Stage-1：常量表达式

陈炫中　2019011236

- Step 2

实验内容

给整数常量增加一元运算：取负 `-`、按位取反 `~` 以及逻辑非 `!`。

通过借鉴取负的实现方法，在 Python 框架中的 Unary 一元运算节点中加入逻辑非和按位非：

```
    def visitUnary(self, expr: Unary, mv: FuncVisitor) -> None:
        expr.operand.accept(self, mv)

        op = {
            node.UnaryOp.Neg: tacop.UnaryOp.NEG,
            # You can add unary operations here.
            node.UnaryOp.BitNot: tacop.UnaryOp.NOT,
            node.UnaryOp.LogicNot: tacop.UnaryOp.SEQZ,
        }[expr.op]
        expr.setattr("val", mv.visitUnary(op, expr.operand.getattr("val")))
```

思考题

编写如下 minidecaf 表达式：

```
int main() {
    return -~2147483647;
}
```

理论预算结果为 `2147483648`，实际输出为 `-2147483648`，发生越界。

- Step 3

实验内容

增加加 `+`、减 `-`、乘 `*`、整除 `/`、模 `%` 以及括号 `(` `)`。

类似于 Step 2，参考二元加法的实现方法，在 Binary 二元运算节点中加入减、乘、除以及模运算符：

```
    def visitBinary(self, expr: Binary, mv: FuncVisitor) -> None:
        expr.lhs.accept(self, mv)
        expr.rhs.accept(self, mv)

        op = {
            node.BinaryOp.Add: tacop.BinaryOp.ADD,
            # You can add binary operations here.
            # Step 3
            node.BinaryOp.Sub: tacop.BinaryOp.SUB,
            node.BinaryOp.Mul: tacop.BinaryOp.MUL,
            node.BinaryOp.Div: tacop.BinaryOp.DIV,
            node.BinaryOp.Mod: tacop.BinaryOp.REM,
            ……
```

思考题

左操作数为`-2147483648`，右操作数为`-1`，存在未定义行为

在 x86-64 （Ubuntu 18.04 gcc）上的运行结果：`Floating point exception`

在 RISCV-32 的 qemu 模拟器中的运行结果：`-2147483648`

- Step 4

实验内容

增加比较大小和相等的二元操作：`<`、`<=`、`>=`，`>`，`==`，`!=`和逻辑与 `&&`、逻辑或 `||`。

其中，`<`，`>`，`&&`和`||`可直接同 step3 中引入的二元运算节点那样直接加入，在这里就不加以赘述；

而`<=`、`>=`，`==`和`!=`因为没有对应的直接实现的指令，需要借助中间指令，选择合适的 RISC-V 指令来完成翻译工作，因此在`visitBinary`函数中进行修改，具体实现如下：

```
def visitBinary(self, instr: Binary) -> None:
    if instr.op == BinaryOp.EQU:
        self.seq.append(Riscv.Binary(BinaryOp.SUB, instr.dst, instr.lhs,
instr.rhs))
        self.seq.append(Riscv.Unary(UnaryOp.SEQZ, instr.dst, instr.dst))
    elif instr.op == BinaryOp.GEQ:
        self.seq.append(Riscv.Binary(BinaryOp.SLT, instr.dst, instr.lhs,
instr.rhs))
        self.seq.append(Riscv.Unary(UnaryOp.SEQZ, instr.dst, instr.dst))
    elif instr.op == BinaryOp.LEQ:
        self.seq.append(Riscv.Binary(BinaryOp.SGT, instr.dst, instr.lhs,
instr.rhs))
        self.seq.append(Riscv.Unary(UnaryOp.SEQZ, instr.dst, instr.dst))
    elif instr.op == BinaryOp.NEQ:
        self.seq.append(Riscv.Binary(BinaryOp.SUB, instr.dst, instr.lhs,
instr.rhs))
        self.seq.append(Riscv.Unary(UnaryOp.SNEZ, instr.dst, instr.dst))
    else:
        self.seq.append(Riscv.Binary(instr.op, instr.dst, instr.lhs, instr.rhs))
```

思考题

短路求值这一特性广受欢迎的原因：

编程人员可以利用短路求值这一特性去更加精细地控制程序的行为。

它给程序员带来的好处：

1. 空引用检查

```
if(string != null && string.isEmpty())
{
    //we check for string being null before calling isEmpty()
}
```

2. 前置可能触发的条件从而节省计算

```
Boolean b = true;
if(b || foo.timeConsumingCall())
{
   //we entered without calling timeConsumingCall()
}
```