编译原理实验报告：stage-1

- 于新雨 计25 2022010841

step1

思考题

1. 可以，因为在main.py里面删去step_tac里面的namer.transform(p),typer.transform(p)函数不受影响，namer主要与符号表的构建有关，typer主要与类型检查有关，现阶段暂不涉及这方面问题，所以可以删去
2. 在parse过程的词法语法分析中处理，报错为

> Syntax error: line 2, column 7 return; Syntax error: line 3, column 1 } Syntax error: EOF

1. 同时定义`frontend/ast/tree.py:Unary`, `utils/tac/tacop.py:TacUnaryOp`, `utils/riscv.py:RvUnaryOp`是因为会在不同的编译步骤中被使用，如tree.py主要是与AST抽象语法树有关，其中的Unary是AST中节点类型，tacop.py主要是与三地址码有关，其中的TacUnaryOp是三地址码用于生成单目运算表达式的类，RvUnaryOp是RISC-V汇编语言中用于生成单目运算表达式的指令，所以定义了三种不同的单目运算符类型用于不同地方

step2

新增代码

- 首先分析各文件的用途，知道中间代码主要由frontend/tacgen中的tacgen.py产生，于是仿照`visitUnary`函数中的提示

```
    op = {
            node.UnaryOp.Neg: tacop.TacUnaryOp.NEG,
            # You can add unary operations here.
        }[expr.op]
```

在`visitUnary`函数中添加代码

```
node.UnaryOp.LogicNot:tacop.TacUnaryOp.SEQZ,
node.UnaryOp.BitNot:tacop.TacUnaryOp.NOT,
```

- 然后为了三地址码的正常生成，继续修改utils，根据文件tacop.py里NEG的用法在TacUnaryOp里增加如下定义

```
class TacUnaryOp(Enum):
    NEG = auto()
```

```
    SEQZ = auto()
    NOT = auto()
```

- 继续为了输出正确三地址码，增加tacinstr.py中的Unary类中的`__str__()`支持的单目运算符种类：

```
    s=""
    if self.op == TacUnaryOp.NEG:
        s="-"
    elif self.op==TacUnaryOp.SEQZ:
        s="!"
    else:
        s="~"
```

- 至此，应该可以支持三地址码生成
- 现在为了后端增加对riscv代码的生成，先在utils里的riscv.py中增加单目运算符种类

```
    SEQZ=auto()
    NOT=auto()
```

- 然后修改riscvasmemitter.py中的`visitUnary`函数，增加对单目运算符的生成

```
    TacUnaryOp.SEQZ:RvUnaryOp.SEQZ,
    TacUnaryOp.NOT:RvUnaryOp.NOT,
```

思考题

- -~2147483647
- INT_MAX是2147483647，INT_MIN是-2147483648，INT_MAX按位取反可得INT_MIN,但是INT_MIN数学上的相反数比INT_MAX大一，所以取负时溢出

step3

新增代码

大部分操作和单目运算符比较类似

- 对于三地址码产生，先修改tacgen.py里面的visitBinary支持的op,类似单目运算符操作增加如下

```
            node.BinaryOp.Sub:tacop.TacBinaryOp.SUB,
            node.BinaryOp.Div:tacop.TacBinaryOp.DIV,
            node.BinaryOp.Mul:tacop.TacBinaryOp.MUL,
            node.BinaryOp.Mod:tacop.TacBinaryOp.MOD,
```

- 同时修改Utils里面的class TacBinaryOp

```
    SUB = auto()
    MUL=auto()
    DIV=auto()
    MOD=auto()
```

- 因为tacinstr.py里Binary类的__str__()函数支持本stage里所有二元运算符种类，故不用修改
- 为后端增加riscv代码生成，修改riscvasmemitter.py中的visitBinary函数，增加如下：

```
        TacBinaryOp.SUB:RvBinaryOp.SUB,
        TacBinaryOp.DIV:RvBinaryOp.DIV,
        TacBinaryOp.MUL:RvBinaryOp.MUL,
        TacBinaryOp.MOD:RvBinaryOp.REM,
```

- 同时增加riscv.py中的RvBinaryOp支持的种类

```
    SUB=auto()
    MUL=auto()
    DIV=auto()
    REM=auto()
```

思考题

- 另一种情况为整数除法的被除数为INT_MIN,除数为-1，由于INT_MIN数学意义上的相反数比INT_MAX大一，故可能溢出，故结果未定义
- 本机x86-64下wsl2测试，结果会正常编译但是运行时抛出异常为Floating point exception
- RISCV-32 的 qemu 模拟器中输出为INT_MIN除以-1的结果为-2147483648

step4

新增代码

- 首先还是和前面step类似，修改tacgen.py里面的visitBinary支持的op

```
            node.BinaryOp.EQ:tacop.TacBinaryOp.EQU,
            node.BinaryOp.NE:tacop.TacBinaryOp.NEQ,
            node.BinaryOp.BitAnd:tacop.TacBinaryOp.AND,
            node.BinaryOp.BitOr:tacop.TacBinaryOp.OR,
            node.BinaryOp.LT:tacop.TacBinaryOp.SLT,
            node.BinaryOp.LE:tacop.TacBinaryOp.LEQ,
            node.BinaryOp.GT:tacop.TacBinaryOp.SGT,
            node.BinaryOp.GE:tacop.TacBinaryOp.GEQ,
            node.BinaryOp.LogicAnd:tacop.TacBinaryOp.LAND,
            node.BinaryOp.LogicOr:tacop.TacBinaryOp.LOR,
```

- 同时在tacop.py中新增对应的TacBinaryOp

```
    MOD = auto()
    EQU = auto()
    NEQ = auto()
    SLT = auto()
    LEQ = auto()
    SGT = auto()
    GEQ = auto()
    LAND = auto()
    AND = auto()
    OR = auto()
```

- 同时修改`tacinstr.py`,增加如下

```
TacBinaryOp.LAND:"&",
TacBinaryOp.LOR:"|",
```

- riscv.py中，同样增加RvBinaryOp.AND和RvBinaryOp.SGT和RvBinaryOp.SLT

```
    AND = auto()
    SGT=auto()
    SLT=auto()
```

- 本步与前几个step相比稍微复杂的是riscvasmemitter.py的修改，主要是可以只用单条指令表示的只有AND,OR,SGT,SLT运算符，所以其他运算符需要用多于一条指令实现，主要是在riscvasmemitter.py里面实现更改
- 首先像前几步一样增加对AND,OR,SGT,SLT支持

```
                    TacBinaryOp.AND:RvBinaryOp.AND,
                    TacBinaryOp.OR:RvBinaryOp.OR,
                    TacBinaryOp.SGT:RvBinaryOp.SGT,
                    TacBinaryOp.SLT:RvBinaryOp.SLT,
```

- 接着在函数一开始加入对instr.op是否为EQU,NEQ,LEQ,GEQ等判断，并且按照算数规则加入相应汇编代码可以注意其中汇编代码可按照文档里先写个小程序再用`riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 foo.c -S -O3 -o foo.s`的命令行查看汇编代码的实现

```
            elif instr.op==TacBinaryOp.EQU:
                self.seq.append(Riscv.Binary(RvBinaryOp.SUB, instr.dst, instr.lhs,
instr.rhs))
                self.seq.append(Riscv.Unary(RvUnaryOp.SEQZ, instr.dst, instr.dst))
```

```
            elif instr.op==TacBinaryOp.NEQ:
                self.seq.append(Riscv.Binary(RvBinaryOp.SUB, instr.dst, instr.lhs,
instr.rhs))
                self.seq.append(Riscv.Unary(RvUnaryOp.SNEZ, instr.dst, instr.dst))
            elif instr.op==TacBinaryOp.LEQ:
                self.seq.append(Riscv.Binary(RvBinaryOp.SGT, instr.dst, instr.lhs,
instr.rhs))
                self.seq.append(Riscv.Unary(RvUnaryOp.SEQZ, instr.dst, instr.dst))
            elif instr.op==TacBinaryOp.GEQ:
                self.seq.append(Riscv.Binary(RvBinaryOp.SLT, instr.dst, instr.lhs,
instr.rhs))
                self.seq.append(Riscv.Unary(RvUnaryOp.SEQZ, instr.dst, instr.dst))
            elif instr.op==TacBinaryOp.LAND:
                self.seq.append(Riscv.Unary(RvUnaryOp.SNEZ, instr.dst, instr.lhs))
                self.seq.append(Riscv.Unary(RvUnaryOp.NEG,instr.dst,instr.dst))

self.seq.append(Riscv.Binary(RvBinaryOp.AND,instr.dst,instr.dst,instr.rhs))
                self.seq.append(Riscv.Unary(RvUnaryOp.SNEZ,instr.dst,instr.dst))
```

思考题

- 短路求值的受欢迎：很大程度是因为可以提升性能，利用逻辑或，逻辑与等逻辑运算符本身的性质减少执行的指令数。
- 给程序员带来的好处：
  - 可以使得代码更加简洁可读，比如可以在if的条件判断写成逻辑与的形式，而非用两层if判断，增加代码可读性
  - 可以使得代码保证安全性的情况下更加简洁，就如下列语句

```
if((i<32)&&(i>=0)&&(array[i]>=0)){
    //do something
}
```

该语句是表达在大小为32的array数组中以array[i]作为条件进行判断，表述简洁，同时由于短路求值不会出现数组越界访问的安全问题