Stage1 实验报告

姓名：丁俊辉

学号：2021011145

step1

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;
}
```

能正常编译。这两个步骤完成的任务是语义分析，`namer.transform` 进行符号表的构建，`typer.transform` 进行类型的检查。通过这两步分析，可以确保如果出现了语义错误，编译器能及时报错，但是其本身不会对抽象语法树进行修改。对于上述的代码，其本身没有语义的错误，并不会在这两个步骤中检查出错误，因此即便不进行这两步，也不会影响后序的步骤。

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？

在语法分析的时候处理。在 `frontend/praser/ply_praser.py` 中定义了 `p_error` 用来处理语法分析时遇到的错误。当它发现语法错误的时候，它会创建一个 `DecafSyntaxError` 对象，并将其添加到 `error_stack` 中。然后，它会尝试从错误中恢复，以继续解析剩余的输入。

在 `p_return` 中，`Return` 后面要跟一个 `expression`。因此，当它尝试解析没有返回值的 `return` 语句时，就会出现 `DecafSyntaxError`：

```
(minidecaf) 2021011145@compile:~/minidecaf-2021011145$ python main.py --input
example.c --parse
Syntax error: line 1, column 20
int main() { return; }
Syntax error: line 1, column 22
int main() { return; }
Syntax error: EOF
```

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？

这三种不同的一元运算符类型 `Unary`、`TacUnaryOp`、`RvUnaryOp` 是为了在编译器的不同阶段处理和表示一元运算符：

- `Unary`：`Unary` 是在抽象语法树（AST）阶段使用的类型。它表示源代码中的一元运算符，并用于语法分析和语义分析阶段。这个类型主要用于构建和表示源代码的结构。

- `TacUnaryOp`：`TacUnaryOp` 是在中间表示（IR）阶段使用的类型。它表示三地址码（TAC）中的一元运算符，用于中间代码生成和优化阶段。这个类型主要用于将源代码转换为中间表示，以便进行进一步的优化和转换。
- `RvUnaryOp`：`RvUnaryOp` 是在目标代码生成阶段使用的类型。它表示 RISC-V 汇编代码中的一元运算符，用于将中间表示转换为目标机器代码。这个类型主要用于生成最终的汇编代码，以便在目标机器上执行。

step2

实验过程

在 ply 中，这三个一元运算符的 token 和语法都已经定义好了，在 `frontend/ast/tree.py` 中也定义好了相应的节点，也就是说已经可以生成对应的 AST。但是在后续的中间代码和目标代码生成过程中，并没有引入对 `~` 和 `!` 这两种操作的支持。

因此，要在 `TacUnaryOp` 中加入两种新的操作：

```
@unique
class TacUnaryOp(Enum):
    NEG = auto()
    NOT = auto()
    LNOT = auto()
```

在 `RvUnaryOp` 也要加入：

```
@unique
class RvUnaryOp(Enum):
    NEG = auto()
    SNEZ = auto()
    NOT = auto()
    SEQZ = auto()
```

然后，在 `frontend/tacgen/tacgen.py` 中，加入 `UnaryOp` 到 `TacUnaryOp` 的转换：

```
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.LNOT,
    }[expr.op]
    expr.setattr("val", mv.visitUnary(op, expr.operand.getattr("val")))
```

在 `backend/riscv/riscvasmemitter.py` 中加入 `TacUnaryOp` 到 `RvUnaryOp` 的转换：

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

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。

设计为：

```
-~2147483647
```

`~2147483647` 结果为 `-2147483648`，`--2147483648` 计算得到 `-2147483648` 越界。

step3

实验过程

实验的过程和上一步类似，需要在 `TacBinaryOp` 和 `RvBinaryOp` 中加入 `-`、`*`、`/` 和 `%` 四种操作的支持。

```
@unique
class TacBinaryOp(Enum):
    ADD = auto()
    LOR = auto()
    SUB = auto()
    MUL = auto()
    DIV = auto()
    MOD = auto()

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

然后，要在 `frontend/tacgen/tacgen.py` 和 `backend/riscv/riscvasmemitter.py` 中加入相应的转换：

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


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))
    else:
        op = {
            TacBinaryOp.ADD: RvBinaryOp.ADD,
            # You can add binary operations here.
            TacBinaryOp.SUB: RvBinaryOp.SUB,
            TacBinaryOp.MUL: RvBinaryOp.MUL,
            TacBinaryOp.DIV: RvBinaryOp.DIV,
            TacBinaryOp.MOD: RvBinaryOp.REM,
        }[instr.op]
        self.seq.append(Riscv.Binary(op, instr.dst, instr.lhs, instr.rhs))
```

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）

```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;
}
```

左操作数取 `-2147483648`，右操作数取 `-1`。

我的电脑架构为 x86-64，运行结果如下：

```
djh592@DJH592:~/dev/complie/test$ gcc -O0 main.c -o main
djh592@DJH592:~/dev/complie/test$ ./main
Floating point exception (core dumped)
```

在 RISCV-32 的 qemu 模拟器中，运行结果如下：

```
2021011145@compile:~/test$ riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 -O0
main.c
2021011145@compile:~/test$ qemu-riscv32 a.out
-2147483648
```

step4

实验过程

首先在 `TacBinaryOp` 和 `RvBinaryOp` 中加入 `<`、`<=`、`>`、`>=`、`$$` 和 `||` 四种操作的支持。

在 `TacBinaryOp` 中，我添加了相应的几个二元运算：

```python
@unique
class TacBinaryOp(Enum):
    ...
    LAND = auto()
    EQU = auto()
    NEQ = auto()
    SLT = auto()
    LEQ = auto()
    SGT = auto()
    GEQ = auto()
```

在 `RvBinaryOp` 中，我添加了 `AND`、`XOR`、`SLT` 这三个指令：

```python
@unique
class RvBinaryOp(Enum):
    ...
    AND = auto()
    XOR = auto()
    SLT = auto()
```

在 `frontend/tacgen/tacgen.py` 中，我加入了 TAC 码的生成：

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

    op = {
        ...
        node.BinaryOp.LogicAnd: tacop.TacBinaryOp.LAND,
        node.BinaryOp.EQ: tacop.TacBinaryOp.EQU,
        node.BinaryOp.NE: tacop.TacBinaryOp.NEQ,
        node.BinaryOp.LT: tacop.TacBinaryOp.SLT,
        node.BinaryOp.LE: tacop.TacBinaryOp.LEQ,
        node.BinaryOp.GT: tacop.TacBinaryOp.SGT,
        node.BinaryOp.GE: tacop.TacBinaryOp.GEQ,
    }[expr.op]
    expr.setattr(
        "val", mv.visitBinary(op, expr.lhs.getattr("val"),
expr.rhs.getattr("val"))
    )
```

在 `backend/riscv/riscvasmemitter.py` 中，我完成了 TAC 码到 Riscv 汇编的映射：

```
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.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.EQU:
        self.seq.append(Riscv.Binary(RvBinaryOp.XOR, 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.XOR, instr.dst, instr.lhs,
instr.rhs))
        self.seq.append(Riscv.Unary(RvUnaryOp.SNEZ, instr.dst, instr.dst))
    elif instr.op == TacBinaryOp.SLT:
        self.seq.append(Riscv.Binary(RvBinaryOp.SLT, instr.dst, instr.lhs,
instr.rhs))
    elif instr.op == TacBinaryOp.SGT:
        self.seq.append(Riscv.Binary(RvBinaryOp.SLT, instr.dst, instr.rhs,
instr.lhs))
    elif instr.op == TacBinaryOp.LEQ:
```

```
        self.seq.append(Riscv.Binary(RvBinaryOp.SLT, instr.dst, instr.rhs,
instr.lhs))
        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:
        ...
```

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？

短路求值可以让编程更加灵活，很多时候利用这个功能可以实现类似分支的功能。可以通过短路执行额外的表达式，也可以通过短路避免执行不应该执行的表达式。

我认为的好处有：

- 效率提升：短路求值可以避免执行不必要的表达式。
- 安全性：在一些情况下，第二个表达式可能包含有副作用的代码或者可能导致错误的操作（如访问空指针）。短路求值可以避免执行这些潜在危险的操作。
- 代码清晰性：短路求值允许将多个条件检查组合成一个表达式，而不必使用多个`if`语句，这可以使代码更简洁。
- 条件控制：程序员可以利用短路求值来控制程序流程，例如，在逻辑OR表达式中使用一个修复错误的代码块作为第二个操作数。