Dafny (`www.rise4fun.com/Dafny`)

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

Dafny:Keywords

- `requires`: pré-condição
- `ensures`: pós-condição
- `invariant`: invariante
- `decreases`: variante
- é verificado estáticamente *correção total* dos programas : todos os programas e funções têm de terminar
- `assert`: condição que se tem de verificar sempre
- `reads`: para ser usado para indicar quais as posições de memória a que uma condição (definida por funções ou predicados) pode aceder. Corresponde ao *frame* da condição.

Dafny

- verifica a correção de funções em relação às pós-condições anotadas
- `method` define uma sequência de código executável
- sendo dado o tipo dos argumentos
- e `returns`
- indica o tipo e a variável que retorna
- `method M(a: A, b: B, c: C) returns (x: X, y: Y, z: Y)`
- declaração de variáveis locais : `var x:T`

Dafny

- Linguagem de Programação
- – Multi-paradigma: combina estilos imperativos, funcionais e orientada a objectos
  – permite a escrita de implementações (programas) e especificações (asserções e anotações)
- Ambiente de Programação
- – Verificador Z3
  – Compilador para C# (ou Go)
  – Extensão para o VSCode
  – Extensão para o Emacs
  – Na shell usar

Paradigmas de Programação

- Funcional
- – tipos imutáveis
  – Funções e Predicados sem side-effects (puramente declarativas)
  – Tipificadas
- Imperativa (estruturas de dados e objectos)
- – Variáveis tipificadas
  – Instruções
  – Métodos
  – Módulos
  – Classes
  – ...

Funções

Adequadas para especificações, definem a semântica que se pretende do programa imperativo (e não produzem código executável, excepto se forem declarados como `function method`))

```
function fact(n: int): int
   requires 0 < = n
   ensures 1 <= fact(n)
   decreases n
  {if n == 0 then 1 else fact(n-1) * n}
```

Podemos também defenir para `nat` evitando a pré-condição.

Fibonacci

```
function fib(n: nat): nat
     decreases n
{
   if n == 0 then 0 else
   if n == 1 then 1 else
                  fib(n - 1) + fib(n - 2)
}
method ComputeFib(n: nat) returns (b: nat)
   ensures b == fib(n)
{
   if n == 0 { return 0; }
   var i: int := 1;
   var a := 0;
       b := 1;
   while i < n
      invariant 0 < i <= n
      invariant a == fib(i - 1)
      invariant b == fib(i)
   {
      a, b := b, a + b;
      i := i + 1;
   }
}
```

Sistema de tipos

Ver Dafny Cheat Sheet

- Tipos imutáveis (valores)
-
  - tipos básicos: bool, int , nat, real, char
  - tuplos,
  - coleções,
  - inductivos
- Mutáveis (referências): arrays, classes, etc

Operadores Básicos

bool	$!\ ==\ !=\ \&\&\ ==>\ <==\ <==>$
$int, nat, real$	$==\ !=\ <\ <=\ =>\ +\ -\ *\ /$
char	$==\ !=\ <\ <=\ >=\ >$

Arrays

Instrução	Sintaxe	Exemplo
Declaração	`array<T>`	`var a=array<int>= new int[3];`
Instancia	`new T[n]`	
Atribuição	`a[i]:=value`	`a[1],a[2]:=2,4`
Comprimento	`a.length`	`assert a.Lenght ==3;`
Sequência	`a{lo..hi]`	`assert a[..]=[1,5,6]`

Arrays

```
method Find(a: array<int>, key: int) returns (i: int)
```

- `a.Lenght` : indica sempre o tamanho do array
- Nas condições podemos usar quantificadores sobre o valor das variáveis
- `forall k: int :: 0 <= k < a.Length ==>  a[k] != key`

Procura de um valor num array

```
method Find(a: array<int>, key: int)
                 returns (index: int)
   ensures 0 <= index ==> index < a.Length
       && a[index] == key
   ensures index < 0 ==>
 forall k :: 0 <= k < a.Length ==>
              a[k] != key
{
   index := 0;
   while index < a.Length
      invariant 0 <= index <= a.Length
      invariant forall k :: 0 <= k < index ==> a[k] != key
   {
      if a[index] == key { return; }
      index := index + 1;
   }
   index := -1;
}
```

Máximo de um array

```
method maxarray(arr:array? <int>) returns(max:int)
 requires arr!=null && arr.Length > 0
ensures 0<= max < arr.Length
```

```
 ensures (forall j :int :: (j >= 0 && j < arr.Length
                    ==> arr[max] >= arr[j]))
{
  max:=0;
  var i:int :=1;
  while(i < arr.Length)
  invariant (1<=i<=arr.Length)
  invariant 0<= max < i
  invariant (forall j:int :: j>=0 && j<i ==>
     arr[max] >= arr[j]);
    decreases (arr.Length-i);
  {
    if(arr[i] > arr[max]){max := i;}
    i := i + 1;
  }
}
```

Predicados

Permitem escrever condições (pós/pré). Predicados são funções que retornam um valor booleano (e portanto é omitido na especificação).

```
predicate sorted(a: array?<int>)
   requires a != null
   reads a
{
   forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
}
```

Pesquisa Binária

```
method BinarySearch(a: array?<int>, value: int) returns (index: int)
   requires a != null && 0 <= a.Length && sorted(a)
   ensures 0 <= index ==> index < a.Length && a[index] == value
   ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != value
{
   var low, high := 0, a.Length;
   while low < high
      invariant 0 <= low <= high <= a.Length
      invariant forall i ::
         0 <= i < a.Length && !(low <= i < high) ==> a[i] != value
   {
      var mid := (low + high) / 2;
      if a[mid] < value
      {
```

```
            low := mid + 1;
         }
         else if value < a[mid]
         {
            high := mid;
         }
         else
         {
            return mid;
         }
      }
      return -1;
   }
```