MiniDecaf-Stage2实验报告

计06 赵翊哲 2020010998

一、实验目的

在Stage1的基础上引入对变量和分支语句的支持，距离真正的编程更进一步。

变量的部分主要包括变量的声明、使用和赋值。

分支语句的部分主要包括if语句和条件表达式

二、实验内容

2.1 Step 5

- 修改内容：增加了对变量的支持
  - 声明变量（可选：赋值）
  - 对变量的赋值
  - 使用变量
- 实现：
  - 前端：主要需要针对变量进行语义分析的检查，修改内容主要在 `namer.py` ：
    - 声明变量：检查符号表中是否声明过，若有则报错重命名，否则添加到符号表中
    - 使用变量：检查符号表中是否声明过，若无则报错重命名，否则添加到符号表中
    - 核心代码：

```
def visitIdentifier(self, ident: Identifier, ctx: ScopeStack) -> None:
  # ScopeStack.lookup : 检查是否已定义
  symbol = ctx.lookup(ident.value)
  if symbol == None:  # 无定义报错
    raise DecafUndefinedVarError(ident.value)
  else:   # 否则设置 symbol 属性
    ident.setattr("symbol", symbol)
def visitAssignment(self, expr: Assignment, ctx: ScopeStack) -> None:
  # ScopeStack.lookup 检查左值是否定义
  symbol = ctx.lookup(expr.lhs.value)
  if symbol == None:  # 无定义报错
    raise DecafUndefinedVarError(expr.lhs.value)
  else:
    expr.lhs.setattr("symbol", symbol)
    # 类似 visitBinary
    expr.rhs.accept(self, ctx)
def visitDeclaration(self, decl: Declaration, ctx: ScopeStack) -> None:
  # ScopeStack.findConflict : 报错 or 定义 VarSymbol 对象
  symbol = ctx.findConflict(decl.ident.value)
  if symbol != None:
    raise DecafGlobalVarDefinedTwiceError(decl.ident.value)
    # ScopeStack.declare : 加入符号表
```

```
    else:
      symbol = VarSymbol(decl.ident.value, decl.var_t.type)
      ctx.declare(symbol)
      # Declaration.setattr : VarSymbol 存入 AST
      decl.setattr("symbol", symbol)
      # 初值表达式
      if decl.init_expr != NULL:
        decl.init_expr.accept(self, ctx
```

- 中端：主要增加处理变量相关操作的 TAC 代码生成的部分，值得注意的是在这一部分中需要用一个临时变量来存储变量对应的符号，因此在相关操作中要注意到使用的是 `val` 还是 `temp`，相关代码主要在 `tacgen.py` 中：

```
def visitIdentifier(self, ident: Identifier, mv: FuncVisitor) -> None:
  ident.setattr("val", ident.getattr("symbol").temp)
def visitDeclaration(self, decl: Declaration, mv: FuncVisitor) -> None:
  # 使用 getattr 方法获得在符号表构建阶段建立的符号
  symbol = decl.getattr("symbol")
  # FuncVisitor.freshTemp 函数获取一个新的临时变量 temp 存储该变量
  symbol.temp = mv.freshTemp()
  # 如果有设置初值， visitAssignment 赋值
  if decl.init_expr != NULL:
    decl.init_expr.accept(self, mv)
    mv.visitAssignment(getattr(symbol, "temp"), decl.init_expr.getattr("val"))
def visitAssignment(self, expr: Assignment, mv: FuncVisitor) -> None:
  expr.rhs.accept(self, mv)
  # 使用 temp 添加 TAC 指令，而非 var
  temp = expr.lhs.getattr("symbol").temp
  mv.visitAssignment(temp, expr.rhs.getattr("val"))
  # 设置表达式 val
  expr.setattr("val", mv.visitAssignment(temp, expr.rhs.getattr("val")))
```

  - 在实现过程中遇到的问题是一开始使用 `Symbol` 类来在 `namer.py` 中作为符号，但事实上应该使用 `VarSymbol`，后者包含需要存储临时变量 `Temp` 的成员

- 后端：用 `mv` 指令实现对 TAC 指令的翻译即可

```
def visitAssign(self, instr: Assign) -> None:
  self.seq.append(Riscv.Move(instr.dst, instr.src))
```

2.2 Step 6 if语句和条件表达式

- 修改内容：
  - 增加了if语句（框架代码已实现）
  - 增加了条件表达式
- 实现：
  - 主要修改了两个部分，分别是
    - `namer.py` 中的 `visitCondExpr` 函数，逻辑和框架中的 `visitIf` 函数逻辑一直

    - `tacgen.py` 中的 `visitCondExpr` 函数，逻辑和 `visitIf` 函数基本一致，只是相比于 if 的处理方式而言，条件表达式需要额外考虑到整个表达式的赋值处理，实现主要是将 `then` 和 `otherwise` 的表达式对应的值根据 `condition` 赋给整个条件表达式
  - （`tacgen.py`）核心修改代码如下：

```
def visitCondExpr(self, expr: ConditionExpression, mv: FuncVisitor) -> None:
  """
  1. Refer to the implementation of visitIf and visitBinary.
  """
  # Step6-2 仿照 visitIf 实现
  expr.cond.accept(self, mv)
  skipLabel = mv.freshLabel()
  exitLabel = mv.freshLabel()
  # 临时变量存储表达式的值
  temp = mv.freshTemp()
  mv.visitCondBranch(
    tacop.CondBranchOp.BEQ, expr.cond.getattr("val"), skipLabel
  )
  expr.then.accept(self, mv)
  # 临时变量存储 then 表达式的值
  mv.visitAssignment(temp, expr.then.getattr("val"))
  mv.visitBranch(exitLabel)
  mv.visitLabel(skipLabel)
  expr.otherwise.accept(self, mv)
  # 临时变量存储 otherwise 表达式的值
  mv.visitAssignment(temp, expr.otherwise.getattr("val"))
  mv.visitLabel(exitLabel)
  # 完成条件表达式后应进行赋值
  expr.setattr("val", temp)
```

三、思考题

3.1 Step5 思考题

Q1：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。）

> 目前的栈帧空间范围：栈底（sp+size of stack）~sp
>
> 若希望扩大空间，将栈顶所指向的地址向低地址移动即可

A：见下方代码

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

Q2：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）：

```
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？）

> 主要需要修改的部分在`namer.py`中，不需要在定义的时候判断是否重定义，查找变量时也需要找到最近一次的即可，而在题目所说的这种情况下也只有最近的变量能被查找到，最初的同名变量已经无法再被使用了，因此可以直接用后来的同名变量覆盖前者。

A：定义变量：原先在`namer.py`中的`visitDeclaration`函数中设计了重定义报错，现取消报错，只搜索是否定义过，是则覆盖，否则声明并加入符号表。

```
def visitDeclaration(self, decl: Declaration, ctx: ScopeStack) -> None:
  # Step5-3 处理声明
  # ScopeStack.findConflict : 找到是否存在同名变量
  symbol = ctx.findConflict(decl.ident.value)
  if symbol != None:
    # 修改部分
    decl.setattr("symbol", symbol)
  # ScopeStack.declare : 加入符号表
  else:
    # 和原先一致
    symbol = VarSymbol(decl.ident.value, decl.var_t.type)
    ctx.declare(symbol)
    # Declaration.setattr : VarSymbol 存入 AST
    decl.setattr("symbol", symbol)
    # 初值表达式
    if decl.init_expr != NULL:
      decl.init_expr.accept(self, ctx)
```

查找变量：在这种定义变量的处理方式之下，每次都只能找到最新定义的同名变量，因此不需要更改查找方式。

3.2 Step6 思考题

Q1：你使用语言的框架里是如何处理悬吊 else 问题的？请简要描述。

> 使用的 python 语言框架
>
> 参考了 yacc 文档：https://ply.readthedocs.io/en/latest/ply.html#dealing-with-ambiguous-grammars
>
> *By default, all shift/reduce conflicts are resolved in favor of shifting. Therefore, in the above example, the parser will always shift the `+` instead of reducing. Although this strategy works in many cases (for example, the case of "if-then" versus "if-then-else"), it is not enough for arithmetic expressions.*

A：优先结合**最近的**空 `if` 。在文档的说明中可以发现 `yacc` 在面对“移位-归约冲突”时会优先采取移位，因此当面对形如 `if E then if E then S` 的内容出现在栈中时，会优先将 `else` 移进，这样后续就会使用 `if E then S else S` 的规则归约，也就是将 `else` 和最近的 `if` 结合，从而解决了悬吊else问题。

Q2：在实验要求的语义规范中，条件表达式存在短路现象。即：

```
int main() {
    int a = 0;
    int b = 1 ? 1 : (a = 2);
    return a;
}
```

会返回 0 而不是 2。如果要求条件表达式不短路，在你的实现中该做何种修改？简述你的思路。

> 需要修改 `tacgen.py` 中的内容，主要是要求条件为真的时候，也执行 otherwise 表达式，只是最终赋值的时候使用 `then` 表达式的值。

A：在执行完 `then` 表达式后不跳转到 `exitLabel` ，而是继续执行 `otherwise` 表达式，分别用两个临时变量存储两个表达式的值，然后再进行条件判断，选择对应的临时变量赋给整个条件表达式。可能的代码实现如下

```
def visitCondExpr(self, expr: ConditionExpression, mv: FuncVisitor) -> None:
  # 不短路修改
  expr.cond.accept(self, mv)
  skipLabel = mv.freshLabel()
  exitLabel = mv.freshLabel()
  # 临时变量存储表达式的值
  then = mv.freshTemp()
  otherwise = mv.freshTemp()
  expr.then.accept(self, mv)
  mv.visitAssignment(then, expr.then.getattr("val"))
  expr.otherwise.accept(self, mv)
  mv.visitAssignment(otherwise, expr.otherwise.getattr("val"))
  mv.visitCondBranch(
    tacop.CondBranchOp.BEQ, expr.cond.getattr("val"), skipLabel
  )
  expr.setattr("val", then)
  mv.visitBranch(exitLabel)
  mv.visitLabel(skipLabel)
  expr.setattr("val", otherwise)
  mv.visitLabel(exitLabel)
```

四、参考资料

实现过程中主要参考了实验文档和实验思路QA墙

思考题参考了yacc文档