5.- Value Types

- Dafny type system distinguishes value and reference types (as in C#) where reference types are pointers, whose referent is dynamically heap-allocated.
- Value types are types whose members represent some information that does not depend on the state of the heap (inmutable), whereas objects have a state (represented in the heap) that is mutable.
- Values have a mathematical flair: they cannot be modified once they are created.

- Value types can be stored in fields (i.p. var) on the heap, and used in real code in addition to specifications.
- Variables that contain a value type can be updated to have a new value of that type.
- Dafny's built-in value types comprise
  - integers (implemented as arbitrary-precision integers), natural numbers, characters, booleans, reals (implemented as quotients of arbitrary-precision integers), and
  - sequences, sets, multisets, maps (both finite and infinite). This are provided mainly to be used in specification context.

Ghost values

- Specifications and ghost constructs are used only during verification; the compiler omits them from the executable code.
- A field x of some type T can be declared to be a ghost field as:

```
ghost var x: T;
```

- User do not need to worry about the impact on performance that manipulating ghost variables would otherwise have.
- Also parameters and results of methods can be declared to be a ghost by preceding the declaration with the keyword `ghost`.

- A ghost variable is useful for computing a value which allows to specify some interesting property, but that value is not really needed in the real code. For example:
  - a ghost variable could allows us to (easily) specify and prove a property.
  - termination proofs
  - to specify class invariants in OO programming
  - etc.
- Non-ghost variables cannot be calculated in terms of a ghost variable.
- In general, real code (boolean conditions of if, while, ...) cannot depend on ghost variables.

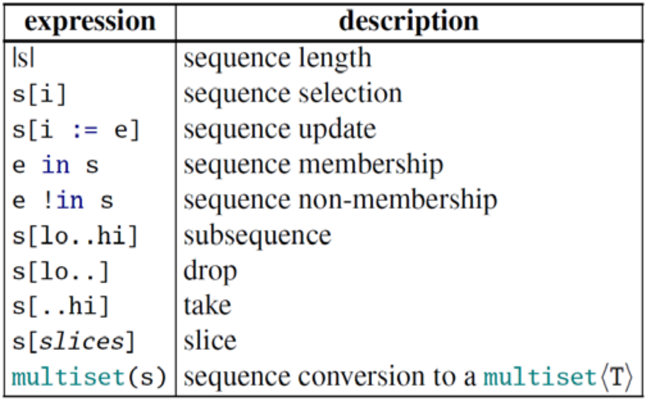
Sequences

- Sequences are an immutable value type.
- A sequence can be formed using an ordered list of expressions enclosed in square brackets.

```
        [3, 1, 4, 1, 5, 9, 3] [4+2, 1+5, a*b]
```

- For any type T, a value of type seq<T> denotes a sequence of elements of type T, that is, a mapping from a finite set of consecutive natural numbers (called indicies) to T values.

```
    []                          // empty sequence
    s + r                       // concatenation
    ≤                           // prefix
    <                           // proper prefix
```



expression	description
|s|	sequence length
s[i]	sequence selection
s[i := e]	sequence update
e in s	sequence membership
e !in s	sequence non-membership
s[lo..hi]	subsequence
s[lo..]	drop
s[..hi]	take
s[*slices*]	slice
multiset(s)	sequence conversion to a multiset⟨T⟩

```
predicate sorted(s: seq<int>)
{
   ∀ i,j • 0 ≤ i < j < |s| ⟹ s[i] ≤ s[j]
}
```

A generic predicate (requiring equality)

```
predicate palindrome<T(=)> (s:seq<T>)
{
  // ∀ i • 0 ≤ i ≤(|s|-1)/2 ⟹ s[i]=s[|s|-i-1]
  ∀ i • 0 ≤ i < |s| ⟹ s[i] = s[|s|-1-i]
                          //easier to understand and to validate
}
```

A recursive/inductive definition

```
function sum(v: seq<int>): int
{
if v = [] then 0 else v[0] + sum(v[1..])
}
```

An iterative algorithm

```
method vector_Sum(v:seq<int>) returns (x:int)
//ensures x = sum(v)
{
var n := 0 ;
x := 0;
while n ≠ |v|
    invariant 0 ≤ n ≤ |v|
    {
    x, n := x + v[n], n + 1;
    }
}
```

Structural induction on sequences: To prove that any sequence `s: seq<T>` satisfies some proprety `P`:

- Base step: show that `P([])` holds.
- Inductive step: show that for all `x: T`: `P([[x]+s)` holds, provided that the induction hypothesis that `P(s)` holds.

```
lemma left_sum_Lemma(r: seq<int>, k: int)
requires 0 ≤ k < |r|
ensures sum(r[..k]) + r[k] == sum(r[..k+1])
```

Structural induction on non-empty sequences: To prove that any non-empty sequence `s: seq<T>` satisfies some proprety `P`:

- Base step: show that `P([x])` holds for all `x: T`.
- Inductive step: show that for all `x: T`: `P([[x]+s)` holds, provided that the induction hypothesis that `P(s)` holds for non-empty `s`.

- General structural induction on sequences:
  - Base step(s) depending on the inductive case(s).
  - Inductive step(s): show that `P(s)` holds, provided that the induction hypothesis (one or more) `P(s1)`, `P(s2)`, ... hold for any `s1`, `s2`, ... that are simpler than `s`.
- Induction on the length of a sequence:
  - Base step(s) (depending on the inductive case(s)) for length 0, 1, etc
  - Inductive step(s): show that `P(s)` holds, provided that the induction hypothesis (one or more) `P(s1)`, `P(s2)`, ... hold for any `s1`, `s2`, ... that are shorter than `s`.
- The idea to set up inductive and base steps is the same as recursive and simple cases in a recursive algorithm.
- Termination should be ensured in both: algorithms and lemma proofs.

Sets

- Sets are an immutable value type.
- For any type T, a value of type set<T> is a finite set of elements/values of type T.
- A set can be defined by a *set display expression*

```
      {}.  {2, 70, 15, 23} {43+25, 10+5, (a*b)+c}
```

- Operations on sets:

```
     s1 + s2          // set union
     s1 - s2          // set difference
     s1 * s2          // intersection
     |s|              // cardinality
     e in s           // membership
```

- Boolean (relational) operations: equality (==), disequality (!=), subset (<=), superset (>=), proper subset (<), proper superset (>), disjointness (!!).
- Set comprenhesions: `set x: T | P(x) • f(x)`
  - `T` can be inferred, so it can be omitted.
  - This mechanism has the potential to create an infinite set, which is not allowed in Dafny. To prevent this, Dafny employs heuristics in an attempt to prove that that the resulting set will be finite.
  - If `f` is identity: `set x: T | P(x)`

Multisets

- Multisets are like sets in almost every way, except that they keep track of how many copies of each element they have.
- The multiset type is almost the same as sets: `multiset<T>` declares a finite multiset of elements of type `T`.
- The operations defined on sets are also available for multisets, taken into account multiplicity of elements in union and difference.
- Multisets can be created from both sequences by using `multiset` with parentheses:

```
multiset([3,1,1,5,5]) == multiset{3,1,1,5,5};
```

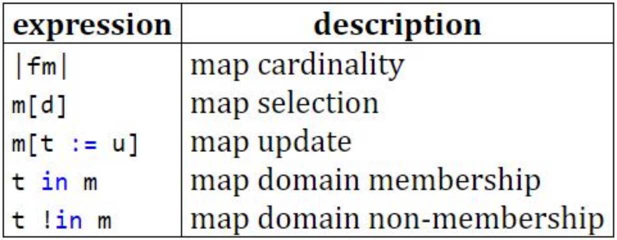
- Multiplicity of `e` in `s`: `s[e]`

Maps

- For any types T and U, a value of type map<T,U> denotes a finite map from T to U.
- It is a look-up table indexed by T.
- The domain of the map is a finite set of T values that have associated U values.
- A map can be formed using a map display expression, which is a possibly empty, ordered list of *maplets* of the form t := u where t is an expression of type T and u is an expression of type U, enclosed in square brackets after the keyword map.

```
map[]
map[20 := true, 3 := false, 20 := false]
map[a+b := c+d]
```

- If the same key occurs more than once, only the last occurrence appears in the resulting map.



expression	description
`|fm|`	map cardinality
`m[d]`	map selection
`m[t := u]`	map update
`t in m`	map domain membership
`t !in m`	map domain non-membership

- Dafny has built-in syntax for domain and values of a map:
  - `m.Keys` returns the set of keys in the map `m`, aka domain. (`x in m.Keys` can be abbreviated to `x in m`).
  - `m.Values` returns the set of values in the map `m`, aka range.
- Finite maps can be also used in non-ghost contexts:

```
method containsValue(m: map<int,char>, val: char)
                              returns (b: bool)
ensures b ⟺ ∃ i • i in m ∧ m[i] = val;
{
return val in m.Values;
}
```

- A value of type `imap<T,U>` denotes a (possibly) infinite map.
- Map comprehension expression: defines a finite or infinite map value by defining a domain (using the optional condition following the |) and for each value in the domain, giving the mapped value using the expression following the "::"

```
m := map x : int | 0 ≤ x ≤ 10 • x * x;

im1 := imap x : int | x%2 = 0 • x * x;

im2 := imap x : int • x * x;
```

- Map comprehensions: `map x: T | P(x) • f(x)`