3.- Methods, Specifications and Annotations

- Dafny resembles a typical imperative programming language: there are methods, variables, types, loops, arrays, ...
- One of the basic units of any Dafny program is the method.
- A method is a piece of imperative, executable code (the term "function" is reserved for a different concept in Dafny).
- Methods can return several results, each one with its own name and type, like the parameters.
- The method body is the code contained within the braces

```
method MultipleReturns(x: int, y: int) returns (more: int, less: int)
{
  more := x + y;
  less := x - y;
}
```

- Dafny allows to annotate methods to specify their behavior.
- The most basic annotations are method pre- and post-conditions, that is method contracts by **requires** and **ensures**.

```
method MultipleReturns(x: int, y: int) returns (more: int, less: int)
    requires 0 < y
    ensures less < x < more
{
  more := x + y;
    assert more > x;
  less := x - y;
}
```

- Unlike pre- and post-conditions, an assertion (**assert**) is placed somewhere in the middle of a method.

- Functions can be used directly in specifications, but only in specifications.
- Unlike a method, which can have all sorts of statements in its body, a function body must consist of exactly one expression, with the correct type.

```
function abs(x: int): int
{
  if x < 0 then -x else x
}

method ComputeAbs(x: int) returns (y int)
    ensures y = abs(x)
{
  if x < 0
     { return -x; }
  else
     { return x; }
}
```

Predicates

- A predicate is a function which returns a boolean.
- The use of predicates makes our code shorter, as we do not need to write out a long property over and over.
- ```
  predicate isPrime (x: nat)
  {
  x > 1 ∧ forall y • 1 < y < x ⟹ x % y ≠ 0
  }
  ```
  ```

Goldbach conjecture (1742)

Every even number greater than 2 is the sum of two primes.

```
predicate isPrime (x: nat)
{
x > 1 ∧ forall y • 1 < y < x ⟹ x % y ≠ 0
}

predicate isEven (x: nat)
{
x % 2 = 0
}

lemma Goldbach ()
ensures forall x • x > 2 ∧ isEven(x)
                 ⟹ ∃ y1 : nat, y2 : nat •
                   isPrime(y1) ∧ isPrime(y2)
                   ∧ x = y1 + y2
//TODO
```

Even numbers, at least, until 4.10^{18}, have passed the test.

Example: A method for computing (in f) the factorial (of n)

```
                    Precondition:   n ≥ 0
                    Postcondition  f = n!

function factorial (n: int): int
    requires n ≥ 0
{
if n = 0 then 1 else n * factorial(n−1)
}

method ComputeFact (n: int) returns f: int
    requires n ≥ 0
    ensures f = factorial(n)
{
  f := 1;
  x := n;
  while x > 0
        {
        f := f * x;
        x := x − 1;
        }
}
```

Annotated Methods

- To make it possible for Dafny to work with loops, you need to provide loop **invariant**s, another kind of annotation.
- Dafny proves that code terminates, i.e. does not loop forever, by using **decreases** annotations.
- Dafny is often able to guess the right **decreases** annotations, but sometimes it needs to be made explicit.
- Sometimes also **asserts** are required by the verifier (as hints) to complete the proof.
- Users can utilize **assert**s for help in thinking about the program.
- Commented **asserts** serve as documentation.

```
method ComputeFact (n: int) returns f: int
   requires n ≥ 0
   ensures f = factorial(n)
{
  var x := n;
  f := 1;
  while x > 0
    invariant 0 ≤ x ≤ n
    invariant f * factorial(x) = factorial(n);
    // decreases x; // In this case Dafny guesses it.
    {
    f := f * x;
        // assert f * factorial(x-1) = factorial(n);
    x := x - 1;
    }
}
```

Dafny Language (Core)

- Built-in specifications
  - pre- and postconditions (**requires** and **ensures**)
  - loop invariants (**invariant**), inline assertions (**assert**)
  - termination metrics (**decreases**)
  - framing (**reads**, **modifies**, **old**),
- Specification support (does not generate code)
  - sets, multisets, sequences, algebraic datatypes
  - user-defined functions/predicates
  - **ghost** variables and methods (**lemma**)
- Object-based language
  - generic classes, no subclassing
  - object references, dynamic allocation
  - sequential control