刘雅迪

计26

学号：2021010521

step1: 仅一个return的main函数

思考题

1. 在我们的框架中，从 AST 向 TAC 的转换经过了 `namer.transform`，`typer.transform` 两个步骤，如果没有这两个步骤，以下代码能正常编译吗，为什么？

   ```
   int main(){
       return 10;
   }
   ```

   答：如果没有这两个步骤，则这段代码还是可以正常编译。因为这个程序比较简单，只是返回了一个整数，我们只需要提取 return 语句返回的常量，为之分配一个临时变量，再生成相应的 TAC 返回指令即可，不涉及局部变量和全局变量作用域的绑定等问题，不涉及符号表的构建与类型检查，`namer.transform`和`typer.transform`没有发挥其作用，所以仍然可以正常编译。

2. 我们的框架现在对于 `return` 语句没有返回值的情况是在哪一步处理的？报的是什么错？

   答：在语法分析这一步处理的，会报Syntax error的错误。

3. 为什么框架定义了 `frontend/ast/tree.py:Unary`、`utils/tac/tacop.py:TacUnaryOp`、`utils/riscv.py:RvUnaryOp` 三种不同的一元运算符类型？

   答：这三种类型分别对应AST、TAC和目标代码阶段，为了适应不同的抽象层次，所以定义了三种不同的一元运算符类型。

step2: 一元操作

思考题

1. 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。

   答：-(~2147483647)

实验内容

我仔细阅读了“通过例子学习”部分，然后根据其内容梳理了整个编译过程。

为了实现c代码到ast到tac再到汇编代码的转换：

我首先在tacop.py和riscv.py中添加了逻辑非和按位非的符号。

然后在tacgen.py的transform函数中添加了op的其他两种类型，将node.UnaryOp转换为TacUnaryOp；在riscvasmemitter.py的visitUnary函数中同理操作，将TacUnaryOp转换为RvUnaryOp。

在tacinstr.py中的Unary类的__str__函数添加了self.op等于逻辑非和按位非的情况。

这样就可以实现一元操作的编译过程。

此外，在完成step2时我遇到了一个坑，就是由于取负操作在汇编代码中的指令就是neg，所以可以直接将NEG转化为小写字母，而逻辑非和按位非这样做不行，不能直接将BITNOT和LOGICNOT作为RvUnaryOp的名称，所以我采用NOT作为BITNOT的名称、SEQZ作为LOGICNOT的名称，最终得以编译成功。

step3: 加减乘除模

思考题

1. 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）

   ```
   #include <stdio.h>

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

   答：取a = 0x80000000，b = -1：

   ```
   #include <stdio.h>

   int main() {
     int a = 0x80000000;
     int b = -1;
     printf("%d\n", a / b);
     return 0;
   }
   ```

   在我的电脑（x86-64）上运行：

   ```
   > gcc -O0 minidecaf-tests/testcases/step3/a_test.c -o a_test
   > ./a_test
   Floating point exception
   ```

   在RISCV-32的qemu模拟器中运行：

```
>riscv64-unknown-elf-gcc -O0 minidecaf-tests/testcases/step3/a_test.c -march=rv32im -
mabi=ilp32 -o test
>qemu-riscv32 test
-2147483648
```

实验内容

和step2一元操作一致，只需要将node.BinaryOp、tacop.TacBinaryOp和RvBinaryOp的指令一一对应即可。

step4: 比较和逻辑表达式

思考题

1. 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？

   答：因为这一特性可以减少不必要的计算过程，从而减少开销；此外，短路求值可以在一定程度上避免一些潜在的错误，程序停止在某个地方可以避免这段程序后续地方出错。

   这一特性能够让程序员更高效地编写代码，同时避免一些可能的错误，提高代码的准确率。

实验内容

我一开始还是按照step2和step3的思路来写，还搜索了半天类似"<="在riscv中的指令等等，结果发现step4中的操作符必须写成一元和二元操作符的组合。

`||`和`&&`按照实验手册中给的汇编代码写的，写的过程还理解了一番dst, lhs和rhs与汇编代码中的对应。

`<=`、`>=`、`==`、`!=`都是按照我对式子的理解写的，比如`<=`是用`>`来比较两数大小，然后再对结果求反得到的，`>=`同理。而`==`和`!=`是先将两数相减，再将结果与0比较得到。在这个过程中由于我一开始弄混了逻辑非和按位非，所以还花了点时间debug。

```
def visitBinary(self, instr: Binary) -> None:
            """
            For different tac operation, you should translate it to different RiscV code
            A tac operation may need more than one RiscV instruction
            """
            if instr.op == TacBinaryOp.LOR: #||
                self.seq.append(Riscv.Binary(RvBinaryOp.OR, instr.dst, instr.lhs,
instr.rhs))
                self.seq.append(Riscv.Unary(RvUnaryOp.SNEZ, instr.dst, instr.dst))
            elif instr.op == TacBinaryOp.AND: #&&
                self.seq.append(Riscv.Unary(RvUnaryOp.SNEZ, instr.dst, instr.lhs))
                self.seq.append(Riscv.Binary(RvBinaryOp.SUB, instr.dst, Riscv.ZERO,
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))
            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.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))
            else:
                op = {
                    TacBinaryOp.ADD: RvBinaryOp.ADD,
                    TacBinaryOp.SUB: RvBinaryOp.SUB,
                    TacBinaryOp.MUL: RvBinaryOp.MUL,
                    TacBinaryOp.DIV: RvBinaryOp.DIV,
                    TacBinaryOp.MOD: RvBinaryOp.REM,
                    TacBinaryOp.SLT: RvBinaryOp.SLT,
                    TacBinaryOp.SGT: RvBinaryOp.SGT,
                    # You can add binary operations here.
                }[instr.op]
                self.seq.append(Riscv.Binary(op, instr.dst, instr.lhs, instr.rhs))
```