

Spec#

Ana Paiva

apaiva@fe.up.pt www.fe.up.pt/~apaiva

Tuples

```
<int, String, int> date = <27, "Oct", 2003>
```

```
<string,int> pythagoras = <"Pythagoras", 2582>;
```

```
<string,int> aristoteles = "Aristoteles" :> 2387;
```

Access

```
aristoteles.First // "Aristoteles"
pythagoras.Second // 2582
date.Third // 2003
```

Patterns

```
let <n, y> = pythagoras; // n = "Pythagoras", y = 2582
```

```
WriteLine(n + ", " + y); // writes "Pythagoras, 2582"
```

Constructor patterns

```
Cons(h,t) // example pattern
```

matches this expression

```
Cons(23,Cons(1,Nil())) // matching expression
```

Constructor patterns match from left to right and outermost to innermost.

In this example, h matches 23 and t matches Cons(1,Nil()).

Universal pattern

The underscore `_` is the universal pattern. It matches anything and binds nothing. It is used as the default case, or as a placeholder in constructor patterns.

Set

Operators

Boolean operators.

```
x in S // x is an element of S
```

```
x notin S // x is not an element of S
```

```
S < T // S is a proper subset of T
```

```
S <= T // S is a subset of T, or is equal to T
```

```
S > T // S is a proper superset of T
```

```
S >= T // S is a proper superset of T, or is equal to T
```

Returning a new set.

```
S + T // union of S and T
```

```
S * T // intersection of S and T
```

```
S - T // difference of S and T
```

Properties

```
S.Size // number of elements in set S
```

```
S.IsEmpty // S is empty
```

Set

- **Updates:** Indexers for sets use Booleans as argument to include and exclude elements. In the following examples, S and T are sets, and x and y are elements:
`S[x] = true;` // add element x to set S
`T[x] = false;` // remove element x from set T
- Attempting to remove an element which is not in the set has no effect (it is not an error).
- A set can be updated with another set.
`S += T;` // add elements of T to S
`S -= T;` // remove elements of T from S

Set

- **Instance methods:** These methods are defined for sets. Both choose methods are nondeterministic
`S.Add(x)` // return new set like S but with element x added
`S.ChooseSubset()` // return subset of elements of S, possibly empty
`S.ChooseNonEmptySubset()` // return subset of elements of S, not empty
- **Static methods:** In the following examples SS is a set of sets. Each method returns a set.
`BigUnion(SS)` // return union of all the sets in SS
`BigIntersection(SS)` // return intersection of all the sets in SS

Set

- Examples

```
Set<string> weekDays = Set{"Mon", "Tue", "Wed", "Thur", "Fri"};
```

```
Set<int> theFirstHundredOddSquares =  
    Set{ j in theFirstHundredNaturals  
        , j % 2 == 1  
        ; j * j  
        };
```

Seq

- ```
s[i] // value of element at i'th index in s
```
- **Operators**  
`x in s` // x is an element of sequence s  
`x notin s` // x is not an element of sequence s  
`s + t` // return concatenation of sequences s and t
  - **Properties**  
`s.Size` // number of elements in sequence s (same as Length)  
`s.Length` // number of elements in sequence s (same as Size)  
`s.IsEmpty` // sequence s is empty  
`s.Head` // first element in s  
`s.Tail` // sequence of all but the first element in s  
`s.Last` // last element in s  
`s.Front` // sequence of all but the last element in s  
`s.Indices` // set of indices of sequence s  
`s.IndexRange` // sequence of indices of sequence s  
`s.Values` // set of values in sequence s

## Seq

### ■ Updates

```
s[i] = x; // assign x to element at i'th index in s
 // If i is not an index of s, attempting to evaluate or assign s[i] throws an
 // IndexOutOfBoundsException.
s += t; // concatenate elements of t to s (append t to s)
```

### ■ Instance methods

```
s.Add(x) // return sequence like s but with element x added to the end
s.Drop(n) // return sequence of all but the first n elements of s
s.Take(n) // return sequence of the first n elements of s
s.Subseq(i,j) // return sequence starting with i'th element of s, up to j'th
s.IndexOf(x) // return index where sequence x first appears in s
s.LastIndexOf(x) // return index where sequence x last appears in s
```

## Seq

### ■ Static methods

```
Zip(s,t) //returns sequence of pairs, i'th pair is i'th elements of s and t
Unzip(p) //returns two sequences, seqs. of 1st and 2nd elements in p
Flatten(ss) //returns one sequence, concatenation of all the sequences in ss
```

## Seq

```
Seq<char> hallo = Seq{'h','a','l','l','o'};
```

```
Seq<char> explode(string s) {
 return Seq{ c as char in s; c };
}
```

## Map

### ■ Access

```
u[i] // value in map u at index i
```

### ■ Operators

```
i in u // i is an index in the map u
i notin u // i is not an index in the map u
u + v // override: return new map of all maplets from u and v
 // except where maplets have the same index, use the
 // maplet from v only
```

## Map

### ■ Properties

```
u.Size // number of maplets in map u
u.IsEmpty // map u is empty
u.Indices // set of indices in map u
u.Values // set of values in map u
```

### ■ Updates

```
u[i] = x; // update value in map u at index i to x, or add new maplet i := x
u[j] = none; // remove maplet at index j from map u
u += v; // override u with v
```

### ■ Instance methods

```
u.Add(i := x) // return new map like u but with maplet i := x added
```

### ■ Static methods

```
Union(u, v) // return map of maplets in both u and v, all at different indices
```

## Map

```
Map<string,int> directory = Map{"emergency" :=> 911, "info" :=>
411, "microsoft" :=> 8828080};
```

```
Map<string,int> calling_costs =
```

```
Map{ s in directory, directory[s] < 1000; s :=> 0 } +
```

```
Map{ s in directory, directory[s] > 1000; s :=> 25 };
```

## Enumerated types

- It is possible to form a set of all the values in an enumerated type. This makes it possible to iterate over them, use them in binders, etc.

```
[Enumerated]
class Agent{int id;}
```

```
Set<Agent> CreateAgents(int n){
 for(int i= 0; i < n; i++) new Agent(i);
 return enumof(Agent);
}
```

## Structures

- Spec# **structures** are similar to C# structs. They are value types: equality is by structure, not reference; assignment copies the entire value. Creating an instance of a structure does not use the **new** keyword.

```
structure Point{
 readonly int x,y;
 Point Move(int dx, int dy){
 Point p = this;
 p.x += dx;
 p.y += dy;
 return p;}
}
```

## Quantifiers

### Forall

Forall { binder; condition } tests whether all bindings satisfy condition.

```
int[] A = new int[] {1,2,3,4,6,8};
bool X1 = Forall{ i in A; i % 2 == 0 };
```

Forall over the empty set returns true.

### Exists and Exists1

Exists { binder; condition } tests whether at least one binding satisfies condition.

Exists1 tests whether there is exactly one binding that meets the condition. This expression checks whether all array elements are distinct (note the nested quantifiers).

```
bool X2 = Exists{ i in A; i >= 4 && i % 2 == 1 };
bool X3 = Exists1{ i in A; i % 8 == 0 };
```

The expression X2 will return false, since there is no odd number in A greater 4.

The expression X3 is true, since there is exactly one element in A which is divisible by 8.

Exists over the empty set returns false.

## Choose and Choose1

- The **Choose** and **Choose1** expressions return an element from a binding depending on whether a given condition has been existentially met by at least one example and by exactly one example, respectively. Choose provides nondeterministic choice, while Choose1 provides the unique element satisfying some condition.

```
structure Cd{ readonly string Title; readonly int recorded; }
Cd [] carusel = new Cd[] { Cd("BoogieWoogie",1954),
 Cd("SingAlong",1940),
 Cd("halulaHu",1960),
 Cd("SingAlong",1972)};
```

```
Cd X4 = Choose{ s in carusel, s.Title == "SingAlong"; s };
```

```
Cd X5 = Choose1{ s in carusel, s.Title == "SingAlong";
 s ; default Cd("EvenBetter",2003) };
```

## Reduction: Max and Min

- Reduction{ binder; expression } // syntax for reductions

// Reduction is Max, Min, or ...

The value of the reduction is computed from the collection of all the values formed by evaluating the expression with all the values in the bindings.

- Spec# provides the reductions Min and Max. In these examples S and T are collections.

```
Max{ x in S, y in T; x + y }
```

```
Min{ x in S + T; x }
```

## Ranges and Collections

- Ranges

```
Set<int> theFirstHundredNaturals = Set{ 1..100 };
```

- Comprehensions

A comprehension is an expression whose value is a collection. The syntax is similar to reductions:

```
Collection{ binder; expression } // syntax for comprehensions
```

// Collection is Set, Sequence, Map or ...

## Control structure

### ▪ Foreach

Spec# allows the use of patterns and binders for defining the iteration variable of a **foreach** statement. If the pattern doesn't match the element is simply ignored.

```
int Average1 (object [] a) {
 int sum = 0;
 foreach (i as int in a) sum+=i; // some elements of a might not be int
 return sum/a.Length;
}
```

Filters refine the evaluation of **foreach** statements even further. Here type inference infers that the type of *i* is int.

```
int Average2(int [] a) {
 int sum = 0;
 foreach (i in a, i > 2) sum+=i;
 return sum/a.Length;
}
```

## Control structure

### ▪ Switch

```
structure List {
 static int Sum(List<int> l) {
 switch(l) {
 case Nil : return 0;
 case Cons(h,t) : return h + Sum(t);
 default : throw new Exception("Impossible");
 }
 }
}
```

### ▪ Parallel

```
parallel { // Swap using parallel assignment, no temporary variable needed
 x = y;
 y = x;
}
```

## Foreach parallel

- **Foreach parallel:** The foreach parallel statement provides parallel assignment within the scope of a binder. All the bindings can be used in the assignments. Here is an example.

```
var Set<int> SumSquares = Set{};
readonly SmallInts = Set{ 1..5 };
```

```
foreach parallel (i in SmallInts) {
 SumSquares[i*i] = true;
} // now SumSquares == { 1, 4, 9, 16, 25 }
```

## Exercise

- 1) Especifique, em Spec#, uma máquina de venda de produtos (bebidas, snacks, bolos, bolachas,...).
  - Esta máquina aceita moedas de 0.05, 0.1, 0.2, 0.5 e 1 euros.
  - A máquina tem um stock de moedas que servem para dar o troco aos clientes que o necessitem.
  - A máquina tem um stock de produtos e cada produto tem um preço.
  - A máquina de venda deve fornecer os seguintes serviços:
  - Dois estados: em configuração ou à espera de interação com o cliente;
  - **Em configuração:** é possível actualizar o stock de produtos e de moedas;
  - **Em espera:**
    - a máquina deve mostrar aos clientes os produtos disponíveis.
    - os clientes devem seleccionar o produto que pretendem e depois inserir o valor correspondente em euros;
    - a máquina deve dar troco sempre que possível ou devolver o dinheiro ao cliente sem efectuar a venda.
- 2) Construa uma implementação numa linguagem suportada pela plataforma .NET.
- 3) Teste a conformidade da implementação usando o Spec Explorer.