Stage-6

计11班 周韧平 2021010699

实验内容

Step10 全局变量

前端

首先为program增加文法，使其可以生成声明和函数，并在 `./frontend/ast/tree.py` 中为 `Program` 类增加 `global_vars` 方法，用于访问全局变量子节点

```
def p_decleration_program(p):
    """
    program :  declaration Semi program
    """
    p[3].children = [p[1]] + p[3].children
    p[0] = p[3]
```

```
def global_vars(self) -> dict[str, Declaration]:
    return {decl.ident.value: decl for decl in self if
isinstance(decl,Declaration)}
```

中端

在 `./frontend/typecheck/namer.py` 中的 `visitDeclaration` 函数中根据当前作用域对声明的变量进行初始化操作，如果是全局变量，还应该检查是否为常量初始化，并修改 `visitProgram` 使其可以遍历全局变量

```
def visitDeclaration(self, decl: Declaration, ctx: ScopeStack) -> None:
    ...
    if ctx.current_scope().isGlobalScope() and decl.init_expr is not NULL and
isinstance(decl.init_expr,IntLiteral) == False: # 判断是否是常量初始化
            raise DecafGlobalVarBadInitValueError
    decl_var =
VarSymbol(decl.ident.value,decl.var_t,ctx.current_scope().isGlobalScope())
```

```
def visitProgram(self, program: Program, ctx: ScopeStack) -> None:
    # Check if the 'main' function is missing
    if not program.hasMainFunc():
        raise DecafNoMainFuncError
    for vars in program.children:
        if isinstance(vars, Declaration):
            vars.accept(self,ctx)
    for func in program.funct_list: # 弃用原有用字典遍历的方式，从而可以访问重复的函数名
        func.accept(self, ctx)
```

和step9类似，在 `./utils/tac/tacinstr.py` 中添加 `LoadSymbol` 和 `Load` 类，用于三地址码生成，不过不用在 `TacVistor` 中完善访问函数了（上一个 Step 其实应该也不用，会去直接调用 `RiscvInstrSelector` 中的继承方法），然后修改 `./frontend/tacgen/tacgen.py` 中的逻辑，实验指导书在这里的解释略有模糊，根据我自己的理解，在访问变量时（对应 `visitIdentifier`），如果该变量为全局变量，则应该先读取全局符号表的地址，再 Load 变量的值，而在为变量赋值时（对应 `visitAssignment`），如果该变量是全局变量，也应该Store回原来的地址。

```
class LoadSymbol(TACInstr):
    def __init__(self, output_temp:Temp, symbol: str) -> None:
        super().__init__(InstrKind.SEQ, [output_temp], [], None)
        self.output = output_temp
        self.symbol = symbol

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

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

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

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

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

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

    def __str__(self) -> str:
        return "Store %s, %s, %s" % (str(self.input), str(self.src),
str(self.offset))

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

```

```
class TACFuncEmitter(TACVisitor):
    def __init__(
        self, entry: FuncLabel, numArgs: int, labelManager:
LabelManager,global_vars:dict[str,Declaration]
    ) -> None:
        ...
```

```
        self.global_vars = global_vars

    def visitLoadSymbol(self,symbol:str) -> None:
        output_temp = self.freshTemp()
        self.func.add(LoadSymbol(output_temp,symbol))
        return output_temp

    def visitLoadGlobal(self, src: Temp, offset:int) -> None:
        output_temp = self.freshTemp()
        self.func.add(LoadGlobal(output_temp,src,offset))
        return output_temp


class TACGen(Visitor[TACFuncEmitter, None]):
    ...
    def visitIdentifier(self, ident: Identifier, mv: TACFuncEmitter) ->
None:
        """
        1. Set the 'val' attribute of ident as the temp variable of the
'symbol' attribute of ident.
        """
        symbol = ident.getattr('symbol')
        if symbol.isGlobal:
            symbol_addr_temp = mv.visitLoadSymbol(symbol.name)
            symbol.temp = mv.visitLoadGlobal(symbol_addr_temp,0)

        # else:
        ident.setattr('val',symbol.temp)
        return

    def visitAssignment(self, expr: Assignment, mv: TACFuncEmitter) -> None:
        """
        1. Visit the right hand side of expr, and get the temp variable of
left hand side.
        2. Use mv.visitAssignment to emit an assignment instruction.
        3. Set the 'val' attribute of expr as the value of assignment
instruction.
        """
        expr.lhs.accept(self,mv)
        expr.rhs.accept(self,mv)
        symbol = expr.lhs.getattr('symbol')
        temp =
mv.visitAssignment(expr.lhs.getattr('symbol').temp,expr.rhs.getattr('val'))
        if symbol.isGlobal:
            symbol_addr_temp = mv.visitLoadSymbol(symbol.name)
            mv.visitStoreGlobal(temp, symbol_addr_temp, 0)
        expr.setattr('val',temp)
        return
```

后端

后端首先要修改 `./backend/riscv/riscvasmemitter.py` 的 `RiscvAsmEmitter` 类的逻辑，增加对 `.data` 字段和 `.bss` 字段的定义，这里不可避免地要用到中端存储的全局变量符号表，因此选择在 `main.py` 中传递，此外，还需要实现 `LoadSymbol`，`LoadGlobal` 和 `StoreGlobal` 三个访问函数的定义

```
class RiscvAsmEmitter(AsmEmitter):
    def __init__(
        self,
        allocatableRegs: list[Reg],
        callerSaveRegs: list[Reg],
        global_vars: dict[str, Declaration]
    ) -> None:
        super().__init__(allocatableRegs, callerSaveRegs)


        # the start of the asm code
        # int step10, you need to add the declaration of global var here
        bss = [(name,decl) for name,decl in global_vars.items() if
decl.init_expr is NULL ]
        data = [(name,decl) for name,decl in global_vars.items() if
decl.init_expr is not NULL ]
        if data is not None:
            self.printer.println(".data")
            for (name,decl) in data:
                self.printer.println(".global %s" % (name))
                self.printer.println("%s:" % (name))
                self.printer.println("  .word %s" % (decl.init_expr.value))
        if bss is not None:
            self.printer.println(".bss")
            for (name,decl) in bss:
                self.printer.println(".global %s" % (name))
                self.printer.println("%s:" % (name))
                self.printer.println("  .space 4")
        self.printer.println(".text")
        self.printer.println(".global main")
        self.printer.println("")
   ...

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

        def visitLoadGlobal(self, instr: LoadGlobal) -> None:
            self.seq.append(Riscv.LoadGlobal(instr.output, instr.src,
instr.offset))

        def visitStoreGlobal(self, instr: LoadGlobal) -> None:
            self.seq.append(Riscv.StoreGlobal(instr.input, instr.src,
instr.offset))
        # in step11, you need to think about how to store the array
```

在 `riscv.py` 中完善所需要的函数，用来生成真正的汇编码

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

    def __str__(self) -> str:
        return "la " + Riscv.FMT2.format(
```

```
            str(self.dsts[0]),
            str(self.symbol)
        )

class LoadGlobal(TACInstr):
    def __init__(self, output: Temp, src: Temp, offset:int) -> None:
        super().__init__(InstrKind.SEQ, [output], [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 StoreGlobal(TACInstr):
    def __init__(self, input: Temp, src: Temp, offset:int) -> None:
        super().__init__(InstrKind.SEQ, [], [input,src], 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])
        )

```

Step11 数组

前端

首先在 `tree.py` 中增加 `TArray` 和 `IndexExpr` 结点的定义，前者用于定义一种新的变量类型（原来只有INT），后者用于查找索引，并定义相应的文法，使得可以声明一个数组，这里我还稍微修改了全局变量部分的文法，因为之前的文法要求必须在声明全局变量后再声明函数，本步中的一些测例并不遵循这个规范。

```
class TArray(TypeLiteral):
    "AST node of type `array(int)`."
    def __init__(self, array:ArrayType) -> None:
        super().__init__("type_int_array", INT)
        self.array = array

    def __getitem__(self, key: int) -> Node:
        raise _index_len_err(key, self)

    def __len__(self) -> int:
        return 0

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

class IndexExpr(Expression):
    """
    AST node of arry index.
    """

```

```
    def __init__(self, base:Identifier, index:Expression) -> None:
        super().__init__("index_expr")
        self.base = base
        self.index = index

    def __getitem__(self, key: int) -> Node:
        raise (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 f"{self.base}[{self.index}]"

```

```
def p_declaration_array(p):
    """
    declaration : type Identifier index
    """
    array = ArrayType(p[1], len(p[3]))
    p[0] = Declaration(TArray(array), p[2], p[3])


def p_index(p):
    """
    index :  LBracket Integer RBracket index
    """
    p[0] = [p[2]] + p[4]

def p_index_empty(p):
    """
    index :  empty
    """
    p[0] = []
```

中端

需要在 `namer.py` 中增加 `IndexExpr` 的访问函数，同时在`visitUnary` 和 `visitBinary` 和 `visitDeclaration`中增加判断，用于检查数组引用是否规范，即必须指向存储的元素，维度和下标都应该符合要求。

```
 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.lookup(decl.ident.value,only_top=True) is not None:
```

```
            raise DecafDeclConflictError
        if ctx.current_scope().isGlobalScope() and decl.init_expr is not NULL and isinstance(decl.var_t, TArray) == False and isinstance(decl.init_expr,IntLiteral) == False: # 判断是否是常量初始化
            raise DecafGlobalVarBadInitValueError
        if isinstance(decl.var_t, TArray):
            array = decl.var_t.array
            while isinstance(array, ArrayType):
                if array.length <= 0:
                    raise DecafBadArraySizeError
                array = array.base
        # import pdb; pdb.set_trace()
        decl_var = VarSymbol(decl.ident.value,decl.var_t,ctx.current_scope().isGlobalScope())
        ctx.declare(decl_var)
        decl.setattr('symbol',decl_var)
        if decl.init_expr is not None:
            decl.init_expr.accept(self,ctx)
        return

def visitIndexExpr(self, array_expr:IndexExpr ,ctx:ScopeStack) -> None:
    symbol = ctx.lookup(array_expr.base.value)
    if symbol is None:
        raise DecafUndefinedVarError(array_expr.base.value)
    if symbol.type.array.dim != len(array_expr.index_list):
        raise DecafBadIndexError
    array_expr.base.accept(self,ctx)
    for index in array_expr.index_list:
        if isinstance(index, IntLiteral) == False and isinstance(index, IndexExpr) == False and isinstance(index, Binary) == False and isinstance(index, Unary) == False and isinstance(index, Identifier) == False:
            # import pdb; pdb.set_trace()
            raise DecafBadIndexError
        index.accept(self,ctx)
```

```
def visitUnary(self, expr: Unary, ctx: Scope) -> None:
    if hasattr(expr.operand, "value"):
        symbol = ctx.lookup(expr.operand.value)
        if hasattr(symbol,"type") and isinstance(symbol.type,TArray) and not hasattr(expr.operand,"index_list"):
                raise DecafBadIndexError
    expr.operand.accept(self, ctx)

def visitBinary(self, expr: Binary, ctx: Scope) -> None:
    if hasattr(expr.lhs,"value"):
        symbol = ctx.lookup(expr.lhs.value)
        if hasattr(symbol,"type") and isinstance(symbol.type,TArray) and not hasattr(expr.lhs,"index_list"):
            raise DecafBadIndexError
    if hasattr(expr.rhs,"value"):
        symbol = ctx.lookup(expr.rhs.value)
```

```
        if hasattr(symbol,"type") and isinstance(symbol.type,TArray) and not hasattr(expr.rhs,"index_list"):
            raise DecafBadIndexError
    expr.lhs.accept(self, ctx)
    expr.rhs.accept(self, ctx)
```

然后是在 `tacgen.py` 和 `tacinstr.py` 中为下标访问定义访问函数，这里我的实现方法是，从后向前遍历数组的每一维度下表，用 MUL + ADD 的方式计算整体的offset。另外一个是要在定义和引用访问函数中增加对数组的特判，这里修改的比较细碎，核心思路是首先要判断是否为全局变量数组，如果是的话在引用时只进行 `LoadSymbol` 这一步（普通的全局变量还要从对应地址取数，但对数组而言这一步就是要取地址，所以省去第二步），而在赋值时，如果是数组，则应该改将计算结果写入对应的地址而非寄存器，这里直接使用了上一节实现的 `visitStoreGlobal` 访问函数。另外参考文档设计，在定义局部数组时，引入TAC指令ALLOC，用来规定需要的栈空间大小

```
def visitIndexExpr(self, array_expr: IndexExpr, mv: TACFuncEmitter) -> None:
    array_expr.base.accept(self,mv)
    for index in array_expr.index_list:
        index.accept(self,mv)
    symbol:VarSymbol = array_expr.base.getattr("symbol")
    # offset = 0
    size = 4
    array = symbol.type.array
    id = 1
    add_result_temp = symbol.temp
    while isinstance(array, ArrayType):
        mul_result_temp = mv.visitBinary(tacop.TacBinaryOp.MUL, array_expr.index_list[-id].getattr("val"), mv.visitLoad(size))
        add_result_temp = mv.visitBinary(tacop.TacBinaryOp.ADD, add_result_temp, mul_result_temp)
        id = id + 1
        size = size * array.length
        array = array.base
    # symbol.type.array =

    array_expr.setattr("addr", add_result_temp)
    array_expr.setattr("val", mv.visitLoadGlobal(add_result_temp, 0))
    # import pdb; pdb.set_trace()


```

后端

后端要解决的问题主要是如何处理全局数组以及为局部数组变量分配合适的栈空间，全局数组的增加只需要对 `RiscvAsmEmitter` 的构造函数稍加改动即可，而对于局部数组的栈空间分配，我采取的办法是首先扫描一遍函数内的所有指令，用字典的方式存储所有定义的数组，并为之分配对应的栈空间地址，然后将这个字典通过 `SubroutineInfo` 传给 `RiscvSubroutineEmitter` 类和 `RiscvInstrSelector` 类。

```
class RiscvAsmEmitter(AsmEmitter):
    def __init__(
        self,
        allocatableRegs: list[Reg],
        callerSaveRegs: list[Reg],
```

```
        global_vars: dict[str, Declaration]
    ) -> None:
        super().__init__(allocatableRegs, callerSaveRegs)


        # the start of the asm code
        # int step10, you need to add the declaration of global var here
        bss = [(name,decl) for name,decl in global_vars.items() if
decl.init_expr is NULL ]
        data = [(name,decl) for name,decl in global_vars.items() if
decl.init_expr is not NULL ]
        if data is not None:
            self.printer.println(".data")
            for (name,decl) in data:
                self.printer.println(".globl %s" % (name))
                self.printer.println("%s:" % (name))
                self.printer.println("  .word %s" % (decl.init_expr.value))
        if bss is not None:
            self.printer.println(".bss")
            for (name,decl) in bss:
                self.printer.println(".globl %s" % (name))
                self.printer.println("%s:" % (name))
                if isinstance(decl.var_t, TArray):
                    self.printer.println("  .space %s" %
(decl.var_t.array.size))
                else:
                    self.printer.println("  .space 4")
        self.printer.println(".text")
        self.printer.println(".global main")
        self.printer.println("")

```

```

class SubroutineInfo:
    def __init__(self, funcLabel: FuncLabel, array_offset:int,
local_array_offset_dict) -> None:
        self.funcLabel = funcLabel
        self.array_offset = array_offset
        self.local_array_offset_dict = local_array_offset_dict

    def __str__(self) -> str:
        return "funcLabel: {}, array_offset:{}".format(
            self.funcLabel.name, str(self.array_offset)
        )
```

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

    def __str__(self) -> str:
        return "addi " + Riscv.FMT3.format(
            str(self.dsts[0]),  str(Riscv.SP), str(self.offset)
        )

```

Step12 数组进阶

前端

主要是增加数组初始化和允许参数传参的文法，文法定义的形式和函数传参整体很像，需要注意的是考虑传参数可以有 a[100000] 和 a[] 这两种方法

```python
def p_list_parameter(p):
    """
    parameter : type Identifier LBracket Integer RBracket
    """
    array = ArrayType(p[2], p[4].value)
    p[0] = Parameter(TArray(array),p[2])

def p_list_parameter_empty(p):
    """
    parameter : type Identifier LBracket RBracket
    """
    array = ArrayType(p[2])
    p[0] = Parameter(TArray(array), p[2])

def p_declaration_init(p):
    """
    declaration : type Identifier Assign expression
    """
    p[0] = Declaration(p[1], p[2], p[4])

def p_declaration_init_array(p):
    """
    declaration : type Identifier index_array Assign array_init
    """
    array = p[1]
    for integer in p[3]:
        array = ArrayType(array, integer.value)
    p[0] = Declaration(TArray(array), p[2], p[5])

def p_array_init(p):
    """
    array_init : LBrace Integer Integer_list RBrace
    """
    p[3].init_array = [p[2]] + p[3].init_array
    p[0] = p[3]

```

```
def p_Integer_list(p):
    """
    Integer_list : Comma Integer Integer_list
    """
    p[3].init_array = [p[2]] + p[3].init_array
    p[0] = p[3]

def p_Integer_list_empty(p):
    """
    Integer_list : empty
    """
    p[0] = ArrayList([])
```

中端

由于测例相对有限，这一节在语义分析上不需要做过多的语法检查，只需要在参数位置增加数组类型相应的判定即可。生成tac码时，需要在 `visitDeclaration` 中增加列表初始化的判定（`if decl.init_expr is not NULL`下面），我这里的办法是比较定义的数组长度和列表本身的长度比较，对于不足的手动加入 `IntLiteral(0)`

```
def visitDeclaration(self, decl: Declaration, mv: TACFuncEmitter) -> None:
    """
    1. Get the 'symbol' attribute of decl.
    2. Use mv.freshTemp to get a new temp variable for this symbol.
    3. If the declaration has an initial value, use mv.visitAssignment to
set it.
    """
    symbol:VarSymbol = decl.getattr('symbol')
    symbol.temp = mv.freshTemp()
    if isinstance(symbol.type, TArray):
        # import pdb; pdb.set_trace()
        mv.visitAlloc(symbol.temp, symbol.type.array.size)
    # import pdb; pdb.set_trace()
    if decl.init_expr is not NULL:
        decl.init_expr.accept(self,mv)
        if not isinstance(symbol.type, TArray):
            mv.visitAssignment(symbol.temp,decl.init_expr.getattr('val'))
        else:
            array_list =  decl.init_expr.getattr("val")
            for i in range(symbol.type.array.length):
                integer = array_list[i] if i < len(array_list) else
IntLiteral(0)
                # int_literal = IntLiteral(integer)
                self.visitIntLiteral(integer,mv)
                mv.visitStoreGlobal(integer.getattr('val'), symbol.temp,
4*i)
    return
```

后端

在定义全局数组时，后端可能会出现需要在data段定义一个数组的情况，这里我通过查阅 risc-v 规范了解了data段对应的格式，并修改该字段生成的逻辑

```

```

```
bss = [(name,decl) for name,decl in global_vars.items() if decl.init_expr is NULL ]
data = [(name,decl) for name,decl in global_vars.items() if decl.init_expr is not NULL ]
if data is not None:
    self.printer.println(".data")
    for (name,decl) in data:
        self.printer.println(".globl %s" % (name))
        self.printer.println("%s:" % (name))
        if isinstance(decl.var_t,TArray):
            for integer in decl.init_expr.init_array:
                self.printer.println("  .word %s" % (integer.value))
            self.printer.println("  .zero %s" % (4*(decl.var_t.array.length-len( decl.init_expr.init_array))))

        else:
            self.printer.println("  .word %s" % (decl.init_expr.value))
if bss is not None:
    self.printer.println(".bss")
    for (name,decl) in bss:
        self.printer.println(".globl %s" % (name))
        self.printer.println("%s:" % (name))
        if isinstance(decl.var_t, TArray):
            self.printer.println("  .space %s" % (decl.var_t.array.size))
        else:
            self.printer.println("  .space 4")
```

数组传参部分，后端没有做过多的修改，栈结构依然保持之前的形式，区别在于如果传入参数为数组的话，传入的其实是数组的指针，而不是把数组再复制一份到栈上。

思考题

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）

在 Non-PIC 下，程序的指令和数据引用不依赖于固定的内存地址，而是在运行时通过绝对地址来访问。`la` 伪指令可能被翻译成 `lui` 指令，和 `addi` 指令的组合，而不依赖于pc。

```
lui v0, %hi(a)
addi v0, v0, %lo(a)
```

`%hi` 和 `%lo` 分别表示符号的高位和低位

在 PIC 下， 代码和数据的引用使用相对地址或符号，而不是绝对地址。这时候 `la` 指令可能翻译为

```
auipc v0, %pcrel_hi(a)
addi v0, v0, %pcrel_lo(a)
```

其中 `%pcrel_hi` 和 `%pcrel_lo` 是 用于计算相对地址的伪操作数，他会计算符号和当前 PC 地址的相对偏移量，得到一个相对于pc的高位地址后，再和对应的低位相对偏移地址相加，得到符号对应的完整地址，实现了基于 PC 相对地址的访问

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?

目前我们的栈设计是这样的

```
++++++++++++++++++++++
(Caller-saved registers)
++++++++++++++++++++++
Arg n-1
Arg n-2
...
Arg 1
Arg 0
++++++++++++++++++++++ FP
Loacl Array n
...
Loacl Array 2
Loacl Array 1
(Callee-saved registers)
++++++++++++++++++++++ SP
```

由于我们知道每个数组需要的栈空间是多大，因此只需要在进入函数前预留出对应大小的栈空间就行了，但在引入可变长度的数组后，我们可以改变原本 `Loacl Array n` 位置存储的信息，比如，可以为每个变长数组分配8字节的空间，用于存储数组在栈空间中的起始地址和数组长度，起作用相当于指针，而真正的数组在起作用域内可以通过 SP 的变化来分配空间，也就是说，当进入该数组的作用域时，SP减去相应的空间大小，并设置对应的数组指针中的地址，当离开作用域时，将SP加回，释放对应的栈空间。对于 Loacl Array pointer 本身的寻址，可以通过记录当前栈空间的大小来通过SP寻址，也可以利用 FP 来寻址，后者由于在一次函数调用内不需要变化因此可能实现起来更为容易

```
++++++++++++++++++++++
(Caller-saved registers)
++++++++++++++++++++++
Arg n-1
Arg n-2
...
Arg 1
Arg 0
++++++++++++++++++++++ FP
Loacl Array pointer n
...
Loacl Array pointer 2
Loacl Array pointer 1
(Callee-saved registers)
Loacl Array n
Loacl Array n-1
...
Loacl Array 1
++++++++++++++++++++++ SP
```

作为函数参数的数组类型第一维可以为空。事实上，在 C/C++ 中即使标明了第一维的大小，类型检查依然会当作第一维是空的情况处理。如何理解这一设计?

因为 c/c++ 在传数组参数时其实传递的是数组的指针，也即首元素地址，而非传递整个数组，因此只要知道了数据类型（每个元素所占的字节数）和数组长度就可以计算出对应元素的位置以及所占空间的大小，这样做一方面可以使数组的传参更加灵活，比如把一个大数组传入但参数定义的是一个小数组并不会报错，而且这样做**对于习惯于使用指针的 C/C++** 来说也更符合其设计规范，此外，这种做法可以节省内存空间，同时也避免了在传递数组时进行内存拷贝操作，在时间和空间开销上都是更高效的。但可能会存在数组越界等问题，且不容易在静态语法检查时查出，只能在运行时报错，增加了调试的难度。