Array Implementation

- The key insight here is that arrays can be viewed as **modifiers** to some primitive type such as int or float, e.g. int[10]
- This is expressed with the grammar rules:

```plaintext
data_type : primitive_type
           | primitive_type [ INTEGER ]

primitive_type : int
                | float
                | string
```
Array Implementation

- We also need to allow for array initializers of the form `int[2] a = {1,2}` in addition to the scalar initializers
The last thing we need to address are the contexts array expression can appear in:

- Left hand side of an assignment statement
- Within an expression

We do this with the idea of a `storable`:

```
stmt : (data_type | void) ID \( \text{formal_args? \)} \) stmt
    | data_type ID initializer? ;?
    | ID \( \text{actual_args? \)} ;?
    | storable = exp ;?
    | get ID ;?
    | put exp ;?
    | return exp? ;?
    | while (exp) stmt
    | if (exp) stmt (else stmt)?
    | \{ stmt_list \}
```

```
primary : INTEGER
    | FLOAT
    | STRING
    | ID \( \text{actual_args? \)}
    | storable
    | \( \text{exp \)}
    | - primary
    | not primary
```

```
storable : ID
    | ID [ exp ]
    | ID \( \text{actual_args? \)} [ exp ]
```
The Frontend

This grammar can easily be transformed into an LL(1) by factoring common prefixes.

Listing 12.1: A grammar for the Cuppa5 language.

```
stmt_list : (stmt)*
stmt : (data_type | void) ID \(\{\) formal_args? \(\}\) stmt
    | data_type ID initializer? ;?
    | ID \(\{\) actual_args? \(\}\) ;?
    | storable \{\} exp ;?
    | get ID ;?
    | put exp ;?
    | return exp? ;?
    | while \{\} exp \} stmt
    | if \{\} exp \} stmt (else stmt)?
    | \} stmt_list \}
data_type : primitive_type
    | primitive_type [ INTEGER ]
primitive_type : int
    | float
    | string
initializer : = exp
    | = \{\} exp (;, exp)* \}
storable : ID
    | ID \} exp \}
    | ID \{\} actual_args? \} \} exp \}
exp : exp_low
exp_low : exp_med ((= | =<) exp_med)*
exp_med : exp_high ((\+ | -) exp_high)*
exp_high : primary ((\* | /) primary)*
primary : INTEGER
    | FLOAT
    | STRING
    | ID \{\} actual_args? \}
    | storable
    | \} exp \}
    | - primary
    | not primary
formal_args : data_type ID (, data_type ID)*
actual_args : exp (, exp)*
ID : <any valid variable name>
INTEGER : <any valid int number>
FLOAT : <any valid floating point number>
STRING : <any valid quoted str constant>
```
Array Types

- We expand our notion of type tuples with the introduction of array types.
- We have to capture the nuances of array types, the type
  \[ \text{int}[10] \]
  is different from the type
  \[ \text{int}[20] \]
  and is certainly different from the type
  \[ \text{int} \]
Array Types

- Adding array types to our type system gives us
  - (‘INTEGER_TYPE’,)
  - (‘FLOAT_TYPE’,)
  - (‘STRING_TYPE’,)
  - (‘VOID_TYPE’,)
  - (‘FUNCTION_TYPE, <return-type>, <list-of-formal-arg-types>’)
  - (‘ARRAY_TYPE’, <elem-type>, <size>){}
int[2] a = {1,2};
a[0] = a[1];
Type Checking

- We have to extend our Cuppa4 type checker in order to include arrays.

```python
def safe_assign(target, source):
    # array types are structured types. there is no nice way to do lookups
    # in a table so we have to compute if it safe to assign.
    if target[0] == 'ARRAY_TYPE' and source[0] == 'ARRAY_TYPE':
        (ARRAY_TYPE, ttype, (SIZE, tsize)) = target
        (ARRAY_TYPE, stype, (SIZE, ssize)) = source
        # compare base types and size -- have to be exactly the same!
        if ttype == stype and tsize == ssize:
            return True
        else:
            return False
    else:
        # check for regular operations
        supported(target)
        supported(source)
        return _safe_assign_table.get(target[0]).get(source[0])
```
def arraydecl_stmt(node):
    (ARRAYDECL, (ID, name), type, (LIST, init_val_list)) = node
    (ARRAY_TYPE, base_type, (SIZE, size)) = type

    if not size > 0:
        raise ValueError("illegal array size")

    if len(init_val_list) != size:
        raise ValueError("array size {} and length of initializer {} don't agree".format(size, length(init_val_list)))

    # walk through initializers and make sure they are type safe
    for ix in range(size):
        ti = walk(init_val_list[ix])
        if not safe_assign(base_type, ti):
            raise ValueError("type {} of initializer is not compatible with declaration type {}".format(ti[0], base_type[0]))

    symtab.declare(name, type)

    return None
Type Checking

```python
def assign_stmt(node):
    (ASSIGN, storable, exp) = node

    ts = walk(storable)
    te = walk(exp)

    if not safe_assign(ts, te):
        raise ValueError("left type {} is not compatible with right type {}"
                         .format(ts[0], te[0]))

    return None
```

```python
def array_access_exp(node):
    (ARRAY_ACCESS, array_exp, (IX, ix)) = node

    type = walk(array_exp)
    ix_type = walk(ix)

    if type[0] != 'ARRAY_TYPE':
        raise ValueError("{} not an array".format(name))

    if ix_type[0] != 'INTEGER_TYPE':
        raise ValueError("array index has to be of type INTEGER_TYPE")

    (ARRAY_TYPE, base_type, size) = type

    return base_type
```
Interpretation

```python
def arraydecl_stmt(node):
    (ARRAYDECL, (ID, name), array_type, (LIST, init_val_list)) = node
    # we use the memory allocated for the list of initializers
    # as the memory for the array in the symbol table.
    # therefore we bind the list into the symbol table as
    # part of the declaration
    # Note: we only bind actual Python values into the symbol table,
    # therefore we need to convert the init_val_list into a list of values.
    symtab.declare(name,
      ('ARRAYVAL',
       array_type,
       ('LIST', value_list(init_val_list))))
    return None

def assign_stmt(node):
    (ASSIGN, storable, exp) = node
    update_storable(storable, exp)
    return None

def array_access_exp(node):
    (ARRAY_ACCESS, array_exp, (IX, ix)) = node
    (tarray, varray) = walk(array_exp)
    (tix, vix) = walk(ix)
    (ARRAY_TYPE, base_type, (SIZE, size)) = tarray
    if vix < 0 or vix > size-1:
        raise ValueError("array index {} out of bounds".format(vix))
    return (base_type, varray[vix])
```
Storables

```python
def location(storable):
    ...
    we are interested in the locations of storable because we
    perhaps want to update them. we have two categories
    of locations for storables:
    a[i] -- the is a memory access of the array a
    a -- we are referencing the storable by name (id)
    ...

    if storable[0] == 'ARRAY_ACCESS':
        # memory access
        (ARRAY_ACCESS, name_exp, (IX, ix)) = storable
        (tmemory, memory) = walk(name_exp)
        (t, offset) = walk(ix)
        return ('LOCATION', ('MEMORY', (tmemory, memory)), ('OFFSET', offset))
    else:
        # access via name
        (ID, name) = storable
        return ('LOCATION', ('ID', name), ('NIL',))

def update_storable(storable, exp):
    ...  
    update a storable location with the value of exp
    ...
    # evaluate source
    (t, v) = walk(exp)
    # get information about target
    (LOCATION, location_type, offset) = location(storable)
    if location_type[0] == 'MEMORY':
        # we are copying a value into a single element, e.g.
        # a[i] = x
        (MEMORY, (tmemory, memory)) = location_type
        (ARRAY_TYPE, base_type, (SIZE, size)) = tmemory
        if offset[1] < 0 or offset[1] > size-1:
            raise ValueError("array index {}[[i]] out of bounds".
                .format(name, offset))
        # update memory location of array
        memory[offset[1]] = v
    elif location_type[0] == 'ID':
        # we are copying value(s) based on name, e.g.
        # a = x
        (ID, name) = location_type
        val = symtab.lookup_sym(name)
        if val[0] == 'CONST':
            # id refers to a scalar, copy scalar value
            (CONST, ts, (VALUE, value)) = val
            symtab.update_sym(name, ('CONST', ts, ('VALUE', coerce(ts, t)(v))))
    elif val[0] == 'ARRAYVAL':
        # id refers to an array, copy the whole array
        (ARRAYVAL, ts, (LIST, smemory)) = val
        # we are copying the whole array
        # Note: we don't want to loose the reference to our memory
        # so we are copying each element separately
        (ARRAY_TYPE, base_type, (SIZE, size)) = ts
        # Note: we could use Python shallow array copy here but
        # this makes it explicit that we are copying elements.
        # we CANNOT copy Python list reference because then both
        # arrays in Cuppa5 would share the same memory.
        for i in range(size):
            smemory[i] = v[i]
        else:
            raise ValueError("internal error on {}".format(val))
    else:
        raise ValueError("internal error on {}".format(location_type))
```

The call-by-reference for arrays is implemented in the `declare_formal_args` function.
Test Driving the Interpreter

```python
lutz$ cat array_simple.txt
int[3] a = {1,2,3};
int[3] b;
put a;
b = a;
b[1] = -a[1];
put b;
lutz$ python3 cuppa5_interp.py array_simple.txt
[1, 2, 3]
[1, -2, 3]
lutz$

$ cat fun_assign.txt
{
    return a;
}

int[3] c = {1,2,3};
ident(c)[1] = 0;
put c;

$ python3 cuppa5_interp.py fun_assign.txt
[1, 0, 3]
$