Stage-3 实验报告

> 沙之洲 2020012408

Step 7

实验内容

在这一阶段，我们需要支持局部作用域。具体来说，要在 visitBlock 的时候打开一个新的作用域，namer.py 的代码如下

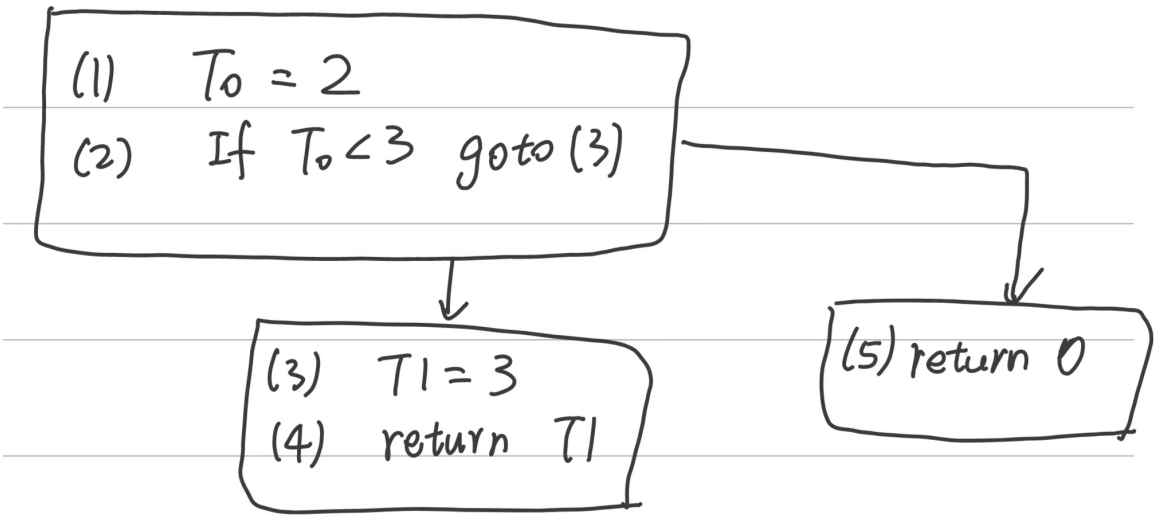
```
        func.body.accept(self, ctx) # then visit the block of the function, recursively

    def visitBlock(self, block: Block, ctx: ScopeStack) -> None:
        # a function block is constructed by several sentence, we should visit all of them

        # we have to define a scope for each block
        # open a scope here
        ctx.open(Scope(ScopeKind.LOCAL))

        for child in block:
            child.accept(self, ctx)

        # close the scope
        ctx.close()
```

对于后端而言，这一阶段我们要考虑寄存器的分配。我们知道中间代码 tac 可以根据一些规则分为若干个 basic block，这里 bruteallocate 实现的策略是对于每一个 bb 重新分配寄存器，因此我们要做的是只给那些程序可能能到达的 bb 分配寄存器，也就是那些可达的 bb，这里我才用的是 dfs 判断是否可达，具体代码如下

```

    # my codes are below
    def unreachable(self, node) -> bool:
        # true for unreachable, false for reachable

        # serach for index
        id = self.nodes.index(node)

        # Implement dfs here
        stack = []
        stack.append(0)
        have_searched_list = []
        have_searched_list.append(0)

        while stack:
            current = stack.pop()
            nodes = list(self.getSucc(current))
            for i in nodes[::-1]: # since the last in stach, will be poped first, therefore need reverse here
                if i not in have_searched_list:
                    stack.append(i)
                    have_searched_list.append(i)

            if current == id:
                return False

        # unreachable! sad
        return True

```

思考题

控制流图如下



根据题面给出的代码

1. 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;
 }
}
```

发现第 8 行的 return a 实际上是不可达的，因此这一行代码没有出现在 cfg 中。同时根据 C 的规范，main 函数没有返回值的时候，默认返回 0，所以我在 cfg 中将 return 0 显式表达出来了。

Step 8

实验内容

首先我们需要在解析 ast 树的时候增加 for 和 do while 的解析，代码如下

```
    p[0] = While(p[3], p[5])

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 RParen statement_unmatched
                        | 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
    """
    p[0] = For(p[3], p[5], p[7], p[9])

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

def p_continue(p):
```

接下来需要在前端的 namer 和 tacgen 的对应部分增加对应的 visit 函数，这里以 tacgen 中的 visitFor 为例

```
        mv.closeLoop()

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

        if not isinstance(stmt.init, node.NullType):
            stmt.init.accept(self, mv)

        mv.openLoop(breakLabel, loopLabel)

        mv.visitLabel(beginLabel)

        if not isinstance(stmt.cond, node.NullType):
            stmt.cond.accept(self, mv)
            mv.visitCondBranch(tacop.CondBranchOp.BEQ, stmt.cond.getattr("val"), breakLabel)
        # 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 isinstance(stmt.update, node.NullType):
            stmt.update.accept(self, mv)
        mv.visitBranch(beginLabel)

        mv.visitLabel(breakLabel)
        mv.closeLoop()

```

在 visit 函数中需要做的操作包括，打开局部作用域，指定标签等。

思考题

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）：

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?

首先给出结论：第二种翻译方式使用的指令条数更少，因此第二种翻译方式更好。

原因如下：

可以看到第一种和第二种翻译方式的区别主要在于，第一种翻译方式是在进入 body 之前进行 condition 的判断，也就是说，每次 body 结束之后，都需要一个无条件的跳转，才能开始下一轮的 condition 的判断和迭代。但是对于第二种翻译方式，将跳转和 condition 判断这两步操作放到同一条指令中进行，也就是第 7 条的 bnez BEGIN_LABEL 指令。

不难看出，对于每一轮迭代，第一种翻译方式都会比第二种翻译方式多执行一条指令。因此，第二种翻译方式更好。