3.- Introduction to Dafny

- Dafny is a programming language for verification, as well as its compiler and verifier.
- Dafny was created by the Research in Software Engineering (RiSE) group at Microsoft Research (coordinated by Rustan Leino).
- Dafny includes built-in specification constructs and comes with a static verifier to validate the functional correctness of programs.
- Dafny is tought at many universities around the world.
- Dafny is used in program verification competitions and benchmark challenges.

Dafny Programming/Specification Language

- Dafny programming language is mainly imperative, sequential, strongly and statically typed with type inference, generic, modular and object-oriented.
- Dafny specifications are based on pre- and postconditions, along with invariants, frame specifications, and termination metrics.
- To support further specification the language provides recursive functions and suitable types like sets and sequences.
- Specification material is consumed in verification time, so that it is omitted from executable code.

- The Dafny user should mix the two different contexts:
  - effective programming
  - pure specification
- Dafny two main top level constructs are:
  - Methods (i.p. lemmas) are imperative procedures with named parameters (passed by value) and named return values.
  - Functions (i.p. predicates) are definitions of mathematical functions.
  - The body of a method is a list of statements
    - to produce an effect, in the case of non-ghost methods; or
    - to provide a proof, in the case of ghost methods or lemmas.
  - The body of a function is an expressions which admit recursive and mutually recursive calls.
- Functions are most likely used for specification, but they can be also used as functional programs (function method).

- Entities which are not written to generate executable code and used only for specification or verification are named `ghost`.
  - Functions are ghost by default.
  - Methods are not ghost, but lemmas are exactly ghost methods (that do not return any result)
- Methods, variables and also parameters (of methods) can be declared to be `ghost` with this keyword.

`assert` and `assume`

- `assert` φ
  - Dafny first tries to prove φ, and if successful, then φ can be used in the rest of proof.
  - Provides a non-instantiable lemma φ:
    A property that is previously and separately proved and helps to prove other properties.
- `assume` φ
  - Dafny assumes that φ is true: it is enabled to use φ in the current proof (without proving it).
  - Dafny does not consider verified any file with one `assume`.
- In verified software development:
  - `assume` φ for checking if φ is the required property;
  - If OK then change from `assume` to `assert`;
  - If "assertion violation" then φ must be proved in a `lemma` or some previous `assert`(s) must be inserted (as lemmas).

`assert` and `lemma`

- Lemmas have parameters, hence their re-usability by instantiation is an advantage of lemmas with respect to inline asserts.
- The induction hypothesis of inductive lemmas is invoked as a lemma call (over smaller parameters).
- A lemma is a ghost method whose contract represents the property it warrants/proves.

```
lemma Example_Lemma (x1 : T1, ... , xn : Tn)
requires P(x1,...,xn)
ensures Q(x1,...,xn)
{
Body
}
```

where `x1,...,xn` is the tuple of formal parameters and `T1,...,Tn` the tuple of respective types.

- The contract of lemma `Example_Lemma` means

$$\forall x_1 \ldots \forall x_n (P(x_1, \ldots, x_n) \rightarrow Q(x_1, \ldots, x_n))$$

- `Body` is a proof (code, asserts, etc) of such property.

- A lemma call like `Example_Lemma(a)` where a is the tuple of current parameters corresponds to

```
assert P[a/x];
assume Q[a/x];
```

- If the precondition `P` –for the current parameters `a`– can be proved, we can assume that the postcondition `Q` holds –for the current parameters `a`.
- The assume clause is discharged whenever the lemma is proved.

Basic Types of Proof of $(P \to Q)^\forall$

- Proof by induction (mathematical or structural)
- Direct proof or proof by construction (Unfolding definitions, cases, etc)
- Proof by contradiction

  A proof by contradiction *of* $(P \to Q)^\forall$ *can be made by getting* `false` *from supposing that* P *and* $\neg Q$ *hold.*

- Proof by contrapositive

  A proof by contrapositive *of* $(P \to Q)^\forall$ *can be made by getting* $\neg P$ *from supposing that* $\neg Q$ *holds.*

Proofs by contradiction/contraposition in Dafny

```
lemma Contradiction_Lemma(...)
requires P
ensures Q
{
if ¬Q {
      ...
      ...
      assert false;
      }
}
```

```
lemma Contraposition_Lemma(...)
requires P
ensures Q
{
if ¬Q {
      ...
      ...
      assert ¬P;
      }
}
```

Verified Calculations

- A calculation in Dafny is an statement that proves a property by a chain of expressions, each transformed into the next.
- The grammar for calculations is:

```
CalcStatement ::= calc {
                  CalcBody
                  }
CalcBody ::= Line
           (Op Hint
            Line)*
Line ::= Expression ;

Op ::=  ≤ | < | ≥ | > | ⟹ | ⟸ | ⟺ | ≠ | =

Hint ::= { (BlockStatement | CalcStatement)* }
```

where a BlockStatement is one or more `assert`/`assume` clauses and lemma calls.

Calculations $\Longrightarrow$ in proofs by contradiction and contraposition

```
lemma Contradiction_Lemma(...)
requires P
ensures Q
{
if ¬Q { calc {
                  ...
               ⟹ ...
               ⟹ false;
            }
       }
}
```

```
lemma Contraposition_Lemma(...)
requires P
ensures Q
{
if ¬Q {
       ⟹ ...
       ⟹ ...
       ⟹ ¬P;
        }
}
```

Methods

- 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 specifications shorter, as we do not need to write out a long property over and over.

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

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

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

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 `assert`s are required by the verifier (as hints) to complete the proof.
- Users can utilize `assert`s for help in thinking about the program.
- Commented `assert`s serve as documentation.
- A program-proof is not complete until all verification conditions have been discharged, i.e., all assume statements have been removed (or replaced by asserts), and all the lemmas have been proved.

```
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