写在前面

当我写完近一万字的文档时，再次回顾，我深刻感受到了编译知识体系的庞大，更超过了CO与OS。感谢课程组的辛苦付出。还记得和杨老师商量延期时老师的综合考量与通情达理。唯一的遗憾是，这学期因为生病以及密接去隔离了三次，耽误了不少时间。之前我以为自己肯定能写完，已经实现了函数内联、窥孔很优化等基础优化，但最后发现时间不够了，后端的bug没有de完。从分数的角度来说，甚至不如直接摆烂做后端；但学到了知识也是很开心的。希望编译课程可以越来越好！

编译器总体设计

实现一个将SysY语言翻译为MIPS汇编语言的编译器，编译过程可以分为五个阶段：

- 词法分析
- 语法分析
- 语义分析和中间代码生成
- 代码优化
- 目标代码生成

采用"高内聚、低耦合"的解耦思想设计，实现一个将SysY翻译为MIPS汇编语言的编译器，主要分为词法分析、语法分析、语义分析与中间代码生成、代码优化与目标代码生成五个阶段。其中符号表管理和错误处理穿插在语法分析与语义分析阶段中

用linux下tree命令分析，总体结构如下:

```
Compiler/
├── Lexer.iml
├── condition.c
├── condition.exe
├── error.txt
├── llvm_ir.txt
├── main.ll
├── mips.txt
├── out
│   └── production
│       └── Lexer
│           ├── Compiler.class
│           ├── backend
│           │   ├── Global.class
│           │   ├── Instr
│           │   │   ├── BranchInstr.class
│           │   │   ├── JumpInstr.class
│           │   │   ├── LuiInstr.class
│           │   │   ├── MemInstr.class
│           │   │   ├── MfInstr.class
│           │   │   ├── MipsInstr.class
│           │   │   ├── MulDivInstr.class
│           │   │   ├── RegImmInstr.class
│           │   │   ├── RegRegInstr.class
│           │   │   ├── SyscallInstr.class
│           │   │   └── Tag.class
│           │   ├── MipsBlock.class
│           │   ├── MipsFunc.class
```

```
|           |   ├── MipsGen$1.class
|           |   ├── MipsGen.class
|           |   ├── MipsModule.class
|           |   └── STR.class
|           ├── errorHandler
|           |   ├── ErrorHandler.class
|           |   ├── ErrorTable.class
|           |   └── Ident.class
|           ├── frontend
|           |   ├── Lexer
|           |   |   ├── FS.class
|           |   |   ├── Id.class
|           |   |   ├── Lexer.class
|           |   |   ├── LexicalType.class
|           |   |   ├── Num.class
|           |   |   ├── Op.class
|           |   |   ├── Reserve.class
|           |   |   ├── Sep.class
|           |   |   └── Token.class
|           |   ├── MyPair.class
|           |   ├── Parser
|           |   |   ├── Block
|           |   |   |   ├── BLockItem.class
|           |   |   |   ├── Block.class
|           |   |   |   ├── Cond.class
|           |   |   |   ├── FmStr.class
|           |   |   |   └── Stmt.class
|           |   |   ├── Btype.class
|           |   |   ├── CompUnit.class
|           |   |   ├── Decl
|           |   |   |   ├── ConstDecl.class
|           |   |   |   ├── ConstDef.class
|           |   |   |   ├── ConstExp.class
|           |   |   |   ├── ConstInitval.class
|           |   |   |   ├── Decl.class
|           |   |   |   ├── InitVal.class
|           |   |   |   ├── VarDecl.class
|           |   |   |   └── VarDef.class
|           |   |   ├── Exp
|           |   |   |   ├── AddExp.class
|           |   |   |   ├── EqExp.class
|           |   |   |   ├── Exp.class
|           |   |   |   ├── LAndExp.class
|           |   |   |   ├── LOrExp.class
|           |   |   |   ├── LVal.class
|           |   |   |   ├── MulExp.class
|           |   |   |   ├── Number.class
|           |   |   |   ├── PrimaryExp.class
|           |   |   |   ├── RelExp.class
|           |   |   |   ├── UnaryExp.class
|           |   |   |   └── UnaryOp.class
|           |   |   ├── Func
|           |   |   |   ├── FuncDef.class
|           |   |   |   ├── FuncFParam.class
|           |   |   |   ├── FuncFParams.class
|           |   |   |   ├── FuncRParams.class
```

```
|           |   |   |   ├── FuncType.class
|           |   |   |   └── MainFuncDef.class
|           |   |   ├── Parser.class
|           |   |   └── SyntacticalType.class
|           |   ├── SymbolTable.class
|           |   ├── Visitor$1.class
|           |   ├── Visitor$10.class
|           |   ├── Visitor$11.class
|           |   ├── Visitor$12.class
|           |   ├── Visitor$2.class
|           |   ├── Visitor$3.class
|           |   ├── Visitor$4.class
|           |   ├── Visitor$5.class
|           |   ├── Visitor$6.class
|           |   ├── Visitor$7.class
|           |   ├── Visitor$8.class
|           |   ├── Visitor$9.class
|           |   └── Visitor.class
|           ├── ir
|           |   ├── FactoryBuilder$1.class
|           |   ├── FactoryBuilder.class
|           |   ├── Type
|           |   |   ├── ArrayType.class
|           |   |   ├── FunctionType.class
|           |   |   ├── IntegerType.class
|           |   |   ├── LabelType.class
|           |   |   ├── PointerType.class
|           |   |   ├── Type.class
|           |   |   └── VoidType.class
|           |   └── Value
|           |       ├── BasicBlock.class
|           |       ├── Constant
|           |       |   ├── Constant.class
|           |       |   ├── ConstantArray.class
|           |       |   ├── ConstantInt.class
|           |       |   └── ConstantVar.class
|           |       ├── Function.class
|           |       ├── GlobalVariable.class
|           |       ├── Instruction
|           |       |   ├── BinaryInst$1.class
|           |       |   ├── BinaryInst.class
|           |       |   ├── IO
|           |       |   |   ├── Getint.class
|           |       |   |   ├── Putint.class
|           |       |   |   └── Putstr.class
|           |       |   ├── Instruction.class
|           |       |   ├── Mem
|           |       |   |   ├── AllocaInst.class
|           |       |   |   ├── GepInst.class
|           |       |   |   ├── LoadInst.class
|           |       |   |   ├── MemInst.class
|           |       |   |   ├── PhiInst.class
|           |       |   |   ├── StoreInst.class
|           |       |   |   └── ZextInst.class
|           |       |   ├── TAG.class
|           |       |   └── Terminate
```

```
|           |       |       ├── BrInst.class
|           |       |       ├── CallInst.class
|           |       |       ├── RetInst.class
|           |       |       └── TerminateInst.class
|           |       ├── Module.class
|           |       ├── Str.class
|           |       ├── User.class
|           |       └── Value.class
|           ├── pass
|           |   ├── Inline$1.class
|           |   ├── Inline.class
|           |   ├── Mem2Reg.class
|           |   └── Merge.class
|           └── src (1).zip
├── output.txt
├── pre.cpp
├── src
|   ├── Compiler.java
|   ├── backend
|   |   ├── Global.java
|   |   ├── Instr
|   |   |   ├── BranchInstr.java
|   |   |   ├── JumpInstr.java
|   |   |   ├── LuiInstr.java
|   |   |   ├── MemInstr.java
|   |   |   ├── MfInstr.java
|   |   |   ├── MipsInstr.java
|   |   |   ├── MulDivInstr.java
|   |   |   ├── RegImmInstr.java
|   |   |   ├── RegRegInstr.java
|   |   |   ├── SyscallInstr.java
|   |   |   └── Tag.java
|   |   ├── MipsBlock.java
|   |   ├── MipsFunc.java
|   |   ├── MipsGen.java
|   |   ├── MipsModule.java
|   |   └── STR.java
|   ├── errorHandler
|   |   ├── ErrorHandler.java
|   |   ├── ErrorTable.java
|   |   └── Ident.java
|   ├── frontend
|   |   ├── Lexer
|   |   |   ├── FS.java
|   |   |   ├── Id.java
|   |   |   ├── Lexer.java
|   |   |   ├── LexicalType.java
|   |   |   ├── Num.java
|   |   |   ├── Op.java
|   |   |   ├── Reserve.java
|   |   |   ├── Sep.java
|   |   |   └── Token.java
|   |   ├── MyPair.java
|   |   ├── Parser
|   |   |   ├── Block
|   |   |   |   ├── BLockItem.java
```

```
|   |   |   |   ├── Block.java
|   |   |   |   ├── Cond.java
|   |   |   |   ├── FmStr.java
|   |   |   |   └── Stmt.java
|   |   |   ├── Btype.java
|   |   |   ├── CompUnit.java
|   |   |   ├── Decl
|   |   |   |   ├── ConstDecl.java
|   |   |   |   ├── ConstDef.java
|   |   |   |   ├── ConstExp.java
|   |   |   |   ├── ConstInitval.java
|   |   |   |   ├── Decl.java
|   |   |   |   ├── InitVal.java
|   |   |   |   ├── VarDecl.java
|   |   |   |   └── VarDef.java
|   |   |   ├── Exp
|   |   |   |   ├── AddExp.java
|   |   |   |   ├── EqExp.java
|   |   |   |   ├── Exp.java
|   |   |   |   ├── LAndExp.java
|   |   |   |   ├── LOrExp.java
|   |   |   |   ├── LVal.java
|   |   |   |   ├── MulExp.java
|   |   |   |   ├── Number.java
|   |   |   |   ├── PrimaryExp.java
|   |   |   |   ├── RelExp.java
|   |   |   |   ├── UnaryExp.java
|   |   |   |   └── UnaryOp.java
|   |   |   ├── Func
|   |   |   |   ├── FuncDef.java
|   |   |   |   ├── FuncFParam.java
|   |   |   |   ├── FuncFParams.java
|   |   |   |   ├── FuncRParams.java
|   |   |   |   ├── FuncType.java
|   |   |   |   └── MainFuncDef.java
|   |   |   ├── Parser.java
|   |   |   └── SyntacticalType.java
|   |   ├── SymbolTable.java
|   |   └── Visitor.java
|   ├── ir
|   |   ├── FactoryBuilder.java
|   |   ├── Type
|   |   |   ├── ArrayType.java
|   |   |   ├── FunctionType.java
|   |   |   ├── IntegerType.java
|   |   |   ├── LabelType.java
|   |   |   ├── PointerType.java
|   |   |   ├── Type.java
|   |   |   └── VoidType.java
|   |   └── Value
|   |       ├── BasicBlock.java
|   |       ├── Constant
|   |       |   ├── Constant.java
|   |       |   ├── ConstantArray.java
|   |       |   ├── ConstantInt.java
|   |       |   └── ConstantVar.java
```

```
|   |       ├── Function.java
|   |       ├── GlobalVariable.java
|   |       ├── Instruction
|   |       |   ├── BinaryInst.java
|   |       |   ├── IO
|   |       |   |   ├── Getint.java
|   |       |   |   ├── Putint.java
|   |       |   |   └── Putstr.java
|   |       |   ├── Instruction.java
|   |       |   ├── Mem
|   |       |   |   ├── AllocaInst.java
|   |       |   |   ├── GepInst.java
|   |       |   |   ├── LoadInst.java
|   |       |   |   ├── MemInst.java
|   |       |   |   ├── PhiInst.java
|   |       |   |   ├── StoreInst.java
|   |       |   |   └── ZextInst.java
|   |       |   ├── TAG.java
|   |       |   └── Terminate
|   |       |       ├── BrInst.java
|   |       |       ├── CallInst.java
|   |       |       ├── RetInst.java
|   |       |       └── TerminateInst.java
|   |       ├── Module.java
|   |       ├── Str.java
|   |       ├── User.java
|   |       └── Value.java
|   ├── pass
|   |   ├── Inline.java
|   |   ├── Mem2Reg.java
|   |   └── Merge.java
|   └── src (1).zip

43 directories, 280 files
```

架构设计

词法分析设计

设计方法

词法分析要根据给出的文法设计一个DFA，将字符串转化为词法的token，便于语法分析进行递归下降处理。

词法的token具有某种架构上的统一性，因此设计了一个token接口满足统一的功能，用不同的词法类来实现token接口实现各自不同的需求

```
package frontend.Lexer;

public interface Token {

    public int getLine();

    public String getType();

    public String getName();
}
```

具体实现

词法分析过程本质是根据文法所用到的终结符构建一个 DFA，识别出所有的关键字及标识符。由于实现的文法关键词较少，文法相对简单，关键词长度不超过 10，实现过程中并不需要真正构建出一个 DFA，只需模拟 DFA 进行贪心匹配即可。

具体来说，设计的关键点如下：

- 由于windows与linux环境下换行符在'\r'上存在区分，因此采用文件IO中的readline()来消除影响
- 对于源代码中存在的注释与FormatString，需要设置两个变量来标记当前状态：
  - noteSt标记注释状态，0代表无注释，1代表单行，2代表多行
  - 单行注释，从 // 开始，到换行符结束，故每读入新的一行需要去掉单行注释状态；多行注释，从 /* 开始，到 */ 结束，因此随时都有可能更新，需要预读下一个字符
  - fsSt表示是否正在读格式字符串，FormatString从 " 开始，到 " 结束；同时每读入新的一行需要清空格式字符串内容
- 空白符自动跳过即可
- 数字开头必然为常量，故直接贪心匹配即可。注意非0数字不会以0开头，因此直接可以特判
- 对于保留字，其构造词法和普通变量完全一致，只是名称特殊，故放在变量扫描中处理，在读完整个单词后直接特判即可
- 字符'/'与'*'与'"'需要特判，因其涉及到注释与格式字符串
- 对于剩下的字符，可以将其分为三类：
  - 双字符遵循最长匹配原则，因此需要预读下一个字符判断是什么符号

    ```
    public boolean isDouble(char c) {
            return c == '!' || c == '<' || c == '>' || c == '=';
    }
    ```

  - 另一种读入就可以直接判断：

    ```
    public boolean isSingle(char c) {
            return c == '&' || c == '|' || c == '+' || c == '-' || c == '%'
    || c == '*';
    }
    ```

  - 最后一种统一视为分隔符：

```java
public boolean isSep(char c) {
        return c == '[' || c == ']' || c == '{' || c == '}' || c == '('
|| c == ')' || c == ',' || c == ';';
}
```

- 所有预读下一个字符都需要判断是否越界
- 保留单词所在行号以便后续功能实现
- 所有词法类统一继承Token接口，以方便输出评测

心得

- 词法做出来成分本质上是为了**语法分析程序可以去利用分析结果来分析语法成分**
- 此外，要**防止词法分析与语法分析过于耦合**，所以有**单词类和语法成分类两个分类划分**

语法分析设计

设计分析

```
编译单元     CompUnit → {Decl} {FuncDef} MainFuncDef  // 1.是否存在Decl 2.是否存在
FuncDef
声明   Decl → ConstDecl | VarDecl // 覆盖两种声明
常量声明     ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';' // 1.花括号内重
复0次 2.花括号内重复多次
基本类型     BType → 'int' // 存在即可
常数定义     ConstDef → Ident { '[' ConstExp ']' } '=' ConstInitVal  // 包含普通变
量、一维数组、二维数组共三种情况
常量初值     ConstInitVal → ConstExp
    | '{' [ ConstInitVal { ',' ConstInitVal } ] '}' // 1.常表达式初值 2.一维数组初值
3.二维数组初值
变量声明     VarDecl → BType VarDef { ',' VarDef } ';' // 1.花括号内重复0次 2.花括号内
重复多次
变量定义     VarDef → Ident { '[' ConstExp ']' } // 包含普通变量、一维数组、二维数组定义
    | Ident { '[' ConstExp ']' } '=' InitVal
变量初值     InitVal → Exp | '{' [ InitVal { ',' InitVal } ] '}'// 1.表达式初值 2.一
维数组初值 3.二维数组初值
函数定义     FuncDef → FuncType Ident '(' [FuncFParams] ')' Block // 1.无形参 2.有形
参
主函数定义    MainFuncDef → 'int' 'main' '(' ')' Block // 存在main函数
函数类型     FuncType → 'void' | 'int' // 覆盖两种类型的函数
函数形参表    FuncFParams → FuncFParam { ',' FuncFParam } // 1.花括号内重复0次 2.花括号
内重复多次
函数形参     FuncFParam → BType Ident ['[' ']' { '[' ConstExp ']' }] // 1.普通变量
2.一维数组变量 3.二维数组变量
语句块      Block → '{' { BlockItem } '}' // 1.花括号内重复0次 2.花括号内重复多次
语句块项     BlockItem → Decl | Stmt // 覆盖两种语句块项
```

观察文法定义可以发现，SysY文法中左侧均为非终结符，文法为上下文无关文法。因此无需结合上下文环境判断，可以通过递归下降进行处理

具体实现

- 采用递归下降处理，每一部分的分析函数只需要根据文法实现即可，例如顶层文法CompUnit → {Decl} {FuncDef} MainFuncDef：

```
public void parseCompUnit() {
        while (tokens.get(pos).getName().equals("const") ||
tokens.get(pos).getName().equals("int")) {
            if (tokens.get(pos).getName().equals("const")) {
                compUnit.addConstDecl(parseConstDecl());
            } else {
                if (tokens.get(pos + 2).getName().equals("(")) {
                    break;
                }
                compUnit.addVarDecl(parseVarDecl());
            }
        }
        while (tokens.get(pos).getName().equals("void") ||
tokens.get(pos).getName().equals("int")) {
            if (tokens.get(pos).getName().equals("void")) {
                compUnit.addFuncDef(parseFuncDef());
            } else {
                if (tokens.get(pos + 1).getName().equals("main")) {
                    break;
                }
                compUnit.addFuncDef(parseFuncDef());
            }
        }
        compUnit.addMainFuncDef(parseMainFuncDef());
        prints.add("<CompUnit>");
    }
```

- 对每个语法成分都建类处理，这样可以降低耦合度并更加灵活地增加功能；而对于那些需要统一实现的功能，我们只需要增加接口使其继承即可
- 注意单词流指针位置的控制，处理每个类时，该类的处理需要读掉属于该类的全部单词，同时不能读掉该类之外的单词，防止混乱
- 需要区分可选项[]（使用if判断）以及重复0次至多次的成分{}（使用while等循环结构判断）
- 左递归文法消除：采用文法改写，例如关系表达式RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp：
    - 首先改写文法，S→a|Sb 可改写为 S→a{b}，故上述文法可改写为RelExp → AddExp ('<' | '>' | '<=' | '>=') AddExp ，直接循环处理即可
    - 改写文法后需要注意对于文法类的分析：

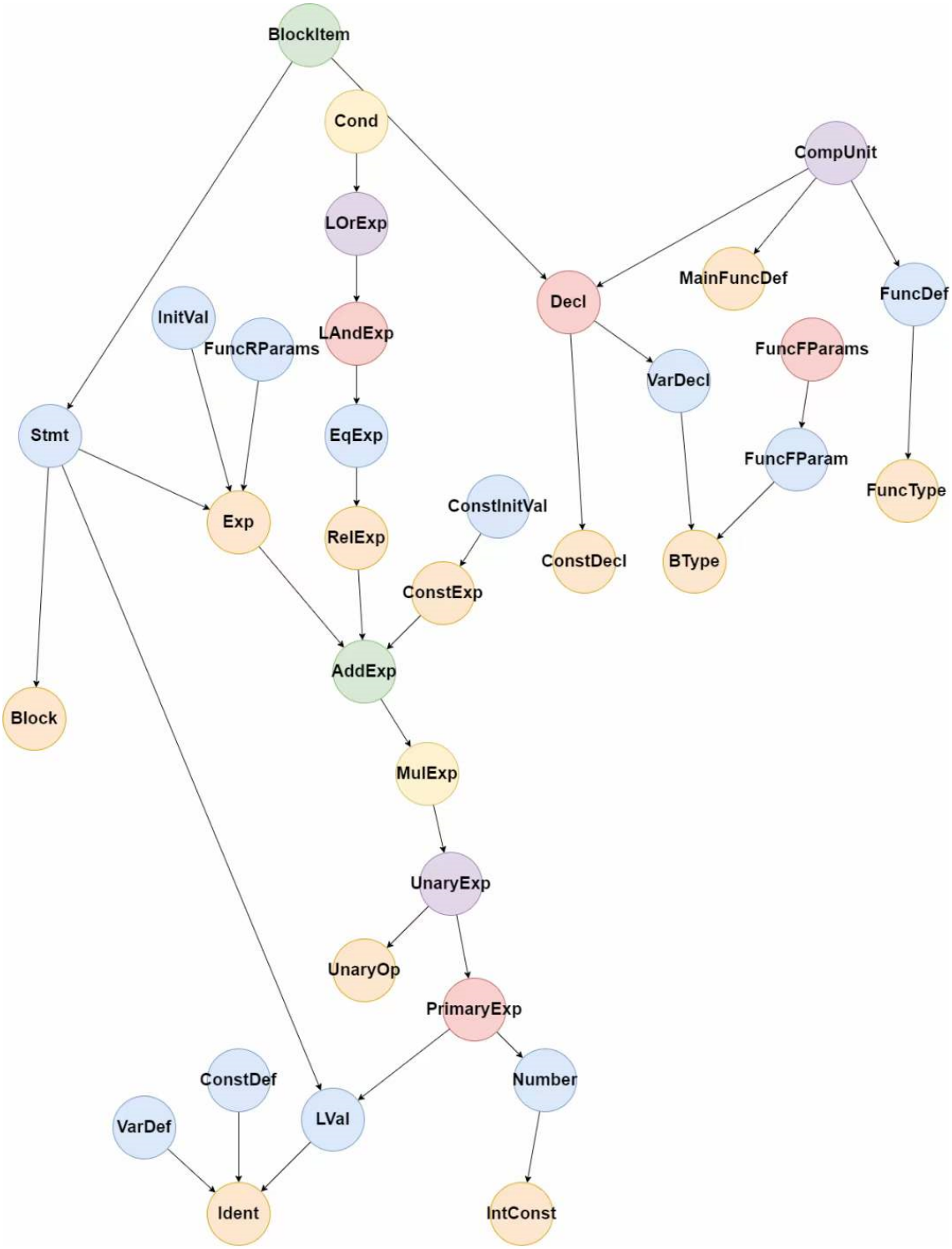
```
public RelExp parseRelExp() {
        RelExp relExp = new RelExp();
        relExp.addAddExp(parseAddExp());
        prints.add("<RelExp>");
        while (syntacticalType.isRelOp(tokens.get(pos))) {
            relExp.addOp(((Op) tokens.get(pos)));
            addToken(1);
            relExp.addAddExp(parseAddExp());
            prints.add("<RelExp>");
        }
        return relExp;
}
```

其中第一处对应原文法中RelExp → AddExp；第二处对应原文法中的RelExp → RelExp ('<' | '>' | '<=' | '>=') AddExp

- 文法中存在回溯问题，因为某些非终结符规则右部存在多个选择，且first集相交。采用超前扫描处理，直到可以确定
- 语法分析中最复杂的部分是Stmt语句识别，因其可能情况较多

```
1 语句 Stmt → LVal '=' Exp ';'
2 | [Exp] ';'
3 | Block
4 | 'if' '( Cond ')' Stmt [ 'else' Stmt ]
5 | 'while' '(' Cond ')' Stmt
6 | 'break' ';' | 'continue' ';'
7 | 'return' [Exp] ';'
8 | LVal = 'getint''('')'';'
9 | 'printf' '('FormatString {',' Exp} ')'';'
```

  - 对于打印语句、返回语句、跳转语句、循环语句、分支语句，可以很方便地通过第一个标识符（保留字）进行识别。如果第一个符号是左括号，则判断为语句块
  - 需要区分的是LVal和Exp。注意到Exp成分中没有赋值号"="，故可以从当前位置向后找，判断先遇到分号还是先遇到赋值符号('=')。如果先遇到赋值符号，则说明是赋值语句，否则是表达式语句
- 单词流的移动统一使用addToken方法，模拟真正的词法分析过程，语法成分在完成该部分词法分析后加入



错误处理设计

设计分析

错误处理实际上应该贯穿整个编译阶段，但编译实验中给出的错误处理较为独立，而之后的代码生成与竞速作业均保证代码正确的，因此可以在语法分析过程中建立栈式符号表判断。

在语法分析的基础上，我增加了如下变量以及符号表类来辅助错误处理：

```
    private ErrorTable errorTable = new ErrorTable();//错误处理符号表
    public ArrayList<MyPair<String, Integer>> errors = new ArrayList<>();//记录错误类型
    private int numWhile = 0;//记录有几层循环块
    private String retType = "";//函数的返回类型
    private boolean judgeA = true;//判定合法字符串
    private boolean assign = false;//特判stmt
    private boolean funcBlock;//标记是函数的block而非stmt的
    private boolean isDim = false;//是否为数组的维数
    private int dim2 = 0;//专门用来记录第二个维数，一遍对比
    private boolean pre = false;//记录预读lval方法
    private String curType = "";//当前参数类型
```

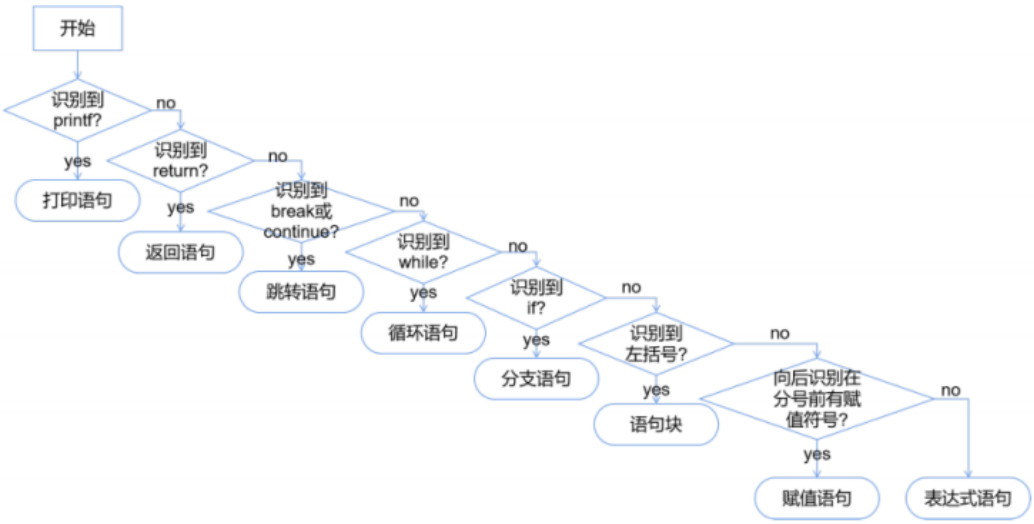
```
public class ErrorTable {
    private ArrayList<HashMap<String, Ident>> tables;//栈式符号表，用来记录变量

    public ArrayList<ArrayList<String>> types;//栈式类型表，用来记录嵌套函数调用的形参与实参类型

    public ErrorTable() {
        tables = new ArrayList<>();
        types = new ArrayList<>();
    }

    public HashMap<String, Ident> top() {
        return tables.get(tables.size() - 1);
    }

    public ArrayList<String> topType() {
        return types.get(types.size() - 1);
    }

    public Ident find(String name) {
        for (int i = tables.size() - 1; i >= 0; i--) {
            Ident t = tables.get(i).get(name);
            if (t != null) {
                return t;
            }
        }
        return null;
    }

    public Ident findB(String name) {
        return top().get(name);//如果是函数的话，其在定义的时候处于第一层，因此直接top()即可
        //如果是常变量的话，也是直接本层即可
    }

    //...

}
```

需要特别注意的一点是，原本语法分析中预读判断遇到分号与赋值号顺序来区分赋值与表达式语句的方法，因为分号缺失等可能出现的错误已经不再适用

首先对于打印语句、返回语句、跳转语句、循环语句、分支语句，可以很方便地通过第一个标识符（保留字）进行识别。如果第一个符号是左括号，则判断为语句块。

对于赋值语句和表达式语句，本文的识别方法是这样的：从当前位置向后找，判断先遇到分号还是先遇到赋值符号(‘=’)。如果先遇到赋值符号，则说明是赋值语句，否则是表达式语句。

判断思路如下图所示。



因此这里我采用了编译理论课中通过预读，判断first与follow集合的方法来区分两种表达式

- FIRST(lval)=FIRST(ident)
- FIRST(exp)=FISRT(addexp)=FIRST(mulexp)=FISRT(unaryexp)=
  - FIRST(ident '(' [FuncRParams] ')')
  - FIRST(primaryexp)
    - '('Exp')'
    - Lval：ident
    - Number
  - FIRST(unaryOp)

其他错误具体处理方法

- 非法符号：扫描一遍字符串即可
- 名字重定义、未定义名字：通过符号表判断即可
- 函数参数个数不匹配、函数参数类型不匹配：错误处理符号表中用栈式类型表记录了函数的参数与返回类型，只要比较是否相等且一致即可
- 无返回值存在不匹配的return语句：每次读到return语句时，判断当前函数的返回类型即可
- 有返回值的函数缺少return语句：实验降低了难度，不需要考虑数据流，只需考虑函数的末尾是否存在，因此在对函数分析结束后，取出函数最后一个块最后一个语句，判断其是否为return语句且返回了int类型即可
- 改变了常量的值：查询符号表判断类型即可

- 缺少分号、右小括号、右中括号：在递归下降过程中特判进行处理，同时要改变语法分析中原来判断的字符
- printf中表达式字符与表达式个数不匹配：扫描输出字符串之后，比较格式字符与后续表达式数量
- 非循环块中使用break与continue：使用变量记录循环块的深度，并动态维护，如果深度为0说明不存在循环块，则返回错误

语义分析与中间代码生成

语义分析的目的时生成健壮的中间代码，通过代码优化再将其转化为目标mips代码。我同样结合栈式符号表，采用递归下降的办法生成中间代码的语法树，我采用llvm作为我的中间代码，因其较为规范、具有SSA形式、易于优化、且每次优化后都可以确定优化的正确性。

llvm架构

- llvm ir基本单位为module
- 一个module中右若干个全局变量global与函数function
- 函数中右若干个基本块basicblock
- 一个基本块有若干指令，且必须以终结指令跳转或返回结尾

llvm指令

instructions

llvm ir	usage	intro
add	`<result> = add <ty> <op1>, <op2>`	/
sub	`<result> = sub <ty> <op1>, <op2>`	/
mul	`<result> = mul <ty> <op1>, <op2>`	/
sdiv	`<result> = sdiv <ty> <op1>, <op2>`	有符号除法
icmp	`<result> = icmp <cond> <ty> <op1>, <op2>`	比较指令
and	`<result> = and <ty> <op1>, <op2>`	与
or	`<result> = or <ty> <op1>, <op2>`	或
call	`<result> = call [ret attrs] <ty> <fnptrval>(<function args>)`	函数调用
alloca	`<result> = alloca <type>`	分配内存
load	`<result> = load <ty>, <ty>* <pointer>`	读取内存
store	`store <ty> <value>, <ty>* <pointer>`	写内存
getelementptr	`<result> = getelementptr <ty>, * {, [inrange] <ty> <idx>}*` `<result> = getelementptr inbounds <ty>, <ty>* <ptrval>{, [inrange] <ty> <idx>}*`	计算目标元素的位置（仅计算）

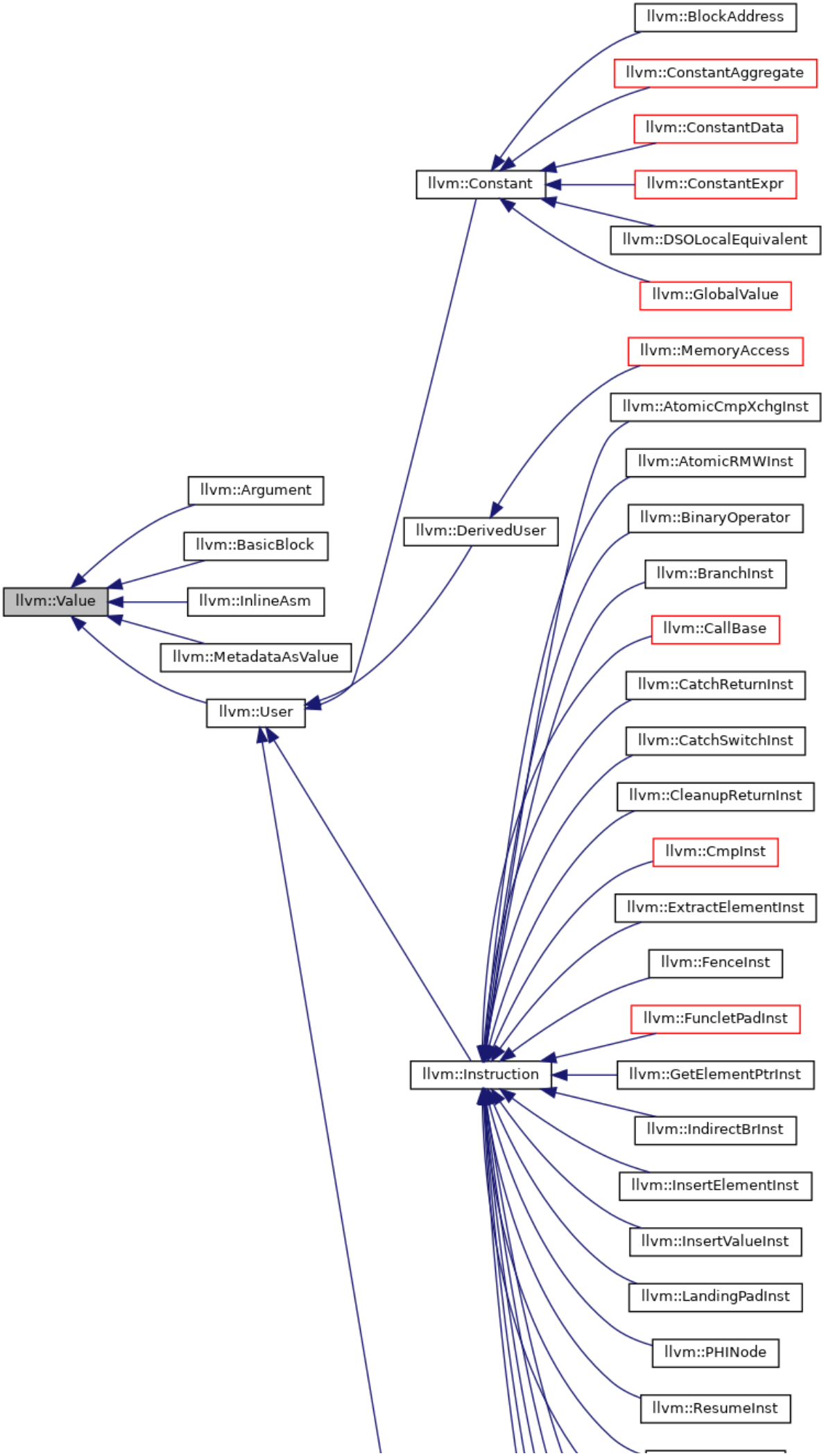
llvm ir	usage	intro
phi	`<result> = phi [fast-math-flags] <ty> [ <val0>, <label0>], ...`	
zext..to	`<result> = zext <ty> <value> to <ty2>`	类型转换，将 `ty` 的 `value` 的 type转换为 `ty2`

terminator instrs

llvm ir	usage	intro
br	`br i1 <cond>, label <iftrue>, label <iffalse>` `br label <dest>`	改变控制流
ret	`ret <type> <value>` ,`ret void`	退出当前函数，并返回值（可选）

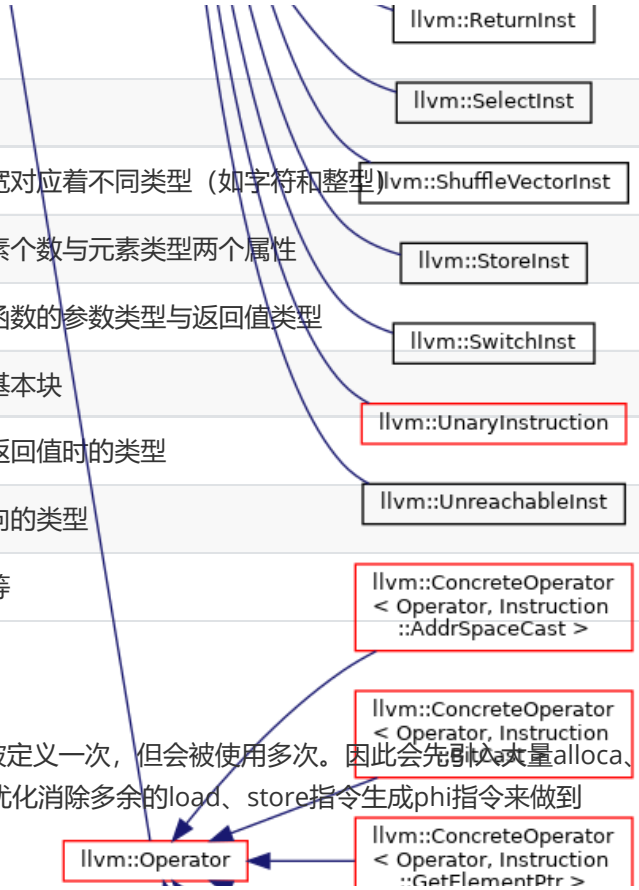
llvm类型

llvm中最重要的概念就是value、use与user，其继承关系如下图：

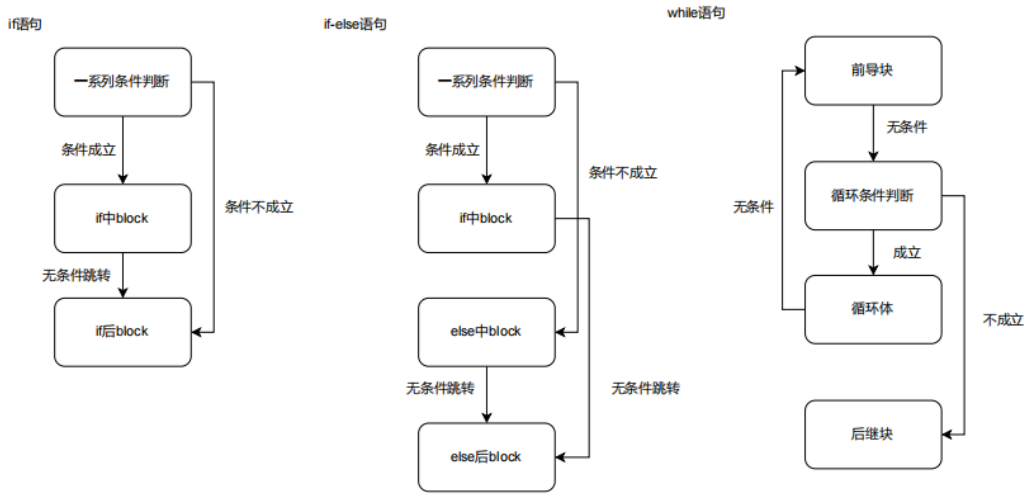

llvm::BlockAddress
llvm::ConstantAggregate
llvm::ConstantData
llvm::Constant
llvm::ConstantExpr
llvm::DSOLocalEquivalent
llvm::GlobalValue
llvm::MemoryAccess
llvm::AtomicCmpXchgInst
llvm::AtomicRMWInst
llvm::Argument
llvm::DerivedUser
llvm::BinaryOperator
llvm::BasicBlock
llvm::BranchInst
llvm::Value
llvm::InlineAsm
llvm::CallBase
llvm::MetadataAsValue
llvm::CatchReturnInst
llvm::User
llvm::CatchSwitchInst
llvm::CleanupReturnInst
llvm::CmpInst
llvm::ExtractElementInst
llvm::FenceInst
llvm::FuncletPadInst
llvm::Instruction
llvm::GetElementPtrInst
llvm::IndirectBrInst
llvm::InsertElementInst
llvm::InsertValueInst
llvm::LandingPadInst
llvm::PHINode
llvm::ResumeInst

这里我们主要关心llvm特有的类型系统：

type	介绍
IntegerType	整数类型，不同位宽对应着不同类型（如字符和整型）
ArrayType	数组类型，包括元素个数与元素类型两个属性
FunctionType	函数类型，会记录函数的参数类型与返回值类型
LabelType	标签类型，对应着基本块
VoidType	空类型，函数没有返回值时的类型
PointerType	指针类型，包含指向的类型
OtherType	其他类型，如指令等

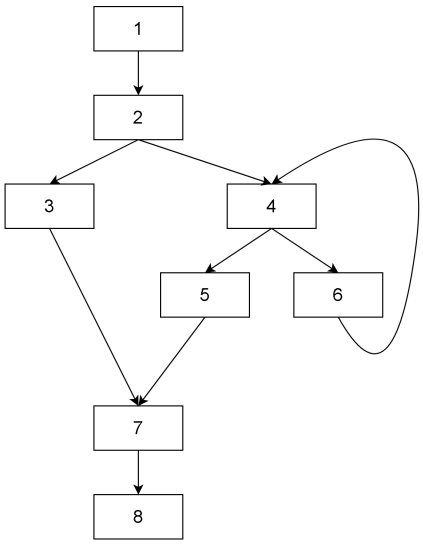
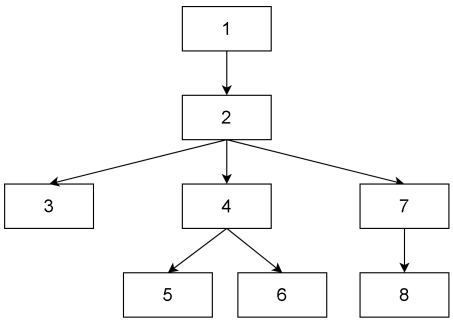


生成llvm

- 必须保证虚拟寄存器的SSA形式，即只会被定义一次，但会被使用多次。因此会先引入大量alloca、load与store指令，再通过终端mem2reg优化消除多余的load、store指令生成phi指令来做到
- 处理循环与分支
  - 

    - 需要特别注意的是短路求值。文法中Cond条件定义如下：
      - Cond → LOrExp
      - LOrExp → LAndExp | LOrExp '||' LAndExp
      - LAndExp → EqExp | LAndExp '&&' EqExp
      - 因此，对于每一个LAndExp与EqExp，我们都需要单独生成一个基本块来判断，并通过正确块与错误块的传递来实现短路求值与正确跳转
- 对break与continue的翻译：
  - 因在翻译时，我们不清楚当前指令在哪一层，故需要设计一个栈式结构来储存break与continue指令，在访问结束后再回填
  - ```
    Stack<ArrayList<BrInst>> backPatch = new Stack<>();
    //...
    //'while' '(' Cond ')' Stmt
    public void visitWhile(Stmt stmt) {
    ```
    ```

```
        BasicBlock parent = curBlock;
        BasicBlock whileBlock =
f.basicBlock(getBlockNum().concat("_while"), curFunction);
        BasicBlock condBlock = f.basicBlock(getBlockNum().concat("_if"),
curFunction);
        BasicBlock nextBlock =
f.basicBlock(getBlockNum().concat("_next"), curFunction);
        backPatch.push(new ArrayList<>());
        f.brInst(null, condBlock, null, parent);
        curBlock = condBlock;
        stmt.whileCond.lOrExp.trueBlock = whileBlock;
        stmt.whileCond.lOrExp.falseBlock = nextBlock;
        visitLOrExp(stmt.whileCond.lOrExp);
        curBlock = whileBlock;
        visitStmt(stmt.stmts.get(0));
        f.brInst(null, condBlock, null, curBlock);
        for (BrInst brInst : backPatch.pop()) {
            if (brInst.judge.equals("continue")) {
                brInst.set(condBlock);
            } else {
                brInst.set(nextBlock);
            }
        }
        curBlock = nextBlock;
    }

    public void visitBreak(Stmt stmt) {
        backPatch.peek().add(f.brInst("break", curBlock));
    }

    public void visitContinue(Stmt stmt) {
        backPatch.peek().add(f.brInst("continue", curBlock));
    }
```

目标mips代码生成

- 生成部分核心在于指令选择与寄存器分配，其将在代码优化部分详细展开。需要额外注意的是运行栈设计。起初我为每个变量分配了固定的内存空间，但后续意识到mips代码在运行时，可能会产生递归函数这种情况，运行时栈会变化，函数中同名局部变量会对应不同地址，因此需要特殊处理。
- 我在函数头与函数尾提前”占位“生成$sp的移动指令，在遍历完函数的代码后便可以知道函数定义了多少变量和数组，便也知道了函数所占用的栈大小，再把栈大小回填到预先生成的移动$sp的指令当中。在函数内部分配每个变量的偏移量即可。
- 对于全局变量，存在.data区，使用$gp寄存器访存；对应局部变量，使用$sp寄存器访存；而对于中间代码的gep指令，会运算出其具体地址

代码优化设计

函数内联

函数内联时需要将寄存器中值全部写回内存，调用结束还需要写回，调用函数本身还会对栈移动，以上会生成大量访存指令；此外还需要传参，因此开销很大。

函数内联是指对非递归函数，不再调用，而是直接将其内联到调用者函数中。其具体实现如下：

- 传参使用assign实现（具体为add fParam,rParam,0），而返回值可能存在多个，因此使用phi指令实现
- 对于原函数中的跳转指令，因为块的流图问题，无论是顺序遍历、DFS还是BFS，都可能存在跳转到还未遍历到的基本块当中的问题，因此需要像目标代码生成中移动$sp指令一样，先占位生成跳转指令，在原函数内联完成后，再将基本块回填
- 变量名字会重复，因此需要重命名

划分基本块、建立CFG与基本块合并

基本块的概念与llvm ir中基本块的定义相同，其从块的开头顺序执行，直到最后一条跳转或返回指令。其划分在生成llvm时已经完成，而跳转关系也已经通过br指令建立起来了，因此我们可以直接使用并在此基础上进行后续优化。

而对于基本块的合并，如果控制流图中存在a→b，且a的出度与b入度均恰好为1，即可以合并。合并时将b开头的phi指令转换为assign语句，再去掉跳转指令即可。由于llvm基本块与函数内联生成时会生成很多这种入度出度均为1的基本块，合并可以有效减少基本块数量以及phi指令数量

mem2reg

Φ函数定义

在 mem2reg 中，LLVM 会识别出局部变量的 `alloca` 指令，将对应的局部变量改为虚拟寄存器中的 SSA 形式的变量，将该变量的 `store/load` 修改为虚拟寄存器之间的 def-use/use-def 关系，并在适当的地方加入 `phi` 指令和进行变量的重命名

$x_k \leftarrow \phi(x_1, \dots x_n)$，Φ函数被放置于基本块的首端，右侧每一个变量对应一个基本块，其代表着不同的基本块移动到当前基本块时x_k的不同取值

有关基本块的定义

- **定义**：对变量进行初始化、赋值等改变变量的值的行为。
- **使用**：在语句/指令中将变量的值作为参数的行为。
- **控制流图**（Control Flow Graph, CFG）：一个程序中所有基本块执行的可能流向图，图中的每个节点代表一个基本块，有向边代表基本块间的跳转关系. CFG 有一个入口基本块和一/多个出口基本块，分别对应程序的开始和终止.
- **支配**（dominate）：对于 CFG 中的节点 n1 和 n2，**n1 支配 n2** 当且仅当所有从入口节点到 n2 的路径中都包含 n1，即 **n1 是从入口节点到 n2 的必经节点 每个基本块都支配自身**
- **严格支配**（strictly dominate）：n1 严格支配 n2 当且仅当 n1 支配 n2 且 n_1 != n_2
- **直接支配者**（immediate dominator, idom）：节点 n 的直接支配者是**离 n 最近的严格支配 n 的节点**（标准定义是：**严格支配 n**，且不严格支配**任何严格支配 n 的节点**的节点）. 入口节点以外的节点都有直接支配者. 节点之间的直接支配关系可以形成一棵支配树（dominator tree）.

控制流图



支配树



- **支配边界**（dominance frontier）：节点 n 的支配边界是 CFG 中**刚好**不被 n 支配到的节点集合. 形式化一点的定义是：节点 n 的支配边界 DF(n) = {x | n 支配 x 的前驱节点(左图)，n 不严格支配 x}

插入Φ指令

- 如果变量x 在基本块b处有定义，那么所有 DF(*b*) 中的点都需要插入关于变量x 的Φ函数
- DF(*b*) 中的基本块也有关于变量x 的定义，因此需要继续遍历其支配边界进行Φ 函数的插入，直到迭代到不动点

变量重命名

- 遍历所有基本块，在每个基本块中分别遍历每条指令
- 遇到局部 `int` 类型变量对应的 `alloca` 指令，直接删除即可
- 遇到局部 `int` 类型变量对应的 `load` 指令，将所有其他指令中对该指令的使用替换为对该变量到达定义的使用，删除 `load` 指令
- 遇到局部 `int` 类型变量对应的 `store` 指令，需要更新该变量的到达定义，删除 `store` 指令
- 遍历完成一个基本块中的所有指令后，维护该基本块的所有后继基本块中的 `phi` 指令，将对应来自此基本块的值设为对应变量的到达定义

寄存器分配

- 临时寄存器分配：使用LRU算法维护一个寄存器池，如果有空寄存器则直接分配；否则分配权值最小的寄存器，并将其权值设置为最大
- 寄存器写回
  - 当一个虚拟寄存器占用是实际寄存器要分配给其他变量时，其可能需要写回
  - 写回的判断标准是，该变量在当前基本块被定义，且后面可能会被用到
  - 采用dfs方法遍历后续可能到达的中间代码，如果有指令使用了该变量，则需要写回
- 跨基本块分配：

指令选择

init方法，将一个常数赋给寄存器

```
    public void init(String reg, int value) {
        if (-32768 <= value && value <= 32767) {
            new RegImmInstr(curBlock, "addiu", reg, "$0", value);
        } else if (0 <= value && value <= 65535) {
            new RegImmInstr(curBlock, "ori", reg, "$0", value);
        } else if ((value & 0xffff) == 0) {
            new LuiInstr(curBlock, "lui", reg, (value >> 16) & 0xffff);
        } else {
            new LuiInstr(curBlock, "lui", reg, (value >> 16) & 0xffff);
            new RegImmInstr(curBlock, "ori", reg, reg, value & 0xffff);
        }
    }
```

add指令的翻译

```
public void add(Instruction instr) {
        Value left = instr.operands.get(0);
        Value right = instr.operands.get(1);
        String name = reg();
        if (left instanceof ConstantInt && right instanceof ConstantInt) {
            int value = ((ConstantInt) left).value + ((ConstantInt) right).value;
            init(name, value);
        } else if (left instanceof ConstantInt) {
            int value = ((ConstantInt) left).value;
            if (Short.MIN_VALUE <= value && value <= Short.MAX_VALUE) {
                new RegImmInstr(curBlock, "addiu", name, regs.get(right.name),
value);
            } else {
                init("$1", value);
                new RegRegInstr(curBlock, "addu", name, regs.get(right.name),
"$1");
            }
        } else if (right instanceof ConstantInt) {
            int value = ((ConstantInt) right).value;
            if (Short.MIN_VALUE <= value && value <= Short.MAX_VALUE) {
                new RegImmInstr(curBlock, "addiu", name, regs.get(left.name),
value);
            } else {
                init("$1", value);
                new RegRegInstr(curBlock, "addu", name, regs.get(left.name),
"$1");
            }
        } else {
            new RegRegInstr(curBlock, "add", name, regs.get(left.name),
regs.get(right.name));
        }
        regs.put(instr.name, name);
    }
```

shorterSeq，判断一个寄存器与一个常数是否相等

```
    public void shorterSeq(String reg, int value) {
        if (-Short.MAX_VALUE <= value && value <= -Short.MIN_VALUE) {
            new RegImmInstr(curBlock, "addiu", "$1", reg, -value);
        } else if (0 <= value && value <= Short.MAX_VALUE - Short.MIN_VALUE) {
            new RegImmInstr(curBlock, "xori", "$1", reg, value);
        } else {
            init("$1", value);
            new RegRegInstr(curBlock, "xor", "$1", reg, "$1");
        }
    }
```

llvm ir	mips
`<result> = add <ty> <op1>, <op2>`	如果op1与op2均为常数，则计算出后init；如果其中一个为常数，根据其值的范围判断是addiu还是add；如果两个均非常数，则直接相加即可
`<result> = sub <ty> <op1>, <op2>`	与add相似
`<result> = mul <ty> <op1>, <op2>`	如果有常数需要先addiu
`<result> = sdiv <ty> <op1>, <op2>`	如果有常数需要先addiu
`<result> = srem <ty> <op1>, <op2>`	如果有常数需要先addiu
`<result> = icmp <cond> <ty> <op1>, <op2>`	slt指令可以直接翻译，结合xori指令来取反，可以翻译出slt、sle、sgt、sge、sne与seq六种指令
`<result> = call [ret attrs] <ty> <fnptrval>(<function args>)`	做好函数参数的传递与返回值的取出即可，同时保护需要保护的寄存器
`<result> = alloca <type>`	在栈上分配一个区域即可
`<result> = load <ty>, <ty>* <pointer>`	直接翻译
`store <ty> <value>, <ty>* <pointer>`	直接翻译
`<result> = getelementptr <ty>, * {, [inrange] <ty> <idx>}*` `<result> = getelementptr inbounds <ty>, <ty>* <ptrval> {, [inrange] <ty> <idx>}*`	将原内存中储存的指针值取出来，再计算新指针的地址
`<result> = phi [fast-math-flags] <ty> [ <val0>, <label0>], ...`	在mem2reg部分
= zext to	直接assign即可
`br i1 <cond>, label <iftrue>, label <iffalse>` `br label <dest>`	会生成branch与jump两条指令来对应两个后继块；一个后继块时直接jump即可

llvm ir	mips
`ret <type> <value>`,`ret void`	回写$ra地址，如有返回值再将其赋给$v0

窥孔优化

存在一些较为明显的优化：

- 循环优化：
  - ```
    label1:
    if !<Cond> goto label2
    <Stmt>
    goto label1
    label2:

    优化后：
    if !<Cond> goto label2
    label1:
    <Stmt>
    if <Cond> goto label1
    label2:
    ```
- 运算优化：
  - 加减乘运算中一个运算对象是常数0，可以直接赋值
  - 乘法运算中一个运算对象是 1，可以直接赋值
  - 除法运算，被除数是 0 或除数是 1，可以直接赋值

运算强度削弱

乘法优化

- 对于变量乘常数
  - 乘法代价为4，因此常数值小于5时，展开为加法
  - 对二的整数次幂可以用位移指令代替
- 两变量相乘时，只能直接相乘

除法优化

- 两变量相乘时，同样只能直接相除
- 变量除以常数
  - 除数是二的整数次幂，转化为位移指令
  - 任意常数时，可以转化为乘法与右移，我参考的主要是该文章：https://zhuanlan.zhihu.com/p/151038723
    ```

取模优化

$$x \bmod y = x - x \left\lfloor \frac{x}{y} \right\rfloor$$

因此可以转化为减法与除法优化；当x<y时，x mod y = x；由于除法指令代价为50，因此我们生成几条指令来取代是很有价值的

学期总结

从开发的角度而言，文法与中间代码、汇编代码的各种要求就像是未来客户提出的业务需求，面对各种文法中各种情况完成需求的过程，本身就是一个满足各种业务进行的迭代开发过程。因而开发编译器极大丰富了我项目开发的经验。此外，编译本身属于体系结构领域，编译器设计也让我重温了C与汇编等若干底层的知识，对程序的编译运行有了更深刻的理解

对语言特性更加熟悉

上学期OO课程只是让我们初步体验到了JAVA面向对象开发，其题目、需求较为固定；而这学期编译器的开发中，需求多种多样、不断增加，经常需要局部增加一些功能函数。这倒逼我去了解JAVA更多的语言特性。如在对基本块进行BFS，转化break、continue等语句时使用了内置栈、队列等结构，而不是是像上学期局限于ArrayList、HashMap；在产生llvm中间代码的时候更熟悉了继承与接口、采用工程模式等设计模式来开发、同时进一步学习了JAVA深浅克隆方面的语言特性。

包管理与版本管理

OO课程中类的数量比较少，可以放在同一个文件夹下；而编译器设计会使用上百个类，它们分别隶属于不同的功能模块，因此课设锻炼了我包管理的习惯；此外，我也第一次用git来管理自己的代码，在出现负优化以及错误的debug时，版本回退可以有效管理代码，这比本地存很多文件有效得多，对每一次commit命名也让我更熟悉了开发的历程。

解耦

计组课程中，尽管我们已知悉“高内聚、低耦合”的设计原则，但受开发能力所限、以及为了满足应试的需求，模块之间的解耦不够彻底。在编译器开发中，我彻底实践了解耦的原则。具体表现为，将整个编译器划分为几个模块，不同模块间仅保留易于交互的数据结构。这样的好处在于，在添加新功能进行开发时，只会对项目的一部分进行修改，出现错误时也便于调试，而不会牵一发而动全身。而这种设计模式的关键在于，对交互部分数据结构都定义。这可以看作为两模块之间的协议或者是接口。这一点和本学期数据库大作业的前后端解耦开发非常像。只有设计了合理的交互结构，才能让开发更规范，同时可以更好地满足新的功能。

设计优先

有时候我们往往着急去编码，但就我实践的经验来看，好的设计可以有效减少编码时间、无效代码量以及后期debug时间。对编译实验来说，先设计后编码的一个挑战是：之前没有类似的项目经验，除了词法分析较为简单以及语法分析有OO的递归下降经验外，其他只能边学边写，有些时候难以避免做一些无用功。但是在总体上仍然要坚持先设计后编码。

体系结构有关知识

编译帮我有效复习了C语言与mips等若干底层知识，如C语言中一维数组头实际上是一个int型指针，二维数组头实际是一个指向一维数组的指针；此外还有短路求值的具体实现：在中间代码与汇编代码时插入多个基本块；还接触到了工业界的编译中间代码——llvm，以及类型系统等诸多概念；还温习了mips代码中数据代码若干规范、栈寄存器维护等若干知识……以上种种让我对计算机最底层的知识有了更好的掌握，也对程序运行有了更深入的理解