STAGE3 实验报告

致理-数理 1 赵宇峰 2021013376

1. 实验代码任务

本次实验难度较大，首先需要自己建立一个类，实现作用域栈：

```
from typing import Optional
from frontend.symbol.symbol import Symbol
from .scope import Scope,ScopeKind
default_capacity = 1024
class ScopeStack:
    def __init__(self, globalscope: Scope, capacity: int = default_capacity ) -> None:
        self.stack = [globalscope]
        self.innerstack = []
        self.globalscope = globalscope
        self.stack_capacity = capacity
        self.repeat = 0

    def get_current_scope(self) -> Scope:
        if not self.stack:
            return self.globalscope
        else:
            return self.stack[-1]

    def open(self,scope:Scope) -> None:
        if(len(self.stack)>0):
            self.stack.append(scope)
        else:
            raise OverflowError

    def close(self) -> None:
        if(len(self.stack)>=0):
            self.stack.pop()
        else:
            raise RuntimeError


    def lookup(self, name: str) -> Optional[Symbol]:
        for i in range(len(self.stack)):
            looking_scope = self.stack[len(self.stack)-1-i]
            if (looking_scope.containsKey(name)):
                return looking_scope.get(name)
        return None

    # To declare a symbol.
    def declare(self, symbol: Symbol) -> None:
```

```
            self.get_current_scope().declare(symbol)

        # To check if this is a global scope.
        def isGlobalScope(self) -> bool:
            if(self.stack is not None):
                if (len(self.stack) == 1):
                    return True
            return False
        # To find if there is a name conflict in the current scope.

        def findConflict(self, name: str) -> Optional[Symbol]:
            if self.get_current_scope().containsKey(name):
                self.repeat += 1
                if(self.repeat>=2):
                    raise AttributeError("Cannot redefine!")
                return self.get_current_scope().get(name)
            return None

        def __enter__(self):
            pass

        def __exit__(self, exc_type, exc_val, exc_tb):
            self.innerstack.pop()()

        def local(self):
            self.open(Scope(ScopeKind.LOCAL))
            self.innerstack += [self.close]
            return self

        def global_(self):
            self.open(Scope(ScopeKind.GLOBAL))
            self.innerstack += [self.close]
            return self

        def declare(self, symbol: Symbol) -> None:
            self.get_current_scope().declare(symbol)
```

除了教程中已经指导的函数外，在实际测试过程中还需要实现以下函数：

findConflict：此方法在当前作用域中查找是否存在与给定名称冲突的符号。如果存在冲突，返回该符号，否则返回 None。同时检查是否出现了重复的新定义。

__enter__ 和 __exit__：这是 Python 的上下文管理器协议的一部分，允许使用 with 语句来管理作用域的打开和关闭。在这个实现中，__enter__方法什么也不做，而__exit__方法则调用

close 方法来关闭当前作用域。

local 和 global_：这两个方法用于打开新的本地或全局作用域。它们首先调用 open 方法来打开新的作用域，然后将 close 方法添加到 innerstack 列表中，以便在退出语句时调用，这将在 for 循环等实现中将发挥作用。

依据教程，需要在符号表构建 namer.py 中做以下调整。这里借助 findConflict(decl.ident.value)判定冲突并在当前作用域符号表中插入新的项。

```
    def visitDeclaration(self, decl: Declaration, ctx :ScopeStack) -> None:
        """
        1. Use ctx.lookup to find if a variable with the same name has been declared.
        2. If not, build a new VarSymbol, and put it into the current scope using ctx.declare.
        3. Set the 'symbol' attribute of decl.
        4. If there is an initial value, visit it.
        """

        if ctx.findConflict(decl.ident.value) is not None:
            symbol = ctx.findConflict(decl.ident.value)
            newvar = VarSymbol(decl.ident.value, decl.var_t.type)
            ctx.declare(newvar)
            decl.setattr('symbol', newvar)
        else:
            symbol = ctx.lookup(decl.ident.value)
            if(decl.getattr('symbol') is None):
                newvar = VarSymbol(decl.ident.value, decl.var_t.type)
                ctx.declare(newvar)
                decl.setattr('symbol', newvar)
        if symbol is None:
            newvar = VarSymbol(decl.ident.value, decl.var_t.type)
            ctx.declare(newvar)
            decl.setattr('symbol', newvar)
        if decl.init_expr is not NULL:
            decl.init_expr.accept(self, ctx)
```

cfg.py 中只需要按要求做一次数据流图的 DFS,且仅需对可到达的基本块分配寄存器:

```
        """
        You can start from basic block 0 and do a DFS traversal of the CFG
        to find all the reachable basic blocks.
        """
        self.reachable = set()
        q = [0]
        while q:
            visited_node = q.pop(0)
            self.reachable.add(visited_node)
```

```
for n in self.links[visited_node][1].difference(self.reachable):
    q.append(n)
```

思考题

可以借助实验框架生成 TAC 码：

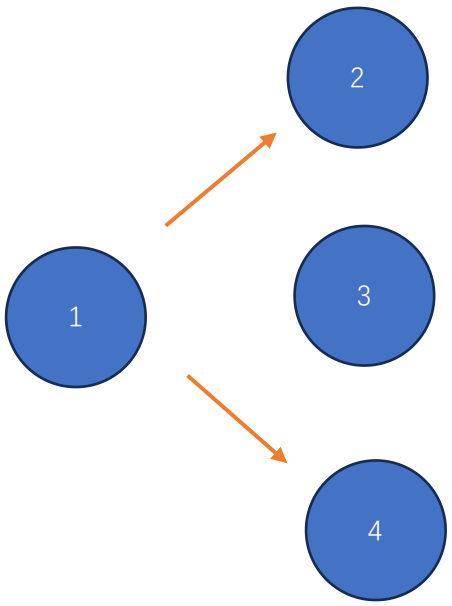
```
FUNCTION<main>:
    _T1 = 2
    _T0 = _T1
    _T2 = 3
    _T3 = (_T0 < _T2)
    if (_T3 == 0) branch _L4 //1
    _T5 = 3
    _T4 = _T5
    return _T4 //2
    return _T0 //3
_L4:
    Return //4
```

并利用规则，可以很容易划分出基本块：基本块的划分算法比较简单：从头到尾扫描操作序列，当遇到以下情况时结束当前基本块，并开始一个新的基本块建立过程：

当遇到一个 Label 标记而且存在跳转语句跳转到这个行号时。

当遇到 Branch、CondBranch 或者 Return 等跳转语句时。

可以得到数据流图如下：



在划分好基本块之后，需要从头到尾依次扫描所有的基本块建立控制流图：

1. 如果当前基本块以 Branch 结尾，则在当前基本块与所跳转到的目标基本块之间加入一条有向边。

2. 如果当前基本块以 CondBranch 结尾，则在当前基本块和跳转条件成立与不成立的目标基本块之间分别加入一条有向边（共 2 条边）。//1->2,1->4

3. 如果当前基本块以 Return 结尾，则不需要加入新的边