Stage-1 实验报告

计 03 王文琦 2020010915

实验内容

Step 1

`Step 1` 的主要任务是熟悉整个框架并且熟悉 `visitor` 模式。

- 首先程序会进入前端部分完成词法分析和语法分析。位于 `frontend/lexer/ply_lexer.py` 将程序转化为 `Token` 流；然后再由位于 `frontend/parser/ply_parser.py` 定义的语法分析器转化为 `AST` 。
  - 在 `frontend/ast/tree.py` 加入新的 AST 节点定义（以及相应的其它东西），并且在 `frontend/ast/visitor.py 加入相应的分派函数。`
  - 在 `frontend/parser/ply_parser.py` 里加入新的 grammer rule 。
- 紧接着，我们会使用 `Visitor` 模式来遍历整个 AST ，本质上就是一个 `DFS` 遍历的过程。类似于 `Program Function` 等节点继承自抽象节点 `Node` ，并且每个节点都包含了一个 accept() 函数，来调用遍历自己的节点的 `visitor` 的相应函数。我们可以在 `Namer Typer` 类中编写每一种语法节点的遍历方法。
  - `Namer` 类代表构建符号表的遍历， `Typer` 类型代表类型检查的遍历。
- 在遍历进行语义检查之后，我们已经构建好了符号表，接下来需要再次遍历一遍语法树，对于每一个节点进行一次翻译处理。
  - `frontend/tacgen/tacgen.py` 通过一遍扫描 `AST` 生成三地址码。
  - `frontend/utils/tac.py` 实现了生成三地址码的基层类。
  - `tacinstr.py` 实现了各种 `TAC` 指令。 `funcvisitor.py` 中可以添加相应的接口。
- 在生成中间代码之后，需要生成目标代码。
  - `backend/riscvasmemitter.py` 对每个函数内的 TAC 指令选择相应的 RISC-V 指令，然后会进行数据流分析、寄存器分配等流程，最后通过 `emitEnd` 方法生成每个函数的 `RISCV` 汇编。
  - `utils/riscv.py` 包含了一些关于 `RISCV` 汇编的规定，例如各个寄存器的称号等等。

Step 2

`Step 2` 的主要任务是给整数增加一元运算符 `- ~ !` 。

- 前端词法分析和语法分析已经完成好了，我们需要完成中间代码和目标代码的生成

```
    def visitUnary(self, expr: Unary, mv: FuncVisitor) -> None:
        expr.operand.accept(self, mv)

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

- 因为 `Riscv` 本身就包含有上述三条指令，所以在翻译目标代码阶段无需做多余操作，默认转化成小写就可以了。

Step 3

`Step 3` 的主要任务是补充二元运算符 `+ - * / % ()`。

- 前端分析已经完善，我们需要在 `tacop tacinstr` 里面添加相关的指令符号和对应的字符串，然后在 `tacgen` 里面将节点对应为相应的二进制操作符。
- 同理，`Riscv` 中就包含了相应的单条指令，所以无需做多余的操作。

Step 4

`Step 4` 的主要任务就是补充比较和逻辑运算 `< <= > >= == != && ||` 。

- 需要在二进制操作符中额外添加逻辑与和逻辑或，其余操作和 `Step 3` 相同。
- 在由中间代码翻译成目标代码的阶段，因为这些运算符都是没有对应的单条 `Riscv` 指令，所以需要用多条指令组合生成。

```
//equ
sub rd, r1, r2
seqz rd, rd

//neq
sub rd, r1, r2
snez rd, rd

//leq
sgt rd, r1, r2 seqz rd, rd
```

思考题

1. **Step 2:** 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。

   > 表达式 -0 会溢出。对 0 补码取反等于对各位取反加 1，会导致最高位进位溢出。

2. **Step 3:** 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化）

```c
#include <stdio.h>

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

> 这两个数的除法会导致结果溢出，是未定义的行为。在 x86 下运行不开优化会卡死，开优化会得到溢出的结果 -2147483648。在 qemu 模拟器中会得到结果 -2147483648。

3. **Step 4:** 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？

> 短路求值可以剪枝掉无需进行判断的部分，加快程序的性能；同时，短路求值的特性可以帮助程序员少写一些分支判断，尤其是当分支的后面的条件依赖于前面的条件时，可以不用拆分成多级分支判断，只需要利用短路求值的特性，就可以在一个分支里完成相应的判断。