姚季涵 2018010305

一、实验内容

- Step1

本实验主要在 tacgen.py 中添加对一元操作符的解析：

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

    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")))
```

其中一元操作符对应的汇编代码分别为：

- a : neg t0, t0

~ a : not t0, t0

! a : seqz t0, t0

由于后端对于三段码的一元操作符的解析已经写好，没有进行更改。

- Step2、Step3

本实验主要在 tacgen.py 中添加对二元操作符的解析：

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

    if expr.op == node.BinaryOp.LogicOr:
        newReg = mv.visitBinary(tacop.BinaryOp.OR, expr.lhs.getattr("val"), expr.rhs.getattr("val"))
        expr.setattr("val", mv.visitUnarySelf(tacop.UnaryOp.SNEZ, newReg))
    elif expr.op == node.BinaryOp.LogicAnd:
        newReg1 = mv.visitUnary(tacop.UnaryOp.SNEZ, expr.lhs.getattr("val"))
        newReg2 = mv.visitUnary(tacop.UnaryOp.SNEZ, expr.rhs.getattr("val"))
        expr.setattr("val", mv.visitBinarySelf(tacop.BinaryOp.AND, newReg1, newReg2))
    elif expr.op == node.BinaryOp.LE:
        newReg = mv.visitBinary(tacop.BinaryOp.SGT, expr.lhs.getattr("val"), expr.rhs.getattr("val"))
        expr.setattr("val", mv.visitUnarySelf(tacop.UnaryOp.SEQZ, newReg))
    elif expr.op == node.BinaryOp.GE:
        newReg = mv.visitBinary(tacop.BinaryOp.SLT, expr.lhs.getattr("val"), expr.rhs.getattr("val"))
        expr.setattr("val", mv.visitUnarySelf(tacop.UnaryOp.SEQZ, newReg))
    elif expr.op == node.BinaryOp.EQ:
        newReg = mv.visitBinary(tacop.BinaryOp.SUB, expr.lhs.getattr("val"), expr.rhs.getattr("val"))
        expr.setattr("val", mv.visitUnarySelf(tacop.UnaryOp.SEQZ, newReg))
    elif expr.op == node.BinaryOp.NE:
        newReg = mv.visitBinary(tacop.BinaryOp.SUB, expr.lhs.getattr("val"), expr.rhs.getattr("val"))
        expr.setattr("val", mv.visitUnarySelf(tacop.UnaryOp.SNEZ, newReg))
    else:
        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,
            node.BinaryOp.LT: tacop.BinaryOp.SLT,
```

其中二元操作符对应的汇编代码分别为：

a + b : add t2, t0, t1

a - b : sub t2, t0, t1

a * b : mul t2, t0, t1

a / b : div t2, t0, t1

a % b : rem t2, t0, t1

a < b : slt t2, t0, t1

a > b : sgt t2, t0, t1

a <= b : sgt t2, t0, t1; seqz t2, t2

a >= b : slt t2, t0, t1; sqez t2, t2;

a == b : sub t2, t0, t1; seqz t2, t2;

a != b : sub t2, t0, t1; snez t2, t2;

a && b : snez t2, t0; snez t3, t1; and t3, t2, t3;

a || b : or t2, t0, t1; snez t2, t2;

由于后端对于三段码的一元、二元操作符的解析已经写好，没有进行更改。

二、思考题

- 我们在语义规范中规定整数运算越界是未定义行为，运算越界可以简单理解成理论上的运算结果没有办法保存在 32 位整数的空间中，必须截断高于 32 位的内容。请设计一个 minidecaf 表达式，只使用-~!这三个单目运算符和从 0 到 2147483647 范围内的非负整数，使得运算过程中发生越界。

  一个越界的例子是 -0。-0 = ~0 + 1，0 取反后为 0xffffffff，再加 1 就会发生越界。

- 我们知道“除数为零的除法是未定义行为”，但是即使除法的右操作数不是 0，仍然可能存在未定义行为。请问这时除法的左操作数和右操作数分别是什么？请将这时除法的左操作数和右操作数填入下面的代码中，分别在你的电脑（请标明你的电脑的架构，比如 x86-64 或 ARM）中和 RISCV-32 的 qemu 模拟器中编译运行下面的代码，并给出运行结果。（编译时请不要开启任何编译优化）

```c
#include <stdio.h>

int main() {
  int a = 左操作数;
  int b = 右操作数;
  printf("%d\n", a / b);
  return 0;
}
```

左操作数为 INT_MIN，右操作数为-1。分别在我的电脑（x86_64）和 qemu 中运行，在我的电脑中没有运行结果，抛出异常 Arithmetic exception，在 qemu 中运行结果为-2147483648。

- 在 MiniDecaf 中，我们对于短路求值未做要求，但在包括 C 语言的大多数流行的语言中，短路求值都是被支持的。为何这一特性广受欢迎？你认为短路求值这一特性会给程序员带来怎样的好处？

短路求值可以加快程序的运行速度，当短路求值的表达式较为复杂或者计算较为耗时的时候（例如第二个表达式的值为函数运行结果），短路求值在第一个表达式结果已经使整个表达式结果为真/假时可以自动略过第二个表达式，等效于自动剪枝。此外短路还可以防止程序错误，优化代码结构。我们经常使用可能无效的值，例如零或 null 值，在继续处理这些情况之前，我们首先必须检查该值是否有效，这种情况下，短路求值可以防止因为无效值而遇到错误，同时也省去了不必要的判断语句。

程序员主要应用第二条优势。例如我们可以直接写 if (a && a.b == c)以及 if (a != 0 && b / a == c)等语句。