STAGE-4

计13 占海川 2021050009

STEP-7

实验内容

语义分析

`frontend/typecheck/namer.py`

- 依次访问条件表达式的类成员`cond, then, otherwise`, 且子节点一定存在

  ```
  def visitCondExpr(self, expr: ConditionExpression, ctx: ScopeStack) -> None:
      expr.cond.accept(self, ctx)
      expr.then.accept(self, ctx)
      expr.otherwise.accept(self, ctx)
  ```

中间代码生成

`frontend/tacgen/tacgen.py`

- 访问条件表达式的子节点, 并添加对应TAC语句, 并为条件表达式的返回值分配临时寄存器

```
def visitCondExpr(self, expr: ConditionExpression, mv: TACFuncEmitter) -> None:
        expr.cond.accept(self, mv)

        # 分配临时变量
        temp = mv.freshTemp()
        skipLabel = mv.freshLabel()
        exitLabel = mv.freshLabel()

        mv.visitCondBranch (
            # 为0跳转到L2
            tacop.CondBranchOp.BEQ, expr.cond.getattr("val"), skipLabel
        )
        # L1
        expr.then.accept(self, mv)
        mv.visitAssignment(temp, expr.then.getattr("val"))
        mv.visitBranch(exitLabel)
        # L2
        mv.visitLabel(skipLabel)
        expr.otherwise.accept(self, mv)
        mv.visitAssignment(temp, expr.otherwise.getattr("val"))
        mv.visitLabel(exitLabel)

        expr.setattr("val", temp)
```

思考题

1. 在`frontend/parser/ply_parser`文法分析部分，`If`与`Else`之间为`statement_matched`，即中间一定不会是仅有一个if分支的情形.
2. 将对子节点`cond,then,otherwise`的访问提前，再调用`visit`加入TAC语句

STEP-8

实验内容

抽象语法树

`frontend/ast/tree.py`

- 仿照While与Break添加For与Continue类

```
class For(Statement):
"""
AST node of For statement.
"""
def __init__(self, init: Expression, cond: Expression, update: Expression, body: Statement
    super().__init__("for")
    self.init = init
    self.cond = cond
    self.update = update
    self.body = body

def __getitem__(self, key: int) -> Node:
    return (self.cond, self.body)[key]

def __len__(self) -> int:
    return 2

def accept(self, v: Visitor[T, U], ctx: T):
    return v.visitFor(self, ctx)


class Continue(Statement):
"""
AST node of Continue statement.
"""
def __init__(self) -> None:
    super().__init__("continue")

def __getitem__(self, key: int) -> Node:
    raise _index_len_err(key, self)

def __len__(self) -> int:
    return 0

def accept(self, v: Visitor[T, U], ctx: T):
    return v.visitContinue(self, ctx)

def is_leaf(self):
    return True
```

`frontend/ast/visitor.py`

- 仿照visitWhile与visitBreak添加visitFor与visitContinue类

```
def visitFor(self, that: For, ctx: T) -> Optional[U]:
    return self.visitOther(that, ctx)


def visitContinue(self, that: Continue, ctx: T) -> Optional[U]:
    return self.visitOther(that, ctx)
```

语法词法分析

`frontend/lexer/lex.py`

- 加入reserved keyword

```
"for": "For",
"continue": "Continue"
```

`forntend/parse/ply_parser.py`

- 仿照while与break的实现,添加对for与continue的语法分析

```
def p_for(p):
# 这里的for语句的init可能是expression也可能是declaration
"""
statement_matched : For LParen declaration Semi expression Semi expression RParen statemen
statement_matched : For LParen expression Semi expression Semi expression RParen statement_
statement_unmatched : For LParen declaration Semi expression Semi expression RParen statem
statement_unmatched : For LParen expression Semi expression Semi expression RParen stateme
"""
p[0] = For(p[3], p[5], p[7], p[9])


def p_continue(p):
"""
statement_matched : Continue Semi
"""
p[0] = Continue()
```

语义分析

`frontend/typecheck/namer.py`

- 对visitWhile进行修改, 添加对循环层数的记录

```
def visitWhile(self, stmt: While, ctx: ScopeStack) -> None:
    stmt.cond.accept(self, ctx)
    ctx.openloop()
    stmt.body.accept(self, ctx)
    ctx.closeloop()
```

- 仿照前者完成visitFor

```
def visitFor(self, stmt: For, ctx: ScopeStack) -> None:
    ctx.push(Scope(ScopeKind.LOCAL)) # 新建一个局部作用域

    stmt.init.accept(self, ctx)
    stmt.cond.accept(self, ctx)
    stmt.update.accept(self, ctx)

    ctx.openloop()
    stmt.body.accept(self, ctx)
    ctx.closeloop()
    ctx.pop()
```

- 在visitBreak与visitContinue中添加对是否在循环内的检查

```
if ctx.checkLoop() == 0:
        raise DecafBreakOutsideLoopError()
```

中间代码生成

`frontend/tacgen/tacgen.py`

- 仿照visitWhile实现visitFor

```
def visitFor(self, stmt: For, mv: TACFuncEmitter) -> None:
    beginLabel = mv.freshLabel()
    loopLabel = mv.freshLabel()
    breakLabel = mv.freshLabel()
    mv.openLoop(breakLabel, loopLabel)

    stmt.init.accept(self, mv)

    mv.visitLabel(beginLabel)
    stmt.cond.accept(self, mv)
    mv.visitCondBranch(tacop.CondBranchOp.BEQ, stmt.cond.getattr("val"), breakLabel)

    stmt.body.accept(self, mv)
    mv.visitLabel(loopLabel)
    stmt.update.accept(self, mv)

    mv.visitBranch(beginLabel)
    mv.visitLabel(breakLabel)
    mv.closeLoop()
```

- 仿照visitBreak实现visitContinue

```
def visitContinue(self, stmt: Continue, mv: TACFuncEmitter) -> None:
    mv.visitBranch(mv.getContinueLabel())
```

思考题

1. 对于第一种翻译, 从本轮迭代完成到判断是否满足条件之间需要一步跳转指令, 而第二种body与cond之间不需要跳转, 少执行一次指令
2. 我认为单目标分支指令更合理. 在如下情况下

```
FUNCTION<main>:
    _T1 = 0
    _T0 = _T1
    _T2 = 1
    if (_T2 == 0) branch _L1
    _T4 = 2
    _T3 = _T4
    branch _L2
_L1:
    _T5 = 3
    _T3 = _T5
_L2:
    _T0 = _T3
    return _T0
```

单分支指令只需要执行一次跳转, 对于双分支指令, 跳转到L1并执行完毕后还需要跳过L2分支以执行剩余指令