Sequences

Sequences

A sequence is an ordered list of elements. Similar to an array in a programming language

Constructor

```
seq of <type>{..};
seq of int{};
seq of int{1,2,4,5,8,9};
seq of Money{Money{10}};
```

Sequences

x # seq1	returns frequency of occurrence of x in seq1
x in seq1	returns `true` if x is an element of seq1; otherwise `false`
#seq1	returns number of elements in seq1
seq1.max	returns the maximum element in seq1
seq1.min	returns the minimum element in seq1

Sequences

seq1.empty	returns true if #seq1 = 0; false otherwise
seq1.begins(seq2)	returns true if seq2 is a prefix of seq1; false otherwise
seq1.ends(seq2)	returns true if seq2 is a postfix of seq1; false otherwise

Sequences

seq1.isndec	returns `true` if `seq1` is not a decreasing sequence; `false` otherwise
seq1.isninc	returns `true` if `seq1` is not an increasing sequence; `false` otherwise
seq1.unique	returns `true` if all elements in `seq1` unique; `false` otherwise

Sequences

seq1[k]	returns element at index position k, where 0 <= k < #seq1
seq1.dom	returns the set of index values of seq1
seq1.ran	returns the set of values in seq1, duplicates removed
seq1.ranb	returns a bag of the values in seq1, ordering lost

Sequences

seq1.findFirst(x)

returns the index of the leftmost occurrence of x in seq1; -1 if x~in seq1

seq1.findLast(x)

returns the index of the rightmost occurrence of x in seq1; -1 if x~in seq1

seq1.take(n)	returns a sequence comprising the first n elements of seq1, where 0<=n<#seq1
seq1.drop(n)	returns a sequence with the first n elements of seq1 removed, where 0<=n<#seq1

Sequences

```
const s1 : seq of int
  ^=seq of int{1,2,3,4,5,8,9};
property assert
        100 in s2;
        s1.isndec;
        s1.unique;
        ~s1.isninc;
        s1.dom = set of nat{0,1,2,3,4,5,6};
        s1.drop(2) = seq of int{3,4,5,8,9};
        s1.begins(seq of int{1,2,3});
        s1.findFirst(3) = 2;
```

Sequences

seq1 ++ seq2	returns a sequence comprising seq1 followed by seq2, #(seq1 ++ seq2) = #seq1 + #seq2
seq1.append(x)	returns a new sequence with x appended to seq1
seq1.remove(k)	remove element whose index is k, 0<=k<#seq1

seq1.insert(k,x) — return a new sequence with x inserted before the element at position k, 0 <= k < #seq1

seq1.rev — return a sequence with the order of the elements in seq1 reversed

Sequences

seq1.head	returns the first element of the non-empty sequence seq1
seq1.tail	return the sequence seq1 with seq1.head removed
seq1.last	returns the last element of the non-empty sequence seq1
seq1.front	return the sequence seq1 with seq1.last removed

Definitions

seq1 = seq1.head++ seq1.tail

seq1 = seq1.front++ seq1.last

seq1.permndec returns a permutation of seq1 sorted by nondecreasing order

seq1.permninc returns a permutation of seq1 sorted by nonincreasing order

Sequences

```
const s1 : seq of int
  ^=seq of int{1,2,3,4,5,8,9};
property assert
        s1.rev = seq of int{9,8,5,4,3,2,1};
        s1.remove(6) = seq of int{1,2,3,4,5,8};
        s1.head = 1;
        s1.tail = seq of int{2,3,4,5,8,9};
        s1.last = 9;
        s1.front = seq of int{1,2,3,4,5,8};
        s1.permninc = seq of int{9,8,5,4,3,2,1};
```

`over` expression

`op over s` applies `op` to all elements in the collection `s`,

where `s` is a non-empty set, sequence or bag, and `op` is a binary operator whose operand and return types are all equal to the type of the elements of `s`.

For sequences it is defined as:

```
( [#s = 1]: that s,
  []: (op over s.front) op s.last
 )
```

`over` expression

In the case of a set or bag the definition is:

```
( [#s = 1]: that s,
  []:
    (let tmp ^= any s;
    (op over s.remove(tmp)) op tmp)
 )
```

Note

The operator must not have any precondition and the collection must have at least one element.

over expression

For example

To sum the elements in an int sequence seq1 use

```
+ over seq1
```

Given a sequence m1 of Money the total value of the elements in the sequence is given by:

```
+ over m1
```

SeqEx

```
class SeqEx ^=
    abstract
        var data : seq of int;
    interface
    ..
    build{h : seq of int}
        post data! = h;
end;
```

Append a new element

```
schema !add(x : int)
    post data! = data.append(x);
```

Delete element at position k

```
schema !delete(k : int)
    pre 0 <= k < #data
    post
        data! = data.remove(k);
```

Remove element at position 0

```
schema !deleteHead
  pre #data > 0
  post data! = data.tail;
```

Insert an element at the head of the sequence

```
schema !insertAtHead(x:int)
  post data! = data.insert(0,x);
```

An alternative post condition is:

```
data! = seq of int{x} ++ data;
```

SetEx

Retrieve the sequence

```
function data;
```

Retrieve frequency of occurrence of `x`

```
function frequency(x : int):int
    ^= x # data;
```

Retrieve number of elements in `data`

```
function size : nat
    ^= #data;
```

Search for x

```
function search(x : int):bool
  ^= x in data;
```

Retrieve element given index value

```
function get(index :nat): int
      pre index < #data
  ^= data[index];
```

SetEx

Retrieve subsequence of all odd elements

```
function getAllOdd: seq of int
  ^= (those x :: data :- x % 2 ~= 0);
```

Check if all elements are positive

```
function allPositive : bool
  ^= (forall x :: data :- x > 0);
```

SetEx

Retrieve indices of all occurrences of x, if any

```
function findIndices(x : int):seq of int
  ^= (those j :: 0..<#data :- data[j] = x);
```

Calculate the sum of the elements

```
function getSum : int
      pre #data > 0
  ^= + over data;
```

SetEx

Calculate the sum of the even elements

```
function sumEven : int
  pre #data > 0 &
       (exists x :: data :- x % 2 = 0)
 ^= + over (those x :: data :- x % 2 = 0);
```

CinemaQueue

Specify a queue of people waiting to gain entry to the cinema. People gain entry in the order in which they arrive.

CinemaQueue

We can model the queue as a sequence of `Person`, where class `Person` is defined as:

```
class Person ^=
  abstract
    var
      person : string;
  interface
      function person;
      build{!person:string};
end;
```

CinemaQueue

```
import "Person.pd";
class CinemaQueue ^=
  abstract
    var
      queue : seq of Person,
      max : nat;
    invariant
      #queue <= max
```

CinemaQueue

```
interface
    ...
    build{m : nat}
    post queue! = seq of Person{}, max! = m;
end;
```

```
function queue;

function front : Person
  pre #queue > 0
  ^= queue.head;

function size : nat
  ^= #queue;
```

CinemaQueue

```
function full : bool
    ^= #queue = max;

function empty : bool
    ^= queue.empty;
```

CinemaQueue

```
schema !join(p:Person)
  pre #queue < max
  post
     queue! = queue.append(p);

schema !leave
  pre #queue > 0
  post queue! = queue.tail;
```

Changing elements in a sequence

Given a sequence of elements write schemas to change some or all of the elements in the sequence.

Given `data : seq of int` write a schema to increment each element by 1.

```
schema !incAll
 post
     forall j :: 0..<#data :-
                       data[j]! = data[j] + 1;
```

Changing elements in a sequence

Change element at position n to x

```
schema !setAt(n : nat, x : int)
    pre 0 <= n < #data
    post data[n]!  = x;
```

Changing elements in a sequence

Set all even values to 0

```
schema !zeroEven
  post
        forall j :: 0..<#data :-
               ([data[j] % 2 = 0]:
                       data[j]! = 0,
                 []: pass
               );
```

Changing elements in a sequence

Given a class Room as follows:

```
import "RoomId.pd";
class Room ^=
  abstract
      var
        locked : bool,
        room : RoomId;
   interface
     function room;
     function locked;
     schema !lock
         post locked! = true;
```

Changing elements in a sequence

```
    schema !unlock
        post locked! = false;
    build{r : RoomId}
          post locked! = true, room! = r;
    build{r : RoomId, lockStatus : bool}
       post locked! = lockStatus, room! = r;
end;
```

Note: See Question 9 Worksheet 3 for details.

Changing elements in a sequence

Given an abstract variable rooms : seq of Room, a schema to unlock a room given its index value is:

```
schema !openRoom(n : nat)
  pre 0 <= n < #rooms
  post
     rooms[n]!unlock;
```