stage-6

计13 沈佳茗 2021010745

step10

一、实验内容

1.前端

在 `frontend/parser/ply_parser.py` 中新增语法 `union : function| declaration Semi` ，并实现解析多个 union 的语法，以此完成 Program 允许变量定义 Declaration 节点作为它的孩子。

```
def p_union(p):
    """
    union : function
        | declaration Semi
    """
    p[0] = p[1]

def p_program_union_base(p):
    """
    unions : union
    """
    p[0] = [p[1]]

def p_program_unions(p):
    """
    unions : unions union
    """
    p[1].append(p[2])
    p[0] = p[1]


def p_program_base_function(p):
    """
    functions : function
    """
    p[0] = [p[1]]

def p_program_function(p):
    """
    functions : functions function
    """
    # print(p[1])
    p[1].append(p[2])
    p[0] = p[1]

def p_program(p):
    """
```

```
    program : unions
    """
    p[0] = Program(p[1])
```

然后修改 `frontend/ast/tree.py` 中的 `Program` 节点，将其成员变量 `children` 的类型改为 `List[Union[Function, Declaration]]`。

2. 中端

2.1 语义分析

在 `ScopeStack` 类中增加 `declare_global` 函数用来声明全局变量。

修改 `frontend/typecheck/namer.py` 中的 `visitProgram` 函数，新增对全局变量的检查，修改 `visitDeclaration` 函数，增加对全局变量的处理。

```
@@ -178,15 +181,30 @@ class Namer(Visitor[ScopeStack, None]):
         4. If there is an initial value, visit it.
         """
         sym = ctx.lookup_current(decl.ident.value)
-        # print(sym, ctx.current_scope().symbols)
         if sym != None:
             raise DecafDeclConflictError(decl.ident.value)
         else:
-            new_varsymbol = VarSymbol(decl.ident.value, decl.var_t.type)
-            ctx.declare(new_varsymbol)
-            decl.setattr("symbol", new_varsymbol)
-            if decl.init_expr is not NULL:
-                decl.init_expr.accept(self, ctx)
+            if ctx.isGlobalScope():
+                isInit = False
+                if decl.init_expr is not NULL:
+                    isInit = True
+                new_varsymbol = VarSymbol(decl.ident.value, decl.var_t.type, True,
isInit)
+                ctx.declare_global(new_varsymbol)
+                init = 0
+                if isInit:
+                    if not isinstance(decl.init_expr, IntLiteral):
+                        raise DecafGlobalVarBadInitValueError(decl.ident.value)
+                    decl.init_expr.accept(self, ctx)
+                    init = decl.init_expr.value
+
+                new_varsymbol.setInitValue(init)
+                decl.setattr("symbol", new_varsymbol)
+            else:
+                new_varsymbol = VarSymbol(decl.ident.value, decl.var_t.type, False)
+                ctx.declare(new_varsymbol)
+                decl.setattr("symbol", new_varsymbol)
+                if decl.init_expr is not NULL:
+                    decl.init_expr.accept(self, ctx)
         # raise NotImplementedError
```

2.2 中间代码生成

在 `utils/tac/tacinstr.py` 中增加 `LoadWord, SaveWord, LoadSymbol` 三条新的 `TACInstr`，其中 `LoadWord, LoadSymbol` 组合用来载入全局变量，`SaveWord, LoadSymbol` 组合用来写入全局变量。

在 `frontend/tacgen/tacgen.py` 中的 `TACFuncEmitter` 类中添加 `visitLoadWord, visitSaveWord, visitLoadSymbol` 三个函数用来添加 `LoadWord, SaveWord, LoadSymbol` 三条 TACInstr。修改 `TACGen` 类中的 `visitIdentifier, visitAssignment` 函数，增加对全局变量的判断。

```
@@ -180,8 +193,20 @@ class TACGen(Visitor[TACFuncEmitter, None]):
         """
         1. Set the 'val' attribute of ident as the temp variable of the 'symbol'
attribute of ident.
         """
-        temp = ident.getattr('symbol').temp
-        ident.setattr('val', temp)
+        sym = ident.getattr('symbol')
+        # print(sym, sym.is)
+        if sym.isGlobal:
+            address = mv.freshTemp()
+            mv.visitLoadSymbol(address, sym)
+            temp = mv.freshTemp()
+            mv.visitLoadWord(temp, address, 0)
+            sym.temp = temp
+            ident.setattr('symbol', sym)
+            ident.setattr('val', sym.temp)
+        else:
+        # print(ident.value,ident.getattr('symbol'))
+            temp = ident.getattr('symbol').temp
+            ident.setattr('val', temp)

@@ -218,9 +244,20 @@ class TACGen(Visitor[TACFuncEmitter, None]):
         3. Set the 'val' attribute of expr as the value of assignment instruction.
         """
         expr.rhs.accept(self, mv)
-        temp = expr.lhs.getattr('symbol').temp
-        mv.visitAssignment(temp, expr.rhs.getattr('val'))
-        expr.setattr('val', expr.rhs.getattr('val'))
+        lhs_sym = expr.lhs.getattr("symbol")
+        if lhs_sym.isGlobal:
+            address = mv.freshTemp()
+            mv.visitLoadSymbol(address, lhs_sym)
+            temp = mv.freshTemp()
+            mv.visitAssignment(temp, expr.rhs.getattr('val'))
+            mv.visitSaveWord(temp, address, 0)
+            lhs_sym.temp = temp
+            expr.lhs.setattr('symbol', lhs_sym)
+            expr.lhs.setattr('val', lhs_sym.temp)
+        else:
+            temp = expr.lhs.getattr('symbol').temp
+            mv.visitAssignment(temp, expr.rhs.getattr('val'))
+            expr.setattr('val', expr.rhs.getattr('val'))
```

3. 后端

在 `utils/riscv.py` 中的 class Riscv 中加入与 `LoadWord, SaveWord, LoadSymbol` 对应的 Riscv 指令，实现转换。

在 `backend/riscv/riscvasmemitter.py` 中的 `RiscvAsmEmitter` 类的构造函数中增加对全局变量的声明。

```
@@ -24,13 +28,25 @@ class RiscvAsmEmitter(AsmEmitter):
         callerSaveRegs: list[Reg],
     ) -> None:
         super().__init__(allocatableRegs, callerSaveRegs)
+        self.ctx = GlobalScope


         # the start of the asm code
         # int step10, you need to add the declaration of global var here
+        for name, symbol in GlobalScope.symbols.items():
+            if isinstance(symbol, VarSymbol):
+                if symbol.isInit:
+                    self.printer.println(".data")
+                else:
+                    self.printer.println(".bss")
+                self.printer.println(f".globl {name}")
+                self.printer.println(f"{name}:")
+                self.printer.println(f"    .word {symbol.initValue}")
+
         self.printer.println(".text")
         self.printer.println(".global main")
         self.printer.println("")
```

二、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）。

```
# PIC
# delta = symbol-pc
auipc rd, delta[31:12]+delta[11]
lw rd, delta[11:0(rd)
```

```
# non PIC
# delta = GOT[symbol]-pc
auipc rd, delta[31:12]+delta[11]
addi rd, rd, delta[11:0]
```

step 11

一、实验内容

1. 前端

在 `frontend/ast/tree.py` 类中修改 `Declaration` 类，新增 `is_array` 变量，判断声明的变量是否为数组。修改 `TArray` 类的构造函数，改为 `super().__init__("type_array", ArrayType.multidim(_type, *dims))` 生成对应类型和维数的数组。新增 `ArrayElement` 类表示数组中的元素，包含 `ident, value, indexes` 成员变量。

在 `frontend/parser/ply_parser.py` 修改并新增文法以支持数组。

```
@@ -352,7 +352,7 @@ def p_unary_expression(p):

 def p_binary_expression(p):
     """
-    assignment : Identifier Assign expression
+    assignment : unary Assign expression
     logical_or : logical_or Or logical_and
     logical_and : logical_and And bit_or
     bit_or : bit_or BitOr xor
@@ -399,7 +399,49 @@ def p_brace_expression(p):
     primary : LParen expression RParen
     """
     p[0] = p[2]
-
+
+def p_one_dim_array(p):
+    """
+    one_dim_array : type Identifier LBracklet Integer RBracklet
+    """
+    p[0] = Declaration(TArray(p[1].type, [p[4].value]), p[2], NULL,True,
[p[4].value])
+
+def p_multi_dim_array(p):
+    """
+    multi_dim_array : one_dim_array LBracklet Integer RBracklet
+        | multi_dim_array LBracklet Integer RBracklet
+    """
+    p[1].dims.append(p[3].value)
+    p[1].var_t = TArray(p[1].var_t.type.full_indexed, p[1].dims)
+    p[0] = p[1]
+
+def p_array_declaration(p):
+    """
+    declaration : one_dim_array
+        | multi_dim_array
+    """
+    p[0] = p[1]
+
+def p_one_dim_postfix(p):
+    """
+    one_dim_postfix : Identifier LBracklet expression RBracklet
```

```
+    """
+    p[0] = ArrayElement(p[1], [p[3]])
+
+def p_multi_dim_postfix(p):
+    """
+    multi_dim_postfix : one_dim_postfix LBracklet expression RBracklet
+        | multi_dim_postfix LBracklet expression RBracklet
+    """
+    p[1].indexes.append(p[3])
+    p[0] = p[1]
+
+def p_array_postfix(p):
+    """
+    postfix : one_dim_postfix
+        | multi_dim_postfix
+    """
+    p[0] = p[1]
```

2. 中端

2.1 语义检查

在 `VarSymbol` 类中新增 `is_array, dims` 变量，用来判断当前符号是否是数组以及其维数。

在 `frontend/typecheck/namer.py` 中，修改 `visitDeclaration` ，增加对数组的检查，如果是数组，则检查维度中的每一项是否在 1- MAX_INT 之间，如果不是则报错。修改 `visitUnary, visitBinary` ，单目运算式的 operand 不能是数组，双目运算式两侧不能一个是数组，一个是数（int）。新增 `visitArrayElement` 函数，首先检查 symbol 是否存在，然后检查是数组，接着检查符号与 array_element 的维度是否相同，最后检查是否存在越界以及对每个 array_element.indexes 中的元素进行进一步检查，设置 array_element 的 symbol。

2.2 中间代码生成

在 `frontend/tacgen/tacgen.py` 的 `TACFuncEmitter` 类中增加 `visitArrayElement` 和 `visitAlloc` 函数，分别用来访问数组元素与为数组分配空间。

```
@@ -123,6 +123,15 @@ class TACFuncEmitter(TACVisitor):

     def visitLoadSymbol(self, dst: Temp, global_symbol: VarSymbol) -> None:
         self.func.add(LoadSymbol(dst, global_symbol))
+
+    def visitArrayElement(self, dst: Temp, src: Temp, size: int) -> None:
+        size_temp = self.freshTemp()
+        self.func.add(LoadImm4(size_temp, size))
+        self.func.add(Binary(TacBinaryOp.MUL, size_temp, size_temp, src))
+        self.func.add(Binary(TacBinaryOp.ADD, dst, dst, size_temp))
+
+    def visitAlloc(self, dst: Temp, size: int) -> None:
+        self.func.add(Alloc(dst, size))
```

在 `TACGen` 中修改 `visitDeclaration` 函数，如果是数组则使用 `visitAlloc` 函数在栈上为其分配空间。修改 `visitIdentifier` 函数，首先判断是否是全局变量，如果是，则继续判断是否是数组，如果是数组则载入数组地址作为 val 保存在 ident 中，否则载入地址处的值作为 val。修改 `visitBinary` ，增加对 lhs 是否是数组的判断，如果是则需要根据 indexes 计算相应的地址，并将结果通过 SaveWord 保存。增加 `visitArrayElement` 函数，通过不断计算 index 找到地址，最后载入地址处的值。

```
+    def visitArrayElement(self, array_element: ArrayElement, mv: TACFuncEmitter) ->
None:
+        symbol = array_element.getattr("symbol")
+        address = mv.freshTemp()
+        # print(address)
+        val = mv.freshTemp()
+        if symbol.isGlobal:
+            mv.visitLoadSymbol(address, symbol)
+        else:
+            mv.visitAssignment(address, symbol.temp)
+
+        decaf_type = symbol.type # BuiltIn_type or Array_type
+        for idx in array_element.indexes:
+            # print(idx)
+            idx.accept(self, mv)
+            mv.visitArrayElement(address, idx.getattr("val"), decaf_type.base.size)
+            decaf_type = decaf_type.base
+
+        mv.visitLoadWord(val, address, 0)
+        # print(val, address)
+        array_element.setattr("val", val)
```

3. 后端

修改 `backend/riscv/riscvasmemitter.py` 中的 `RiscvAsmEmitter` 类的构造函数，如果是数组，则在分配空间时使用数组维数的连乘积作为空间大小。在 `RiscvInstrSelector` 类中新增 `visitAlloc` 函数，用来增加一条 Riscv.Alloc 指令。在 `RiscvSubroutineEmitter` 中增加 `emitAlloc` 函数，在栈上分配空间并修改 `nextLocalOffset` 。

在 `backend/reg/bruteregalloc.py` 中的 `localAlloc` 的 for 循环中检查指令是否是 Alloc，如果是的话首先分配一个寄存器作为保存数组基地址的寄存器，然后使用 `emitAlloc` 分配空间。

二、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?

   在三地址码中的 ALLOC 指令修改为 `ALLOC size` ，size 为一个可变值（寄存器）。在翻译为汇编代码的过程中，将数组的首地址保存为 sp，再将 sp 减去 type_size * size 的大小获得新的 sp。之前以 sp 为基准的寻址方式改为以 fp 为基准的寻址方式。

step 12

一、实验内容

1. 前端

在 `frontend/ast/tree.py` 中修改 `Parameter` 节点，增加 `is_arrray, dims` 属性，用来表示参数是否是数组以及数组的维数。新增 `Int_list` 节点，用来保存数组初始化的值（列表）。

由于加入了数组传参，而在参数中的数据第一维可以为空，所以需新增文法以构造第一维为空的数组。此外，由于支持数组初始化，所以构造文法完成 int_list 的构造。

```
@@ -88,15 +88,52 @@ def p_type(p):

 def p_parameter(p):
     """
-    parameter : type Identifier
+    parameter_int : type Identifier
     """
     p[0] = Parameter(p[1], p[2])
+
+def p_parameter_array_empty(p):
+    """
+    parameter_empty_dim : parameter_int LBracklet empty RBracklet
+    """
+    p[1].dims = [0]
+    p[1].var_t = TArray(p[1].var_t.type, p[1].dims)
+    p[1].is_array = True
+    p[0] = p[1]
+
+def p_parameter_array_one_dim(p):
+    """
+    parameter_one_dim : parameter_int LBracklet Integer RBracklet
+    """
+    p[1].dims = [p[3].value]
+    p[1].var_t = TArray(p[1].var_t.type, p[1].dims)
+    p[1].is_array = True
+    p[0] = p[1]
+
+def p_array_parameter_multi(p):
+    """
+    parameter_multi_dim : parameter_one_dim LBracklet Integer RBracklet
+        | parameter_empty_dim LBracklet Integer RBracklet
+    """
+    p[1].expand_dims(p[3].value)
+    p[1].var_t = TArray(p[1].var_t.type.full_indexed, p[1].dims)
+    p[1].is_array = True
+    p[0] = p[1]
+
+def p_parameter_array(p):
+    """
```

```
+    parameter : parameter_empty_dim
+        | parameter_one_dim
+        | parameter_multi_dim
+        | parameter_int
+    """
+    p[0] = p[1]

 def p_parameter_list_base(p):
     """
-    parameter_list_base : type Identifier Coma
+    parameter_list_base : parameter Coma
     """
-    p[0] = Parameter(p[1], p[2])
+    p[0] = p[1]


 def p_parameter_list_prefix_first(p):
@@ -400,6 +437,33 @@ def p_brace_expression(p):
     """
     p[0] = p[2]

+def p_int_list_empty(p):
+    """
+    int_list : empty
+    """
+    p[0] = Int_list([])
+
+def p_int_list_base(p):
+    """
+    int_list : Integer
+    """
+    p[0] = Int_list([p[1].value])
+
+def p_int_list(p):
+    """
+    int_list : int_list Coma Integer
+    """
+    p[1].add_value(p[3].value)
+    p[0] = p[1]
+
+def p_array_init(p):
+    """
+    declaration : one_dim_array Assign LBrace int_list RBrace
+        | multi_dim_array Assign LBrace int_list RBrace
+    """
+    p[1].set_init(p[4])
+    p[0] = p[1]
+
```

2. 中端

2.1 语义分析

在 `frontend/typecheck/namer.py` 中，修改 `visitParameter` ，在设置符号时需要设置 `is_array, dims` 。修改 `visitDeclaration` 函数，增加对初始化值与变量是否符合的检测，若一个是数组、一个不是则报错。修改 `visitBinary` 函数，增加对左式与右式类型是否相同的检测，如果一个是数组、一个不是则报错。

2.2 代码生成

在 `frontend/tacgen/tacgen.py` 中，在 `TACFuncEmitter` 类中新增 `visitFilln` 函数，同来生成调用 `fill_n` 的代码，实现数组的清零。

```
def visitFilln(self, array: Temp, decl: Declaration) -> None:
    address = array
    size = int(decl.var_t.type.size / 4)
    zero_temp = self.visitLoad(0)
    size_temp = self.visitLoad(size)
    self.visitParam(address)
    self.visitParam(zero_temp)
    self.visitParam(size_temp)
    label = FuncLabel("fill_n")
    ret_temp = self.freshTemp()
    self.func.add(CALL(ret_temp, label, [address, zero_temp, size_temp]))
```

修改 `visitDeclaration` 函数，增加对数组是否具有初始值的检查，如果有初始值，则调用 `visitFilln` 清空数组，再使用 `load, save` 对数组进行初始化。

3. 后端

本次实验中仅对全局数组的初始化进行了改动，在初始化时先写入对应数量的初始值 `.word {val}` ，未初始化的地方用 0 填充 `.zero {4 * (prod(symbol.dims) - len(symbol.initValue))}` 。

二、思考题

1. 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？

   在 C/C++ 中，对数组的访问是基于数组的首地址加上偏移来寻址访问的，数组作为参数时，传入的仅为地址，函数体不需要为其分配空间，所以不需要所有维的维数来计算空间大小，但计算偏移量时需要知道除了第一维的其余维数。但第一维的维数不需要知道，在运行时得到即可，所以类型检查时不需要第一维的信息，可以当作第一维是空的情况处理。