Stage 4 报告

程思翔 2021010761

1 实验内容

1.1 step 7

语义分析: 直接全部访问 `ConditionExpression` 的子节点即可.

```
# frontend/typecheck/namer.py
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: 直接参考对 `if` 语句的实现, 由于 `ConditionExpression` 需要返回值, 在 `mv.visitCondBranch` 结束后, 使用 `expr.cond.getattr("val")` 储存该表达式的值 (这也是与 `visitIf` 的不同点).

```
# frontend/tacgen/tacgen.py
def visitCondExpr(self, expr: ConditionExpression, mv:
TACFuncEmitter) -> None:
    expr.cond.accept(self, mv)
    skipLabel = mv.freshLabel()
    exitLabel = mv.freshLabel()
    exprVal = expr.cond.getattr("val")
    mv.visitCondBranch(
        tacop.CondBranchOp.BEQ, exprVal, skipLabel
    )
    expr.then.accept(self, mv)
    mv.visitAssignment(exprVal, expr.then.getattr("val")) #
Here!
    mv.visitBranch(exitLabel)
    mv.visitLabel(skipLabel)
```

```
    expr.otherwise.accept(self, mv)
    mv.visitAssignment(exprVal, expr.otherwise.getattr("val")) # Here!
    mv.visitLabel(exitLabel)
    expr.setattr("val", exprVal) # Here!
```

1.2 step 8

这一部分需要实现 `For` 与 `Continue` 语句.

词法&语法分析: 在 `frontend/lexer/lex.py` 添加 `For` 与 `Continue` 的关键字类型, 在 `frontend/ast/tree.py` 和 `frontend/ast/visitor.py` 实现 AST 节点及访问函数.

```python
# frontend/ast/tree.py
class For(Statement):
    def __init__(
        self,
        init: Expression,
        cond: Expression,
        update: Expression,
        body: Statement
    ) -> None:
        super().__init__("for")
        self.init = init
        self.cond = cond
        self.update = update
        self.body = body
    ...

class Continue(Statement):
    def __init__(self) -> None:
        super().__init__("continue")
    ...
```

其中 `For` 节点的成员可能为空, 但为了语法分析的简便, 没有使用 `Optional` 进行定义. 为消除可能带来的隐患, 只需在**中间代码生成**中添加相应检查即可. 定义上下文无关文法, 注意 `init` 成员可能为 `Declaration` 或 `Expression`.

```
# frontend/parser/ply_parser.py
def p_for(p):
    """
    statement_matched : For LParen init_expression Semi opt_expression Semi opt_expression RParen statement_matched
    statement_unmatched : For LParen init_expression Semi opt_expression Semi opt_expression RParen statement_unmatched
    """
    p[0] = For(p[3], p[5], p[7], p[9])

def p_for_init(p):
    """
    init_expression : opt_expression
    init_expression : declaration
    """
    p[0] = p[1]
```

语义分析: 在 `ScopeStack` 数据结构中增加维护当前 `loop` 的层数.

```
# frontend/scope/scopestack.py
class ScopeStack:
    def __init__(self, globalScope: Scope) -> None:
        ...
        self.loopCount = 0

    def enterLoop(self) -> None:
        self.loopCount += 1

    def exitLoop(self) -> None:
        self.loopCount -= 1

```

```
    def insideLoop(self) -> None:
        return self.loopCount
```

为 `For` 和 `While` 循环添加 `loop` 层数信息, 同时在进入 `For` 循环体时打开一个新的 `scope`.

```
# frontend/typecheck/namer.py
def visitFor(self, stmt: For, ctx: ScopeStack) -> None:
    ctx.open()
    stmt.init.accept(self, ctx)
    stmt.cond.accept(self, ctx)
    stmt.update.accept(self, ctx)
    ctx.enterLoop()
    stmt.body.accept(self, ctx)
    ctx.exitLoop()
    ctx.close()

def visitWhile(self, stmt: While, ctx: ScopeStack) -> None:
    stmt.cond.accept(self, ctx)
    ctx.enterLoop()
    stmt.body.accept(self, ctx)
    ctx.exitLoop()

def visitBreak(self, stmt: Break, ctx: ScopeStack) -> None:
    if not ctx.insideLoop():
        raise DecafBreakOutsideLoopError()
    ctx.exitLoop()

def visitContinue(self, stmt: Continue, ctx: ScopeStack) ->
None:
    if not ctx.insideLoop():
        raise DecafContinueOutsideLoopError()
```

中间代码生成: 参考 `visitWhile` 实现了 `visitFor`, 参考 `visitBreak` 实现了 `visitContinue`. `For` 循环的控制流参考了这里给出的 TAC 码, 需要注意由于我们在 AST 节点实现中挖的坑, `stmt.cond` 可能为 `NULL`, 需要以此为依据判定是否添加 `BEQ` 指令跳转 (虽然测例里都是完整的 `for` 循环, 不会出现这个问题).

```
# frontend/tacgen/tacgen.py
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)
    if stmt.cond:
        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()
```

2 思考题

2.1 step 7

1. 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.

   答: `statement` 有 `statement_matched` 代表 `if` 与 `else` 匹配, 以及 `statement_unmatched` 代表仅有 `if`.

若出现 `if`, `else` 匹配, 其之间一定是 `statement_matched` 的匹配类型, 这使得后续的悬吊 `else` 只能与同层的 `if` 结合, 构成 `statement_matched` 后, 这就是 `else` 与一个最近未匹配 `if` 的匹配.

2. 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.

答: 修改 `frontend/tacgen/tacgen.py:visitCondExpr`, 将 `expr.then.accept(self, mv)`, `expr.otherwise.accept(self, mv)` 均提至函数首部即可, 这样条件表达式必定会访问 `then` 及 `otherwise`, 从而避免短路.

2.2 step 8

1. 将循环语句翻译成 IR 有许多可行的翻译方法, 例如 `while` 循环可以有以下两种翻译方式:

   第一种 (即实验指导中的翻译方式):

   - `label BEGINLOOP_LABEL`: 开始下一轮迭代
   - `cond 的 IR`
   - `beqz BREAK_LABEL`: 条件不满足就终止循环
   - `body 的 IR`
   - `label CONTINUE_LABEL`: continue 跳到这
   - `br BEGINLOOP_LABEL`: 本轮迭代完成
   - `label BREAK_LABEL`: 条件不满足, 或者 break 语句都会跳到这儿

   第二种:

- `cond 的 IR`
- `beqz BREAK_LABEL`: 条件不满足就终止循环
- `label BEGINLOOP_LABEL`: 开始下一轮迭代
- `body 的 IR`
- `label CONTINUE_LABEL`: continue 跳到这
- `cond 的 IR`
- `bnez BEGINLOOP_LABEL`: 本轮迭代完成，条件满足时进行下一次迭代
- `label BREAK_LABEL`: 条件不满足，或者 break 语句都会跳到这儿

从执行的指令的条数这个角度 (`label` 不算做指令, 假设循环体至少执行了一次), 请评价这两种翻译方式哪一种更好?

答: 第二种翻译方式更好. 假设循环执行到不满足 `cond` 结束, 第一种翻译每个循环需要执行 `body`, `cond` 以及两次跳转, 而第二种翻译除了首次进入循环需要判定 `cond`, 其余每个循环只需执行一次跳转. 就普遍意义而言, 第二种翻译方式生成的程序比第一种每个循环少执行一次跳转, 这种翻译方式更好.

2. 我们目前的 TAC IR 中条件分支指令采用了单分支目标 (标签) 的设计, 即该指令的操作数中只有一个是标签; 如果相应的分支条件不满足, 则执行流会继续向下执行. 在其它 IR 中存在双目标分支 (标签) 的条件分支指令, 其形式如下:

```
br cond, false_target, true_target
```

其中 `cond` 是一个临时变量, `false_target` 和 `true_target` 是标签. 其语义为: 如果 `cond` 的值为 0 (假), 则跳转到 `false_target` 处; 若 `cond` 非 0 (真), 则跳转到 `true_target` 处. 它与我们的条件分支指令的区别在于执行流总是会跳转到两个标签中的一个. 你认为中间表示的哪种条件分支指令设计 (单目标 vs 双目标)更合理? 为什么?

答: 我认为选择 "双目标分支" 更合理:

- 双目标分支指令更加灵活, 符合编程语言的控制流结构, 允许根据条件跳转到两个不同位置, 在实现 `if-elif-else` 等结构会更方便.

- 双目标分支指令含义直观, 容易理解和调试, 能够提高代码的可读性和维护性.