From “Program Proofs”, K. Rustan M. Leino, MIT Press, 2023.

Appendix A

Dafny Syntax Cheat Sheet

This appendix shows snippets of Dafny syntax. These are intended to jog your memory of, or to suggest, how to use various constructs in Dafny, not to give you a tutorial introduction of the constructs. The snippets are therefore given without much explanation. To find uses of the constructs in this book, consult the Index. For full details, see the Dafny reference manual [36].

A.0. Declarations

A Dafny program is a hierarchy of nested modules. Dependencies among modules are announced by **`import`** declarations. A program's import relation must not contain cycles. The export set of a module determines which of the module's declarations are visible to importers.

```
module MyModule {
  export
    provides A, B, C
    reveals D, E, F
  import L = LibraryA // L is a local name for imported module LibraryA
  import LibraryB // shorthand for: import LibraryB = LibraryB

  // declarations of types and module-level members...
}
```

The outermost module of a program is implicit. Therefore, small programs can define methods and functions without needing to wrap them inside a **`module`** declaration.

A.0.0. Types and type declarations

Here are some example type declarations:

```
datatype Color = Brown | Blue | Hazel | Green
datatype Unary = Zero | Suc(Unary)
datatype List<X> = Nil | Cons(head: X, tail: List<X>)

class C<X> {
  // class member declarations...
}

type OpaqueType

type TypeSynonym = int
```

The `X` in these examples is a type parameter.

Examples of types:

```
bool          int           nat           real
set<X>        seq<X>        multiset<X>   map<X, Y>
char          string        X -> Y
()            (X, Y)        (X, Y, Z)
array<X>      array?<X>     array2<X>
object        object?       MyClass<X>    MyClass?<X>
```

The types shown here with parentheses denote 0-, 2-, and 3-tuples.

A.0.1. Member declarations

```
method M(a: A, b: B) returns (c: C, d: D)
  requires Pre
  modifies obj0, obj1, objectSet
  ensures Post // old(E) refers to the value of E on entry to the method
  decreases E0, E1, E2
```

A **constructor** (in a **class**) or **lemma** has the same syntax as a **method**. For an anonymous **constructor**, omit the name `M`.

```
function F(a: A, b: B): C
  requires Pre
  reads obj0, obj1, objectSet
  ensures Post // F(a, b) refers to the result of the function
  decreases E0, E1, E2
```

If `C` is **bool**, then the first line of the function declaration can be written as

```
predicate F(a: A, b: B)
```

Declarations of fields and constants:

```
var b: B // mutable field, can be used only in classes
```

```
const n: nat
const greeting: string := "hello"
const year := 1402
```

function, **predicate**, **var**, and **const** declarations can be preceded by **ghost**.

A.1. Statements

Each primitive statement ends with a `;` (semi-colon). In contrast, a statement with a body (enclosed in curly braces) does not end with a `;`.

Declaration of local variables:

```
var x: X;
```

The "`: X`" can be omitted if the type can be inferred. When declaring more than one variable, the `:` (colon) binds stronger than `,` (comma). That is,

```
var x, y: Y;
```

declares `y` to have type `Y` and leaves the type of `x` to be inferred.

Assignments:

```
x := E;        // := is pronounced "gets" or "becomes" (NOT "equals"!)
x, y := E, F;  // simultaneous assignment
x :| E;        // assign x a value that makes E hold (assign such that)
```

A declaration of a variable and an assignment to the same variable can be combined into one statement, like **var** `x := E;`.

Dynamic allocation of objects and arrays:

```
c := new C(...);
a := new T[n];
a := new T[n](i => ...);
```

Method calls with 0, 1, and 2 out-parameters:

```
MethodWithNoResults(E, F);
x := MethodWithOneResult(E, F);
x, y := MethodWithTwoResults(E, F);
```

Other primitive statements:

```
assert E;        return;        return E, F, G;        new;
```

Some composite statements:

```
if E {
  // statements...
} else {
  // statements...
}
```

```
if {
  case E0 => // statements...
  case E1 => // statements...
}

match E {
  case Pattern0(x, y) => // statements...
  case Pattern1(z, _) => // statements...
}

while Guard
  invariant Inv
  modifies obj0, obj1, objectSet
  decreases E0, E1, E2
{
  // statements...
}

forall x: X | Range {
  // assignment statement
}

calc {
  E0;
==  { assert HintWhyE0EqualsE1; }
  E1;
==  { LemmaThatExplainsWhyE1EqualsE2(); }
  E2;
}
```

In the **if** statement (unlike in the **if-then-else** *expression*), the **else** branch is optional, and the curly braces are required. When an **if-case** or **match** statement is given last in a statement list, the curly braces that surround the **case**s can be omitted. Without the curly braces, each **case** is stylistically not indented but kept flush with the **if** or **match** keyword. The **forall** statement is an aggregate statement that simultaneously performs the given assignment statement for every value of `x` that satisfies `Range`. The **calc** statement is used to write a structured proof calculation.

A.2. Expressions

Figure A.0 shows common operators. Operators in the same section have the same binding power, and the sections are ordered from lowest to highest binding power.

`<==>`				iff (lowest binding power)
`==>`	`<==`			implication, reverse implication
`&&`	`||`			and, or
`==`	`!=`			equality, disequality
`<`	`<=`	`=>`	`>`	inequality comparisons
`in`	**`!in`**			collection membership
`!!`				set disjointness
`+`	`-`			plus/union/concatenation/merge, minus
`*`	`/`	`%`		multiplication/intersection, division, modulus
`_` **`as int`**				conversion to integer
`!`	`-`			boolean not, unary negation
`_.x`				member selection
`_[_]`		`_[_ := _]`		element selection, update
`_[_ .. _]`				subrange
`_[.. _]`		`_[_ ..]`		take, drop
`_[..]`				array-elements to sequence

Figure A.0. Operator binding powers.

For sets, `<=` denotes subset, `+` denotes union, `*` denotes intersection, and `-` denotes set difference. For multisets, those operators denote the analogous multiset operations. For sequences, `<=` denotes prefix and `+` denotes concatenation. For maps, `+` denotes map merge (where the right-hand operand takes priority) and `-` denotes map domain subtraction. The operator `<` is the strict version of `<=`.

In the member-selection expression `E.x`, `E` is an expression (typically a reference or datatype value) and `x` is a member of the type of `E`.

The expression `E[J]` selects member `J` from `E`, where `E` is an array, sequence, or map and `J` either denotes an index into the array or sequence or denotes a key in the map. For a multiset `E`, `E[J]` denotes the multiplicity of element `J`. The elements of a tuple are selected using numerically named members; for example, the 3 members of a triple `E` are selected by `E.0`, `E.1`, and `E.2`.

If `E` is a sequence, map, or multiset, the update expression `E[J := V]` returns a collection like `E` except that element `J`, key `J`, or the multiplicity of `J`, respectively, has been replaced by `V`.

For an array or sequence `E`, the subsequence expression `E[lo..hi]` is the sequence of `hi - lo` elements from `E` starting at `lo`. If the lower bound is omitted, it defaults to `0`, and if the upper bound is omitted, it defaults to the length of the array or sequence. For an array `E`, the expression `E[..]`, which has the same meaning as `E[0..E.Length]`, obtains the sequence of all elements of `E`.

If `E` is a set, multiset, or sequence, then the expression `|E|` denotes the total number of elements of `E` (which is known as the *cardinality* of the set or multiset, and the length

of the sequence). The expression `E.Keys` denotes the set of keys in a map `E`. The number of elements in an array `E` is written `E.Length`, and the lengths of the dimensions of a 2-dimensional array `E` are written `E.Length0` and `E.Length1`.

The following table shows tuples, set displays, multiset displays, sequence displays, and map displays with 0 and 3 elements (or fewer for the set, if some of `a`, `b`, and `c` are equal):

```
()          (a, b, c)
{}          {a, b, c}
multiset{}  multiset{a, b, c}
[]          [a, b, c]
map[]       map[x := a, y := b, z := c]
```

Here are some literals and other expressions:

```
44             1.618          'D'            "hello"
this           null           old(E)         fresh(E)

seq(E, i => ...)     // sequence comprehension

if E then E0 else E1

match E {
  case Pattern0(x, y) => E0
  case Pattern1(z, _) => E1
}

assert E0; E1        // like E1, but first asserts E0
MyLemma(); E         // like E1, but first calls MyLemma()

var x := E0; E1      // pronounced "let x be E0 in E1"

set x: X | Range
forall x: X :: Expr  // Expr typically has the form E0 ==> E1
exists x: X :: Expr  // Expr often uses &&, seldom ==>
```

In the **`if`**-**`then`**-**`else`** expression (unlike in the **`if`** *statement*), the **`else`** branch is required, and there are no curly braces around `E0` and `E1` (except if they happen to be set-display expressions).

Unless you're nesting one **`match`** expression inside another, you can omit the curly braces. Without the curly braces, each **`case`** is stylistically not indented but kept flush with the **`match`** keyword.