STAGE-4：函数和全局变量

计04 何秉翔 2020010944

1. 实验内容

1.1 step-9：函数

1.1.1 语法分析

首先我们在前端部分新增语法节点以及对应的语法解析过程，主要修改的是 `ply_parser.py`。

- 首先 `program` 包含多个函数而不再只是一个：

```
program : program function
        | empty
```

- 然后修改函数的产生式，增加参数列表，并区分函数的声明和函数的定义：

```
function : function_def
        | function_decl
# 函数声明
function_decl : type Identifier LParen parameter_list RParen Semi
# 函数定义
function_def : type Identifier LParen parameter_list RParen LBrace block RBrace
```

- 接着新增参数列表的产生式，由于函数声明可能有以下两种形式：

```
int fun(int, int);
int fun(int x, int y);
```

我们对 `type` 和 `identifier` 统一成 `type_identifier_union` 进行解析

```
parameter_list : parameter_list Comma type_identifier_union
        | type_identifier_union
        | empty

type_identifier_union : type
        | type Identifier
```

- 我们再解析函数调用：

```
postfix : Identifier LParen expression_list RParen
```

- 其中，我们仿照参数列表，解析函数的表达式列表：

```
expression_list : expression_list Comma expression
        | expression
        | empty
```

- 下面我们给出函数参数列表构建的例子：

```python
def p_parameter_list(p):
    """
    parameter_list : parameter_list Comma type_identifier_union
        | type_identifier_union
    """
    if len(p) > 2:
        p[1].children.append(p[3])
        p[0] = p[1]
    else:
        p[0] = ParameterList()
        p[0].children.append(p[1])

def p_type_identifier_union(p):
    """
    type_identifier_union : type
        | type Identifier
    """
    if len(p) > 2:
        p[0] = Parameter(p[1], p[2])
    else:
        p[0] = Parameter(p[1])

def p_parameter_list_empty(p):
    """
    parameter_list : empty
    """
    p[0] = ParameterList()
```

1.1.2 语法树建立

然后我们根据 `ply_parser.py` 中新增的产生式来构建语法树节点：`ParameterList`、`Parameter`、`Function`、`Call`、`ExpressionList`。

下面以参数列表的语法树节点作为示例：

```python
class ParameterList(ListNode["Parameter"]):
    """
    AST node that represents a parameter list
    """
    def __init__(self, *parameter: Parameter) -> None:
        super().__init__("parameterList", list(parameter))

    def __len__(self) -> int:
        return len(self.children)

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

class Parameter(Node):
    """
    AST node that represents a parameter
    """
    def __init__(self, var_t: TypeLiteral, ident: Optional[Identifier] = None,) ->
None:
        super().__init__("parameter")
        self.var_t = var_t
        self.ident = ident or NULL

    def __getitem__(self, key: int) -> Node:
        return (
```

```
            self.var_t,
            self.ident,
        )[key]

    def __len__(self) -> int:
        return 2

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

到此为止前端部分已经实现完毕，通过 `--parse` 应该能得到对应的语法树

1.1.3 语义分析 & 符号表构建

在这部分，我们主要关注 `namer.py` 进行符号表的构建以及语义的检查。我们首先在抽象的基类 `Visitor` 中加上关于刚建立的五个语法树节点对应的 `visit` 函数，接着在 `namer.py` 中考虑其具体实现。同样以参数列表的构建来举例：

```
# visitor.py
def visitParameterList(self, that: ParameterList, ctx: T) ->
Optional[Sequence[Optional[U]]]:
    return self.visitOther(that, ctx)
```

在 `namer.py` 中，我们按照从上到下的顺序来分析，首先是 `visitProgram`，在其中我们主要完成以下几件事：

- 遍历所有 `program` 的 `children`，即所有函数，去调用对应的 `visitFunction`。
- 通过全局符号表 `ctx.globalscope.symbols` 检查函数符号是否都已经被定义，我们为函数符号 `funcSymbol` 增加 `isDefined` 属性用于此处判断

因为函数声明和定义的位置可能在调用之前，也可能在调用之后，因此我们在所有函数都 `visit` 过后再统一检查是否未定义。

接下来我们考虑 `visitFunction`，在其中我们主要完成以下几件事：

- 若还没被声明，则声明
- 若被声明了，看是否被定义。若被定义了，则报重复定义的异常，若没被定义，则检查声明和定义的参数个数是否对应，由于参数类型都是 `int`，在此我们不做检查。
- 为函数设置 `symbol` `attribute`，用于之后三地址码生成。

由此，我们细分为如下的步骤：

```
"""
1. whether the func with the same name has been declared
    1.1 If yes, get the funcSymbol, whether has been defined
        1.1.1 If yes, raise
        1.1.2 If not, whether the func.body exists
            1.1.2.1 If yes, check params, if not the same, raise, else going on
            1.1.2.2 If not, raise DecafDeclConflictError
2. build a new FuncSymbol, and put it into the global scope, and open a local
scope
3. Set the 'symbol' attribute of func. (has been declared)
4. visit the parameter list and the ident
5. check whether func.body exists
    5.1 If yes, visit it, and set 'isDefined' attr to True
    5.2 If not, set 'isDefined' attr to False
6. close the scope
"""
```

最终的代码如下：

```
def visitFunction(self, func: Function, ctx: ScopeStack) -> None:
    if ctx.globalscope.containsKey(func.ident.value): # has been declared
        funcSymbol = ctx.lookup(func.ident.value)
        if funcSymbol.isDefined: # has been defined
            raise DecafDeclConflictError(func.ident.value)
        else: # has not been defined
            if func.body != NULL:
                # check lens
                if funcSymbol.parameterNum != len(func.params):
                    raise DecafDeclConflictError(func.ident.value)
                else:
                    # TODO: check types?
                    ctx.open(Scope(ScopeKind.LOCAL))
                    func.params.setattr('funcSymbol', funcSymbol)
                    func.params.accept(self, ctx)
                    for child in func.body:
                        child.accept(self, ctx)
                    funcSymbol.isDefined = True
                    ctx.close()
            else:
                raise DecafDeclConflictError(func.ident.value)
    else: # not been declared
        funcSymbol = FuncSymbol(func.ident.value, func.ret_t.type,
ctx.globalscope)
        ctx.declareGlobal(funcSymbol)
        ctx.open(Scope(ScopeKind.LOCAL))
        func.setattr('funcSymbol', funcSymbol)
        func.ident.accept(self, ctx)
        func.ident.setattr('funcSymbol', funcSymbol)
        func.params.setattr('funcSymbol', funcSymbol)
        func.params.accept(self, ctx)
        if func.body != NULL: # definition
            for child in func.body:
                child.accept(self, ctx)
            funcSymbol.isDefined = True
        else: # declaration
            funcSymbol.isDefined = False
        ctx.close()
```

对于函数定义或声明时的参数列表，以及调用时的表达式列表，这两者基本一致，我们以参数列表举例。

在参数列表的 `visit` 中，我们对每一个参数调用对应的 `visit` 函数，在 `visitParameter` 中我们主要完成以下几件事：

- 若 `ident` 不存在，对应函数声明的情况：`int fun(int, int)`，则直接调用 `addParaType` 增加参数类型即可
- 若 `ident` 存在，则为该 `ident` 进行声明，仿照 `visitDeclaration` 进行声明。

```
def visitParameter(self, param: Parameter, ctx: ScopeStack) -> None:
    """
    0. if ident not exists, just add type to the func symbol
    1. Use ctx.findConflict to find if a param 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 param.
    4. visit the ident of param
```

```
    5. add the type to the func symbol
    """
    if param.ident != NULL:
        symbol = ctx.findConflict(param.ident.value)
        if symbol == None: # has not been declared
            symbol = VarSymbol(param.ident.value, param.var_t.type)
            ctx.declare(symbol)
        else:
            raise DecafDeclConflictError(param.ident.value)
        param.setattr('symbol', symbol)
        param.ident.accept(self, ctx)
    if param.getattr('add'):
        param.getattr("funcSymbol").addParaType(
```

在函数调用 `visitCall` 时，我们完成如下事情：

- 新开局部作用域
- 检查函数符号是否被定义
- 检查参数列表长度和表达式列表长度是否匹配
- 对表达式列表进行 `visit`

1.1.4 TAC 生成

该部分我们主要关注 `tacgen.py`，在其中完成对新增的五个语法树节点对应的 `visit` 函数。

我们同理按照自顶向下的顺序进行一一分析：

首先是 `transform`，我们需要对每个定义的函数（包括 `main` 函数）进行 `visit`，跳过函数的声明部分。

```
def transform(self, program: Program) -> TACProg:
    pw = ProgramWriter(list(program.functions().keys()))
    for func in program.children:
        if func.body == NULL:
            continue
        mv = pw.visitFunc(func.ident.value, len(func.params))
        func.accept(self, mv)
        mv.visitEnd()
    return pw.visitEnd()
```

对于 `visitFunction`，我们递归对函数的参数列表以及函数体进行 `visit`。

对参数列表的访问只需递归对每一个参数进行访问即可，而对于 `visitParameter`，我们主要为该参数符号分配一个中间寄存器，该参数符号通过 `getattr` 拿到，然后调用 `freshTemp()` 进行分配。

对函数调用的访问只需对函数调用的表达式列表进行访问即可，然后把表达式列表的返回值设置为 `val` 属性的值。

对表达式列表的访问，即 `visitExpressionList`，我们主要完成以下几件事情：

- 对每一个表达式进行访问，得到表达式的值。
- 将表达式的值进行传参。
    - 构造 `TACinstr` 节点 `Call` 和 `Param`，用于生成三地址码（`tacinstr.py`）

```python
# 以 Param 举例
class Param(TACInstr):
    def __init__(self, src: Temp) -> None:
        super().__init__(InstrKind.SEQ, [], [src], None)
        self.src = src

    def __str__(self) -> str:
        return "PARAM %s" % self.src

    def accept(self, v: TACVisitor) -> None:
        v.visitParameter(self)
```

- 在 `mv: FuncVisitor` 中我们新增 `visitParameter` 和 `visitCall` 方法（`funcvisitor.py`）

```python
def visitParameter(self, param: Temp) -> None:
    self.func.add(Param(param))

def visitCall(self, dst: Temp, target: Label) -> None:
    self.func.add(Call(dst, target))
```

- 调用 `mv.visitParameter` 进行传参
- 特别注意的是，此处 `visitExpression` 和 `mv.visitParameter` 的顺序会影响 `loc.liveIn` 和 `loc.liveOut` 的结果，即会影响后端寄存器分配，我们先全部 `accept`，再全部 `visit`，控制变量的生存周期，以便于后端寄存器分配，最后结果应该具有如下形式：

```
_T0 = 0
...
_T8 = 9
_T9 = 10
_T10 = 11
PARAM _T0
...
PARAM _T8
PARAM _T9
PARAM _T10
```

- 处理函数的 `label` 以及返回值寄存器，调用 `mv.visitCall` 进行函数调用的 `TAC` 生成
- 最后为表达式列表的 `val` 赋值。

到此为止中端部分已经实现完毕，通过 `--tac` 应该能得到对应的 TAC

1.1.5 后端

在后端部分，入口在 `asm.py`，与该 `step` 相关的有两条指令：`self.emitter.selectInstr` 和 `self.regAlloc.accept`。于是我们分为两个阶段，第一个阶段是访问 `TAC`，生成" `riscv` 中间指令"，即该部分的指令大部分已经是 `riscv`，但对于函数传参和调用还依赖于之后的寄存器分配，因此还不全是可以直接跑的 `riscv` 汇编。第二个阶段是对前面的 `riscv` 中间指令进行寄存器分配、传参以及函数调用的最终实现。

我们主要关注的是 `bruteregalloc.py` 以及 `riscvasmemitter.py` 两个文件，首先关注后者。

在 `RiscvInstrSelector` 中，我们增加 `visitParameter` 和 `visitCall` 方法，对应在 `TACvisitor` 的基类中增加基类方法。

在 `visitParameter` 中，我们向第一个阶段生成的中间指令 `self.seq` 中添加临时的传参指令 `Param`，统一用栈进行传参，到后续第二个阶段我们再对参数的个数进行分别传参，即前八个参数直接用寄存器，后面的参数才用栈传。

```
def visitParameter(self, instr: Param) -> None:
    self.seq.append(Riscv.Param(instr.src, (self.offset << 2)))
    self.offset += 1 # update the offset
```

在 `visitCall` 中，我们向第一个阶段生成的中间指令 `self.seq` 中添加函数调用指令 `Call` 和返回值的移动 `Move`

```
def visitCall(self, instr: Call) -> None:
    self.seq.append(Riscv.Call(instr.dst, instr.target))
    self.seq.append(Riscv.Move(instr.dst, Riscv.A0))
    self.offset = 0
```

其中 `Riscv.Call` 和 `Riscv.Move` 如下：

```
class Param(TACInstr):
    def __init__(self, src: Temp, offset: int) -> None:
        super().__init__(InstrKind.SEQ, [], [src], None)
        self.offset = offset

    def __str__(self) -> str:
        return "sw " + Riscv.FMT_OFFSET.format(str(self.srcs[0]), self.offset,
str(Riscv.SP))

class Call(TACInstr):
    def __init__(self, dst: Temp, target: Label) -> None:
        super().__init__(InstrKind.SEQ, [dst], [], target)
        self.target = target

    def __str__(self) -> str:
        return "call " + str(self.target.name)
```

接下来我们观察 `RiscvSubroutineEmitter` 里的 `emitEnd` 方法，该方法是所有汇编指令最终生成的地方，除了 `start of prologue` 等初始化的地方，主要的部分位于 `start of body` 和 `end of body` 之间，对缓冲的 `buf` 进行输出，而这些 `buf`，就是第一个阶段生成的中间指令经过第二个阶段的转化之后最终的 `riscv` 汇编。

```
self.printer.printComment("start of body")
for instr in self.buf:
    self.printer.printInstr(instr)
self.printer.printComment("end of body")
```

接下来我们主要考虑第二个阶段，主要修改的是 `allocForLoc` 方法。

首先是对寄存器进行分配与绑定，然后我们要就指令的类型做一个判断，特别地，对于 `Riscv.Param` 和 `Riscv.Call` 这两条，需要做额外的处理。

对于 `Riscv.Param` 类型指令，我们主要做以下两件事：

- 对不超过 8 个的参数传到对应的传参寄存器内，调用 `Riscv.Move` 指令。由于有可能出现 `move` 的 `src` 寄存器即为前面传参时已经被分配的寄存器的情况，我们对传参的寄存器设置属性 `args_occupied`，在 `allocRegFor` 中对这个属性做判断，若 `True` 则不会分配。
- 对超过 8 个的参数存到栈上，由于存到栈上需要从右向左压栈，我们需要知道参数的个数，因此在此处对这部分参数进行缓存，直到 `Call` 指令到来时开始调用。
- 以上过程实际上是在函数发生调用的时候才执行，因为我们还不知道后面函数的参数情况，有可能后面被分配的寄存器里含有参数寄存器，这样会发生覆盖冲突：

```
mv a1, t1
...
mv a7, a1
```

对于 `Riscv.Call` 类型指令，我们在 `call` **之前**主要做以下四件事：

- 将正在使用的 `caller_saved` 寄存器存到栈上
  - 若 `caller_saved` 中含有 `ArgRegs`，说明分配寄存器时分配到了参数寄存器，我们要做个记录
  - 在最后 `load` 的时候不能用 `mv`，否则会发生前面所说的覆盖冲突问题，对此我们选择将其存到栈上，需要传参时再 `load` 回来

```
for reg in self.emitter.callerSaveRegs:
    if reg.isUsed():
        self.savedRegs.append(reg)
        subEmitter.emitStoreToStack(reg)
        if reg in Riscv.ArgRegs:
            self.stored_temps.append(reg.temp)
```

  - 我们用一个 `stored_temps` 变量记下来这些寄存器
- 将前面缓存的超过 8 个的参数按照从右往左的顺序压栈，并设置好对应 `sp` 指针移动位置。
  - 在此我们先往下拉 `sp`，为栈上参数分配空间
  - 结束时我们再往上拉 `sp`，为之后从栈上把前面存储的存在覆盖问题的寄存器的值拿回来

```
if len(self.params) > 8:
    subEmitter.emitNative(Riscv.SPAdd(-4 * (len(self.params) - 8)))
    for i, srcReg in enumerate(self.params[8:]):
        subEmitter.emitNative(Riscv.NativeStoreWord(srcReg, Riscv.SP, 4 * i))
    # temporarily add back the sp
    subEmitter.emitNative(Riscv.SPAdd(4 * (len(self.params) - 8)))
```

- 处理前 8 个参数，并设置后 `sp` 指针移动位置。
  - 若来自参数寄存器，则从缓存的栈上读取，在这里我们调用 `emitLoadFromStack`，结合前面缓存的 `stored_temps`，读取栈上值
  - 否则直接用 `mv` 即可

```
for i, srcReg in enumerate(self.params):
    if i < 8:
        argReg = Riscv.ArgRegs[i]
        if srcReg not in Riscv.ArgRegs:
            subEmitter.emitNative(Riscv.Move(argReg, srcReg)) # mv a0, t0
        else: # load from stack
            subEmitter.emitLoadFromStack(argReg,
self.stored_temps[self.cur_cnt])
            self.cur_cnt += 1
```

- 最后如果前面暂时往上拉栈，此时重新往下拉栈：

```
if len(self.params) > 8:
    # sub back the sp
    subEmitter.emitNative(Riscv.SPAdd(-4 * (len(self.params) - 8)))
```

对于 `Riscv.Call` 类型指令，我们在 `call` **之后**主要做以下两件事：

- 将存到栈上的 `caller_saved` 寄存器存到对应寄存器内，特别地，我们不对 `a0` 进行恢复，防止覆盖返回值，在分配寄存器的时候，控制 `a0` 不会被分配到

- 恢复 `sp` 指针

```
if isinstance(instr, Riscv.Call):
    if len(self.params) > 8:
        subEmitter.emitNative(Riscv.SPAdd(4 * (len(self.params) - 8)))
    self.cur_cnt = 0
    self.params.clear()
    self.stored_temps.clear()
    for reg in self.savedRegs:
        if reg != Riscv.A0:
            subEmitter.emitLoadFromStack(reg, reg.temp)
```

接着，我们在 `emitLoadFromStack` 时做一些调整，对于没出现在 `offset` 中的 `temp` 寄存器，即应该从对应的参数寄存器或者栈上读取：

```
if src.index not in self.offsets:
    if src.index < 8:
        self.buf.append(
            Riscv.Move(dst, Riscv.ArgRegs[src.index])
        )
    else:
        self.buf.append(
            Riscv.NativeLoadWord(dst, Riscv.SP, 4 * (src.index - 8) +
self.nextLocalOffset)
        )
```

最后，我们在函数的调用开始前和结束前，调整 `sp` 指针，在 `init` 中预留了 `RA` 的位置，我们直接放在 `callee_saved` 寄存器之后：

```
# 开始
self.printer.printInstr(Riscv.NativeStoreWord(Riscv.RA, Riscv.SP, 4 *
len(Riscv.CalleeSaved)))
# 结束
self.printer.printInstr(Riscv.NativeLoadWord(Riscv.RA, Riscv.SP, 4 *
len(Riscv.CalleeSaved)))
```

到此为止后端部分已经实现完毕，通过 `--riscv` 应该能得到对应的 `riscv` 汇编

1.2 step-10: 全局变量

我们从前端到后端依次进行分析。

1.2.1 前端

首先，我们根据语法的改变，修改 `ply_parser.py` 文件，使得 `program` 的产生式不再仅仅生成函数，还可以生成全局变量。

```
program : program function
    | program declaration Semi
```

然后对于语法树节点的更改，我们只需把 `Program` 节点的 `children` 改为 `Union[Declaration, Function]` 即可，借用 `Declaration` 的节点即可。

1.2.2 中端

对于符号表的建立，我们主要关注 `namer.py`，在 `visitProgram` 入口处，我们对根节点 `Program` 的孩子分类讨论，如果是 `Function` 还按照之前那样分析，而对于 `Declaration` 类型的，说明是全局变量，我们为其增加一个 `global` 属性，设置为 `True`，然后借用 `visitDeclaration` 即可，接下来我们稍微修改一下 `visitDeclaration`：

```
# 若没找到符号
if decl.getattr('global'):
    symbol.isGlobal = True
    ctx.declareGlobal(symbol)
# 若找到了符号
 if decl.getattr('global'):
    raise DecafGlobalVarDefinedTwiceError(decl.ident.value)
```

即对全局变量做检查，并且在栈底全局作用域进行声明，设置符号的 `isGlobal` 属性为 `True`

接下来我们看 `TAC` 的生成，主要关注 `tacgen.py`，首先在入口方法 `transform` 处，我们要对根节点 `Program` 的孩子分类讨论，如果是 `Function` 照常，而对于 `Declaration` 类型的，说明是全局变量。我们需要做的是在之后函数的 `visit` 时，若 `visit` 到了一个全局变量，我们能够找到它，而关于函数的 `visitor` 是 `mv`，其是 `pw: ProgramWritter` 初始化的，因此我们考虑在 `pw` 中设置一个记录全局变量的属性 `globalVars`，在 `transform` 中，我们记录每一个出现过的全局变量。

因此，首先考虑记录全局变量的 `TAC` 节点，因此我们在 `tacinstr.py` 中增加一个 `GlobalVar`：

```
class GlobalVar(TACInstr):
    def __init__(self, symbol: str, init_flag: bool, init_value: int = 0) -> None:
        super().__init__(InstrKind.SEQ, [], [], None)
        self.symbol = symbol
        self.init_value = init_value
        self.init_flag = init_flag # 是否含有初值

    def __str__(self) -> str:
        if self.init_flag:
            return "global variable %s = %d" % (self.symbol, self.init_value)
        return "global variable %s" % self.symbol

    def accept(self, v: TACVisitor) -> None:
        v.visitGlobalVar(self)
```

然后在 `transform` 中记录的时候，通过 `init_expr` 的有无分别创建 `GlobalVar` 的节点，并记录在 `ProgramWritter` 中。同时我们目前只支持整数字面量的初始化，因此多判断一步即可：

```
if not isinstance(child.init_expr, IntLiteral):
    raise DecafGlobalVarBadInitValueError(child.ident.value)
```

最后在 `pw.visitEnd` 中打印 `TAC` 指令时，我们为全局变量定义也打印对应的 `TAC`，因此我们在 `TACProg` 初始化时，传入 `ProgramWritter` 的 `globalVars` 变量：

```
# programwritter.py
def visitEnd(self) -> TACProg:
    return TACProg(self.ctx.funcs, self.globalVars)
```

最后在 `TACProg` 中 `printTo` 进行修改：

```python
# tacprog.py
def printTo(self) -> None:
    for globalVar in self.globalVars:
        print(str(globalVar))
    for func in self.funcs:
        func.printTo()
```

接下来在函数中访问到全局变量时，无论是对全局变量的读还是写，我们都需要新增 `TAC` 指令来表示。按照实验指导书，我们将全局变量的读分为以下两步：

```
_T0 = load global symbol x # 获取基址
load _T1, offset(_T0) # 得到全局变量的值：load _T0, 0(_T1): load the value store in (0 + _T1) to _T0
```

同理，写分为以下两步：

```
_T0 = 4 # 待写入的值
_T1 = load global symbol x # 获取基址
store _T0, 0(_T1) # 更新全局变量的值：store _T0, 0(_T1): store _T0 to the addr of (0 + _T1)
```

因此，我们设计三条 `TAC` 指令：

```python
class LoadGlobalVarSymbol(TACInstr):
    def __init__(self, symbol: str, dst: Temp) -> None:
        super().__init__(InstrKind.SEQ, [dst], [], None)
        self.symbol = symbol

    def __str__(self) -> str:
        return "%s = load global symbol %s" % (self.dsts[0], self.symbol)

    def accept(self, v: TACVisitor) -> None:
        v.visitLoadGlobalVarSymbol(self)

class LoadGlobalVarAddr(TACInstr):
    def __init__(self, src: Temp, dst: Temp, offset: int) -> None:
        super().__init__(InstrKind.SEQ, [dst], [src], None)
        self.offset = offset

    def __str__(self) -> str: #  load _T0, 4(_T1): load the value store in (4 + _T1) to _T0
        return "load %s, %d(%s)" % (self.dsts[0], self.offset, self.srcs[0])

    def accept(self, v: TACVisitor) -> None:
        v.visitLoadGlobalVarAddr(self)
# ... StoreGlobalVarAddr 同理
```

接着在 `mv: FuncVisitor` 中增加对应的 `visit` 方法，供 `tacgen.py` 生成 `TAC` 时调用：

```python
def visitLoadGlobalVarAddr(self, dst: Temp, src: Temp, offset: int) -> None:
    self.func.add(LoadGlobalVarAddr(src, dst, offset))

def visitStoreGlobalVarAddr(self, dst: Temp, src: Temp, offset: int) -> None:
    self.func.add(StoreGlobalVarAddr(src, dst, offset))

def visitLoadGlobalVarSymbol(self, dst: Temp, symbol: str) -> None:
    self.func.add(LoadGlobalVarSymbol(symbol, dst))
```

接着在 `tacgen.py` 中，我们对于全局变量的读和写分别调整，分别对应 `visitIdentifier` 读取全局变量和 `visitAssignment` 写入全局变量：

在 `visitIdentifier` 中，我们首先通过 `symbol` 属性取到语义分析阶段设置的变量符号，若不是全局变量，则照常进行，若是全局变量符号，则执行以下操作：

- 为基址分配 `TAC` 寄存器 `addrTemp`
- `load` 基址
- 为全局变量分配 `TAC` 寄存器 `valueTemp`
- `load` 全局变量值
- 设置 `val` 属性

```python
# visitIdentifier
addrTemp = mv.freshTemp() # the base addr temp of global var
mv.visitLoadGlobalVarSymbol(addrTemp, symbol.name)
valueTemp = mv.freshTemp() # the value temp of global var
mv.visitLoadGlobalVarAddr(valueTemp, addrTemp, 0)
ident.setattr('val', valueTemp)
```

在 `visitAssignment` 中，我们同理对 `lhs` 判断是否是全局变量，若是，则执行以下操作：

- 为右侧表达式求值，为基址分配 `TAC` 寄存器 `addrTemp`
- `load` 基址
- 将右侧表达式的值存入全局变量

```python
# visitAssignment
expr.rhs.accept(self, mv)
addrTemp = mv.freshTemp() # the base addr temp of global var
mv.visitLoadGlobalVarSymbol(addrTemp, symbol.name)
mv.visitStoreGlobalVarAddr(addrTemp, expr.rhs.getattr("val"), 0)
```

1.2.3 后端

我们为 `TAC` 新增的三条指令增加对应的 `riscv` 指令：

```python
class LoadGlobalVarSymbol(TACInstr):
    def __init__(self, dst: Temp, symbol: str) -> None:
        super().__init__(InstrKind.SEQ, [dst], [], None)
        self.symbol = symbol

    def __str__(self) -> str:
        return "la " + Riscv.FMT2.format(str(self.dsts[0]), self.symbol)

class LoadGlobalVarAddr(TACInstr):
    def __init__(self, dst: Temp, src: Temp, offset: int) -> None:
        super().__init__(InstrKind.SEQ, [dst], [src], None)
        self.offset = offset

```

```
    def __str__(self) -> str:
        assert -2048 <= self.offset <= 2047  # Riscv imm [11:0]
        return "lw " + Riscv.FMT_OFFSET.format(str(self.dsts[0]), self.offset,
str(self.srcs[0]))
# ... StoreGlobalVarAddr 同理
```

然后在 `riscvasmemitter.py` 中增加对应的 `visit` 指令用于生成 `riscv` 指令：

```
def visitLoadGlobalVarSymbol(self, instr: LoadGlobalVarSymbol) -> None:
    self.seq.append(Riscv.LoadGlobalVarSymbol(instr.dsts[0], instr.symbol))

def visitLoadGlobalVarAddr(self, instr: LoadGlobalVarAddr) -> None:
    self.seq.append(Riscv.LoadGlobalVarAddr(instr.dsts[0], instr.srcs[0],
instr.offset))

def visitStoreGlobalVarAddr(self, instr: StoreGlobalVarAddr) -> None:
    self.seq.append(Riscv.StoreGlobalVarAddr(instr.dsts[0], instr.srcs[0],
instr.offset))
```

接下来，我们要考虑全局变量在 `data` 段或者 `bss` 段的声明，注意 `bss` 段要在 `.text` 段的前面，不然汇编执行会报错。因此我们考虑将 `RiscvAsmEmitter` 初始化 `__init__` 中的 `.text .global main` 等输出放在后面的一个方法 `emitText` 中，并新增一个方法 `emitGlobalVars` 用于输出全局变量的声明信息。这两个方法的调用放在生成汇编的入口 `asm.py` 中的 `transform` 里。

在 `emitGlobalVars` 中，我们先将所有的全局变量分类到 `dataGlobalVars` 和 `bssGlobalVars` 里，然后分别遍历，生成对应的 `riscv` 汇编：

```
def emitGlobalVars(self, globalVars: list[GlobalVar]):
    dataGlobalVars = []
    bssGlobalVars = []
    for globalVar in globalVars:
        if globalVar.init_flag:
            dataGlobalVars.append(globalVar)
        else:
            bssGlobalVars.append(globalVar)
    if len(dataGlobalVars):
        self.printer.println(".data")
    for globalVar in dataGlobalVars:
        self.printer.println(".global %s" % globalVar.symbol)
        self.printer.println("%s:" % globalVar.symbol)
        self.printer.println("    .word %d" % globalVar.init_value)

    if len(bssGlobalVars):
        self.printer.println(".bss")
    for globalVar in bssGlobalVars:
        self.printer.println(".global %s" % globalVar.symbol)
        self.printer.println("%s:" % globalVar.symbol)
        self.printer.println("    .space 4")
```

在 `emitText` 中，我们把 `__init__` 中输出的部分拿过来：

```
def emitText(self):
    self.printer.println(".text")
    self.printer.println(".global main")
    self.printer.println("")
```

至此，`step-10` 已经完成。

2. 思考题

2.1 step-9 思考题

问题：MiniDecaf 的函数调用时参数求值的顺序是未定义行为。试写出一段 MiniDecaf 代码，使得不同的参数求值顺序会导致不同的返回结果。

解答：

```
int func(int x, int y){
    return x + y;
}
int main(){
    int a = 0;
    int b = 1;
    return func(a = a + 1, a + b);
}
```

两种求值顺序分别计算出来 3 和 2。

2.2 step-10 思考题

问题：写出 `la v0, a` 这一 RiscV 伪指令可能会被转换成哪些 RiscV 指令的组合（说出两种可能即可）。

解答：

```
# 1
auipc v0, delta[31:12] + delta[11]
addi v0, v0, delta[11:0]
# 2
auipc v0, delta[31:12] + delta[11]
lw v0, v0, delta[11:0]

# delta = (label a) - pc
```