Functions and predicates

Functions in Dafny define mathematical functions and must be side effect free.

For example we could define

```
function min( a : int, b : int ) : int
{
  if a < b then a else b
}
```

Note

- The body of a function is a single expression.
- The body of this function uses an if-then-else expression.

Function methods

By default functions are not compiled for execution and can only be used in specification contexts (i.e. contracts, asserts, etc.)

To ensure functions are compiled use "**function method**":

```
function method min( a : int, b : int ) : int
{
  if a < b then a else b
}
```

Partial functions

We can define partial functions by using a **requires** clause:

```
function mod( a : int, b : int ) : int
    requires b != 0
{
    a % b
}
```

Recursive functions

Functions may be recursive.

```
function Pow( a : int, b : nat ) : int
        decreases b ;
{
        if b == 0 then 1 else a * Pow( a, b-1 )
}
```

Note

- Recursive functions declare a variant.
- Recursive function calls must decrease the variant.
- Here $b - 1 < b$ is universally true, so the call is okay.
- This ensures that the function definition is well defined and that any generated code terminates.
- If you don't supply a **decreases** clause, Dafny's verifier will try to infer a variant for you.
- Recursive methods also need a **decreases** clause.

Consider

```
function Bad( a : nat ) : nat
{
    if Bad( a + 1 ) == 2 then 0 else Bad( a + 1 ) + 1 ;
}
```

There is more than one function that satisfies the property

$$\forall a : \mathbb{N} \cdot f(a) = \begin{cases} 0 & \text{, if } f(a+1) = 2 \\ f(a+1) - 1 & \text{, if } f(a+1) \neq 2 \end{cases}$$

for example

$$[0, 2, 1, 0, 2, 1, 0, 2, ...]$$

and

$$[1, 0, 2, 1, 0, 2, 1, 0, ...]$$

so the ‘definition’ is not a definition at all.

Now consider

```
function Worse( a : nat ) : nat
{
    Worse( a+1) + 1 ;
}
```

There is no function that satisfies this ‘definition‘.

The variant ensures there is one function that satisfies the definition.

Predicates

Predicates are just functions that have a boolean result.

```
predicate Even( x : int ) { x % 2 == 0 }
```

Lemmas

[Optional material]

Recall that input parameters are immutable.

Consider a method that has no side effects and no result parameters!

method SomeLemma($x : T$)
  requires $P(x)$
  ensures $Q(x)$
{
  body
}

Computationally this method is pointless. The body can not affect any locations other than local variables.

If the method is correct, then it must be a theorem that

$$\forall x \in T \cdot P(x) \Rightarrow Q(x)$$

If the method verifies, then the verifier has proved this theorem!

Dafny calls such methods *lemmas*.

Lemmas are not translated for execution. Nor are calls to lemmas.

Example

lemma PositivePowerLemma(a : **int**, b : **int**)
    requires b > 0
    ensures Pow(a,b) == a*Pow(a, b-1)
{ }

This method verifies.

Another example:

```
lemma EvenPowerLemma( a : int, b : int )
        requires Even(b) && b >= 0
        ensures Pow(a, b) == Pow( a*a, b / 2 )
        decreases b ;
{ }
```

This method is correct, but it does not verify. (Spurious failure.)

The verifier is just not that clever. Yet.

We have 3 options:

- Live with the verification error message.
- Prove the lemma.
- Use an assume command.

Proofs

To prove the lemma, we need to write a body that the verifier can verify.

These bodies typically use a bunch of asserts, ifs, and even loops. For example, suppose the verifier can not prove

$$\forall x \in T \cdot P(x) \Rightarrow Q(x)$$

But it can prove each of

$$\forall x \in T \cdot P(x) \wedge x < 0 \Rightarrow R(x)$$
$$\forall x \in T \cdot P(x) \wedge x \geq 0 \Rightarrow S(x)$$
$$\forall x \in T \cdot R(x) \Rightarrow U(x)$$
$$\forall x \in T \cdot S(x) \Rightarrow U(x)$$
$$\forall x \in T \cdot U(x) \Rightarrow Q(x)$$

The following should verify

```
lemma SomeLemma( x : T )
    requires P(x)
    ensures Q(x)
{
    if( x < 0 ) {
        assert R(x) ; }
    else {
        assert S(x) ; }
    assert U(x) ;
}
```

The EvenPowerLemma can be proved by induction (i.e. using a recursive call).

The practical aspects of writing proofs in Dafny are beyond the scope of this course.

Assume commands

Assume commands can be used to tell the verifier that something is true at the given point in the program.

```
lemma EvenPowerLemma( a : int, b : int )
        requires Even(b) && b >= 0
        ensures Pow(a, b) == Pow(a*a, b/2)
        decreases b ;
{
        assume false ;
}
```

The effect of this **assume** command is to suppress the error message.

The verifier will check your assertions, but not your assumptions. Use **assume** with care.

Using lemmas

Suppose we want to verify some code:

C

D

But the verification fails because the verifier needs to know that $Q(a)$ is true after C.

We can try

C

assert $Q(a)$;

D

But suppose verification now fails because the verifier is not able to verify that $Q(a)$ is true after C.

We can use our lemma to help

C

assert $P(a)$;

$\mathrm{SomeLemma}(\ a\ )$;

assert $Q(a)$;

D

Now the verifier only needs to verify that $P(a)$ holds after C. That $Q(a)$ is true before D follows from the postcondition of $\mathrm{SomeLemma}$.

Example:

Consider the following Russian Peasant exponentiation algorithm.

```
method power( a0 : int, b0 : nat ) returns (c : int )
ensures c == Pow( a0, b0 ) ;
{
    c := 1 ;
    var a := a0 ;
    var b : nat := b0 ;
    while b != 0
    invariant Pow(a0, b0) == c * Pow(a, b) ← Fails
    decreases b {
        if b%2 == 1 {
            c := c * a ;
            b := b - 1 ; }
        assert Even( b ) ;
        b := b/2 ;
        a := a * a ; }
}
```

Even with the guidance that b is even, this fails to verify.

The algorithm works because $a^b = \left(a^2\right)^{b \operatorname{div} 2}$ for even b.

Which is what the verifier proved by verifying EvenPowerLemma

However the prover is not clever enough to use the lemma without being told to.

We can guide the prover to use the lemma by rewriting the loop body as

```
if b%2 == 1 {
        c := c * a ;
        b := b - 1 ; }
assert Even( b ) ;
EvenPowerLemma(a, b) ;
assert Pow( a, b ) == Pow( a*a, b / 2 ) ;
b := b/2 ;
a := a*a ; }
```

This verifies.

- The assert commands here are unnecessary. I put them in for readability.
- The call to EvenPowerLemma does not appear in the C# code, since it is a lemma.