STAGE-5：数组

计04 何秉翔 2020010944

1. 实验内容

1.1 step-11：数组

我们按照从前端、中端到后端的顺序分别介绍，在中间主要介绍实现的思路以及关键部分的代码。

1.1.1 语法分析 & 语法树建立

数组首先需要有关于数组的声明，在此我们沿用原有的 `Declaration` 结点，在其中增加一些属性，比如数组初始化维度列表，初始值列表，来标记关于数组初始化的性质。

```
class Declaration(Node):
    """
    AST node of declaration.
    """
    def __init__(
        self,
        var_t: TypeLiteral,
        ident: Identifier,
        array_dim_list: Optional[list[IntLiteral]] = None,
        init_expr: Optional[Expression] = None,
        init_array_elements: Optional[list[IntLiteral]] = None,
    ) -> None:
        super().__init__("declaration")
        self.var_t = var_t
        self.ident = ident
        self.array_dim_list = array_dim_list or NULL
        self.init_expr = init_expr or NULL
        self.init_array_elements = init_array_elements or NULL
...
```

然后，对于数组元素的访问，比如读或者写，我们需要一个语法树节点，在此我们建立一个 `ArrayElement` 结点，在其中我们存放数组的标识符以及访问元素的 `index` 列表，比如对于 `a[0][1][2]`，访问列表就是 `{0, 1, 2}`

```
class ArrayElement(Expression):
    """
    AST node of array element.
    """
    def __init__(
        self,
        ident: Identifier,
        array_dim_list: list[Expression],
    ) -> None:
        super().__init__("arrayElement")
        self.ident = ident
        self.array_dim_list = array_dim_list
```

接着我们更改对于 `Declaration` 的语法解析：

```
"""
declaration : type Identifier
    | type Identifier index_list
"""
```

其中 `index_list` 为数组索引：

```
index_list : index_list_r LBrack expression RBrack
index_list_r : index_list_r LBrack expression RBrack
    | empty
```

然后我们支持对于数组元素的赋值语句 `Assign`：

```
assignment : Identifier index_list Assign expression
```

对于数组值参数的传递，我们修改 `postfix` 如下，以支持数组元素传参：

```
postfix : Identifier index_list
```

到此，前端部分已经完成。

1.1.2 语义分析 & 符号表构建

在中端，我们首先对数组的语义进行检查，比如数组维度为负数，对数组元素的访问超出了数组的声明维度列表。在声明的时候，我们首先检查数组的维度，同时在符号表里增加一个属性 `is_array_symbol` 来判断该符号是否是数组标识符对应的符号：

```
if len(decl.array_dim_list):
    decl.ident.is_array_ident = True
    prod = 1
    for index in decl.array_dim_list:
        prod *= index.value
        if index.value <= 0:
            raise DecafBadArraySizeError
symbol = VarSymbol(decl.ident.value, decl.var_t.type, array_dim_list=decl.array_dim_list)
symbol.is_array_symbol = decl.ident.is_array_ident
```

接着在访问数组元素的时候，要从 `visitAssignment` 进入，因此我们在此判断赋值的合法性，比如一个 `int` 类型的值不能赋值给数组的标识符，反过来一个数组的标识符不能赋值给一个普通变量：

```
# lhs should be var ident, not array ident
if isinstance(expr.lhs, Identifier):
    symbol = ctx.lookup(expr.lhs.value)
    if symbol.is_array_symbol:
        raise DecafBadAssignTypeError
    else:
        if isinstance(expr.rhs, Identifier):
            symbol = ctx.lookup(expr.rhs.value)
            if symbol.is_array_symbol:
                raise DecafBadAssignTypeError
```

涉及到访问数组元素时，我们进入到 `visitArrayElement` 函数，在其中要首先判断访问的维度和声明的维度数目是否一致，比如声明 `int a[1][2]`，访问 `a[0]` 就是不合法的行为；其次，要判断访问的维度是否超出了声明的维度范围，比如声明 `int a[2]`，访问 `a[2]` 就是不合法的行为：

```
if len(arrayElement.array_dim_list) != len(symbol.array_dim_list):
    raise DecafBadIndexError('dim not match!')
for (i, index) in enumerate(arrayElement.array_dim_list):
    index.accept(self, ctx)
    if isinstance(index, IntLiteral): # 只对 IntLiteral 检查语义
        if i == 0:
            continue
        if index.value >= symbol.array_dim_list[i].value:
            raise DecafBadIndexError(str(index.value))
```

最后是对于数组元素的一元操作以及二元操作需要保证合法，比如不能加上一个数组标识符，或者 非 一个数组标识符：

```
def visitUnary(self, expr: Unary, ctx: ScopeStack) -> None:
    # should be var ident, not array ident
    if isinstance(expr.operand, Identifier):
        symbol = ctx.lookup(expr.operand.value)
        if symbol.is_array_symbol:
            raise DecafBadAssignTypeError
    expr.operand.accept(self, ctx)

def visitBinary(self, expr: Binary, ctx: ScopeStack) -> None:
    # should be var ident, not array ident
    if isinstance(expr.lhs, Identifier):
        lhs_symbol = ctx.lookup(expr.lhs.value)
        if lhs_symbol.is_array_symbol:
            raise DecafBadAssignTypeError
    if isinstance(expr.rhs, Identifier):
        rhs_symbol = ctx.lookup(expr.rhs.value)
        if rhs_symbol.is_array_symbol:
            raise DecafBadAssignTypeError
    expr.lhs.accept(self, ctx)
    expr.rhs.accept(self, ctx)
```

1.1.3 TAC 生成

在三地址码生成阶段，我们先针对局部数组做处理，再对全局数组做处理。

在局部数组的声明部分，我们在 visitDeclaraion 中对声明普通变量和声明数组变量分别处理，根据 Declaration 的初始维度列表来判断，对于数组，我们先计算需要分配的空间 prod，然后调用 mv.visitAllocForArray 生成 alloc 的中间指令：

```
if decl.array_dim_list == NULL:
    # if not hasattr(symbol, 'temp'):
    if decl.init_expr != NULL: # with initial value
        decl.init_expr.accept(self, mv)
        mv.visitAssignment(symbol.temp, decl.init_expr.getattr("val"))
    # decl.ident.accept(self, mv)
    else:
        prod = 1
        for index in decl.array_dim_list:
            prod *= index.value
        mv.visitAllocForArray(symbol.temp, prod * 4)
```

我们定义 Alloc 中间指令如下：

```python
class AllocForArray(TACInstr):
    def __init__(self, dst: Temp, cnt_bytes: int) -> None:
        super().__init__(InstrKind.SEQ, [dst], [], None)
        self.cnt_bytes = cnt_bytes

    def __str__(self) -> str: # store _T0, 4(_T1): store _T0 to the addr of (4 + _T1)
        return "%s = alloc %s" % (self.dsts[0], str(self.cnt_bytes))

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

在赋值中，我们对赋值号左边的变量分四类情况来考虑，分别是

- 全局数组元素
- 局部数组元素
- 全局普通变量
- 局部普通变量

对于全局的变量，我们按照 `step 10` 的思路，先 `visitLoadGlobalVarSymbol` 再将赋值号右边的值 `visitStoreGlobalVarAddr` 进去；对于局部的变量，我们直接将右边的值通过 `visitAssignment(普通变量)` 和 `visitStoreGlobalVarAddr(数组元素)` 赋值：

```python
# visitAssignment
if len(symbol.array_dim_list) == 0:
    if symbol.isGlobal:
        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"))
        expr.setattr('val', expr.rhs.getattr("val"))
    else:
        expr.lhs.accept(self, mv)
        expr.rhs.accept(self, mv)
        temp = expr.lhs.getattr("val")
        mv.visitAssignment(temp, expr.rhs.getattr("val"))
        expr.setattr('val', expr.rhs.getattr("val"))
else:
    if symbol.isGlobal:
        expr.lhs.accept(self, mv)
        expr.rhs.accept(self, mv)
        addrTemp = mv.freshTemp() # the base addr temp of global var
        mv.visitLoadGlobalVarSymbol(addrTemp, symbol.name)
        addrTemp = mv.visitBinary(tacop.BinaryOp.ADD, addrTemp, expr.lhs.getattr("offset"))
        mv.visitStoreGlobalVarAddr(addrTemp, expr.rhs.getattr("val"),
expr.lhs.getattr("offset"))
        expr.setattr('val', expr.rhs.getattr("val"))
    else:
        expr.lhs.accept(self, mv)
        expr.rhs.accept(self, mv)
        addrTemp = mv.visitBinary(tacop.BinaryOp.ADD, symbol.temp,
expr.lhs.getattr("offset"))
        mv.visitStoreGlobalVarAddr(addrTemp, expr.rhs.getattr("val"),
expr.lhs.getattr("offset"))
        expr.setattr('val', expr.rhs.getattr("val"))
```

在访问到具体数组元素时，我们需要根据访问索引计算出具体的偏移，计算方法采用指导手册提供的思路：

> 那么，如何将数组下标对应到偏移地址？对一维数组，下标的常数倍（int 型的大小为 4 个字节，倍数为4）即为偏移量。而对于高维数组，我们可以将其视为一个展开成一维的大数组。对于数组 a[d1][d2]...[dn]，访问元素 a[i1][i2]...[in] 可以等价于访问 a[i1*d2*d3...dn + i2*d3*...*dn + ... + in]。在将数组索引翻译成 TAC 时，同学们需要自行将数组下标转换成地址计算指令。这个步骤并不困难，但可能比较繁琐，同学们在实现时要注意细节，避免错误。

计算出偏移量后，我们为 `ArrayElement` 设置一个新的 `attr`，利用 `setattr` 和 `getattr` 来传递计算出来的偏移量的 `Temp`。然后再对访问的数组元素分为四类来分别处理：

- 全局
  - 赋值号左边：只计算偏移量，利用 `setattr('offset')` 来存储
  - 赋值号右边：计算偏移量和元素值，利用 `setattr('offset')` 和 `setattr('val')` 来分别存储
- 局部
  - 赋值号左边：只计算偏移量，利用 `setattr('offset')` 来存储
  - 赋值号右边：计算偏移量和元素值，利用 `setattr('offset')` 和 `setattr('val')` 来分别存储

```python
def visitArrayElement(self, arrayElement: ArrayElement, mv: FuncVisitor) -> None:
    """
    0. symbol.temp is the base addr of array, for local one
    1. arrayElement.array_dim_list (int or ident), symbol.array_dim_list (int)
        1.1 ident.getattr("val") is the temp of ident
    2. whatever global or not, calculate the offset
    3. for global one
        3.1 lhs: only offset
        3.2 rhs: offset + value
    4. for local one
        4.1 lhs: only offset
        4.2 rhs: offset + value
    """
    symbol = arrayElement.ident.getattr('symbol')
    prod = mv.visitLoad(1) # 累乘值的 temp
    offset = mv.visitLoad(0) # 计算偏移量的 temp
    length = len(symbol.array_dim_list)
    for (i, index) in enumerate(symbol.array_dim_list):
        arrayElement.array_dim_list[length - 1 - i].accept(self, mv)
        symbol.array_dim_list[length - 1 - i].accept(self, mv)

        mul_temp = mv.visitBinary(tacop.BinaryOp.MUL, prod, arrayElement.array_dim_list[length - 1 - i].getattr('val'))
        offset = mv.visitBinary(tacop.BinaryOp.ADD, offset, mul_temp)

        prod = mv.visitBinary(tacop.BinaryOp.MUL, prod, symbol.array_dim_list[length - 1 - i].getattr('val'))

    temp_4 = mv.visitLoad(4)
    offset = mv.visitBinary(tacop.BinaryOp.MUL, offset, temp_4)
    arrayElement.setattr('offset', offset)

    if symbol.isGlobal:
        if not arrayElement.getattr('lhs'):
            addrTemp = mv.freshTemp() # the base addr temp of global var
            mv.visitLoadGlobalVarSymbol(addrTemp, symbol.name)
            valueTemp = mv.freshTemp() # the value temp of global var
            addrTemp = mv.visitBinary(tacop.BinaryOp.ADD, addrTemp, offset)
            mv.visitLoadGlobalVarAddr(valueTemp, addrTemp, offset)
            arrayElement.setattr('val', valueTemp)
    else:
        if not arrayElement.getattr('lhs'): # 不是左值，就去 load
            valueTemp = mv.freshTemp() # the value temp of arrayElement
```

```
            addrTemp = mv.visitBinary(tacop.BinaryOp.ADD, symbol.temp, offset)
            mv.visitLoadGlobalVarAddr(valueTemp, addrTemp, offset)
            arrayElement.setattr('val', valueTemp)
```

最后我们再对全局数组做处理，我们在 `GlobalVar` 类中依葫芦画瓢，类似 `Declaration` 存储全局数组的维度列表和对应的符号 `symbol`，然后在 `transform` 中处理全局变量时初始化好 `GlobalVar` 类变量：

```
if child.init_expr == NULL and len(child.init_array_elements) == 0:
    globalVar = GlobalVar(child.ident.value, False, array_dim_list=child.array_dim_list, decl=child)
else:
    if isinstance(child.init_expr, IntLiteral):
        globalVar = GlobalVar(child.ident.value, True, child.init_expr.value, array_dim_list=child.array_dim_list, decl=child)
    elif len(child.init_array_elements): # 全局数组初始化
        globalVar = GlobalVar(child.ident.value, True, array_dim_list=child.array_dim_list, init_array_elements=child.init_array_elements, decl=child)
    else:
        raise DecafGlobalVarBadInitValueError(child.ident.value)
pw.globalVars.append(globalVar)
```

并在初始化 `GlobalVar` 类全局变量时，计算好需要分配的空间 `cnt_bytes`，为后续全局变量在 `data` 或者 `bss` 段内初始化时做准备：

```
prod = 1
for index in array_dim_list:
    prod *= index.value
self.cnt_bytes = prod * 4
```

至此为止中端部分基本完成。

1.1.4 后端

我们首先在 `selectInstr` 时为新增的 `TAC` 指令设置对应的中间 `riscv` 指令：

```
class AllocForArray(TACInstr):
    def __init__(self, dst: Temp, src: Temp) -> None:
        super().__init__(InstrKind.SEQ, [dst], [src], None)

    def __str__(self) -> str:
        # 数组栈上基址
        return "add " + Riscv.FMT3.format(
            str(self.dsts[0]), str(Riscv.SP), str(self.srcs[0])
        )
```

为了将全局变量和数组关于 `load` 和 `store` 统一处理，我们在生成中间 `TAC` 时让偏移量寄存器都直接与基址寄存器相加算好，也就是后续生成 `riscv` 汇编时不会出现类似 `sw t0, 4(t1)` 的汇编，在具体汇编中"偏移量"都为 `0`，以统一处理。

```
def visitAllocForArray(self, instr: AllocForArray) -> None:
    self.seq.append(Riscv.LoadImm(instr.dsts[0], self.array_offset))
    self.seq.append(Riscv.AllocForArray(instr.dsts[0], instr.dsts[0]))
    self.array_offset += instr.cnt_bytes
```

在全局变量初始化处理时，我们为数组加上 `bss` 初始分配空间即可。然后在 `RiscvSubroutineEmitter` 类中，我们需要基于数组分配的空间来调整 `sp` 指针的偏移量，我们选择在 `CalleeSaved` 寄存器存储的上面直接分配所有的数组栈空间，这可以通过在 `SubroutineInfo` 来记录 `selectInstr` 时分配的所有数组的栈空间大小。在数组栈空间之上，再放置 `ra` 寄存器以及随后的寄存器保存值：

```
# 存储 ra
self.printer.printInstr(Riscv.NativeStoreWord(Riscv.RA, Riscv.SP, 4 * len(Riscv.CalleeSaved) +
self.array_offset))
# CalleeSaved
for i in range(len(Riscv.CalleeSaved)):
    if Riscv.CalleeSaved[i].isUsed():
        self.printer.printInstr(
            Riscv.NativeStoreWord(Riscv.CalleeSaved[i], Riscv.SP, 4 * i + self.array_offset)
        )
```

在结束再全部对应 `load` 回来即可。

至此后端已经完成。

1.2 step-12: 为数组添加更多支持

我们从前端到后端依次进行分析。

1.2.1 前端

为了支持数组的初始化，我们主要修改的是 `declaration` 节点，新增加关于数组的初始化 `array_init_list`：

```
declaration : type Identifier Assign expression
    | type Identifier index_list Assign LBrace array_init_list RBrace

array_init_list : array_init_list Comma Integer
    | Integer
```

在 `Declaration` 节点中新增 `init_array_elements` 来记录初始化元素。

为了支持数组传参，我们主要修改的是参数的解析：

```
type_identifier_union : type
        | type Identifier int_index_list
        | type Identifier LBrack RBrack int_index_list
```

我们支持第一个维度为空，其他维度必须指定，至此前端部分已经完成。

1.2.2 中端

我们首先考虑 `namer.py` 里的修改，对于局部数组初始化，我们需要判断初始化元素个数是否超出了数组声明时分配的总空间大小：

```
... # 计算总大小 prod
if len(decl.init_array_elements) > prod:
    raise DecafGlobalVarBadInitValueError(decl.ident.value)
```

其次，对于数组元素的初始化还需考虑赋值号左右两边的类型，若将 `int` 类型赋值给数组类型，或者反过来都是不允许的：

```
# visitDeclaration
if len(decl.array_dim_list):
    if decl.init_expr != NULL:
        raise DecafBadAssignTypeError
else:
    if isinstance(decl.init_expr, Identifier):
        symbol = ctx.lookup(decl.init_expr.value)
        if symbol.is_array_symbol:
            raise DecafBadAssignTypeError
```

对于函数传参，我们需要多检查函数对应位置的参数类型是否匹配，比如将数组类型的实参传给 `int` 的形参或者反过来也都是不允许的：

```
# visitCall
for i in range(funcSymbol.parameterNum):
    type_ = funcSymbol.getParaType(i)
    if isinstance(call.argument_list[i], Identifier):
        symbol = ctx.lookup(call.argument_list[i].value)
        if isinstance(type_, Symbol): # array
            if symbol.is_array_symbol and len(symbol.array_dim_list) !=
len(type_.array_dim_list):
                raise DecafBadFuncCallError(call.ident.value)
        else:
            if symbol.is_array_symbol:
                raise DecafBadFuncCallError(call.ident.value)
```

对于 `visitAssignment` 的赋值同样考虑前面几种情况即可。接着我们看 `TAC` 生成的部分，主要关注 `tacgen.py` 里的修改。

首先对于全局数组的初始化处理，我们在 `GlobalVar` 中增加用于记录初始化元素的属性 `init_array_elements`，对于全局数组初始化，我们考虑在 `main` 的 `visitFunction` 里来进行，在其中判断当前函数是否是 `main` 函数，然后按照实验指导书，传参调用 `fill_n` 后再初始化一些非零值：

```
# 对于 main func，初始化所有带有初始化的全局数组
if mv.func.entry.func == "main":
    for globalVar in self.pw.globalVars:
        if len(globalVar.array_dim_list) and globalVar.init_flag:
            prod = 1
            for index in globalVar.array_dim_list:
                prod *= index.value
            # fill_n
            symbol = globalVar.decl.getattr('symbol')

            addrTemp = mv.freshTemp() # the base addr temp of global var
            mv.visitLoadGlobalVarSymbol(addrTemp, symbol.name)

            mv.visitParameter(addrTemp)
            array_len_temp = mv.visitLoad(prod)  # n
            mv.visitParameter(array_len_temp)
            init_val_temp = mv.visitLoad(0)  # 0
            mv.visitParameter(init_val_temp)

            func_temp = mv.freshTemp()
            mv.visitCall(func_temp, FuncLabel('fill_n'))

            for i, ele in enumerate(globalVar.init_array_elements):
                offset = mv.visitLoad(4 * i) # 计算偏移量的 temp
                new_addrTemp = mv.visitBinary(tacop.BinaryOp.ADD, addrTemp, offset)
                rhs = mv.visitLoad(ele.value)
                mv.visitStoreGlobalVarAddr(new_addrTemp, rhs, offset)
```

在传参调用 `visitIdentifier` 时，对于数组类型，我们需要 `visitLoadGlobalVarAddr` 来获得首地址：

```
# visitIdentifier
if not symbol.is_array_symbol: # 普通变量
    mv.visitLoadGlobalVarAddr(valueTemp, addrTemp, None)
    ident.setattr('val', valueTemp)
else: # array
    ident.setattr('val', addrTemp)
```

对于 `visitDeclaration`，我们仿照全局数组初始化方法进行局部数组初始化（略去代码）。至此，中端部分已经基本完成。

1.2.3 后端

在后端部分，我们主要处理全局数组初始化的问题，对于初始化的全局数组，我们在 `data` 段进行初始化，有初始值的用 `word`，没有初始值的为 0 的部分我们用 `zero` 统一初始化：

```
if len(dataGlobalVars):
    self.printer.println(".data")
for globalVar in dataGlobalVars:
    self.printer.println(".global %s" % globalVar.symbol)
    self.printer.println("%s:" % globalVar.symbol)
    if len(globalVar.array_dim_list):
        for ele in globalVar.init_array_elements:
            self.printer.println("    .word %d" % ele.value)
        self.printer.println("    .zero %d" % (globalVar.cnt_bytes -
len(globalVar.init_array_elements) * 4))
    else:
        self.printer.println("    .word %d" % globalVar.init_value)
```

至此，后端已经基本完成。

2. 思考题

2.1 step-11 思考题

问题：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？

解答：如果改为动态分配的形式，在中端我们无法利用 `Alloc` 指令计算出分配的空间，将使用一个中间寄存器 `Temp` 来记录分配的字节数。导致在 `sp` 分配栈上数组空间时，无法计算出一个明确的分配值。但是我们仍可以以寄存器的形式来 "计算" 出正确的偏移量，只是这个偏移量基于动态分配的数组空间以及静态分配的数组空间的寄存器的四则运算得到的，因此在 `riscv` 汇编没真正执行到这一步时是无法真正得到正确的 `sp` 偏移量的，因此我们可以考虑在上下拉动 `sp` 时以寄存器的形式去计算，以动态地改变 `sp` 的值，从而在后端能够准确地指向栈上对应的位置的元素。

2.2 step-12 思考题

问题：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？

解答：这一设计可能是想数组传参主要以传指针的方式进行，而对于一个多维数组，我们认为其在内存中是连续存放的，因此可以通过实验指导手册的方法来计算出实际访存数组的偏移量：

> 那么，如何将数组下标对应到偏移地址？对一维数组，下标的常数倍（int 型的大小为 4 个字节，倍数为4）即为偏移量。而对于高维数组，我们可以将其视为一个展开成一维的大数组。对于数组 a[d1][d2]...[dn]，访问元素 a[i1][i2]...[in] 可以等价于访问 a[i1*d2*d3...dn + i2*d3*...*dn + ... + in]。在将数组索引翻译成 TAC 时，同学们需要自行将数组下标转换成地址计算指令。这个步骤并不困难，但可能比较繁琐，同学们在实现时要注意细节，避免错误。

我们可以发现上面访问偏移的计算中不涉及 d_1，因此对于编译器来说，最高维度是多少实际上是不太重要的。