stage-3

计11班 周韧平 2021010699

实验内容

本节中要做的是增加块语句的支持，具体来说，原来的代码将所有语句都视作一个块内定义的，而本节中我实现了类 `ScopeStack` 将所有的块以栈的数据结构存储起来。在 `ScopeStack` 中实现了几个功能：

- `__init__`：初始化栈，传入一个全局作用域，并将其作为栈底元素
- `open`：”打开“一个作用域，即将新的局部作用域入栈
- `close`：”关闭“一个作用域，即将栈顶作用域弹出
- `lookup`：和 Scope一样，`ScopeStack`也应该有查找符号的操作，用来在定义和赋值时找到对应的符号，具体来说，该函数接收参数 `only_top` ，当 `only_top=True` 时，仅查找栈顶内是否有该名称变量，`only_top=False` 时候，从栈顶向栈底查找最先出现的符号

```
class ScopeStack:
    def __init__(self, global_scope:Scope):
        self.global_scope = global_scope
        self.stack = [global_scope]
    def open(self, scope:Scope):
        '''
            在作用域栈中打开一个作用域
        '''
        self.stack += [scope]
    def close(self):
        '''
            在作用域栈中关闭一个作用域
        '''
        assert(len(self.stack)>0)
        self.stack = self.stack[:-1]
    def lookup(self, name: str, only_top=False) -> Optional[Symbol]:
        '''
            only_top = True
                找到最接近栈顶的同名变量
            only_top = False
                仅查找栈顶是否存在同名变量
        '''
        if only_top:
            # print(only_top)
            return self.stack[-1].lookup(name)
        # import pdb; pdb.set_trace()
        for scope in reversed(self.stack):
            # pdb.set_trace()
            find_name = scope.lookup(name)
            if find_name is not None:
                return find_name
        return None
    def declare(self, symbol: Symbol) -> None:
        '''
            调用栈顶scope的 declare 方法
        '''

```

```
        self.stack[-1].declare(symbol)
```

在 `Namer` 类中，我将原来的 `Scope` 类改为了 `ScopeStack` 类，在 `visitBlock` 中，每次访问时，我会将当前 Block 推入栈，并递归地访问下面的 Block，在访问结束后，将当前Block弹出

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

此外，我还修改了 `visitDeclaration`，`visitAssignment` 和 `visitIntLiteral` 函数传入地 `ctx`，具体来说，`visitDeclaration` 只需要查询栈顶块，查找本块内是否有重名变量，而其余两者需要查询至栈底，因为对象可能在更外层块内被定义。

此外，在后端，由于引入了块语句，程序的控制流图变得更加复杂，因此我对寄存器分配的部分做了一些修改，在 `CFG` 类初始化时，我引入了 变量，用来存储所有可达的节点，并通过 DFS 算法找到这些节点

```
def __init__(self, nodes: list[BasicBlock], edges: list[(int, int)]) ->
None:
    ......
    self.reachable_nodes = set()
    def dfs(node_index: int, visited: set[int]):
        visited.add(node_index)
        self.reachable_nodes.add(self.nodes[node_index])
        for neighbor in self.links[node_index][1]:
            if neighbor not in visited:
                self.dfs(neighbor, visited)
    dfs(0,set())

......

def unreachable(self,node:BasicBlock):
    if node in self.reachable_nodes:
        return False
    else:
        return True
```

最后，在 `BruteRegAlloc.accept` 方法中，在遍历 `graph` 时，我只为可达的部分分配寄存器

```
 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):
            continue
        if bb.label is not None:
            subEmitter.emitLabel(bb.label)
        self.localAlloc(bb, subEmitter)
    subEmitter.emitEnd()
```

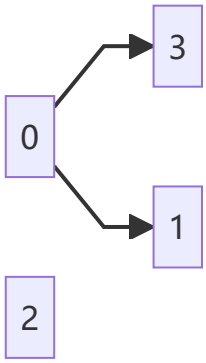
思考题

> 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;
 }
}
```

```
FUNCTION<main>:                 #B_0
    _T1 = 2                     #B_0
    _T0 = _T1                   #B_0
    _T2 = 3                     #B_0
    _T3 = (_T0 < _T2)           #B_0
    if (_T3 == 0) branch _L1    #B_0
    _T5 = 3                     #B_1
    _T4 = _T5                   #B_1
    return _T4                  #B_1
    return _T0                  #B_2
_L1:                            #B_3
    return                      #B_3
```

其中，基本快以 # 的方式标注出来，程序流图如下


3
0
1
2