Stage6 实验报告

姓名：丁俊辉

学号：2021011145

step10

实验过程

在 `tree.py` 中，我更改了 `Program` 的定义：

```
class Program(ListNode[Union["Function", "Declaration"]]):
    ...

    def globals(self) -> list[Declaration]:
        return [decl for decl in self if isinstance(decl, Declaration)]


    ...
```

首先，使其继承自 `ListNode[Union["Function", "Declaration"]]`，这样 children 的类型就可以是 `Function` 或者 `Declaration`；然后，我添加了一个 `globals` 属性，便于找到所有的 `Declaration` 子节点。

在 `ply_parser.py` 中，我同步更新了 `Program` 的 productions：

```
def p_program(p):
    """
    program : program function
    program : program declaration Semi
    """
    p[1].children.append(p[2])
    p[0] = p[1]
```

至此，可以正确生成全局变量的 AST。

我希望记录符号表中的全局变量符号，为此我先更改了 `VarSymbol` 类型的定义：

```
class VarSymbol(Symbol):
    def __init__(self, name: str, type: DecafType, isGlobal: bool = False) ->
None:
        super().__init__(name, type)
        self.temp: Temp
        self.isGlobal = isGlobal
        self.initValue = None

    def __str__(self) -> str:
        return "variable %s : %s" % (self.name, str(self.type))

    # To set the initial value of a variable symbol (used for global variable).
    def setInitValue(self, value: Optional[int]) -> None:
        self.initValue = value
```

这里，我将 `initValue` 的类型从 `int` 变为 `Optional[int]`，我认为这样更易表现出没有初始化的情况。

在 `namer.py` 中，我修改了 `visitDeclaration`，加入了对于全局变量和其初始值的记录：

```
class Namer(Visitor[ScopeStack, None]):
    ...

    def visitDeclaration(self, decl: Declaration, ctx: ScopeStack) -> None:
        if ctx.top().isGlobalScope():
            symbol = VarSymbol(decl.ident.value, decl.var_t.type, True)
            if decl.init_expr is not NULL:
                if isinstance(decl.init_expr, IntLiteral):
                    symbol.setInitValue(decl.init_expr.value)
                else:
                    # other types of initial value are not supported yet
                    raise NotImplementedError
            else:
                symbol.setInitValue(None)
            if ctx.globalscope.lookup(decl.ident.value) is not None:
                sameNamedSymbol = ctx.globalscope.lookup(decl.ident.value)
                if isinstance(sameNamedSymbol, FuncSymbol):
                    # function and variable with the same name
                    raise DecafDeclConflictError(decl.ident.name)
                elif isinstance(sameNamedSymbol, VarSymbol):
                    if sameNamedSymbol.initValue is not None:
                        if symbol.initValue is not None:
                            # redeclaration of a global variable
                            raise DecafDeclConflictError(decl.ident.name)
                        else:
                            # use the initial value of the previous declaration
                            symbol.setInitValue(sameNamedSymbol.initValue)
                else:
                    # other types of symbols are not supported yet
                    raise NotImplementedError
            ctx.globalscope.declare(symbol)
            decl.setattr("symbol", symbol)
        else:
            symbol = VarSymbol(decl.ident.value, decl.var_t.type, False)
            if ctx.top().lookup(decl.ident.value) is not None:
                raise DecafDeclConflictError(decl.ident.name)
            ctx.top().declare(symbol)
            decl.setattr("symbol", symbol)
            if decl.init_expr is not NULL:
                decl.init_expr.accept(self, ctx)
```

这里主要是加入对于全局变量声明的处理。我将全局变量符号的 `isGlobal` 设为 `True`，并且给有初始值的全局变量附上了初始值。

此外，我按照实验文档的要求，对全局变量的符号表做了检查：对于重复定义的全局变量，会报错；对于全局变量和函数的重名，也会报错（我是参照 C 语言的报错信息写的）；对于全局变量的声明和定义冲突，我会让定义覆盖声明。

在 `tacinstr.py` 中，我添加了一下几种 `TACInstr`：

```
# Loading the address of a symbol.
class LoadSymbol(TACInstr):
    def __init__(self, dst: Temp, symbol: str) -> None:
        super().__init__(InstrKind.SEQ, [dst], [], None)
        self.dst = dst
        self.symbol = symbol

    def __str__(self) -> str:
        return "%s = LOAD_SYMBOL %s" % (self.dst, self.symbol)

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


# Loading from memory at an offset from a base address.
class Load(TACInstr):
    def __init__(self, dst: Temp, src: Temp, offset: int) -> None:
        super().__init__(InstrKind.SEQ, [dst], [src], None)
        self.dst = dst
        self.src = src
        self.offset = offset

    def __str__(self) -> str:
        return "%s = LOAD %s, %d" % (self.dst, self.src, self.offset)

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


# Storing to memory at an offset from a base address.
class Store(TACInstr):
    def __init__(self, dst: Temp, offset: int, src: Temp) -> None:
        super().__init__(InstrKind.SEQ, [], [dst, src], None)
        self.dst = dst
        self.offset = offset
        self.src = src

    def __str__(self) -> str:
        return "STORE %s, %d, %s" % (self.dst, self.offset, self.src)

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

目的是实现 TAC 中的全局变量赋值。

在 `tacprog.py` 中，我改了 `TACProg` 类的定义：

```
class TACProg:
    def __init__(self, funcs: list[TACFunc], globals: list[VarSymbol]) -> None:
        self.funcs = funcs
        self.globals = globals

    def printTo(self) -> None:
        for func in self.funcs:
            func.printTo()
```

TAC 码的生成并不需要描述全局变量，我加入 `globals` 属性只是为了将符号表中的全局变量传递到下一阶段。

在 `tacgen.py` 中，我加入了全局符号的传递和全局变量赋值的处理：

```
class TACFuncEmitter(TACVisitor):

    ...

    def visitLoadSymbol(self, symbol: VarSymbol) -> Temp:
        temp = self.freshTemp()
        self.func.add(LoadSymbol(temp, symbol.name))
        return temp

    def visitLoadMem(self, src: Temp, offset: int) -> Temp:
        temp = self.freshTemp()
        self.func.add(Load(temp, src, offset))
        return temp

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


class TACGen(Visitor[TACFuncEmitter, None]):
    # Entry of this phase
    def transform(self, program: Program) -> TACProg:
        labelManager = LabelManager()
        tacFuncs = []
        for funcName, astFunc in program.functions().items():
            # in step9, you need to use real parameter count
            emitter = TACFuncEmitter(FuncLabel(funcName), len(astFunc.params),
labelManager)
            astFunc.params.accept(self, emitter)
            astFunc.body.accept(self, emitter)
            tacFuncs.append(emitter.visitEnd())
        globals: list[VarSymbol] = []
        for decl in program.globals():
            globals.append(decl.getattr("symbol"))
        return TACProg(tacFuncs, globals)

    ...

    def visitIdentifier(self, ident: Identifier, mv: TACFuncEmitter) -> None:
        symbol: VarSymbol = ident.getattr('symbol')
        if symbol.isGlobal:
            temp = mv.visitLoadSymbol(symbol)
            ident.setattr("val", mv.visitLoadMem(temp, 0))
        else:
            ident.setattr("val", symbol.temp)

    def visitAssignment(self, expr: Assignment, mv: TACFuncEmitter) -> None:
        expr.rhs.accept(self, mv)
        lhsSymbol: VarSymbol = expr.lhs.getattr("symbol")
        if lhsSymbol.isGlobal:
```

```
            mv.visitStoreMem(expr.rhs.getattr("val"),
mv.visitLoadSymbol(lhsSymbol), 0)
            expr.setattr("val", expr.rhs.getattr("val"))
        else:
            expr.lhs.accept(self, mv)
            expr.setattr("val",
mv.visitAssignment(expr.lhs.getattr("symbol").temp, expr.rhs.getattr("val")))
```

至此，可以正确地生成符号表和 TAC 码，并且将全局变量的符号保存起来传递。

在 `riscv.py` 中，我加了几个指令：

```
class Riscv:
    ...

    class LoadAddr(BackendInstr):
        def __init__(self, dst: Temp, label: Label) -> None:
            super().__init__(InstrKind.SEQ, [dst], [], None)
            self.label = label

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

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

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

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

        def __str__(self) -> str:
            return "sw " + Riscv.FMT_OFFSET.format(str(self.srcs[0]),
str(self.offset), str(self.srcs[1]))
```

这里的 `LoadWord` 和 `StoreWord` 主要方便处理全局变量。

在

```
class RiscvAsmEmitter():
    def __init__(
        self,
        allocatableRegs: list[Reg],
        callerSaveRegs: list[Reg],
        globals: list[VarSymbol],
    ):
        self.allocatableRegs = allocatableRegs
        self.callerSaveRegs = callerSaveRegs
        self.globals = globals
```

```
        self.printer = AsmCodePrinter()

        # the start of the asm code
        # int step10, you need to add the declaration of global var here

        # Add global variable declarations
        initialized_globals = [var for var in globals if var.initValue is not
None]
        uninitialized_globals = [var for var in globals if var.initValue is None]

        if len(initialized_globals) > 0:
            self.printer.println(".data")
            for var in initialized_globals:
                self.printer.println(f".globl {var.name}")
                self.printer.println(f"{var.name}:")
                self.printer.println(f"    .word {var.initValue}")
            self.printer.println("")

        if len(uninitialized_globals) > 0:
            self.printer.println(".bss")
            for var in uninitialized_globals:
                self.printer.println(f".globl {var.name}")
                self.printer.println(f"{var.name}:")
                self.printer.println("    .space 4")
            self.printer.println("")

        self.printer.println(".text")
        self.printer.println(".global main")
        self.printer.println("")

    ...

    class RiscvInstrSelector(TACVisitor):

        ...

        def visitLoadSymbol(self, instr: LoadSymbol) -> None:
            self.seq.append(Riscv.LoadAddr(instr.dst, instr.symbol))

        def visitLoad(self, instr: Load) -> None:
            self.seq.append(Riscv.LoadWord(instr.dst, instr.src, instr.offset))

        def visitStore(self, instr: Store) -> None:
            self.seq.append(Riscv.StoreWord(instr.src, instr.dst, instr.offset))
```

至此，可以正确生成全局变量的 ASM 代码。

思考题

1. 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）。

A:

如果 `a` 可以通过一个 12 位的立即数直接跳转（PC 相对寻址），可以使用以下指令：

```
auipc v0, offset_upper  # 将符号a的地址的高20位与PC的高20位相加，并将结果放入v0
addi v0, v0, offset_lower  # 将符号a的地址的低12位加到v0中
```

这里 `offset_upper` 是符号 `a` 的地址的高20位与当前指令地址的高20位之差的高20位，`offset_lower` 是符号 `a` 的地址相对于 `auipc` 指令地址的低12位。

如果符号 `a` 的地址不能通过一个 12 位的立即数直接跳转，可能需要通过两个寄存器间接完成：

```
lui t0, offset_upper  # 将符号a的地址的高20位加载到临时寄存器t0
addi t0, t0, offset_lower  # 将符号a的地址的低12位加到t0中
add v0, t0, zero  # 将t0的值（即a的地址）复制到v0
```

这里 `t0` 是一个临时寄存器，用于中间计算。

step11

实验过程

在 `tree.py` 中，我定义了两种新节点。`ArraySpecifier` 是数组类型，`IndexExpression` 是索引表达式：

```python
class ArraySpecifier(Identifier):
    """
    AST node of array identifier.
    """

    def __init__(self, base: Union[Identifier, ArraySpecifier], size: IntLiteral)
-> None:
        super().__init__(base.value)
        self.base = base
        self.size = size

    @property
    def ident(self) -> Identifier:
        if isinstance(self.base, ArraySpecifier):
            return self.base.ident
        return self

    @property
    def sizes(self) -> list[IntLiteral]:
        if isinstance(self.base, ArraySpecifier):
            return self.base.sizes + [self.size]
        return [self.size]

    @property
    def dim(self) -> int:
        if isinstance(self.base, ArraySpecifier):
            return self.base.dim + 1
        return 1

    def __getitem__(self, key: int) -> Node:
        return (self.base, self.size)[key]

    def __len__(self) -> int:
```

```
        return 2

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

    def __str__(self) -> str:
        return "array_specifier({}[{}])".format(
            self.base,
            self.size,
        )


class IndexExpression(Expression):
    """
    AST node of index expression.
    """

    def __init__(self, base: Union[Identifier, IndexExpression], index:
Expression) -> None:
        super().__init__("index_expr")
        self.base = base
        self.index = index

    @property
    def ident(self) -> Identifier:
        if isinstance(self.base, Identifier):
            return self.base
        return self.base.ident

    @property
    def indexes(self) -> list[Expression]:
        if isinstance(self.base, Identifier):
            return [self.index]
        return  self.base.indexes + [self.index]

    @property
    def dim(self) -> int:
        if isinstance(self.base, Identifier):
            return 1
        return self.base.dim + 1

    def __getitem__(self, key: int) -> Node:
        return (self.base, self.index)[key]

    def __len__(self) -> int:
        return 2

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

    def __str__(self) -> str:
        return "{}[{}]".format(
            self.base,
            self.index,
        )
```

这两个节点的定义我是参照 `ArrayType` 写的，可以递归地获得每一层信息。

此外，我还改了其他一些节点的定义：

```
class Assignment(Binary):

    def __init__(self, lhs: Union[Identifier, Unary], rhs: Expression) -> None:
        super().__init__(BinaryOp.Assign, lhs, rhs)
```

`Assignment` 的左值现在可以是 `Unary`。之所以是 `Unary`，我是完全照着实验指导书规定的产生式来写的，但是现在只支持 `IndexExpression`。

```
class Declaration(Node):

    def __init__(
        self,
        var_t: TypeLiteral,
        ident: Union[Identifier, ArraySpecifier],
        init_expr: Optional[Expression] = None,
    ) -> None:
        super().__init__("declaration")
        self.var_t = var_t
        self.ident = ident
        self.init_expr = init_expr or NULL
```

`Declaration` 定义也变了，现在可以声明数组。

在 `lex.py` 中，我添加了新的括号：

```
t_LBracket = "["
t_RBracket = "]"
```

在 `ply_parser.py` 中，我添加了新的产生式：

```
def p_array_specifier(p):
    """
    array_specifier : Identifier LBracket Integer RBracket
    array_specifier : array_specifier LBracket Integer RBracket
    """
    p[0] = ArraySpecifier(p[1], p[3])


def p_declaration(p):
    """
    declaration : type Identifier
    declaration : type array_specifier
    """
    p[0] = Declaration(p[1], p[2])


def p_declaration_init(p):
    """
    declaration : type Identifier Assign expression
    declaration : type array_specifier Assign expression
```

```
    """
    p[0] = Declaration(p[1], p[2], p[4])
```

至此，可以生成正确的 AST。

在 `namer.py` 中，我加入了数组的符号生成和类型检查。

```
class Namer(Visitor[ScopeStack, None]):
    ...

    def visitDeclaration(self, decl: Declaration, ctx: ScopeStack) -> None:
        if isinstance(decl.ident, ArraySpecifier):
            sizes = [size.value for size in decl.ident.sizes]
            dims = sizes[::-1]
            if any([size <= 0 for size in sizes]):
                raise DecafBadArraySizeError(decl.ident.base.value)
            symbolType = ArrayType.multidim(decl.var_t.type, *dims)
        elif isinstance(decl.ident, Identifier):
            symbolType = decl.var_t.type
        else:
            raise NotImplementedError
        ...

    def visitAssignment(self, expr: Assignment, ctx: ScopeStack) -> None:
        if isinstance(expr.lhs, Identifier):
            lhsSymbol = ctx.lookup(expr.lhs.value)
            if lhsSymbol is None:
                raise DecafUndefinedVarError(expr.lhs.value)
            if isinstance(lhsSymbol.type, ArrayType):
                raise DecafTypeMismatchError()
        ...

    def visitIndexExpr(self, expr: IndexExpression, ctx: ScopeStack) -> None:
        for index in expr.indexes:
            index.accept(self, ctx)
        symbol = ctx.lookup(expr.ident.value)
        if symbol is None:
            raise DecafUndefinedVarError(expr.ident.value)
        if not isinstance(symbol.type, ArrayType):
            raise DecafBadIndexError(expr.ident.value)
        else:
            arrayDim = symbol.type.dim
            indexDim = expr.dim
            if not arrayDim == indexDim:
                raise DecafBadIndexError(expr.ident.value)
        expr.setattr("symbol", symbol)

    def visitIdentifier(self, ident: Identifier, ctx: ScopeStack) -> None:
        """
        1. Use ctx.lookup to find the symbol corresponding to ident.
        2. If it has not been declared, raise a DecafUndefinedVarError.
        3. Set the 'symbol' attribute of ident.
        """
        symbol = ctx.lookup(ident.value)
        if symbol is None:
```

```
            raise DecafUndefinedVarError(ident.value)
        if isinstance(symbol.type, ArrayType):
            raise DecafTypeMismatchError()
        ident.setattr("symbol", symbol)

    def visitArraySpecifier(self, array: ArraySpecifier, ctx: ScopeStack) ->
None:
        symbol = ctx.lookup(array.base.value)
        if symbol is None:
            raise DecafUndefinedVarError(array.base.value)
        array.setattr("symbol", symbol)
```

这里只展示改动的部分。

在 `tacinstr.py` 中，加入了 ALLOC 指令：

```
# Allocating for an array.
class Alloc(TACInstr):
    def __init__(self, dst: Temp, size: int) -> None:
        super().__init__(InstrKind.SEQ, [dst], [], None)
        self.dst = dst
        self.size = size

    def __str__(self) -> str:
        return "%s = ALLOC %d" % (self.dst, self.size)

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

在 `tacgen.py` 中，加入 ALLOC 指令和 offset 的计算：

```
class TACFuncEmitter(TACVisitor):
    ...

    def visitAlloc(self, size: int) -> Temp:
        temp = self.freshTemp()
        self.func.add(Alloc(temp, size))
        return temp

    def visitDeclaration(self, decl: Declaration, mv: TACFuncEmitter) -> None:
        symbol: VarSymbol = decl.getattr("symbol")
        if isinstance(symbol.type, ArrayType):
            symbol.temp = mv.visitAlloc(symbol.type.size)
            if decl.init_expr is not NULL:
                raise NotImplementedError
        else:
            symbol.temp = mv.freshTemp()
            if decl.init_expr is not NULL:
                decl.init_expr.accept(self, mv)
                mv.visitAssignment(symbol.temp, decl.init_expr.getattr("val"))

    def visitAssignment(self, expr: Assignment, mv: TACFuncEmitter) -> None:
        expr.rhs.accept(self, mv)
        if isinstance(expr.lhs, IndexExpression):
            rhsTemp = expr.rhs.getattr("val")
```

```
            lhsSymbol: VarSymbol = expr.lhs.getattr("symbol")
            sizes = lhsSymbol.type.dims
            indexes = expr.lhs.indexes
            indexes[0].accept(self, mv)
            offset: Temp = indexes[0].getattr("val")
            for size, index in zip(sizes[1:], indexes[1:]):
                length: Temp = mv.visitLoad(size)
                offset: Temp = mv.visitBinary(tacop.TacBinaryOp.MUL, offset,
length)
                index.accept(self, mv)
                indexTemp: Temp = index.getattr("val")
                offset: Temp = mv.visitBinary(tacop.TacBinaryOp.ADD, offset,
indexTemp)
            if lhsSymbol.isGlobal:
                temp = mv.visitLoadSymbol(lhsSymbol)
            else:
                temp = lhsSymbol.temp
            sizeTemp = mv.visitLoad(lhsSymbol.type.full_indexed.size)
            offset: Temp = mv.visitBinary(tacop.TacBinaryOp.MUL, offset,
sizeTemp)
            addr: Temp = mv.visitBinary(tacop.TacBinaryOp.ADD, temp, offset)
            mv.visitStoreMem(rhsTemp, addr, 0)
            expr.setattr("val", rhsTemp)
        elif isinstance(expr.lhs, Identifier):
            lhsSymbol: VarSymbol = expr.lhs.getattr("symbol")
            if lhsSymbol.isGlobal:
                mv.visitStoreMem(expr.rhs.getattr("val"),
mv.visitLoadSymbol(lhsSymbol), 0)
                expr.setattr("val", expr.rhs.getattr("val"))
            else:
                expr.lhs.accept(self, mv)
                expr.setattr("val",
mv.visitAssignment(expr.lhs.getattr("symbol").temp, expr.rhs.getattr("val")))
        else:
            raise NotImplementedError

    def visitIndexExpr(self, expr: IndexExpression, mv: TACFuncEmitter) -> None:
        for index in expr.indexes:
            index.accept(self, mv)
        symbol: VarSymbol = expr.getattr("symbol")
        sizes = symbol.type.dims
        indexes = expr.indexes
        indexes[0].accept(self, mv)
        offset: Temp = indexes[0].getattr("val")
        for size, index in zip(sizes[1:], indexes[1:]):
            length: Temp = mv.visitLoad(size)
            offset: Temp = mv.visitBinary(tacop.TacBinaryOp.MUL, offset, length)
            index.accept(self, mv)
            indexTemp: Temp = index.getattr("val")
            offset: Temp = mv.visitBinary(tacop.TacBinaryOp.ADD, offset,
indexTemp)
        if symbol.isGlobal:
            temp = mv.visitLoadSymbol(symbol)
        else:
            temp = symbol.temp
        sizeTemp = mv.visitLoad(symbol.type.full_indexed.size)
```

```
        offset: Temp = mv.visitBinary(tacop.TacBinaryOp.MUL, offset, sizeTemp)
        addr = mv.visitBinary(tacop.TacBinaryOp.ADD, temp, offset)
        expr.setattr("val", mv.visitLoadMem(addr, 0))
```

至此，可以生成正确的 TAC 码。

在 `riscv.py` 中，加入 ALLOC 对应的汇编：

```
class Riscv:
    ...

    class AllocStack(BackendInstr):
        def __init__(self, offset: int) -> None:
            super().__init__(InstrKind.SEQ, [], [], None)
            self.offset = offset

        def __str__(self) -> str:
            return "addi " + Riscv.FMT3.format(str(Riscv.SP), str(Riscv.SP),
str(-self.offset))
```

在 `riscvasmemitter.py` 中，我实现的 ALLOC 的支持，并且改变了栈帧分配空间的方式：

```
class RiscvAsmEmitter():
    ...

    class RiscvInstrSelector(TACVisitor):
        ...

        def visitAlloc(self, instr: Alloc) -> None:
            self.seq.append(Riscv.AllocStack(instr.size))
            self.seq.append(Riscv.Move(instr.dst, Riscv.SP))

class RiscvSubroutineEmitter():
    ...

    def emitFunc(self):
        self.printer.printComment("start of prologue")
        self.printer.printInstr(Riscv.SPAdd(-self.nextLocalOffset))
        self.printer.printInstr(Riscv.NativeStoreWord(Riscv.FP, Riscv.SP,
len(Riscv.CalleeSaved) * 4 + 4))
        self.printer.printInstr(Riscv.Move(Riscv.FP, Riscv.SP))


        # in step9, you need to think about how to store RA here
        # you can get some ideas from how to save CalleeSaved regs
        for i in range(len(Riscv.CalleeSaved)):
            if Riscv.CalleeSaved[i].isUsed():
                self.printer.printInstr(
                    Riscv.NativeStoreWord(Riscv.CalleeSaved[i], Riscv.FP, 4 * i)
                )

        self.printer.printInstr(Riscv.NativeStoreWord(Riscv.RA, Riscv.FP,
len(Riscv.CalleeSaved) * 4))

        self.printer.printComment("end of prologue")
```

```
        self.printer.println("")

        self.printer.printComment("start of body")

        # in step9, you need to think about how to pass the parameters here
        # you can use the stack or regs

        # using asmcodeprinter to output the RiscV code
        for instr in self.buf:
            self.printer.printInstr(instr)

        self.printer.printComment("end of body")
        self.printer.println("")

        self.printer.printLabel(
            Label(LabelKind.TEMP, self.info.funcLabel.name +
Riscv.EPILOGUE_SUFFIX)
        )
        self.printer.printComment("start of epilogue")

        for i in range(len(Riscv.CalleeSaved)):
            if Riscv.CalleeSaved[i].isUsed():
                self.printer.printInstr(
                    Riscv.NativeLoadWord(Riscv.CalleeSaved[i], Riscv.FP, 4 * i)
                )

        self.printer.printInstr(Riscv.NativeLoadWord(Riscv.RA, Riscv.FP,
len(Riscv.CalleeSaved) * 4))

        self.printer.printInstr(Riscv.Move(Riscv.SP, Riscv.FP))
        self.printer.printInstr(Riscv.NativeLoadWord(Riscv.FP, Riscv.SP,
len(Riscv.CalleeSaved) * 4 + 4))
        self.printer.printInstr(Riscv.SPAdd(self.nextLocalOffset))
        self.printer.printComment("end of epilogue")
        self.printer.println("")

        self.printer.printInstr(Riscv.NativeReturn())
        self.printer.println("")
```

我的实现有点类似于思考题中 VLA 的分配方式，所有 ALLOC 出来的东西都是不在 LocalOffset 的范围中的，而是随着 sp 一直改变，将指针记录下来。能这样实现是因为我用 fp 存了一开始 sp 的值，之后恢复。

至此，可以生成正确的 ASM 代码。

思考题

1. C 语言规范规定，允许局部变量是可变长度的数组（Variable Length Array，VLA），在我们的实验中为了简化，选择不支持它。请你简要回答，如果我们决定支持一维的可变长度的数组(即允许类似 `int n = 5; int a[n];` 这种，但仍然不允许类似 `int n = ...; int m = ...; int a[n][m];` 这种)，而且要求数组仍然保存在栈上（即不允许用堆上的动态内存申请，如 `malloc` 等来实现它），应该在现有的实现基础上做出那些改动?

首先改动语法规则，当前我使用 `ArraySpecifier` 来递归地表示数组，每一层的 `size` 属性都是一个 `IntLiteral`。但是现在顶层的 `size` 允许是 `Identifier` 了，不仅要对 `size` 做修改，还要加入判定，是否只有顶层是 `Identifier` 。

在 namer 中，数组一层索引的检查需要修改，没什么好的检查方法我觉得可以跳过检查第一层，运行时出了问题直接报错。

在 TAC 生成的步骤，第一层 offset 计算方式需要微调，如果是由 `Identifier` 给定的，就应该直接使用 `Identifier` 的临时变量计算，而不是 load 当前索引长度的立即数。还要设计新的 ALLOC，可以 ALLOC 一个变量。

在汇编代码生成的步骤，由于我当前的实现不会把 ALLOC 操作考虑到 LocalOffset 的计算中（即，我当前已经没有在进入函数时统一给局部变量分配内存），我认为已经大部分解决了 VLA 分配内存的问题，直接按照 TAC 一条条转换计算指令，然后改 sp 就行了。唯一可能要注意的是，应当把 ALLOC 的大小保存一下。