编译器设计文档

总体设计

项目总体目录结构如下，理论上第一部内容可以在一遍中完成，不过为了方便代码的解耦与维护，我分成了三个模块完成——词法分析，语法分析和语义分析。具体每个部分的设计如后文所示。

```
Compiler/src
| Compiler.java
|- lexical
 | Lexical.java
 |- assist
  | Reserved.java
  | Signal.java
|- syntax
 | SyntaxAnalysis.java
 |- exps
  | BiExp.java
  | CallExp.java
  | LeftExp.java
  | NumExp.java
  | UnaryExp.java
 |- nodes
  | Block.java
  | CompUnit.java
  | Decl.java
  | Def.java
  | FuncDef.java
  | FuncFParam.java
  | MainFuncDef.java
  | RootNode.java
  | Stmt.java
 |- stmts
  | Assign.java
  | Break.java
  | Continue.java
  | If.java
  | Print.java
  | Return.java
  | Stdin.java
  | While.java
|- generate
 | Mediate.java
 | VarTable.java
 | ForMidPrint.java
 |- usage
  | Block.java
```

```
   | BlockType.java
   | Const.java
   | Function.java
   | Instr.java
   | Phi.java
   | Program.java
   | Stack.java
   | Use.java
   | Value.java
   |- instr
     | ...
 |- mips
  | Backend.java
  | ForMipsPrint.java
```

词法分析

实现时遍历输入的文本文件，并进行状态转移操作，在TEXT模式下读取关键词和说明符，在STRING模式下读取字符串常量，在NUMBER模式下读取数字，此外其余的终结符均只有一个或两个字母，在NORMAL模式下使用HASHMAP来判断类型，以上得到的元素通过`gen`函数生成单词，存入返回的数组里。

为了方便源程序与词法分析后格式的转换，设置了两个辅助类用于读入转换。

gen函数实现

```
public void gen(int lineNum) {
        String nowBuffer = buffer.toString();
        if (nowBuffer.isEmpty()) {
            return;
        }
        if (state.equals("Num")) {
            items.add(new Pair<>(new Pair<>("INTCON", nowBuffer), lineNum+1));
        }
        if (state.equals("String")) {
            items.add(new Pair<>(new Pair<>("STRCON", nowBuffer), lineNum+1));
        }
        if (state.equals("Text")) {
            String str = nowBuffer.toLowerCase();
            if (new Reserved().isReserved(nowBuffer)) {
                items.add(new Pair<>(new Pair<>(new Reserved().getType(str), str),
                        lineNum+1));
            } else {
                items.add(new Pair<>(new Pair<>("IDENFR", nowBuffer), lineNum+1));
            }
        }
        if (state.equals("Normal")) {
            items.add(new Pair<>(new Pair<>(new Signal().getType(nowBuffer),
nowBuffer), lineNum+1));
        }
```

```
        buffer = new StringBuilder("");
        state = "Normal";
    }
```

辅助类与读入转换

```
public void make() {
        all.put("!", "NOT");
        all.put("&&", "AND");
        all.put("||", "OR");
        all.put("+", "PLUS");
        all.put("-", "MINU");
        all.put("*", "MULT");
        all.put("/", "DIV");
        all.put("%", "MOD");
        all.put("<", "LSS");
        all.put("<=", "LEQ");
        all.put(">", "GRE");
        all.put(">=", "GEQ");
        all.put("==", "EQL");
        all.put("!=", "NEQ");
        all.put("=", "ASSIGN");
        all.put(";", "SEMICN");
        all.put(",", "COMMA");
        all.put("(", "LPARENT");
        all.put(")", "RPARENT");
        all.put("[", "LBRACK");
        all.put("]", "RBRACK");
        all.put("{", "LBRACE");
        all.put("}", "RBRACE");
    }
```

```
public void make() {
        reserves.put("main", "MAINTK");
        reserves.put("const", "CONSTTK");
        reserves.put("int", "INTTK");
        reserves.put("break", "BREAKTK");
        reserves.put("continue", "CONTINUETK");
        reserves.put("if", "IFTK");
        reserves.put("else", "ELSETK");
        reserves.put("while", "WHILETK");
        reserves.put("getint", "GETINTTK");
        reserves.put("printf", "PRINTFTK");
        reserves.put("return", "RETURNTK");
        reserves.put("void", "VOIDTK");
    }
```

语法分析

这一部分主要使用递归下降进行实现。首先定义了一个SyntaxAnalysis类，分析之前词法分析得到的结果数组，使用一个index变量用于记录当前遍历的位置。通过若干个get<语法成分名>函数，不断递归分析语法成分。

依据文法要求，我设置了`exps`、`nodes`、`stmts`三个包，并在其中定义实现了文法要求的几乎所有成分，并根据自己对题目的理解，实现了部分简化。在`exps`中，我自定义了5类表达式，方便语法分析与整合。

```
├─exps
│  │  BiExp
│  │  CallExp
│  │  LeftExp
│  │  NumExp
│  │  UnaryExp
```

最后，有一个地方超出了我原先的设计，在语句`Lval=EXP;`和`EXP;`中，`EXP`的开头部分可以是`Lval`，也就是两者的首部完全相同，且一旦尝试读取`Lval`但实际是`Exp`的话就无法跳回递归下降的队规环境，此处采用了回溯的方法进行处理——在涉及方法中采用"预读"的措施，并记录之前进入预读装态中位置，定义getRoll()方法将状态回滚回开启预读之前，在预读过程中禁用所有输出。这样通过一个简单的回溯解决了这一首部相同问题。

错误处理

在错误处理中我改造了之前的递归下降子程序，生成了语法树。而具体错误处理中我将错误分为两类，第一类是语义错误如语句缺少分号，括号等，这一部分将在语法分析中进行处理，第二部分是和语义相关的内容，如重定义等。通过符号表的建立处理语义错误，采用栈式符号表进行管理，每个块内使用hash方式存储，分别给函数和变量建立不同的符号表，存储它们的类型信息，函数包括返回值，参数数量和各自类型。变量只有类型，通过查表判定是否匹配。

主要实现和上述一致，在建立语法树的时候建立了部分抽象，其中所有类的基类是RootNode，关于表达式的基类为Exp，语句的基类是Stmt，这样的抽象有利于容器中的类型一致，有相同的接口。在递归下降子程序中返回生成的对应语法成分的指针，在每个对应的递归下降子程序中，分别读取每一个部分，存入node中。

中间代码

中间代码生成我主要在上一步生成的语法树上进行，构建SSA形式的中间代码。主要方法参考了《simple and efficient construction of static single assignment form》一文，最终生成的结果向LLVM形式靠近。

在我的IR指令中，最上层的是Value类，和LLVM中所有的东西都是Value这一思想一致，Value类中有两种类型，一种是Instr 代表着代码中具体会执行的指令的类型，比如存取，读出，双目计算等等，它可以被def也可以被use，而Const是常数类型，在代码中只能被use不能被def。

所有的IR通过Def-Use关系进行串联，和LLVM的设置比较类似。对于每个Instruction，我们使用一个operand记录了它所使用的所有数据，而对于每一个Value，其中有一个Use的链表维护了它的所有的使用点，这样的双向引用很容易完成中间代码的稀疏迭代。

SSA

SSA的构建主要参考了《simple and efficient construction of static single assignment form》一文，通过在语法树上的遍历，对语句的重新标号，在递归读取变量定义时往BLock中插入Phi等一系列操作完成了SSA形式的构建。

代码生成

目标代码生成部分主要对应`mips`包，其中主要完成了对于中间代码到Mips的翻译工作，对于常规指令的生成勿用赘述，对于Branch指令会生成branch/j 指令完成对目标块的无条件跳转，而对于Phi结点选择向其前驱结点的尾部（所有块之后，最后一个跳转之前）插入move指令完成Phi结点的消除。

代码优化

寄存器分配

使用了LinearScan算法，首先为所有的变量构造声明周期，其中等效于做了一个简单的数据流分析。随后使用了POLETTO的原始版本的LinearScan算法，使用优先队列维护所有Interval对其进行遍历，然后按论文中的策略进行寄存器的分配与spill。寄存器分配时，使用一个栈来维护寄存器的列表，可以尽可能少的分配寄存器，方便之后的函数调用进行优化。

窥孔优化

Jump/Branch类型指令的窥孔优化，我们将基本块按照输入的顺序进行输出，这样基本保持原有的顺序，然后观察基本块之后的跳转目标是不是就是直接后继，如果是的话就可以去掉这一跳转

乘除法优化

参照论文`Division by Invariant Integers using Multiplication`和`libDivide`库实现了常量除法的优化

函数调用优化

本来在函数调用时无条件保存/恢复了所有可能的寄存器，由于上述寄存器分配尽可能少原则，只要将两个函数使用的寄存器取交集就可以得到了真正可能受害的指令（最优的处理需要求出函数调用时的活跃变量），这样对于没有内连的小函数也尽量减少了开销。

死代码删除

借助于生成interval期间产生的活跃信息，我们可以方便的在生成目标代码时直接跳过那些不活跃的代码，但是为了保持语法严谨性，对于函数等没有用上返回值的”死代码“只能保守分析。

参考编译器——LLVM

架构特点

- 不同的前端后端使用统一的中间代码LLVM Intermediate Representation (LLVM IR)
- 如果需要支持一种新的编程语言，那么只需要实现一个新的前端
- 如果需要支持一种新的硬件设备，那么只需要实现一个新的后端
- **优化阶段是一个通用的阶段，它针对的是统一的LLVM IR，不论是支持新的编程语言，还是支持新的硬件设备，都不需要对优化阶段做修改**