STAGE5：数组 实验报告

邢竞择 2020012890

1 Step11

1.1 数组设计

本步骤主要需要支持 1）数组的定义 2）数组的索引，整体上我采用了 type::Type 嵌套的方式来表示数组类型，并额外设计了以下语法树节点。

- ast::ArrayType （继承 ast::Type ），它将为 ast::Lvalue 提供类型指引，它包括如下成员
  - ast::Type *base： 由于数组的维度信息和基本类型信息在源代码中不是连续的，所以记录下来基本类型，在 build_sym 中调用 setBaseType 函数来设置数组基本类型（当然对于本实验只需要支持 int 数组，其实不需要实现这一方法）
- ast::ArrayIndex： 支持数组索引，只记录一维的索引，通过嵌套来索引整个数组，它包含以下成员
  - ArrayIndex *lower： 指向下一级索引，在我的实现中 lower 是更低的维度
  - Expr *offset： 这一级索引的表达式
  - type::Type *ATTR(dim)： 记录了当前维度以及更低维度的数组形状信息，数组最高维的 ATTR(dim) 在 type_check 中从符号表中获取，再递归地设置给各个低维索引节点。在这一设置过程中能方便地检查索引是否符合数组维度，也能快捷地将数组展开成一维。
  - tac::Temp ATTR(offset)： 计算出的展开成一维后的索引
- ast::ArrayRef： 所有数组元素访问都会由这个节点处理，它包含以下成员
  - ArrayIndex *index： 索引信息
  - symb::Variable *ATTR(sym)： 符号信息
  - std::string name： 变量名

对于存放在栈中的数组，将它的首指针存在 Variable 的 Temp 中，方便 ArrayRef 使用。接下来具体描述各阶段的改动。

1.2 前端

添加两类非终结符

```
%nterm<mind::type::Type*> ArrayDim
%nterm<mind::ast::ArrayIndex*> ArrayIndex
```

分别表示数组维度和数组索引

1.3 中端

在 type_check 阶段检查索引符合要求的 visit 函数如下

```
void SemPass2::visit(ast::ArrayIndex *a) {
    a->offset->accept(this);
    if (!a->ATTR(dim)->isArrayType()) {
```

```
        issue(a->getLocation(), new NotArrayError());
        return;
    }
    ArrayType *p = static_cast<ArrayType *>(a->ATTR(dim));
    if (a->lower == NULL) { // lowest dim
        if (!p->getElementType()->isBaseType())
            issue(a->getLocation(), new NotArrayError());
        return;
    } else {
        a->lower->ATTR(dim) = p->getElementType();
        a->lower->accept(this);
    }
}
```

在 translation 中，将索引展开成一维的 visit 函数如下

```
void Translation::visit(ast::ArrayIndex *idx) {
    if (idx->lower != NULL)
        idx->lower->accept(this);
    idx->offset->accept(this);
    ArrayType *at = static_cast<ArrayType *>(idx->ATTR(dim));
    Temp newdim = tr->genLoadImm4(at->getElementType()->getSize());
    if (idx->lower != NULL)
        idx->ATTR(offset) =
            tr->genAdd(tr->genMul(idx->offset->ATTR(val), newdim),
                       idx->lower->ATTR(offset));
    else
        idx->ATTR(offset) = tr->genMul(idx->offset->ATTR(val), newdim);
}
```

还需要修改 ast::AssignExpr ，若左值是一个数组元素，则直接将结果写到内存中；修改 ast::LvalueExpr ，对于数组元素，需将值从内存中取出。

添加了三地址码 ALLOC ，它的参数是一个 Temp 和一个 int ，表示在栈上开辟一定的空间，并将这段空间的开头位置存入 Temp 中，这个 Temp 应当在此 TAC 处被首次定义。

1.4 后端

添加了 ALLOC 的翻译函数

```
void RiscvDesc::emitAllocTac(Tac *t) {
    addInstr(RiscvInstr::ADDI, _reg[RiscvReg::SP], _reg[RiscvReg::SP], NULL,
             -t->op1.ival, EMPTY_STR, NULL);
    int r0 = getRegForWrite(t->op0.var, 0, 0, t->LiveOut);
    addInstr(RiscvInstr::MOVE, _reg[r0], _reg[RiscvReg::SP], NULL, 0, EMPTY_STR,
             NULL);
}
```

1.5 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?

   答: 需要在 parser 中需要支持用表达式定义数组维度，并且在 build_sym 阶段检查用表达式定义维度的数组是一维的。三地址码 ALLOC 本身的逻辑就是将 SP 减小指定的大小，所以自动支持新的定义。在生成汇编码时，需要在函数开始就减小 SP，给临时变量留足空间，修改后需要使用 BP 寄存器作为临时变量的基地址，这样在新开数组的时候就可以避免覆盖掉存在栈上的临时变量了。

2 Step12

2.1 前端

为了支持初始化，添加了 `typedef util::List<int> Initializer` 来储存用于初始化的数列。

```
ArrayInitRecurse  :
                      { $$ = new ast::Initializer(); }
                  | ArrayInitRecurse ICONST COMMA
                      { $1->append($2); $$ = $1; }
                  ;
ArrayInit :       ArrayInitRecurse ICONST
                      { $1->append($2); $$ = $1; }
```

2.2 中端

初始化的工作自然要交给 `ast::VarDecl` 来完成，故给 `ast::VarDecl` 添加成员 `Initializer *arrayinit` 来存放数组的初始化值。此外，由于全局数组在后端处理，而后端只能访问到符号表，故需要在 `Variable` 中添加成员 `Initializer *arrinit` 以及相关的设置和读取方法。

若数组被作为函数的实参，则它必定被解析为 `ast::VarRef` 节点，相应的翻译发生在 `ast::LvalueExpr` 中，所以我对 `ast::LvalueExpr` 的处理逻辑进行了较大改动。

- 对于全局变量，如果是普通变量，则生成 `LOAD_SYMBOL` 与 `LOAD`，将值从数组加载到寄存器中；如果是数组变量，则仅生成一个 `LOAD_SYMBOL` 即可得到其首地址作为参数
- 对于非全局变量，`Variable` 对于普通变量保存其数值，对于数组变量保存其首指针，故只需使用 `Variable` 中存的值即可

```
void Translation::visit(ast::LvalueExpr *e) {
    e->lvalue->accept(this);
    ast::Lvalue *lv = e->lvalue;
    if (lv->ATTR(lv_kind) == ast::Lvalue::SIMPLE_VAR) {
        ast::VarRef *var = static_cast<ast::VarRef *>(lv);
        // a variable or ptr of an array
        ast::VarRef *var = dynamic_cast<ast::VarRef *>(lv);
        if (var->ATTR(sym)->isGlobalVar()) {
            tr->genLoad(var->ATTR(sym)->getTemp(), tr->genLoadSymbol(var->var),
```

```
                        0);
            if (var->ATTR(type)->isBaseType()) {
                e->ATTR(val) = tr->getNewTempI4();
                tr->genLoad(e->ATTR(val), tr->genLoadSymbol(var->var), 0);
            } else {
                e->ATTR(val) = tr->genLoadSymbol(var->var);
            }
        } else {
            e->ATTR(val) = var->ATTR(sym)->getTemp();
        }
        e->ATTR(val) = var->ATTR(sym)->getTemp();
    } else {
        // an array element
        ast::ArrayRef *var = static_cast<ast::ArrayRef *>(lv);
        e->ATTR(val) = tr->getNewTempI4();
        if (var->ATTR(sym)->isGlobalVar()) {
            Temp baseptr = tr->genLoadSymbol(var->var);
            Temp ptr = tr->genAdd(baseptr, var->index->ATTR(offset));
            tr->genLoad(e->ATTR(val), ptr, 0);
        } else {
            Temp ptr =
                tr->genAdd(var->ATTR(sym)->getTemp(), var->index->ATTR(offset));
            tr->genLoad(e->ATTR(val), ptr, 0);
        }
    }
}
```

此外，带初始化的局部数组也在 translation 中生成初始化指令，具体而言就是调用 fill_n 再进行数次 STORE ，具体步骤如下

```
        if (decl->arrayinit != NULL) {
            tr->genParam(baseptr, 0);
            tr->genParam(tr->genLoadImm4(0), 1);
            tr->genParam(tr->genLoadImm4(t->getSize() / 4), 2);
            Label dst = tr->getNewLabel();
            dst->str_form = std::string("fill_n");
            tr->genCall(dst);
            int offset = 0;
            for (auto it = decl->arrayinit->begin();
                 it != decl->arrayinit->end(); it++) {
                tr->genStore(tr->genLoadImm4(*it), baseptr, offset);
                offset += 4;
            }
        }
```

2.3 后端

全局数组的定义与初始化在此完成，对于无初始化或初始化数值全为 0 的数组，将其放在 `.bss` 段，只需说明其占用的空间即可；对于其他情况，则放在 `.data` 段，手动添加 0 补满空间。

2.4 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?

   答：在实现中，不难发现将多维数组索引转化成一维的并不需要第一维的大小信息，C++实际上只需要数组的首地址以及除第一维以外的维度信息，即可正常编译函数内的全部数组使用，因此只需要知道第一维存在，而其大小是无用信息，可以舍弃。