STAGE-3：作用域和循环

计04 何秉翔 2020010944

1. 实验内容

1.1 step-7：作用域和块语句

1.1.1 语义分析

在中端 `namer.py` 中，访问到块语句 `Block` 时需要压入一个新的作用域，在离开块语句时需要弹出该作用域。

```
def visitBlock(self, block: Block, ctx: ScopeStack) -> None:
    ctx.open(Scope(ScopeKind.LOCAL))
    for child in block:
        child.accept(self, ctx)
    ctx.close()
```

1.1.2 目标代码生成

此外，对于数据流图 `CFG` 上的每个节点，我们都需要判断该节点是否可达，在 `backend/dataflow/cfg.py` 中新增一个方法用于判断节点 `self.nodes[id]` 是否可达，为了实现此方法，我们考虑在 `CFG` 初始化时就先对每个节点判断是否可达，采用 `BFS` 实现：

```
self.reachableNodes = [False] * len(self.nodes) # flag for all nodes, indicates whether can be reached

import queue
queue = queue.Queue()
queue.put(0) # add the first node
while not queue.empty():
    curNodeId = queue.get() # get the front node
    self.reachableNodes[curNodeId] = True
    for id in self.getSucc(curNodeId): # trace all the succ nodes
        if not self.reachableNodes[id]: # if not reachable
            queue.put(id) # add into the queue
```

其中，`self.reachableNodes` 记录了每个节点是否可达。

于是新增的方法可以为：

```
def reachable(self, id): # tell whether the block self.nodes[id] can be reached
    return self.reachableNodes[id]
```

接下来在 `backend/reg/bruteregalloc.py` 中的 `accept` 方法中，判断基本块是否可达，若不可达则不必分配寄存器。

```
if not graph.reachable(bb.id): # not reachable
    continue
if bb.label is not None:
    subEmitter.emitLabel(bb.label)
self.localAlloc(bb, subEmitter)
```

1.2 step-8: 循环语句

此步骤不需要关注目标代码的生成，只需关注前面的即可。

1.2.1 词法语法分析

首先在 `lex.py` 中新定义一些关键词：

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

新建立一些 `AST` 节点，需注意 `for` 循环的 `cond` 若为 `NULL`，需要补成 `1`：

```
class DoWhile(Statement):
    """
    AST node of doWhile statement.
    """

    def __init__(self, cond: Expression, body: Statement) -> None:
        super().__init__("doWhile")
        self.cond = cond
        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.visitDoWhile(self, ctx)


class For(Statement):
    """
    AST node of for statement.
    """

    def __init__(
        self,
        init: Union[Declaration, Expression],
        cond: Expression,
        update: Expression,
        body: Statement
    ) -> None:
        super().__init__("for")
        self.init = init
        self.cond = cond if cond != NULL else IntLiteral(1) # replaced by non-zero constant like 1
        self.update = update
        self.body = body

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

    def __len__(self) -> int:
        return 4

```

```
    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
```

在 `ply_parser.py` 中设置词法语法解析规则：

```
def p_do_while(p):
    """
    statement_matched : Do statement_matched While LParen expression RParen Semi
    statement_unmatched : Do statement_unmatched While LParen expression RParen Semi
    """
    p[0] = DoWhile(p[5], p[2])


def p_for(p):
    """
    statement_matched : For LParen opt_expression Semi opt_expression Semi opt_expression RParen statement_matched
      | For LParen declaration Semi opt_expression Semi opt_expression RParen statement_matched
    statement_unmatched : For LParen opt_expression Semi opt_expression Semi opt_expression RParen statement_unmatched
      | For LParen declaration Semi opt_expression Semi opt_expression Semi opt_expression RParen statement_unmatched
    """
    p[0] = For(p[3], p[5], p[7], p[9])

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

1.2.2 语义分析

在 `visitor.py` 中新建立一些访问节点的访问器：

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

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)
```

接下来考虑中端的 `namer.py`，参考 `while` 和 `break` 的实现，在其中实现新增加的三个树节点的 `visit` 函数，并添加对应的作用域即可：

```
def visitFor(self, stmt: For, ctx: ScopeStack) -> None:
    ctx.open(Scope(ScopeKind.LOCAL)) # local scope

    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.close()

def visitDoWhile(self, stmt: DoWhile, ctx: ScopeStack) -> None:
    ctx.openLoop()
    stmt.body.accept(self, ctx)
    ctx.closeLoop()
    stmt.cond.accept(self, ctx)

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

1.2.3 中间代码生成

接下来在 `tacgen.py` 中，参考 `while` 和 `break` 的实现，在其中实现新增加的三个树节点的 `visit` 函数。`dowhile` 相比 `while` 只需调换 `cond` 和 `body` 的执行顺序，`for` 相比 `while` 只需在合适的位置加上 `init` 和 `update` 的执行即可：

```
def visitDoWhile(self, stmt: DoWhile, mv: FuncVisitor) -> None:
    beginLabel = mv.freshLabel()
    loopLabel = mv.freshLabel()
    breakLabel = mv.freshLabel()
    mv.openLoop(breakLabel, loopLabel)

    mv.visitLabel(beginLabel)
    stmt.body.accept(self, mv)

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

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

def visitFor(self, stmt: For, mv: FuncVisitor) -> 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()

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

2. 思考题

2.1 step-7 思考题

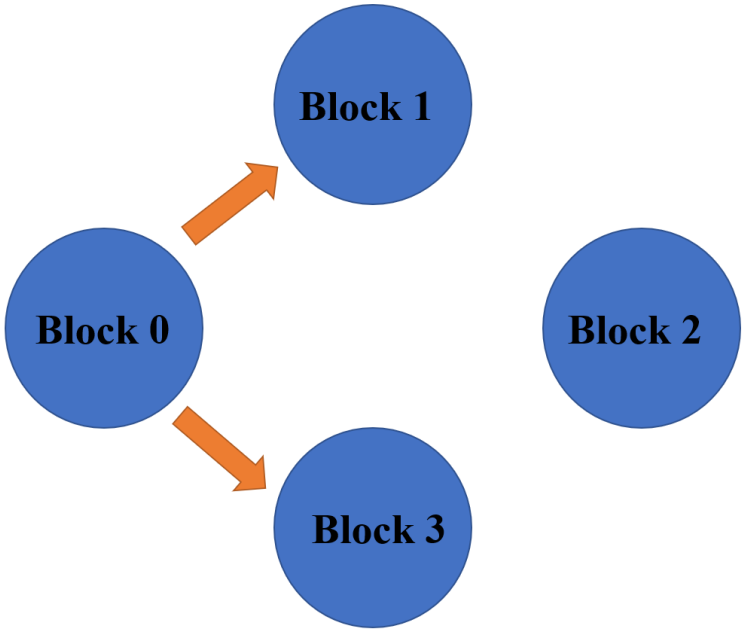
1. **问题**：请画出下面 MiniDecaf 代码的控制流图。

```
int main(){
 int a = 2;
 if (a < 3) {
     {
         int a = 3;
         return a;
     }
     return a;
 }
}
```

解答：对应的 `TAC` 为：

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

`CFG` 为：



2.2 step-8 思考题

问题：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）：

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

第二种：

1. `cond 的 IR`
2. `beqz BREAK_LABEL`：条件不满足就终止循环
3. `label BEGINLOOP_LABEL`：开始下一轮迭代
4. `body 的 IR`
5. `label CONTINUE_LABEL`：continue 跳到这
6. `cond 的 IR`
7. `bnez BEGINLOOP_LABEL`：本轮迭代完成，条件满足时进行下一次迭代
8. `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` 指令不计算在内，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?

解答：设 `cond` 的 `IR` 有 `m` 条，`body` 的 `IR` 平均执行了 `n` 条（break 和 continue），循环体执行了 `p` ($p \geq 1$) 次，则：

第一种方法的执行条数为：$p \times (m + n + 2) + m + 1$

第二种方法的执行条数为：$m + 1 + p \times (m + n + 1)$

第二种方法更优。