

# Stage-3 Report

曹伦郗 2020011020

## 实验内容

### step7

#### 语义分析

在`Namer.visitBlock`中，在访问块内部每条语句之前，创建一个局部作用域对象，再调用`ctx.open`以开启这个新的作用域，将其加入作用域栈顶。而访问块内部每条语句结束后，调用`ctx.close`将此作用域关闭，即退栈。

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

#### 控制流和数据流分析

实现了`CFG.unreachable`，具体实现方式为：

新增了一个成员数组`reachable`，代表从0号基本块能否到达该号基本块。在`__init__`并对其做全`False`初始化后，从0号基本块开始进行DFS，将访问到的每个基本块的`reachable`数组对应项置为`True`，表示可达。

值得注意的是，由于`step8`中加入了循环语句，导致控制流图中出现了环路，因此，每个被访问到的基本块，在递归地访问其后继的各个基本块时，只访问那些尚未确定可达，即满足`reachable[i] == False`的基本块，以避免无穷调用。

如此，在初始化阶段，就确定了每个基本块是否可达。`CFG.unreachable`直接根据成员`reachable`数组的内容，返回对应项的值取反即可。

```
def __init__(self, nodes: list[BasicBlock], edges: list[(int, int)]) ->
None:
    self.nodes = nodes
    self.edges = edges

    self.links = []

    for i in range(len(nodes)):
        self.links.append((set(), set()))

    for (u, v) in edges:
        self.links[u][1].add(v)
        self.links[v][0].add(u)

    self.reachable = [False for _ in range(len(nodes))]
    self.dfs(0)
```

```
def dfs(self, root):
    self.reachable[root] = True
    for child in self.getSucc(root):
        if not self.reachable[child]:
            self.dfs(child)
```

```
def unreachable(self, id):
    return not self.reachable[id]
```

## 寄存器分配

修改了 *BruteRegAlloc.accept*, 对控制流图中的每个基本块 *bb*, 先调用 *bb.unreachable*, 若其不可达, 则跳过它。

```
def accept(self, graph: CFG, info: SubroutineInfo) -> None:
    subEmitter = self.emitter.emitSubroutine(info)
    for bb in graph.iterator():
        # you need to think more here
        # maybe we don't need to alloc regs for all the basic blocks
        if graph.unreachable(bb.id):
            continue
        if bb.label is not None:
            subEmitter.emitLabel(bb.label)
        self.localAlloc(bb, subEmitter)
    subEmitter.emitEnd()
```

## step8

### 语法分析

在 *tree.py* 中添加了 *For*、*DoWhile*、*Continue* 三个类, 作为分别对应 *for*、*do while*、*continue* 语句的 AST 节点类, 其中大部分方法的实现可参照 *While* 和 *Break* 类。

值得注意的是, *for* 语句的 *init* 既可能是声明也可能是表达式, 而初始化参数中除循环体 *body* 外, *init*、*cond*、*update* 都可能为空。其中, 若 *cond* 为空, 对应的数据成员应为整数字面量 1, 即 *IntLiteral(1)*。

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

    def __init__(
        self, init: Declaration | Expression, cond: Expression, update:
        Expression, body: Statement) -> None:
        super().__init__('for')
        self.init = init
        self.cond = cond if not cond is NULL else IntLiteral(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 Dowhile(Statement):
    """
    AST node of do while statement.
    """

    def __init__(self, body: Statement, cond: Expression) -> None:
        super().__init__('do while')
        self.body = body
        self.cond = cond

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

    def __len__(self) -> int:
        return 2

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

```

而在Visitor类下添加了visitFor、visitDoWhile、visitContinue。

```

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

def visitDowhile(self, that: Dowhile, ctx: T) -> Optional[U]:
    return self.visitOther(that, ctx)

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

```

在`lex.py`下保留字`reserved`下添加了`for`、`do`、`continue`三个关键字。

```

# Reserved keywords
reserved = {
    "return": "Return",
    "int": "Int",
    "if": "If",
    "else": "Else",
    "while": "While",
    'for': 'For',
    'do': 'Do',
    'continue': 'Continue',
    "break": "Break",
}

```

在`ply_parser.py`下添加了`for`循环、`do while`循环、`continue`语句的对应文法。由于`for`循环的`init`、`cond`、`update`都可能为空，故产生式右侧使用了非终结符`opt_expression`。

```

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

```

```

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[2], p[5])

```

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

## 语义分析

实现了Namer.visitFor, 即:

- ①为for循环开启一个新的局部作用域;
- ②若init非空, 则访问之; 访问cond; 若update非空, 则访问之;
- ③调用ctx.openLoop记录循环层数;
- ④访问body;
- ⑤调用ctx.closeLoop, 代表该层循环结束;
- ⑥关闭该局部作用域。

```
def visitFor(self, stmt: For, ctx: ScopeStack) -> None:
    """
    1. Open a local scope for stmt.init.
    2. Visit stmt.init, stmt.cond, stmt.update.
    3. Open a loop in ctx (for validity checking of break/continue)
    4. Visit body of the loop.
    5. Close the loop and the local scope.
    """
    ctx.open(Scope(ScopeKind.LOCAL))
    if not stmt.init is NULL:
        stmt.init.accept(self, ctx)
    stmt.cond.accept(self, ctx)
    if not stmt.update is NULL:
        stmt.update.accept(self, ctx)
    ctx.openLoop()
    stmt.body.accept(self, ctx)
    ctx.closeLoop()
    ctx.close()
```

实现了Namer.visitDoWhile, 即:

- ①调用ctx.openLoop记录循环层数;
- ②访问body;
- ③调用ctx.closeLoop, 代表该层循环结束;
- ④访问cond;

```
def visitDowhile(self, stmt: Dowhile, ctx: ScopeStack) -> None:
    """
    1. Open a loop in ctx (for validity checking of break/continue)
    2. Visit body of the loop.
    3. Close the loop.
    4. Visit the condition of the loop.
    """
    ctx.openLoop()
    stmt.body.accept(self, ctx)
    ctx.closeLoop()
    stmt.cond.accept(self, ctx)
```

实现了 *Namer.visitContinue*，和 *Namer.visitBreak* 行为一致，只需通过检查当前循环层数，确保 *continue* 语句位于某循环内，否则抛出 *DecafContinueOutsideLoopError*。

```
def visitContinue(self, stmt: Continue, ctx: ScopeStack) -> None:
    """
    1. Refer to the implementation of visitBreak.
    """
    if not ctx.inLoop():
        raise DecafContinueOutsideLoopError()
```

## 中间代码生成

实现了 *TACGen.visitFor*、*TACGen.visitDoWhile*、*TACGen.visitContinue*。

前两方法同 *TACGen.visitWhile* 实现类似，调整循环体、控制表达式、跳转标签等部分的位置即可。

*TACGen.visitContinue* 也与 *TACGen.visitBreak* 类似。

```
def visitFor(self, stmt: For, mv: FuncVisitor) -> None:
    beginLabel = mv.freshLabel()
    loopLabel = mv.freshLabel()
    breakLabel = mv.freshLabel()
    mv.openLoop(breakLabel, loopLabel)
    if not stmt.init is NULL:
        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)
    if not stmt.update is NULL:
        stmt.update.accept(self, mv)
    mv.visitBranch(beginLabel)
    mv.visitLabel(breakLabel)
    mv.closeLoop()
```

```

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)
    mv.visitLabel(loopLabel)
    stmt.cond.accept(self, mv)
    mv.visitCondBranch(tacop.CondBranchOp.BEQ, stmt.cond.getattr('val'),
breakLabel)
    mv.visitBranch(beginLabel)
    mv.visitLabel(breakLabel)
    mv.closeLoop()

```

```

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

```

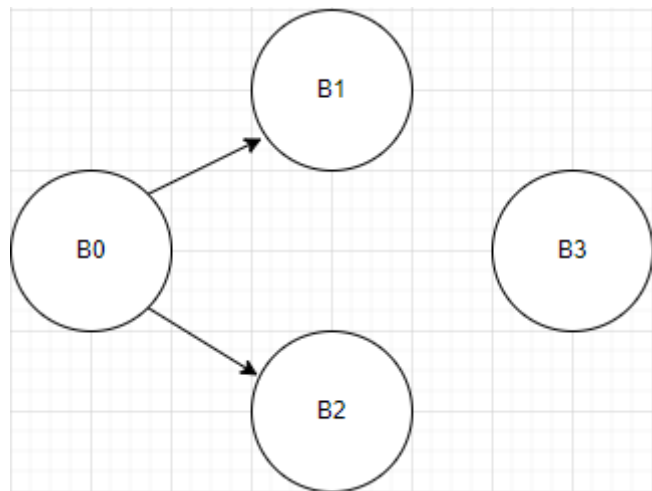
## 思考题

### step7

将该程序转换为TAC指令序列如下，方便划分基本块：

<b>_Main.max:</b>	
<b>_T0 = 2</b>	
<b>_T1 = 3</b>	B0
<b>_T2 = (_T0 &lt; _T1)</b>	
<b>if (_T2 == 1) branch _L2</b>	_____
<b>return</b>	___ B1 _____
<b>_L2:</b>	
<b>_T0 = 3</b>	B2
<b>return _T0</b>	_____
<b>_L3:</b>	
<b>return _T0</b>	B3

控制流图为：



## step8

设循环体执行了 $n$ 次。

则两种翻译方式下 $cond$ 和 $body$ 部分执行次数一样，因此在这两部分，两种翻译方式下CPU执行的指令条数一样多。

除去 $cond$ 和 $body$ 部分指令：

第一种翻译方式下，CPU执行的指令条数为 $2n + 1$ ，其中 $n + 1$ 条 $beqz$ 指令， $n$ 条 $br$ 指令；

而第二种翻译方式下，CPU执行的指令条数为 $n + 1$ ，其中1条 $beqz$ 指令， $n$ 条 $bnez$ 指令。

由于循环体执行次数一定的情况下，执行的指令条数越少越好，而第二种翻译方式少执行 $n$ 条指令，且考虑到实际情况中，一段程序中所有 $while$ 循环的循环体执行次数均为0的概率较低，因此第二种翻译方式更好。