stage-1

学号：2021012806

姓名：尤梓锐

step1

思考题

删去transform

能正常运行

该段代码的AST如下：

```
program [
  function [
    type(int)
    identifier(main)
    block [
      return [
        int(10)
      ]
    ]
  ]
]
```

根据namer的代码，上述AST处理过程中namer依次调用自己的visitProgram，visitFunction，visitBlock，visitReturn，visitIntLiteral，这几个函数只完成了对是否有main函数的检验以及对int是否超出范围的检验，就这个特例程序而言，这几个检验都会通过，因此namer的visit并没有影响，删去也可以正常编译。但对程序进行修改就不行了，可能会涉及到namer中其他的visit类型，只是这个程序涉及的几个visit函数对这个程序可以忽略。

根据typer的代码，会直接返回，并没有实际的代码执行，删去也可以正常编译。

main没有返回值

在词法分析&语法分析阶段处理，即生成AST的过程中进行处理

报错信息为“Syntax error: EOF”，在return的后面遇到了不应该出现的字符EOF

Unary

以下面的程序为例：

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

frontend/ast/tree.py:Unary是指AST中的一元运算符，即下面的unary(-)，由parser模块生成，在tacgen.transform() 生成TAC时被转化成TacUnaryOp

```
program [
  function [
    type(int)
    identifier(main)
    block [
      return [
        unary(-) [
          int(10)
        ]
      ]
    ]
  ]
]
```

utils/tac/tacop.py:TacUnaryOp是指TAC中的一元运算符，即下面的 _T1 = - _T0 ， 由 tacgen.transform() 生成，在 asm.transfrom() 生成汇编代码时被转化成RvUnaryOp

```
FUNCTION<main>:
    _T0 = 10
    _T1 = - _T0
    return _T1
```

utils/riscv.py:RvUnaryOp是指生成RISCV汇编代码中的一元运算符，会将具体的类型再结合寄存器等转化成RISCV的字符串输出，例如负号运算的输出形式为

```
def __str__(self) -> str:
    return "{} ".format(self.op) + Riscv.FMT2.format(
        str(self.dsts[0]), str(self.srcs[0])
    )
```

设计三种不同的Unary是因为三者对应一元运算符在不同阶段的存在形式，分别要进行不同的处理。

step2

实验内容

参考NEG的实现方法，补充了BITNOT和LOGICNOT的计算

主要包括在tacunaryop中增加了tac层对bitnot和logicnot的定义以及对应Unary(TACInstr)的打印方法：

```
@@ -92,7 +92,7 @@ class Unary(TACInstr):
     def __str__(self) -> str:
         return "%s = %s %s" % (
             self.dst,
-            ("-" if (self.op == TacUnaryOp.NEG) else "!"),
+            str(self.op),
             self.operand,
         )

diff --git a/utils/tac/tacop.py b/utils/tac/tacop.py
index 1fe136f..2239830 100644
--- a/utils/tac/tacop.py
+++ b/utils/tac/tacop.py
@@ -20,6 +20,11 @@ class InstrKind(Enum):
 @unique
 class TacUnaryOp(Enum):
     NEG = auto()
+    BITNOT = auto()
+    LOGICNOT = auto()
+
+    def __str__(self) -> str:
+        return ['-', '~', '!'][self.value-1]
```

在tacgen中增加了bitnot、logicnot从node.Unary 到 tacop.TacUnaryOp的映射，实现从unary从ast到tac的转化

```
@@ -222,6 +222,8 @@ class TACGen(Visitor[TACFuncEmitter, None]):

         op = {
             node.UnaryOp.Neg: tacop.TacUnaryOp.NEG,
+            node.UnaryOp.BitNot: tacop.TacUnaryOp.BITNOT,
+            node.UnaryOp.LogicNot: tacop.TacUnaryOp.LOGICNOT
             # You can add unary operations here.
         }[expr.op]
         expr.setattr("val", mv.visitUnary(op, expr.operand.getattr("val")))
```

同样的，在riscv层也加入了bitnot和logicnot，分别是not和seqz，规定了二者tac到riscv的转化对应：

```
@@ -79,6 +81,8 @@ class RiscvAsmEmitter(AsmEmitter):
         def visitUnary(self, instr: Unary) -> None:
             op = {
                 TacUnaryOp.NEG: RvUnaryOp.NEG,
+                TacUnaryOp.BITNOT: RvUnaryOp.NOT,
+                TacUnaryOp.LOGICNOT: RvUnaryOp.SEQZ
                 # You can add unary operations here.
             }[instr.op]
             self.seq.append(Riscv.Unary(op, instr.dst, instr.operand))
```

思考题

-~2147483647

2147483647编码为0111……111，~2147483647编码为1000……000，在数值上为-2147483648，再取负则会先转成反码1111……111，再转成补码末位加1发生溢出。

step3

实验内容

类似step2，定义了sub，mul，div，mod的TacBinaryOp类型，以及从ast到tac的映射：

```
@@ -234,6 +234,10 @@ class TACGen(Visitor[TACFuncEmitter, None]):

         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.LOR,
             # You can add binary operations here.
         }[expr.op]
```

在riscv中也定义了对应的运算指令sub，mul，div以及rem，完成从tac到riscv的映射：

```
            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))
```

思考题

左操作数是-2147483648，右操作数是-1，相除后会发生溢出

在自己的x86-64电脑上结果是程序无法运行，发生错误

```
D:\3.1\Principles and Practice of Compiler Construction>gcc example.c

D:\3.1\Principles and Practice of Compiler Construction>a.exe
```

在RISCV-32 的 qemu 模拟器中，运行结果是-2147483648

```
2021012806@compiler-lab:~/minidecaf-2021012806$ riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 example.c
2021012806@compiler-lab:~/minidecaf-2021012806$ qemu-riscv32 a.out
-2147483648
```

step 4

实验内容

类似地完成tac层的运算符定义与从ast到tac的映射以及tac指令的打印方法：

```
@@ -239,6 +239,13 @@ class TACGen(Visitor[TACFuncEmitter, None]):
             node.BinaryOp.Div: tacop.TacBinaryOp.DIV,
             node.BinaryOp.Mod: tacop.TacBinaryOp.MOD,
             node.BinaryOp.LogicOr: tacop.TacBinaryOp.LOR,
+            node.BinaryOp.LogicAnd: tacop.TacBinaryOp.LAND,
+            node.BinaryOp.EQ: tacop.TacBinaryOp.EQ,
+            node.BinaryOp.NE: tacop.TacBinaryOp.NE,
+            node.BinaryOp.LT: tacop.TacBinaryOp.LT,
+            node.BinaryOp.GT: tacop.TacBinaryOp.GT,
+            node.BinaryOp.LE: tacop.TacBinaryOp.LE,
+            node.BinaryOp.GE: tacop.TacBinaryOp.GE
             # You can add binary operations here.
         }[expr.op]
```

```
@@ -117,6 +117,13 @@ class Binary(TACInstr):
             TacBinaryOp.DIV: "/",
             TacBinaryOp.MOD: "%",
             TacBinaryOp.LOR: "||",
+            TacBinaryOp.LAND: "&&",
+            TacBinaryOp.EQ: "==",
+            TacBinaryOp.NE: "!=",
+            TacBinaryOp.LT: "<",
+            TacBinaryOp.GT: ">",
+            TacBinaryOp.LE: "<=",
+            TacBinaryOp.GE: ">=",
         }[self.op]
         return "%s = (%s %s %s)" % (self.dst, self.lhs, opStr, self.rhs)
```

完成tac指令到riscv指令的转化，这里指令的转化不再是一对一的，出现了一对多的形式，逐条append即可，具体翻译成什么样的riscv指令可以自行写c代码进行汇编得到：

```
+            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.LAND:
+                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.EQ:
+                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.NE:
+                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.LE:
+                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.GE:
+                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.LT: RvBinaryOp.SLT,
+                    TacBinaryOp.GT: RvBinaryOp.SGT
+                    # You can add binary operations here.
+                }[instr.op]
+                self.seq.append(Riscv.Binary(op, instr.dst, instr.lhs, instr.rhs))
```

思考题

1、短路求值可以提高代码的运行速度，当前部分逻辑结果可以决定表达式的值时就可以略去后面逻辑的计算，从而加快程序的运行。

2、短路求值可以避免一些错误的发生，也帮助简化代码。比如一个a&&b的逻辑判断，本身b表达式在a表达式为假的时候计算会发生错误，但短路求值保证了在计算b表达式时a表达式一定为真，从而避免了一些错误的可能，我们在编写代码时也可以借助这一特性省去一些错误检验，从而简化代码。