Arrays

1

Arrays in CPRL

- CPRL supports one-dimensional array types.
  - indices are integer values
  - index of the first element in the array is 0
  - arrays of arrays can be declared
- An array type declaration specifies
  - the array type name (an identifier)
  - the number of elements in the array, which must be an integer literal or constant
  - the type of the elements in the array
- Examples

```
type T1 = array[100] of Integer;
type T2 = array[10] of T1;
```


2

Using CPRL Arrays

- To create array objects, you must first declare an array type and then declare one or more variables of that type.
- Examples

```
type T1 = array[100] of Integer;
type T2 = array[10] of T1;
var a1 : T1;  // contains 100 integers; indexed from 0 to 99
var a2 : T2;  // contains 10 arrays of integers;
              // indexed from 0 to 9
...

a1[0]     // the integer at index 0 of a1 (the first integer)
a2[3]     // the array at index 3 of a2 (the fourth array)
a2[4][3]  // the integer at index 3 of the array
          // at index 4 of a2
```


3

Type Equivalence for Arrays
(Name Equivalence versus Structural Equivalence)

- Array objects in CPRL are considered to have the same type only if they have the same type name. Thus, two distinct array type definitions are considered different even though they may be structurally identical. This is referred to as "name equivalence" of types.
- Two array objects with the same type are assignment compatible. Two array objects with different types are not assignment compatible, even if they have identical structure.

4

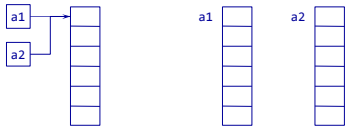
Examples: Array Assignment

```
type T1 = array[100] of Integer;
type T2 = array[10] of T1;
type T3 = array[100] of Integer;

var a1  : T1;
var a1x : T1;   // a1x has the same type as a1
var a2  : T2;
var a2x : T2;   // a2x has the same type as a2
var a3  : T3;   // a3 does not have the same type as a1
...

a1 := a1x;   // legal (same types)
a2 := a2x;   // legal (same types)
a1 := a3;    // *** Illegal in CPRL (different types) ***
```


5

Reference Semantics versus Value Semantics

- CPRL uses value semantics for assignment of arrays.
  (Java uses reference semantics.)
- Example: `a1 := a2`



reference semantics
(references are copied)

value semantics
(array values are copied)

Consider the effect of modifying `a2[0]` after the assignment.

6

Additional Examples: Array Assignment

```
type T1 = array[100] of Integer;
type T2 = array[10] of T1;
var  x, y : T2;
...

x := y;               // array assignment (type T2)
                      // copies 1000 integers (4000 bytes)

x[2] := y[5];         // array assignment (type T1)
                      // copies 100 integers (400 bytes)

x[2][7] := y[5][0]    // Integer assignment
                      // copies 1 integer (4 bytes)
```


7

Passing Arrays as Parameters

- As with parameters of other (non-structured) types, array parameters have semantics similar to assignment.
- Passing an array as a value parameter will allocate space for and copy the entire array.
  - can be inefficient use of memory if you don't actually need to copy the entire array
- Passing an array as a variable (`var`) parameter will simply allocate space for the address of the array.
  - has semantics similar to that of Java

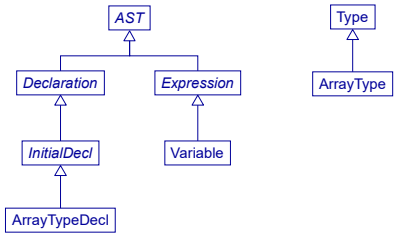
8

Grammar Rules Relevant to Arrays

```
initialDecl = constDecl | arrayTypeDecl | varDecl .

arrayTypeDecl = "type" typeId "=" "array" "["
    intConstValue "]" "of" typeName ";" .

typeName = "Integer" | "Boolean" | "Char" | typeId .

variable = ( varId | paramId ) ( "[" expression "]" )* .
```


9

Relevant AST Classes and Auxiliary Classes



10

Relevant Parser Methods

- `fun parseInitialDecl()   : InitialDecl?`
- `fun parseArrayTypeDecl() : ArrayTypeDecl?`
- `fun parseTypeName()      : Type`
- `fun parseVariable()      : Variable?`

11

Class `ArrayType`

- An array type declaration creates a new type – an array type.
- Class `ArrayType` encapsulates the properties of an array type.
  - `name` – the name of the array type
  - `numElements` – the number of elements in the array type
  - `elementType` – the element type (type of elements in the array)
  - `size` – the size (number of bytes) of a variable with this type
    (computed as `numElements*elementType.size`)

12

Address of an Array Object

- The relative address for a variable of an array type is the relative address of the first byte in the array.
- The relative address for the element of the array at index n is the sum of the relative address of the array plus the offset of nth element, computed as follows:

```
relAddr(a[n]) = relAddr(a) + n*elementType.size
```

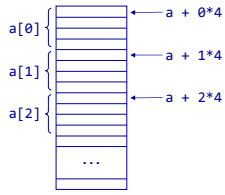
- Note that if the element type of the array is Boolean, then the relative address for the element at index n can be simplified to

```
relAddr(a[n]) = relAddr(a) + n
```


13

Index Example

```
type T = array[100] of Integer;
var a : T;
```



If the actual memory address of a is 60, then the actual address of `a[0]` is 60, the actual address of `a[1]` is 64, the actual address of `a[2]` is 68, etc.

14

Constraint Rules for Arrays

- Array Type Declaration
  - Type Rule: The constant value specifying the number of items in the array must have type Integer, and the associated value must be a positive number.
- Variable (and therefore also for NamedValue)
  - Index expressions are permitted only for variables with an array type.
  - Each index expression must have type Integer.

15

Method `checkConstraints()` for Class `Variable`

- Consider the following declarations:

```
type T is array[10] of Integer;
var a : T;
```

- Observation: `a` has type `T`, but `a[i]` has type `Integer`
- For each index expression, `checkConstraints()` must
  - check that the type of the index expression is `Integer`
  - check that the type of the variable is an array type
  - set the type of the expression to the element type for the array

16

Method `emit()` for Class `Variable`

- First, as with non-array types, `emit()` must generate code to leave the relative address of the variable on the stack (i.e., the address of the first byte of the array)
  - no change required to existing code
- Then, for each index expression, `emit()` must
  - generate code to compute the value of the index expression

```
expr.emit()
```

  - generate code to multiply this value by the element type's size

```
emit("LDCINT ${arrayType.elementType.size}")
emit("MUL")
```

  - generate code to add the result to the relative address of the variable

```
emit("ADD")
```


17

Method `emit()` for Class `Variable` (continued)

- As an optimization, don't generate code for the second step above if the array's element type has size 1 (e.g., if the element type is `Boolean`).

18