Stage 2 报告

程思翔 2021010761

1 实验内容

1.1 step 5

语义分析: 实现了声明语句、赋值语句、标识符的语义分析函数.

```
# frontend/typecheck/namer.py:class Namer
def visitDeclaration(self, decl: Declaration, ctx: Scope) ->
None:
    if ctx.lookup(decl.ident.value):
        raise DecafDeclConflictError(decl.ident.value)
    # 构造 VarSymbol 对象, 将其加入符号表, 并设置 decl 的 symbol 属性
    symbol = VarSymbol(decl.ident.value, decl.var_t.type)
    ctx.declare(symbol)
    decl.setattr("symbol", symbol)
    if decl.init_expr:
        decl.init_expr.accept(self, ctx)

def visitAssignment(self, expr: Assignment, ctx: Scope) -> None:
    # 参考 `visitBinary` 的实现
    expr.lhs.accept(self, ctx)
    expr.rhs.accept(self, ctx)

def visitIdentifier(self, ident: Identifier, ctx: Scope) ->
None:
    symbol = ctx.lookup(ident.value)
    if not symbol:
        raise DecafUndefinedVarError(ident.value)
    # 设置 ident 的 symbol 属性
    ident.setattr("symbol", symbol)
```

中间代码生成: 为标识符、声明语句、赋值语句实现了中间代码生成函数. 递归访问每个子节点, 为节点设置返回值 `val`. 需要注意的是赋值时, 左端项需要是左值.

```
# frontend/tacgen/tacgen.py:class TACGen
def visitIdentifier(self, ident: Identifier, mv: TACFuncEmitter)
-> None:
    # 设置返回值为标识符对应的 temp 寄存器
    ident.setattr("val", ident.getattr("symbol").temp)

def visitDeclaration(self, decl: Declaration, mv:
TACFuncEmitter) -> None:
    decl.getattr("symbol").temp = mv.freshTemp()
    if decl.init_expr:
        # 对子节点进行 accept
        decl.init_expr.accept(self, mv)
        # 模仿 `visitAssignment` 函数进行赋值
        decl.setattr(
            "val",
mv.visitAssignment(decl.getattr("symbol").temp,
decl.init_expr.getattr("val"))
        )

def visitAssignment(self, expr: Assignment, mv: TACFuncEmitter)
-> None:
    # 对子节点进行 accept
    expr.lhs.accept(self, mv)
    expr.rhs.accept(self, mv)
    # 设置返回值为赋值指令的返回值, 赋值操作更新左值, 左端项是左值 temp
    expr.setattr(
        "val",
mv.visitAssignment(expr.lhs.getattr("symbol").temp,
expr.rhs.getattr("val"))
    )
```

目标代码生成: 为赋值语句实现了 `visitAssign` 函数进行目标代码生成.

```python
# backend/riscv/riscvasmemitter.py:class RiscvAsmEmitter
def visitAssign(self, instr: Assign) -> None:
    self.seq.append(Riscv.Move(instr.dst, instr.src))
```

2 思考题

2.1 step 5

1. 我们假定当前栈帧的栈顶地址存储在 sp 寄存器中, 请写出一段 **risc-v 汇编代码**, 将栈帧空间扩大 16 字节 (提示1: 栈帧由高地址向低地址延伸; 提示2: risc-v 汇编中 `addi reg0, reg1, <立即数>` 表示将 `reg1` 的值加上立即数存储到 `reg0` 中).

   答: 汇编代码为:

   ```
   addi sp, sp, -16
   ```

2. 有些语言允许在同一个作用域中多次定义同名的变量, 例如这是一段合法的 Rust 代码 (你不需要精确了解它的含义, 大致理解即可):

   ```rust
   fn main() {
       let a = 0;
       let a = f(a);
       let a = g(a);
   }
   ```

   其中 `f(a)` 中的 `a` 是上一行的 `let a = 0;` 定义的, `g(a)` 中的 `a` 是上一行的 `let a = f(a);`.

   如果 MiniDecaf 也允许多次定义同名变量, 并规定新的定义会覆盖之前的同名定义, 请问在你的实现中, 需要对定义变量和查找变量的逻辑做怎样的修改 (提示: 如何区分一个作用域中**不同位置**的变量定义?).

   答: 在语义分析部分 `frontend/typecheck/name.py` 中:

   - `visitDeclaration` 定义变量时, 不查询是否有同名变量, 即不抛出同名异常. 先访问初始化语句, 再访问变量声明, 并覆盖原始变量. 如果存在重名变量定义, 这可以正确获得变量的新初始值.

- `visitIdentifier` 查找变量无需修改, 变量被新定义的变量覆盖后, 只需寻找当前作用域中的符号, 即是最新定义的变量.