DAFNY QUICK REFERENCE*

*NON-EXHAUSTIVE

J. Pascoal Faria

FEUP, MIEIC, MFES, 2020/21

INDEX

WHAT IS DAFNY?

WHAT IS DAFNY?

A powerful programming language & system from Microsoft Research, targeting the development of verified programs, and used in several real world projects requiring formal verification.

Powerful programming language:

- Multi-paradigm: combines the functional, imperative and object-oriented styles
- Allows writing programs (implementations) and specifications (with several sorts of assertions & annotations)

Powerful programming system:

- Verifier using Z3
- Compiler to C#
- Extension for VSCode

PROGRAMMING STYLES

Functional

- Immutable types (value types)
- Side-effect free functions and predicates
- …

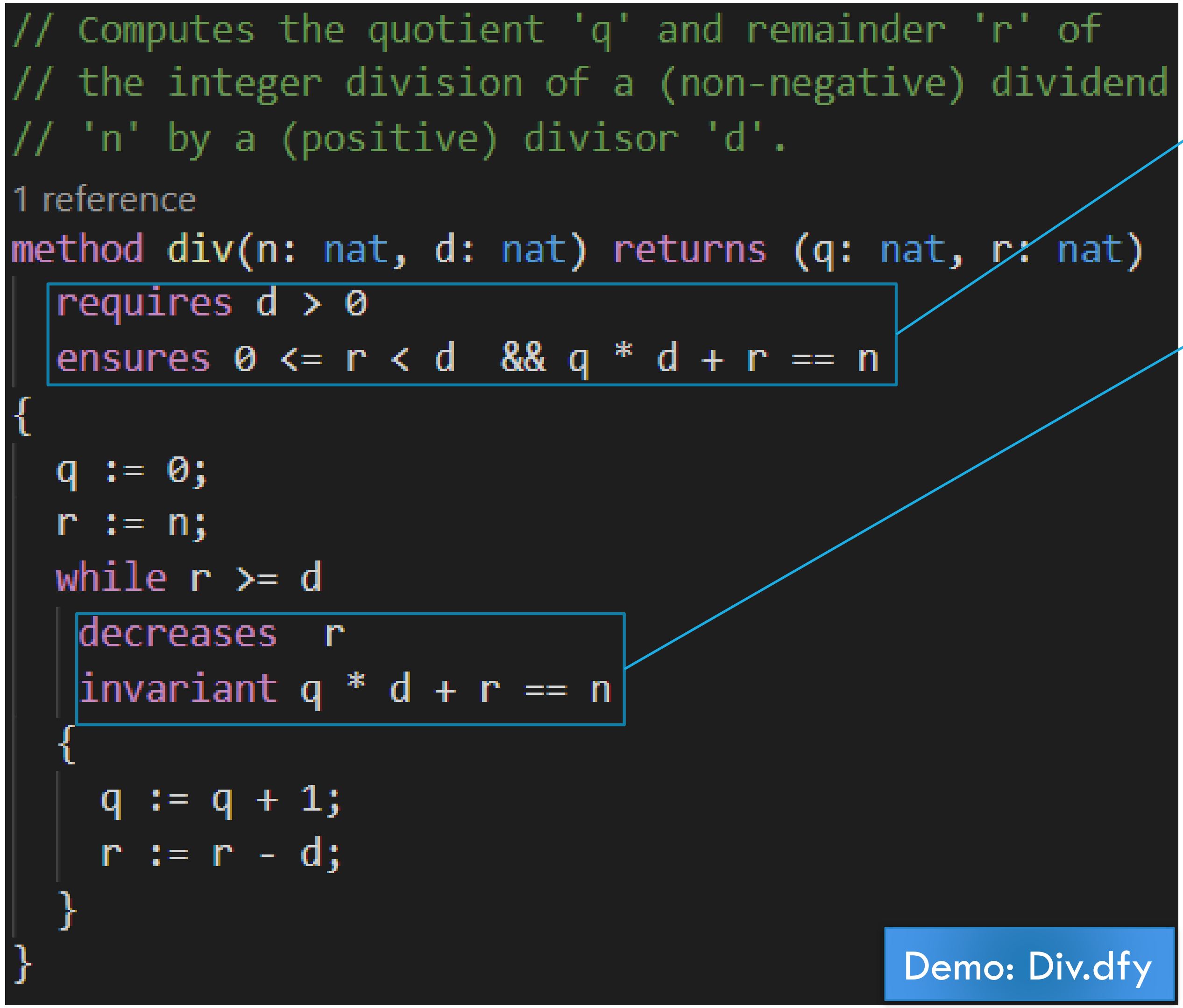
→ *Best suited for writing specifications!*

Imperative (structured & object-oriented)

- Statements
- Methods
- Modules
- Classes
- …

→ *Best suited for writing implementations!*

PROGRAMS (IMPLEMENTATIONS) & SPECIFICATIONS



```
// Computes the quotient 'q' and remainder 'r' of
// the integer division of a (non-negative) dividend
// 'n' by a (positive) divisor 'd'.
1 reference
method div(n: nat, d: nat) returns (q: nat, r: nat)
  requires d > 0
  ensures 0 <= r < d  && q * d + r == n
{
  q := 0;
  r := n;
  while r >= d
   decreases  r
   invariant q * d + r == n
  {
    q := q + 1;
    r := r - d;
  }
}
```

Demo: Div.dfy

Formal specification of method pre- and post-conditions (method semantics).

Formal specification of loop variant and invariant, to help proving that the implementation is correct.

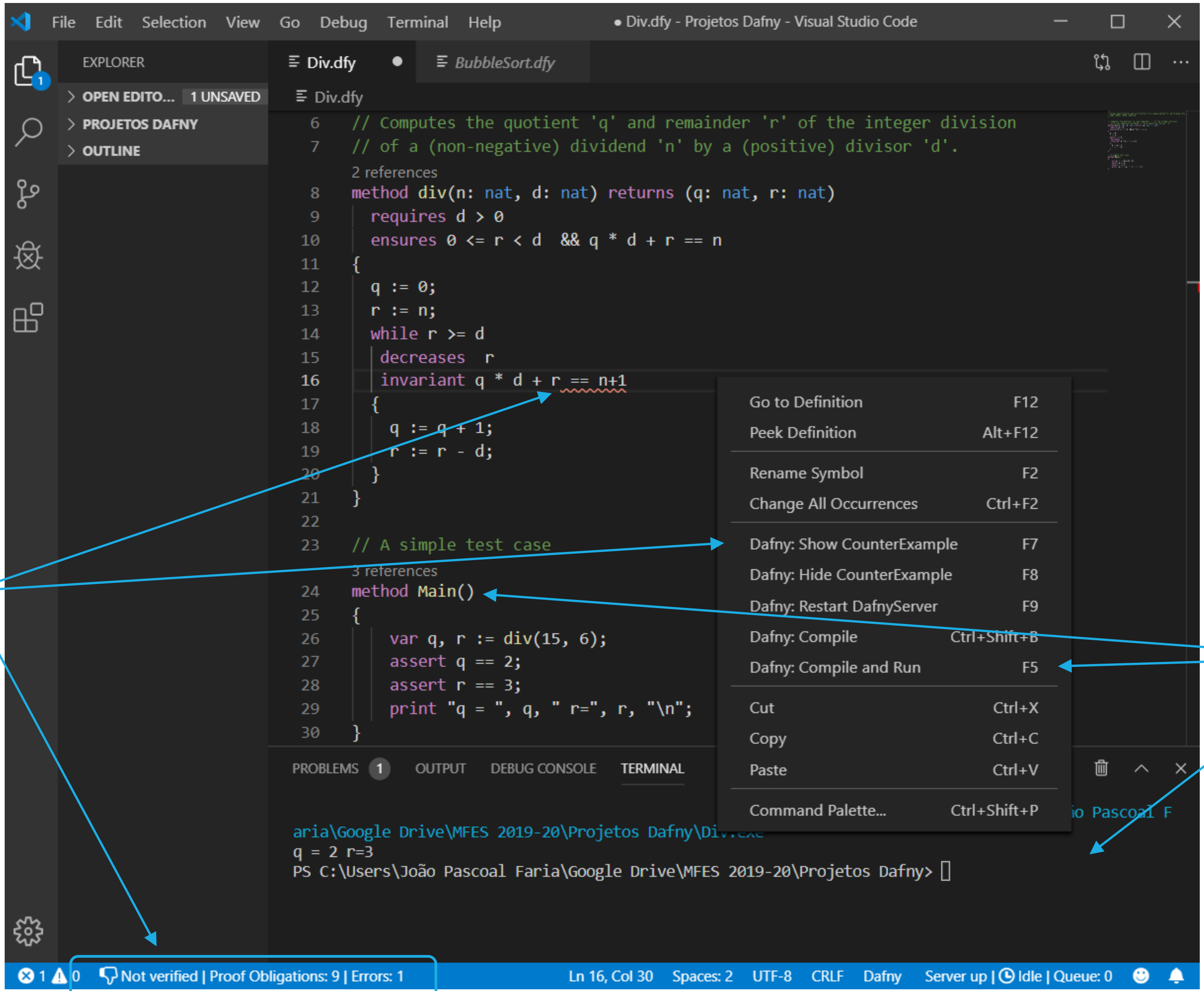
Other specification & proof constructs:

- assert
- lemma
- ghost
- …

All these constructs are used for verification purposes (by static analysis), but don't go into the executable program!

VERIFIER, COMPILER & VSCODE EXTENSION

The **verifier** continuously checks the consistency between specs & implementation.
It generates “proof obligations” that are checked with the Z3 theorem prover (that can also generate counter-examples).



File Edit Selection View Go Debug Terminal Help
● Div.dfy - Projetos Dafny - Visual Studio Code
EXPLORER
OPEN EDITO... 1 UNSAVED
PROJETOS DAFNY
OUTLINE
Div.dfy
BubbleSort.dfy
Go to Definition F12
Peek Definition Alt+F12
Rename Symbol F2
Change All Occurrences Ctrl+F2
Dafny: Show CounterExample F7
Dafny: Hide CounterExample F8
Dafny: Restart DafnyServer F9
Dafny: Compile Ctrl+Shift+B
Dafny: Compile and Run F5
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Command Palette... Ctrl+Shift+P
PROBLEMS 1
OUTPUT
DEBUG CONSOLE
TERMINAL

```
// Computes the quotient 'q' and remainder 'r' of the integer division
// of a (non-negative) dividend 'n' by a (positive) divisor 'd'.
2 references
method div(n: nat, d: nat) returns (q: nat, r: nat)
  requires d > 0
  ensures 0 <= r < d  && q * d + r == n
{
  q := 0;
  r := n;
  while r >= d
   decreases  r
   invariant q * d + r == n+1
  {
    q := q + 1;
    r := r - d;
  }
}

// A simple test case
3 references
method Main()
{
    var q, r := div(15, 6);
    assert q == 2;
    assert r == 3;
    print "q = ", q, " r=", r, "\n";
}
```

```
aria\Google Drive\MFES 2019-20\Projetos Dafny\Div.exe
q = 2 r=3
PS C:\Users\João Pascoal Faria\Google Drive\MFES 2019-20\Projetos Dafny> 
```

```
1 ⚠ 0  Not verified | Proof Obligations: 9 | Errors: 1    Ln 16, Col 30   Spaces: 2   UTF-8   CRLF   Dafny   Server up | Idle | Queue: 0
```

The **compiler** first generates C# code, and subsequently an executable that can be run autonomously (Main method).

Demo: Div.dfy

TYPE SYSTEM

TYPE SYSTEM

Value types - instances are immutable; equality & assignment compare/copy values

- Basic types
- Collection types
- Tuple types
- Inductive (and co-inductive) data types

→ *Best suited for modeling data types!*

Reference types – instances are mutable; equality & assignment compare/copy pointers; allow null (if type declared with "?")

- Arrays
- Classes
- Traits

→ *Best suited (namely classes) for modeling system state!*

BASIC TYPES

Description	Declaration	Operators	Example literals
Boolean	bool	== != ! && || ==> <== <==>	true, false
Integer	int		-1, 0, 1
Natural	nat	== != < <= >= > + - * / % -	0, 1, 2
Real	real		-3.0, 1.0, 0.577
Character	char	== != < <= >= >	'a', '\n'

- When combining && and ||, parenthesis are mandatory
- Comparison operators can be chained, as in: 3 == 2+1 == 5-2 < 4
- x.Floor gives the floor of a real number x

ARRAYS

Task	Syntax	Examples
Declare type	**array**<T>	var a : **array**<int> := **new** int[3] [1, 2, 0];
Create instance	**new** T[n]	
Create and initialize	**new** T[n] [a_0, …, a_{n-1}]	
Update element	a[i] := value	a[0], a[1], a[2] := 1, 5, 3;
Select element	a[i]	assert a[0] == 1;
Get length	a.Length	assert a.Length == 3;

Convert to sequence	a[lo..hi] (lo/hi optional,
	lo included, hi excluded)
		assert a[..] == [1, 5, 3];
		assert a[1..2] == [5];

- Multidimensional arrays are also supported

COLLECTIONS

Description	Declaration	Example literals (display expressions)
Set	set<T>	var s : set<int> := {1, 3, 6} *(order & duplicates ignored)*
Sequence	seq<T>	var s : seq<int> := [3, 5, 4, 4]
Multiset	multiset<T>	var m: multiset<int> := multiset{3, 5, 4, 4} *(order ignored)*
Map	map<K, V>	var m: map<string, int> := map["one“ := 1, "two“ := 2]
String	string	“Hello, world\n"

- Dafny also supports potentially infinite sets (iset<T>) and maps (imap<K,V>).
- string is the same as seq<char>

SETS - OPERATORS AND EXPRESSIONS

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

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

expression	description
`|s|`	set cardinality
`e in s`	set membership
`e !in s`	set non-membership

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

Set comprehension

For the full form

$$\texttt{var S := set x1}\overbrace{\texttt{:T1}}^{\textit{type (optional)}}\texttt{, x2:T2 ... | }\overbrace{\texttt{P(x1, x2, ...)}}^{\textit{predicate}}\texttt{ :: }\overbrace{\texttt{Q(x1, x2, ...)}}^{\textit{value (optional)}}$$

the elements of `S` will be all values resulting from evaluation of `Q(x1, x2, ...)` for all combinations of quantified variables `x1, x2, ...` such that predicate `P(x1, x2, ...)` holds. For example,

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

yields `S == {(0, 0), (0, 1), (1, 0), (1,1) }`

SEQUENCES - OPERATORS AND EXPRESSIONS

operator	description
`<`	proper prefix
`<=`	prefix

operator	description
`+`	concatenation

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

slices

```
var t := [3.14, 2.7, 1.41, 1985.44, 100.0, 37.2][1:0:3];
assert |t| == 3 && t[0] == [3.14] && t[1] == [];
assert t[2] == [2.7, 1.41, 1985.44];
```

<u>Sequence comprehension</u>

Syntax: seq(*length*, *index* => *expression*);

Example: assert seq(3, i => i+1) == [1, 2, 3];

MULTISETS - OPERATORS AND EXPRESSIONS

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)

There is no multiset comprehension expression.

MAPS - OPERATORS AND EXPRESSIONS

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

Map comprehension has a syntax similar to set comprehension, as in:

$$\texttt{map}\ \overbrace{\texttt{x}}^{\textit{key}}\ \texttt{:}\ \overbrace{\texttt{int}}^{\textit{type (optional)}}\ |\ \overbrace{\texttt{0 <= x <= 10}}^{\textit{predicate}}\ \texttt{::}\ \overbrace{\texttt{x * x}}^{\textit{Value corresponding to key}}$$

TUPLES

Task	Syntax	Examples
Declare a tuple type with component types T, U, …	(T, U, …)	type Point = (real, real)
Create a tuple with component values t, u, …	(t, u, …)	var p: Point := (1.0, -1.0);
Select the *i*-th component (starting in 0) of a tuple x	x.i	assert p.0 == 1.0;

INDUCTIVE DATA TYPES

Task	Syntax	Examples
Declare type	**datatype** D<T> = *Ctor1* | *Ctor2* | …	**datatype** List<T> = Nil | Cons(head: T, tail: List<T>) **datatype** Semaphore = Green | Yellow | Red

Construct instance	(explicit constructor call)	var list1 := Cons(5, Nil);
		var sem1 := Green;

Update instance	d[*CtorParam* := *Value*]	list1 := list1[head := 1];

Constructor check	d.Ctor?	function Length(x: List<T>) : nat {
		  if x.Cons? then 1 + Length(x.tail) else 0
		}

Field selector	d.CtorParam	

Case analysis	Match expression
	(for a match statement, use "=> *stmt;*" instead of "=> *expression*")
	{} are optional
		function Length(x: List<T>) : nat {
		  match x {
		    case Nil => 0
		    case Cons(h, t) => 1 + Length(t) }
		}

CLASSES (1)

```
              class Account {
fields            var balance : real;
constructors      constructor (balance: real) {this.balance := balance; }
                  method deposit(amount: real)  modifies this { balance := balance + amount ;}
methods           method withdraw(amount: real) modifies this { balance := balance - amount ;}
                  method getBalance() returns (res : real) { return balance; }
              }
```

```
method Main() {
   var a := new Account(10.0);
   a.deposit(5.0);
   var b := a.getBalance();
   assert b == 15.0;
}
```

- Classes may **extend** other classes or traits.
- Methods may be declared **static**.
- Classes may also include functions and predicates.
- Constructors may be anonymous (as above) or have a name.
- The supertype for all reference types is **object**.
- Immutable fields are declared with **const** instead of var.

CLASSES (2)

A common convention is to have a predicate named **Valid** that describes the **class invariant,** required by all (public) methods and ensured by all (public) constructors and modifier methods.

With the :autocontracts attribute, Dafny takes care automatically of the class invariant enforcement (requires/ensured Valid()) and frame generation (reads/modifies).

```
class Account {

  var balance : real;

  predicate Valid()
   reads this
  {balance >= 0}

  constructor (balance: real)
   requires balance >= 0
   ensures Valid()
  { this.balance := balance; }

  method withdraw(amount: real)
   requires Valid() && 0 < amount <= balance
   modifies this
   ensures Valid()
  { balance := balance - amount ;}

  method getBalance() returns (res : real)
   requires Valid()
  { return balance; }

}
```

TRAITS

Traits are **abstract classes**, so cannot have constructors and cannot be directly instantiated.

May have **abstract methods**, declared by omitting the body {…}; only abstract methods may be redeclared in classes that extend the trait.

```
trait Shape
{
  function method Width(): real
    reads this
  method Move(dx: real, dy: real)
    modifies this
  method MoveH(dx: real)
    modifies this
  {
    Move(dx, 0.0);
  }
}
```

extends

```
class UnitSquare extends Shape
{
  var x: real, y: real
  function method Width(): real {
    1.0
  }
  method Move(dx: real, dy: real)
    modifies this
  {
    x, y := x + dx, y + dy;
  }
}
```

PROGRAM ORGANIZATION

& BASIC SPECIFICATION & VERIFICATION CONSTRUCTS

METHODS

```
                                        type
           attributes            name   params        in-parameters                          out-parameters
method {:att1}{:att2} M<T1, T2>(a: A, b: B, c: C) returns (x: X, y: Y, z: Z)
  requires Pre        precondition (boolean expression)
  modifies Frame      objects whose fields may be updated by the method
  ensures Post        postcondition (boolean expression)
  decreases Rank      variant function (to prove termination of recursive methods)
{
  Body   imperative style (statement or sequence of statements)
}
```

- Precondition: condition on the input params and initial object state that must hold on entry.
- Postcondition: condition on the output params and final object state (possibly in relation with input params and initial object state) that must hold on exit (assuming the precondition holds on entry).
- Initial object states are accessed with **old**(…).
- Newly created objects may be indicated in additional clause **fresh**(obj).
- Together, the pre and postcondition define the method semantics.
- Methods marked as **ghost** don't go into the executable code.

FUNCTIONS AND PREDICATES

```
                                     type                               result
                  attributes    name parameters        parameters       type
function {:att1}{:att2} F<T1, T2>(a: A, b: B, c: C): T
   requires Pre    precondition
   reads Frame    objects (includes arrays) whose fields the function body may depend on
   ensures Post    postcondition (usually not needed)
   decreases Rank    variant function (for recursive functions)
{
   Body    functional style (expression without side-efefects)
}
```

- By default, functions are ghost (don't go into the executable). To make a function non-ghost, declare it is as **function method**.
- Functions that return a bool result may instead be declared with the **predicate** keyword, removing the declaration of the result type.
- The function result is accessed in the postcondition as F(a, b, c) (like a function invocation).

EXPRESSIONS

(In addition to expressions already presented)

Name	Syntax	Example
Conditional expression	**if** condition **then** value-if-true **else** value-if-false	**if** x > y **then** x **else** y
Universal quantifier	**forall** x:X, y:Y, … :: P(x, y, …) ==> Q(x, y, …) *(P – finite search scope; Q – property to check)*	**forall** k :: 0<=k<|a| ==> a[k]=x
Existential quantifier	**exists** x: X, y: Y, … :: P(x, y, …) && Q(x, y, …) *(P – finite search scope; Q – property to check)*	**exists** k :: 0<=k<|a| && a[k]==x
Let expression	**var** v := *value; expression-on-v* **var** v :| *predicate-on-v; expression-on-v*	**var** sum := x + y; sum * sum **var** x :| 0 <= x <= 100; x * x

STATEMENTS (1/2)

Name	Syntax / examples	Notes

Variable declaration	**var** x, y : int;
	var x := 1;
		Explicit type
		Inferred type

Update	x := 1;
	x, y := y, x; // swap
	y :| y in Y;
		Simple assignment
		Multiple (parallel) assignment
		Assign such that (choice)

If	**if** *condition* { *statement(s)* } [**else** {*statement(s)*}]	{ } are mandatory

Multibranch if	**if** { **case** Cond1 => stmt1;
	case Cond2 => stmt2; …}
		Guard conditions are unordered and at least one needs to evaluate to true.

Binding guard	**if** x :| P(x) {…} [**else** {…}]	If a value x exists that satisfies *P*(*x*), the "then" part is executed with such a *x*.

STATEMENTS (2/2)

Name	Syntax / examples	Notes

While	**while** *guard-condition*
	    invariant loopInvariant
	    decrases loopVariant
	{
	    statement(s)
	}
		The loop invariant(s) must hold on entry, on exit, and before/after each iteration.
		The loop variant is a strictly decreasing function, integer (or similar), non-negative.

Forall	**forall** x:X, y:Y, … | P(x, y, …)
	  [**ensures** Q(x, y, …)]
	{*body*}
		Executes the body in parallel for all quantified values in the specified range (P), in 3 use cases:
		• *Assign* - simultaneous assignment of array elements or object fields;
		• *Call* - calls to a ghost method without side effects;
		• *Proof* – with "ensures" expression to be proved by the body.

Return	**return** x+1;
	return min, max;
		Simple return
		Multiple return

MODULES

Modules provide a way to group together related types, classes, methods, functions, and other modules, as well as control the scope of declarations (like namespaces).

It is possible to abstract over modules to separate an implementation from an interface (refinement relationship, possibly strengthening postconditions).

Modules may import other modules for code reuse.

```
abstract module Sorting {
  type T = int
  predicate isSorted(a: array<T>) reads a {…}
  predicate isPermutation(a: seq<T>, b: seq<T>) {…}
  method sort(a: array<T>)
    modifies a
    ensures isSorted(a) && isPermutation(a[..], old(a[..]))
}
```

Refine

```
module BubbleSort refines Sorting {
  method sort(a: array<T>) {…}
}
```

Import

```
module TestSorting {
  import opened BubbleSort
  …
  method Main() {
    var a := new int[5];
    a[0], a[1], a[2], a[3], a[4] := 9, 4, 6, 3, 8;
    sort(a);
    printArray(a);
  }
}
```

Demo BubbleSort.dfy

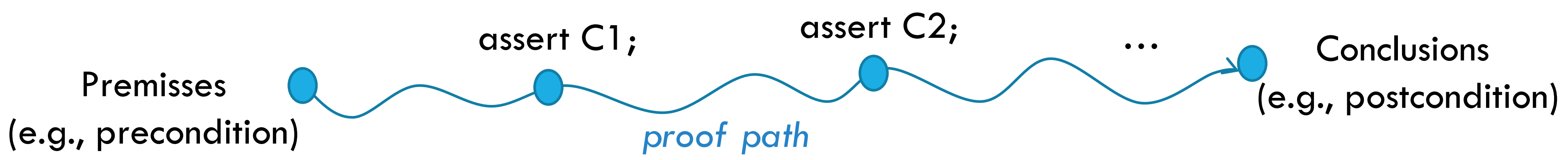
ADVANCED SPECIFICATION & VERIFICATION CONSTRUCTS

ASSERT AND ASSUME

The "requires", "ensures" & "invariant" clauses declare assertions that must hold on specific parts of the program and are statically checked by Dafny.

Other assertions may be declared with the **assert** statement (e.g., for testing, debugging, or proof purposes).

When Dafny is unable to check a proof obligation, providing assertions in the proof path may help Dafny conducting the proof, by breaking it down into smaller steps.

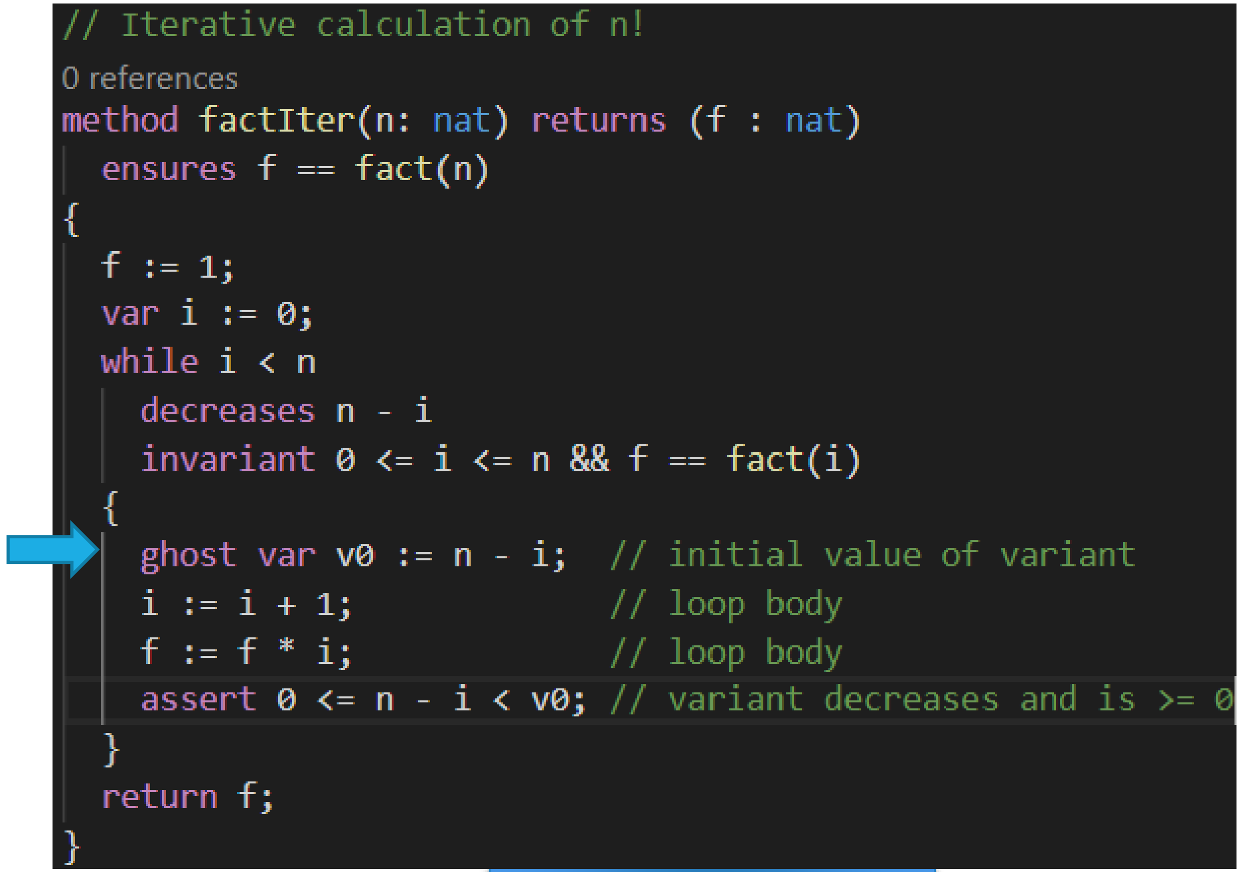


The **assume** statement instructs Dafny to accept as true (without verification) the given expression; useful for incremental development and debugging purposes, need to be converted to "assert" statements or removed before executable code generation.

GHOST VARIABLES

Used only for specification & verification purposes (don't go into the executable code).

Example to prove loop termination (done behind the scenes by Dafny):



```
// Iterative calculation of n!
0 references
method factIter(n: nat) returns (f : nat)
  ensures f == fact(n)
{
  f := 1;
  var i := 0;
  while i < n
    decreases n - i
    invariant 0 <= i <= n && f == fact(i)
  {
    ghost var v0 := n - i;  // initial value of variant
    i := i + 1;             // loop body
    f := f * i;             // loop body
    assert 0 <= n - i < v0; // variant decreases and is >= 0
  }
  return f;
}
```

Demo: Factorial2.dfy

LEMMAS

Sometimes there are steps of logic required to prove a program correct, but they are too complex for Dafny to discover and use on its own.

When this happens, we can often give Dafny assistance by providing a lemma.

This is done by declaring a method with the **lemma** keyword.

Lemmas are implicitly ghost methods.

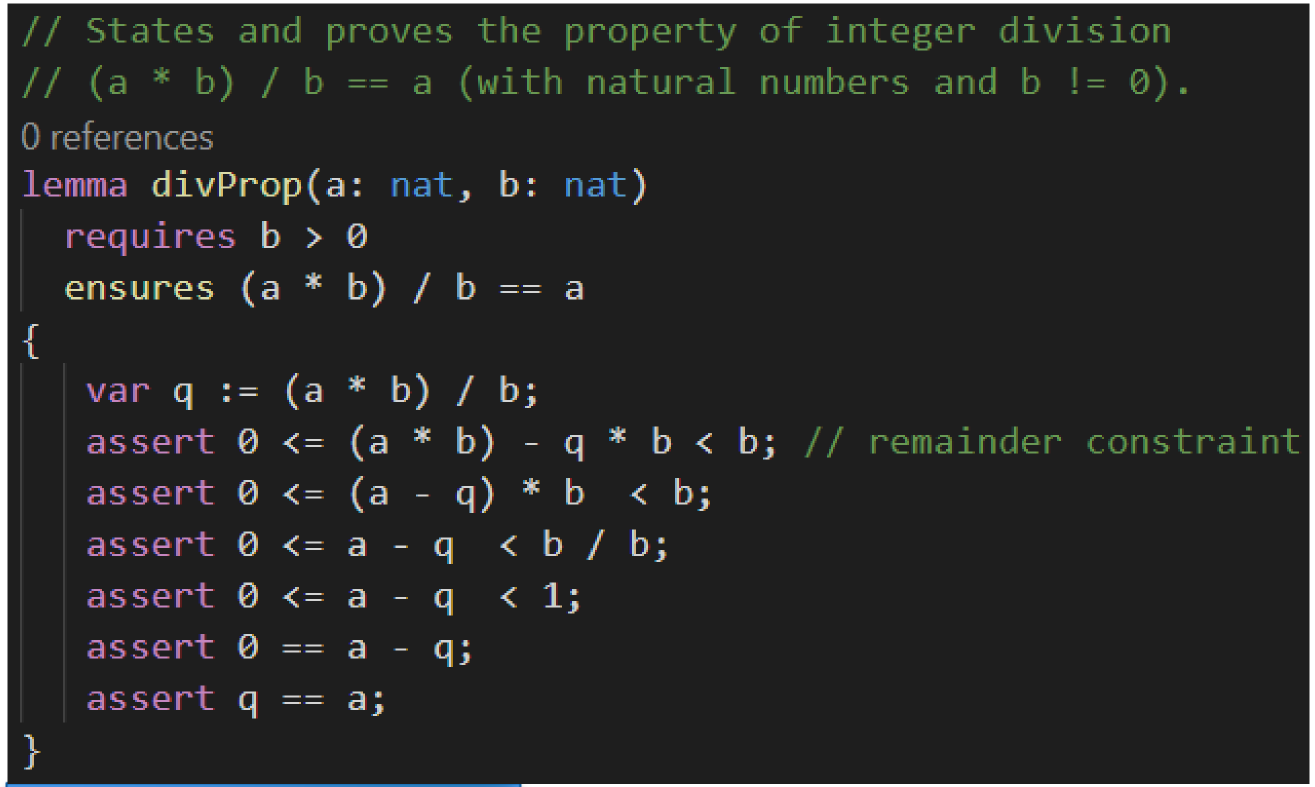
Lemmas need to be explicitly invoked (like method calls) when needed.

The header describes the property (for all values of parameters, if pre-conditions are satisfied, then the post-conditions hold).

The body gives the proof steps.

```
lemma Name(parameters)
  requries pre-conditions
  ensures post-conditions
{
   proof steps (with assert, calc, forall,
    conditionals, other lemma invocations, etc.)
}
```

LEMMAS - EXAMPLES



```
// States and proves the property of integer division
// (a * b) / b == a (with natural numbers and b != 0).
0 references
lemma divProp(a: nat, b: nat)
  requires b > 0
  ensures (a * b) / b == a
{
   var q := (a * b) / b;
   assert 0 <= (a * b) - q * b < b; // remainder constraint
   assert 0 <= (a - q) * b  < b;
   assert 0 <= a - q  < b / b;
   assert 0 <= a - q  < 1;
   assert 0 == a - q;
   assert q == a;
}
```

Demo1: Div_Lemmas.dfy

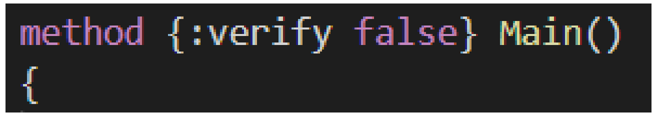
Demo2: QuickSort.dfy

Demo3: PriorityQueue.dfy

ATTRIBUTES

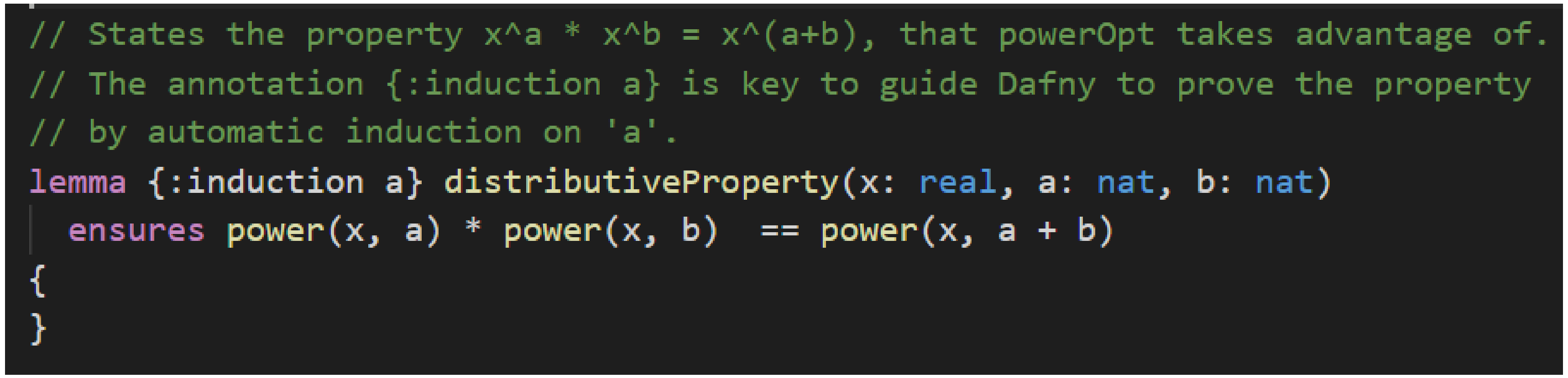
Attributes are annotations in the code, according to the syntax **{:attribute value}**, that can be used to control the behavior of the Dafny verifier and compiler.

Examples:



```
method {:verify false} Main()
{
```

Disables static verification for this method



```
// States the property x^a * x^b = x^(a+b), that powerOpt takes advantage of.
// The annotation {:induction a} is key to guide Dafny to prove the property
// by automatic induction on 'a'.
lemma {:induction a} distributiveProperty(x: real, a: nat, b: nat)
  ensures power(x, a) * power(x, b)  == power(x, a + b)
{
}
```

Lemma may be proved by induction on "a"

Demo: Power.dfy

REFERENCES & FURTHER READING

Dafny Tutorials, https://rise4fun.com/Dafny/tutorial

Dafny Lecture Notes (see copy in Moodle)

Dafny Reference Manual (see copy in Moodle)

Dafny extension for Visual Studio Code, https://marketplace.visualstudio.com/items?itemName=correctnessLab.dafny-vscode