stage1 实验报告

姓名：郑友捷　学号：2021010771

工作内容

step2：一元操作

为了实现一元操作符，本程序完善了`frontend/tacgen/tacgen.py`中的`visitUnary`方法，在生成中间代码时，对一元操作符的选择范围扩展到`!/~/-`三种一元操作符，对应代码部分为：

```
def visitUnary(self, expr: Unary, mv: FuncVisitor) -> None:
    expr.operand.accept(self, mv)
    # 先解析表达式，再解析符号
    op = {
        node.UnaryOp.Neg: tacop.UnaryOp.NEG,
        node.UnaryOp.BitNot: tacop.UnaryOp.NOT,
        node.UnaryOp.LogicNot: tacop.UnaryOp.SEQZ
    }[expr.op]
    expr.setattr("val", mv.visitUnary(op, expr.operand.getattr("val")))
```

step3：加减乘除模

为了实现加减乘除模操作，主要思路与step2思路一致，对相同文件的`visitBinary`方法进行修改，在生成中间代码时扩展二元操作符的选择范围，使其可以支持加减乘除取模等操作。

```
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,
            node.BinaryOp.Sub: tacop.BinaryOp.SUB,
            node.BinaryOp.Mul: tacop.BinaryOp.MUL,
            node.BinaryOp.Div: tacop.BinaryOp.DIV,
            node.BinaryOp.Mod: tacop.BinaryOp.REM,
        }[expr.op]
        expr.setattr(
            "val", mv.visitBinary(op, expr.lhs.getattr("val"),
expr.rhs.getattr("val"))
        )
```

step4：比较和逻辑表达式

step4与step3均是要求编译器支持二元操作符，但step4的困难在于所要求的操作并非仅由一条汇编代码可以完成。因此需要对`visitBinary`方法进行修改，对于某一些无法用一条汇编完成的操作符（如等号）使用多条汇编代码完成对应操作。

特殊操作符与对应操作如下：

- 判等`a==b`：先用a减去b，将值存储在c中。判断c是否不为0。

对应汇编码为：（结果存储在t1寄存器中）

```
sub t2 t0 t1
seqz t3 t2
```

对应代码为：

```
if expr.op == node.BinaryOp.EQ:
    new_temp =  mv.visitBinary(tacop.BinaryOp.SUB, expr.lhs.getattr("val"),
expr.rhs.getattr("val"))
    # new_temp为两者之差所在的寄存器名称
    expr.setattr("val", new_temp)
    expr.setattr("val", mv.visitUnary(tacop.UnaryOp.SEQZ, new_temp))
    # 判断new_temp所存储的值是否为0
```

- 判不等`!=`：思路与判等相似，故不加赘述。
- 判小于等于`<=`：先用a减去b，将值存储在c中。接下来设布尔值d代表c是否大于0。最后对d取逻辑反即可得到`<=`的结果。

  对应汇编码为：

  ```
  sub t2 t0 t1
  sgtz t3 t2
  seqz t4 t3
  ```

  对应代码为：

  ```
  if expr.op == node.BinaryOp.LE:
      new_temp =  mv.visitBinary(tacop.BinaryOp.SUB, expr.lhs.getattr("val"),
  expr.rhs.getattr("val"))
      expr.setattr("val", new_temp)
      # 对应sub操作
      new_temp = mv.visitUnary(tacop.UnaryOp.SGTZ, new_temp)
      expr.setattr("val", new_temp)
      # 对应sgtz操作
      expr.setattr("val", mv.visitUnary(tacop.UnaryOp.SEQZ, new_temp))
      # 对应seqz操作
  ```

其余操作符均可以由一条汇编码完成，直接在step3的基础上往字典中添加键值对即可，不在此赘述。

思考题

step2

操作表达式如下：

```
-~2147483647
```

step3

左操作数为-2147483648，右操作数为-1。

代码为

```
#include <stdio.h>

int main() {
  int a = -2147483648;
  int b = -1;
  printf("%d\n", a / b);
  return 0;
}
```

电脑架构为x86-64，运行环境为Ubuntu20-04，此时在RISCV-32 的 qemu 模拟器中编译运行代码，结果为-2147483648。

step4

因为在支持短路求值的情况下，若左表达式不成立，触发了短路求值，则不会运行右表达式。这一特性可以带来非常多的好处，包括：

- 有时右表达式需要依赖左表达式的成立，若先执行的不成立而计算后执行的可能导致运行错误。此时若不支持短路求值，可能会导致难以预料的错误。
- 同上，若左右表达式存在依赖关系，如果支持短路求值，就不需要把这些表达式单独写做两条运行语句严格保证运行前后顺序，而是可以一起写在一个逻辑符两侧，从而使代码简洁。
- 很多场景中，短路求值两侧的表达式可能各自计算量都非常大或者其中一条计算量很大，此时使用短路求值，可以免去其中一个大计算量的表达式执行，提高算法时间效率。

由于上述等多种优点，短路求值广受程序员欢迎。