Stage1 Report

计11 周韧平 2021010699

实验内容

Step2

本节中我采用和取负相似的方法，只需要将对应的一元操作AST节点对应到TAC，再转换成对应的RISV命令就可以了

```
        op = {
            node.UnaryOp.Neg: tacop.TacUnaryOp.NEG,
            node.UnaryOp.BitNot: tacop.TacUnaryOp.BitNot,       You, 23小时前 • stage1 .
            node.UnaryOp.LogicNot: tacop.TacUnaryOp.LogicNot,
            # You can add unary operations here.
        }[expr.op]
        expr.setattr("val", mv.visitUnary(op, expr.operand.getattr("val")))
```

```
@unique
class TacUnaryOp(Enum):
    NEG = auto()
    BitNot = auto()       You, 23小时前 • stage1 …
    LogicNot = auto()

# Kinds of binary operations
```

```
@unique
class RvUnaryOp(Enum):
    NEG = auto()
    SNEZ = auto()
    NOT = auto()       You, 23小时前 • stage1 …
    SEQZ = auto()

@unique
```

```

        def visitUnary(self, instr: Unary) -> None:
            op = {
                TacUnaryOp.NEG: RvUnaryOp.NEG,
                TacUnaryOp.BitNot: RvUnaryOp.NOT,       You, 23小时前 • stage1 …
                TacUnaryOp.LogicNot: RvUnaryOp.SEQZ,
                # You can add unary operations here.
            }[instr.op]
            self.seq.append(Riscv.Unary(op, instr.dst, instr.operand))

```

step3

和一元运算相似，本节中只需要将二元运算的AST节点映射到TAC操作符，并对应到RISV字符就可以了，需要注意的是MOD对应的RISV符号为rem

```
    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.LOR,
```

```
            else:
                op = {
                    TacBinaryOp.ADD: RvBinaryOp.ADD,
                    TacBinaryOp.SUB: RvBinaryOp.SUB,
                    TacBinaryOp.MUL: RvBinaryOp.MUL,
                    TacBinaryOp.DIV: RvBinaryOp.DIV,
                    TacBinaryOp.MOD: RvBinaryOp.REM,
```

```
class TacBinaryOp(Enum):
    ADD = auto()
    LOR = auto()

    LAnd = auto()
    SUB = auto()
    MUL = auto()
    DIV = auto()
    MOD = auto()
```

```
@unique
class RvBinaryOp(Enum):
    ADD = auto()
    ADDI = auto()
    SUB = auto()
    MUL = auto()
    DIV = auto()
    REM = auto()
    OR = auto()
```

step4

本节中实现了逻辑运算符号，除了添加基本的从AST到TAC以及TAC到RISV的翻译外，由于一些操作没有对应的单条RISV指令，因此需要设计多条TAC来实现

具体来说

- 对于 `<=` 和 `>=` ，可以采用 `>` 和 `<` 和 逻辑非实现
- `!=` 和 `==` ，可以采用相减再判断是否为0实现
- `&&` 和 `||` 参考实验指导的实现

```
            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,
            node.BinaryOp.EQ: tacop.TacBinaryOp.EQ,
            # You can add binary operations here.
```

```
class TacBinaryOp(Enum):
    ADD = auto()
    LOR = auto()

    LAnd = auto()
    SUB = auto()
    MUL = auto()
    DIV = auto()
    MOD = auto()
    EQ = auto()
    NE = auto()
    LT = auto()
    GT = auto()
    LE = auto()
    GE = auto()
```

```
                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))
            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.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.rhs, instr.lhs))
                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,
                    # TacBinaryOp.NE: RvBinaryOp.NE,
```

```
@unique
class RvBinaryOp(Enum):
    ADD = auto()
    ADDI = auto()
    SUB = auto()
    MUL = auto()
    DIV = auto()
    REM = auto()
    OR = auto()
    # EQ = auto()
    NE = auto()
    SLT = auto()
    SGT = auto()
    AND= auto()

```

思考题

> **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;
}
```

可以正常编译，实测如果注释掉这两个步骤仍然可以正常编译生成汇编代码。

`Namer` 类将 AST 的符号进行重命名，并存储到符号表中。这段程序里面没有需要解析的符号，所以并不需要 `Namer`

`Typer` 类用于对 AST 进行类型检查，首先考虑到这段代码编译不需要对变量进行类型检查，其次在提供的代码框架中 Typer 的实现也已经省略，因此省略这一步并不会影响编译。

> **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？**

在 `./frontend/parser/ply_parser.py` 中，t如果为空则会跳入 `error_stack.append(DecafSyntaxError(t, "EOF"))` 分支，向 error_stack 中压入错误信息，比如我们可以向这一函数中加入调试信息，并编译一个没有返回值的 `main` 函数

```
def p_error(t): # t为空，跳入 error_stack.append(DecafSyntaxError(t, "EOF")) 分支
    """
    A naive (and possibly erroneous) implementation of error recovering.
    """
    print(f"in p_error, t is {t}")
    if not t:
        error_stack.append(DecafSyntaxError(t, "EOF"))
        return

    inp = t.lexer.lexdata
    error_stack.append(DecafSyntaxError(t, f"\n{inp.splitlines()[t.lineno - 1]}"))

    parser.errok()
    return parser.token()
```

```
Syntax error: EOF
2021010699@compiler-lab:~/minidecaf-2021010699$ python main.py --input ./example.c --tac
in p_error, t is LexToken(Semi,';',2,24)
in p_error, t is LexToken(RBrace,'}',3,26)
in p_error, t is None
Syntax error: line 2, column 12
    return ;
Syntax error: line 3, column 1
}
Syntax error: EOF
```

可以看到 `t is None` 跳入报错，程序生成 `DecafSyntaxError` 错误类型，提示 `t` 为 `EOF`

> **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?**

`frontend/ast/tree.py:Unary` 作用于抽象语法树，将 minidecaf 语言翻译成AST抽象语法树，在这一过程中执行语义检查，获取符号表、类型等信息，其中一元运算的节点用 `Unary` 表示

`utils/tac/tacop.py:TacUnaryOp` 是三地址码中的的一元运算符，用于优化和转换源代码。通过 Vistor 模式翻译 AST，将每个结点做翻译处理，得到的三地址码中的一元运算用 `TacUnaryOp` 表示

`utils/riscv.py:RvUnaryOp` 是 `riscv` 汇编代码中的一元运算符，通过对中间代码进行进一步翻译得到

这些不同的一元运算符类型的定义，使得在不同的上下文中可以方便地处理和表示一元运算符，并根据需要进行不同的操作和转换。

> **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
```

2147483647 比特表示为0x7FFFFFFF，取反后为0x80000000，表示-2147483648，取负后应该为2147483648，此时产生越界

> **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）**

```
#include <stdio.h>

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

参考上一道思考题的思路，可以通过将负数取负的方式使得整数计算超出范围

```
#include <stdio.h>

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

使用服务器环境 Ubuntu 23.04 x86-64

```
2021010699@compiler-lab:~/hw1$ gcc undef_div.c -O0 -o undef_div.out
2021010699@compiler-lab:~/hw1$ ./undef_div.out
Floating point exception (core dumped)
2021010699@compiler-lab:~/hw1$
```

抛出异常，提示计算异常

```
2021010699@compiler-lab:~/hw1$ riscv64-unknown-elf-gcc undef_div.c -O0 -o u
ndef_div2.out
2021010699@compiler-lab:~/hw1$ ./undef_div2.out
-2147483648
```

计算得到整数除法溢出结果（0x8FFFFFFF）

> **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？**

短路求值是一种用于逻辑表达式的求值策略，它在遇到能够确定整个表达式结果的情况下会立即停止求值。在大多数流行的编程语言中支持短路求值，这样做的主要原因是提高程序的效率和简化代码编写。

首先，**短路求值可以提高程序的效率**。当逻辑表达式包含多个条件时，使用短路求值可以减少不必要的计算。例如，在条件表达式中使用逻辑与（&&）操作符，如果第一个条件为假，后续条件将不会被计算，从而节约了时间和资源。这对于有大量复杂条件的情况下特别有用，可以提高程序的执行速度。

其次，**短路求值可以简化代码编写**。通过利用短路求值特性，我们能够使用更简洁的代码实现常见的逻辑控制。例如，使用逻辑与操作符（&&）可以方便地进行条件检查和合并，而无需编写额外的 if 语句。这样可以使代码更加清晰简洁，并且减少了出错的可能性。