Bags

Bags

A bag is a multi valued set that allows duplicates.

Constructor

```
bag of <type>{comma separated list};

bag of int{1,2,3,1,3};
bag of Money{Money{20}};
bag of Money{};
```

Bags

x # bag1	returns frequency of occurrence of x in bag1
x in bag1	returns true if x is an element of bag1; otherwise false
#bag1	returns number of elements in bag1
bag1.max	returns the maximum element in bag1
bag1.min	returns the minimum element in bag1

Bags

`bag1.unique`	returns `true` if all elements are unique; `false` otherwise
`bag1.ran`	returns a set of the elements in `bag1` – duplicates removed
`bag1.rep(n)`	returns a bag with each element replicated `n` times.

Bags

```
const b1 : bag of int
   ^= bag of int{1,2,3,1,1,3};
property assert 1 # b1 = 3;
property assert 2 in b1;
property assert #b1 = 6;
property assert b1.max = 3;
property assert b1.ran = set of int{1,2,3};
property assert b1.unique = false;
```

Bags

```
property assert bag of int{1,2,3}.rep(2)
                = bag of int{1,1,2,2,3,3};
```

Bags

Bag addition(union)

b1 ++ b2 returns the union of b1 and b2. It is defined as:

```
operator++(a: bag of X): bag of X
    satisfy forall x:X :-
            x # result = x # self + x # a;

property assert
     bag of int{2,5,5,6} ++ bag of int{2,5,7} =
                              bag of int{2,2,5,5,5,6,7};
```

Bag intersection

b1 ** b2 returns the intersection of b1 and b2. It is defined as:

```
operator**(a: bag of X): bag of X
  satisfy forall x:X :-
            x # result = min(x # self, x # a);

property assert
     bag of int{2,5,5,6} ** bag of int{2,5,7} =
                              bag of int{2,5};
```

Bags

Bag difference

b1 -- b2 returns a bag of elements x whose frequency is max(x#b1 – x#b2,0). It is defined as:

```
operator -- (a: bag of X): bag of X
  satisfy forall x:X :-
      x # result = max(x # self - x # a, 0);


property assert
      bag of int{2,5,5,6} -- bag of int{2,5,7}
      = bag of int{5,6};
```

Bag inclusion

b1 <<= b2 returns true if all the elements of b1 are contained in b2; false otherwise. It's definiton is:

```
operator<<=(a: bag of X): bool
  ^= forall x::self :- x # a <= x # self;
```

Bags

b.append(x)	returns a new bag that is the same as b with the frequency of occurrences of x is increased by 1
b.remove(x)	returns a new bag that is the same as b with the frequency of occurrences of x is decreased by 1, if x is an element of b.

Example

bag of int{1,2,3}.append(3) = bag of int{1,2,3,3};

bag of int{1,2,3}. remove(3) = bag of int{1,2};

Bag Example

```
class BagEx ^=
abstract
   var
    data : bag of int;
interface
   function data;
   function frequency(x : int): nat
    ^= x # data;
   function cardinality : nat
     ^= #data;
```

BagEx

```
function evenBag : bag of int
 ^= those y :: data :- y % 2 = 0;

function allPositive : bool
 ^= forall y :: data :- y > 0;
```

BagEx

```
schema !add(x:int)
    post data! = data.append(x);
schema !add(b : bag of int)
    post
        data! = data ++ b;
build{}
    post data! = bag of int{};
end;
```

Specify a Purse

Specify a class to model a purse of coins. The purse may contain multiple instances of the same coin.

Solution

1. Specify class Coin
2. Specify class Purse

Coin

```
class Coin ^=
 abstract
   var
      coin : nat;
   invariant
     coin in set of nat{1,2,5,10,20,50,100,200};
```

Coin

```
interface
   function coin;
   redefine function toString : string
    ^=
     ([coin in set of nat{1,2,5,10,20,50}]:
         coin.toString++"Cent",
      []:
         (coin/100).toString ++ "Euro"
     );
```

Coin

```
operator +(other:Coin):nat
    ^= coin + other.coin;

operator +(other : nat) : nat
  ^= coin + other;
```

Coin

```
operator ~~(other)
  ^= coin ~~ other.coin;

build{x : nat}
 pre
    x in set of nat{1,2,5,10,20,50,100,200}
 post
     coin! = x;
end;
```

Purse

A purse is a bag of coins that has the following schemas:

- add a coin
- add a bundle of coins
- remove a coin
- remove a sum of money

Purse

It should have functions that return:

- all the coins in the purse
- the frequency of a given coin
- the total value of a given coin
- `true` if a given coin is contained in the purse; `false` otherwise
- the set of coins in the purse
- the total value of all coins in the purse
- `true` if the purse contains a bag of coins whose value equals some given amount of money; `false` otherwise
- that bag of coins in the purse whose sum equals some given value

Purse

```
import "Coin.pd";
class Purse ^=
abstract
   var
    purse : bag of Coin;
interface
  …
   build{}
         post purse! = bag of Coin{};
   build{x : bag of Coin}
      post purse! = x;
end;
```

Purse

All coins in purse

```
function purse;
```

Frequency of a given coin in purse

```
function freqCoin(c : Coin): nat
  ^= c # purse;
```

Total value of a given coin

```
function valueOfCoins(c : Coin) : nat
  ^= (c # purse)*c.coin;
```

Purse

Check if a given coin is contained in the purse

```
function contains(c : Coin) : bool
  ^= c in purse;
```

The set of different coins in the purse

```
function listOfCoin : set of Coin
  ^= purse.ran;
```

Purse

Total value of all coins in the purse

```
function sum : nat
  ^= sum(purse);
```

where

```
function sum(p : bag of Coin)  : nat
  decrease #p
^=
    (  [#p = 0] : 0,
       []: ( let t ^= any p;
             (t + sum(p.remove(t)))
           )
    );
```

Purse

The purse contains a bag of coin whose sum equals a given value

```
function containsSum(x : nat) : bool
  ^= (exists b:bag of Coin :-
                        b <<= purse & sum(b) = x);
```

Purse

Retrieve a bag of coin whose sum equals a given value

```
function bagCoins(x : nat) : bag of Coin
    pre containsSum(x)
    satisfy (result <<= purse & sum(result) = x);
```

Retrieve a bag of coin whose sum is at least a given value

```
function bagCoins1(x : nat) : bag of Coin
    pre containsSum(x)
    satisfy (result <<= purse & sum(result) >= x);
```

Add Schemas

```
schema !add(c : Coin)
  post
     purse! = purse.append(c);

schema !add(b : bag of Coin)
  post
    purse! = purse ++ b;
```

Remove Schemas

```
schema !take(c : Coin)
  pre c in purse
  post purse! = purse.remove(c);

schema !take(x : nat)
  pre x > 0 & containsSum(x)
  post
     purse! = purse -- bagCoins(x);
```