Collections

Description	Declaration	Examples
Sets	set<T>	var s:set<int>:={1,2,3}
(Without repetions)		
Sequences	seq<T>	var s:seq<int>:=[1,2,3,3]
(Lists)		
Multiset	multiset<T>	var s:multiset<int>:= multiset{2,3,3}
(With Repetions)		
String	string	"hello world\n"
Map	map<K,V>	map<string, int>:= map["one":=1,"two":= 2]
(Dictionary)		

Set operations

operator	description
<	proper subset
<=	subset
>=	superset
>	proper superset

operator	description
!!	disjointness
+	set union
-	set difference
*	set intersection

<table>
<tr><th>expression</th><th>description</th></tr>
<tr><td>|s|</td><td>set cardinality</td></tr>
<tr><td>e in s</td><td>set membership</td></tr>
<tr><td>e !in s</td><td>set non-membership</td></tr>
</table>

multiset(s): set conversion to multiset<T>

Sets defined by comprehension: values $Q(x_1, \ldots, x_n)$ that satisfy $P(x_1, \ldots, x_n)$:

$$var\ S \quad := \quad x_1 : T_1, \ldots, x_n T_n \ldots \mid P(x_1, \ldots, x_n) :: Q(x_1, \ldots, x_n)$$

Example:
```
var S := set x:nat, y:nat | x < 2 && y < 2 :: (x, y)
```
gets

$$S = \{(0,0), (0,1), (1,0), (1,1)\}$$

Sequence operators

operator	description
<	proper prefix
<=	prefix

operator	description
+	concatenation

<table>
<tr><th>expression</th><th>description</th></tr>
<tr><td>|s|</td><td>sequence length</td></tr>
<tr><td>s[i]</td><td>sequence selection 0 <= i < |s|</td></tr>
<tr><td>s[i := e]</td><td>sequence update</td></tr>
<tr><td>e in s</td><td>sequence membership</td></tr>
<tr><td>e !in s</td><td>sequence non-membership</td></tr>
<tr><td>s[lo..hi]</td><td>subsequence 0 <= lo <= hi <= |s|</td></tr>
<tr><td>s[lo..]</td><td>drop</td></tr>
<tr><td>s[..hi]</td><td>take</td></tr>
<tr><td>s[slices]</td><td>slice</td></tr>
<tr><td>multiset(s)</td><td>sequence conversion to a multiset<T></td></tr>
</table>

Multiset operations

operator	description
<	proper multiset subset
<=	multiset subset
>=	multiset superset
>	proper multiset superset

operator	description
!!	multiset disjointness
+	multiset union
-	multiset difference
*	multiset intersection

expression	description
|s|	multiset cardinality
e in s	multiset membership
e !in s	multiset non-membership
s[e]	multiplicity of e in s
s[e := n]	multiset update (change of multiplicity)

Map operators

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

The comprehension maps are defined as sets are.

$$map\ x : int \mid 0 <= x <= 10 :: x * x$$

Proving theorems with Dafny

Proving a number theory theorem:

Theorem 1. $\forall k > 0, 2^{3k} - 3^k$ *is divisible by* 5.

The theorem can be proven by induction on k. We can simulate that with a program and ensure that the program validates the theorem.

The proof although using assertions is similar to the one that can be done with an interactive theorem prover.

$2^{3k} - 3^k$ is divisible by 5

```
function f(k: int): int
requires k >= 1;
{ (exp(2,3*k) - exp(3,k)) / 5 }
```

```
function exp(x: int ,e: int): int
 requires e >= 0
 decreases e
{ if e==0 then 1 else x * exp(x,e - 1) }
```

$2^{3k} - 3^k$ is divisible by 5

```
method compute5f (k: int) returns (r: int)
requires k >= 1
ensures r == 5*f(k)
{
var i, t1, t2:= 0, 1, 1;
while i < k
decreases k-i
invariant 0 <= i <= k;
invariant t1 == exp(2,3*i);
invariant t2 == exp(3,i);
{
i, t1, t2 := i+1, 8*t1, 3*t2;
}
r := t1 - t2;
}
```

Assume and Assert

We use the directive `assume`.

```
assume t1 ==  exp(2,3*i)
```

and assert the post condition

```
assert r == exp(2,3*k) - exp(3,k);
```

To verify the program one needs to change `assume` to `assert`.

But Dafny cannot prove the assertion.

Solution: try to construct a tableaux using the weakest precondition technique

```
assume 8*t1 == exp(2,3*(i+1));
i, t1, t2 := i+1, 8*t1, 3*t2;
assert t1 == exp(2,3*i);
```

Lemmas

But the assert still does not hold

```
assert 8*t1 == 8*exp(2,3*i)== exp(2,3*(i+1))==exp(2,3*i+3);
```

Some lemmas are needed and then we use them instead of `asserts`.

```
lemma expPlus3_Lemma (x: int , e: int )
requires e >= 0;
ensures x * x * x * exp(x,e) == exp(x,e+3)
// to be proved

lemma DivBy5_Lemma (k: int )
requires k >= 1
ensures (exp(2,3*k) - exp(3,k)) % 5 == 0
// to be proved
```

In general a `lemma` is like a `method` but is not executed in runtime. They allow proofs by induction.

```
lemma Ex Lemma (x1: T1,...,xn: Tn)
requires phi
ensures psi
{ body }
```

It means $\forall x_1, \ldots, x_n (\phi \rightarrow \psi)$ and the body is the proof.

A call to `Lemma(a)` corresponds to variables instantiation.

`expPlus3Lemma`

For the first lemma we just use an iterative computation

```
lemma expPlus3_Lemma (x: int , e: int )
requires e >= 0;
ensures x * x * x * exp(x,e) == exp(x,e+3);
{
    assert x * x * x * exp(x,e) == x * x * exp(x,e+1) == x * exp(x,e+2) == exp(x,e+3);
   // assert x* exp(x,e) == exp(x,e+1);
}
```

`DivBy5Lemma`

For the second one needs to use `calc` that allows to perform algebraic calculations where one can use also `assert, lemma`, etc to justify each step (*hints*, that are written with {}).

Each step is separated by a logic operator : $==$, $\rightarrow$, etc

```
Lemma {:induction k} DivBy5_Lemma (k: int )
requires k >= 1
decreases k
ensures (exp(2,3*k) - exp(3,k)) % 5 == 0
{
if k==1 {
} else {
calc {
(exp(2,3*k)- exp(3,k)) % 5; ==
(exp(2,3*(k-1)+3) - exp(3,(k-1)+1)) % 5;
{
expPlus3_Lemma(2,3*(k-1));
}
(8*exp(2,3*(k-1)) - exp(3,(k-1))*3) % 5;
==
 (3 *( exp(2,3*(k-1)) - exp(3,k-1) ) + 5*exp(2,3*(k-1))) % 5;
  ==
 { DivBy5_Lemma (k-1);
 }
// assert(exp(2,3*(k-1))- exp(3,k-1)) % 5 == 0;
0;}
}
}
```

The previous lemma can be simplified just stating what is needed to the proof.

```
lemma DivBy5LemmaS (k: int )
requires k >= 1
ensures (exp(2,3*k) - exp(3,k)) % 5 == 0
{ if k > 1
{expPlus3_Lemma(2,3*(k-1));
     DivBy5LemmaS(k-1);
}
}
```

Then the annotated program is

```
method compute5f (k: int) returns (r: int)
requires k >= 1
ensures r == 5*f(k)
{
var i, t1, t2:= 0, 1, 1;
while i < k
decreases k-i;
invariant 0 <= i <= k;
invariant t1 == exp(2,3*i);
```

```
invariant t2 == exp(3,i);
{
    expPlus3_Lemma(2,3*i) ;
   // assert 8*t1 == 8*exp(2,3*i)== exp(2,3*(i+1))==exp(2,3*i+3);
    i, t1, t2 := i+1, 8*t1, 3*t2;
    assert t1 == exp(2,3*i);
}
r := t1 - t2;
DivBy5Lemma(k);
//assert (exp(2,3*k) - exp(3,k))%5 == 0;
//assert r == 5 * f(k);
}
}
```