Stage 1 报告

程思翔 2021010761

1 实验内容

在 `utils/tac/tacop.py`, `utils/riscv.py` 的 `TacXXXOp` 和 `RvXXXOp` 类中添加运算符, 在 `frontend/tacgen/tacgen.py`, `backend/riscv/riscvasmemitter.py` 中的 `visitUnary` 和 `visitBinary` 方法实现运算符翻译与计算过程.

1.1 step 2

```python
# utils/tac/tacop.py
@unique
class TacUnaryOp(Enum):
    ...
    BIT_NOT = auto() # 取反
    LOGIC_NOT = auto() # 取非
```

```python
# utils/riscv.py
@unique
class RvUnaryOp(Enum):
    ...
    NOT = auto() # 取反, 为伪指令
    SEQZ = auto() # 取非, 为伪指令
```

```python
# frontend/tacgen/tacgen.py
def visitUnary(self, expr: Unary, mv: TACFuncEmitter) -> None:
    expr.operand.accept(self, mv)
    op = {
        ...
        node.UnaryOp.BitNot: tacop.TacUnaryOp.BIT_NOT,
        node.UnaryOp.LogicNot: tacop.TacUnaryOp.LOGIC_NOT,
    }[expr.op]
    ...
```

```python
# backend/riscv/riscvasmemitter.py
def visitUnary(self, instr: Unary) -> None:
    op = {
        ...
        TacUnaryOp.BIT_NOT: RvUnaryOp.NOT,
        TacUnaryOp.LOGIC_NOT: RvUnaryOp.SEQZ,
    }[instr.op]
    ...
```

1.2 step 3

```python
# utils/tac/tacop.py
@unique
class TacBinaryOp(Enum):
    ...
    SUB = auto()
    MUL = auto()
    DIV = auto()
    MOD = auto()
```

```python
# utils/riscv.py
@unique
class RvBinaryOp(Enum):
    ...
    SUB = auto()
    MUL = auto()
    DIV = auto()
    REM = auto() # 取模
```

```python
# frontend/tacgen/tacgen.py
def visitBinary(self, expr: Binary, mv: TACFuncEmitter) -> None:
    ...
    op = {
        ...
        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]
    ...
```

```python
# backend/riscv/riscvasmemitter.py
def visitBinary(self, instr: Binary) -> None:
    ...
    op = {
        ...
        TacBinaryOp.SUB: RvBinaryOp.SUB,
        TacBinaryOp.MUL: RvBinaryOp.MUL,
        TacBinaryOp.DIV: RvBinaryOp.DIV,
        TacBinaryOp.MOD: RvBinaryOp.REM,
    }[instr.op]
```

1.3 step 4

```python
# utils/tac/tacop.py
@unique
class TacBinaryOp(Enum):
    ...
    LAND = auto()
    EQU = auto()
    NEQ = auto()
    SLT = auto()
    LEQ = auto()
    SGT = auto()
    GEQ = auto()
```

```python
# utils/riscv.py
@unique
class RvUnaryOp(Enum):
    ...
    SLTZ = auto() # 小于 0 则置位, 为伪指令
    SGTZ = auto() # 大于 0 则置位, 为伪指令

@unique
class RvBinaryOp(Enum):
    ...
    AND = auto() # 按位与
    SLT = auto() # 小于
    SGT = auto() # 大于
```

```python
# frontend/tacgen/tacgen.py
def visitBinary(self, expr: Binary, mv: TACFuncEmitter) -> None:
    ...
    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.GT: tacop.TacBinaryOp.SGT,
        node.BinaryOp.LE: tacop.TacBinaryOp.LEQ,
        node.BinaryOp.GE: tacop.TacBinaryOp.GEQ,
    }[expr.op]
    ...
```

```
# backend/riscv/riscvasmemitter.py
def visitBinary(self, instr: Binary) -> None:
    # 特殊 Tac 操作符与对应操作
    # 利用了 [https://godbolt.org/] 给出的结果
    if instr.op == TacBinaryOp.LOR:
        ...
    elif instr.op == TacBinaryOp.LAND:
        ...
    elif instr.op == TacBinaryOp.EQU:
        ...
    elif instr.op == TacBinaryOp.NEQ:
        ...
    elif instr.op == TacBinaryOp.LEQ:
        ...
    elif instr.op == TacBinaryOp.GEQ:
        ...
    else:
        op = {
            ...
            # 只有这两条 Tac 指令无需使用其他 riscv 指令翻译
            TacBinaryOp.SLT: RvBinaryOp.SLT,
            TacBinaryOp.SGT: RvBinaryOp.SGT,
        }[instr.op]
        ...
```

2 思考题

2.1 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;
}
```

答: 能正常编译. 这两个步骤用于**语义分析**阶段实现**符号表构建**和**类型检查**, 主要作用是解析标识符的声明和引用, 将其存储在符号表中, 验证语句和表达式操作是否符合类型规则. 这段代码并没有涉及函数或变量等标识符的使用, 因此没有这两个步骤这段代码依旧可以**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?

答: 在 `frontend/parser/ply_parser.py` 进行语法分析时处理, 报错为

```
Syntax error: line 2, column 11
    return;
Syntax error: line 3, column 1
}
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?

答: 在编译器框架中, 三种不同的一元运算符类型用于不同的编译器阶段和组件, 以适应不同层次的运算符表示和需求:

- `Unary` 用于 AST 表示, 进行语法分析和语义分析.
- `TacUnaryOp` 用于 TAC 表示, 进行中间代码生成.
- `RvUnaryOp` 用于 RISC-V 表示, 进行目标代码生成.

不同阶段所需的一元运算符类型不完全相同 (如取模运算的 `BinaryOp.Mod`, `TacBinaryOp.MOD`, `RvUnaryOp.REM`), 需要经过相应的转化翻译过程, 这么分离定义可以实现有助于模块化和灵活性, 使不同阶段独立处理运算符.

2.2 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.

   答: 设计如下:

   ```
   -~2147483647
   ```

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

   答: 左操作数为 `-2147483648`, 右操作数为 `-1`.

```c
#include <stdio.h>
int main() {
    int a = -2147483648;
    int b = -1;
    printf("%d\n", a / b);
    return 0;
}
```

电脑架构为 Arm64, 使用 clang 编译运行代码, 结果为 `-2147483648`;

使用 RISCV-32 的 qemu 模拟器编译运行代码, 结果为 `-2147483648`.

2.4 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?

   答: 短路求值是一种逻辑表达式计算策略, 当第一个运算数无法确定逻辑运算的结果时, 才对第二个运算数进行求值. 这一特性广受欢迎, 有以下好处:

   - **效率**: 使用逻辑运算符连接多个布尔表达式时, 如果第一个表达式确定了结果, 那么后面的表达式不会被计算. 涉及到昂贵的计算或函数调用时, 使用短路求值可以免去表达式执行成本, 提高运行效率.
   - **安全性**: 表达式的计算可能具有副作用, 如修改变量的值或执行其他操作, 短路求值确保这些副作用只在需要时才会发生. 如右表达式需要依赖左表达式的成立, 支持短路求值后可以在左表达式不成立后避免错误计算右表达式.
   - **自然**: 短路求值可以避免深度嵌套的条件语句, 编写更简洁易读的代码, 使得程序员能够更自由地表达各种逻辑关系和条件.