Stage-1

沈佳茗 2021010745

一、实验内容

1. step 2

在 frontend/tacgen/tacgen.py 的 visitUnary 函数中将 AST 上的 BitNot 和 LogicNot 运算类型翻译为 TAC 中的 NOT 和 SEQZ 运算，即在 op 中 添加 node.UnaryOp 与 tacop.TacUnaryOp 对。

在 backend/riscv/riscvasmemitter.py 的 visitUnary 函数中，将 TAC 运算直接翻译为 RISC-V 指令，即在 op 中添加对应的 TacUnaryOp 与 RvUnaryOp 对。

```
# frontend/tacgen/tacgen.py
def visitUnary(self, expr: Unary, mv: TACFuncEmitter) -> None:
        expr.operand.accept(self, mv)

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

```
# backend/riscv/riscvasmemitter.py
def visitUnary(self, instr: Unary) -> None:
            op = {
                TacUnaryOp.NEG: RvUnaryOp.NEG,
                TacUnaryOp.NOT: RvUnaryOp.NOT,
                TacUnaryOp.SEQZ: RvUnaryOp.SEQZ,
                # You can add unary operations here.
            }[instr.op]
            self.seq.append(Riscv.Unary(op, instr.dst, instr.operand))
```

2. step 3

在 frontend/tacgen/tacgen.py 的 visitUnary 函数中将 AST 上的 Add、Sub、Mul、Div、Mod 运算类型分别翻译为 TAC 中的 ADD、SUB、MUL、DIV、MOD 运算，即在 op 中添加 node.BinaryOp 与 tacop.TacBinaryOp 对。

在 backend/riscv/riscvasmemitter.py 的 visitUnary 函数中，将 TAC 运算直接翻译为 RISC-V 指令，即在 op 中添加对应的 TacBinaryOp 与 RvBInaryOp 对。

```
# frontend/tacgen/tacgen.py
 def visitBinary(self, expr: Binary, mv: TACFuncEmitter) -> None:
        expr.lhs.accept(self, mv)
        expr.rhs.accept(self, mv)
```

```
        op = {
            node.BinaryOp.Add: tacop.TacBinaryOp.ADD,
            node.BinaryOp.Sub: tacop.TacBinaryOp.SUB,

            node.BinaryOp.Mul: tacop.TacBinaryOp.MUL,
            node.BinaryOp.Div: tacop.TacBinaryOp.DIV,
            node.BinaryOp.Mod: tacop.TacBinaryOp.MOD,
            # You can add binary operations here.
        }[expr.op]
        expr.setattr(
            "val", mv.visitBinary(op, expr.lhs.getattr("val"),
expr.rhs.getattr("val"))
        )
```

```
# backend/riscv/riscvasmemitter.py
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.OR:
                self.seq.append(Riscv.Binary(RvBinaryOp.OR, 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,

                    # You can add binary operations here.
                }[instr.op]
                self.seq.append(Riscv.Binary(op, instr.dst, instr.lhs, instr.rhs))
```

3. step 4

在 frontend/tacgen/tacgen.py 的 visitUnary 函数中将 AST 上的 LogicOr、LogicAnd、EQ、NE、LT、GT、LE、GE 运算类型分别翻译为 TAC 中的 OR、AND、EQU、NEQ、SLT、SGT、LEQ、GEQ 运算，即在 op 中添加 node.BinaryOp 与 tacop.TacBinaryOp 对。

在 backend/riscv/riscvasmemitter.py 的 visitUnary 函数中，将 TAC 运算直接翻译为 RISC-V 指令。具体实现为，若 TAC 指令为 SLT 或 SGT，则在 op 中直接添加对应的 TacBinaryOp 与 RvBInaryOp 对（只用一条 RISC-V指令即可实现功能）；否则，根据 TAC 指令依次添加多条 RISC-V 指令（LogicOr 和 LogicAnd 参考实验文档中的实现，其他指令部分参考 godbolt 网站编译结果）。

```python
# frontend/tacgen/tacgen.py
 def visitBinary(self, expr: Binary, mv: TACFuncEmitter) -> None:
        expr.lhs.accept(self, mv)
        expr.rhs.accept(self, mv)

        op = {
            node.BinaryOp.Add: tacop.TacBinaryOp.ADD,
            node.BinaryOp.Sub: tacop.TacBinaryOp.SUB,

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

            node.BinaryOp.LogicOr: tacop.TacBinaryOp.OR,
            node.BinaryOp.LogicAnd: tacop.TacBinaryOp.AND,

            node.BinaryOp.EQ: tacop.TacBinaryOp.EQU,
            node.BinaryOp.NE: tacop.TacBinaryOp.NEQ,

            node.BinaryOp.LT: tacop.TacBinaryOp.SLT,
            node.BinaryOp.GT: tacop.TacBinaryOp.SGT,
            node.BinaryOp.LE: tacop.TacBinaryOp.LEQ,
            node.BinaryOp.GE: tacop.TacBinaryOp.GEQ,
            # You can add binary operations here.
        }[expr.op]
        expr.setattr(
            "val", mv.visitBinary(op, expr.lhs.getattr("val"),
expr.rhs.getattr("val"))
        )
```

```python
# backend/riscv/riscvasmemitter.py
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.OR:
                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.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))
            else:
                op = {
                    TacBinaryOp.ADD: RvBinaryOp.ADD,
                    TacBinaryOp.SUB: RvBinaryOp.SUB,

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

                    TacBinaryOp.SGT: RvBinaryOp.SGT,
                    TacBinaryOp.SLT: RvBinaryOp.SLT,
                    # You can add binary operations here.
                }[instr.op]
                self.seq.append(Riscv.Binary(op, instr.dst, instr.lhs, instr.rhs))
```

二、思考题

1. step 1

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;
   }
   ```

   能正确编译，在初始的框架中 typer.tranform 直接返回 program，因此不会对编译产生影响。namer.transform 检查源程序中是否出现语义错误，如果没有语义错误，则返回传入的程序，这个过程不会改变抽象语法树。所以即使没有这两个过程，由于以上程序本身是可以被 parse 的，所以可以正常编译。

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

   是在词法分析 & 语法分析中处理的，报错为 Syntax error。

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？

这三种一元运算符类型分别代表在抽象语法树中的一元运算符（数学符号），在 TAC 中的一元运算符和 riscv 中的一元运算符（riscv 指令），由于在三种语言中一元运算符的表示不同，所以需要定义三种不同的一元运算符类型。

2. step 2

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。

   -(~2147483647) = -(-2147483648) = 2147483648（理论上结果，实际上越界）

3. step 3

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

   WSL x86-64

   ```
   $ gcc div.cpp -O0
   $ ./a.out
   Floating point exception
   ```

   qemu 模拟器

   ```
   $ riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 div.cpp -O0 -o div.o
   $ qemu-riscv32 div.o
   -2147483648
   ```

4. step 4

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？

   短路求值可以减少计算量，当前面的表达式已经可以确定逻辑运算结果时，之后的表达式都可以不用再次进行计算。并且当后计算的表达式依赖于先计算的表达式时，可以直接写成 `expression_1 $$ expression_2` 的形式，不用进行分支的嵌套。