Stage-2 实验报告

沙之洲 2020012408

Step 5

实验内容

这一阶段我们主要处理变量的声明和赋值的问题，具体来说，我们需要修改 负责变量分配的 namer.py 文件 和 负责 tac 生成的 tacgen.py 文件

对于 namer.py 中的修改，我们以 visitDeclaration 为例进行介绍

```
name = decl.ident.value
type = decl.var_t.type
init_expr = decl.init_expr

conflict_result = ctx.findConflict(name)
if conflict_result:
    # conflict
    raise DecafDeclConflictError(name)
else:
    # no conflict
    new_decalre = VarSymbol(name, type)
    ctx.declare(new_decalre)
    # decl.ident.setattr('symbol', new_decalre)
    decl.setattr('symbol', new_decalre)

    # init value
    if not init_expr is NULL:
        # have init value, visit it
        init_expr.accept(self, ctx)
```

首先我们要检查当前声明的变量是否已经在 当前 context 中声明过了。如果声明过了，便是重复声明，这时候要报错，也就是 raise DecafDeclConflictError

如果没有冲突的话，我们需要在给它分配一个新的 varsymbol 并在当前 context 中声明这个变量，并且将新生成的 varsymbol 作为当前 AST 节点的 symbol 的 value 挂在 AST 树上，方便后续 tacgen 的过程

对于 tacgen.py 中的修改，我们也以 visitDeclaration 为例进行介绍

```
new_var = decl.getattr('symbol')
new_var.temp = mv.freshTemp()
init_expr = decl.init_expr
if not init_expr is NULL:
    init_expr.accept(self, mv) # 翻译成三地址码指令
    value = init_expr.getattr("val") # 所有表达式都有一个值，三地址码左边的虚拟寄存器是
val
    mv.visitAssignment(new_var.temp, value) # 生成了一条三地址码的赋值语句
```

首先我们获得 AST 节点的 varsymbol ，并给他分配一个新的寄存器，由于 TAC 部分的寄存器都是虚拟寄存器，所以我们在这个阶段可以分配任意多的虚拟寄存器。

接下来如果这个变量有初始值的话，那么我们需要先访问初始值节点，并且在访问结束后将初始值赋给当前声明的变量。

思考题

1. `addi sp, sp, -16`
2. 对于定义变量而言，当有变量重复声明的时候，我们可以考虑在我们的编译过程中将其视为两个不同的变量。具体来说，对于例子中的变量 a 的第二次声明，我们可以将其存为变量 a_2 。也就是说，对于变量 a 的第 k 次声明，在我们的解析中，我们将其命名为 a_{k + 1}。这样就可以解决命名冲突的问题。

   对于查找变量的时候，我们总是返回最新版本的变量。这样就能实现后声明的变量将之前声明的变量“隐藏”起来。那么对于变成的用户而言，后声明的变量就将之前声明变量的值“覆盖”掉了。

Step 6

实验内容

这一阶段我们主要实现条件表达式。具体来说，我们需要在 namer.py 和 tacgen.py 文件中实现 visitCondExpr 函数。

对于 namer.py ，我们只需要顺序访问条件表达式的三个部分即可，代码如下

```
expr.cond.accept(self, ctx)
expr.then.accept(self, ctx)
expr.otherwise.accept(self, ctx)
```

对于 tacgen.py 而言，我们要生成对应的 tac 代码

```
skipLabel = mv.freshLabel()
exitLabel = mv.freshLabel()
temp = mv.freshTemp()
expr.setattr("val", temp)

# visit condition
expr.cond.accept(self, mv)
mv.visitCondBranch(
    tacop.CondBranchOp.BEQ, expr.cond.getattr("val"), skipLabel
)

expr.then.accept(self, mv)
mv.visitAssignment(temp, expr.then.getattr("val"))
mv.visitBranch(exitLabel) # jump to exit

mv.visitLabel(skipLabel)
expr.otherwise.accept(self, mv)
mv.visitAssignment(temp, expr.otherwise.getattr("val"))

mv.visitLabel(exitLabel)
```

对于条件表达式，我们可以将其简单理解为有 else 的 if 语句。因此，我们需要两个标签 skiptag 和 exittag。对于每一个分支内部的逻辑与 If 的逻辑大致相同，不再赘述。

这里需要注意的是 条件表达式 需要有自己的值，因此我们首先给条件表达式分配了一个寄存器。当走到 then 分支的时候，我们用 assign 将分支将 then 语句的值赋给 condexpr。同理，当程序走到 otherwise 分支的时候，我们同样使用 assign 进行赋值。

思考题

1. 对于 python 框架而言，处理 else 悬挂的代码部分如下

```
87  def p_statement(p):
88      """
89      statement : statement_matched
90                | statement_unmatched
91      """
92      p[0] = p[1]
93
94
95  def p_if_else(p):
96      """
97      statement_matched : If LParen expression RParen statement_matched Else statement_matched
98      statement_unmatched : If LParen expression RParen statement_matched Else statement_unmatched
99      """
100     p[0] = If(p[3], p[5], p[7])
101
102
103 def p_if(p):
104     """
105     statement_unmatched : If LParen expression RParen statement
106     """
107     p[0] = If(p[3], p[5])
```

这里注意到 P_if_else 解析中 if 和 else 之间只能生成 statement_matched ，而根据产生式规则，statement_matched 只能生成成对的 if 和 else。

而对于那些只有 if 没有 else 的情况，会被归约为 statement_unmathced 的情况，而肯定不能被归约为 statement_matched

通过上述行为，Python 框架实现了 if 和 else 的就近匹配，解决了 else 的悬挂二义性的问题。

2. 对于思考题中的例子，我们现在实现的是条件判断之后直接跳转到不同的 Label 执行相应的代码，因此会造成短路现象。

   如果我们想改进成不短路，我们需要首先计算两个分支的值，再去根据条件选择赋哪一个值。

```
expr.otherwise.accept(self, mv)
expr.then.accept(self, mv)
expr.cond.accept(self, mv)
mv.visitCondBranch(
tacop.CondBranchOp.BEQ, expr.cond.getattr("val"), skipLabel
)
```

具体实现中，在 tacgen 这一步，我们先去访问 then 和 otherwise 的分支，最后再根据 cond 跳转，为目标变量选择最后的值。