编译原理实验报告：stage-2

- 于新雨 计25 2022010841

step5

思考题

1. > 请写出一段 risc-v 汇编代码，将栈帧空间扩大 16 字节

- `addi sp,sp ,-16`
- sp寄存器储存了栈顶地址，利用addi指令将sp减去16的值赋给sp

2. > 如果 MiniDecaf 也允许多次定义同名变量，并规定新的定义会覆盖之前的同名定义，请问在你的实现中，需要对定义变量和查找变量的逻辑做怎样的修改？

    - 区分一个作用域中**不同位置**的变量定义：该代码示例中处理方式比较简单，即是同一作用域里面后面用"let"的变量定义覆盖前面的定义即可
    - 所以处理方式也较为简单，而且只涉及前端的语义分析，在visitDeclaration中，如果查找到了相同名字的变量，则直接覆盖原先符号表中定义，即把代码实现中`if ctx.lookup(decl.ident.value):`的条件判断删去即可

新增代码

- 语义分析

    - 考虑到`visitAssignment`和`visitBinary`的相似性，在namer.py里面先仿照`visitBinary`函数修改`visitAssignment`函数

```
    expr.lhs.accept(self,ctx)
    expr.rhs.accept(self,ctx)
```

来先对赋值语句的左值和右值进行访问

    - 在处理声明语句时，在`visitDeclaration`中增加如下：

```
    if not ctx.lookup(decl.ident.value):
        var=VarSymbol(decl.ident.value,decl.ident.type)
        ctx.declare(var)
        decl.setattr('symbol',var)
        if decl.init_expr:
            decl.init_expr.accept(self,ctx)
```

我们先看Declaration类的组成，有`var_t: TypeLiteral,ident: Identifier,init_expr: Optional[Expression]`这样的成员，其中var_t是类型名，Identifier是变量名，init_expr是可能存在的初始赋值

按照提示，我们注意到scope.py里面的lookup函数，该函数查找该作用域内是否定义该symbol，返回一个Option[Symbol],其中如果没有找到该symbol的话会返回None，所以我们以该函数返回值作为条件，如果没有定义过的symbol，则创建一个VarSymbol并将其加入当前作用域，利用ctx.declare函数，我们可以将其放入该作用域的符号表中，接着我们按照提示将decl的symbol属性指向我们创建的VarSymbol，最后，如果存在初始赋值，则将初始赋值语句的值赋给该VarSymbol，即完成语义分析

- 接着我们处理变量符号，按照提示，该类型是Identifier，所以我们在namer.py的`visitIdentifier`里面增加如下代码：

```
  if not ctx.lookup(ident.value):
      raise DecafUndefinedVarError(ident.value)
  ident.setattr("symbol",ctx.lookup(ident.value))
```

因为我们应该在在visitIdentifier的时候已经将符号表中的符号建立了ident.symbol，所以这里我们先检查该符号表中是否存在该变量，如果不存在，则抛出异常，否则将查到的该变量的Symbol类型赋给该symbol属性

- 中间代码生成
  - 在visitIdentifier中，我们增加如下代码：

```
ident.setattr("val", ident.getattr("symbol").temp)
```

按照提示，我们把ident的symbol属性分配到的临时变量赋给val属性

- 在visitDeclaration中，我们增加如下代码：

```
  decl.getattr("symbol").temp=mv.freshTemp()
  if decl.init_expr:
      decl.init_expr.accept(self, mv)

mv.visitAssignment(decl.getattr("symbol").temp,decl.init_expr.getattr("val")
)
```

首先注意到getattr("symbol")的返回值为Symbol，该类有如下组成`self.temp,self.isGlobal,self.initValue`，注意到为每一个变量符号分配了一个临时的tmp符号来方便中间码生成，于是我们新建一个temp为symbol的temp，然后如果存在初始赋值，先用accept函数求值，然后用visitAssignment函数将该temp的值赋给symbol的temp，这样就完成了中间码生成

- 在visitAssignment中，我们增加如下代码：

```
      expr.lhs.accept(self,mv)
      expr.rhs.accept(self,mv)
```

```
expr.setattr("val",mv.visitAssignment(expr.lhs.getattr("val"),expr.rhs.getat
tr("val")))
```

因为赋值语句可以视为一种特殊的二元运算，所以仿照`visitBinary`函数，先对赋值语句的左值右值进行访问，最后将该expression的val属性设置为vsitAssignment函数的返回值，这样就完成了中间码生成

- 后端代码生成
    - 通过试验将C代码编译成为riscv汇编，可知赋值语句没有用新的riscv指令，而是还是用mv指令实现，因此，我们在riscvasmemitter.py的visitAssign函数中增加如下：

```
self.seq.append(Riscv.Move(instr.dst,instr.src))
```

即直接将src的值赋给dst，这样就完成了后端代码生成