，我对编译优化的整体架构发生了巨大的改变，我意识到我之前眼中的优化只不过是着眼于小局部处的小修小补，站在更高的角度才能更好的完成对编译的优化。

死代码删除

函数删除

这个非常简单，只需要删掉没有被调用的函数就ok

代码删除

版本1

死代码删除是完全基于上面的`Mem2Reg`优化做的，我的本意是对所有必须存在的语句比如`call, syscall, br, store, alloca`做`def-use`闭包，将完全无关的指令删除，这样的思路是相当自然的，但是其实也是相当低效的，举个简单的例子

```
void f(int a[]) {
    a[1] + a[2];
    a[3] + a[4];
}
```

这样的语句是完全没有意义的语句，但是我并没有对其进行删除，导致我在某个自己手捏的样例进行测试的时候发现效率相当一般，这个时候cjh同学给我提供了另一种解决思路：

版本2

从上面可以知道，我是在寻找有用的指令，然后保留下来，但是其实寻找无用的指令可能是更加高效更加果断的死代码删除方式，思路就是

```
    private void deleteDeadCode() {
        for (Function function : module.getFunctionList()) {
            HashSet<Instr> deadInstrSet = new HashSet<>();
            HashSet<Instr> records = new HashSet<>();
            for (BasicBlock block : function.getBlockArrayList()) {
                for (Instr instr : block.getInstrs()) {
                    if (instr.canBeDeleted(deadInstrSet, records)) {
                        deadInstrSet.add(instr);
                    }
                }
```

```
            }
            for (BasicBlock block : function.getBlockArrayList()) {
                ArrayList<Instr> instrs = block.getInstrs();
                instrs.removeIf(instr -> deadInstrSet.contains(instr));
            }
        }
    }


    public boolean canBeDeleted(HashSet<Instr> deadInstrSet, HashSet<Instr>
records) {
        if (this instanceof BranchInstr || this instanceof ReturnInstr || this
instanceof CallInstr || this instanceof StoreInstr)
            return false;
        if (deadInstrSet.contains(this)) return true;
        if (records.contains(this)) return false;
        records.add(this);
        for (Value value : usedByList) {
            if (value instanceof Instr instr && !instr.canBeDeleted(deadInstrSet,
records)) return false;
        }
        deadInstrSet.add(this);
        return true;
    }
```

就是去查找指令是不是被不可删除的指令使用或者本身是不可删除的指令，因此可以大大加强对死代码的审核程度，于是就完成了代码部分的死代码删除。

感想

死代码删除是每一个学习编译原理的学生都能想到的优化，但是具体的实现却互相之间不尽相同，我在这里也遇到了小部分困难，删除的效率不高，但是在同学们的帮助下，也成功提高了删除的效率。

函数内联

说来有趣，函数内联是所有中端优化中最简单明了的，即使是完全没有接触过编译原理的大一学生也完全可以理解和讲出函数内联的思路：

把函数插入到基本块中不就行了？！

说起来是容易，做起来恶心。

思路确实是非常简单的，但是由于我在`llvm`中坚持了一切皆`Value`的设计方法，我的每个`Value`之间实际上是高度耦合的，好处就是在生成MIPS并不需要根据其名字来寻找其所在的寄存器，坏处就是拷贝起来非常复杂。

函数代码拷贝

为了更好的映射，我对每个Value包括Block构建了映射表，根据其本身的值寻找得到其相对的映射，其中所有的使用关系也被替换成映射，此时可以继续保持丝丝相扣的关系。对于参数，我将形参的映射镜像设定为实参，就可以很好的把实参插入到内联的函数中。

插入新的`phi`

因为内联之后不存在`ret`语句，如果函数的类型是`int`，那我们需要把其结果映射到一个`phi`函数中，此时就需要对每个`ret`进行记录，将其存入到一个`phi`中。

基本块的分割

基本块的分割是函数内联的核心之一，因为函数往往是存在多个块，因此必须将之前的块分割开，为`b1`和`b2`，这时有一个细节需要注意，因为`phi`指令判断的块是根据其最后一个指令，而对于`branch`是跳入到块的第一个指令，因此对于原本的块`b0`，在`branch`语句中被替换为`b1`，在`phi`语句中被替换为`b2`。而对于函数内联产生的块而言，将其插入到`b1`和`b2`之间就Ok，其`phi`则插入到`b2`的开头处就ok

感想

函数内联是所有中端优化中思维难度最低的优化，但是可能是我花时间最久的优化，前前后后写了两三天，最终还好也是写出来了，但是刚开始他的效果却是一个问号，因为我到底有多少寄存器够他用？如果我把函数都内联进来了，我的寄存器怎么办？这也坚定了我写图着色优化的信心，因为真的是很有必要，如果内联函数块一直占着寄存器不放的话，我的寄存器资源会被大大浪费，寄存器分配和活跃变量分析也就配套而生。

重新构建`CFG`

由于函数内联增加了许多多余的块，我重新构建了`CFG`，为了方便之后的使用。方法和上面的相同

`GVN`

很遗憾，这个优化我并没有完全写完，我只写了一个简易的版本，由于我没有时间去写`GCM`，因此我的优化并不是非常激进的优化，思路的话非常简单，我在生成每个`Value`的除了固定的整数，都生成了一个固定的`Hash`，我只需要将算式中的进行hash标号就ok，按照字典顺序进行标号也可以保证对于`a+b`和`b+a`这种类型算式的相同，同时在进行计算的时候我也进行了简单的计算指令的优化，实在是不值一提，就不在赘述。

具体的操作

在我看来，`GVN`是除了`Mem2Reg`之外和支配关系最大的中端优化，因为在一个支配链或者支配树中，父节点是子节点的必经之路。因此在父节点的标号在子节点也是可以确定可以使用的，但是对于兄弟子树就无法进行使用了，因此我和之前的变量重命名使用了类似的方法。前序遍历支配树，在出节点的时候消去在本结点产生的所有表达式即可。

感想

非常好写而且符合直觉的优化，是在中端优化中算是非常简单的优化。但是非常有效。

后端优化

消去`phi`

由于`phi`指令需要确定来自的基本块，因此需要在前驱基本块的结尾处插入`move`指令，而在有多个前驱基本块的情况下需要插入新的基本块来保证数据来源的正确性。

同时由于`phi`指令是并行赋值，所以需要在插入`move`序列的时候插入新的变量，其实还有一种思路是进行拓扑排序，确定赋值的前后关系，由于时间原因，没有实现，不在赘述。

基本参考教程完成

基本块整合

基本块消除的是相邻的块，前驱块是通往后继块的唯一路径。而且后继块的唯一前驱就是前驱块。此时可以将两个基本块融合，整合成为唯一基本块。

图着色寄存器分配

因为时间关系，我没有实现教程中的图着色，在机缘巧合下，我看到了https://github.com/gyp2847399255/SysY-compiler/blob/master/%E7%94%B3%E4%BC%98%E6%96%87%E6%A1%A3.md这位学长的申优文档，了解到了一个不需要真正建立冲突图的寄存器算法，了解了基本想法之后，我采用了这个算法。

基本步骤就是对每个块进行活跃变量分析，然后对块进行具体的分析，具体步骤如下：

0. 首先建立一个寄存器的集合，表达空闲的寄存器
1. 首先存储块中的每个`Value`的最后一次使用
2. 对指令进行遍历，如果当前指令为某个之前的`Value`的最后一次使用，而且该基本块的$outset$中不含有该`value`，将其所占有的寄存器释放。
3. 如果指令是定义型指令，对该指令的结果进行一个寄存器分配，如果没有足够的寄存器，则进行权衡，进行spill操作，spill操作的准则是：1. 在in集合的寄存器不能释放 2. 计算每个`Value`的使用次数，选出不在in中最小的一个进行释放
4. 处理完基本块之后，我们选择遍历支配树，对其子节点进行分配，首先查看当前的子节点的in集合，我们可以将in集合之外的那些属于out集合的`Value`进行释放，之后再进行恢复即可。
5. 离开该基本块之后，释放所有寄存器

想了很久我对于`spill`的情况到底该怎么办，还是没有一个很好的想法，没有办法了，如果想得到一个好的`spill`效果，循环分析什么的必然少不了，而我已经没有时间了，只能去选择消去使用次数最少的`value`。。。

困难

寄存器分配的困难主要在于无法找到对应的寄存器，这往往会造成各种无法预测的错误，为了防止这种错误，我将分配释放的寄存器信息打印了出来进行具体的查看去寻找bug

一些补充

这种寄存器算法想要释放好的效果最好在拆除`phi`之前，否则在`move`指令中可能会造成活跃变量分析的bug

感想

提升效果最明显的优化，但是其实是与之前的优化相辅相成，很可惜没有尝试真正的图着色。

修改

基本无太多设计上的修改。主要修改是`debug`

目标代码生成

我选择的目标代码是`MIPS`，在我看来`MIPS`和`llvm`之间有着很强烈的映射关系。几乎每一条`llvm`语句都可以直接翻译为`MIPS`指令。

由于在之前已经做过了寄存器分配，所以在翻译`llvm`的过程中只需要取出`value`对应的寄存器即可，如果其没有被分配寄存器则将其压入栈中，将其基于当前函数`$sp`的偏移量记录下来即可。

对于基本块，在其之前加上一个`label`用于进行跳转。

对于数组，局部变量则全部压入栈中，全局变量则分配到`.data`段。

基本框架与`llvm`高度类似，也是使用单例模式进行全局控制。区别在于遍历的对象由抽象语法树改为了`llvm`语法成分树。最终形成一颗`MIPS`指令树。

函数调用

在压入参数的时候需要先将寄存器压入栈内，或者进行拓扑排序，避免一些无畏的错误。

需要保存现场

putint putchar

对于`IO`类指令，只需要调用相应的`syscall`即可。

对于此类函数，保存现场只需要将`v0,a0`等寄存器压入栈。

修改

一些小修小补的`bug`，主要针对`llvm`进行翻译，难度并不大。

参考编译器介绍

https://github.com/saltyfishyjk/BUAA-Compiler（yjk学长的编译器，在设计`llvm`时有所参考）

https://github.com/Thysrael/Pansy（qs学长的编译器，学习了一些设计的思想）

https://github.com/Hyggge/Petrichor（czh学长的编译器，参考了语法树结构和文件结构）

https://github.com/gyp2847399255/SysY-compiler（gyp学长的编译器，参考了寄存器算法）