Dafny Reference Manual

K. Rustan M. Leino, David R. Cok, and the Dafny contributors

June 26, 2024

Abstract: This is the Dafny reference manual; it describes the Dafny programming language and how to use the Dafny verification system. Parts of this manual are more tutorial in nature in order to help the user understand how to do proofs with Dafny.

Acknowledgements

Rustan Leino is the designer of the Dafny language and the chief implementor of the dafny tools. Leino began this document, with additions from Richard Ford, David R. Cok, and other members of the Dafny project. Many others contributed to the implementation and to documenting various aspects of the language, as described on the project website at https://github.com/dafny-lang/dafny. David R. Cok is the primary author and editor.

Contents

Abstract: This is the Dafny reference manual; it describes the Dafny programming language and how to use the Dafny verification system. Parts of this manual are more tutorial in nature in order to help the user understand how to do proofs with Dafny.

(Link to current document as html)

1. Introduction

Dafny (Leino 2010) is a programming language with built-in specification constructs, so that verifying a program's correctness with respect to those specifications is a natural part of writing software. The Dafny static program verifier can be used to verify the functional correctness of programs. This document is a reference manual for the programming language and a user guide for the `dafny` tool that performs verification and compilation to an executable form.

The Dafny programming language is designed to support the static verification of programs. It is imperative, sequential, supports generic classes, inheritance and abstraction, methods and functions, dynamic allocation, inductive and coinductive datatypes, and specification constructs. The specifications include pre- and postconditions, frame specifications (read and write sets), and termination metrics. To further support specifications, the language also offers updatable ghost variables, recursive functions, and types like sets and sequences. Specifications and ghost constructs are used only during verification; the compiler omits them from the executable code.

The `dafny` verifier is run as part of the compiler. As such, a programmer interacts with it in much the same way as with the static type checker—when the tool produces errors, the programmer responds by changing the program's type declarations, specifications, and statements.

(This document typically uses "Dafny" to refer to the programming language and `dafny` to refer to the software tool that verifies and compiles programs in the Dafny language.)

The easiest way to try out the Dafny language is to download the supporting tools and documentation and run `dafny` on your machine as you follow along with the Dafny tutorial. The `dafny` tool can be run from the command line (on Linux, MacOS, Windows or other platforms) or from IDEs such as emacs and VSCode, which can provide syntax highlighting and code manipulation capabilities.

The verifier is powered by Boogie (Barnett et al. 2006; Leino 2008b; Leino and Rümmer 2010) and Z3 (Moura and Bjørner 2008).

From verified programs, the `dafny` compiler can produce code for a number of different backends: the .NET platform via intermediate C# files, Java, Javascript, Go, and C++. Each language provides a basic Foreign Function Interface (through uses of `:extern`) and a supporting runtime library.

This reference manual for the Dafny verification system is based on the following references: (Leino 2010), (Leino 2008a), (Leino and Polikarpova 2013), (Leino and Moskal 2014a), Co-induction Simply.

The main part of the reference manual is in top down order except for an initial section that deals with the lowest level constructs.

The details of using (and contributing to) the dafny tool are described in the User Guide (Section 13).

1.1. Dafny 4.0

The most recent major version of the Dafny language is Dafny 4.0, released in February 2023. It has some backwards incompatibilities with Dafny 3, as decribed in the migration guide.

The user documentation has been expanded with more examples, a FAQ, and an error explanation catalog. There is even a new book, Program Proofs by Dafny designer Rustan Leino.

The IDE now has a framework for showing error explanation information and corresponding quick fixes are being added, with refactoring operations on the horizon.

More details of 4.0 functionality are described in the release notes. ## 1.2. Dafny Example {#sec-example} To give a flavor of Dafny, here is the solution to a competition problem.

```
// VSComp 2010, problem 3, find a 0 in a linked list and return
// how many nodes were skipped until the first 0 (or end-of-list)
// was found.
// Rustan Leino, 18 August 2010.
//
// The difficulty in this problem lies in specifying what the
// return value 'r' denotes and in proving that the program
// terminates.  Both of these are addressed by declaring a ghost
// field 'List' in each linked-list node, abstractly representing
// the linked-list elements from the node to the end of the linked
// list.  The specification can now talk about that sequence of
// elements and can use 'r' as an index into the sequence, and
// termination can be proved from the fact that all sequences in
// Dafny are finite.
//
// We only want to deal with linked lists whose 'List' field is
// properly filled in (which can only happen in an acyclic list,
// for example).  To that end, the standard idiom in Dafny is to
// declare a predicate 'Valid()' that is true of an object when
// the data structure representing that object's abstract value
// is properly formed.  The definition of 'Valid()' is what one
// intuitively would think of as the ''object invariant'', and
// it is mentioned explicitly in method pre- and postconditions.
//
// As part of this standard idiom, one also declares a ghost
// variable 'Repr' that is maintained as the set of objects that
// make up the representation of the aggregate object--in this
// case, the Node itself and all its successors.
module {:options "--function-syntax:4"} M {
class Node {
  ghost var List: seq<int>
  ghost var Repr: set<Node>
  var head: int
  var next: Node? // Node? means a Node value or null

  ghost predicate Valid()
    reads this, Repr
  {
    this in Repr &&
    1 <= |List| && List[0] == head &&
    (next == null ==> |List| == 1) &&
    (next != null ==>
      next in Repr && next.Repr <= Repr && this !in next.Repr &&
      next.Valid() && next.List == List[1..])
  }

  static method Cons(x: int, tail: Node?) returns (n: Node)
    requires tail == null || tail.Valid()
    ensures n.Valid()
    ensures if tail == null then n.List == [x]
                            else n.List == [x] + tail.List
  {
    n := new Node;
    n.head, n.next := x, tail;
    if (tail == null) {
      n.List := [x];
      n.Repr := {n};
    } else {
      n.List := [x] + tail.List;
      n.Repr := {n} + tail.Repr;
    }
  }
}

method Search(ll: Node?) returns (r: int)
  requires ll == null || ll.Valid()
  ensures ll == null ==> r == 0
  ensures ll != null ==>
            0 <= r && r <= |ll.List| &&
            (r < |ll.List| ==>
              ll.List[r] == 0 && 0 !in ll.List[..r]) &&
            (r == |ll.List| ==> 0 !in ll.List)
{
  if (ll == null) {
    r := 0;
  } else {
    var jj,i := ll,0;
    while (jj != null && jj.head != 0)
      invariant jj != null ==>
            jj.Valid() &&
            i + |jj.List| == |ll.List| &&
            ll.List[i..] == jj.List
      invariant jj == null ==> i == |ll.List|
      invariant 0 !in ll.List[..i]
      decreases |ll.List| - i
    {
      jj := jj.next;
      i := i + 1;
    }
    r := i;
  }
}

method Main()
{
  var list: Node? := null;
  list := list.Cons(0, list);
  list := list.Cons(5, list);
  list := list.Cons(0, list);
  list := list.Cons(8, list);
  var r := Search(list);
  print "Search returns ", r, "\n";
  assert r == 1;
}
}
```

2. Lexical and Low Level Grammar

As with most languages, Dafny syntax is defined in two levels. First the stream of input characters is broken up into *tokens*. Then these tokens are parsed using the Dafny grammar.

The Dafny grammar is designed as an *attributed grammar*, which is a conventional BNF-style set of productions, but in which the productions can have arguments. The arguments control some alternatives within the productions, such as whether an alternative is allowed or not in a specific context. These arguments allow for a more compact and understandable grammar.

The precise, technical details of the grammar are presented together in Section 17. The expository parts of this manual present the language structure less formally. Throughout this document there are embedded hyperlinks to relevant grammar sections, marked as grammar.

2.1. Dafny Input

Dafny source code files are readable text encoded in UTF-8. All program text other than the contents of comments, character, string and verbatim string literals consists of printable and white-space ASCII characters, that is, ASCII characters in the range `!` to `~`, plus space, tab, carriage return and newline (ASCII 9, 10, 13, 32) characters. (In some past versions of Dafny, non-ASCII, unicode representations of some mathematical symbols were permitted in Dafny source text; these are no longer recognized.)

String and character literals and comments may contain any unicode character, either directly or as an escape sequence.

2.2. Tokens and whitespace

The characters used in a Dafny program fall into four groups:

- White space characters: space, tab, carriage return and newline
- alphanumerics: letters, digits, underscore (`_`), apostrophe (`'`), and question mark (`?`)
- punctuation: ``(){}[],.`;``
- operator characters (the other printable characters)

Except for string and character literals, each Dafny token consists of a sequence of consecutive characters from just one of these groups, excluding white-space. White-space is ignored except that it separates tokens and except in the bodies of character and string literals.

A sequence of alphanumeric characters (with no preceding or following additional alphanumeric characters) is a *single* token. This is true even if the token is syntactically or semantically invalid and the sequence could be separated into more than one valid token. For example, `assert56` is one identifier token, not a keyword `assert` followed by a number; `ifb!=0` begins with the token `ifb` and not with the keyword `if` and token `b`; `0xFFFFZZ` is an illegal token, not a valid hex number `0xFFFF` followed by an identifier `ZZ`. White-space must be used to separate two such tokens in a program.

Somewhat differently, operator tokens need not be separated. Only specific sequences of operator characters are recognized and these are somewhat context-sensitive. For example, in `seq<set<int>>`, the grammar knows that `>>` is two individual `>` tokens terminating the nested type parameter lists; the right shift operator `>>` would never be valid here. Similarly, the sequence `==>` is always one token; even if it were invalid in its context, separating it into `==` and `>` would always still be invalid.

In summary, except for required white space between alphanumeric tokens, adding or removing white space between tokens can never result in changing the meaning of a Dafny program. For most of this document, we consider Dafny programs as sequences of tokens.

2.3. Character Classes

This section defines character classes used later in the token definitions. In this section

- a backslash is used to start an escape sequence (so for example `'\n'` denotes the single linefeed character)
- double quotes enclose the set of characters constituting a character class
- enclosing single quotes are used when there is just one character in the class (perhaps expressed with a `\` escape character)
- `+` indicates the union of two character classes
- `-` is the set-difference between the two classes
- `ANY` designates all unicode characters.

name	description
letter	ASCII upper or lower case letter; no unicode characters
digit	base-ten digit ("0123456789")
posDigit	digits, excluding 0 ("123456789")
posDigitFrom2	digits excluding 0 and 1 ("23456789")
hexdigit	a normal hex digit ("0123456789abcdefABCDEF")
special	'?_"
cr	carriage return character (ASCII 10)
lf	line feed character (ASCII 13)
tab	tab character (ASCII 9)
space	space character (ASCII 32)
nondigitIdChar	characters allowed in an identifier, except digits (letter + special)
idchar	characters allowed in an identifier (nondigitIdChar + digits)
nonidchar	characters not in identifiers (ANY - idchar)
charChar	characters allowed in a character constant (ANY - '' ' - '\' - cr - lf)
stringChar	characters allowed in a string constant (ANY - '"' - '\' - cr - lf)
verbatimStringChar	characters allowed in a verbatim string constant (ANY - '"')

The *special* characters are the characters in addition to alphanumeric characters that are allowed to appear in a Dafny identifier. These are

- `'` because mathematicians like to put primes on identifiers and some ML programmers like to start names of type parameters with a `'`,
- `_` because computer scientists expect to be able to have underscores in identifiers, and
- `?` because it is useful to have `?` at the end of names of predicates, e.g., `Cons?`.

A `nonidchar` is any character except those that can be used in an identifier. Here the scanner generator will interpret `ANY` as any unicode character. However, `nonidchar` is used only to mark the end of the `!in` token; in this context any character other than whitespace or printable ASCII will trigger a subsequent scanning or parsing error.

2.4. Comments

Comments are in two forms.

- They may go from `/*` to `*/` .
- They may go from `//` to the end of the line.

A comment is identified as a token during the tokenization of input text and is then discarded for the purpose of interpreting the Dafny program. (It is retained to enable auto-formatting and provide accurate source locations for error messages.) Thus comments are token separators: `a/*x*/b` becomes two tokens `a` and `b`.

Comments may be nested, but note that the nesting of multi-line comments is behavior that is different from most programming languages. In Dafny,

```
method m() {
  /* comment
     /* nested comment
     */
     rest of outer comment
  */
}
```

is permitted; this feature is convenient for commenting out blocks of program statements that already have multi-line comments within them. Other than looking for end-of-comment delimiters, the contents of a comment are not interpreted. Comments may contain any characters.

Note that the nesting is not fool-proof. In

```
method m() {
  /* var i: int;
     // */ line comment
     var j: int;
  */
}
```

and

```
method m() {
  /* var i: int;
     var s: string := "a*/b";
     var j: int;
   */
}
```

the `*/` inside the line comment and the string are seen as the end of the outer comment, leaving trailing text that will provoke parsing errors.

2.5. Documentation comments

Like many other languages, Dafny permits *documentation comments* in a program file. Such comments contain natural language descriptions of program elements and may be used by IDEs and documentation generation tools to present information to users.

In Dafny programs. * Documentation comments (a) either begin with `/**` or (b) begin with `//` or `/`*in specific locations * Doc-comments may be associated with any declaration, including type definitions, export declarations, and datatype constructors. * They may be placed before or after the declaration. * If before, it must be a*`/`**comment and may not have any blank or white-space lines between the comment and the declaration. * If after, any comments are placed after the signature (with no intervening lines), but before any specifications or left-brace that starts a body, and may be**`//`***or***`/`*or*`/`comments. * If doc-comments are in both places, only the comments after the declaration are used. * Doc-comments after the declaration are preferred. * If the first of a series of single-line or multi-line comments is interpreted as a doc-string, then any subsequent comments are appended to it, so long as there are no intervening lines, whether blank, all white-space or containing program text. * The extraction of the doc-string from a multiline comment follow these rules * On the first line, an optional*right after*`/`and an optional space are removed, if present * On other lines, the indentation space (with possibly one star in it) is removed, as if the content was supposed to align with A if the comment started with`/** A' for example. * The documentation string is interpreted as plain text, but it is possible to provide a user-written plugin that provides other interpretations. VSCode as used by Dafny interprets any markdown syntax in the doc-string.

Here are examples:

```
const c0 := 8
/** docstring about c0 */

/** docstring about c1 */
const c1 := 8

/** first line of docstring */
const c2 := 8
```

```
/** second line of docstring */

const c3 := 8
// docstring about c3
// on two lines

const c4 := 8

// just a comment


// just a comment
const c5 := 8
```

Datatype constructors may also have comments:

```
datatype T =  // Docstring for T
  | A(x: int,
      y: int) // Docstring for A
  | B()       /* Docstring for B */ |
    C()       // Docstring for C

/** Docstring for T0*/
datatype T0 =
  | /** Docstring for A */
    A(x: int,
      y: int)
  | /** Docstring for B */
    B()
  | /** Docstring for C */
    C()
```

As can `export` declarations:

```
module M {
const A: int
const B: int
const C: int
const D: int

export
  // This is the eponymous export set intended for most clients
  provides A, B, C


export Friends extends M
  // This export set is for clients who need to know more of the
  // details of the module's definitions.
  reveals A
  provides D
}
```

2.6. Tokens (grammar)

The Dafny tokens are defined in this section.

2.6.1. Reserved Words

Dafny has a set of reserved words that may not be used as identifiers of user-defined entities. These are listed here.

In particular note that

- `array`, `array2`, `array3`, etc. are reserved words, denoting array types of given rank. However, `array1` and `array0` are ordinary identifiers.
- `array?`, `array2?`, `array3?`, etc. are reserved words, denoting possibly-null array types of given rank, but not `array1?` or `array0?`.
- `bv0`, `bv1`, `bv2`, etc. are reserved words that denote the types of bitvectors of given length. The sequence of digits after 'array' or 'bv' may not have leading zeros: for example, `bv02` is an ordinary identifier.

2.6.2. Identifiers

In general, an `ident` token (an identifier) is a sequence of `idchar` characters where the first character is a `nondigitIdChar`. However tokens that fit this pattern are not identifiers if they look like a character literal or a reserved word (including array or bit-vector type tokens). Also, `ident` tokens that begin with an `_` are not permitted as user identifiers.

2.6.3. Digits

A `digits` token is a sequence of decimal digits (`digit`), possibly interspersed with underscores for readability (but not beginning or ending with an underscore). Example: `1_234_567`.

A `hexdigits` token denotes a hexadecimal constant, and is a sequence of hexadecimal digits (`hexdigit`) prefaced by `0x` and possibly interspersed with underscores for readability (but not beginning or ending with an underscore). Example: `0xffff_ffff`.

A `decimaldigits` token is a decimal fraction constant, possibly interspersed with underscores for readability (but not beginning or ending with an underscore). It has digits both before and after a single period (`.`) character. There is no syntax for floating point numbers with exponents. Example: `123_456.789_123`.

2.6.4. Escaped Character

The `escapedChar` token is a multi-character sequence that denotes a non-printable or non-ASCII character. Such tokens begin with a backslash characcter (\) and denote a single- or double-quote character, backslash, null, new line, carriage return, tab, or a Unicode character with given hexadecimal representation. Which Unicode escape form is allowed depends on the value of the `--unicode-char` option.

If `--unicode-char:false` is stipulated, `\uXXXX` escapes can be used to specify any UTF-16 code unit.

If `--unicode-char:true` is stipulated, `\U{X..X}` escapes can be used to specify any Unicode scalar value. There must be at least one hex digit in between the braces, and at most six. Surrogate code points are not allowed. The hex digits may be interspersed with underscores for readability (but not beginning or ending with an underscore), as in `\U{1_F680}`. The braces are part of the required character sequence.

Note that although Unicode letters are not allowed in Dafny identifiers, Dafny does support Unicode in its character, string, and verbatim strings constants and in its comments.

2.6.5. Character Constant Token

The `charToken` token denotes a character constant. It is either a `charChar` or an `escapedChar` enclosed in single quotes.

2.6.6. String Constant Token

A `stringToken` denotes a string constant. It consists of a sequence of `stringChar` and `escapedChar` characters enclosed in double quotes.

A `verbatimStringToken` token also denotes a string constant. It is a sequence of any `verbatimStringChar` characters (which includes newline characters), enclosed between `@"` and `"`, except that two successive double quotes represent one quote character inside the string. This is the mechanism for escaping a double quote character, which is the only character needing escaping in a verbatim string. Within a verbatim string constant, a backslash character represents itself and is not the first character of an `escapedChar`.

2.6.7. Ellipsis

The `ellipsisToken` is the character sequence `...` and is typically used to designate something missing that will later be inserted through refinement or is already present in a parent declaration.

2.7. Low Level Grammar Productions

2.7.1. Identifier Variations

2.7.1.1. Identifier A basic ordinary identifier is just an `ident` token.

It may be followed by a sequence of suffixes to denote compound entities. Each suffix is a dot (`.`) and another token, which may be

- another `ident` token
- a `digits` token
- the `requires` reserved word

- the `reads` reserved word

Note that

- Digits can be used to name fields of classes and destructors of datatypes. For example, the built-in tuple datatypes have destructors named 0, 1, 2, etc. Note that as a field or destructor name, a digit sequence is treated as a string, not a number: internal underscores matter, so `10` is different from `1_0` and from `010`.
- `m.requires` is used to denote the precondition for method `m`.
- `m.reads` is used to denote the things that method `m` may read.

2.7.1.2. No-underscore-identifier

A `NoUSIdent` is an identifier except that identifiers with a **leading** underscore are not allowed. The names of user-defined entities are required to be `NoUSIdent`s or, in some contexts, a `digits`. We introduce more mnemonic names for these below (e.g. `ClassName`).

A no-underscore-identifier is required for the following:

- module name
- class or trait name
- datatype name
- newtype name
- synonym (and subset) type name
- iterator name
- type variable name
- attribute name

A variation, a no-underscore-identifier or a `digits`, is allowed for

- datatype member name
- method or function or constructor name
- label name
- export id
- suffix that is a typename or constructor

All *user-declared* names do not start with underscores, but there are internally generated names that a user program might *use* that begin with an underscore or are just an underscore.

2.7.1.3. Wild identifier

A wild identifier is a no-underscore-identifier except that the singleton `_` is allowed. The `_` is replaced conceptually by a unique identifier distinct from all other identifiers in the program. A `_` is used when an identifier is needed, but its content is discarded. Such identifiers are not used in expressions.

Wild identifiers may be used in these contexts:

- formal parameters of a lambda expression
- the local formal parameter of a quantifier
- the local formal parameter of a subset type or newtype declaration
- a variable declaration
- a case pattern formal parameter

- binding guard parameter
- for loop parameter
- LHS of update statements

2.7.2. Qualified Names

A qualified name starts with the name of a top-level entity and then is followed by zero or more `DotSuffix`s which denote a component. Examples:

- `Module.MyType1`
- `MyTuple.1`
- `MyMethod.requires`
- `A.B.C.D`

The identifiers and dots are separate tokens and so may optionally be separated by whitespace.

2.7.3. Identifier-Type Combinations

Identifiers are typically declared in combination with a type, as in

```
var i: int
```

However, Dafny infers types in many circumstances, and in those, the type can be omitted. The type is required for field declarations and formal parameters of methods, functions and constructors (because there is no initializer). It may be omitted (if the type can be inferred) for local variable declarations, pattern matching variables, quantifiers,

Similarly, there are circumstances in which the identifier name is not needed, because it is not used. This is allowed in defining algebraic datatypes.

In some other situations a wild identifier can be used, as described above.

2.7.4. Quantifier Domains (grammar)

Several Dafny constructs bind one or more variables to a range of possible values. For example, the quantifier `forall x: nat | x <= 5 :: x * x <= 25` has the meaning "for all integers x between 0 and 5 inclusive, the square of x is at most 25". Similarly, the set comprehension `set x: nat | x <= 5 :: f(x)` can be read as "the set containing the result of applying f to x, for each integer x from 0 to 5 inclusive". The common syntax that specifies the bound variables and what values they take on is known as the *quantifier domain*; in the previous examples this is `x: nat | x <= 5`, which binds the variable `x` to the values `0`, `1`, `2`, `3`, `4`, and `5`.

Here are some more examples.

- `x: byte` (where a value of type `byte` is an int-based number `x` in the range `0 <= x < 256`)
- `x: nat | x <= 5`
- `x <- integerSet`
- `x: nat <- integerSet`
- `x: nat <- integerSet | x % 2 == 0`
- `x: nat, y: nat | x < 2 && y < 2`

- `x: nat | x < 2, y: nat | y < x`
- `i | 0 <= i < |s|, y <- s[i] | i < y`

A quantifier domain declares one or more *quantified variables*, separated by commas. Each variable declaration can be nothing more than a variable name, but it may also include any of three optional elements:

1. The optional syntax `: T` declares the type of the quantified variable. If not provided, it will be inferred from context.
2. The optional syntax `<- C` attaches a collection expression `C` as a *quantified variable domain*. Here a collection is any value of a type that supports the `in` operator, namely sets, multisets, maps, and sequences. The domain restricts the bindings to the elements of the collection: `x <- C` implies `x in C`. The example above can also be expressed as `var c := [0, 1, 2, 3, 4, 5]; forall x <- c :: x * x <= 25`.
3. The optional syntax `| E` attaches a boolean expression `E` as a *quantified variable range*, which restricts the bindings to values that satisfy this expression. In the example above `x <= 5` is the range attached to the `x` variable declaration.

Note that a variable's domain expression may reference any variable declared before it, and a variable's range expression may reference the attached variable (and usually does) and any variable declared before it. For example, in the quantifier domain `i | 0 <= i < |s|, y <- s[i] | i < y`, the expression `s[i]` is always well-formed because the range attached to `i` ensures `i` is a valid index in the sequence `s`.

Allowing per-variable ranges is not fully backwards compatible, and so it is not yet allowed by default; the `--quantifier-syntax:4` option needs to be provided to enable this feature (See Section 13.9.5).

2.7.5. Numeric Literals (grammar)

Integer and bitvector literals may be expressed in either decimal or hexadecimal (`digits` or `hexdigits`).

Real number literals are written as decimal fractions (`decimaldigits`).

3. Programs (grammar)

At the top level, a Dafny program (stored as files with extension `.dfy`) is a set of declarations. The declarations introduce (module-level) constants, methods, functions, lemmas, types (classes, traits, inductive and coinductive datatypes, newtypes, type synonyms, abstract types, and iterators) and modules, where the order of introduction is irrelevant. Some types, notably classes, also may contain a set of declarations, introducing fields, methods, and functions.

When asked to compile a program, Dafny looks for the existence of a `Main()` method. If a legal `Main()` method is found, the compiler will emit an executable appropriate to the target language; otherwise it will emit a library or individual files. The conditions for a legal `Main()` method are described in the User Guide (Section 3.4). If there is more than one `Main()`, Dafny will emit an error message.

An invocation of Dafny may specify a number of source files. Each Dafny file follows the grammar of the `Dafny` non-terminal.

A file consists of - a sequence of optional *include* directives, followed by - top level declarations, followed by - the end of the file.

3.1. Include Directives (grammar)

Examples:

```
include "MyProgram.dfy"
include @"/home/me/MyFile.dfy"
```

Include directives have the form `"include" stringToken` where the string token is either a normal string token or a verbatim string token. The `stringToken` is interpreted as the name of a file that will be included in the Dafny source. These included files also obey the `Dafny` grammar. Dafny parses and processes the transitive closure of the original source files and all the included files, but will not invoke the verifier on the included files unless they have been listed explicitly on the command line or the `--verify-included-files` option is specified.

The file name may be a path using the customary `/`, `.`, and `..` punctuation. The interpretation of the name (e.g., case-sensitivity) will depend on the underlying operating system. A path not beginning with `/` is looked up in the underlying file system relative to the *location of the file in which the include directive is stated*. Paths beginning with a device designator (e.g., `C:`) are only permitted on Windows systems. Better style advocates using relative paths in include directives so that groups of files may be moved as a whole to a new location.

Paths of files on the command-line or named in `--library` options are relative the the current working directory.

3.2. Top Level Declarations (grammar)

Examples:

```
abstract module M { }
trait R { }
```

```
class C { }
datatype D = A | B
newtype pos = i: int | i >= 0
type T = i: int | 0 <= i < 100
method m() {}
function f(): int
const c: bool
```

Top-level declarations may appear either at the top level of a Dafny file, or within a (sub)module declaration. A top-level declaration is one of various kinds of declarations described later. Top-level declarations are implicitly members of a default (unnamed) top-level module.

Declarations within a module or at the top-level all begin with reserved keywords and do not end with semicolons.

These declarations are one of these kinds: - methods and functions, encapsulating computations or actions - const declarations, which are names (of a given type) initialized to an unchanging value; declarations of variables and mutable fields are not allowed at the module level - type declarations of various kinds (Section 5 and the following sections)

Methods, functions and const declarations are placed in an implicit class declaration that is in the top-level implicit module. These declarations are all implicitly `static` (and may not be declared explicitly static).

3.3. Declaration Modifiers (grammar)

Examples:

```
abstract module M {
  class C {
    static method m() {}
  }
}
ghost opaque const c : int
```

Top level declarations may be preceded by zero or more declaration modifiers. Not all of these are allowed in all contexts.

The `abstract` modifier may only be used for module declarations. An abstract module can leave some entities underspecified. Abstract modules are not compiled.

The `ghost` modifier is used to mark entities as being used for specification only, not for compilation to code.

The `opaque` modifier may be used on const declarations and functions.

The `static` modifier is used for class members that are associated with the class as a whole rather than with an instance of the class. This modifier may not be used with declarations that are implicitly static, as are members of the top-level, unnamed implicit class.

The following table shows modifiers that are available for each of the kinds of declaration. In the table we use already-ghost (already-non-ghost) to denote that the item is not allowed to have the ghost modifier because it is already implicitly ghost (non-ghost).

Declaration	allowed modifiers
module	abstract
class	-
trait	-
datatype or codatatype	-
field (const)	ghost opaque
newtype	-
synonym types	-
iterators	-
method	ghost static
lemma	already-ghost static
least lemma	already-ghost static
greatest lemma	already-ghost static
constructor	ghost
function	ghost static opaque (Dafny 4)
function method	already-non-ghost static opaque (Dafny 3)
function (non-method)	already-ghost static opaque (Dafny 3)
predicate	ghost static opaque (Dafny 4)
predicate method	already-non-ghost static opaque (Dafny 3)
predicate (non-method)	already-ghost static opaque (Dafny 3)
least predicate	already-ghost static opaque
greatest predicate	already-ghost static opaque

3.4. Executable programs

Dafny programs have an important emphasis on verification, but the programs may also be executable.

To be executable, the program must have exactly one `Main` method and that method must be a legal main entry point.

- The program is searched for a method with the attribute `{:main}`. If exactly one is found, that method is used as the entry point; if more than one method has the `{:main}` attribute, an error message is issued.
- Otherwise, the program is searched for a method with the name `Main`. If more than one is found an error message is issued.

Any abstract modules are not searched for candidate entry points, but otherwise the entry point may be in any module or type. In addition, an entry-point candidate must satisfy the following conditions:

- The method has no type parameters and either has no parameters or one non-ghost parameter of type `seq<string>`.

- The method has no non-ghost out-parameters.
- The method is not a ghost method.
- The method has no requires or modifies clauses, unless it is marked `{:main}`.
- If the method is an instance (that is, non-static) method and the enclosing type is a class, then that class must not declare any constructor. In this case, the runtime system will allocate an object of the enclosing class and will invoke the entry-point method on it.
- If the method is an instance (that is, non-static) method and the enclosing type is not a class, then the enclosing type must, when instantiated with auto-initializing type parameters, be an auto-initializing type. In this case, the runtime system will invoke the entry-point method on a value of the enclosing type.

Note, however, that the following are allowed:

- The method is allowed to have `ensures` clauses
- The method is allowed to have `decreases` clauses, including a `decreases *`. (If `Main()` has a `decreases *`, then its execution may go on forever, but in the absence of a `decreases *` on `Main()`, `dafny` will have verified that the entire execution will eventually terminate.)

If no legal candidate entry point is identified, `dafny` will still produce executable output files, but they will need to be linked with some other code in the target language that provides a `main` entry point.

If the `Main` method takes an argument (of type `seq<string>`), the value of that input argument is the sequence of command-line arguments, with the first entry of the sequence (at index 0) being a system-determined name for the executable being run.

The exit code of the program, when executed, is not yet specified.

4. Modules (grammar)

Examples:

```
module N  { }
import A
export A reveals f
```

Structuring a program by breaking it into parts is an important part of creating large programs. In Dafny, this is accomplished via *modules*. Modules provide a way to group together related types, classes, methods, functions, and other modules, as well as to control the scope of declarations. Modules may import each other for code reuse, and it is possible to abstract over modules to separate an implementation from an interface.

Module declarations are of three types: - a module definition - a module import - a module export definition

Module definitions and imports each declare a submodule of its enclosing module, which may be the implicit, undeclared, top-level module.

4.1. Declaring New Modules (grammar)

Examples:

```
module P { const i: int }
abstract module A.Q { method m() {} }
module M { module N { } }
```

A *module definition* - has an optional modifier (only `abstract` is allowed) - followed by the keyword "module" - followed by a name (a sequence of dot-separated identifiers) - followed by a body enclosed in curly braces

A module body consists of any declarations that are allowed at the top level: classes, datatypes, types, methods, functions, etc.

```
module Mod {
  class C {
    var f: int
    method m()
  }
  datatype Option = A(int) | B(int)
  type T
  method m()
  function f(): int
}
```

You can also put a module inside another, in a nested fashion:

```
module Mod {
  module Helpers {
    class C {
      method doIt()
      var f: int
    }
  }
}
```

Then you can refer to the members of the `Helpers` module within the `Mod` module by prefixing them with “Helpers.”. For example:

```
module Mod {
  module Helpers {
    class C {
      constructor () { f := 0; }
      method doIt()
      var f: int
    }
  }
  method m() {
    var x := new Helpers.C();
    x.doIt();
    x.f := 4;
  }
}
```

Methods and functions defined at the module level are available like classes, with just the module name prefixing them. They are also available in the methods and functions of the classes in the same module.

```
module Mod {
  module Helpers {
    function addOne(n: nat): nat {
      n + 1
    }
  }
  method m() {
    var x := 5;
    x := Helpers.addOne(x); // x is now 6
  }
}
```

Note that everything declared at the top-level (in all the files constituting the program) is implicitly part of a single implicit unnamed global module.

4.2. Declaring nested modules standalone

As described in the previous section, module declarations can be nested. It is also permitted to declare a nested module *outside* of its "containing" module. So instead of

```
module A {
  module B {
  }
}
```

one can write

```
module A {
}
module A.B {
}
```

The second module is completely separate; for example, it can be in a different file. This feature provides flexibility in writing and maintenance; for example, it can reduce the size of module `A` by extracting module `A.B` into a separate body of text.

However, it can also lead to confusion, and program authors need to take care. It may not be apparent to a reader of module `A` that module `A.B` exists; the existence of `A.B` might cause names to be resolved differently and the semantics of the program might be (silently) different if `A.B` is present or absent.

4.3. Importing Modules (grammar)

Examples:

```
import A
import opened B
import A = B
import A : B
import A.B
import A`E
import X = A.B`{E,F}
```

Sometimes you want to refer to things from an existing module, such as a library. In this case, you can *import* one module into another. This is done via the `import` keyword, which has two forms with different meanings. The simplest form is the concrete import, which has the form `import A = B`. This declaration creates a reference to the module `B` (which must already exist), and binds it to the new local name `A`. This form can also be used to create a reference to a nested module, as in `import A = B.C`. The other form, using a `:`, is described in Section 4.6.

As modules in the same scope must have different names, this ability to bind a module to a new name allows disambiguating separately developed external modules that have the same name. Note that the new name is only bound in the scope containing the import declaration; it does not create a global alias. For example, if `Helpers` was defined outside of `Mod`, then we could import it:

```
module Helpers {
  function addOne(n: nat): nat {
    n + 1
  }
}
module Mod {
  import A = Helpers
  method m() {
    assert A.addOne(5) == 6;
  }
}
```

Note that inside `m()`, we have to use `A` instead of `Helpers`, as we bound it to a different name. The name `Helpers` is not available inside `m()` (or anywhere else inside `Mod`), as only names that have been bound inside `Mod` are available. In order to use the members from another module, that other module either has to be declared there with `module` or imported with `import`. (As described below, the resolution of the `ModuleQualifiedName` that follows the `=` in the `import` statement or the `refines` in a module declaration uses slightly different rules.)

We don't have to give `Helpers` a new name, though, if we don't want to. We can write `import Helpers = Helpers` to import the module under its own name; Dafny even provides the shorthand `import Helpers` for this behavior. You can't bind two modules with the same name at the same time, so sometimes you have to use the = version to ensure the names do not clash. When importing nested modules, `import B.C` means `import C = B.C`; the implicit name is always the last name segment of the module designation.

The first identifier in the dot-separated sequence of identifers that constitute the qualified name of the module being imported is resolved as (in order) - a submodule of the importing module, - or a sibling module of the importing module, - or a sibling module of some containing module, traversing outward. There is no way to refer to a containing module, only sibling modules (and their submodules).

Import statements may occur at the top-level of a program (that is, in the implicit top-level module of the program) as well. There they serve as a way to give a new name, perhaps a shorthand name, to a module. For example,

```
module MyModule { } // declare MyModule
import MyModule  // error: cannot add a module named MyModule
                 // because there already is one
import M = MyModule // OK. M and MyModule are equivalent
```

4.4. Opening Modules

Sometimes, prefixing the members of the module you imported with its name is tedious and ugly, even if you select a short name when importing it. In this case, you can import the module as `opened`, which causes all of its members to be available without adding the module name. The `opened` keyword, if present, must immediately follow `import`. For example, we could write the previous example as:

```
module Helpers {
  function addOne(n: nat): nat {
    n + 1
  }
}
module Mod {
  import opened Helpers
  method m() {
    assert addOne(5) == 6;
  }
}
```

When opening modules, the newly bound members have lower priority than local definitions. This means if you define a local function called `addOne`, the function from `Helpers` will no longer be available under that name. When modules are opened, the original name binding is still present however, so you can always use the name that was bound to get to anything that is hidden.

```
module Helpers {
  function addOne(n: nat): nat {
    n + 1
  }
}
module Mod {
  import opened H = Helpers
  function addOne(n: nat): nat {
    n - 1
  }
  method m() {
    assert addOne(5) == 6; // this is now false,
                           // as this is the function just defined
    assert H.addOne(5) == 6; // this is still true
  }
}
```

If you open two modules that both declare members with the same name, then neither member can be referred to without a module prefix, as it would be ambiguous which one was meant. Just opening the two modules is not an error, however, as long as you don't attempt to use members with common names. However, if the ambiguous references actually refer to the same declaration, then they are permitted. The `opened` keyword may be used with any kind of `import` declaration, including the module abstraction form.

An `import opened` may occur at the top-level as well. For example,

```
module MyModule {  } // declares MyModule
import opened MyModule // does not declare a new module, but does
```

```
                             // make all names in MyModule available in
                             // the current scope, without needing
                             // qualification
import opened M = MyModule // names in MyModule are available in
                             // the current scope without qualification
                             // or qualified with either M (because of this
                             // import) or MyModule (because of the original
                             // module definition)
```

The Dafny style guidelines suggest using opened imports sparingly. They are best used when the names being imported have obvious and unambiguous meanings and when using qualified names would be verbose enough to impede understanding.

There is a special case in which the behavior described above is altered. If a module `M` declares a type `M` and `M` is `import opened` without renaming inside another module `X`, then the rules above would have, within `X`, `M` mean the module and `M.M` mean the type. This is verbose. So in this somewhat common case, the type `M` is effectively made a local declaration within `X` so that it has precedence over the module name. Now `M` refers to the type. If one needs to refer to the module, it will have to be renamed as part of the `import opened` statement.

This special-case behavior does give rise to a source of ambiguity. Consider the example

```
module Option {
  const a := 1
  datatype Option = A|B { static const a := 2 }
}

module X {
  import opened Option
  method M() { print Option.a; }
}
```

`Option.a` now means the `a` in the datatype instead of the `a` in the module. To avoid confusion in such cases, it is an ambiguity error if a name that is declared in both the datatype and the module is used when there is an `import open` of the module (without renaming).

4.5. Export Sets and Access Control (grammar)

Examples:

```
export E extends F reveals f,g provides g,h
export E reveals *
export reveals f,g provides g,h
export E
export E ... reveals f
```

In some programming languages, keywords such as `public`, `private`, and `protected` are used to control access to (that is, visibility of) declared program entities. In Dafny, modules and export sets provide that capability. Modules combine declarations into logically related groups. Export sets then permit selectively exposing subsets of a module's declarations; another module can import the export set appropriate to its needs. A user can define as many export sets as are needed to provide different kinds of access to the module's declarations. Each export set designates a list of names, which must be names that are declared in the module (or in a refinement parent).

By default (in the absence of any export set declarations) all the names declared in a module are available outside the module using the `import` mechanism. An *export set* enables a module to disallow the use of some declarations outside the module.

An export set has an optional name used to disambiguate in case of multiple export sets; If specified, such names are used in `import` statements to designate which export set of a module is being imported. If a module `M` has export sets `E1` and `E2`, we can write `import A = M`E1`` to create a module alias `A` that contains only the names in `E1`. Or we can write `import A = M`{E1,E2}`` to import the union of names in `E1` and `E2` as module alias `A`. As before, `import M`E1`` is an abbreviation of `import M = M`E1``.

If no export set is given in an import statement, the default export set of the module is used.

There are various defaults that apply differently in different cases. The following description is with respect to an example module `M`:

M has no export sets declared. Then another module may simply `import Z = M` to obtain access to all of M's declarations.

M has one or more named export sets (e.g., E, F). Then another module can write `import Z = M`E`` or `import Z = M`{E,F}`` to obtain access to the names that are listed in export set `E` or to the union of those in export sets `E` and `F`, respectively. If no export set has the same name as the module, then an export set designator must be used: in that case you cannot write simply `import Z = M`.

M has an unnamed export set, along with other export sets (e.g., named E). The unnamed export set is the default export set and implicitly has the same name as the module. Because there is a default export set, another module may write either `import Z = M` or `import Z = M`M`` to import the names in that default export set. You can also still use the other export sets with the explicit designator: `import Z = M`E``

M declares an export set with the same name as the module. This is equivalent to declaring an export set without a name. `import M` and `import M`M`` perform the same function in either case; the export set with or without the name of the module is the default export set for the module.

Note that names of module aliases (declared by import statements) are just like other names in a module; they can be included or omitted from export sets. Names brought into a module by *refinement* are treated the same as locally declared names and can be listed in export set declarations. However, names brought into a module by `import opened` (either into a module or a refinement parent of a module) may not be further exported. For example,

```
module A {
  const a := 10
  const z := 10
}
module B {
  import opened Z = A // includes a, declares Z
  const b := Z.a // OK
}
module C {
  import opened B // includes b, Z, but not a
  method m() {
    //assert b == a; // error: a is not known
    //assert b == B.a; // error: B.a is not valid
    //assert b == A.a; // error: A is not known
    assert b == Z.a; // OK: module Z is known and includes a
  }
}
```

However, in the above example,

- if `A` has one export set `export Y reveals a` then the import in module `B` is invalid because `A` has no default export set;
- if `A` has one export set `export Y reveals a` and `B` has `import Z = A`Y` then `B`'s import is OK. So is the use of `Z.a` in the assert because `B` declares `Z` and `C` brings in `Z` through the `import opened` and `Z` contains `a` by virtue of its declaration. (The alias `Z` is not able to have export sets; all of its names are visible.)
- if `A` has one export set `export provides z` then `A` does have a default export set, so the import in `B` is OK, but neither the use of `a` in `B` nor as `Z.a` in C would be valid, because `a` is not in `Z`.

The default export set is important in the resolution of qualified names, as described in Section 4.8.

There are a few unusual cases to be noted: - an export set can be completely empty, as in `export Nothing` - an eponymous export set can be completely empty, as in `export`, which by default has the same name as the enclosing module; this is a way to make the module completely private - an export set declaration followed by an extreme predicate declaration looks like this: `export least predicate P() { true }` In this case, the `least` (or `greatest`) is the identifier naming the export set. Consequently, `export least predicate P[nat]() { true }` is illegal because `[nat]` cannot be part of a non-extreme predicate. So, it is not possible to declare an eponymous, empty export set by omitting the export id immediately prior to a declaration of an extreme predicate, because the `least` or `greatest` token is parsed as the export set identifier. The workaround for this situation is to either put the name of the module in explicitly as the export ID (not leaving it to the default) or reorder the declarations. - To avoid confusion, the code

```
module M {
  export
  least predicate P() { true }
}
```

provokes a warning telling the user that the `least` goes with the `export`.

4.5.1. Provided and revealed names

Names can be exported from modules in two ways, designated by `provides` and `reveals` in the export set declaration.

When a name is exported as *provided*, then inside a module that has imported the name only the name is known, not the details of the name's declaration.

For example, in the following code the constant `a` is exported as provided.

```
module A {
  export provides a
  const a := 10
  const b := 20
}

module B {
  import A
  method m() {
    assert A.a == 10; // a is known, but not its value
    // assert A.b == 20; // b is not known through A`A
  }
}
```

Since `a` is imported into module `B` through the default export set `` A`A ``, it can be referenced in the assert statement. The constant `b` is not exported, so it is not available. But the assert about `a` is not provable because the value of `a` is not known in module `B`.

In contrast, if `a` is exported as *revealed*, as shown in the next example, its value is known and the assertion can be proved.

```
module A {
  export reveals a
  const a := 10
  const b := 20
}

module B {
  import A
  method m() {
    assert A.a == 10; // a and its value are known
    // assert A.b == 20; // b is not known through A`A
  }
}
```

The following table shows which parts of a declaration are exported by an export set that `provides` or `reveals` the declaration.

```
 declaration          | what is exported     | what is exported
                      | with provides        | with reveals
----------------------|----------------------|----------------------
 const x: X := E      | const x: X           | const x: X := E
----------------------|----------------------|----------------------
 var x: X             | var x: X             | not allowed
----------------------|----------------------|----------------------
 function F(x: X): Y  | function F(x: X): Y  | function F(x: X): Y
   specification...   |   specification...   |   specification...
 {                    |                      | {
   Body               |                      |   Body
 }                    |                      | }
----------------------|----------------------|----------------------
 method M(x: X)       | method M(x: X)       | not allowed
   returns (y: Y)     |   returns (y: Y)     |
   specification...   |   specification...   |
 {                    |                      |
   Body;              |                      |
 }                    |                      |
----------------------|----------------------|----------------------
 type Opaque          | type Opaque          | type Opaque
 {                    |                      |
   // members...      |                      |
 }                    |                      |
----------------------|----------------------|----------------------
 type Synonym = T     | type Synonym         | type Synonym = T
----------------------|----------------------|----------------------
 type S = x: X        | type S               | type S = x: X
   | P witness E      |                      |   | P witness E
----------------------|----------------------|----------------------
 newtype N = x: X     | type N               | newtype N = x: X
   | P witness E      |                      |   | P witness E
 {                    |                      |
   // members...      |                      |
 }                    |                      |
```

```
----------------------|----------------------|----------------------
 datatype D =         | type D               | datatype D =
     Ctor0(x0: X0)    |                      |    Ctor0(x0: X0)
   | Ctor1(x1: X1)    |                      |  | Ctor1(x1: X1)
   | ...              |                      |  | ...
 {                    |                      |
   // members...      |                      |
 }                   |                     |
---------------------|---------------------|---------------------
 class Cl            | type Cl             | class Cl
   extends T0, ...   |                     |   extends T0, ...
 {                   |                     | {
   constructor ()    |                     |   constructor ()
     spec...         |                     |     spec...
   {                 |                     |
     Body;           |                     |
   }                 |                     |
   // members...     |                     |
 }                   |                     | }
---------------------|---------------------|---------------------
 trait Tr            | type Tr             | trait Tr
   extends T0, ...   |                     |   extends T0, ...
 {                   |                     |
   // members...     |                     |
 }                   |                     |
---------------------|---------------------|---------------------
 iterator Iter(x: X) | type Iter           | iterator Iter(x: X)
   yields (y: Y)     |                     |   yields (y: Y)
   specification...  |                     |   specification...
 {                   |                     |
   Body;             |                     |
 }                   |                     |
---------------------|---------------------|---------------------
 module SubModule    | module SubModule    | not allowed
   ...               |   ...               |
 {                   | {                   |
   export SubModule  |   export SubModule  |
     ...             |     ...             |
   export A ...      |                     |
   // decls...       |   // decls...       |
 }                   | }                   |
---------------------|---------------------|---------------------
 import L = MS       | import L = MS       | not allowed
---------------------|---------------------|---------------------
```

Variations of functions (e.g., `predicate`, `twostate function`) are handled like `function` above, and variations of methods (e.g., `lemma` and `twostate lemma`) are treated like `method` above. Since the whole signature is exported, a function or method is exported to be of the same kind, even through `provides`. For example, an exported `twostate lemma` is exported as a `twostate lemma` (and thus is known by importers to have two implicit heap parameters), and an exported `least predicate P` is exported as a `least predicate P` (and thus importers can use both `P` and its prefix predicate `P#`).

If `C` is a `class`, `trait`, or `iterator`, then `provides C` exports the non-null reference type `C` as an abstract type. This does not reveal that `C` is a reference type, nor does it export the nullable type `C?`.

In most cases, exporting a `class`, `trait`, `datatype`, `codatatype`, or abstract type does not automatically export its members. Instead, any member to be exported must be listed explicitly. For example, consider the type declaration

```
trait Tr {
  function F(x: int): int { 10 }
  function G(x: int): int { 12 }
  function H(x: int): int { 14 }
}
```

An export set that contains only `reveals Tr` has the effect of exporting

```
trait Tr {
}
```

and an export set that contains only `provides Tr, Tr.F reveals Tr.H` has the effect of exporting

```
type Tr {
  function F(x: int): int
  function H(x: int): int { 14 }
}
```

There is no syntax (for example, `Tr.*`) to export all members of a type.

Some members are exported automatically when the type is revealed. Specifically: - Revealing a `datatype` or `codatatype` automatically exports the type's discriminators and destructors. - Revealing an `iterator` automatically exports the iterator's members. - Revealing a class automatically exports the class's anonymous constructor, if any.

For a `class`, a `constructor` member can be exported only if the class is revealed. For a `class` or `trait`, a `var` member can be exported only if the class or trait is revealed (but a `const` member can be exported even if the enclosing class or trait is only provided).

When exporting a sub-module, only the sub-module's eponymous export set is exported. There is no way for a parent module to export any other export set of a sub-module, unless it is done via an `import` declaration of the parent module.

The effect of declaring an import as `opened` is confined to the importing module. That is, the ability of use such imported names as unqualified is not passed on to further imports, as the following example illustrates:

```
module Library {
  const xyz := 16
}

module M {
  export
    provides Lib
    provides xyz // error: 'xyz' is not declared locally

  import opened Lib = Library

  const k0 := Lib.xyz
  const k1 := xyz
}

module Client {
  import opened M

  const a0 := M.Lib.xyz
  const a1 := Lib.xyz
  const a2 := M.xyz // error: M does not have a declaration 'xyz'
  const a3 := xyz // error: unresolved identifier 'xyz'
}
```

As highlighted in this example, module `M` can use `xyz` as if it were a local name (see declaration `k1`), but the unqualified name `xyz` is not made available to importers of `M` (see declarations `a2` and `a3`), nor is it possible for `M` to export the name `xyz`.

A few other notes:

- A `provides` list can mention `*`, which means that all local names (except export set names) in the module are exported as `provides`.
- A `reveals` list can mention `*`, which means that all local names (except export set names) in the module are exported as `reveals`, if the declaration is allowed to appear in a `reveals` clause, or as `provides`, if the declaration is not allowed to appear in a `reveals` clause.
- If no export sets are declared, then the implicit export set is `export reveals *`.
- A refinement module acquires all the export sets from its refinement parent.
- Names acquired by a module from its refinement parent are also subject to export lists. (These are local names just like those declared directly.)

4.5.2. Extends list

An export set declaration may include an *extends* list, which is a list of one or more export set names from the same module containing the declaration (including export set names obtained from a refinement parent). The effect is to include in the declaration the union of all the names in the export sets in the extends list, along with any other names explicitly included in the declaration. So for example in

```
module M {
  const a := 10
  const b := 10
  const c := 10
  export A reveals a
  export B reveals b
  export C extends A, B
    reveals c
}
```

export set `C` will contain the names `a`, `b`, and `c`.

4.6. Module Abstraction

Sometimes, using a specific implementation is unnecessary; instead, all that is needed is a module that implements some interface. In that case, you can use an *abstract* module import. In Dafny, this is written `import A : B`. This means bind the name `A` as before, but instead of getting the exact module `B`, you get any module which *adheres* to `B`. Typically, the module `B` may have abstract type definitions, classes with bodiless methods, or otherwise be unsuitable to use directly. Because of the way refinement is defined, any refinement of `B` can be used safely. For example, suppose we start with these declarations:

```
abstract module Interface {
  function addSome(n: nat): nat
    ensures addSome(n) > n
}
abstract module Mod {
  import A : Interface
  method m() {
    assert 6 <= A.addSome(5);
  }
}
```

We can be more precise if we know that `addSome` actually adds exactly one. The following module has this behavior. Further, the postcondition is stronger, so this is actually a refinement of the Interface module.

```
module Implementation {
  function addSome(n: nat): nat
    ensures addSome(n) == n + 1
  {
    n + 1
  }
}
```

We can then substitute `Implementation` for `A` in a new module, by declaring a refinement of `Mod` which defines `A` to be `Implementation`.

```
abstract module Interface {
  function addSome(n: nat): nat
    ensures addSome(n) > n
}
```

```
abstract module Mod {
  import A : Interface
  method m() {
    assert 6 <= A.addSome(5);
  }
}
module Implementation {
  function addSome(n: nat): nat
    ensures addSome(n) == n + 1
  {
    n + 1
  }
}
module Mod2 refines Mod {
  import A = Implementation
  ...
}
```

When you refine an abstract import into a concrete one Dafny checks that the concrete module is a refinement of the abstract one. This means that the methods must have compatible signatures, all the classes and datatypes with their constructors and fields in the abstract one must be present in the concrete one, the specifications must be compatible, etc.

A module that includes an abstract import must be declared `abstract`.

4.7. Module Ordering and Dependencies

Dafny isn't particular about the textual order in which modules are declared, but they must follow some rules to be well formed. In particular, there must be a way to order the modules in a program such that each only refers to things defined **before** it in the ordering. That doesn't mean the modules have to be given textually in that order in the source text. Dafny will figure out that order for you, assuming you haven't made any circular references. For example, this is pretty clearly meaningless:

```
import A = B
import B = A // error: circular
```

You can have import statements at the toplevel and you can import modules defined at the same level:

```
import A = B
method m() {
  A.whatever();
}
module B { method whatever() {} }
```

In this case, everything is well defined because we can put `B` first, followed by the `A` import, and then finally `m()`. If there is no permitted ordering, then Dafny will give an error, complaining about a cyclic dependency.

Note that when rearranging modules and imports, they have to be kept in the same containing module, which disallows some pathological module structures. Also, the imports and submodules are always considered to be before their containing module, even at the toplevel. This means that the following is not well formed:

```
method doIt() { }
module M {
  method m() {
    doIt(); // error: M precedes doIt
  }
}
```

because the module `M` must come before any other kind of members, such as methods. To define global functions like this, you can put them in a module (called `Globals`, say) and open it into any module that needs its functionality. Finally, if you import via a path, such as `import A = B.C`, then this creates a dependency of `A` on `B`, and `B` itself depends on its own nested module `B.C`.

4.8. Name Resolution

When Dafny sees something like `A<T>.B<U>.C<V>`, how does it know what each part refers to? The process Dafny uses to determine what identifier sequences like this refer to is name resolution. Though the rules may seem complex, usually they do what you would expect. Dafny first looks up the initial identifier. Depending on what the first identifier refers to, the rest of the identifier is looked up in the appropriate context.

In terms of the grammar, sequences like the above are represented as a `NameSegment` followed by 0 or more `Suffix`es. The form shown above contains three instances of `AugmentedDotSuffix_`.

The resolution is different depending on whether it is in a module context, an expression context or a type context.

4.8.1. Modules and name spaces

A module is a collection of declarations, each of which has a name. These names are held in two namespaces.

- The names of export sets
- The names of all other declarations, including submodules and aliased modules

In addition names can be classified as *local* or *imported*.

- Local declarations of a module are the declarations that are explicit in the module and the local declarations of the refinement parent. This includes, for example, the `N` of `import N =` in the refinement parent, recursively.
- Imported names of a module are those brought in by `import opened` plus the imported names in the refinement parent.

Within each namespace, the local names are unique. Thus a module may not reuse a name that a refinement parent has declared (unless it is a refining declaration, which replaces both declarations, as described in Section 10).

Local names take precedence over imported names. If a name is used more than once among imported names (coming from different imports), then it is ambiguous to *use* the name without qualification.

4.8.2. Module Id Context Name Resolution

A qualified name may be used to refer to a module in an import statement or a refines clause of a module declaration. Such a qualified name is resolved as follows, with respect to its syntactic location within a module `Z`:

1. The leading identifier of the qualified name is resolved as a local or imported module name of `Z`, if there is one with a matching name. The target of a `refines` clause does not consider local names, that is, in `module Z refines A.B.C`, any contents of `Z` are not considered in finding `A`.
2. Otherwise, it is resolved as a local or imported module name of the most enclosing module of `Z`, iterating outward to each successive enclosing module until a match is found or the default toplevel module is reached without a match. No consideration of export sets, default or otherwise, is used in this step. However, if at any stage a matching name is found that is not a module declaration, the resolution fails. See the examples below.

3a. Once the leading identifier is resolved as say module `M`, the next identifier in the quallified name is resolved as a local or imported module name within `M`. The resolution is restricted to the default export set of `M`.

3b. If the resolved module name is a module alias (from an `import` statement) then the target of the alias is resolved as a new qualified name with respect to its syntactic context (independent of any resolutions or modules so far). Since `Z` depends on `M`, any such alias target will already have been resolved, because modules are resolved in order of dependency.

4. Step 3 is iterated for each identifier in the qualified module id, resulting in a module that is the final resolution of the complete qualified id.

Ordinarily a module must be *imported* in order for its constituent declarations to be visible inside a given module `M`. However, for the resolution of qualified names this is not the case.

This example shows that the resolution of the refinement parent does not use any local names:

```
module A {
  const a := 10
}

module B refines A { // the top-level A, not the submodule A
  module A { const a := 30 }
  method m() { assert a == 10; } // true
}
```

In the example, the `A` in `refines A` refers to the global `A`, not the submodule `A`.

4.8.3. Expression Context Name Resolution

The leading identifier is resolved using the first following rule that succeeds.

0. Local variables, parameters and bound variables. These are things like `x`, `y`, and `i` in `var x;, ... returns (y: int)`, and `forall i :: ....` The declaration chosen is the match from the innermost matching scope.
1. If in a class, try to match a member of the class. If the member that is found is not static an implicit `this` is inserted. This works for fields, functions, and methods of the current class (if in a static context, then only static methods and functions are allowed). You can refer to fields of the current class either as `this.f` or `f`, assuming of course that `f` is not hidden by one of the above. You can always prefix `this` if needed, which cannot be hidden. (Note that a field whose name is a string of digits must always have some prefix.)
2. If there is no `Suffix`, then look for a datatype constructor, if unambiguous. Any datatypes that don't need qualification (so the datatype name itself doesn't need a prefix) and also have a uniquely named constructor can be referred to just by name. So if `datatype List = Cons(List) | Nil` is the only datatype that declares `Cons` and `Nil` constructors, then you can write `Cons(Cons(Nil))`. If the constructor name is not unique, then you need to prefix it with the name of the datatype (for example `List.Cons(List.Nil)))`. This is done per constructor, not per datatype.
3. Look for a member of the enclosing module.
4. Module-level (static) functions and methods

In each module, names from opened modules are also potential matches, but only after names declared in the module. If an ambiguous name is found or a name of the wrong kind (e.g. a module instead of an expression identifier), an error is generated, rather than continuing down the list.

After the first identifier, the rules are basically the same, except in the new context. For example, if the first identifier is a module, then the next identifier looks into that module. Opened modules only apply within the module it is opened into. When looking up into another module, only things explicitly declared in that module are considered.

To resolve expression `E.id`:

First resolve expression E and any type arguments.

- If `E` resolved to a module `M`:
  0. If `E.id<T>` is not followed by any further suffixes, look for unambiguous datatype constructor.
  1. Member of module M: a sub-module (including submodules of imports), class, datatype, etc.
  2. Static function or method.
- If `E` denotes a type:
  3. Look up id as a member of that type

- If `E` denotes an expression:
    4. Let T be the type of E. Look up id in T.

4.8.4. Type Context Name Resolution

In a type context the priority of identifier resolution is:

1. Type parameters.
2. Member of enclosing module (type name or the name of a module).

To resolve expression `E.id`:

- If `E` resolved to a module `M`:
    0. Member of module M: a sub-module (including submodules of imports), class, datatype, etc.
- If `E` denotes a type:
    1. Then the validity and meaning of `id` depends on the type and must be a user-declared or pre-defined member of the type.

5. Types

A Dafny type is a (possibly-empty) set of values or heap data-structures, together with allowed operations on those values. Types are classified as mutable reference types or immutable value types, depending on whether their values are stored in the heap or are (mathematical) values independent of the heap.

Dafny supports the following kinds of types, all described in later sections of this manual: * builtin scalar types, * builtin collection types, * reference types (classes, traits, iterators), * tuple types (including as a special case a parenthesized type), * inductive and coinductive datatypes, * function (arrow) types, and * types, such as subset types, derived from other types.

5.1. Kinds of types

5.1.1. Value Types

The value types are those whose values do not lie in the program heap. These are:

- The basic scalar types: `bool`, `char`, `int`, `real`, `ORDINAL`, bitvector types
- The built-in collection types: `set`, `iset`, `multiset`, `seq`, `string`, `map`, `imap`
- Tuple Types
- Inductive and coinductive types
- Function (arrow) types
- Subset and newtypes that are based on value types

Data items having value types are passed by value. Since they are not considered to occupy *memory*, framing expressions do not reference them.

The `nat` type is a pre-defined subset type of `int`.

Dafny does not include types themselves as values, nor is there a type of types.

5.1.2. Reference Types

Dafny offers a host of *reference types*. These represent *references* to objects allocated dynamically in the program heap. To access the members of an object, a reference to (that is, a *pointer* to or *object identity* of) the object is *dereferenced*.

The reference types are class types, traits and array types. Dafny supports both reference types that contain the special `null` value (*nullable types*) and reference types that do not (*non-null types*).

5.1.3. Named Types (grammar)

A *Named Type* is used to specify a user-defined type by a (possibly module- or class-qualified) name. Named types are introduced by class, trait, inductive, coinductive, synonym and abstract type declarations. They are also used to refer to type variables. A Named Type is denoted by a dot-separated sequence of name segments (Section 9.32).

A name segment (for a type) is a type name optionally followed by a *generic instantiation*, which supplies type parameters to a generic type, if needed.

The following sections describe each of these kinds of types in more detail.

5.2. Basic types

Dafny offers these basic types: `bool` for booleans, `char` for characters, `int` and `nat` for integers, `real` for reals, `ORDINAL`, and bit-vector types.

5.2.1. Booleans (grammar)

There are two boolean values and each has a corresponding literal in the language: `false` and `true`.

Type `bool` supports the following operations:

operator	precedence	description
`<==>`	1	equivalence (if and only if)
`==>`	2	implication (implies)
`<==`	2	reverse implication (follows from)
`&&`	3	conjunction (and)
`||`	3	disjunction (or)
`==`	4	equality
`!=`	4	disequality
`!`	10	negation (not)

Negation is unary; the others are binary. The table shows the operators in groups of increasing binding power, with equality binding stronger than conjunction and disjunction, and weaker than negation. Within each group, different operators do not associate, so parentheses need to be used. For example,

```
A && B || C    // error
```

would be ambiguous and instead has to be written as either

```
(A && B) || C
```

or

```
A && (B || C)
```

depending on the intended meaning.

5.2.1.1. Equivalence Operator

The expressions `A <==> B` and `A == B` give the same value, but note that `<==>` is *associative* whereas `==` is *chaining* and they have different precedence. So,

```
A <==> B <==> C
```

is the same as

```
A <==> (B <==> C)
```

and

```
(A <==> B) <==> C
```

whereas

```
A == B == C
```

is simply a shorthand for

```
A == B && B == C
```

Also,

```
A <==> B == C <==> D
```

is

```
A <==> (B == C) <==> D
```

5.2.1.2. Conjunction and Disjunction Conjunction and disjunction are associative. These operators are *short circuiting (from left to right)*, meaning that their second argument is evaluated only if the evaluation of the first operand does not determine the value of the expression. Logically speaking, the expression `A && B` is defined when `A` is defined and either `A` evaluates to `false` or `B` is defined. When `A && B` is defined, its meaning is the same as the ordinary, symmetric mathematical conjunction `&`. The same holds for `||` and `|`.

5.2.1.3. Implication and Reverse Implication Implication is *right associative* and is short-circuiting from left to right. Reverse implication `B <== A` is exactly the same as `A ==> B`, but gives the ability to write the operands in the opposite order. Consequently, reverse implication is *left associative* and is short-circuiting from *right to left*. To illustrate the associativity rules, each of the following four lines expresses the same property, for any `A`, `B`, and `C` of type `bool`:

```
A ==> B ==> C
A ==> (B ==> C) // parentheses redundant, ==> is right associative
C <== B <== A
(C <== B) <== A // parentheses redundant, <== is left associative
```

To illustrate the short-circuiting rules, note that the expression `a.Length` is defined for an array `a` only if `a` is not `null` (see Section 5.1.2), which means the following two expressions are well-formed:

```
a != null ==> 0 <= a.Length
0 <= a.Length <== a != null
```

The contrapositives of these two expressions would be:

```
a.Length < 0 ==> a == null  // not well-formed
a == null <== a.Length < 0  // not well-formed
```

but these expressions might not necessarily be well-formed, since well-formedness requires the left (and right, respectively) operand, `a.Length < 0`, to be well-formed in their context.

Implication `A ==> B` is equivalent to the disjunction `!A || B`, but is sometimes (especially in specifications) clearer to read. Since, `||` is short-circuiting from left to right, note that

```
a == null || 0 <= a.Length
```

is well-formed by itself, whereas

```
0 <= a.Length || a == null  // not well-formed
```

is not if the context cannot prove that `a != null`.

In addition, booleans support *logical quantifiers* (forall and exists), described in Section 9.31.4.

5.2.2. Numeric Types (grammar)

Dafny supports *numeric types* of two kinds, *integer-based*, which includes the basic type `int` of all integers, and *real-based*, which includes the basic type `real` of all real numbers. User-defined numeric types based on `int` and `real`, either *subset types* or *newtypes*, are described in Section 5.6.3 and Section 5.7.

There is one built-in *subset type*, `nat`, representing the non-negative subrange of `int`.

The language includes a literal for each integer, like `0`, `13`, and `1985`. Integers can also be written in hexadecimal using the prefix "`0x`", as in `0x0`, `0xD`, and `0x7c1` (always with a lower case `x`, but the hexadecimal digits themselves are case insensitive). Leading zeros are allowed. To form negative literals, use the unary minus operator, as in `-12`, but not `-(12)`.

There are also literals for some of the reals. These are written as a decimal point with a nonempty sequence of decimal digits on both sides, optionally prefixed by a `-` character. For example, `1.0`, `1609.344`, `-12.5`, and `0.5772156649`. Real literals using exponents are not supported in Dafny. For now, you'd have to write your own function for that, e.g.

```
// realExp(2.37, 100) computes 2.37e100
function realExp(r: real, e: int): real decreases if e > 0 then e else -e {
  if e == 0 then r
  else if e < 0 then realExp(r/10.0, e+1)
  else realExp(r*10.0, e-1)
}
```

For integers (in both decimal and hexadecimal form) and reals, any two digits in a literal may be separated by an underscore in order to improve human readability of the literals. For example:

```
const c1 := 1_000_000        // easier to read than 1000000
const c2 := 0_12_345_6789    // strange but legal formatting of 123456789
const c3 := 0x8000_0000      // same as 0x80000000 -- hex digits are
                             // often placed in groups of 4
const c4 := 0.000_000_000_1  // same as 0.0000000001 -- 1 Angstrom
```

In addition to equality and disequality, numeric types support the following relational operations, which have the same precedence as equality:

operator	description
`<`	less than
`<=`	at most
`>=`	at least
`>`	greater than

Like equality and disequality, these operators are chaining, as long as they are chained in the "same direction". That is,

```
A <= B < C == D <= E
```

is simply a shorthand for

```
A <= B && B < C && C == D && D <= E
```

whereas

```
A < B > C
```

is not allowed.

There are also operators on each numeric type:

operator	precedence	description
`+`	6	addition (plus)
`-`	6	subtraction (minus)
`*`	7	multiplication (times)
`/`	7	division (divided by)
`%`	7	modulus (mod) – int only
`-`	10	negation (unary minus)

The binary operators are left associative, and they associate with each other in the two groups. The groups are listed in order of increasing binding power, with equality binding less strongly than any of these operators. There is no implicit conversion between `int` and `real`: use `as int` or `as real` conversions to write an explicit conversion (cf. Section 9.10).

Modulus is supported only for integer-based numeric types. Integer division and modulus are the *Euclidean division and modulus*. This means that modulus always returns a non-negative value, regardless of the signs of the two operands. More precisely, for any integer `a` and non-zero integer `b`,

```
a == a / b * b + a % b
0 <= a % b < B
```

where `B` denotes the absolute value of `b`.

Real-based numeric types have a member `Floor` that returns the *floor* of the real value (as an int value), that is, the largest integer not exceeding the real value. For example, the following properties hold, for any `r` and `r'` of type `real`:

```
method m(r: real, r': real) {
  assert 3.14.Floor == 3;
  assert (-2.5).Floor == -3;
  assert -2.5.Floor == -2; // This is -(2.5.Floor)
  assert r.Floor as real <= r;
  assert r <= r' ==> r.Floor <= r'.Floor;
}
```

Note in the third line that member access (like `.Floor`) binds stronger than unary minus. The fourth line uses the conversion function `as real` from `int` to `real`, as described in Section 9.10.

5.2.3. Bit-vector Types (grammar)

Dafny includes a family of bit-vector types, each type having a specific, constant length, the number of bits in its values. Each such type is distinct and is designated by the prefix `bv` followed (without white space) by a positive integer without leading zeros or zero, stating the number of bits. For example, `bv1`, `bv8`, and `bv32` are legal bit-vector type names. The type `bv0` is also legal; it is a bit-vector type with no bits and just one value, `0x0`.

Constant literals of bit-vector types are given by integer literals converted automatically to the designated type, either by an implicit or explicit conversion operation or by initialization in a declaration. Dafny checks that the constant literal is in the correct range. For example,

```
const i: bv1 := 1
const j: bv8 := 195
const k: bv2 := 5 // error - out of range
const m := (194 as bv8) | (7 as bv8)
```

Bit-vector values can be converted to and from `int` and other bit-vector types, as long as the values are in range for the target type. Bit-vector values are always considered unsigned.

Bit-vector operations include bit-wise operators and arithmetic operators (as well as equality, disequality, and comparisons). The arithmetic operations truncate the high-order bits from the results; that is, they perform unsigned arithmetic modulo 2^{number of bits}, like 2's-complement machine arithmetic.

operator	precedence	description
`<<`	5	bit-limited bit-shift left
`>>`	5	unsigned bit-shift right

operator	precedence	description
+	6	bit-limited addition
-	6	bit-limited subtraction
*	7	bit-limited multiplication
&	8	bit-wise and
|	8	bit-wise or
^	8	bit-wise exclusive-or
-	10	bit-limited negation (unary minus)
!	10	bit-wise complement
.RotateLeft(n)	11	rotates bits left by n bit positions
.RotateRight(n)	11	rotates bits right by n bit positions

The groups of operators lower in the table above bind more tightly.[1] All operators bind more tightly than equality, disequality, and comparisons. All binary operators are left-associative, but the bit-wise `&`, `|`, and `^` do not associate together (parentheses are required to disambiguate). The `+`, `|`, `^`, and `&` operators are commutative.

The right-hand operand of bit-shift operations is an `int` value, must be non-negative, and no more than the number of bits in the type. There is no signed right shift as all bit-vector values correspond to non-negative integers.

Bit-vector negation returns an unsigned value in the correct range for the type. It has the properties `x + (-x) == 0` and `(!x) + 1 == -x`, for a bitvector value `x` of at least one bit.

The argument of the `RotateLeft` and `RotateRight` operations is a non-negative `int` that is no larger than the bit-width of the value being rotated. `RotateLeft` moves bits to higher bit positions (e.g., `(2 as bv4).RotateLeft(1) == (4 as bv4)` and `(8 as bv4).RotateLeft(1) == (1 as bv4)`); `RotateRight` moves bits to lower bit positions, so `b.RotateLeft(n).RotateRight(n) == b`.

Here are examples of the various operations (all the assertions are true except where indicated):

```
const i: bv4 := 9
const j: bv4 := 3

method m() {
  assert (i & j) == (1 as bv4);
  assert (i | j) == (11 as bv4);
  assert (i ^ j) == (10 as bv4);
  assert !i == (6 as bv4);
```

[1]The binding power of shift and bit-wise operations is different than in C-like languages.

```
  assert -i == (7 as bv4);
  assert (i + i) == (2 as bv4);
  assert (j - i) == (10 as bv4);
  assert (i * j) == (11 as bv4);
  assert (i as int) / (j as int) == 3;
  assert (j << 1) == (6 as bv4);
  assert (i << 1) == (2 as bv4);
  assert (i >> 1) == (4 as bv4);
  assert i == 9; // auto conversion of literal to bv4
  assert i * 4 == j + 8 + 9; // arithmetic is modulo 16
  assert i + j >> 1 == (i + j) >> 1; // + - bind tigher than << >>
  assert i + j ^ 2 == i + (j^2);
  assert i * j & 1 == i * (j&1); // & | ^ bind tighter than + - *
}
```

The following are incorrectly formed:

```
const i: bv4 := 9
const j: bv4 := 3

method m() {
  assert i & 4 | j == 0 ; // parentheses required
}
```

```
const k: bv4 := 9

method p() {
  assert k as bv5 == 9 as bv6; // error: mismatched types
}
```

These produce assertion errors:

```
const i: bv4 := 9

method m() {
  assert i as bv3 == 1; // error: i is out of range for bv3
}
```

```
const j: bv4 := 9

method n() {
  assert j == 25; // error: 25 is out of range for bv4
}
```

Bit-vector constants (like all constants) can be initialized using expressions, but pay attention to how type inference applies to such expressions. For example,

```
const a: bv3 := -1
```

is legal because Dafny interprets `-1` as a `bv3` expression, because `a` has type `bv3`. Consequently the `-` is `bv3` negation and the `1` is a `bv3` literal; the value of the expression `-1` is the `bv3` value `7`, which is then the value of `a`.

On the other hand,

```
const b: bv3 := 6 & 11
```

is illegal because, again, the `&` is `bv3` bit-wise-and and the numbers must be valid `bv3` literals. But `11` is not a valid `bv3` literal.

5.2.4. Ordinal type (grammar)

Values of type `ORDINAL` behave like `nat`s in many ways, with one important difference: there are `ORDINAL` values that are larger than any `nat`. The smallest of these non-nat ordinals is represented as ω in mathematics, though there is no literal expression in Dafny that represents this value.

The natural numbers are ordinals. Any ordinal has a successor ordinal (equivalent to adding `1`). Some ordinals are *limit* ordinals, meaning they are not a successor of any other ordinal; the natural number `0` and ω are limit ordinals.

The *offset* of an ordinal is the number of successor operations it takes to reach it from a limit ordinal.

The Dafny type `ORDINAL` has these member functions: - `o.IsLimit` – true if `o` is a limit ordinal (including `0`) - `o.IsSucc` – true if `o` is a successor to something, so `o.IsSucc <==> !o.IsLimit` - `o.IsNat` – true if `o` represents a `nat` value, so for `n` a `nat`, `(n as ORDINAL).IsNat` is true and if `o.IsNat` is true then `(o as nat)` is well-defined - `o.Offset` – is the `nat` value giving the offset of the ordinal

In addition, - non-negative numeric literals may be considered `ORDINAL` literals, so `o + 1` is allowed - `ORDINAL`s may be compared, using `== != < <= > >=` - two `ORDINAL`s may be added and the result is `>=` either one of them; addition is associative but not commutative - `*`, `/` and `%` are not defined for `ORDINAL`s - two `ORDINAL`s may be subtracted if the RHS satisfies `.IsNat` and the offset of the LHS is not smaller than the offset of the RHS

In Dafny, `ORDINAL`s are used primarily in conjunction with extreme functions and lemmas.

5.2.5. Characters (grammar)

Dafny supports a type `char` of *characters*.
Its exact meaning is controlled by the command-line switch `--unicode-char:true|false`.

If `--unicode-char` is disabled, then `char` represents any UTF-16 code unit. This includes surrogate code points.

If `--unicode-char` is enabled, then `char` represents any Unicode scalar value. This excludes surrogate code points.

Character literals are enclosed in single quotes, as in `'D'`. To write a single quote as a character literal, it is necessary to use an *escape sequence*. Escape sequences can also be used to write other characters. The supported escape sequences are the following:

escape sequence	meaning
`\'`	the character `'`
`\"`	the character `"`
`\\`	the character `\`
`\0`	the null character, same as `\u0000` or `\U{0}`
`\n`	line feed
`\r`	carriage return
`\t`	horizontal tab
`\u`*xxxx*	UTF-16 code unit whose hexadecimal code is *xxxx*, where each *x* is a hexadecimal digit
`\U{`*x..x*`}`	Unicode scalar value whose hexadecimal code is *x..x*, where each *x* is a hexadecimal digit

The escape sequence for a double quote is redundant, because `'"'` and `'\"'` denote the same character—both forms are provided in order to support the same escape sequences in string literals (Section 5.5.3.5).

In the form `\u`*xxxx*, which is only allowed if `--unicode-char` is disabled, the `u` is always lower case, but the four hexadecimal digits are case insensitive.

In the form `\U{`*x..x*`}`, which is only allowed if `--unicode-char` is enabled, the `U` is always upper case, but the hexadecimal digits are case insensitive, and there must be at least one and at most six digits. Surrogate code points are not allowed. The hex digits may be interspersed with underscores for readability (but not beginning or ending with an underscore), as in `\U{1_F680}`.

Character values are ordered and can be compared using the standard relational operators:

operator	description
`<`	less than
`<=`	at most
`>=`	at least
`>`	greater than

Sequences of characters represent *strings*, as described in Section 5.5.3.5.

Character values can be converted to and from `int` values using the `as int` and `as char` conversion operations. The result is what would be expected in other programming languages, namely, the `int` value of a `char` is the ASCII or Unicode numeric value.

The only other operations on characters are obtaining a character by indexing into a string, and the implicit conversion to string when used as a parameter of a `print` statement.

5.3. Type parameters (grammar)

Examples:

```
type G1<T>
type G2<T(0)>
type G3<+T(==),-U>
```

Many of the types, functions, and methods in Dafny can be parameterized by types. These *type parameters* are declared inside angle brackets and can stand for any type.

Dafny has some inference support that makes certain signatures less cluttered (described in Section 12.2).

5.3.1. Declaring restrictions on type parameters

It is sometimes necessary to restrict type parameters so that they can only be instantiated by certain families of types, that is, by types that have certain properties. These properties are known as *type characteristics*. The following subsections describe the type characteristics that Dafny supports.

In some cases, type inference will infer that a type-parameter must be restricted in a particular way, in which case Dafny will add the appropriate suffix, such as `(==)`, automatically.

If more than one restriction is needed, they are either listed comma-separated, inside the parentheses or as multiple parenthesized elements: `T(==,0)` or `T(==)(0)`.

When an actual type is substituted for a type parameter in a generic type instantiation, the actual type must have the declared or inferred type characteristics of the type parameter. These characteristics might also be inferred for the actual type. For example, a numeric-based subset or newtype automatically has the `==` relationship of its base type. Similarly, type synonyms have the characteristics of the type they represent.

An abstract type has no known characteristics. If it is intended to be defined only as types that have certain characteristics, then those characteristics must be declared. For example,

```
class A<T(00)> {}
type Q
const a: A<Q>
```

will give an error because it is not known whether the type `Q` is non-empty (`00`). Instead, one needs to write

```
class A<T(00)> {}
type Q(00)
const a: A?<Q> := null
```

5.3.1.1. Equality-supporting type parameters: `T(==)` Designating a type parameter with the `(==)` suffix indicates that the parameter may only be replaced in non-ghost contexts with types that are known to support run-time equality comparisons (`==` and `!=`). All types support equality in ghost contexts, as if, for some types, the equality function is ghost.

For example,

```
method Compare<T(==)>(a: T, b: T) returns (eq: bool)
{
  if a == b { eq := true; } else { eq := false; }
}
```

is a method whose type parameter is restricted to equality-supporting types when used in a non-ghost context. Again, note that *all* types support equality in *ghost* contexts; the difference is only for non-ghost (that is, compiled) code. Coinductive datatypes, arrow types, and inductive datatypes with ghost parameters are examples of types that are not equality supporting.

5.3.1.2. Auto-initializable types: `T(0)` At every access of a variable `x` of a type `T`, Dafny ensures that `x` holds a legal value of type `T`. If no explicit initialization is given, then an arbitrary value is assumed by the verifier and supplied by the compiler, that is, the variable is *auto-initialized*, but to an arbitrary value. For example,

```
class Example<A(0), X> {
  var n: nat
  var i: int
  var a: A
  var x: X

  constructor () {
    new; // error: field 'x' has not been given a value`
    assert n >= 0; // true, regardless of the value of 'n'
    assert i >= 0; // possibly false, since an arbitrary 'int' may be negative
    // 'a' does not require an explicit initialization, since 'A' is auto-init
  }
}
```

In the example above, the class fields do not need to be explicitly initialized in the constructor because they are auto-initialized to an arbitrary value.

Local variables and out-parameters are however, subject to definite assignment rules. The following example requires `--relax-definite-assignment`, which is not the default.

```
method m() {
  var n: nat; // Auto-initialized to an arbitrary value of type `nat`
  assert n >= 0; // true, regardless of the value of n
  var i: int;
  assert i >= 0; // possibly false, arbitrary ints may be negative
}
```

With the default behavior of definite assignment, `n` and `i` need to be initialized to an explicit value of their type or to an arbitrary value using, for example, `var n: nat := *;`.

For some types (known as *auto-init types*), the compiler can choose an initial value, but for others it does not. Variables and fields whose type the compiler does not auto-initialize are subject to *definite-assignment* rules. These ensure that the program explicitly assigns a value to a variable before it is used. For more details see Section 12.6 and the `--relax-definite-assignment` command-line option. More detail on auto-initializing is in this document.

Dafny supports auto-init as a type characteristic. To restrict a type parameter to auto-init types, mark it with the `(0)` suffix. For example,

```
method AutoInitExamples<A(0), X>() returns (a: A, x: X)
{
  // 'a' does not require an explicit initialization, since A is auto-init
  // error: out-parameter 'x' has not been given a value
}
```

In this example, an error is reported because out-parameter `x` has not been assigned—since nothing is known about type `X`, variables of type `X` are subject to definite-assignment rules. In contrast, since type parameter `A` is declared to be restricted to auto-init types, the program does not need to explicitly assign any value to the out-parameter `a`.

5.3.1.3. Nonempty types: `T(00)`

Auto-init types are important in compiled contexts. In ghost contexts, it may still be important to know that a type is nonempty. Dafny supports a type characteristic for nonempty types, written with the suffix `(00)`. For example, with `--relax-definite-assignment`, the following example happens:

```
method NonemptyExamples<B(00), X>() returns (b: B, ghost g: B, ghost h: X)
{
  // error: non-ghost out-parameter 'b' has not been given a value
  // ghost out-parameter 'g' is fine, since its type is nonempty
  // error: 'h' has not been given a value
}
```

Because of `B`'s nonempty type characteristic, ghost parameter `g` does not need to be explicitly assigned. However, Dafny reports an error for the non-ghost `b`, since `B` is not an auto-init type, and reports an error for `h`, since the type `X` could be empty.

Note that every auto-init type is nonempty.

In the default definite-assignment mode (that is, without `--relax-definite-assignment`) there will be errors for all three formal parameters in the example just given.

For more details see Section 12.6.

5.3.1.4. Non-heap based: `T(!new)`

Dafny makes a distinction between types whose values are on the heap, i.e. references, like classes and arrays, and those that are strictly value-based, like basic types and datatypes. The practical implication is that references depend on allocation state (e.g., are affected by the `old` operation) whereas non-reference values are not. Thus it can be relevant to know whether the values of a type parameter are heap-based or not. This is indicated by the mode suffix `(!new)`.

A type parameter characterized by `(!new)` is *recursively* independent of the allocation state. For example, a datatype is not a reference, but for a parameterized data type such as

```
datatype Result<T> = Failure(error: string) | Success(value: T)
```

the instantiation `Result<int>` satisfies `(!new)`, whereas `Result<array<int>>` does not.

Note that this characteristic of a type parameter is operative for both verification and compilation. Also, abstract types at the topmost scope are always implicitly `(!new)`.

Here are some examples:

```
datatype Result<T> = Failure(error: string) | Success(v: T)
datatype ResultN<T(!new)> = Failure(error: string) | Success(v: T)

class C {}

method m() {
  var x1: Result<int>;
  var x2: ResultN<int>;
  var x3: Result<C>;
  var x4: ResultN<C>; // error
  var x5: Result<array<int>>;
  var x6: ResultN<array<int>>; // error
}
```

5.3.2. Type parameter variance

Type parameters have several different variance and cardinality properties. These properties of type parameters are designated in a generic type definition. For instance, in `type A<+T> = ...`, the `+` indicates that the `T` position is co-variant. These properties are indicated by the following notation:

notation	variance	cardinality-preserving
(nothing)	non-variant	yes
`+`	co-variant	yes
`-`	contra-variant	not necessarily
`*`	co-variant	not necessarily
`!`	non-variant	not necessarily

- *co-variance* (`A<+T>` or `A<*T>`) means that if `U` is a subtype of `V` then `A<U>` is a subtype of `A<V>`
- *contra-variance* (`A<-T>`) means that if `U` is a subtype of `V` then `A<V>` is a subtype of `A<U>`
- *non-variance* (`A<T>` or `A<!T>`) means that if `U` is a different type than `V` then there is no subtyping relationship between `A<U>` and `A<V>`

Cardinality preserving means that the cardinality of the type being defined never exceeds the cardinality of any of its type parameters. For example `type T<X> = X -> bool` is illegal and returns the error message `formal type parameter 'X' is not used according to its variance specification (it is used left of an arrow) (perhaps try declaring 'X'` `as '-X' or '!X')` The type `X -> bool` has strictly more values than the type `X`. This affects certain uses of the type, so Dafny requires the declaration of `T` to explicitly say so. Marking the type parameter `X` with `-` or `!` announces that the cardinality of `T<X>` may by larger than that of `X`. If you use `-`, you're also declaring `T` to be contravariant in its type argument, and if you use `!`, you're declaring that `T` is non-variant in its type argument.

To fix it, we use the variance `!`:

```
type T<!X> = X -> bool
```

This states that `T` does not preserve the cardinality of `X`, meaning there could be strictly more values of type `T<E>` than values of type `E` for any `E`.

A more detailed explanation of these topics is here.

5.4. Generic Instantiation (grammar)

A generic instantiation consists of a comma-separated list of 1 or more Types, enclosed in angle brackets (`< >`), providing actual types to be used in place of the type parameters of the declaration of the generic type. If there is no instantion for a generic type, type inference will try to fill these in (cf. Section 12.2).

5.5. Collection types

Dafny offers several built-in collection types.

5.5.1. Sets (grammar)

For any type `T`, each value of type `set<T>` is a finite set of `T` values.

Set membership is determined by equality in the type `T`, so `set<T>` can be used in a non-ghost context only if `T` is equality supporting.

For any type `T`, each value of type `iset<T>` is a potentially infinite set of `T` values.

A set can be formed using a *set display* expression, which is a possibly empty, unordered, duplicate-insensitive list of expressions enclosed in curly braces. To illustrate,

```
{}          {2, 7, 5, 3}          {4+2, 1+5, a*b}
```

are three examples of set displays. There is also a *set comprehension* expression (with a binder, like in logical quantifications), described in Section 9.31.5.

In addition to equality and disequality, set types support the following relational operations:

operator	precedence
`<`	4
`<=`	4
`>=`	4
`>`	4

Like the arithmetic relational operators, these operators are chaining.

Sets support the following binary operators, listed in order of increasing binding power:

operator	precedence	description
`!!`	4	disjointness
`+`	6	set union
`-`	6	set difference
`*`	7	set intersection

The associativity rules of `+`, `-`, and `*` are like those of the arithmetic operators with the same names. The expression `A !! B`, whose binding power is the same as equality (but which neither associates nor chains with equality), says that sets `A` and `B` have no elements in common, that is, it is equivalent to

```
A * B == {}
```

However, the disjointness operator is chaining though in a slightly different way than other chaining operators: `A !! B !! C !! D` means that `A`, `B`, `C` and `D` are all mutually disjoint, that is

```
A * B == {} && (A + B) * C == {} && (A + B + C) * D == {}
```

In addition, for any set `s` of type `set<T>` or `iset<T>` and any expression `e` of type `T`, sets support the following operations:

expression	precedence	result type	description
`e in s`	4	`bool`	set membership
`e !in s`	4	`bool`	set non-membership
`|s|`	11	`nat`	set cardinality (not for `iset`)

The expression `e !in s` is a syntactic shorthand for `!(e in s)`.

(No white space is permitted between `!` and `in`, making `!in` effectively the one example of a mixed-character-class token in Dafny.)

5.5.2. Multisets (grammar)

A *multiset* is similar to a set, but keeps track of the multiplicity of each element, not just its presence or absence. For any type `T`, each value of type `multiset<T>` is a map from `T` values to natural numbers denoting each element's multiplicity. Multisets in Dafny are finite, that is, they contain a finite number of each of a finite set of elements. Stated differently, a multiset maps only a finite number of elements to non-zero (finite) multiplicities.

Like sets, multiset membership is determined by equality in the type `T`, so `multiset<T>` can be used in a non-ghost context only if `T` is equality supporting.

A multiset can be formed using a *multiset display* expression, which is a possibly empty, unordered list of expressions enclosed in curly braces after the keyword `multiset`. To illustrate,

```
multiset{}   multiset{0, 1, 1, 2, 3, 5}   multiset{4+2, 1+5, a*b}
```

are three examples of multiset displays. There is no multiset comprehension expression.

In addition to equality and disequality, multiset types support the following relational operations:

operator	precedence
`<`	4
`<=`	4
`>=`	4
`>`	4

Like the arithmetic relational operators, these operators are chaining.

Multisets support the following binary operators, listed in order of increasing binding power:

operator	precedence	description
`!!`	4	multiset disjointness
`+`	6	multiset sum
`-`	6	multiset difference
`*`	7	multiset intersection

The associativity rules of `+`, `-`, and `*` are like those of the arithmetic operators with the same names. The `+` operator adds the multiplicity of corresponding elements, the `-` operator subtracts them (but 0 is the minimum multiplicity), and the `*` has multiplicity that is the minimum of the multiplicity of the operands. There is no operator for multiset union, which would compute the maximum of the multiplicities of the operands.

The expression `A !! B` says that multisets `A` and `B` have no elements in common, that is, it is equivalent to

```
A * B == multiset{}
```

Like the analogous set operator, `!!` is chaining and means mutual disjointness.

In addition, for any multiset `s` of type `multiset<T>`, expression `e` of type `T`, and non-negative integer-based numeric `n`, multisets support the following operations:

expression	precedence	result type	description
`e in s`	4	`bool`	multiset membership
`e !in s`	4	`bool`	multiset non-membership
`|s|`	11	`nat`	multiset cardinality
`s[e]`	11	`nat`	multiplicity of `e` in `s`
`s[e := n]`	11	`multiset<T>`	multiset update (change of multiplicity)

The expression `e in s` returns `true` if and only if `s[e] != 0`. The expression `e !in s` is a syntactic shorthand for `!(e in s)`. The expression `s[e := n]` denotes a multiset like `s`, but where the multiplicity of element `e` is `n`. Note that the multiset update `s[e := 0]` results in a multiset like `s` but without any occurrences of `e` (whether or not `s` has occurrences of `e` in the first place). As another example, note that `s - multiset{e}` is equivalent to:

```
if e in s then s[e := s[e] - 1] else s
```

5.5.3. Sequences (grammar)

For any type `T`, a value of type `seq<T>` denotes a *sequence* of `T` elements, that is, a mapping from a finite downward-closed set of natural numbers (called *indices*) to `T` values.

5.5.3.1. Sequence Displays A sequence can be formed using a *sequence display* expression, which is a possibly empty, ordered list of expressions enclosed in square brackets. To illustrate,

```
[]          [3, 1, 4, 1, 5, 9, 3]          [4+2, 1+5, a*b]
```

are three examples of sequence displays.

There is also a sequence comprehension expression (Section 9.28):

```
seq(5, i => i*i)
```

is equivalent to `[0, 1, 4, 9, 16]`.

5.5.3.2. Sequence Relational Operators In addition to equality and disequality, sequence types support the following relational operations:

operator	precedence
<	4
<=	4

Like the arithmetic relational operators, these operators are chaining. Note the absence of `>` and `>=`.

5.5.3.3. Sequence Concatenation Sequences support the following binary operator:

operator	precedence
`+`	6

Operator `+` is associative, like the arithmetic operator with the same name.

5.5.3.4. Other Sequence Expressions In addition, for any sequence `s` of type `seq<T>`, expression `e` of type `T`, integer-based numeric index `i` satisfying `0 <= i < |s|`, and integer-based numeric bounds `lo` and `hi` satisfying `0 <= lo <= hi <= |s|`, noting that bounds can equal the length of the sequence, sequences support the following operations:

expression	precedence	result type	description
`e in s`	4	`bool`	sequence membership
`e !in s`	4	`bool`	sequence non-membership
`|s|`	11	`nat`	sequence length
`s[i]`	11	`T`	sequence selection
`s[i := e]`	11	`seq<T>`	sequence update
`s[lo..hi]`	11	`seq<T>`	subsequence

expression	precedence	result type	description
`s[lo..]`	11	`seq<T>`	drop
`s[..hi]`	11	`seq<T>`	take
`s[`*slices*`]`	11	`seq<seq<T>>`	slice
`multiset(s)`	11	`multiset<T>`	sequence conversion to a `multiset<T>`

Expression `s[i := e]` returns a sequence like `s`, except that the element at index `i` is `e`. The expression `e in s` says there exists an index `i` such that `s[i] == e`. It is allowed in non-ghost contexts only if the element type `T` is equality supporting. The expression `e !in s` is a syntactic shorthand for `!(e in s)`.

Expression `s[lo..hi]` yields a sequence formed by taking the first `hi` elements and then dropping the first `lo` elements. The resulting sequence thus has length `hi - lo`. Note that `s[0..|s|]` equals `s`. If the upper bound is omitted, it defaults to `|s|`, so `s[lo..]` yields the sequence formed by dropping the first `lo` elements of `s`. If the lower bound is omitted, it defaults to `0`, so `s[..hi]` yields the sequence formed by taking the first `hi` elements of `s`.

In the sequence slice operation, *slices* is a nonempty list of length designators separated and optionally terminated by a colon, and there is at least one colon. Each length designator is a non-negative integer-based numeric; the sum of the length designators is no greater than `|s|`. If there are *k* colons, the operation produces *k + 1* consecutive subsequences from `s`, with the length of each indicated by the corresponding length designator, and returns these as a sequence of sequences. If *slices* is terminated by a colon, then the length of the last slice extends until the end of `s`, that is, its length is `|s|` minus the sum of the given length designators. For example, the following equalities hold, for any sequence `s` of length at least `10`:

```
method m(s: seq<int>) {
  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];
  var u := [true, false, false, true][1:1:];
  assert |u| == 3 && u[0][0] && !u[1][0] && u[2] == [false, true];
  assume |s| > 10;
  assert s[10:][0] == s[..10];
  assert s[10:][1] == s[10..];
}
```

The operation `multiset(s)` yields the multiset of elements of sequence `s`. It is allowed in non-ghost contexts only if the element type `T` is equality supporting.

5.5.3.5. Strings (grammar) A special case of a sequence type is `seq<char>`, for which Dafny provides a synonym: `string`. Strings are like other sequences, but provide additional syntax for sequence display expressions, namely *string literals*. There are two forms of the syntax for string literals: the *standard form* and the *verbatim form*.

String literals of the standard form are enclosed in double quotes, as in `"Dafny"`. To include a double quote in such a string literal, it is necessary to use an escape sequence. Escape sequences can also be used to include other characters. The supported escape sequences are the same as those for character literals (Section 5.2.5). For example, the Dafny expression `"say \"yes\""` represents the string `'say "yes"'`. The escape sequence for a single quote is redundant, because `"\'"` and `"\'"` denote the same string—both forms are provided in order to support the same escape sequences as do character literals.

String literals of the verbatim form are bracketed by `@"` and `"`, as in `@"Dafny"`. To include a double quote in such a string literal, it is necessary to use the escape sequence `""`, that is, to write the character twice. In the verbatim form, there are no other escape sequences. Even characters like newline can be written inside the string literal (hence spanning more than one line in the program text).

For example, the following three expressions denote the same string:

```
"C:\\tmp.txt"
@"C:\tmp.txt"
['C', ':', '\\', 't', 'm', 'p', '.', 't', 'x', 't']
```

Since strings are sequences, the relational operators `<` and `<=` are defined on them. Note, however, that these operators still denote proper prefix and prefix, respectively, not some kind of alphabetic comparison as might be desirable, for example, when sorting strings.

5.5.4. Finite and Infinite Maps (grammar)

For any types `T` and `U`, a value of type `map<T,U>` denotes a *(finite) map* from `T` to `U`. In other words, it is a look-up table indexed by `T`. The *domain* of the map is a finite set of `T` values that have associated `U` values. Since the keys in the domain are compared using equality in the type `T`, type `map<T,U>` can be used in a non-ghost context only if `T` is equality supporting.

Similarly, for any types `T` and `U`, a value of type `imap<T,U>` denotes a *(possibly) infinite map*. In most regards, `imap<T,U>` is like `map<T,U>`, but a map of type `imap<T,U>` is allowed to have an infinite domain.

A map can be formed using a *map display* expression (see Section 9.30), which is a possibly empty, ordered list of *maplets*, each maplet having the form `t := u` where `t` is an expression of type `T` and `u` is an expression of type `U`, enclosed in square brackets after the keyword `map`. To illustrate,

```
map[]
map[20 := true, 3 := false, 20 := false]
map[a+b := c+d]
```

are three examples of map displays. By using the keyword `imap` instead of `map`, the map produced will be of type `imap<T,U>` instead of `map<T,U>`. Note that an infinite map (`imap`) is allowed to have a finite domain, whereas a finite map (`map`) is not allowed to have an infinite domain. If the same key occurs more than once in a map display expression, only the last occurrence appears in the resulting map.[2] There is also a *map comprehension expression*, explained in Section 9.31.8.

For any map `fm` of type `map<T,U>`, any map `m` of type `map<T,U>` or `imap<T,U>`, any expression `t` of type `T`, any expression `u` of type `U`, and any `d` in the domain of `m` (that is, satisfying `d in m`), maps support the following operations:

expression	precedence	result type	description
`t in m`	4	`bool`	map domain membership
`t !in m`	4	`bool`	map domain non-membership
`|fm|`	11	`nat`	map cardinality
`m[d]`	11	`U`	map selection
`m[t := u]`	11	`map<T,U>`	map update
`m.Keys`	11	(i)`set<T>`	the domain of `m`
`m.Values`	11	(i)`set<U>`	the range of `m`
`m.Items`	11	(i)`set<(T,U)>`	set of pairs (t,u) in `m`

`|fm|` denotes the number of mappings in `fm`, that is, the cardinality of the domain of `fm`. Note that the cardinality operator is not supported for infinite maps. Expression `m[d]` returns the `U` value that `m` associates with `d`. Expression `m[t := u]` is a map like `m`, except that the element at key `t` is `u`. The expression `t in m` says `t` is in the domain of `m` and `t !in m` is a syntactic shorthand for `!(t in m)`.[3]

The expressions `m.Keys`, `m.Values`, and `m.Items` return, as sets, the domain, the range, and the 2-tuples holding the key-value associations in the map. Note that `m.Values` will have a different cardinality than `m.Keys` and `m.Items` if different keys are associated with the same value. If `m` is an `imap`, then these expressions return `iset` values. If `m` is a map, `m.Values` and `m.Items` require the type of the range `U` to support equality.

Here is a small example, where a map `cache` of type `map<int,real>` is used to cache computed values of Joule-Thomson coefficients for some fixed gas at a given temperature:

```
if K in cache {  // check if temperature is in domain of cache
  coeff := cache[K];  // read result in cache
} else {
  coeff := ComputeJTCoefficient(K); // do expensive computation
  cache := cache[K := coeff];  // update the cache
}
```

Dafny also overloads the `+` and `-` binary operators for maps. The `+` operator merges two maps or imaps of the same type, as if each (key,value) pair of the RHS is added in turn to the LHS (i)map. In this use, `+` is not commutative; if a key exists in both (i)maps, it is the value from the RHS (i)map that is present in the result.

[2]This is likely to change in the future to disallow multiple occurrences of the same key.

[3]This is likely to change in the future as follows: The `in` and `!in` operations will no longer be supported on maps, with `x in m` replaced by `x in m.Keys`, and similarly for `!in`.

The - operator implements a map difference operator. Here the LHS is a `map<K,V>` or `imap<K,V>` and the RHS is a `set<K>` (but not an `iset`); the operation removes from the LHS all the (key,value) pairs whose key is a member of the RHS set.

To avoid causing circular reasoning chains or providing too much information that might complicate Dafny's prover finding proofs, not all properties of maps are known by the prover by default. For example, the following does not prove:

```
method mmm<K(==),V(==)>(m: map<K,V>, k: K, v: V) {
    var mm := m[k := v];
    assert v in mm.Values;
  }
```

Rather, one must provide an intermediate step, which is not entirely obvious:

```
method mmm<K(==),V(==)>(m: map<K,V>, k: K, v: V) {
    var mm := m[k := v];
    assert k in mm.Keys;
    assert v in mm.Values;
  }
```

5.5.5. Iterating over collections

Collections are very commonly used in programming and one frequently needs to iterate over the elements of a collection. Dafny does not have built-in iterator methods, but the idioms by which to do so are straightforward. The subsections below give some introductory examples; more detail can be found in this power user note.

5.5.5.1. Sequences and arrays Sequences and arrays are indexable and have a length. So the idiom to iterate over the contents is well-known. For an array:

```
method m(a: array<int>) {
  var i := 0;
  var sum := 0;
  while i < a.Length {
    sum := sum + a[i];
    i := i + 1;
  }
}
```

For a sequence, the only difference is the length operator:

```
method m(s: seq<int>) {
  var i := 0;
  var sum := 0;
  while i < |s| {
    sum := sum + s[i];
    i := i + 1;
  }
}
```

The `forall` statement (Section 8.21) can also be used with arrays where parallel assignment is needed:

```
method m(s: array<int>) {
  var rev := new int[s.Length];
  forall i | 0 <= i < s.Length {
    rev[i] := s[s.Length-i-1];
  }
}
```

See Section 5.10.2 on how to convert an array to a sequence.

5.5.5.2. Sets

There is no intrinsic order to the elements of a set. Nevertheless, we can extract an arbitrary element of a nonempty set, performing an iteration as follows:

```
method m(s: set<int>) {
  var ss := s;
  while ss != {}
    decreases |ss|
  {
    var i: int :| i in ss;
    ss := ss - {i};
    print i, "\n";
  }
}
```

Because `iset`s may be infinite, Dafny does not permit iteration over an `iset`.

5.5.5.3. Maps

Iterating over the contents of a `map` uses the component sets: `Keys`, `Values`, and `Items`. The iteration loop follows the same patterns as for sets:

```
method m<T(==),U(==)> (m: map<T,U>) {
  var items := m.Items;
  while items != {}
    decreases |items|
  {
    var item :| item in items;
    items := items - { item };
    print item.0, " ", item.1, "\n";
  }
}
```

There are no mechanisms currently defined in Dafny for iterating over `imap`s.

5.6. Types that stand for other types (grammar)

It is sometimes useful to know a type by several names or to treat a type abstractly. There are several mechanisms in Dafny to do this:

- (Section 5.6.1) A typical *synonym type*, in which a type name is a synonym for another type
- (Section 5.6.2) An *abstract type*, in which a new type name is declared as an uninterpreted type
- (Section 5.6.3) A *subset type*, in which a new type name is given to a subset of the values of a given type
- ([Section 0.0]{#sec-newtypes)) A *newtype*, in which a subset type is declared, but with restrictions on converting to and from its base type

5.6.1. Type synonyms (grammar)

```
type T = int
type SS<T> = set<set<T>>
```

A *type synonym* declaration:

```
type Y<T> = G
```

declares `Y<T>` to be a synonym for the type `G`. If the `= G` is omitted then the declaration just declares a name as an uninterpreted *abstract* type, as described in Section 5.6.2. Such types may be given a definition elsewhere in the Dafny program.

Here, `T` is a nonempty list of type parameters (each of which optionally has a type characteristics suffix), which can be used as free type variables in `G`. If the synonym has no type parameters, the "`<T>`" is dropped. In all cases, a type synonym is just a synonym. That is, there is never a difference, other than possibly in error messages produced, between `Y<T>` and `G`.

For example, the names of the following type synonyms may improve the readability of a program:

```
type Replacements<T> = map<T,T>
type Vertex = int
```

The new type name itself may have type characteristics declared, and may need to if there is no definition. If there is a definition, the type characteristics are typically inferred from the definition. The syntax is like this:

```
type Z(==)<T(0)>
```

As already described in Section 5.5.3.5, `string` is a built-in type synonym for `seq<char>`, as if it would have been declared as follows:

```
type string_(==,0,!new) = seq<char>
```

If the implicit declaration did not include the type characteristics, they would be inferred in any case.

Note that although a type synonym can be declared and used in place of a type name, that does not affect the names of datatype or class constructors. For example, consider

```
datatype Pair<T> = Pair(first: T, second: T)
type IntPair = Pair<int>

const p: IntPair := Pair(1,2) // OK
const q: IntPair := IntPair(3,4) // Error
```

In the declaration of `q`, `IntPair` is the name of a type, not the name of a function or datatype constructor.

5.6.2. Abstract types (grammar)

Examples:

```
type T
type Q { function toString(t: T): string }
```

An abstract type is a special case of a type synonym that is underspecified. Such a type is declared simply by:

```
type Y<T>
```

Its definition can be stated in a refining module. The name `Y` can be immediately followed by a type characteristics suffix (Section 5.3.1). Because there is no defining RHS, the type characteristics cannot be inferred and so must be stated. If, in some refining module, a definition of the type is given, the type characteristics must match those of the new definition.

For example, the declarations

```
type T
function F(t: T): T
```

can be used to model an uninterpreted function `F` on some arbitrary type `T`. As another example,

```
type Monad<T>
```

can be used abstractly to represent an arbitrary parameterized monad.

Even as an abstract type, the type may be given members such as constants, methods or functions. For example,

```
abstract module P {
  type T {
    function ToString(): string
  }
}

module X refines P {
  newtype T = i | 0 <= i < 10 {
    function ToString(): string {  "" }
  }
}
```

The abstract type `P.T` has a declared member `ToString`, which can be called wherever `P.T` may be used. In the refining module `X`, `T` is declared to be a `newtype`, in which `ToString` now has a body.

It would be an error to refine `P.T` as a simple type synonym or subset type in `X`, say `type T = int`, because type synonyms may not have members.

5.6.3. Subset types (grammar)

Examples:

```
type Pos = i: int | i > 0 witness 1
type PosReal = r | r > 0.0 witness 1.0
type Empty = n: nat | n < 0 witness *
type Big = n: nat | n > 1000 ghost witness 10000
```

A *subset type* is a restricted use of an existing type, called the *base type* of the subset type. A subset type is like a combined use of the base type and a predicate on the base type.

An assignment from a subset type to its base type is always allowed. An assignment in the other direction, from the base type to a subset type, is allowed provided the value assigned does indeed satisfy the predicate of the subset type. This condition is checked by the verifier, not by the type checker. Similarly, assignments from one subset type to another (both with the same base type) are also permitted, as long as it can be established that the value being assigned satisfies the predicate defining the receiving subset type. (Note, in contrast, assignments between a newtype and its base type are never allowed, even if the value assigned is a value of the target type. For such assignments, an explicit conversion must be used, see Section 9.10.)

The declaration of a subset type permits an optional `witness` clause, to declare that there is a value that satisfies the subset type's predicate; that is, the witness clause establishes that the defined type is not empty. The compiler may, but is not obligated to, use this value when auto-initializing a newly declared variable of the subset type.

Dafny builds in three families of subset types, as described next.

5.6.3.1. Type `nat`

The built-in type `nat`, which represents the non-negative integers (that is, the natural numbers), is a subset type:

```
type nat = n: int | 0 <= n
```

A simple example that puts subset type `nat` to good use is the standard Fibonacci function:

```
function Fib(n: nat): nat
{
  if n < 2 then n else Fib(n-2) + Fib(n-1)
}
```

An equivalent, but clumsy, formulation of this function (modulo the wording of any error messages produced at call sites) would be to use type `int` and to write the restricting predicate in pre- and postconditions:

```
function Fib(n: int): int
  requires 0 <= n  // the function argument must be non-negative
  ensures 0 <= Fib(n)  // the function result is non-negative
{
  if n < 2 then n else Fib(n - 2) + Fib(n - 1)
}
```

5.6.3.2. Non-null types Every class, trait, and iterator declaration `C` gives rise to two types.

One type has the name `C?` (that is, the name of the class, trait, or iterator declaration with a `?` character appended to the end). The values of `C?` are the references to `C` objects, and also the value `null`. In other words, `C?` is the type of *possibly null* references (aka, *nullable* references) to `C` objects.

The other type has the name `C` (that is, the same name as the class, trait, or iterator declaration). Its values are the references to `C` objects, and does not contain the value `null`. In other words, `C` is the type of *non-null* references to `C` objects.

The type `C` is a subset type of `C?`:

```
type C = c: C? | c != null
```

(It may be natural to think of the type `C?` as the union of type `C` and the value `null`, but, technically, Dafny defines `C` as a subset type with base type `C?`.)

From being a subset type, we get that `C` is a subtype of `C?`. Moreover, if a class or trait `C` extends a trait `B`, then type `C` is a subtype of `B` and type `C?` is a subtype of `B?`.

Every possibly-null reference type is a subtype of the built-in possibly-null trait type `object?`, and every non-null reference type is a subtype of the built-in non-null trait type `object`. (And, from the fact that `object` is a subset type of `object?`, we also have that `object` is a subtype of `object?`.)

Arrays are references and array types also come in these two flavors. For example, `array?` and `array2?` are possibly-null (1- and 2-dimensional) array types, and `array` and `array2` are their respective non-null types.

Note that `?` is not an operator. Instead, it is simply the last character of the name of these various possibly-null types.

5.6.3.3. Arrow types: `->`, `-->`, and `~>` For more information about arrow types (function types), see Section 5.12. This section is a preview to point out the subset-type relationships among the kinds of function types.

The built-in type

- `->` stands for total functions,
- `-->` stands for partial functions (that is, functions with possible `requires` clauses), and
- `~>` stands for all functions.

More precisely, type constructors exist for any arity (`() -> X`, `A -> X`, `(A, B) -> X`, `(A, B, C) -> X`, etc.).

For a list of types `TT` and a type `U`, the values of the arrow type `(TT) ~> U` are functions from `TT` to `U`. This includes functions that may read the heap and functions that are not defined on all inputs. It is not common to need this generality (and working with such general functions is difficult). Therefore, Dafny defines two subset types that are more common (and much easier to work with).

The type `(TT) --> U` denotes the subset of `(TT) ~> U` where the functions do not read the (mutable parts of the) heap. Values of type `(TT) --> U` are called *partial functions*, and the subset type `(TT) --> U` is called the *partial arrow type*. (As a mnemonic to help you remember that this is the partial arrow, you may think of the little gap between the two hyphens in `-->` as showing a broken arrow.)

Intuitively, the built-in partial arrow type is defined as follows (here shown for arrows with arity 1):

```
type A --> B = f: A ~> B | forall a :: f.reads(a) == {}
```

(except that what is shown here left of the `=` is not legal Dafny syntax and that the restriction could not be verified as is). That is, the partial arrow type is defined as those functions `f` whose reads frame is empty for all inputs. More precisely, taking variance into account, the partial arrow type is defined as

```
type -A --> +B = f: A ~> B | forall a :: f.reads(a) == {}
```

The type `(TT) -> U` is, in turn, a subset type of `(TT) --> U`, adding the restriction that the functions must not impose any precondition. That is, values of type `(TT) -> U` are *total functions*, and the subset type `(TT) -> U` is called the *total arrow type*.

The built-in total arrow type is defined as follows (here shown for arrows with arity 1):

```
type -A -> +B = f: A --> B | forall a :: f.requires(a)
```

That is, the total arrow type is defined as those partial functions `f` whose precondition evaluates to `true` for all inputs.

Among these types, the most commonly used are the total arrow types. They are also the easiest to work with. Because they are common, they have the simplest syntax (`->`).

Note, informally, we tend to speak of all three of these types as arrow types, even though, technically, the `~>` types are the arrow types and the `-->` and `->` types are subset types thereof. The one place where you may need to remember that `-->` and `->` are subset types is in some error messages. For example, if you try to assign a partial function to a variable whose type is a total arrow type and the verifier is not able to prove that the partial function really is total, then you'll get an error saying that the subset-type constraint may not be satisfied.

For more information about arrow types, see Section 5.12.

5.6.3.4. Witness clauses The declaration of a subset type permits an optional `witness` clause. Types in Dafny are generally expected to be non-empty, in part because variables of any type are expected to have some value when they are used. In many cases, Dafny can determine that a newly declared type has some value. For example, in the absence of a witness clause, a numeric type that includes 0 is known by Dafny to be non-empty. However, Dafny cannot always make this determination. If it cannot, a `witness` clause is required. The value given in the `witness` clause must be a valid value for the type and assures Dafny that the type is non-empty. (The variation `witness *` is described below.)

For example,

```
type OddInt = x: int | x % 2 == 1
```

will give an error message, but

```
type OddInt = x: int | x % 2 == 1 witness 73
```

does not. Here is another example:

```
type NonEmptySeq = x: seq<int> | |x| > 0 witness [0]
```

If the witness is only available in ghost code, you can declare the witness as a `ghost witness`. In this case, the Dafny verifier knows that the type is non-empty, but it will not be able to auto-initialize a variable of that type in compiled code.

There is even room to do the following:

```
type BaseType
predicate RHS(x: BaseType)
type MySubset = x: BaseType | RHS(x) ghost witness MySubsetWitness()

function {:axiom} MySubsetWitness(): BaseType
  ensures RHS(MySubsetWitness())
```

Here the type is given a ghost witness: the result of the expression `MySubsetWitness()`, which is a call of a (ghost) function. Now that function has a postcondition saying that the returned value is indeed a candidate value for the declared type, so the verifier is satisfied regarding the non-emptiness of the type. However, the function has no body, so there is still no proof that there is indeed such a witness. You can either supply a, perhaps complicated, body to generate a viable candidate or you can be very sure, without proof, that there is indeed such a value. If you are wrong, you have introduced an unsoundness into your program.

In addition though, types are allowed to be empty or possibly empty. This is indicated by the clause `witness *`, which tells the verifier not to check for a satisfying witness. A declaration like this produces an empty type:

```
type ReallyEmpty = x: int | false witness *
```

The type can be used in code like

```
method M(x: ReallyEmpty) returns (seven: int)
  ensures seven == 7
{
  seven := 10;
}
```

which does verify. But the method can never be called because there is no value that can be supplied as the argument. Even this code

```
method P() returns (seven: int)
  ensures seven == 7
{
  var x: ReallyEmpty;
  seven := 10;
}
```

does not complain about `x` unless `x` is actually used, in which case it must have a value. The postcondition in `P` does not verify, but not because of the empty type.

5.7. Newtypes (grammar)

Examples:

```
newtype I = int
newtype D = i: int | 0 <= i < 10
newtype uint8 = i | 0 <= i < 256
```

A newtype is like a type synonym or subset type except that it declares a wholly new type name that is distinct from its base type. It also accepts an optional `witness` clause.

A new type can be declared with the *newtype* declaration, for example:

```
newtype N = x: M | Q
```

where `M` is a type and `Q` is a boolean expression that can use `x` as a free variable. If `M` is an integer-based numeric type, then so is `N`; if `M` is real-based, then so is `N`. If the type `M` can be inferred from `Q`, the "`: M`" can be omitted. If `Q` is just `true`, then the declaration can be given simply as:

```
newtype N = M
```

Type `M` is known as the *base type* of `N`. At present, Dafny only supports `int` and `real` as base types of newtypes.

A newtype is a type that supports the same operations as its base type. The newtype is distinct from and incompatible with other types; in particular, it is not assignable to its base type without an explicit conversion. An important difference between the operations on a newtype and the operations on its base type is that the newtype operations are defined only if the result satisfies the predicate `Q`, and likewise for the literals of the newtype.

For example, suppose `lo` and `hi` are integer-based numeric bounds that satisfy `0 <= lo <= hi` and consider the following code fragment:

```
var mid := (lo + hi) / 2;
```

If `lo` and `hi` have type `int`, then the code fragment is legal; in particular, it never overflows, since `int` has no upper bound. In contrast, if `lo` and `hi` are variables of a newtype `int32` declared as follows:

```
newtype int32 = x | -0x8000_0000 <= x < 0x8000_0000
```

then the code fragment is erroneous, since the result of the addition may fail to satisfy the predicate in the definition of `int32`. The code fragment can be rewritten as

```
var mid := lo + (hi - lo) / 2;
```

in which case it is legal for both `int` and `int32`.

An additional point with respect to arithmetic overflow is that for (signed) `int32` values `hi` and `lo` constrained only by `lo <= hi`, the difference `hi - lo` can also overflow the bounds of the `int32` type. So you could also write:

```
var mid := lo + (hi/2 - lo/2);
```

Since a newtype is incompatible with its base type and since all results of the newtype's operations are members of the newtype, a compiler for Dafny is free to specialize the run-time representation of the newtype. For example, by scrutinizing the definition of `int32` above, a compiler may decide to store `int32` values using signed 32-bit integers in the target hardware.

The incompatibility of a newtype and its basetype is intentional, as newtypes are meant to be used as distinct types from the basetype. If numeric types are desired that mix more readily with the basetype, the subset types described in Section 5.6.3 may be more appropriate.

Note that the bound variable `x` in `Q` has type `M`, not `N`. Consequently, it may not be possible to state `Q` about the `N` value. For example, consider the following type of 8-bit 2's complement integers:

```
newtype int8 = x: int | -128 <= x < 128
```

and consider a variable `c` of type `int8`. The expression

```
-128 <= c < 128
```

is not well-defined, because the comparisons require each operand to have type `int8`, which means the literal `128` is checked to be of type `int8`, which it is not. A proper way to write this expression is to use a conversion operation, described in Section 5.7.1, on `c` to convert it to the base type:

```
-128 <= c as int < 128
```

If possible, Dafny compilers will represent values of the newtype using a native type for the sake of efficiency. This action can be inhibited or a specific native data type selected by using the `{:nativeType}` attribute, as explained in Section 11.1.2.

Furthermore, for the compiler to be able to make an appropriate choice of representation, the constants in the defining expression as shown above must be known constants at compile-time. They need not be numeric literals; combinations of basic operations and symbolic constants are also allowed as described in Section 9.39.

5.7.1. Conversion operations

For every type `N`, there is a conversion operation with the name `as N`, described more fully in Section 9.10. It is a partial function defined when the given value, which can be of any type, is a member of the type converted to. When the conversion is from a real-based numeric type to an integer-based numeric type, the operation requires that the real-based argument have no fractional part. (To round a real-based numeric value down to the nearest integer, use the `.Floor` member, see Section 5.2.2.)

To illustrate using the example from above, if `lo` and `hi` have type `int32`, then the code fragment can legally be written as follows:

```
var mid := (lo as int + hi as int) / 2;
```

where the type of `mid` is inferred to be `int`. Since the result value of the division is a member of type `int32`, one can introduce yet another conversion operation to make the type of `mid` be `int32`:

```
var mid := ((lo as int + hi as int) / 2) as int32;
```

If the compiler does specialize the run-time representation for `int32`, then these statements come at the expense of two, respectively three, run-time conversions.

The `as N` conversion operation is grammatically a suffix operation like `.`field and array indexing, but binds less tightly than unary operations: `- x as int` is `(- x) as int`; `a + b as int` is `a + (b as int)`.

The `as N` conversion can also be used with reference types. For example, if `C` is a class, `c` is an expression of type `C`, and `o` is an expression of type `object`, then `c as object` and `c as object?` are upcasts and `o is C` is a downcast. A downcast requires the LHS expression to have the RHS type, as is enforced by the verifier.

For some types (in particular, reference types), there is also a corresponding `is` operation (Section 9.10) that tests whether a value is valid for a given type.

5.8. Class types (grammar)

Examples:

```
trait T {}
class A {}
class B extends T {
  const b: B?
  var v: int
  constructor (vv: int) { v := vv; b := null; }
  function toString(): string { "a B" }
  method m(i: int) { var x := new B(0); }
  static method q() {}
}
```

Declarations within a class all begin with keywords and do not end with semicolons.

A *class* `C` is a reference type declared as follows:

```
class C<T> extends J1, ..., Jn
{
  _members_
}
```

where the <>-enclosed list of one-or-more type parameters `T` is optional. The text "`extends J1, ..., Jn`" is also optional and says that the class extends traits `J1` … `Jn`. The members of a class are *fields*, *constant fields*, *functions*, and *methods*. These are accessed or invoked by dereferencing a reference to a `C` instance.

A function or method is invoked on an *instance* of `C`, unless the function or method is declared `static`. A function or method that is not `static` is called an *instance* function or method.

An instance function or method takes an implicit *receiver* parameter, namely, the instance used to access the member. In the specification and body of an instance function or method, the receiver parameter can be referred to explicitly by the keyword `this`. However, in such places, members of `this` can also be mentioned without any qualification. To illustrate, the qualified `this.f` and the unqualified `f` refer to the same field of the same object in the following example:

```
class C {
  var f: int
  var x: int
  method Example() returns (b: bool)
  {
    var x: int;
    b := f == this.f;
  }
}
```

so the method body always assigns `true` to the out-parameter `b`. However, in this example, `x` and `this.x` are different because the field `x` is shadowed by the declaration of the local variable `x`.

There is no semantic difference between qualified and unqualified accesses to the same receiver and member.

A `C` instance is created using `new`. There are three forms of `new`, depending on whether or not the class declares any *constructors* (see Section 6.3.2):

```
c := new C;
c := new C.Init(args);
c := new C(args);
```

For a class with no constructors, the first two forms can be used. The first form simply allocates a new instance of a `C` object, initializing its fields to values of their respective types (and initializing each `const` field with a RHS to its specified value). The second form additionally invokes an *initialization method* (here, named `Init`) on the newly allocated object and the given arguments. It is therefore a shorthand for

```
c := new C;
c.Init(args);
```

An initialization method is an ordinary method that has no out-parameters and that modifies no more than `this`.

For a class that declares one or more constructors, the second and third forms of `new` can be used. For such a class, the second form invokes the indicated constructor (here, named `Init`), which allocates and initializes the object. The third form is the same as the second, but invokes the *anonymous constructor* of the class (that is, a constructor declared with the empty-string name).

The details of constructors and other class members are described in Section 6.3.2.

5.9. Trait types (grammar)

A *trait* is an abstract superclass, similar to an "interface" or "mixin". A trait can be *extended* only by another trait or by a class (and in the latter case we say that the class *implements* the trait). More specifically, algebraic datatypes cannot extend traits.[4]

The declaration of a trait is much like that of a class:

```
trait J
{
  _members_
}
```

where *members* can include fields, constant fields, functions, methods and declarations of nested traits, but no constructor methods. The functions and methods are allowed to be declared `static`.

A reference type `C` that extends a trait `J` is assignable to a variable of type `J`; a value of type `J` is assignable to a variable of a reference type `C` that extends `J` only if the verifier can prove that the reference does indeed refer to an object of allocated type `C`. The members of `J` are available as members of `C`. A member in `J` is not allowed to be redeclared in `C`, except if the member is a non-`static` function or method without a body in `J`. By doing so, type `C` can supply a stronger specification and a body for the member. There is further discussion on this point in Section 5.9.2.

`new` is not allowed to be used with traits. Therefore, there is no object whose allocated type is a trait. But there can of course be objects of a class `C` that implement a trait `J`, and a reference to such a `C` object can be used as a value of type `J`.

5.9.1. Type `object` (grammar)

There is a built-in trait `object` that is implicitly extended by all classes and traits. It produces two types: the type `object?` that is a supertype of all reference types and a subset type `object` that is a supertype of all non-null reference types. This includes reference types like arrays and iterators that do not permit explicit extending of traits. The purpose of type `object` is to enable a uniform treatment of *dynamic frames*. In particular, it is useful to keep a ghost field (typically named `Repr` for "representation") of type `set<object>`.

It serves no purpose (but does no harm) to explicitly list the trait `object` as an extendee in a class or trait declaration.

Traits `object?` and `object` contain no members.

The dynamic allocation of objects is done using `new C`…, where `C` is the name of a class. The name `C` is not allowed to be a trait, except that it is allowed to be `object`. The construction `new object` allocates a new object (of an unspecified class type). The construction can be used to create unique references, where no other properties of those references are needed. (`new object?` makes no sense; always use `new object` instead because the result of `new` is always non-null.)

[4]Traits are new to Dafny and are likely to evolve for a while.

5.9.2. Inheritance

The purpose of traits is to be able to express abstraction: a trait encapsulates a set of behaviors; classes and traits that extend it *inherit* those behaviors, perhaps specializing them.

A trait or class may extend multiple other traits. The traits syntactically listed in a trait or class's `extends` clause are called its *direct parents*; the *transitive parents* of a trait or class are its direct parents, the transitive parents of its direct parents, and the `object` trait (if it is not itself `object`). These are sets of traits, in that it does not matter if there are repetitions of a given trait in a class or trait's direct or transitive parents. However, if a trait with type parameters is repeated, it must have the same actual type parameters in each instance. Furthermore, a trait may not be in its own set of transitive parents; that is, the graph of traits connected by the directed *extends* relationship may not have any cycles.

A class or trait inherits (as if they are copied) all the instance members of its transitive parents. However, since names may not be overloaded in Dafny, different members (that is, members with different type signatures) within the set of transitive parents and the class or trait itself must have different names.[^overload] This restriction does mean that traits from different sources that coincidentally use the same name for different purposes cannot be combined by being part of the set of transitive parents for some new trait or class.

A declaration of member `C.M` in a class or trait *overrides* any other declarations of the same name (and signature) in a transitive parent. `C.M` is then called an override; a declaration that does not override anything is called an *original declaration*.

Static members of a trait may not be redeclared; thus, if there is a body it must be declared in the trait; the compiler will require a body, though the verifier will not.

[//]: # Caution - a newline (not a blank line) ends a footnote [^overload]: It is possible to conceive of a mechanism for disambiguating conflicting names, but this would add complexity to the language that does not appear to be needed, at least as yet.

Where traits within an extension hierarchy do declare instance members with the same name (and thus the same signature), some rules apply. Recall that, for methods, every declaration includes a specification; if no specification is given explicitly, a default specification applies. Instance method declarations in traits, however, need not have a body, as a body can be declared in an override.

For a given non-static method M,

- A trait or class may not redeclare M if it has a transitive parent that declares M and provides a body.
- A trait may but need not provide a body if all its transitive parents that declare M do not declare a body.
- A trait or class may not have more than one transitive parent that declares M with a body.
- A class that has one or more transitive parents that declare M without a body and no transitive parent that declares M with a body must itself redeclare M with a body if it is compiled. (The verifier alone does not require a body.)
- Currently (and under debate), the following restriction applies: if `M` overrides two (or more) declarations, `P.M` and `Q.M`, then either `P.M` must override `Q.M` or `Q.M` must override `P.M`.

The last restriction above is the current implementation. It effectively limits inheritance of a method M to a single “chain” of declarations and does not permit mixins.

Each of any method declarations explicitly or implicitly includes a specification. In simple cases, those syntactically separate specifications will be copies of each other (up to renaming to take account of differing formal parameter names). However they need not be. The rule is that the specifications of M in a given class or trait must be *as strong as* M’s specifications in a transitive parent. Here *as strong as* means that it must be permitted to call the subtype’s M in the context of the supertype’s M. Stated differently, where P and C are a parent trait and a child class or trait, respectively, then, under the precondition of `P.M`,

- C.M’s `requires` clause must be implied by P.M’s `requires` clause
- C.M’s `ensures` clause must imply P.M’s `ensures` clause
- C.M’s `reads` set must be a subset of P.M’s `reads` set
- C.M’s `modifies` set must be a subset of P.M’s `modifies` set
- C.M’s `decreases` expression must be smaller than or equal to P.M’s `decreases` expression

Non-static const and field declarations are also inherited from parent traits. These may not be redeclared in extending traits and classes. However, a trait need not initialize a const field with a value. The class that extends a trait that declares such a const field without an initializer can initialize the field in a constructor. If the declaring trait does give an initial value in the declaration, the extending class or trait may not either redeclare the field or give it a value in a constructor.

When names are inherited from multiple traits, they must be different. If two traits declare a common name (even with the same signature), they cannot both be extendees of the same class or trait.

5.9.3. Example of traits

As an example, the following trait represents movable geometric shapes:

```
trait Shape
{
  function Width(): real
    reads this
    decreases 1
  method Move(dx: real, dy: real)
    modifies this
  method MoveH(dx: real)
    modifies this
  {
    Move(dx, 0.0);
  }
}
```

Members `Width` and `Move` are *abstract* (that is, body-less) and can be implemented differently by different classes that extend the trait. The implementation of method `MoveH` is given in the trait and thus is used by all classes that extend `Shape`. Here are two classes that each extend `Shape`:

```
class UnitSquare extends Shape
{
  var x: real, y: real
  function Width(): real
    decreases 0
  {  // note the empty reads clause
    1.0
  }
  method Move(dx: real, dy: real)
    modifies this
  {
    x, y := x + dx, y + dy;
  }
}

class LowerRightTriangle extends Shape
{
  var xNW: real, yNW: real, xSE: real, ySE: real
  function Width(): real
    reads this
    decreases 0
  {
    xSE - xNW
  }
  method Move(dx: real, dy: real)
    modifies this
  {
    xNW, yNW, xSE, ySE := xNW + dx, yNW + dy, xSE + dx, ySE + dy;
  }
}
```

Note that the classes can declare additional members, that they supply implementations for the abstract members of the trait, that they repeat the member signatures, and that they are responsible for providing their own member specifications that both strengthen the corresponding specification in the trait and are satisfied by the provided body. Finally, here is some code that creates two class instances and uses them together as shapes:

```
method m() {
  var myShapes: seq<Shape>;
  var A := new UnitSquare;
  myShapes := [A];
  var tri := new LowerRightTriangle;
  // myShapes contains two Shape values, of different classes
  myShapes := myShapes + [tri];
  // move shape 1 to the right by the width of shape 0
  myShapes[1].MoveH(myShapes[0].Width());
}
```

5.10. Array types (grammar)

Dafny supports mutable fixed-length *array types* of any positive dimension. Array types are (heap-based) reference types.

`arrayToken` is a kind of reserved token, such as `array`, `array?`, `array2`, `array2?`, `array3`, and so on (but not `array1`). The type parameter suffix giving the element type can be omitted if the element type can be inferred, though in that case it is likely that the `arrayToken` itself is also inferrable and can be omitted.

5.10.1. One-dimensional arrays

A one-dimensional array of `n` `T` elements may be initialized by any expression that returns a value of the desired type. Commonly, array allocation expressions are used. Some examples are shown here:

```
type T(0)
method m(n: nat) {
  var a := new T[n];
  var b: array<int> := new int[8];
  var c: array := new T[9];
}
```

The initial values of the array elements are arbitrary values of type `T`. A one-dimensional array value can also be assigned using an ordered list of expressions enclosed in square brackets, as follows:

```
a := new T[] [t1, t2, t3, t4];
```

The initialization can also use an expression that returns a function of type `nat -> T`:

```
a := new int[5](i => i*i);
```

In fact, the initializer can simply be a function name for the right type of function:

```
a := new int[5](Square);
```

The length of an array is retrieved using the immutable `Length` member. For example, the array allocated with `a := new T[n];` satisfies:

```
a.Length == n
```

Once an array is allocated, its length cannot be changed.

For any integer-based numeric `i` in the range `0 <= i < a.Length`, the *array selection* expression `a[i]` retrieves element `i` (that is, the element preceded by `i` elements in the array). The element stored at `i` can be changed to a value `t` using the array update statement:

```
a[i] := t;
```

Caveat: The type of the array created by `new T[n]` is `array<T>`. A mistake that is simple to make and that can lead to befuddlement is to write `array<T>` instead of `T` after `new`. For example, consider the following:

```
type T(0)
method m(n: nat) {
  var a := new array<T>;
  var b := new array<T>[n];
  var c := new array<T>(n);  // resolution error
  var d := new array(n);  // resolution error
}
```

The first statement allocates an array of type `array<T>`, but of unknown length. The second allocates an array of type `array<array<T>>` of length `n`, that is, an array that holds `n` values of type `array<T>`. The third statement allocates an array of type `array<T>` and then attempts to invoke an anonymous constructor on this array, passing argument `n`. Since `array` has no constructors, let alone an anonymous constructor, this statement gives rise to an error. If the type-parameter list is omitted for a type that expects type parameters, Dafny will attempt to fill these in, so as long as the `array` type parameter can be inferred, it is okay to leave off the "`<T>`" in the fourth statement above. However, as with the third statement, `array` has no anonymous constructor, so an error message is generated.

5.10.2. Converting arrays to sequences

One-dimensional arrays support operations that convert a stretch of consecutive elements into a sequence. For any array `a` of type `array<T>`, integer-based numeric bounds `lo` and `hi` satisfying `0 <= lo <= hi <= a.Length`, noting that bounds can equal the array's length, the following operations each yields a `seq<T>`:

expression	description
`a[lo..hi]`	subarray conversion to sequence
`a[lo..]`	drop
`a[..hi]`	take
`a[..]`	array conversion to sequence

The expression `a[lo..hi]` takes the first `hi` elements of the array, then drops the first `lo` elements thereof and returns what remains as a sequence, with length `hi - lo`. The other operations are special instances of the first. If `lo` is omitted, it defaults to `0` and if `hi` is omitted, it defaults to `a.Length`. In the last operation, both `lo` and `hi` have been omitted, thus `a[..]` returns the sequence consisting of all the array elements of `a`.

The subarray operations are especially useful in specifications. For example, the loop invariant of a binary search algorithm that uses variables `lo` and `hi` to delimit the subarray where the search `key` may still be found can be expressed as follows:

```
key !in a[..lo] && key !in a[hi..]
```

Another use is to say that a certain range of array elements have not been changed since the beginning of a method:

```
a[lo..hi] == old(a[lo..hi])
```

or since the beginning of a loop:

```
ghost var prevElements := a[..];
while // ...
  invariant a[lo..hi] == prevElements[lo..hi]
{
  // ...
}
```

Note that the type of `prevElements` in this example is `seq<T>`, if `a` has type `array<T>`.

A final example of the subarray operation lies in expressing that an array's elements are a permutation of the array's elements at the beginning of a method, as would be done in most sorting algorithms. Here, the subarray operation is combined with the sequence-to-multiset conversion:

```
multiset(a[..]) == multiset(old(a[..]))
```

5.10.3. Multi-dimensional arrays

An array of 2 or more dimensions is mostly like a one-dimensional array, except that `new` takes more length arguments (one for each dimension), and the array selection expression and the array update statement take more indices. For example:

```
matrix := new T[m, n];
matrix[i, j], matrix[x, y] := matrix[x, y], matrix[i, j];
```

create a 2-dimensional array whose dimensions have lengths `m` and `n`, respectively, and then swaps the elements at `i,j` and `x,y`. The type of `matrix` is `array2<T>`, and similarly for higher-dimensional arrays (`array3<T>`, `array4<T>`, etc.). Note, however, that there is no type `array0<T>`, and what could have been `array1<T>` is actually named just `array<T>`. (Accordingly, `array0` and `array1` are just normal identifiers, not type names.)

The `new` operation above requires `m` and `n` to be non-negative integer-based numerics. These lengths can be retrieved using the immutable fields `Length0` and `Length1`. For example, the following holds for the array created above:

```
matrix.Length0 == m && matrix.Length1 == n
```

Higher-dimensional arrays are similar (`Length0`, `Length1`, `Length2`, …). The array selection expression and array update statement require that the indices are in bounds. For example, the swap statement above is well-formed only if:

```
0 <= i < matrix.Length0 && 0 <= j < matrix.Length1 &&
0 <= x < matrix.Length0 && 0 <= y < matrix.Length1
```

In contrast to one-dimensional arrays, there is no operation to convert stretches of elements from a multi-dimensional array to a sequence.

There is however syntax to create a multi-dimensional array value using a function: see Section 9.16.

5.11. Iterator types (grammar)

See Section 7.5 for a description of iterator specifications.

An *iterator* provides a programming abstraction for writing code that iteratively returns elements. These CLU-style iterators are *co-routines* in the sense that they keep track of their own program counter and control can be transferred into and out of the iterator body.

An iterator is declared as follows:

```
iterator Iter<T>(_in-params_) yields (_yield-params_)
  _specification_
{
  _body_
}
```

where `T` is a list of type parameters (as usual, if there are no type parameters, "`<T>`" is omitted). This declaration gives rise to a reference type with the same name, `Iter<T>`. In the signature, in-parameters and yield-parameters are the iterator's analog of a method's in-parameters and out-parameters. The difference is that the out-parameters of a method are returned to a caller just once, whereas the yield-parameters of an iterator are returned each time the iterator body performs a `yield`. The body consists of statements, like in a method body, but with the availability also of `yield` statements.

From the perspective of an iterator client, the `iterator` declaration can be understood as generating a class `Iter<T>` with various members, a simplified version of which is described next.

The `Iter<T>` class contains an anonymous constructor whose parameters are the iterator's in-parameters:

```
predicate Valid()
constructor (_in-params_)
  modifies this
  ensures Valid()
```

An iterator is created using `new` and this anonymous constructor. For example, an iterator willing to return ten consecutive integers from `start` can be declared as follows:

```
iterator Gen(start: int) yields (x: int)
  yield ensures |xs| <= 10 && x == start + |xs| - 1
{
  var i := 0;
  while i < 10 invariant |xs| == i {
    x := start + i;
    yield;
    i := i + 1;
  }
}
```

An instance of this iterator is created using

```
iter := new Gen(30);
```

It is used like this:

```
method Main() {
  var i := new Gen(30);
  while true
    invariant i.Valid() && fresh(i._new)
    decreases 10 - |i.xs|
  {
    var m := i.MoveNext();
    if (!m) {break; }
    print i.x;
  }
}
```

The predicate `Valid()` says when the iterator is in a state where one can attempt to compute more elements. It is a postcondition of the constructor and occurs in the specification of the `MoveNext` member:

```
method MoveNext() returns (more: bool)
  requires Valid()
  modifies this
  ensures more ==> Valid()
```

Note that the iterator remains valid as long as `MoveNext` returns `true`. Once `MoveNext` returns `false`, the `MoveNext` method can no longer be called. Note, the client is under no obligation to keep calling `MoveNext` until it returns `false`, and the body of the iterator is allowed to keep returning elements forever.

The in-parameters of the iterator are stored in immutable fields of the iterator class. To illustrate in terms of the example above, the iterator class `Gen` contains the following field:

```
const start: int
```

The yield-parameters also result in members of the iterator class:

```
var x: int
```

These fields are set by the `MoveNext` method. If `MoveNext` returns `true`, the latest yield values are available in these fields and the client can read them from there.

To aid in writing specifications, the iterator class also contains ghost members that keep the history of values returned by `MoveNext`. The names of these ghost fields follow the names of the yield-parameters with an “`s`” appended to the name (to suggest plural). Name checking rules make sure these names do not give rise to ambiguities. The iterator class for `Gen` above thus contains:

```
ghost var xs: seq<int>
```

These history fields are changed automatically by `MoveNext`, but are not assignable by user code.

Finally, the iterator class contains some special fields for use in specifications. In particular, the iterator specification is recorded in the following immutable fields:

```
ghost var _reads: set<object>
ghost var _modifies: set<object>
ghost var _decreases0: T0
ghost var _decreases1: T1
// ...
```

where there is a `_decreases(`*i*`): T(`*i*`)` field for each component of the iterator's `decreases` clause.[5] In addition, there is a field:

```
ghost var _new: set<object>;
```

to which any objects allocated on behalf of the iterator body are added. The iterator body is allowed to remove elements from the `_new` set, but cannot by assignment to `_new` add any elements.

Note, in the precondition of the iterator, which is to hold upon construction of the iterator, the in-parameters are indeed in-parameters, not fields of `this`.

`reads` clauses on iterators have a different meaning than they do on functions and methods. Iterators may read any memory they like, but because arbitrary code may be executed whenever they `yield` control, they need to declare what memory locations must not be modified by other code in order to maintain correctness. The contents of an iterator's `reads` clauses become part of the `reads` clause of the implicitly created `Valid()` predicate. This means if client code modifies any of this state, it will not be able to establish the precondition for the iterator's `MoveNext()` method, and hence the iterator body will never resume if this state is modified.

It is regrettably tricky to use iterators. The language really ought to have a `foreach` statement to make this easier. Here is an example showing a definition and use of an iterator.

```
iterator Iter<T(0)>(s: set<T>) yields (x: T)
  yield ensures x in s && x !in xs[..|xs|-1]
  ensures s == set z | z in xs
{
  var r := s;
  while (r != {})
    invariant r !! set z | z in xs
    invariant s == r + set z | z in xs
  {
    var y :| y in r;
    assert y !in xs;
    r, x := r - {y}, y;
    assert y !in xs;
```

[5]It would make sense to rename the special fields `_reads` and `_modifies` to have the same names as the corresponding keywords, `reads` and `modifies`, as is done for function values. Also, the various `_decreases\(_i_\)` fields can be combined into one field named `decreases` whose type is a n-tuple. These changes may be incorporated into a future version of Dafny.

```
    yield;
    assert y == xs[|xs|-1]; // a lemma to help prove loop invariant
  }
}

method UseIterToCopy<T(0)>(s: set<T>) returns (t: set<T>)
  ensures s == t
{
  t := {};
  var m := new Iter(s);
  while (true)
    invariant m.Valid() && fresh(m._new)
    invariant t == set z | z in m.xs
    decreases s - t
  {
    var more := m.MoveNext();
    if (!more) { break; }
    t := t + {m.x};
  }
}
```

The design of iterators is under discussion and may change.

5.12. Arrow types (grammar)

Examples:

```
(int) -> int
(bool,int) ~> bool
() --> object?
```

Functions are first-class values in Dafny. The types of function values are called *arrow types* (aka, *function types*). Arrow types have the form `(TT) ~> U` where `TT` is a (possibly empty) comma-delimited list of types and `U` is a type. `TT` is called the function's *domain type(s)* and `U` is its *range type*. For example, the type of a function

```
function F(x: int, arr: array<bool>): real
  requires x < 1000
  reads arr
```

is `(int, array<bool>) ~> real`.

As seen in the example above, the functions that are values of a type `(TT) ~> U` can have a precondition (as indicated by the `requires` clause) and can read values in the heap (as indicated by the `reads` clause). As described in Section 5.6.3.3,

- the subset type `(TT) --> U` denotes partial (but heap-independent) functions
- and the subset type `(TT) -> U` denotes total functions.

A function declared without a `reads` clause is known by the type checker to be a partial function. For example, the type of

```
function F(x: int, b: bool): real
  requires x < 1000
```

is `(int, bool) --> real`. Similarly, a function declared with neither a `reads` clause nor a `requires` clause is known by the type checker to be a total function. For example, the type of

```
function F(x: int, b: bool): real
```

is `(int, bool) -> real`. In addition to functions declared by name, Dafny also supports anonymous functions by means of *lambda expressions* (see Section 9.13).

To simplify the appearance of the basic case where a function's domain consists of a list of exactly one non-function, non-tuple type, the parentheses around the domain type can be dropped in this case. For example, you may write just `T -> U` for a total arrow type. This innocent simplification requires additional explanation in the case where that one type is a tuple type, since tuple types are also written with enclosing parentheses. If the function takes a single argument that is a tuple, an additional set of parentheses is needed. For example, the function

```
function G(pair: (int, bool)): real
```

has type `((int, bool)) -> real`. Note the necessary double parentheses. Similarly, a function that takes no arguments is different from one that takes a 0-tuple as an argument. For instance, the functions

```
function NoArgs(): real
function Z(unit: ()): real
```

have types `() -> real` and `(()) -> real`, respectively.

The function arrows are right associative. For example, `A -> B -> C` means `A -> (B -> C)`, whereas the other association requires explicit parentheses: `(A -> B) -> C`. As another example, `A -> B --> C ~> D` means `A -> (B --> (C ~> D))`.

Note that the receiver parameter of a named function is not part of the type. Rather, it is used when looking up the function and can then be thought of as being captured into the function definition. For example, suppose function `F` above is declared in a class `C` and that `c` references an object of type `C`; then, the following is type correct:

```
var f: (int, bool) -> real := c.F;
```

whereas it would have been incorrect to have written something like:

```
var f': (C, int, bool) -> real := F;  // not correct
```

The arrow types themselves do not divide a function's parameters into ghost versus non-ghost. Instead, a function used as a first-class value is considered to be ghost if either the function or any of its arguments is ghost. The following example program illustrates:

```
function F(x: int, ghost y: int): int
{
  x
}

method Example() {
  ghost var f: (int, int) -> int;
  var g: (int, int) -> int;
  var h: (int) -> int;
  var x: int;
  f := F;
  x := F(20, 30);
  g := F; // error: tries to assign ghost to non-ghost
  h := F; // error: wrong arity (and also tries to assign ghost to non-ghost)
}
```

In addition to its type signature, each function value has three properties, described next.

Every function implicitly takes the heap as an argument. No function ever depends on the *entire* heap, however. A property of the function is its declared upper bound on the set of heap locations it depends on for a given input. This lets the verifier figure out that certain heap modifications have no effect on the value returned by a certain function. For a function `f: T ~> U` and a value `t` of type `T`, the dependency set is denoted `f.reads(t)` and has type `set<object>`.

The second property of functions stems from the fact that every function is potentially *partial*. In other words, a property of a function is its *precondition*. For a function `f: T ~> U`, the precondition of `f` for a parameter value `t` of type `T` is denoted `f.requires(t)` and has type `bool`.

The third property of a function is more obvious—the function's body. For a function `f: T ~> U`, the value that the function yields for an input `t` of type `T` is denoted `f(t)` and has type `U`.

Note that `f.reads` and `f.requires` are themselves functions. Without loss of generality, suppose `f` is defined as:

```
function f<T,U>(x: T): U
  reads R(x)
  requires P(x)
{
  body(x)
}
```

where `P`, `R`, and `body` are declared as:

```
predicate P<T>(x: T)
function R<T>(x: T): set<object>
function body<T,U>(x: T): U
```

Then, `f.reads` is a function of type `T ~> set<object?>` whose `reads` and `requires` properties are given by the definition:

```
function f.reads<T>(x: T): set<object>
  reads R(x)
  requires P(x)
{
  R(x)
}
```

`f.requires` is a function of type `T ~> bool` whose `reads` and `requires` properties are given by the definition:

```
predicate f_requires<T>(x: T)
  requires true
  reads if P(x) then R(x) else *
{
  P(x)
}
```

where `*` is a notation to indicate that any memory location can be read, but is not valid Dafny syntax.

In these examples, if `f` instead had type `T --> U` or `T -> U`, then the type of `f.reads` is `T -> set<object?>` and the type of `f.requires` is `T -> bool`.

Dafny also supports anonymous functions by means of *lambda expressions*. See Section 9.13.

5.13. Tuple types

```
TupleType = "(" [ [ "ghost" ] Type { "," [ "ghost" ] Type } ] ")"
```

Dafny builds in record types that correspond to tuples and gives these a convenient special syntax, namely parentheses. For example, for what might have been declared as

```
datatype Pair<T,U> = Pair(0: T, 1: U)
```

Dafny provides the type `(T, U)` and the constructor `(t, u)`, as if the datatype's name were "" (i.e., an empty string) and its type arguments are given in round parentheses, and as if the constructor name were the empty string. Note that the destructor names are `0` and `1`, which are legal identifier names for members. For an example showing the use of a tuple destructor, here is a property that holds of 2-tuples (that is, *pairs*):

```
method m(){
  assert (5, true).1 == true;
}
```

Dafny declares *n*-tuples where *n* is 0 or 2 or more. There are no 1-tuples, since parentheses around a single type or a single value have no semantic meaning. The 0-tuple type, `()`, is often known as the *unit type* and its single value, also written `()`, is known as *unit*.

The `ghost` modifier can be used to mark tuple components as being used for specification only:

```
const pair: (int, ghost int) := (1, ghost 2)
```

5.14. Algebraic Datatypes (grammar)

Dafny offers two kinds of algebraic datatypes, those defined inductively (with `datatype`) and those defined coinductively (with `codatatype`). The salient property of every datatype is that each value of the type uniquely identifies one of the datatype's constructors and each constructor is injective in its parameters.

5.14.1. Inductive datatypes

The values of inductive datatypes can be seen as finite trees where the leaves are values of basic types, numeric types, reference types, coinductive datatypes, or arrow types. Indeed, values of inductive datatypes can be compared using Dafny's well-founded `<` ordering.

An inductive datatype is declared as follows:

```
datatype D<T> = _Ctors_
```

where *Ctors* is a nonempty `|`-separated list of *(datatype) constructors* for the datatype. Each constructor has the form:

```
C(_params_)
```

where *params* is a comma-delimited list of types, optionally preceded by a name for the parameter and a colon, and optionally preceded by the keyword `ghost`. If a constructor has no parameters, the parentheses after the constructor name may be omitted. If no constructor takes a parameter, the type is usually called an *enumeration*; for example:

```
datatype Friends = Agnes | Agatha | Jermaine | Jack
```

For every constructor `C`, Dafny defines a *discriminator* `C?`, which is a member that returns `true` if and only if the datatype value has been constructed using `C`. For every named parameter `p` of a constructor `C`, Dafny defines a *destructor* `p`, which is a member that returns the `p` parameter from the `C` call used to construct the datatype value; its use requires that `C?` holds. For example, for the standard `List` type

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>)
```

the following holds:

```
method m() {
  assert Cons(5, Nil).Cons? && Cons(5, Nil).head == 5;
}
```

Note that the expression

```
Cons(5, Nil).tail.head
```

is not well-formed by itself, since `Cons(5, Nil).tail` does not necessarily satisfy `Cons?`.

A constructor can have the same name as the enclosing datatype; this is especially useful for single-constructor datatypes, which are often called *record types*. For example, a record type for black-and-white pixels might be represented as follows:

```
datatype Pixel = Pixel(x: int, y: int, on: bool)
```

To call a constructor, it is usually necessary only to mention the name of the constructor, but if this is ambiguous, it is always possible to qualify the name of constructor by the name of the datatype. For example, `Cons(5, Nil)` above can be written

```
List.Cons(5, List.Nil)
```

As an alternative to calling a datatype constructor explicitly, a datatype value can be constructed as a change in one parameter from a given datatype value using the *datatype update* expression. For any `d` whose type is a datatype that includes a constructor `C` that has a parameter (destructor) named `f` of type `T`, and any expression `t` of type `T`,

```
d.(f := t)
```

constructs a value like `d` but whose `f` parameter is `t`. The operation requires that `d` satisfies `C?`. For example, the following equality holds:

```
method m(){
  assert Cons(4, Nil).(tail := Cons(3, Nil)) == Cons(4, Cons(3, Nil));
}
```

The datatype update expression also accepts multiple field names, provided these are distinct. For example, a node of some inductive datatype for trees may be updated as follows:

```
node.(left := L, right := R)
```

The operator `<` is defined for two operands of the same datataype. It means *is properly contained in*. For example, in the code

```
datatype X = T(t: X) | I(i: int)
method comp() {
  var x := T(I(0));
  var y := I(0);
  var z := I(1);
  assert x.t < x;
  assert y < x;
  assert !(x < x);
  assert z < x; // FAILS
}
```

`x` is a datatype value that holds a `T` variant, which holds a `I` variant, which holds an integer `0`. The value `x.t` is a portion of the datatype structure denoted by `x`, so `x.t < x` is true. Datatype values are immutable mathematical values, so the value of `y` is identical to the value of `x.t`, so `y < x` is true also, even though `y` is constructed from the ground up, rather than as a portion of `x`. However, `z` is different than either `y` or `x.t` and consequently `z < x` is not provable. Furthermore, `<` does not include `==`, so `x < x` is false.

Note that only `<` is defined; not `<=` or `>` or `>=`.

Also, `<` is underspecified. With the above code, one can prove neither `z < x` nor `!(z < x)` and neither `z < y` nor `!(z < y)`. In each pair, though, one or the other is true, so `(z < x) || !(z < x)` is provable.

5.14.2. Coinductive datatypes

Whereas Dafny insists that there is a way to construct every inductive datatype value from the ground up, Dafny also supports *coinductive datatypes*, whose constructors are evaluated lazily, and hence the language allows infinite structures. A coinductive datatype is declared using the keyword `codatatype`; other than that, it is declared and used like an inductive datatype.

For example,

```
codatatype IList<T> = Nil | Cons(head: T, tail: IList<T>)
codatatype Stream<T> = More(head: T, tail: Stream<T>)
codatatype Tree<T> = Node(left: Tree<T>, value: T, right: Tree<T>)
```

declare possibly infinite lists (that is, lists that can be either finite or infinite), infinite streams (that is, lists that are always infinite), and infinite binary trees (that is, trees where every branch goes on forever), respectively.

The paper Co-induction Simply, by Leino and Moskal(Leino and Moskal 2014a), explains Dafny's implementation and verification of coinductive types. We capture the key features from that paper in the following section but the reader is referred to that paper for more complete details and to supply bibliographic references that are omitted here.

5.14.3. Coinduction

Mathematical induction is a cornerstone of programming and program verification. It arises in data definitions (e.g., some algebraic data structures can be described using induction), it underlies program semantics (e.g., it explains how to reason about finite iteration and recursion), and it is used in proofs (e.g., supporting lemmas about data structures use inductive proofs). Whereas induction deals with finite things (data, behavior, etc.), its dual, coinduction, deals with possibly infinite things. Coinduction, too, is important in programming and program verification: it arises in data definitions (e.g., lazy data structures), semantics (e.g., concurrency), and proofs (e.g., showing refinement in a coinductive big-step semantics). It is thus desirable to have good support for both induction and coinduction in a system for constructing and reasoning about programs.

Co-datatypes and co-recursive functions make it possible to use lazily evaluated data structures (like in Haskell or Agda). *Greatest predicates*, defined by greatest fix-points, let programs state properties of such data structures (as can also be done in, for example, Coq). For the purpose of writing coinductive proofs in the language, we introduce greatest and least lemmas. A greatest lemma invokes the coinduction hypothesis much like an inductive proof invokes the induction hypothesis. Underneath the hood, our coinductive proofs are actually approached via induction: greatest and least lemmas provide a syntactic veneer around this approach.

The following example gives a taste of how the coinductive features in Dafny come together to give straightforward definitions of infinite matters.

```
// infinite streams
codatatype IStream<T> = ICons(head: T, tail: IStream<T>)

// pointwise product of streams
function Mult(a: IStream<int>, b: IStream<int>): IStream<int>
{ ICons(a.head * b.head, Mult(a.tail, b.tail)) }

// lexicographic order on streams
greatest predicate Below(a: IStream<int>, b: IStream<int>)
{ a.head <= b.head &&
  ((a.head == b.head) ==> Below(a.tail, b.tail))
}

// a stream is Below its Square
greatest lemma Theorem_BelowSquare(a: IStream<int>)
  ensures Below(a, Mult(a, a))
{ assert a.head <= Mult(a, a).head;
  if a.head == Mult(a, a).head {
    Theorem_BelowSquare(a.tail);
  }
}

// an incorrect property and a bogus proof attempt
greatest lemma NotATheorem_SquareBelow(a: IStream<int>)
  ensures Below(Mult(a, a), a) // ERROR
{
  NotATheorem_SquareBelow(a);
}
```

The example defines a type `IStream` of infinite streams, with constructor `ICons` and destructors `head` and `tail`. Function `Mult` performs pointwise multiplication on infinite streams of integers, defined using a co-recursive call (which is evaluated lazily). Greatest predicate `Below` is defined as a greatest fix-point, which intuitively means that the co-predicate will take on the value true if the recursion goes on forever without determining a different value. The greatest lemma states the theorem `Below(a, Mult(a, a))`. Its body gives the proof, where the recursive invocation of the co-lemma corresponds to an invocation of the coinduction hypothesis.

The proof of the theorem stated by the first co-lemma lends itself to the following intuitive reading: To prove that `a` is below `Mult(a, a)`, check that their heads are ordered and, if the heads are equal, also prove that the tails are ordered. The second co-lemma states a property that does not always hold; the verifier is not fooled by the bogus proof attempt and instead reports the property as unproved.

We argue that these definitions in Dafny are simple enough to level the playing field between induction (which is familiar) and coinduction (which, despite being the dual of induction, is often perceived as eerily mysterious). Moreover, the automation provided by our SMT-based verifier reduces the tedium in writing coinductive proofs. For example, it verifies `Theorem_BelowSquare` from the program text given above—no additional lemmas or tactics are needed. In fact, as a consequence of the automatic-induction heuristic in Dafny, the verifier will automatically verify `Theorem_BelowSquare` even given an empty body.

Just like there are restrictions on when an *inductive hypothesis* can be invoked, there are restrictions on how a *coinductive* hypothesis can be *used*. These are, of course, taken into consideration by Dafny's verifier. For example, as illustrated by the second greatest lemma above, invoking the coinductive hypothesis in an attempt to obtain the entire proof goal is futile. (We explain how this works in the section about greatest lemmas) Our initial experience with coinduction in Dafny shows it to provide an intuitive, low-overhead user experience that compares favorably to even the best of today's interactive proof assistants for coinduction. In addition, the coinductive features and verification support in Dafny have other potential benefits. The features are a stepping stone for verifying functional lazy programs with Dafny. Coinductive features have also shown to be useful in defining language semantics, as needed to verify the correctness of a compiler, so this opens the possibility that such verifications can benefit from SMT automation.

5.14.3.1. Well-Founded Function/Method Definitions

The Dafny programming language supports functions and methods. A *function* in Dafny is a mathematical function (i.e., it is well-defined, deterministic, and pure), whereas a *method* is a body of statements that can mutate the state of the program. A function is defined by its given body, which is an expression. To ensure that function definitions are mathematically consistent, Dafny insists that recursive calls be well-founded, enforced as follows: Dafny computes the call graph of functions. The strongly connected components within it are *clusters* of mutually recursive definitions; the clusters are arranged in a DAG. This stratifies the functions so that a call from one cluster in the DAG to a lower cluster is allowed arbitrarily. For an intra-cluster call, Dafny prescribes a proof obligation that is taken through the program verifier's reasoning engine. Semantically, each function activation is labeled by a *rank*—a lexicographic tuple determined by evaluating the function's `decreases` clause upon invocation of the function. The proof obligation for an intra-cluster call is thus that the rank of the callee is strictly less (in a language-defined well-founded relation) than the rank of the caller. Because these well-founded checks correspond to proving termination of executable code, we will often refer to them as "termination checks". The same process applies to methods.

Lemmas in Dafny are commonly introduced by declaring a method, stating the property of the lemma in the *postcondition* (keyword `ensures`) of the method, perhaps restricting the domain of the lemma by also giving a *precondition* (keyword `requires`), and using the lemma by invoking the method. Lemmas are stated, used, and proved as methods, but since they have no use at run time, such lemma methods are typically declared as *ghost*, meaning that they are not compiled into code. The keyword `lemma` introduces such a method. Control flow statements correspond to proof techniques—case splits are introduced with if statements, recursion and loops are used for induction, and method calls for structuring the proof. Additionally, the statement:

```
forall x | P(x) { Lemma(x); }
```

is used to invoke `Lemma(x)` on all `x` for which `P(x)` holds. If `Lemma(x)` ensures `Q(x)`, then the forall statement establishes

```
forall x :: P(x) ==> Q(x).
```

5.14.3.2. Defining Coinductive Datatypes Each value of an inductive datatype is finite, in the sense that it can be constructed by a finite number of calls to datatype constructors. In contrast, values of a coinductive datatype, or co-datatype for short, can be infinite. For example, a co-datatype can be used to represent infinite trees.

Syntactically, the declaration of a co-datatype in Dafny looks like that of a datatype, giving prominence to the constructors (following Coq). The following example defines a co-datatype Stream of possibly infinite lists.

```
codatatype Stream<T> = SNil | SCons(head: T, tail: Stream)
function Up(n: int): Stream<int> { SCons(n, Up(n+1)) }
function FivesUp(n: int): Stream<int>
  decreases 4 - (n - 1) % 5
{
  if (n % 5 == 0) then
    SCons(n, FivesUp(n+1))
  else
    FivesUp(n+1)
}
```

`Stream` is a coinductive datatype whose values are possibly infinite lists. Function `Up` returns a stream consisting of all integers upwards of `n` and `FivesUp` returns a stream consisting of all multiples of 5 upwards of `n` . The self-call in `Up` and the first self-call in `FivesUp` sit in productive positions and are therefore classified as co-recursive calls, exempt from termination checks. The second self-call in `FivesUp` is not in a productive position and is therefore subject to termination checking; in particular, each recursive call must decrease the rank defined by the `decreases` clause.

Analogous to the common finite list datatype, `Stream` declares two constructors, `SNil` and `SCons`. Values can be destructed using match expressions and statements. In addition, like for inductive datatypes, each constructor `C` automatically gives rise to a discriminator `C?` and each parameter of a constructor can be named in order to introduce a corresponding destructor. For example, if `xs` is the stream `SCons(x, ys)`, then `xs.SCons?` and `xs.head == x` hold. In contrast to datatype declarations, there is no grounding check for co-datatypes—since a codatatype admits infinite values, the type is nevertheless inhabited.

5.14.3.3. Creating Values of Co-datatypes To define values of co-datatypes, one could imagine a "co-function" language feature: the body of a "co-function" could include possibly never-ending self-calls that are interpreted by a greatest fix-point semantics (akin to a **CoFixpoint** in Coq). Dafny uses a different design: it offers only functions (not "co-functions"), but it classifies each intra-cluster call as either *recursive* or *co-recursive*. Recursive calls are subject to termination checks. Co-recursive calls may be never-ending, which is what is needed to define infinite values of a co-datatype. For example, function `Up(n)` in the preceding example is defined as the stream of numbers from `n` upward: it returns a stream that starts with `n` and continues as the co-recursive call `Up(n + 1)`.

To ensure that co-recursive calls give rise to mathematically consistent definitions, they must occur only in productive positions. This says that it must be possible to determine each successive piece of a co-datatype value after a finite amount of work. This condition is satisfied if every co-recursive call is syntactically guarded by a constructor of a co-datatype, which is the criterion Dafny uses to classify intra-cluster calls as being either co-recursive or recursive. Calls that are classified as co-recursive are exempt from termination checks.

A consequence of the productivity checks and termination checks is that, even in the absence of talking about least or greatest fix-points of self-calling functions, all functions in Dafny are deterministic. Since there cannot be multiple fix-points, the language allows one function to be involved in both recursive and co-recursive calls, as we illustrate by the function `FivesUp`.

5.14.3.4. Co-Equality

Equality between two values of a co-datatype is a built-in co-predicate. It has the usual equality syntax `s == t`, and the corresponding prefix equality is written `s ==#[k] t`. And similarly for `s != t` and `s !=#[k] t`.

5.14.3.5. Greatest predicates

Determining properties of co-datatype values may require an infinite number of observations. To that end, Dafny provides *greatest predicates* which are function declarations that use the `greatest predicate` keyword phrase. Self-calls to a greatest predicate need not terminate. Instead, the value defined is the greatest fix-point of the given recurrence equations. Continuing the preceding example, the following code defines a greatest predicate that holds for exactly those streams whose payload consists solely of positive integers. The greatest predicate definition implicitly also gives rise to a corresponding prefix predicate, `Pos#`. The syntax for calling a prefix predicate sets apart the argument that specifies the prefix length, as shown in the last line; for this figure, we took the liberty of making up a coordinating syntax for the signature of the automatically generated prefix predicate (which is not part of Dafny syntax).

```
greatest predicate Pos[nat](s: Stream<int>)
{
  match s
  case SNil => true
  case SCons(x, rest) => x > 0 && Pos(rest)
}
```

The following code is automatically generated by the Dafny compiler:

```
predicate Pos#[_k: nat](s: Stream<int>)
  decreases _k
{ if _k == 0 then true else
  match s
  case SNil => true
  case SCons(x, rest) => x > 0 && Pos#[_k-1](rest)
}
```

Some restrictions apply. To guarantee that the greatest fix-point always exists, the (implicit functor defining the) greatest predicate must be monotonic. This is enforced by a syntactic restriction on the form of the body of greatest predicates: after conversion to negation normal form (i.e., pushing negations down to the atoms), intra-cluster calls of greatest predicates must appear only in *positive* positions—that is, they must appear as atoms and must not be negated. Additionally, to guarantee soundness later on, we require that they appear in *continous* positions—that is, in negation normal form, when they appear under existential quantification, the quantification needs to be limited to a finite range[6]. Since the evaluation of a greatest predicate might not terminate, greatest predicates are always ghost. There is also a restriction on the call graph that a cluster containing a greatest predicate must contain only greatest predicates, no other kinds of functions.

A **greatest predicate** declaration of `P` defines not just a greatest predicate, but also a corresponding *prefix predicate* `P#`. A prefix predicate is a finite unrolling of a co-predicate. The prefix predicate is constructed from the co-predicate by

- adding a parameter `_k` of type `nat` to denote the prefix length,
- adding the clause `decreases _k;` to the prefix predicate (the greatest predicate itself is not allowed to have a decreases clause),
- replacing in the body of the greatest predicate every intra-cluster call `Q(args)` to a greatest predicate by a call `Q#[_k - 1](args)` to the corresponding prefix predicate, and then
- prepending the body with `if _k == 0 then true else`.

For example, for greatest predicate `Pos`, the definition of the prefix predicate `Pos#` is as suggested above. Syntactically, the prefix-length argument passed to a prefix predicate to indicate how many times to unroll the definition is written in square brackets, as in `Pos#[k](s)`. In the Dafny grammar this is called a `HashCall`. The definition of `Pos#` is available only at clusters strictly higher than that of `Pos`; that is, `Pos` and `Pos#` must not be in the same cluster. In other words, the definition of `Pos` cannot depend on `Pos#`.

5.14.3.6. Coinductive Proofs

From what we have said so far, a program can make use of properties of co-datatypes. For example, a method that declares `Pos(s)` as a precondition can rely on the stream `s` containing only positive integers. In this section, we consider how such properties are established in the first place.

5.14.3.6.1. Properties of Prefix Predicates

Among other possible strategies for establishing coinductive properties we take the time-honored approach of reducing coinduction to induction. More precisely, Dafny passes to the SMT solver an assumption `D(P)` for every greatest predicate `P`, where:

```
D(P) = forall x • P(x) <==> forall k • P#[k](x)
```

[6] To be specific, Dafny has two forms of extreme predicates and lemmas, one in which `_k` has type `nat` and one in which it has type `ORDINAL` (the default). The continuous restriction applies only when `_k` is `nat`. Also, higher-order function support in Dafny is rather modest and typical reasoning patterns do not involve them, so this restriction is not as limiting as it would have been in, e.g., Coq.

In other words, a greatest predicate is true iff its corresponding prefix predicate is true for all finite unrollings.

In Sec. 4 of the paper [Co-induction Simply] a soundness theorem of such assumptions is given, provided the greatest predicates meet the continous restrictions. An example proof of `Pos(Up(n))` for every `n > 0` is shown here:

```
lemma UpPosLemma(n: int)
  requires n > 0
  ensures Pos(Up(n))
{
  forall k | 0 <= k { UpPosLemmaK(k, n); }
}

lemma UpPosLemmaK(k: nat, n: int)
  requires n > 0
  ensures Pos#[k](Up(n))
  decreases k
{
  if k != 0 {
    // this establishes Pos#[k-1](Up(n).tail)
    UpPosLemmaK(k-1, n+1);
  }
}
```

The lemma `UpPosLemma` proves `Pos(Up(n))` for every `n > 0`. We first show `Pos#[k](Up(n))`, for `n > 0` and an arbitrary `k`, and then use the forall statement to show `forall k • Pos#[k](Up(n))`. Finally, the axiom `D(Pos)` is used (automatically) to establish the greatest predicate.

5.14.3.6.2. Greatest lemmas As we just showed, with help of the `D` axiom we can now prove a greatest predicate by inductively proving that the corresponding prefix predicate holds for all prefix lengths `k`. In this section, we introduce *greatest lemma* declarations, which bring about two benefits. The first benefit is that greatest lemmas are syntactic sugar and reduce the tedium of having to write explicit quantifications over `k`. The second benefit is that, in simple cases, the bodies of greatest lemmas can be understood as coinductive proofs directly. As an example consider the following greatest lemma.

```
greatest lemma UpPosLemma(n: int)
  requires n > 0
  ensures Pos(Up(n))
{
  UpPosLemma(n+1);
}
```

This greatest lemma can be understood as follows: `UpPosLemma` invokes itself co-recursively to obtain the proof for `Pos(Up(n).tail)` (since `Up(n).tail` equals `Up(n+1)`). The proof glue needed to then conclude `Pos(Up(n))` is provided automatically, thanks to the power of the

SMT-based verifier.

5.14.3.6.3. Prefix Lemmas To understand why the above `UpPosLemma` greatest lemma code is a sound proof, let us now describe the details of the desugaring of greatest lemmas. In analogy to how a **greatest predicate** declaration defines both a greatest predicate and a prefix predicate, a **greatest lemma** declaration defines both a greatest lemma and *prefix lemma*. In the call graph, the cluster containing a greatest lemma must contain only greatest lemmas and prefix lemmas, no other methods or function. By decree, a greatest lemma and its corresponding prefix lemma are always placed in the same cluster. Both greatest lemmas and prefix lemmas are always ghost code.

The prefix lemma is constructed from the greatest lemma by

- adding a parameter `_k` of type `nat` to denote the prefix length,
- replacing in the greatest lemma's postcondition the positive continuous occurrences of greatest predicates by corresponding prefix predicates, passing in `_k` as the prefix-length argument,
- prepending `_k` to the (typically implicit) **decreases** clause of the greatest lemma,
- replacing in the body of the greatest lemma every intra-cluster call `M(args)` to a greatest lemma by a call `M#[_k - 1](args)` to the corresponding prefix lemma, and then
- making the body's execution conditional on `_k != 0`.

Note that this rewriting removes all co-recursive calls of greatest lemmas, replacing them with recursive calls to prefix lemmas. These recursive calls are, as usual, checked to be terminating. We allow the pre-declared identifier `_k` to appear in the original body of the greatest lemma.[7]

We can now think of the body of the greatest lemma as being replaced by a **forall** call, for every k , to the prefix lemma. By construction, this new body will establish the greatest lemma's declared postcondition (on account of the `D` axiom, and remembering that only the positive continuous occurrences of greatest predicates in the greatest lemma's postcondition are rewritten), so there is no reason for the program verifier to check it.

The actual desugaring of our greatest lemma `UpPosLemma` is in fact the previous code for the `UpPosLemma` lemma except that `UpPosLemmaK` is named `UpPosLemma#` and modulo a minor syntactic difference in how the `k` argument is passed.

In the recursive call of the prefix lemma, there is a proof obligation that the prefixlength argument `_k - 1` is a natural number. Conveniently, this follows from the fact that the body has been wrapped in an `if _k != 0` statement. This also means that the postcondition must hold trivially when `_k == 0`, or else a postcondition violation will be reported. This is an appropriate design for our desugaring, because greatest lemmas are expected to be used to establish greatest predicates, whose corresponding prefix predicates hold trivially when `_k = 0`. (To prove other predicates, use an ordinary lemma, not a greatest lemma.)

[7]Note, two places where co-predicates and co-lemmas are not analogous are (a) co-predicates must not make recursive calls to their prefix predicates and (b) co-predicates cannot mention `_k`.

It is interesting to compare the intuitive understanding of the coinductive proof in using a greatest lemma with the inductive proof in using a lemma. Whereas the inductive proof is performing proofs for deeper and deeper equalities, the greatest lemma can be understood as producing the infinite proof on demand.

5.14.3.7. Abstemious and voracious functions

Some functions on codatatypes are *abstemious*, meaning that they do not need to unfold a datatype instance very far (perhaps just one destructor call) to prove a relevant property. Knowing this is the case can aid the proofs of properties about the function. The attribute `{:abstemious}` can be applied to a function definition to indicate this.

TODO: Say more about the effect of this attribute and when it should be applied (and likely, correct the paragraph above).

6. Member declarations

Members are the various kinds of methods, the various kinds of functions, mutable fields, and constant fields. These are usually associated with classes, but they also may be declared (with limitations) in traits, newtypes and datatypes (but not in subset types or type synonyms).

6.1. Field Declarations (grammar)

Examples:

```
class C {
  var c: int  // no initialization
  ghost var 123: bv10  // name may be a sequence of digits
  var d: nat, e: real  // type is required
}
```

A field declaration is not permitted in a value type nor as a member of a module (despite there being an implicit unnamed class).

The field name is either an identifier (that is not allowed to start with a leading underscore) or some digits. Digits are used if you want to number your fields, e.g. “0”, “1”, etc. The digits do not denote numbers but sequences of digits, so 0, 00, 0_0 are all different.

A field x of some type T is declared as:

```
var x: T
```

A field declaration declares one or more fields of the enclosing class. Each field is a named part of the state of an object of that class. A field declaration is similar to but distinct from a variable declaration statement. Unlike for local variables and bound variables, the type is required and will not be inferred.

Unlike method and function declarations, a field declaration is not permitted as a member of a module, even though there is an implicit class. Fields can be declared in either an explicit class or a trait. A class that inherits from multiple traits will have all the fields declared in any of its parent traits.

Fields that are declared as `ghost` can only be used in specifications, not in code that will be compiled into executable code.

Fields may not be declared static.

6.2. Constant Field Declarations (grammar)

Examples:

```
const c: int
ghost const d := 5
class A {
  const e: bool
  static const f: int
}
```

A `const` declaration declares a name bound to a value, which value is fixed after initialization.

The declaration must either have a type or an initializing expression (or both). If the type is omitted, it is inferred from the initializing expression.

- A const declaration may include the `ghost`, `static`, and `opaque` modifiers, but no others.
- A const declaration may appear within a module or within any declaration that may contain members (class, trait, datatype, newtype).
- If it is in a module, it is implicitly `static`, and may not also be declared `static`.
- If the declaration has an initializing expression that is a ghost expression, then the ghost-ness of the declaration is inferred; the `ghost` modifier may be omitted.
- If the declaration includes the `opaque` modifier, then uses of the declared variable know its name and type but not its value. The value can be made known for reasoning purposes by using the reveal statement.
- The initialization expression may refer to other constant fields that are in scope and declared either before or after this declaration, but circular references are not allowed.

6.3. Method Declarations (grammar)

Examples:

```
method m(i: int) requires i > 0 {}
method p() returns (r: int) { r := 0; }
method q() returns (r: int, s: int, t: nat) ensures r < s < t { r := 0; s := 1; t := 2; }
ghost method g() {}
class A {
  method f() {}
  constructor Init() {}
  static method g<T>(t: T) {}
}
```

```
lemma L(p: bool) ensures p || !p {}
twostate lemma TL(p: bool) ensures p || !p {}
least lemma LL[nat](p: bool) ensures p || !p {}
greatest lemma GL(p: bool) ensures p || !p {}
abstract module M { method m(i: int) }
module N refines M { method m ... {} }
```

Method declarations include a variety of related types of methods: - method - constructor - lemma - twostate lemma - least lemma - greatest lemma

A method signature specifies the method generic parameters, input parameters and return parameters. The formal parameters are not allowed to have `ghost` specified if `ghost` was already specified for the method. Within the body of a method, formal (input) parameters are immutable, that is, they may not be assigned to, though their array elements or fields may be assigned, if otherwise permitted. The out-parameters are mutable and must be assigned in the body of the method.

An `ellipsis` is used when a method or function is being redeclared in a module that refines another module. (cf. Section 10) In that case the signature is copied from the module that is being refined. This works because Dafny does not support method or function overloading, so the name of the class method uniquely identifies it without the signature.

See Section 7.2 for a description of the method specification.

Here is an example of a method declaration.

```
method {:att1}{:att2} M<T1, T2>(a: A, b: B, c: C)
                                        returns (x: X, y: Y, z: Z)
  requires Pre
  modifies Frame
  ensures Post
  decreases Rank
{
  Body
}
```

where `:att1` and `:att2` are attributes of the method, `T1` and `T2` are type parameters of the method (if generic), `a, b, c` are the method's in-parameters, `x, y, z` are the method's out-parameters, `Pre` is a boolean expression denoting the method's precondition, `Frame` denotes a set of objects whose fields may be updated by the method, `Post` is a boolean expression denoting the method's postcondition, `Rank` is the method's variant function, and `Body` is a list of statements that implements the method. `Frame` can be a list of expressions, each of which is a set of objects or a single object, the latter standing for the singleton set consisting of that one object. The method's frame is the union of these sets, plus the set of objects allocated by the method body. For example, if `c` and `d` are parameters of a class type `C`, then

```
modifies {c, d}
modifies {c} + {d}
modifies c, {d}
modifies c, d
```

all mean the same thing.

If the method is an *extreme lemma* (a `least` or `greatest` lemma), then the method signature may also state the type of the *k* parameter as either `nat` or `ORDINAL`. These are described in Section 12.5.3 and subsequent sections.

6.3.1. Ordinary methods

A method can be declared as ghost by preceding the declaration with the keyword `ghost` and as static by preceding the declaration with the keyword `static`. The default is non-static (i.e., instance) for methods declared in a type and non-ghost. An instance method has an implicit receiver parameter, `this`. A static method M in a class C can be invoked by `C.M(…)`.

An ordinary method is declared with the `method` keyword; the section about constructors explains methods that instead use the `constructor` keyword; the section about lemmas discusses methods that are declared with the `lemma` keyword. Methods declared with the `least lemma` or `greatest lemma` keyword phrases are discussed later in the context of extreme predicates (see the section about greatest lemmas).

A method without a body is *abstract*. A method is allowed to be abstract under the following circumstances:

- It contains an `{:axiom}` attribute
- It contains an `{:extern}` attribute (in this case, to be runnable, the method must have a body in non-Dafny compiled code in the target language.)
- It is a declaration in an abstract module. Note that when there is no body, Dafny assumes that the *ensures* clauses are true without proof.

6.3.2. Constructors

To write structured object-oriented programs, one often relies on objects being constructed only in certain ways. For this purpose, Dafny provides *constructor (method)s*. A constructor is declared with the keyword `constructor` instead of `method`; constructors are permitted only in classes. A constructor is allowed to be declared as `ghost`, in which case it can only be used in ghost contexts.

A constructor can only be called at the time an object is allocated (see object-creation examples below). Moreover, when a class contains a constructor, every call to `new` for a class must be accompanied by a call to one of its constructors. A class may declare no constructors or one or more constructors.

In general, a constructor is responsible for initializating the instance fields of its class. However, any field that is given an initializer in its declaration may not be reassigned in the body of the constructor.

6.3.2.1. Classes with no explicit constructors For a class that declares no constructors, an instance of the class is created with

```
c := new C;
```

This allocates an object and initializes its fields to values of their respective types (and initializes each `const` field with a RHS to its specified value). The RHS of a `const` field may depend on other `const` or `var` fields, but circular dependencies are not allowed.

This simple form of `new` is allowed only if the class declares no constructors, which is not possible to determine in every scope. It is easy to determine whether or not a class declares any constructors if the class is declared in the same module that performs the `new`. If the class is declared in a different module and that module exports a constructor, then it is also clear that the class has a constructor (and thus this simple form of `new` cannot be used). (Note that an export set that `reveals` a class `C` also exports the anonymous constructor of `C`, if any.) But if the module that declares `C` does not export any constructors for `C`, then callers outside the module do not know whether or not `C` has a constructor. Therefore, this simple form of `new` is allowed only for classes that are declared in the same module as the use of `new`.

The simple `new C` is allowed in ghost contexts. Also, unlike the forms of `new` that call a constructor or initialization method, it can be used in a simultaneous assignment; for example

```
c, d, e := new C, new C, 15;
```

is legal.

As a shorthand for writing

```
c := new C;
c.Init(args);
```

where `Init` is an initialization method (see the top of the section about class types), one can write

```
c := new C.Init(args);
```

but it is more typical in such a case to declare a constructor for the class.

(The syntactic support for initialization methods is provided for historical reasons. It may be deprecated in some future version of Dafny. In most cases, a constructor is to be preferred.)

6.3.2.2. Classes with one or more constructors Like other class members, constructors have names. And like other members, their names must be distinct, even if their signatures are different. Being able to name constructors promotes names like `InitFromList` or `InitFromSet` (or just `FromList` and `FromSet`). Unlike other members, one constructor is allowed to be *anonymous*; in other words, an *anonymous constructor* is a constructor whose name is essentially the empty string. For example:

```
class Item {
  constructor I(xy: int) // ...
  constructor (x: int, y: int)
  // ...
}
```

The named constructor is invoked as

```
  i := new Item.I(42);
```

The anonymous constructor is invoked as

```
  m := new Item(45, 29);
```

dropping the "`.`".

6.3.2.3. Two-phase constructors The body of a constructor contains two sections, an initialization phase and a post-initialization phase, separated by a `new;` statement. If there is no `new;` statement, the entire body is the initialization phase. The initialization phase is intended to initialize field variables that were not given values in their declaration; it may not reassign to fields that do have initializers in their declarations. In this phase, uses of the object reference `this` are restricted; a program may use `this`

- as the receiver on the LHS,
- as the entire RHS of an assignment to a field of `this`,
- and as a member of a set on the RHS that is being assigned to a field of `this`.

A `const` field with a RHS is not allowed to be assigned anywhere else. A `const` field without a RHS may be assigned only in constructors, and more precisely only in the initialization phase of constructors. During this phase, a `const` field may be assigned more than once; whatever value the `const` field has at the end of the initialization phase is the value it will have forever thereafter.

For a constructor declared as `ghost`, the initialization phase is allowed to assign both ghost and non-ghost fields. For such an object, values of non-ghost fields at the end of the initialization phase are in effect no longer changeable.

There are no restrictions on expressions or statements in the post-initialization phase.

6.3.3. 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 and the `ghost` keyword cannot be applied to them.

Syntactically, lemmas can be placed where ghost methods can be placed, but they serve a significantly different function. First of all, a lemma is forbidden to have `modifies` clause: it may not change anything about even the ghost state; ghost methods may have `modifies` clauses and may change ghost (but not non-ghost) state. Furthermore, a lemma is not allowed to allocate any new objects. And a lemma may be used in the program text in places where ghost methods may not, such as within expressions (cf. Section 21.1).

Lemmas may, but typically do not, have out-parameters.

In summary, a lemma states a logical fact, summarizing an inference that the verifier cannot do on its own. Explicitly "calling" a lemma in the program text tells the verifier to use that fact at that location with the actual arguments substituted for the formal parameters. The lemma is proved separately for all cases of its formal parameters that satisfy the preconditions of the lemma.

For an example, see the `FibProperty` lemma in Section 12.5.2.

See the Dafny Lemmas tutorial for more examples and hints for using lemmas.

6.3.4. Two-state lemmas and functions

The heap is an implicit parameter to every function, though a function is only allowed to read those parts of the mutable heap that it admits to in its `reads` clause. Sometimes, it is useful for a function to take two heap parameters, for example, so the function can return the difference between the value of a field in the two heaps. Such a *two-state function* is declared by `twostate function` (or `twostate predicate`, which is the same as a `twostate function` that returns a `bool`). A two-state function is always ghost. It is appropriate to think of these two implicit heap parameters as representing a "current" heap and an "old" heap.

For example, the predicate

```
class Cell { var data: int  constructor(i: int) { data := i; } }
twostate predicate Increasing(c: Cell)
  reads c
{
  old(c.data) <= c.data
}
```

returns `true` if the value of `c.data` has not been reduced from the old state to the current. Dereferences in the current heap are written as usual (e.g., `c.data`) and must, as usual, be accounted for in the function's `reads` clause. Dereferences in the old heap are enclosed by `old` (e.g., `old(c.data)`), just like when one dereferences a method's initial heap. The function is allowed to read anything in the old heap; the `reads` clause only declares dependencies on locations in the current heap. Consequently, the frame axiom for a two-state function is sensitive to any change in the old-heap parameter; in other words, the frame axiom says nothing about two invocations of the two-state function with different old-heap parameters.

At a call site, the two-state function's current-heap parameter is always passed in as the caller's current heap. The two-state function's old-heap parameter is by default passed in as the caller's old heap (that is, the initial heap if the caller is a method and the old heap if the caller is a two-state function). While there is never a choice in which heap gets passed as the current heap, the caller can use any preceding heap as the argument to the two-state function's old-heap parameter. This is done by labeling a state in the caller and passing in the label, just like this is done with the built-in `old` function.

For example, the following assertions all hold:

```
method Caller(c: Cell)
  modifies c
{
  c.data := c.data + 10;
  label L:
  assert Increasing(c);
  c.data := c.data - 2;
  assert Increasing(c);
  assert !Increasing@L(c);
}
```

The first call to `Increasing` uses `Caller`'s initial state as the old-heap parameter, and so does the second call. The third call instead uses as the old-heap parameter the heap at label `L`, which is why the third call returns `false`. As shown in the example, an explicitly given old-heap parameter is given after an `@`-sign (which follows the name of the function and any explicitly given type parameters) and before the open parenthesis (after which the ordinary parameters are given).

A two-state function is allowed to be called only from a two-state context, which means a method, a two-state lemma (see below), or another two-state function. Just like a label used with an `old` expression, any label used in a call to a two-state function must denote a program point that *dominates* the call. This means that any control leading to the call must necessarily have passed through the labeled program point.

Any parameter (including the receiver parameter, if any) passed to a two-state function must have been allocated already in the old state. For example, the second call to `Diff` in method `M` is illegal, since `d` was not allocated on entry to `M`:

```
twostate function Diff(c: Cell, d: Cell): int
  reads d
{
  d.data - old(c.data)
}

method M(c: Cell) {
  var d := new Cell(10);
  label L:
  ghost var x := Diff@L(c, d);
  ghost var y := Diff(c, d); // error: d is not allocated in old state
}
```

A two-state function may declare that it only assumes a parameter to be allocated in the current heap. This is done by preceding the parameter with the `new` modifier, as illustrated in the following example, where the first call to `DiffAgain` is legal:

```
twostate function DiffAgain(c: Cell, new d: Cell): int
  reads d
{
  d.data - old(c.data)
}

method P(c: Cell) {
  var d := new Cell(10);
  ghost var x := DiffAgain(c, d);
  ghost var y := DiffAgain(d, c); // error: d is not allocated in old state
}
```

A *two-state lemma* works in an analogous way. It is a lemma with both a current-heap parameter and an old-heap parameter, it can use `old` expressions in its specification (including in the precondition) and body, its parameters may use the `new` modifier, and the old-heap parameter is by default passed in as the caller's old heap, which can be changed by using an `@`-parameter.

Here is an example of something useful that can be done with a two-state lemma:

```
function SeqSum(s: seq<Cell>): int
  reads s
{
  if s == [] then 0 else s[0].data + SeqSum(s[1..])
}

twostate lemma IncSumDiff(s: seq<Cell>)
  requires forall c :: c in s ==> Increasing(c)
  ensures old(SeqSum(s)) <= SeqSum(s)
{
  if s == [] {
  } else {
    calc {
      old(SeqSum(s));
    ==  // def. SeqSum
      old(s[0].data + SeqSum(s[1..]));
    ==  // distribute old
      old(s[0].data) + old(SeqSum(s[1..]));
    <=  { assert Increasing(s[0]); }
      s[0].data + old(SeqSum(s[1..]));
    <=  { IncSumDiff(s[1..]); }
      s[0].data + SeqSum(s[1..]);
    ==  // def. SeqSum
      SeqSum(s);
    }
  }
}
```

A two-state function can be used as a first-class function value, where the receiver (if any), type parameters (if any), and old-heap parameter are determined at the time the first-class value is mentioned. While the receiver and type parameters can be explicitly instantiated in such a use (for example, `p.F<int>` for a two-state instance function `F` that takes one type parameter), there is currently no syntactic support for giving the old-heap parameter explicitly. A caller can work around this restriction by using (fancy-word alert!) eta-expansion, meaning wrapping a lambda expression around the call, as in `x => p.F<int>@L(x)`. The following example illustrates using such an eta-expansion:

```
class P {
  twostate function F<X>(x: X): X
}

method EtaExample(p: P) returns (ghost f: int -> int) {
  label L:
  f := x => p.F<int>@L(x);
}
```

6.4. Function Declarations (grammar)

6.4.1. Functions

Examples:

```
function f(i: int): real { i as real }
function g(): (int, int) { (2,3) }
function h(i: int, k: int): int requires i >= 0 { if i == 0 then 0 else 1 }
```

Functions may be declared as ghost. If so, all the formal parameters and return values are ghost; if it is not a ghost function, then individual parameters may be declared ghost as desired.

See Section 7.3 for a description of the function specification. A Dafny function is a pure mathematical function. It is allowed to read memory that was specified in its `reads` expression but is not allowed to have any side effects.

Here is an example function declaration:

```
function {:att1}{:att2} F<T1, T2>(a: A, b: B, c: C): T
  requires Pre
  reads Frame
  ensures Post
  decreases Rank
{
  Body
}
```

where `:att1` and `:att2` are attributes of the function, if any, `T1` and `T2` are type parameters of the function (if generic), `a, b, c` are the function's parameters, `T` is the type of the function's result, `Pre` is a boolean expression denoting the function's precondition, `Frame` denotes a set of objects whose fields the function body may depend on, `Post` is a boolean expression denoting the function's postcondition, `Rank` is the function's variant function, and `Body` is an expression that defines the function's return value. The precondition allows a function to be partial, that is, the precondition says when the function is defined (and Dafny will verify that every use of the function meets the precondition).

The postcondition is usually not needed, since the body of the function gives the full definition. However, the postcondition can be a convenient place to declare properties of the function that may require an inductive proof to establish, such as when the function is recursive. For example:

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

says that the result of Factorial is always positive, which Dafny verifies inductively from the function body.

Within a postcondition, the result of the function is designated by a call of the function, such as `Factorial(n)` in the example above. Alternatively, a name for the function result can be given in the signature, as in the following rewrite of the example above.

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

Pre v4.0, a function is `ghost` by default, and cannot be called from non-ghost code. To make it non-ghost, replace the keyword `function` with the two keywords "`function method`". From v4.0 on, a function is non-ghost by default. To make it ghost, replace the keyword `function` with the two keywords "`ghost function`". (See the –function-syntax option for a description of the migration path for this change in behavior.}

By default, the body of a function is transparent to its users, but sometimes it is useful to hide it. Functions (including predicates, function-by-methods, two-state functions, and extreme predicates) may be declared opaque using either the `opaque` keyword, or using the `--default-function-opacity` argument. If a function `foo` or `bar` is opaque, then Dafny hides the body of the function, so that it can only be seen within its recursive clique (if any), or if the programmer specifically asks to see it via the statement `reveal foo(), bar();`.

In that case, only the signature and specification of the method is known at its points of use, not its body. The body can be *revealed* for reasoning purposes using the reveal statment.

Like methods, functions can be either *instance* (which they are by default when declared within a type) or *static* (when the function declaration contains the keyword `static` or is declared in a module). An instance function, but not a static function, has an implicit receiver parameter, `this`.
A static function `F` in a class `C` can be invoked by `C.F(…)`. This provides a convenient way to declare a number of helper functions in a separate class.

As for methods, a `...` is used when declaring a function in a module refinement (cf. Section 10). For example, if module `M0` declares function `F`, a module `M1` can be declared to refine `M0` and `M1` can then refine `F`. The refinement function, `M1.F` can have a `...` which means to copy the signature from `M0.F`. A refinement function can furnish a body for a function (if `M0.F` does not provide one). It can also add `ensures` clauses.

If a function definition does not have a body, the program that contains it may still be verified. The function itself has nothing to verify. However, any calls of a body-less function are treated as unverified assumptions by the caller, asserting the preconditions and assuming the postconditions. Because body-less functions are unverified assumptions, Dafny will not compile them and will complain if called by `dafny translate`, `dafny build` or even `dafny run`

6.4.2. Predicates

A function that returns a `bool` result is called a *predicate*. As an alternative syntax, a predicate can be declared by replacing the `function` keyword with the `predicate` keyword and possibly omitting a declaration of the return type (if it is not named).

6.4.3. Function-by-method

A function with a `by method` clause declares a *function-by-method*. A function-by-method gives a way to implement a (deterministic, side-effect free) function by a method (whose body may be nondeterministic and may allocate objects that it modifies). This can be useful if the best implementation uses nondeterminism (for example, because it uses `:|` in a nondeterministic way) in a way that does not affect the result, or if the implementation temporarily makes use of some mutable data structures, or if the implementation is done with a loop. For example, here is the standard definition of the Fibonacci function but with an efficient implementation that uses a loop:

```
function Fib(n: nat): nat {
  if n < 2 then n else Fib(n - 2) + Fib(n - 1)
} by method {
  var x, y := 0, 1;
  for i := 0 to n
    invariant x == Fib(i) && y == Fib(i + 1)
  {
    x, y := y, x + y;
  }
  return x;
}
```

The `by method` clause is allowed only for non-ghost `function` or `predicate` declarations (without `twostate`, `least`, and `greatest`, but possibly with `static`); it inherits the in-parameters, attributes, and `requires` and `decreases` clauses of the function. The method also gets one out-parameter, corresponding to the function's result value (and the name of it, if present). Finally, the method gets an empty `modifies` clause and a postcondition `ensures r == F(args)`, where `r` is the name of the out-parameter and `F(args)` is the function with its arguments. In other words, the method body must compute and return exactly what the function says, and must do so without modifying any previously existing heap state.

The function body of a function-by-method is allowed to be ghost, but the method body must be compilable. In non-ghost contexts, the compiler turns a call of the function-by-method into a call that leads to the method body.

Note, the method body of a function-by-method may contain `print` statements. This means that the run-time evaluation of an expression may have print effects. If `--track-print-effects` is enabled, this use of print in a function context will be disallowed.

6.4.4. Function Transparency

A function is said to be *transparent* in a location if the body of the function is visible at that point. A function is said to be *opaque* at a location if it is not transparent. However the specification of a function is always available.

A function is usually transparent up to some unrolling level (up to 1, or maybe 2 or 3). If its arguments are all literals it is transparent all the way.

The default transparency of a function can be set with the `--default-function-opacity` commandline flag. By default, the `--default-function-opacity` is transparent. The transparency of a function is also affected by whether the function was declared with an `opaque` modifier or `transparent` attribute, the (reveal statement), and whether it was `reveal`ed in an export set.

Inside the module where the function is declared: - If `--default-function-opacity` is set to `transparent` (default), then: - if there is no `opaque` modifier, the function is transparent. - if there is an `opaque` modifier, then the function is opaque. If the function is mentioned in a `reveal` statement, then its body is available starting at that `reveal` statement.

- If `--default-function-opacity` is set to `opaque`, then:
  - if there is no `{:transparent}` attribute, the function is opaque. If the function is mentioned in a `reveal` statement, then the body of the function is available starting at that `reveal` statement.
  - if there is a `{:transparent}` attribute, then the function is transparent.
- If `--default-function-opacity` is set to `autoRevealDependencies`, then:
  - if there is no `{:transparent}` attribute, the function is opaque. However, the body of the function is available inside any callable that depends on this function via an implicitly inserted `reveal` statement, unless the callable has the `{autoRevealDependencies k}` attribute for some natural number `k` which is too low.
  - if there is a `{:transparent}` attribute, then the function is transparent.

Outside the module where the function is declared, the function is visible only if it was listed in the export set by which the contents of its module was imported. In that case, if the function was exported with `reveals`, the rules are the same within the importing module as when the function is used inside its declaring module. If the function is exported only with `provides` it is always opaque and is not permitted to be used in a reveal statement.

More information about the Boogie implementation of opaquenes is here.

6.4.5. Extreme (Least or Greatest) Predicates and Lemmas

See Section 12.5.3 for descriptions of extreme predicates and lemmas.

6.4.6. `older` parameters in predicates

A parameter of any predicate (more precisely, of any boolean-returning, non-extreme function) can be marked as `older`. This specifies that the truth of the predicate implies that the allocatedness of the parameter follows from the allocatedness of the non-`older` parameters.

To understand what this means and why this attribute is useful, consider the following example, which specifies reachability between nodes in a directed graph. A `Node` is declared to have any number of children:

```
class Node {
  var children: seq<Node>
}
```

There are several ways one could specify reachability between nodes. One way (which is used in `Test/dafny1/SchorrWaite.dfy` in the Dafny test suite) is to define a type `Path`, representing lists of `Node`s, and to define a predicate that checks if a given list of `Node`s is indeed a path between two given nodes:

```
datatype Path = Empty | Extend(Path, Node)

predicate ReachableVia(source: Node, p: Path, sink: Node, S: set<Node>)
  reads S
  decreases p
{
  match p
  case Empty =>
    source == sink
  case Extend(prefix, n) =>
    n in S && sink in n.children && ReachableVia(source, prefix, n, S)
}
```

In a nutshell, the definition of `ReachableVia` says

- An empty path lets `source` reach `sink` just when `source` and `sink` are the same node.
- A path `Extend(prefix, n)` lets `source` reach `sink` just when the path `prefix` lets `source` reach `n` and `sink` is one of the children nodes of `n`.

To be admissible by Dafny, the recursive predicate must be shown to terminate. Termination is assured by the specification `decreases p`, since every such datatype value has a finite structure and every recursive call passes in a path that is structurally included in the previous. Predicate `ReachableVia` must also declare (an upper bound on) which heap objects it depends on. For this purpose, the predicate takes an additional parameter `S`, which is used to limit the set of intermediate nodes in the path. More precisely, predicate `ReachableVia(source, p, sink, S)` returns `true` if and only if `p` is a list of nodes in `S` and `source` can reach `sink` via `p`.

Using predicate `ReachableVia`, we can now define reachability in `S`:

```
predicate Reachable(source: Node, sink: Node, S: set<Node>)
  reads S
{
  exists p :: ReachableVia(source, p, sink, S)
}
```

This looks like a good definition of reachability, but Dafny won't admit it. The reason is twofold:

- Quantifiers and comprehensions are allowed to range only over allocated state. Ater all, Dafny is a type-safe language where every object reference is *valid* (that is, a pointer to allocated storage of the right type)—it should not be possible, not even through a bound variable in a quantifier or comprehension, for a program to obtain an object reference that isn't valid.
- This property is ensured by disallowing *open-ended* quantifiers. More precisely, the object references that a quantifier may range over must be shown to be confined to object references that were allocated before some of the non-`older` parameters passed to the predicate. Quantifiers that are not open-ended are called *close-ended*. Note that close-ended refers only to the object references that the quantification or comprehension ranges over—it does not say anything about values of other types, like integers.

Often, it is easy to show that a quantifier is close-ended. In fact, if the type of a bound variable does not contain any object references, then the quantifier is trivially close-ended. For example,

```
forall x: int :: x <= Square(x)
```

is trivially close-ended.

Another innocent-looking quantifier occurs in the following example:

```
predicate IsCommutative<X>(r: (X, X) -> bool)
{
  forall x, y :: r(x, y) == r(y, x) // error: open-ended quantifier
}
```

Since nothing is known about type `X`, this quantifier might be open-ended. For example, if `X` were passed in as a class type, then the quantifier would be open-ended. One way to fix this predicate is to restrict it to non-heap based types, which is indicated with the `(!new)` type characteristic (see Section 5.3.1.4):

```
ghost predicate IsCommutative<X(!new)>(r: (X, X) -> bool) // X is restricted to non-heap types
{
  forall x, y :: r(x, y) == r(y, x) // allowed
}
```

Another way to make `IsCommutative` close-ended is to constrain the values of the bound variables `x` and `y`. This can be done by adding a parameter to the predicate and limiting the quantified values to ones in the given set:

```
predicate IsCommutativeInS<X>(r: (X, X) -> bool, S: set<X>)
{
  forall x, y :: x in S && y in S ==> r(x, y) == r(y, x) // close-ended
}
```

Through a simple syntactic analysis, Dafny detects the antecedents `x in S` and `y in S`, and since `S` is a parameter and thus can only be passed in as something that the caller has already allocated, the quantifier in `IsCommutativeInS` is determined to be close-ended.

Note, the `x in S` trick does not work for the motivating example, `Reachable`. If you try to write

```
predicate Reachable(source: Node, sink: Node, S: set<Node>)
  reads S
{
  exists p :: p in S && ReachableVia(source, p, sink, S) // type error: p
}
```

you will get a type error, because `p in S` does not make sense if `p` has type `Path`. We need some other way to justify that the quantification in `Reachable` is close-ended.

Dafny offers a way to extend the `x in S` trick to more situations. This is where the `older` modifier comes in. Before we apply `older` in the `Reachable` example, let's first look at what `older` does in a less cluttered example.

Suppose we rewrite `IsCommutativeInS` using a programmer-defined predicate `In`:

```
predicate In<X>(x: X, S: set<X>) {
  x in S
}

predicate IsCommutativeInS<X>(r: (X, X) -> bool, S: set<X>)
{
  forall x, y :: In(x, S) && In(y, S) ==> r(x, y) == r(y, x) // error: open-ended?
}
```

The simple syntactic analysis that looks for `x in S` finds nothing here, because the `in` operator is relegated to the body of predicate `In`. To inform the analysis that `In` is a predicate that, in effect, is like `in`, you can mark parameter `x` with `older`:

```
predicate In<X>(older x: X, S: set<X>) {
  x in S
}
```

This causes the simple syntactic analysis to accept the quantifier in `IsCommutativeInS`. Adding `older` also imposes a semantic check on the body of predicate `In`, enforced by the verifier. The semantic check is that all the object references in the value `x` are older (or equally old as) the object references that are part of the other parameters, *in the event that the predicate returns true*. That is, `older` is designed to help the caller only if the predicate returns `true`, and the semantic check amounts to nothing if the predicate returns `false`.

Finally, let's get back to the motivating example. To allow the quantifier in `Reachable`, mark parameter `p` of `ReachableVia` with `older`:

```
class Node {
  var children: seq<Node>
}

datatype Path = Empty | Extend(Path, Node)

ghost predicate Reachable(source: Node, sink: Node, S: set<Node>)
  reads S
{
  exists p :: ReachableVia(source, p, sink, S) // allowed because of 'older p' on ReachableVia
}

ghost predicate ReachableVia(source: Node, older p: Path, sink: Node, S: set<Node>)
  reads S
  decreases p
{
  match p
  case Empty =>
    source == sink
  case Extend(prefix, n) =>
    n in S && sink in n.children && ReachableVia(source, prefix, n, S)
}
```

This example is more involved than the simpler `In` example above. Because of the `older` modifier on the parameter, the quantifier in `Reachable` is allowed. For intuition, you can think of the effect of `older p` as adding an antecedent `p in {source} + {sink} + S` (but, as we have seen, this is not type correct). The semantic check imposed on the body of `ReachableVia` makes sure that, if the predicate returns `true`, then every object reference in `p` is as old as some object reference in another parameter to the predicate.

6.5. Nameonly Formal Parameters and Default-Value Expressions

A formal parameter of a method, constructor in a class, iterator, function, or datatype constructor can be declared with an expression denoting a *default value*. This makes the parameter *optional*, as opposed to *required*.

For example,

```
function f(x: int, y: int := 10): int
```

may be called as either

```
const i := f(1, 2)
const j := f(1)
```

where `f(1)` is equivalent to `f(1, 10)` in this case.

The above function may also be called as

```
var k := f(y := 10, x := 2);
```

using names; actual arguments with names may be given in any order, though they must be after actual arguments without names.

Formal parameters may also be declared `nameonly`, in which case a call site must always explicitly name the formal when providing its actual argument.

For example, a function `ff` declared as

```
function ff(x: int, nameonly y: int): int
```

must be called either by listing the value for x and then y with a name, as in `ff(0, y := 4)` or by giving both actuals by name (in any order). A `nameonly` formal may also have a default value and thus be optional.

Any formals after a `nameonly` formal must either be `nameonly` themselves or have default values.

The formals of datatype constructors are not required to have names. A nameless formal may not have a default value, nor may it follow a formal that has a default value.

The default-value expression for a parameter is allowed to mention the other parameters, including `this` (for instance methods and instance functions), but not the implicit `_k` parameter in least and greatest predicates and lemmas. The default value of a parameter may mention both preceding and subsequent parameters, but there may not be any dependent cycle between the parameters and their default-value expressions.

The well-formedness of default-value expressions is checked independent of the precondition of the enclosing declaration. For a function, the parameter default-value expressions may only read what the function's `reads` clause allows. For a datatype constructor, parameter default-value expressions may not read anything. A default-value expression may not be involved in any recursive or mutually recursive calls with the enclosing declaration.

7. Specifications

Specifications describe logical properties of Dafny methods, functions, lambdas, iterators and loops. They specify preconditions, postconditions, invariants, what memory locations may be read or modified, and termination information by means of *specification clauses*. For each kind of specification, zero or more specification clauses (of the type accepted for that type of specification) may be given, in any order.

We document specifications at these levels:

- At the lowest level are the various kinds of specification clauses, e.g., a `RequiresClause`.
- Next are the specifications for entities that need them, e.g., a `MethodSpec`, which typically consist of a sequence of specification clauses.
- At the top level are the entity declarations that include the specifications, e.g., `MethodDecl`.

This section documents the first two of these in a bottom-up manner. We first document the clauses and then the specifications that use them.

Specification clauses typically appear in a sequence. They all begin with a keyword and do not end with semicolons.

7.1. Specification Clauses

Within expressions in specification clauses, you can use specification expressions along with any other expressions you need.

7.1.1. Requires Clause (grammar)

Examples:

```
method m(i: int)
  requires true
  requires i > 0
  requires L: 0 < i < 10
```

The `requires` clauses specify preconditions for methods, functions, lambda expressions and iterators. Dafny checks that the preconditions are met at all call sites. The callee may then assume the preconditions hold on entry.

If no `requires` clause is specified, then a default implicit clause `requires true` is used.

If more than one `requires` clause is given, then the precondition is the conjunction of all of the expressions from all of the `requires` clauses, with a collected list of all the given Attributes. The order of conjunctions (and hence the order of `requires` clauses with respect to each other) can be important: earlier conjuncts can set conditions that establish that later conjuncts are well-defined.

The attributes recognized for requires clauses are discussed in Section 11.4.

A requires clause can have custom error and success messages.

7.1.2. Ensures Clause (grammar)

Examples:

```
method {:axiom} m(i: int) returns (r: int)
  ensures r > 0
```

An `ensures` clause specifies the post condition for a method, function or iterator.

If no `ensures` clause is specified, then a default implicit clause `ensures true` is used.

If more than one `ensures` clause is given, then the postcondition is the conjunction of all of the expressions from all of the `ensures` clauses, with a collected list of all the given Attributes. The order of conjunctions (and hence the order of `ensures` clauses with respect to each other) can be important: earlier conjuncts can set conditions that establish that later conjuncts are well-defined.

The attributes recognized for ensures clauses are discussed in Section 11.4.

An ensures clause can have custom error and success messages.

7.1.3. Decreases Clause (grammar)

Examples:

```
method m(i: int, j: int) returns (r: int)
  decreases i, j
method n(i: int) returns (r: int)
  decreases *
```

Decreases clauses are used to prove termination in the presence of recursion. If more than one `decreases` clause is given it is as if a single `decreases` clause had been given with the collected list of arguments and a collected list of Attributes. That is,

```
decreases A, B
decreases C, D
```

is equivalent to

```
decreases A, B, C, D
```

Note that changing the order of multiple `decreases` clauses will change the order of the expressions within the equivalent single `decreases` clause, and will therefore have different semantics.

Loops and compiled methods (but not functions and not ghost methods, including lemmas) can be specified to be possibly non-terminating. This is done by declaring the method or loop with `decreases *`, which causes the proof of termination to be skipped. If a `*` is present in a `decreases` clause, no other expressions are allowed in the `decreases` clause. A method that contains a possibly non-terminating loop or a call to a possibly non-terminating method must itself be declared as possibly non-terminating.

Termination metrics in Dafny, which are declared by `decreases` clauses, are lexicographic tuples of expressions. At each recursive (or mutually recursive) call to a function or method, Dafny checks that the effective `decreases` clause of the callee is strictly smaller than the effective `decreases` clause of the caller.

What does "strictly smaller" mean? Dafny provides a built-in well-founded order for every type and, in some cases, between types. For example, the Boolean `false` is strictly smaller than `true`, the integer `78` is strictly smaller than `102`, the set `{2,5}` is strictly smaller than (because it is a proper subset of) the set `{2,3,5}`, and for `s` of type `seq<Color>` where `Color` is some inductive datatype, the color `s[0]` is strictly less than `s` (provided `s` is nonempty).

What does "effective decreases clause" mean? Dafny always appends a "top" element to the lexicographic tuple given by the user. This top element cannot be syntactically denoted in a Dafny program and it never occurs as a run-time value either. Rather, it is a fictitious value, which here we will denote $\top$, such that each value that can ever occur in a Dafny program is strictly less than $\top$. Dafny sometimes also prepends expressions to the lexicographic tuple given by the user. The effective decreases clause is any such prefix, followed by the user-provided decreases clause, followed by $\top$. We said "user-provided decreases clause", but if the user completely omits a `decreases` clause, then Dafny will usually make a guess at one, in which case the effective decreases clause is any prefix followed by the guess followed by $\top$.

Here is a simple but interesting example: the Fibonacci function.

```
function Fib(n: nat) : nat
{
  if n < 2 then n else Fib(n-2) + Fib(n-1)
}
```

In this example, Dafny supplies a `decreases n` clause.

Let's take a look at the kind of example where a mysterious-looking decreases clause like "Rank, 0" is useful.

Consider two mutually recursive methods, `A` and `B`:

```
method A(x: nat)
{
  B(x);
}

method B(x: nat)
{
  if x != 0 { A(x-1); }
}
```

To prove termination of `A` and `B`, Dafny needs to have effective decreases clauses for A and B such that:

- the measure for the callee `B(x)` is strictly smaller than the measure for the caller `A(x)`, and
- the measure for the callee `A(x-1)` is strictly smaller than the measure for the caller `B(x)`.

Satisfying the second of these conditions is easy, but what about the first? Note, for example, that declaring both `A` and `B` with "decreases x" does not work, because that won't prove a strict decrease for the call from `A(x)` to `B(x)`.

Here's one possibility:

```
method A(x: nat)
  decreases x, 1
{
  B(x);
}

method B(x: nat)
  decreases x, 0
{
  if x != 0 { A(x-1); }
}
```

For the call from `A(x)` to `B(x)`, the lexicographic tuple `"x, 0"` is strictly smaller than `"x, 1"`, and for the call from `B(x)` to `A(x-1)`, the lexicographic tuple `"x-1, 1"` is strictly smaller than `"x, 0"`.

Two things to note: First, the choice of "0" and "1" as the second components of these lexicographic tuples is rather arbitrary. It could just as well have been "false" and "true", respectively, or the sets `{2,5}` and `{2,3,5}`. Second, the keyword `decreases` often gives rise to an intuitive English reading of the declaration. For example, you might say that the recursive calls in the definition of the familiar Fibonacci function `Fib(n)` "decreases n". But when the lexicographic tuple contains constants, the English reading of the declaration becomes mysterious and may give rise to questions like "how can you decrease the constant 0?". The keyword is just that—a keyword. It says "here comes a list of expressions that make up the lexicographic tuple we want to use for the termination measure". What is important is that one effective decreases clause is compared against another one, and it certainly makes sense to compare something to a constant (and to compare one constant to another).

We can simplify things a little bit by remembering that Dafny appends $\top$ to the user-supplied decreases clause. For the A-and-B example, this lets us drop the constant from the `decreases` clause of A:

```
method A(x: nat)
   decreases x
{
  B(x);
}

method B(x: nat)
  decreases x, 0
{
  if x != 0 { A(x-1); }
}
```

The effective decreases clause of `A` is $(x, \top)$ and the effective decreases clause of `B` is $(x, 0, \top)$. These tuples still satisfy the two conditions $(x, 0, \top) < (x, \top)$ and $(x - 1, \top) < (x, 0, \top)$. And as before, the constant "0" is arbitrary; anything less than $\top$ (which is any Dafny expression) would work.

Let's take a look at one more example that better illustrates the utility of $\top$. Consider again two mutually recursive methods, call them `Outer` and `Inner`, representing the recursive counterparts of what iteratively might be two nested loops:

```
method Outer(x: nat)
{
  // set y to an arbitrary non-negative integer
  var y :| 0 <= y;
  Inner(x, y);
}

method Inner(x: nat, y: nat)
{
  if y != 0 {
    Inner(x, y-1);
  } else if x != 0 {
    Outer(x-1);
  }
}
```

The body of `Outer` uses an assign-such-that statement to represent some computation that takes place before `Inner` is called. It sets "y" to some arbitrary non-negative value. In a more concrete example, `Inner` would do some work for each "y" and then continue as `Outer` on the next smaller "x".

Using a `decreases` clause (x, y) for `Inner` seems natural, but if we don't have any bound on the size of the y computed by `Outer`, there is no expression we can write in the `decreases` clause of `Outer` that is sure to lead to a strictly smaller value for y when `Inner` is called. $\top$ to the rescue. If we arrange for the effective decreases clause of `Outer` to be $(x, \top)$ and the effective decreases clause for `Inner` to be $(x, y, \top)$, then we can show the strict decreases as required. Since $\top$ is implicitly appended, the two decreases clauses declared in the program text can be:

```
method Outer(x: nat)
  decreases x
{
  // set y to an arbitrary non-negative integer
  var y :| 0 <= y;
  Inner(x, y);
}

method Inner(x: nat, y: nat)
  decreases x,y
{
  if y != 0 {
    Inner(x, y-1);
  } else if x != 0 {
    Outer(x-1);
  }
}
```

Moreover, remember that if a function or method has no user-declared `decreases` clause, Dafny will make a guess. The guess is (usually) the list of arguments of the function/method, in the order given. This is exactly the decreases clauses needed here. Thus, Dafny successfully verifies the program without any explicit `decreases` clauses:

```
method Outer(x: nat)
{
  var y :| 0 <= y;
  Inner(x, y);
}

method Inner(x: nat, y: nat)
{
  if y != 0 {
    Inner(x, y-1);
  } else if x != 0 {
    Outer(x-1);
  }
}
```

The ingredients are simple, but the end result may seem like magic. For many users, however, there may be no magic at all – the end result may be so natural that the user never even has to be bothered to think about that there was a need to prove termination in the first place.

Dafny also prepends two expressions to the user-specified (or guessed) tuple of expressions in the decreases clause. The first expression is the ordering of the module containing the decreases clause in the dependence-ordering of modules. That is, a module that neither imports or defines (as submodules) any other modules has the lowest value in the order and every other module has a value that is higher than that of any module it defines or imports. As a module cannot call a method in a module that it does not depend on, this is an effective first component to the overall decreases tuple.

The second prepended expression represents the position of the method in the call graph within a module. Dafny analyzes the call-graph of the module, grouping all methods into mutually-recursive groups. Any method that calls nothing else is at the lowest level (say level 0). Absent recursion, every method has a level value strictly greater than any method it calls. Methods that are mutually recursive are at the same level and they are above the level of anything else they call. With this level value prepended to the decreases clause, the decreases tuple automatically decreases on any calls in a non-recursive context.

Though Dafny fixes a well-founded order that it uses when checking termination, Dafny does not normally surface this ordering directly in expressions. However, it is possible to write such ordering constraints using `decreases to` expressions.

7.1.4. Framing (grammar)

Examples:

```
*
o
o`a
`a
{ o, p, q }
{}
```

Frame expressions are used to denote the set of memory locations that a Dafny program element may read or write. They are used in `reads` and `modifies` clauses. A frame expression is a set expression. The form `{}` is the empty set. The type of the frame expression is `set<object>`.

Note that framing only applies to the heap, or memory accessed through references. Local variables are not stored on the heap, so they cannot be mentioned (well, they are not in scope in the declaration) in frame annotations. Note also that types like sets, sequences, and multisets are value types, and are treated like integers or local variables. Arrays and objects are reference types, and they are stored on the heap (though as always there is a subtle distinction between the reference itself and the value it points to.)

The `FrameField` construct is used to specify a field of a class object. The identifier following the back-quote is the name of the field being referenced. If the `FrameField` is preceded by an expression the expression must be a reference to an object having that field. If the `FrameField` is not preceded by an expression then the frame expression is referring to that field of the current object (`this`). This form is only used within a method of a class or trait.

A `FrameField` can be useful in the following case: When a method modifies only one field, rather than writing

```
class A {
  var i: int
  var x0: int
  var x1: int
  var x2: int
  var x3: int
  var x4: int
  method M()
    modifies this
    ensures unchanged(`x0) && unchanged(`x1) && unchanged(`x2) && unchanged(`x3) && unchanged(`x4)
  { i := i + 1; }
}
```

one can write the more concise:

```
class A {
  var i: int
  var x0: int
  var x1: int
  var x2: int
  var x3: int
  var x4: int
  method M()
    modifies `i
  { i := i + 1; }
}
```

There's (unfortunately) no form of it for array elements – but to account for unchanged elements, you can always write `forall i | 0 <= i < |a| :: unchanged(a[i])`.

A `FrameField` is not taken into consideration for lambda expressions.

7.1.5. Reads Clause (grammar)

Examples:

```
const o: object
const o, oo: object
function f()
  reads *
function g()
  reads o, oo
function h()
  reads { o }
method f()
  reads *
method g()
  reads o, oo
method h()
  reads { o }
```

Functions are not allowed to have side effects; they may also be restricted in what they can read. The *reading frame* of a function (or predicate) consists of all the heap memory locations that the function is allowed to read. The reason we might limit what a function can read is so that when we write to memory, we can be sure that functions that did not read that part of memory have the same value they did before. For example, we might have two arrays, one of which we know is sorted. If we did not put a reads annotation on the sorted predicate, then when we modify the unsorted array, we cannot determine whether the other array stopped being sorted. While we might be able to give invariants to preserve it in this case, it gets even more complex when manipulating data structures. In this case, framing is essential to making the verification process feasible.

By default, methods are not required to list the memory location they read. However, there are use cases for restricting what methods can read as well. In particular, if you want to verify that imperative code is safe to execute concurrently when compiled, you can specify that a method does not read or write any shared state, and therefore cannot encounter race conditions or runtime crashes related to unsafe communication between concurrent executions. See the `{:concurrent}` attribute for more details.

It is not just the body of a function or method that is subject to `reads` checks, but also its precondition and the `reads` clause itself.

A `reads` clause can list a wildcard `*`, which allows the enclosing function or method to read anything. This is the implicit default for methods with no `reads` clauses, allowing methods to read whatever they like. The default for functions, however, is to not allow reading any memory. Allowing functions to read arbitrary memory is more problematic: in many cases, and in particular in all cases where the function is defined recursively, this makes it next to impossible to make any use of the function. Nevertheless, as an experimental feature, the language allows it (and it is sound). If a `reads` clause uses `*`, then the `reads` clause is not allowed to mention anything else (since anything else would be irrelevant, anyhow).

A `reads` clause specifies the set of memory locations that a function, lambda, or method may read. The readable memory locations are all the fields of all of the references given in the set specified in the frame expression and the single fields given in `FrameField` elements of the frame expression. For example, in

```
class C {
  var x: int
  var y: int

  predicate f(c: C)
    reads this, c`x
  {
    this.x == c.x
  }
}
```

the `reads` clause allows reading `this.x`, `this,y`, and `c.x` (which may be the same memory location as `this.x`). }

If more than one `reads` clause is given in a specification the effective read set is the union of the sets specified. If there are no `reads` clauses the effective read set is empty. If `*` is given in a `reads` clause it means any memory may be read.

If a `reads` clause refers to a sequence or multiset, that collection (call it `c`) is converted to a set by adding an implicit set comprehension of the form `set o: object | o in c` before computing the union of object sets from other `reads` clauses.

An expression in a `reads` clause is also allowed to be a function call whose value is a collection of references. Such an expression is converted to a set by taking the union of the function's image over all inputs. For example, if `F` is a function from `int` to `set<object>`, then `reads F` has the meaning

```
set x: int, o: object | o in F(x) :: o
```

For each function value `f`, Dafny defines the function `f.reads`, which takes the same arguments as `f` and returns that set of objects that `f` reads (according to its reads clause) with those arguments. `f.reads` has type `T ~> set<object>`, where `T` is the input type(s) of `f`.

This is particularly useful when wanting to specify the reads set of another function. For example, function `Sum` adds up the values of `f(i)` where `i` ranges from `lo` to `hi`:

```
function Sum(f: int ~> real, lo: int, hi: int): real
  requires lo <= hi
  requires forall i :: f.requires(i)
  reads f.reads
  decreases hi - lo
{
  if lo == hi then 0.0 else
    f(lo) + Sum(f, lo + 1, hi)
}
```

Its `reads` specification says that `Sum(f, lo, hi)` may read anything that `f` may read on any input. (The specification `reads f.reads` gives an overapproximation of what `Sum` will actually read. More precise would be to specify that `Sum` reads only what `f` reads on the values from `lo` to `hi`, but the larger set denoted by `reads f.reads` is easier to write down and is often good enough.)

Without such `reads` function, one could also write the more precise and more verbose:

```
function Sum(f: int ~> real, lo: int, hi: int): real
  requires lo <= hi
  requires forall i :: lo <= i < hi ==> f.requires(i)
  reads set i, o | lo <= i < hi && o in f.reads(i) :: o
  decreases hi - lo
{
  if lo == hi then 0.0 else
    f(lo) + Sum(f, lo + 1, hi)
}
```

Note, only `reads` clauses, not `modifies` clauses, are allowed to include functions as just described.

Iterator specifications also allow `reads` clauses, with the same syntax and interpretation of arguments as above, but the meaning is quite different! See Section 5.11 for more details.

7.1.6. Modifies Clause (grammar)

Examples:

```
class A { var f: int }
const o: object?
const p: A?
```

```
method M()
  modifies { o, p }
method N()
  modifies { }
method Q()
  modifies o, p`f
```

By default, methods are allowed to read whatever memory they like, but they are required to list which parts of memory they modify, with a `modifies` annotation. These are almost identical to their `reads` cousins, except they say what can be changed, rather than what the definition depends on. In combination with reads, modification restrictions allow Dafny to prove properties of code that would otherwise be very difficult or impossible. Reads and modifies are one of the tools that allow Dafny to work on one method at a time, because they restrict what would otherwise be arbitrary modifications of memory to something that Dafny can reason about.

Just as for a `reads` clause, the memory locations allowed to be modified in a method are all the fields of any object reference in the frame expression set and any specific field denoted by a `FrameField` in the `modifies` clause. For example, in

```
class C {
  var next: C?
  var value: int

  method M()
    modifies next
  {
    ...
  }
}
```

method `M` is permitted to modify `this.next.next` and `this.next.value` but not `this.next`. To be allowed to modify `this.next`, the modifies clause must include `this`, or some expression that evaluates to `this`, or `this`next`.

If an object is newly allocated within the body of a method or within the scope of a `modifies` statement or a loop's `modifies` clause, then the fields of that object may always be modified.

A `modifies` clause specifies the set of memory locations that a method, iterator or loop body may modify. If more than one `modifies` clause is given in a specification, the effective modifies set is the union of the sets specified. If no `modifies` clause is given the effective modifies set is empty. There is no wildcard (`*`) allowed in a modifies clause. A loop can also have a `modifies` clause. If none is given, the loop may modify anything the enclosing context is allowed to modify.

Note that *modifies* here is used in the sense of *writes*. That is, a field that may not be modified may not be written to, even with the same value it already has or even if the value is restored later. The terminology and semantics varies among specification languages. Some define frame conditions in this sense (a) of *writes* and others in the sense (b) that allows writing a field with the same value or changing the value so long as the original value is restored by the end of the scope. For example, JML defines `assignable` and `modifies` as synonyms in the sense (a), though KeY interprets JML's `assigns/modifies` in sense (b). ACSL and ACSL++ use the `assigns` keyword, but with *modify* (b) semantics. Ada/SPARK's dataflow contracts encode *write* (a) semantics.

7.1.7. Invariant Clause (grammar)

Examples:

```
method m()
{
  var i := 10;
  while 0 < i
    invariant 0 <= i < 10
}
```

An `invariant` clause is used to specify an invariant for a loop. If more than one `invariant` clause is given for a loop, the effective invariant is the conjunction of the conditions specified, in the order given in the source text.

The invariant must hold on entry to the loop. And assuming it is valid on entry to a particular iteration of the loop, Dafny must be able to prove that it then holds at the end of that iteration of the loop.

An invariant can have custom error and success messages.

7.2. Method Specification (grammar)

Examples:

```
class C {
  var next: C?
  var value: int

  method M(i: int) returns (r: int)
    requires i >= 0
    modifies next
    decreases i
    ensures r >= 0
  {
    ...
  }
}
```

A method specification consists of zero or more `reads`, `modifies`, `requires`, `ensures` or `decreases` clauses, in any order. A method does not need `reads` clauses in most cases, because methods are allowed to read any memory by default, but `reads` clauses are supported for use cases such as verifying safe concurrent execution. See the `{:concurrent}` attribute for more details.

7.3. Function Specification (grammar)

Examples:

```
class C {
  var next: C?
  var value: int

  function M(i: int): (r: int)
    requires i >= 0
    reads this
    decreases i
    ensures r >= 0
  {
    0
  }
}
```

A function specification is zero or more `reads`, `requires`, `ensures` or `decreases` clauses, in any order. A function specification does not have `modifies` clauses because functions are not allowed to modify any memory.

7.4. Lambda Specification (grammar)

A lambda specification provides a specification for a lambda function expression; it consists of zero or more `reads` or `requires` clauses. Any `requires` clauses may not have labels or attributes. Lambda specifications do not have `ensures` clauses because the body is never opaque. Lambda specifications do not have `decreases` clauses because lambda expressions do not have names and thus cannot be recursive. A lambda specification does not have `modifies` clauses because lambdas are not allowed to modify any memory.

7.5. Iterator Specification (grammar)

An iterator specification may contains `reads`, `modifies`, `decreases`, `requires`, `yield requires,`ensures`and`yield ensures‘ clauses.

An iterator specification applies both to the iterator's constructor method and to its `MoveNext` method. - The `reads` and `modifies` clauses apply to both of them (but `reads` clauses have a different meaning on iterators than on functions or methods). - The `requires` and `ensures` clauses apply to the constructor. - The `yield requires` and `yield ensures` clauses apply to the `MoveNext` method.

Examples of iterators, including iterator specifications, are given in Section 5.11. Briefly - a requires clause gives a precondition for creating an iterator - an ensures clause gives a postcondition when the iterator exits (after all iterations are complete) - a decreases clause is used to show that the iterator will eventually terminate - a yield requires clause is a precondition for calling `MoveNext` - a yield ensures clause is a postcondition for calling `MoveNext` - a reads clause gives a set of memory locations that will be unchanged after a `yield` statement - a modifies clause gives a set of memory locations the iterator may write to

7.6. Loop Specification (grammar)

A loop specification provides the information Dafny needs to prove properties of a loop. It contains `invariant`, `decreases`, and `modifies` clauses.

The `invariant` clause is effectively a precondition and it along with the negation of the loop test condition provides the postcondition. The `decreases` clause is used to prove termination.

7.7. Auto-generated boilerplate specifications

AutoContracts is an experimental feature that inserts much of the dynamic-frames boilerplate into a class. The user simply - marks the class with `{:autocontracts}` and - declares a function (or predicate) called Valid().

AutoContracts then

- Declares, unless there already exist members with these names:

```
ghost var Repr: set(object)
predicate Valid()
```

- For function/predicate `Valid()`, inserts

```
reads this, Repr
ensures Valid() ==> this in Repr
```

- Into body of `Valid()`, inserts (at the beginning of the body)

```
this in Repr && null !in Repr
```

and also inserts, for every array-valued field `A` declared in the class:

```
 (A != null ==> A in Repr) &&
```

and for every field `F` of a class type `T` where `T` has a field called `Repr`, also inserts

```
 (F != null ==> F in Repr && F.Repr SUBSET Repr && this !in Repr && F.Valid())
```

except, if `A` or `F` is declared with `{:autocontracts false}`, then the implication will not be added. - For every constructor, inserts

```
ensures Valid() && fresh(Repr)
```

- At the end of the body of the constructor, adds

```
 Repr := {this};
 if (A != null) { Repr := Repr + {A}; }
 if (F != null) { Repr := Repr + {F} + F.Repr; }
```

In all the following cases, no `modifies` clause or `reads` clause is added if the user has given one.

- For every non-static non-ghost method that is not a “simple query method”, inserts

```
requires Valid()
modifies Repr
ensures Valid() && fresh(Repr - old(Repr))
```

- At the end of the body of the method, inserts

```
if (A != null && !(A in Repr)) { Repr := Repr + {A}; }
if (F != null && !(F in Repr && F.Repr SUBSET Repr)) { Repr := Repr + {F} + F.Repr; }
```

- For every non-static non-twostate method that is either ghost or is a “simple query method”, add:

```
requires Valid()
```

- For every non-static twostate method, inserts

```
requires old(Valid())
```

- For every non-“Valid” non-static function, inserts

```
requires Valid()
reads Repr
```

7.8. Well-formedness of specifications

Dafny ensures that the `requires` clauses and `ensures` clauses, which are expressions, are well-formed independent of the body they belong to. Examples of conditions this rules out are null pointer dereferencing, out-of-bounds array access, and division by zero. Hence, when declaring the following method:

```
method Test(a: array<int>) returns (j: int)
  requires a.Length >= 1
  ensures a.Length % 2 == 0 ==> j >= 10 / a.Length
{
  j := 20;
  var divisor := a.Length;
  if divisor % 2 == 0 {
    j := j / divisor;
  }
}
```

Dafny will split the verification in two assertion batches that will roughly look like the following lemmas:

```
lemma Test_WellFormed(a: array?<int>)
{
  assume a != null;       // From the definition of a
  assert a != null;       // for the `requires a.Length >= 1`
  assume a.Length >= 1;   // After well-formedness, we assume the requires
  assert a != null;       // Again for the `a.Length % 2`
  if a.Length % 2 == 0 {
    assert a != null;     // Again for the final `a.Length`
    assert a.Length != 0; // Because of the 10 / a.Length
  }
}

method Test_Correctness(a: array?<int>)
{ // Here we assume the well-formedness of the condition
  assume a != null;       // for the `requires a.Length >= 1`
  assume a != null;       // Again for the `a.Length % 2`
  if a.Length % 2 == 0 {
    assume a != null;     // Again for the final `a.Length`
    assume a.Length != 0; // Because of the 10 / a.Length
  }

  // Now the body is translated
  var j := 20;
  assert a != null;          // For `var divisor := a.Length;`
  var divisor := a.Length;
  if * {
    assume divisor % 2 == 0;
    assert divisor != 0;
    j := j / divisor;
  }
  assume divisor % 2 == 0 ==> divisor != 0;
  assert a.Length % 2 == 0 ==> j >= 10 / a.Length;
}
```

For this reason the IDE typically reports at least two assertion batches when hovering a method.

8. Statements (grammar)

Many of Dafny's statements are similar to those in traditional programming languages, but a number of them are significantly different. Dafny's various kinds of statements are described in subsequent sections.

Statements have zero or more labels and end with either a semicolon (`;`) or a closing curly brace ('}').

8.1. Labeled Statement (grammar)

Examples:

```
class A { var f: int }
method m(a: A) {
  label x:
  while true {
     if (*) { break x; }
  }
  a.f := 0;
  label y:
  a.f := 1;
  assert old@y(a.f) == 1;
}
```

A labeled statement is just - the keyword `label` - followed by an identifier, which is the label, - followed by a colon - and a statement.

The label may be referenced in a `break` or `continue` statement within the labeled statement (see Section 8.14). That is, the break or continue that mentions the label must be *enclosed* in the labeled statement.

The label may also be used in an `old` expression (Section 9.22). In this case, the label must have been encountered during the control flow en route to the `old` expression. We say in this case that the (program point of the) label *dominates* the (program point of the) use of the label. Similarly, labels are used to indicate previous states in calls of two-state predicates, fresh expressions, unchanged expressions, and allocated expressions.

A statement can be given several labels. It makes no difference which of these labels is used to reference the statement—they are synonyms of each other. The labels must be distinct from each other, and are not allowed to be the same as any previous enclosing or dominating label.

8.2. Block Statement (grammar)

Examples:

```
{
  print 0;
  var x := 0;
}
```

A block statement is a sequence of zero or more statements enclosed by curly braces. Local variables declared in the block end their scope at the end of the block.

8.3. Return Statement (grammar)

Examples:

```
method m(i: int) returns (r: int) {
  return i+1;
}
method n(i: int) returns (r: int, q: int) {
  return i+1, i + 2;
}
method p() returns (i: int) {
  i := 1;
  return;
}
method q() {
  return;
}
```

A return statement can only be used in a method. It is used to terminate the execution of the method.

To return a value from a method, the value is assigned to one of the named out-parameters sometime before a return statement. In fact, the out-parameters act very much like local variables, and can be assigned to more than once. Return statements are used when one wants to return before reaching the end of the body block of the method.

Return statements can be just the `return` keyword (where the current values of the out-parameters are used), or they can take a list of expressions to return. If a list is given, the number of expressions given must be the same as the number of named out-parameters. These expressions are evaluated, then they are assigned to the out-parameters, and then the method terminates.

8.4. Yield Statement (grammar)

A yield statement may only be used in an iterator. See iterator types for more details about iterators.

The body of an iterator is a *co-routine*. It is used to yield control to its caller, signaling that a new set of values for the iterator's yield (out-)parameters (if any) are available. Values are assigned to the yield parameters at or before a yield statement. In fact, the yield parameters act very much like local variables, and can be assigned to more than once. Yield statements are used when one wants to return new yield parameter values to the caller. Yield statements can be just the `yield` keyword (where the current values of the yield parameters are used), or they can take a list of expressions to yield. If a list is given, the number of expressions given must be the same as the number of named iterator out-parameters. These expressions are then evaluated, then they are assigned to the yield parameters, and then the iterator yields.

8.5. Update and Call Statements (grammar)

Examples:

```
class C { var f: int }
class D {
  var i: int
  constructor(i: int) {
    this.i := i;
  }
}
method q(i: int, j: int) {}
method r() returns (s: int, t: int) { return 2,3; }
method m() {
  var ss: int, tt: int, c: C?, a: array<int>, d: D?;
  q(0,1);
  ss, c.f := r();
  c := new C;
  d := new D(2);
  a := new int[10];
  ss, tt := 212, 33;
  ss :| ss > 7;
  ss := *;
}
```

This statement corresponds to familiar assignment or method call statements, with variations. If more than one left-hand side is used, these must denote different l-values, unless the corresponding right-hand sides also denote the same value.

The update statement serves several logical purposes.

8.5.1. Method call with no out-parameters

1) Examples of method calls take this form

```
m();
m(1,2,3) {:attr} ;
e.f().g.m(45);
```

As there are no left-hand-side locations to receive values, this form is allowed only for methods that have no out-parameters.

8.5.2. Method call with out-parameters

This form uses `:=` to denote the assignment of the out-parameters of the method to the corresponding number of LHS values.

```
a, b.e().f := m() {:attr};
```

In this case, the right-hand-side must be a method call and the number of left-hand sides must match the number of out-parameters of the method that is called or there must be just one `Lhs` to the left of the `:=`, which then is assigned a tuple of the out-parameters. Note that the result of a method call is not allowed to be used as an argument of another method call, as if it were an expression.

8.5.3. Parallel assignment

A parallel-assignment has one-or-more right-hand-side expressions, which may be function calls but may not be method calls.

```
    x, y := y, x;
```

The above example swaps the values of `x` and `y`. If more than one left-hand side is used, these must denote different l-values, unless the corresponding right-hand sides also denote the same value. There must be an equal number of left-hand sides and right-hand sides. The most common case has only one RHS and one LHS.

8.5.4. Havoc assignment

The form with a right-hand-side that is `*` is a *havoc* assignment. It assigns an arbitrary but type-correct value to the corresponding left-hand-side. It can be mixed with other assignments of computed values.

```
a := *;
a, b, c := 4, *, 5;
```

8.5.5. Such-that assignment

This form has one or more left-hand-sides, a `:|` symbol and then a boolean expression on the right. The effect is to assign values to the left-hand-sides that satisfy the RHS condition.

```
x, y :| 0 < x+y < 10;
```

This is read as assign values to `x` and `y` such that `0 < x+y < 10` is true. The given boolean expression need not constrain the LHS values uniquely: the choice of satisfying values is non-deterministic. This can be used to make a choice as in the following example where we choose an element in a set.

```
method Sum(X: set<int>) returns (s: int)
{
  s := 0; var Y := X;
  while Y != {}
```

```
    decreases Y
  {
    var y: int;
    y :| y in Y;
    s, Y := s + y, Y - {y};
  }
}
```

Dafny will report an error if it cannot prove that values exist that satisfy the condition.

In this variation, with an `assume` keyword

```
    y :| assume y in Y;
```

Dafny assumes without proof that an appropriate value exists.

Note that the syntax

```
    Lhs ":"
```

is interpreted as a label in which the user forgot the `label` keyword.

8.6. Update with Failure Statement (`:-`) (grammar)

See the subsections below for examples.

A `:-`[8] statement is an alternate form of the `:=` statement that allows for abrupt return if a failure is detected. This is a language feature somewhat analogous to exceptions in other languages.

An update-with-failure statement uses *failure-compatible* types. A failure-compatible type is a type that has the following (non-static) members (each with no in-parameters and one out-parameter):

- a non-ghost function `IsFailure()` that returns a `bool`
- an optional non-ghost function `PropagateFailure()` that returns a value assignable to the first out-parameter of the caller
- an optional function `Extract()` (PropagateFailure and Extract were permitted to be methods (but deprecated) prior to Dafny 4. They will be required to be functions in Dafny 4.)

A failure-compatible type with an `Extract` member is called *value-carrying*.

To use this form of update,

- if the RHS of the update-with-failure statement is a method call, the first out-parameter of the callee must be failure-compatible
- if instead, the RHS of the update-with-failure statement is one or more expressions, the first of these expressions must be a value with a failure-compatible type
- the caller must have a first out-parameter whose type matches the output of `PropagateFailure` applied to the first output of the callee, unless an `expect`, `assume`, or `assert` keyword is used after `:-` (cf. Section 8.6.7).

[8]The `:-` token is called the elephant symbol or operator.

- if the failure-compatible type of the RHS does not have an `Extract` member, then the LHS of the `:-` statement has one less expression than the RHS (or than the number of out-parameters from the method call), the value of the first out-parameter or expression being dropped (see the discussion and examples in Section 8.6.2)
- if the failure-compatible type of the RHS does have an `Extract` member, then the LHS of the `:-` statement has the same number of expressions as the RHS (or as the number of out-parameters from the method call) and the type of the first LHS expression must be assignable from the return type of the `Extract` member
- the `IsFailure` and `PropagateFailure` methods may not be ghost
- the LHS expression assigned the output of the `Extract` member is ghost precisely if `Extract` is ghost

The following subsections show various uses and alternatives.

8.6.1. Failure compatible types

A simple failure-compatible type is the following:

```
datatype Status =
| Success
| Failure(error: string)
{
  predicate IsFailure() { this.Failure?  }
  function PropagateFailure(): Status
    requires IsFailure()
  {
    Failure(this.error)
  }
}
```

A commonly used alternative that carries some value information is something like this generic type:

```
datatype Outcome<T> =
| Success(value: T)
| Failure(error: string)
{
  predicate IsFailure() {
    this.Failure?
  }
  function PropagateFailure<U>(): Outcome<U>
    requires IsFailure()
  {
    Failure(this.error) // this is Outcome<U>.Failure(...)
  }
  function Extract(): T
    requires !IsFailure()
  {
    this.value
  }
}
```

8.6.2. Simple status return with no other outputs

The simplest use of this failure-return style of programming is to have a method call that just returns a non-value-carrying `Status` value:

```
method Callee(i: int) returns (r: Status)
{
  if i < 0 { return Failure("negative"); }
  return Success;
}

method Caller(i: int) returns (rr: Status)
{
  :- Callee(i);
  ...
}
```

Note that there is no LHS to the `:-` statement. If `Callee` returns `Failure`, then the caller immediately returns, not executing any statements following the call of `Callee`. The value returned by `Caller` (the value of `rr` in the code above) is the result of `PropagateFailure` applied to the value returned by `Callee`, which is often just the same value. If `Callee` does not return `Failure` (that is, returns a value for which `IsFailure()` is `false`) then that return value is forgotten and execution proceeds normally with the statements following the call of `Callee` in the body of `Caller`.

The desugaring of the `:- Callee(i);` statement is

```
var tmp;
tmp := Callee(i);
if tmp.IsFailure() {
  rr := tmp.PropagateFailure();
  return;
}
```

In this and subsequent examples of desugaring, the `tmp` variable is a new, unique variable, unused elsewhere in the calling member.

8.6.3. Status return with additional outputs

The example in the previous subsection affects the program only through side effects or the status return itself. It may well be convenient to have additional out-parameters, as is allowed for `:=` updates; these out-parameters behave just as for `:=`. Here is an example:

```
method Callee(i: int) returns (r: Status, v: int, w: int)
{
  if i < 0 { return Failure("negative"), 0, 0; }
  return Success, i+i, i*i;
}

method Caller(i: int) returns (rr: Status, k: int)
{
  var j: int;
  j, k :- Callee(i);
  k := k + k;
  ...
}
```

Here `Callee` has two outputs in addition to the `Status` output. The LHS of the `:-` statement accordingly has two l-values to receive those outputs. The recipients of those outputs may be any sort of l-values; here they are a local variable and an out-parameter of the caller. Those outputs are assigned in the `:-` call regardless of the `Status` value:

- If `Callee` returns a failure value as its first output, then the other outputs are assigned, the *caller's* first out-parameter (here `rr`) is assigned the value of `PropagateFailure`, and the caller returns.
- If `Callee` returns a non-failure value as its first output, then the other outputs are assigned and the caller continues execution as normal.

The desugaring of the `j, k :- Callee(i);` statement is

```
var tmp;
tmp, j, k := Callee(i);
if tmp.IsFailure() {
  rr := tmp.PropagateFailure();
  return;
}
```

8.6.4. Failure-returns with additional data

The failure-compatible return value can carry additional data as shown in the `Outcome<T>` example above. In this case there is a (first) LHS l-value to receive this additional data. The type of that first LHS value is one that is assignable from the result of the `Extract` function, not the actual first out-parameter.

```
method Callee(i: int) returns (r: Outcome<nat>, v: int)
{
  if i < 0 { return Failure("negative"), i+i; }
  return Success(i), i+i;
}

method Caller(i: int) returns (rr: Outcome<int>, k: int)
```

```
{
  var j: int;
  j, k :- Callee(i);
  k := k + k;
  ...
}
```

Suppose `Caller` is called with an argument of `10`. Then `Callee` is called with argument `10` and returns `r` and `v` of `Outcome<nat>.Success(10)` and `20`. Here `r.IsFailure()` is `false`, so control proceeds normally. The `j` is assigned the result of `r.Extract()`, which will be `10`, and `k` is assigned `20`. Control flow proceeds to the next line, where `k` now gets the value `40`.

Suppose instead that `Caller` is called with an argument of `-1`. Then `Callee` is called with the value `-1` and returns `r` and `v` with values `Outcome<nat>.Failure("negative")` and `-2`. `k` is assigned the value of `v` (-2). But `r.IsFailure()` is `true`, so control proceeds directly to return from `Caller`. The first out-parameter of `Caller` (`rr`) gets the value of `r.PropagateFailure()`, which is `Outcome<int>.Failure("negative")`; `k` already has the value `-2`. The rest of the body of `Caller` is skipped. In this example, the first out-parameter of `Caller` has a failure-compatible type so the exceptional return will propagate up the call stack. It will keep propagating up the call stack as long as there are callers with this first special output type and calls that use `:-` and the return value keeps having `IsFailure()` true.

The desugaring of the `j, k :- Callee(i);` statement in this example is

```
var tmp;
tmp, k := Callee(i);
if tmp.IsFailure() {
  rr := tmp.PropagateFailure();
  return;
}
j := tmp.Extract();
```

8.6.5. RHS with expression list

Instead of a failure-returning method call on the RHS of the statement, the RHS can instead be a list of expressions. As for a `:=` statement, in this form, the expressions on the left and right sides of `:-` must correspond, just omitting a LHS l-value for the first RHS expression if its type is not value-carrying. The semantics is very similar to that in the previous subsection.

- The first RHS expression must have a failure-compatible type.
- All the assignments of RHS expressions to LHS values except for the first RHS value are made.
- If the first RHS value (say `r`) responds `true` to `r.IsFailure()`, then `r.PropagateFailure()` is assigned to the first out-parameter of the *caller* and the execution of the caller's body is ended.
- If the first RHS value (say `r`) responds `false` to `r.IsFailure()`, then
  - if the type of `r` is value-carrying, then `r.Extract()` is assigned to the first LHS value of the `:-` statement; if `r` is not value-carrying, then the corresponding LHS l-value is

  omitted
  - execution of the caller's body continues with the statement following the `:-` statement.

A RHS with a method call cannot be mixed with a RHS containing multiple expressions.

For example, the desugaring of

```
method m(r: Status) returns (rr: Status) {
  var k;
  k :- r, 7;
  ...
}
```

is

```
var k;
var tmp;
tmp, k := r, 7;
if tmp.IsFailure() {
  rr := tmp.PropagateFailure();
  return;
}
```

8.6.6. Failure with initialized declaration.

The `:-` syntax can also be used in initialization, as in

```
var s, t :- M();
```

This is equivalent to

```
var s, t;
s, t :- M();
```

with the semantics as described above.

8.6.7. Keyword alternative

In any of the above described uses of `:-`, the `:-` token may be followed immediately by the keyword `expect`, `assert` or `assume`.

- `assert` means that the RHS evaluation is expected to be successful, but that the verifier should prove that this is so; that is, the verifier should prove `assert !r.IsFailure()` (where `r` is the status return from the callee) (cf. Section 8.17)
- `assume` means that the RHS evaluation should be assumed to be successful, as if the statement `assume !r.IsFailure()` followed the evaluation of the RHS (cf. Section 8.18)
- `expect` means that the RHS evaluation should be assumed to be successful (like using `assume` above), but that the compiler should include a run-time check for success. This is equivalent to including `expect !r.IsFailure()` after the RHS evaluation; that is, if the status return is a failure, the program halts. (cf. Section 8.19)

In each of these cases, there is no abrupt return from the caller. Thus there is no evaluation of `PropagateFailure`. Consequently the first out-parameter of the caller need not match the return type of `PropagateFailure`; indeed, the failure-compatible type returned by the callee need not have a `PropagateFailure` member.

The equivalent desugaring replaces

```
if tmp.IsFailure() {
  rr := tmp.PropagateFailure();
  return;
}
```

with

```
expect !tmp.IsFailure(), tmp;
```

or

```
assert !tmp.IsFailure();
```

or

```
assume !tmp.IsFailure();
```

There is a grammatical nuance that the user should be aware of. The keywords `assert`, `assume`, and `expect` can start an expression. For example, `assert P; E` can be an expression. However, in `e :- assert P; E;` the `assert` is parsed as the keyword associated with `:-`. To have the `assert` considered part of the expression use parentheses: `e :- (assert P; E);`.

8.6.8. Key points

There are several points to note.

- The first out-parameter of the callee is special. It has a special type and that type indicates that the value is inspected to see if an abrupt return from the caller is warranted. This type is often a datatype, as shown in the examples above, but it may be any type with the appropriate members.
- The restriction on the type of caller's first out-parameter is just that it must be possible (perhaps through generic instantiation and type inference, as in these examples) for `PropagateFailure` applied to the failure-compatible output from the callee to produce a value of the caller's first out-parameter type. If the caller's first out-parameter type is failure-compatible (which it need not be), then failures can be propagated up the call chain. If the keyword form (e.g. `assume`) of the statement is used, then no `PropagateFailure` member is needed, because no failure can occur, and there is no restriction on the caller's first out-parameter.
- In the statement `j, k :- Callee(i);`, when the callee's return value has an `Extract` member, the type of `j` is not the type of the first out-parameter of `Callee`. Rather it is a type assignable from the output type of `Extract` applied to the first out-value of `Callee`.
- A method like `Callee` with a special first out-parameter type can still be used in the normal way: `r, k := Callee(i)`. Now `r` gets the first output value from `Callee`, of

type `Status` or `Outcome<nat>` in the examples above. No special semantics or exceptional control paths apply. Subsequent code can do its own testing of the value of `r` and whatever other computations or control flow are desired.
- The caller and callee can have any (positive) number of output arguments, as long as the callee's first out-parameter has a failure-compatible type and the caller's first out-parameter type matches `PropagateFailure`.
- If there is more than one LHS, the LHSs must denote different l-values, unless the RHS is a list of expressions and the corresponding RHS values are equal.
- The LHS l-values are evaluated before the RHS method call, in case the method call has side-effects or return values that modify the l-values prior to assignments being made.

It is important to note the connection between the failure-compatible types used in the caller and callee, if they both use them. They do not have to be the same type, but they must be closely related, as it must be possible for the callee's `PropagateFailure` to return a value of the caller's failure-compatible type. In practice this means that one such failure-compatible type should be used for an entire program. If a Dafny program uses a library shared by multiple programs, the library should supply such a type and it should be used by all the client programs (and, effectively, all Dafny libraries). It is also the case that it is inconvenient to mix types such as `Outcome` and `Status` above within the same program. If there is a mix of failure-compatible types, then the program will need to use `:=` statements and code for explicit handling of failure values.

8.6.9. Failure returns and exceptions

The `:-` mechanism is like the exceptions used in other programming languages, with some similarities and differences.

- There is essentially just one kind of 'exception' in Dafny, the variations of the failure-compatible data type.
- Exceptions are passed up the call stack whether or not intervening methods are aware of the possibility of an exception, that is, whether or not the intervening methods have declared that they throw exceptions. Not so in Dafny: a failure is passed up the call stack only if each caller has a failure-compatible first out-parameter, is itself called in a `:-` statement, and returns a value that responds true to `IsFailure()`.
- All methods that contain failure-return callees must explicitly handle those failures using either `:-` statements or using `:=` statements with a LHS to receive the failure value.

8.7. Variable Declaration Statement (grammar)

Examples:

```
method m() {
  var x, y: int; // x's type is inferred, not necessarily 'int'
  var b: bool, k: int;
  x := 1; // settles x's type
}
```

A variable declaration statement is used to declare one or more local variables in a method or function. The type of each local variable must be given unless its type can be inferred, either from a given initial value, or from other uses of the variable. If initial values are given, the number of values must match the number of variables declared.

The scope of the declared variable extends to the end of the block in which it is declared. However, be aware that if a simple variable declaration is followed by an expression (rather than a subsequent statement) then the `var` begins a Let Expression and the scope of the introduced variables is only to the end of the expression. In this case, though, the `var` is in an expression context, not a statement context.

Note that the type of each variable must be given individually. The following code

```
var x, y : int;
var x, y := 5, 6;
var x, y :- m();
var x, y :| 0 < x + y < 10;
var (x, y) := makePair();
var Cons(x, y) = ConsMaker();
```

does not declare both `x` and `y` to be of type `int`. Rather it will give an error explaining that the type of `x` is underspecified if it cannot be inferred from uses of x.

The variables can be initialized with syntax similar to update statements (cf. Section 8.5).

If the RHS is a call, then any variable receiving the value of a formal ghost out-parameter will automatically be declared as ghost, even if the `ghost` keyword is not part of the variable declaration statement.

The left-hand side can also contain a tuple of patterns that will be matched against the right-hand-side. For example:

```
function returnsTuple() : (int, int)
{
    (5, 10)
}

function usesTuple() : int
{
    var (x, y) := returnsTuple();
    x + y
}
```

The initialization with failure operator `:-` returns from the enclosing method if the initializer evaluates to a failure value of a failure-compatible type (see Section 8.6).

8.8. Guards (grammar)

Examples (in `if` statements):

```
method m(i: int) {
  if (*) { print i; }
  if i > 0 { print i; }
}
```

Guards are used in `if` and `while` statements as boolean expressions. Guards take two forms.

The first and most common form is just a boolean expression.

The second form is either `*` or `(*)`. These have the same meaning. An unspecified boolean value is returned. The value returned may be different each time it is executed.

8.9. Binding Guards (grammar)

Examples (in `if` statements):

```
method m(i: int) {
  ghost var k: int;
  if i, j :| 0 < i+j < 10 {
    k := 0;
  } else {
    k := 1;
  }
}
```

An `if` statement can also take a *binding guard*. Such a guard checks if there exist values for the given variables that satisfy the given expression. If so, it binds some satisfying values to the variables and proceeds into the "then" branch; otherwise it proceeds with the "else" branch, where the bound variables are not in scope.

In other words, the statement

```
if x :| P { S } else { T }
```

has the same meaning as

```
if exists x :: P { var x :| P; S } else { T }
```

The identifiers bound by the binding guard are ghost variables and cannot be assigned to non-ghost variables. They are only used in specification contexts.

Here is another example:

```
predicate P(n: int)
{
  n % 2 == 0
}

method M1() returns (ghost y: int)
    requires exists x :: P(x)
    ensures P(y)
{
  if x : int :| P(x) {
      y := x;
  }
}
```

8.10. If Statement (grammar)

Examples:

```
method m(i: int) {
  var x: int;
  if i > 0 {
    x := i;
  } else {
    x := -i;
  }
  if * {
    x := i;
  } else {
    x := -i;
  }
  if i: nat, j: nat :| i+j<10 {
    assert i < 10;
  }
  if i == 0 {
    x := 0;
  } else if i > 0 {
    x := 1;
  } else {
    x := -1;
  }
  if
    case i == 0 => x := 0;
    case i > 0 => x := 1;
    case i < 0 => x := -1;
}
```

The simplest form of an `if` statement uses a guard that is a boolean expression. For example,

```
  if x < 0 {
    x := -x;
  }
```

Unlike `match` statements, `if` statements do not have to be exhaustive: omitting the `else` block is the same as including an empty `else` block. To ensure that an `if` statement is exhaustive, use the `if-case` statement documented below.

If the guard is an asterisk then a non-deterministic choice is made:

```
  if * {
    print "True";
  } else {
    print "False";
  }
```

The then alternative of the if-statement must be a block statement; the else alternative may be either a block statement or another if statement. The condition of the if statement need not (but may) be enclosed in parentheses.

An if statement with a binding guard is non-deterministic; it will not be compiled if `--enforce-determinism` is enabled (even if it can be proved that there is a unique value). An if statement with `*` for a guard is non-deterministic and ghost.

The `if-case` statement using the `AlternativeBlock` form is similar to the `if ... fi` construct used in the book "A Discipline of Programming" by Edsger W. Dijkstra. It is used for a multi-branch `if`.

For example:

```
method m(x: int, y: int) returns (max: int)
{
  if {
    case x <= y => max := y;
    case y <= x => max := x;
  }
}
```

In this form, the expressions following the `case` keyword are called *guards*. The statement is evaluated by evaluating the guards in an undetermined order until one is found that is `true` and the statements to the right of `=>` for that guard are executed. The statement requires at least one of the guards to evaluate to `true` (that is, `if-case` statements must be exhaustive: the guards must cover all cases).

In the if-with-cases, a sequence of statements may follow the `=>`; it may but need not be a block statement (a brace-enclosed sequence of statements).

The form that used `...` (a refinement feature) as the guard is deprecated.

8.11. Match Statement (grammar)

Examples:

```
match list {
  case Nil => {}
  case Cons(head,tail) => print head;
}
match x
case 1 =>
  print x;
case 2 =>
  var y := x*x;
  print y;
case _ =>
  print "Other";
  // Any statement after is captured in this case.
```

The `match` statement is used to do case analysis on a value of an expression. The expression may be a value of a basic type (e.g. `int`), a newtype, or an inductive or coinductive datatype (which includes the built-in tuple types). The expression after the `match` keyword is called the *selector*. The selector is evaluated and then matched against each clause in order until a matching clause is found.

The process of matching the selector expression against the case patterns is the same as for match expressions and is described in Section 9.31.2.

The selector need not be enclosed in parentheses; the sequence of cases may but need not be enclosed in braces. The cases need not be disjoint. The cases must be exhaustive, but you can use a wild variable (`_`) or a simple identifier to indicate "match anything". Please refer to the section about case patterns to learn more about shadowing, constants, etc.

The code below shows an example of a match statement.

```
datatype Tree = Empty | Node(left: Tree, data: int, right: Tree)

// Return the sum of the data in a tree.
method Sum(x: Tree) returns (r: int)
{
  match x {
    case Empty => r := 0;
    case Node(t1, d, t2) =>
      var v1 := Sum(t1);
      var v2 := Sum(t2);
      r := v1 + d + v2;
  }
}
```

Note that the `Sum` method is recursive yet has no `decreases` annotation. In this case it is not needed because Dafny is able to deduce that `t1` and `t2` are *smaller* (structurally) than `x`. If `Tree` had been coinductive this would not have been possible since `x` might have been infinite.

8.12. While Statement (grammar)

Examples:

```
method m() {
  var i := 10;
  while 0 < i
    invariant 0 <= i <= 10
    decreases i
  {
    i := i-1;
  }
  while * {}
  i := *;
  while
     decreases if i < 0 then -i else i
  {
     case i < 0 => i := i + 1;
     case i > 0 => i := i - 1;
  }
}
```

Loops - may be a conventional loop with a condition and a block statement for a body - need not have parentheses around the condition - may have a `*` for the condition (the loop is then non-deterministic) - binding guards are not allowed - may have a case-based structure - may have no body — a bodyless loop is not compilable, but can be reaosnaed about

Importantly, loops need *loop specifications* in order for Dafny to prove that they obey expected behavior. In some cases Dafny can infer the loop specifications by analyzing the code, so the loop specifications need not always be explicit. These specifications are described in Section 7.6 and Section 8.15.

The general loop statement in Dafny is the familiar `while` statement. It has two general forms.

The first form is similar to a while loop in a C-like language. For example:

```
method m(){
  var i := 0;
  while i < 5 {
    i := i + 1;
  }
}
```

In this form, the condition following the `while` is one of these:

- A boolean expression. If true it means execute one more iteration of the loop. If false then terminate the loop.
- An asterisk (`*`), meaning non-deterministically yield either `true` or `false` as the value of the condition

The *body* of the loop is usually a block statement, but it can also be missing altogether. A loop with a missing body may still pass verification, but any attempt to compile the containing program will result in an error message. When verifying a loop with a missing body, the verifier will skip attempts to prove loop invariants and decreases assertions that would normally be asserted at the end of the loop body. There is more discussion about bodyless loops in Section 8.15.4.

The second form uses a case-based block. It is similar to the `do ... od` construct used in the book "A Discipline of Programming" by Edsger W. Dijkstra. For example:

```
method m(n: int){
  var r := n;
  while
    decreases if 0 <= r then r else -r
  {
    case r < 0 =>
      r := r + 1;
    case 0 < r =>
      r := r - 1;
  }
}
```

For this form, the guards are evaluated in some undetermined order until one is found that is true, in which case the corresponding statements are executed and the while statement is repeated. If none of the guards evaluates to true, then the loop execution is terminated.

The form that used `...` (a refinement feature) as the guard is deprecated.

8.13. For Loops (grammar)

Examples:

```
method m() decreases * {
  for i := 0 to 10 {}
  for _ := 0 to 10 {}
  for i := 0 to * invariant i >= 0 decreases * {}
  for i: int := 10 downto 0 {}
  for i: int := 10 downto 0
}
```

The `for` statement provides a convenient way to write some common loops.

The statement introduces a local variable with optional type, which is called the *loop index*. The loop index is in scope in the specification and the body, but not after the `for` loop. Assignments to the loop index are not allowed. The type of the loop index can typically be inferred; if so, it need not be given explicitly. If the identifier is not used, it can be written as `_`, as illustrated in this repeat-20-times loop:

```
for _ := 0 to 20 {
  Body
}
```

There are four basic variations of the `for` loop:

```
for i: T := lo to hi
  LoopSpec
{ Body }

for i: T := hi downto lo
  LoopSpec
{ Body }

for i: T := lo to *
  LoopSpec
{ Body }

for i: T := hi downto *
  LoopSpec
{ Body }
```

Semantically, they are defined as the following respective `while` loops:

```
{
  var _lo, _hi := lo, hi;
  assert _lo <= _hi && forall _i: int :: _lo <= _i <= _hi ==> _i is T;
  var i := _lo;
  while i != _hi
    invariant _lo <= i <= _hi
    LoopSpec
    decreases _hi - i
  {
    Body
    i := i + 1;
  }
}

{
  var _lo, _hi := lo, hi;
  assert _lo <= _hi && forall _i: int :: _lo <= _i <= _hi ==> _i is T;
  var i := _hi;
  while i != lo
    invariant _lo <= i <= _hi
    LoopSpec
    decreases i - _lo
  {
    i := i - 1;
    Body
  }
}
```

```
{
  var _lo := lo;
  assert forall _i: int :: _lo <= _i ==> _i is T;
  var i := _lo;
  while true
    invariant _lo <= i
    LoopSpec
  {
    Body
    i := i + 1;
  }
}

{
  var _hi := hi;
  assert forall _i: int :: _i <= _hi ==> _i is T;
  var i := _hi;
  while true
    invariant i <= _hi
    LoopSpec
  {
    i := i - 1;
    Body
  }
}
```

The expressions `lo` and `hi` are evaluated just once, before the loop iterations start.

Also, in all variations the values of `i` in the body are the values from `lo` to, *but not including*, `hi`. This makes it convenient to write common loops, including these:

```
for i := 0 to a.Length {
  Process(a[i]);
}
for i := a.Length downto 0 {
  Process(a[i]);
}
```

Nevertheless, `hi` must be a legal value for the type of the index variable, since that is how the index variable is used in the invariant.

If the end-expression is not `*`, then no explicit `decreases` is allowed, since such a loop is already known to terminate. If the end-expression is `*`, then the absence of an explicit `decreases` clause makes it default to `decreases *`. So, if the end-expression is `*` and no explicit `decreases` clause is given, the loop is allowed only in methods that are declared with `decreases *`.

The directions `to` or `downto` are contextual keywords. That is, these two words are part of the syntax of the `for` loop, but they are not reserved keywords elsewhere.

Just like for while loops, the body of a for-loop may be omitted during verification. This suppresses attempts to check assertions (like invariants) that would occur at the end of the loop. Eventually, however a body must be provided; the compiler will not compile a method containing a body-less for-loop. There is more discussion about bodyless loops in Section 8.15.4.

8.14. Break and Continue Statements (grammar)

Examples:

```
class A { var f: int }
method m(a: A) {
  label x:
  while true {
    if (*) { break; }
  }
  label y: {
    var z := 1;
    if * { break y; }
    z := 2;
  }

}
```

Break and continue statements provide a means to transfer control in a way different than the usual nested control structures. There are two forms of each of these statements: with and without a label.

If a label is used, the break or continue statement must be enclosed in a statement with that label. The enclosing statement is called the *target* of the break or continue.

A `break` statement transfers control to the point immediately following the target statement. For example, such a break statement can be used to exit a sequence of statements in a block statement before reaching the end of the block.

For example,

```
label L: {
  var n := ReadNext();
  if n < 0 {
    break L;
  }
  DoSomething(n);
}
```

is equivalent to

```
{
  var n := ReadNext();
  if 0 <= n {
    DoSomething(n);
  }
}
```

If no label is specified and the statement lists `n` occurrences of `break`, then the statement must be enclosed in at least `n` levels of loop statements. Control continues after exiting `n` enclosing loops. For example,

```
method m() {
  for i := 0 to 10 {
    for j := 0 to 10 {
      label X: {
        for k := 0 to 10 {
          if j + k == 15 {
            break break;
          }
        }
      }
    }
    // control continues here after the "break break", exiting two loops
  }
}
```

Note that a non-labeled `break` pays attention only to loops, not to labeled statements. For example, the labeled block `X` in the previous example does not play a role in determining the target statement of the `break break;`.

For a `continue` statement, the target statement must be a loop statement. The continue statement transfers control to the point immediately before the closing curly-brace of the loop body.

For example,

```
method m() {
  for i := 0 to 100 {
    if i == 17 {
      continue;
    }
    DoSomething(i);
  }
}
method DoSomething(i:int){}
```

is equivalent to

```
method m() {
  for i := 0 to 100 {
    if i != 17 {
      DoSomething(i);
    }
  }
}
method DoSomething(i:int){}
```

The same effect can also be obtained by wrapping the loop body in a labeled block statement and then using `break` with a label, but that usually makes for a more cluttered program:

```
method m() {
  for i := 0 to 100 {
    label LoopBody: {
      if i == 17 {
        break LoopBody;
      }
      DoSomething(i);
    }
  }
}
method DoSomething(i:int){}
```

Stated differently, `continue` has the effect of ending the current loop iteration, after which control continues with any remaining iterations. This is most natural for `for` loops. For a `while` loop, be careful to make progress toward termination before a `continue` statement. For example, the following program snippet shows an easy mistake to make (the verifier will complain that the loop may not terminate):

```
method m() {
  var i := 0;
  while i < 100 {
    if i == 17 {
      continue; // error: this would cause an infinite loop
    }
    DoSomething(i);
    i := i + 1;
  }
}
method DoSomething(i:int){}
```

The `continue` statement can give a label, provided the label is a label of a loop. For example,

```
method m() {
  label Outer:
  for i := 0 to 100 {
    for j := 0 to 100 {
      if i + j == 19 {
        continue Outer;
      }
      WorkIt(i, j);
    }
    PostProcess(i);
    // the "continue Outer" statement above transfers control to here
  }
}
method WorkIt(i:int, j:int){}
method PostProcess(i:int){}
```

If a non-labeled continue statement lists `n` occurrences of `break` before the `continue` keyword, then the statement must be enclosed in at least `n + 1` levels of loop statements. The effect is to `break` out of the `n` most closely enclosing loops and then `continue` the iterations of the next loop. That is, `n` occurrences of `break` followed by one more `break;` will break out of `n` levels of loops and then do a `break`, whereas `n` occurrences of `break` followed by `continue;` will break out of `n` levels of loops and then do a `continue`.

For example, the `WorkIt` example above can equivalently be written without labels as

```
method m() {
  for i := 0 to 100 {
    for j := 0 to 100 {
      if i + j == 19 {
        break continue;
      }
      WorkIt(i, j);
    }
    PostProcess(i);
    // the "break continue" statement above transfers control to here
  }
}
method WorkIt(i:int, j:int){}
method PostProcess(i:int){}
```

Note that a loop invariant is checked on entry to a loop and at the closing curly-brace of the loop body. It is not checked at break statements. For continue statements, the loop invariant is checked as usual at the closing curly-brace that the continue statement jumps to. This checking ensures that the loop invariant holds at the very top of every iteration. Commonly, the only exit out of a loop happens when the loop guard evaluates to `false`. Since no state is changed between the top of an iteration (where the loop invariant is known to hold) and the evaluation of the loop guard, one can also rely on the loop invariant to hold immediately following the loop. But the loop invariant may not hold immediately following a loop if a loop iteration changes the program state and then exits the loop with a break statement.

For example, the following program verifies:

```
method m() {
  var i := 0;
  while i < 10
```

```
    invariant 0 <= i <= 10
  {
    if P(i) {
      i := i + 200;
      break;
    }
    i := i + 1;
  }
  assert i == 10 || 200 <= i < 210;
}
predicate P(i:int)
```

To explain the example, the loop invariant `0 <= i <= 10` is known to hold at the very top of each iteration, that is, just before the loop guard `i < 10` is evaluated. If the loop guard evaluates to `false`, then the negated guard condition (`10 <= i`) and the invariant hold, so `i == 10` will hold immediately after the loop. If the loop guard evaluates to `true` (that is, `i < 10` holds), then the loop body is entered. If the test `P(i)` then evaluates to `true`, the loop adds `200` to `i` and breaks out of the loop, so on such a path, `200 <= i < 210` is known to hold immediately after the loop. This is summarized in the assert statement in the example. So, remember, a loop invariant holds at the very top of every iteration, not necessarily immediately after the loop.

8.15. Loop Specifications

For some simple loops, such as those mentioned previously, Dafny can figure out what the loop is doing without more help. However, in general the user must provide more information in order to help Dafny prove the effect of the loop. This information is provided by a *loop specification*. A loop specification provides information about invariants, termination, and what the loop modifies. For additional tutorial information see (Koenig and Leino 2012) or the online Dafny tutorial.

8.15.1. Loop invariants

Loops present a problem for specification-based reasoning. There is no way to know in advance how many times the code will go around the loop and a tool cannot reason about every one of a possibly unbounded sequence of unrollings. In order to consider all paths through a program, specification-based program verification tools require loop invariants, which are another kind of annotation.

A loop invariant is an expression that holds just prior to the loop test, that is, upon entering a loop and after every execution of the loop body. It captures something that is invariant, i.e. does not change, about every step of the loop. Now, obviously we are going to want to change variables, etc. each time around the loop, or we wouldn't need the loop. Like pre- and postconditions, an invariant is a property that is preserved for each execution of the loop, expressed using the same boolean expressions we have seen. For example,

```
var i := 0;
while i < n
  invariant 0 <= i
```

```
{
  i := i + 1;
}
```

When you specify an invariant, Dafny proves two things: the invariant holds upon entering the loop, and it is preserved by the loop. By preserved, we mean that assuming that the invariant holds at the beginning of the loop (just prior to the loop test), we must show that executing the loop body once makes the invariant hold again. Dafny can only know upon analyzing the loop body what the invariants say, in addition to the loop guard (the loop condition). Just as Dafny will not discover properties of a method on its own, it will not know that any but the most basic properties of a loop are preserved unless it is told via an invariant.

8.15.2. Loop termination

Dafny proves that code terminates, i.e. does not loop forever, by using `decreases` annotations. For many things, Dafny is able to guess the right annotations, but sometimes it needs to be made explicit. There are two places Dafny proves termination: loops and recursion. Both of these situations require either an explicit annotation or a correct guess by Dafny.

A `decreases` annotation, as its name suggests, gives Dafny an expression that decreases with every loop iteration or recursive call. There are two conditions that Dafny needs to verify when using a `decreases` expression:

- that the expression actually gets smaller, and
- that it is bounded.

That is, the expression must strictly decrease in a well-founded ordering (cf. Section 12.7).

Many times, an integral value (natural or plain integer) is the quantity that decreases, but other values can be used as well. In the case of integers, the bound is assumed to be zero. For each loop iteration the `decreases` expression at the end of the loop body must be strictly smaller than its value at the beginning of the loop body (after the loop test). For integers, the well-founded relation between `x` and `X` is `x < X && 0 <= X`. Thus if the `decreases` value (`X`) is negative at the loop test, it must exit the loop, since there is no permitted value for `x` to have at the end of the loop body.

For example, the following is a proper use of `decreases` on a loop:

```
method m(n: nat){
  var i := n;
  while 0 < i
    invariant 0 <= i
    decreases i
  {
    i := i - 1;
  }
}
```

Here Dafny has all the ingredients it needs to prove termination. The variable `i` becomes smaller each loop iteration, and is bounded below by zero. When `i` becomes 0, the lower bound of the well-founded order, control flow exits the loop.

This is fine, except the loop is backwards compared to most loops, which tend to count up instead of down. In this case, what decreases is not the counter itself, but rather the distance between the counter and the upper bound. A simple trick for dealing with this situation is given below:

```
method m(m: nat, n: int)
  requires m <= n
{
  var i := m;
  while i < n
    invariant 0 <= i <= n
    decreases n - i
  {
    i := i + 1;
  }
}
```

This is actually Dafny's guess for this situation, as it sees `i < n` and assumes that `n - i` is the quantity that decreases. The upper bound of the loop invariant implies that `0 <= n - i`, and gives Dafny a lower bound on the quantity. This also works when the bound `n` is not constant, such as in the binary search algorithm, where two quantities approach each other, and neither is fixed.

If the `decreases` clause of a loop specifies `*`, then no termination check will be performed. Use of this feature is sound only with respect to partial correctness.

8.15.3. Loop framing

The specification of a loop also includes *framing*, which says what the loop modifies. The loop frame includes both local variables and locations in the heap.

For local variables, the Dafny verifier performs a syntactic scan of the loop body to find every local variable or out-parameter that occurs as a left-hand side of an assignment. These variables are called *syntactic assignment targets of the loop*, or *syntactic loop targets* for short. Any local variable or out-parameter that is not a syntactic assignment target is known by the verifier to remain unchanged by the loop.

The heap may or may not be a syntactic loop target. It is when the loop body syntactically contains a statement that can modify a heap location. This includes calls to compiled methods, even if such a method has an empty `modifies` clause, since a compiled method is always allowed to allocate new objects and change their values in the heap.

If the heap is not a syntactic loop target, then the verifier knows the heap remains unchanged by the loop. If the heap *is* a syntactic loop target, then the loop's effective `modifies` clause determines what is allowed to be modified by iterations of the loop body.

A loop can use `modifies` clauses to declare the effective `modifies` clause of the loop. If a loop does not explicitly declare any `modifies` clause, then the effective `modifies` clause of the loop is the effective `modifies` clause of the most tightly enclosing loop or, if there is no enclosing loop, the `modifies` clause of the enclosing method.

In most cases, there is no need to give an explicit `modifies` clause for a loop. The one case where it is sometimes needed is if a loop modifies less than is allowed by the enclosing method. Here are two simple methods that illustrate this case:

```
class Cell {
  var data: int
}

method M0(c: Cell, d: Cell)
  requires c != d
  modifies c, d
  ensures c.data == d.data == 100
{
  c.data, d.data := 100, 0;
  var i := 0;
  while i < 100
    invariant d.data == i
    // Needs "invariant c.data == 100" or "modifies d" to verify
  {
    d.data := d.data + 1;
    i := i + 1;
  }
}

method M1(c: Cell)
  modifies c
  ensures c.data == 100
{
  c.data := 100;
  var i := 0;
  while i < 100
    // Needs "invariant c.data == 100" or "modifies {}" to verify
  {
    var tmp := new Cell;
    tmp.data := i;
    i := i + 1;
  }
}
```

In `M0`, the effective `modifies` clause of the loop is `modifies c, d`. Therefore, the method's postcondition `c.data == 100` is not provable. To remedy the situation, the loop needs to be declared either with `invariant c.data == 100` or with `modifies d`.

Similarly, the effective `modifies` clause of the loop in `M1` is `modifies c`. Therefore, the method's postcondition `c.data == 100` is not provable. To remedy the situation, the loop needs to be declared either with `invariant c.data == 100` or with `modifies {}`.

When a loop has an explicit `modifies` clause, there is, at the top of every iteration, a proof obligation that

- the expressions given in the `modifies` clause are well-formed, and
- everything indicated in the loop `modifies` clause is allowed to be modified by the (effective `modifies` clause of the) enclosing loop or method.

8.15.4. Body-less methods, functions, loops, and aggregate statements

Methods (including lemmas), functions, loops, and `forall` statements are ordinarily declared with a body, that is, a curly-braces pair that contains (for methods, loops, and `forall`) a list of zero-or-more statements or (for a function) an expression. In each case, Dafny syntactically allows these constructs to be given without a body (no braces at all). This is to allow programmers to temporarily postpone the development of the implementation of the method, function, loop, or aggregate statement.

If a method has no body, there is no difference for callers of the method. Callers still reason about the call in terms of the method's specification. But without a body, the verifier has no method implementation to check against the specification, so the verifier is silently happy. The compiler, on the other hand, will complain if it encounters a body-less method, because the compiler is supposed to generate code for the method, but it isn't clever enough to do that by itself without a given method body. If the method implementation is provided by code written outside of Dafny, the method can be marked with an `{:extern}` annotation, in which case the compiler will no longer complain about the absence of a method body; the verifier will not object either, even though there is now no proof that the Dafny specifications are satisfied by the external implementation.

A lemma is a special kind of (ghost) method. Callers are therefore unaffected by the absence of a body, and the verifier is silently happy with not having a proof to check against the lemma specification. Despite a lemma being ghost, it is still the compiler that checks for, and complains about, body-less lemmas. A body-less lemma is an unproven lemma, which is often known as an *axiom*. If you intend to use a lemma as an axiom, omit its body and add the attribute `{:axiom}`, which causes the compiler to suppress its complaint about the lack of a body.

Similarly, calls to a body-less function use only the specification of the function. The verifier is silently happy, but the compiler complains (whether or not the function is ghost). As for methods and lemmas, the `{:extern}` and `{:axiom}` attributes can be used to suppress the compiler's complaint.

By supplying a body for a method or function, the verifier will in effect show the feasibility of the specification of the method or function. By supplying an `{:extern}` or `{:axiom}` attribute, you are taking that responsibility into your own hands. Common mistakes include forgetting to provide an appropriate `modifies` or `reads` clause in the specification, or forgetting that the results of functions in Dafny (unlike in most other languages) must be deterministic.

Just like methods and functions have two sides, callers and implementations, loops also have two sides. One side (analogous to callers) is the context that uses the loop. That context treats the loop in the same way, using its specifications, regardless of whether or not the loop has a body. The other side is the loop body, that is, the implementation of each loop iteration. The verifier checks that the loop body maintains the loop invariant and that the iterations will eventually terminate, but if there is no loop body, the verifier is silently happy. This allows you to temporarily postpone the authoring of the loop body until after you've made sure that the loop specification is what you need in the context of the loop.

There is one thing that works differently for body-less loops than for loops with bodies. It is the computation of syntactic loop targets, which become part of the loop frame (see Section 8.15.3). For a body-less loop, the local variables computed as part of the loop frame are the mutable variables that occur free in the loop specification. The heap is considered a part of the loop frame if it is used for mutable fields in the loop specification or if the loop has an explicit `modifies` clause. The IDE will display the computed loop frame in hover text.

For example, consider

```
class Cell {
  var data: int
  const K: int
}

method BodylessLoop(n: nat, c: Cell)
  requires c.K == 8
  modifies c
{
  c.data := 5;
  var a, b := n, n;
  for i := 0 to n
    invariant c.K < 10
    invariant a <= n
    invariant c.data < 10
  assert a == n;
  assert b == n;
  assert c.data == 5;
}
```

The loop specification mentions local variable `a`, and thus `a` is considered part of the loop frame. Since what the loop invariant says about `a` is not strong enough to prove the assertion `a == n` that follows the loop, the verifier complains about that assertion.

Local variable `b` is not mentioned in the loop specification, and thus `b` is not included in the loop frame. Since in-parameter `n` is immutable, it is not included in the loop frame, either, despite being mentioned in the loop specification. For these reasons, the assertion `b == n` is provable after the loop.

Because the loop specification mentions the mutable field `data`, the heap becomes part of the loop frame. Since the loop invariant is not strong enough to prove the assertion `c.data ==`

5 that follows the loop, the verifier complains about that assertion. On the other hand, had c.data < 10 not been mentioned in the loop specification, the assertion would be verified, since field K is then the only field mentioned in the loop specification and K is immutable.

Finally, the aggregate statement (forall) can also be given without a body. Such a statement claims that the given ensures clause holds true for all values of the bound variables that satisfy the given range constraint. If the statement has no body, the program is in effect omitting the proof, much like a body-less lemma is omitting the proof of the claim made by the lemma specification. As with the other body-less constructs above, the verifier is silently happy with a body-less forall statement, but the compiler will complain.

8.16. Print Statement (grammar)

Examples:

```
print 0, x, list, array;
```

The print statement is used to print the values of a comma-separated list of expressions to the console (standard-out). The generated code uses target-language-specific idioms to perform this printing. The expressions may of course include strings that are used for captions. There is no implicit new line added, so to add a new line you must include "\n" as part of one of the expressions. Dafny automatically creates implementations of methods that convert values to strings for all Dafny data types. For example,

```
datatype Tree = Empty | Node(left: Tree, data: int, right: Tree)
method Main()
{
  var x : Tree := Node(Node(Empty, 1, Empty), 2, Empty);
  print "x=", x, "\n";
}
```

produces this output:

```
x=Tree.Node(Tree.Node(Tree.Empty, 1, Tree.Empty), 2, Tree.Empty)
```

Note that Dafny does not have method overriding and there is no mechanism to override the built-in value->string conversion. Nor is there a way to explicitly invoke this conversion. One can always write an explicit function to convert a data value to a string and then call it explicitly in a print statement or elsewhere.

By default, Dafny does not keep track of print effects, but this can be changed using the --track-print-effects command line flag. print statements are allowed only in non-ghost contexts and not in expressions, with one exception. The exception is that a function-by-method may contain print statements, whose effect may be observed as part of the run-time evaluation of such functions (unless --track-print-effects is enabled).

The verifier checks that each expression is well-defined, but otherwise ignores the print statement.

Note: print writes to standard output. To improve compatibility with native code and external libraries, the process of encoding Dafny strings passed to `print` into standard-output byte strings is left to the runtime of the language that the Dafny code is compiled to (some language runtimes use UTF-8 in all cases; others obey the current locale or console encoding).

In most cases, the standard-output encoding can be set before running the compiled program using language-specific flags or environment variables (e.g. `-Dfile.encoding=` for Java). This is in fact how `dafny run` operates: it uses language-specific flags and variables to enforce UTF-8 output regardless of the target language (but note that the C++ and Go backends currently have limited support for UTF-16 surrogates).

8.17. Assert statement (grammar)

Examples:

```
assert i > 0;
assert IsPositive: i > 0;
assert i > 0 by {
 ...
}
```

`Assert` statements are used to express logical propositions that are expected to be true. Dafny will attempt to prove that the assertion is true and give an error if the assertion cannot be proven. Once the assertion is proved, its truth may aid in proving subsequent deductions. Thus if Dafny is having a difficult time verifying a method, the user may help by inserting assertions that Dafny can prove, and whose truth may aid in the larger verification effort, much as lemmas might be used in mathematical proofs.

`Assert` statements are ignored by the compiler.

In the `by` form of the `assert` statement, there is an additional block of statements that provide the Dafny verifier with additional proof steps. Those statements are often a sequence of lemmas, `calc` statements, `reveal` statements or other `assert` statements, combined with ghost control flow, ghost variable declarations and ghost update statements of variables declared in the `by` block. The intent is that those statements be evaluated in support of proving the `assert` statement. For that purpose, they could be simply inserted before the `assert` statement. But by using the `by` block, the statements in the block are discarded after the assertion is proved. As a result, the statements in the block do not clutter or confuse the solver in performing subsequent proofs of assertions later in the program. Furthermore, by isolating the statements in the `by` block, their purpose – to assist in proving the given assertion – is manifest in the structure of the code.

Examples of this form of assert are given in the section of the `reveal` statement and in *Different Styles of Proof*

An assert statement may have a label, whose use is explained in Section 8.20.1.

The attributes recognized for assert statements are discussed in Section 11.4.

Using `...` as the argument of the statement is deprecated.

An assert statement can have custom error and success messages.

8.18. Assume Statement (grammar)

Examples:

```
assume i > 0;
assume {:axiom} i > 0 ==> -i < 0;
```

The `assume` statement lets the user specify a logical proposition that Dafny may assume to be true without proof. If in fact the proposition is not true this may lead to invalid conclusions.

An `assume` statement would ordinarily be used as part of a larger verification effort where verification of some other part of the program required the proposition. By using the `assume` statement the other verification can proceed. Then when that is completed the user would come back and replace the `assume` with `assert`.

To help the user not forget about that last step, a warning is emitted for any assume statement. Adding the `{:axiom}` attribute to the assume will suppress the warning, indicating the user takes responsibility for being absolutely sure that the proposition is indeed true.

Using `...` as the argument of the statement is deprecated.

8.19. Expect Statement (grammar)

Examples:

```
expect i > 0;
expect i > 0, "i is positive";
```

The `expect` statement states a boolean expression that is (a) assumed to be true by the verifier and (b) checked to be true at run-time. That is, the compiler inserts into the run-time executable a check that the given expression is true; if the expression is false, then the execution of the program halts immediately. If a second argument is given, it may be a value of any type. That value is converted to a string (just like the `print` statement) and the string is included in the message emitted by the program when it halts; otherwise a default message is emitted.

Because the expect expression and optional second argument are compiled, they cannot be ghost expressions.

The `expect` statement behaves like `assume` for the verifier, but also inserts a run-time check that the assumption is indeed correct (for the test cases used at run-time).

Here are a few use-cases for the `expect` statement.

A) To check the specifications of external methods.

Consider an external method `Random` that takes a `nat` as input and returns a `nat` value that is less than the input. Such a method could be specified as

```
method {:extern} Random(n: nat) returns (r: nat)
  ensures r < n
```

But because there is no body for `Random` (only the external non-dafny implementation), it cannot be verified that `Random` actually satisfies this specification.

To mitigate this situation somewhat, we can define a wrapper function, `Random'`, that calls `Random` but in which we can put some run-time checks:

```
method {:extern} Random(n: nat) returns (r: nat)

method Random'(n: nat) returns (r: nat)
  ensures r < n
{
  r := Random(n);
  expect r < n;
}
```

Here we can verify that `Random'` satisfies its own specification, relying on the unverified specification of `Random`. But we are also checking at run-time that any input-output pairs for `Random` encountered during execution do satisfy the specification, as they are checked by the `expect` statement.

Note, in this example, two problems still remain. One problem is that the out-parameter of the extern `Random` has type `nat`, but there is no check that the value returned really is non-negative. It would be better to declare the out-parameter of `Random` to be `int` and to include `0 <= r` in the condition checked by the `expect` statement in `Random'`. The other problem is that `Random` surely will need `n` to be strictly positive. This can be fixed by adding `requires n != 0` to `Random'` and `Random`.

B) Run-time testing

Verification and run-time testing are complementary and both have their role in assuring that software does what is intended. Dafny can produce executables and these can be instrumented with unit tests. Annotating a method with the `{:test}` attribute indicates to the compiler that it should produce target code that is correspondingly annotated to mark the method as a unit test (e.g., an XUnit test) in the target language. Alternatively, the `dafny test` command will produce a main method that invokes all methods with the `{:test}` attribute, and hence does not depend on any testing framework in the target language. Within such methods one might use `expect` statements (as well as `print` statements) to insert checks that the target program is behaving as expected.

C) Debugging

While developing a new program, one work style uses proof attempts and runtime tests in combination. If an assert statement does not prove, one might run the program with a corresponding expect statement to see if there are some conditions when the assert is not actually true. So one might have paired assert/expect statements:

```
assert _P_;
expect _P_;
```

Once the program is debugged, both statements can be removed. Note that it is important that the `assert` come before the `expect`, because by the verifier, the `expect` is interpreted as an `assume`, which would automatically make a subsequent `assert` succeed.

D) Compiler tests

The same approach might be taken to assure that compiled code is behaving at run-time consistently with the statically verified code, one can again use paired assert/expect statements with the same expression:

```
assert _P_;
expect _P_;
```

The verifier will check that P is always true at the given point in a program (at the `assert` statement).

At run-time, the compiler will insert checks that the same predicate, in the `expect` statement, is true. Any difference identifies a compiler bug. Again the `expect` must be after the `assert`: if the `expect` is first, then the verifier will interpret the `expect` like an `assume`, in which case the `assert` will be proved trivially and potential unsoundness will be hidden.

Using `...` as the argument of the statement is deprecated.

8.20. Reveal Statement (grammar)

Examples:

```
reveal f(), L;
```

The `reveal` statement makes available to the solver information that is otherwise not visible, as described in the following subsections.

8.20.1. Revealing assertions

If an assert statement has an expression label, then a proof of that assertion is attempted, but the assertion itself is not used subsequently. For example, consider

```
method m(i: int) {
  assert x: i == 0; // Fails
  assert i == 0; // Fails also because the x: makes the first assertion opaque
}
```

The first assertion fails. Without the label `x:`, the second would succeed because after a failing assertion, the assertion is assumed in the context of the rest of the program. But with the label, the first assertion is hidden from the rest of the program. That assertion can be *revealed* by adding a `reveal` statement:

```
method m(i: int) {
  assert x: i == 0; // Fails
  reveal x;
  assert i == 0; // Now succeeds
}
```

or

```
method m(i: int) {
  assert x: i == 0; // Fails
  assert i == 0 by { reveal x; } // Now succeeds
}
```

At the point of the `reveal` statement, the labeled assertion is made visible and can be used in proving the second assertion. In this example there is no point to labeling an assertion and then immediately revealing it. More useful are the cases where the reveal is in an assert-by block or much later in the method body.

8.20.2. Revealing preconditions

In the same way as assertions, preconditions can be labeled. Within the body of a method, a precondition is an assumption; if the precondition is labeled then that assumption is not visible in the body of the method. A `reveal` statement naming the label of the precondition then makes the assumption visible.

Here is a toy example:

```
method m(x: int, y: int) returns (z: int)
  requires L: 0 < y
  ensures z == x+y
  ensures x < z
{
  z := x + y;
}
```

The above method will not verify. In particular, the second postcondition cannot be proved. However, if we add a `reveal L;` statement in the body of the method, then the precondition is visible and both postconditions can be proved.

One could also use this style:

```
method m(x: int, y: int) returns (z: int)
  requires L: 0 < y
  ensures z == x+y
  ensures x < z
{
  z := x + y;
  assert x < z by { reveal L; }
}
```

The reason to possibly hide a precondition is the same as the reason to hide assertions: sometimes less information is better for the solver as it helps the solver focus attention on relevant information.

Section 7 of http://leino.science/papers/krml276.html provides an extended illustration of this technique to make all the dependencies of an `assert` explicit.

8.20.3. Revealing function bodies

By default, function bodies are transparent and available for constructing proofs of assertions that use those functions. This can be changed using the `--defaul-function-opacity` commandline flag, or by using the `:{opaque}` attribute and treat it as an uninterpreted function, whose properties are just its specifications. This action limits the information available to the logical reasoning engine and may make a proof possible where there might be information overload otherwise.

But then there may be specific instances where the definition of that opaque function is needed. In that situation, the body of the function can be *revealed* using the reveal statement. Here is an example:

```
opaque function f(i: int): int { i + 1 }

method m(i: int) {
  assert f(i) == i + 1;
}
```

Without the [`opaque`] modifier, the assertion is valid; with the modifier it cannot be proved because the body of the function is not visible. However if a `reveal f();` statement is inserted before the assertion, the proof succeeds. Note that the pseudo-function-call in the `reveal` statement is written without arguments and serves to mark `f` as a function name instead of a label.

8.20.4. Revealing constants

A `const` declaration can be `opaque`. If so the value of the constant is not known in reasoning about its uses, just its type and the fact that the value does not change. The constant's identifier can be listed in a reveal statement. In that case, like other revealed items, the value of the constant will be known to the reasoning engine until the end of the block containing the reveal statement.

A label or locally declared name in a method body will shadow an opaque constant with the same name outside the method body, making it unable to be revealed without using a qualified name.

8.21. Forall Statement (grammar)

Examples:

```
forall i | 0 <= i < a.Length {
  a[i] := 0;
}
forall i | 0 <= i < 100 {
  P(i); // P a lemma
}
forall i | 0 <= i < 100
```

```
  ensures i < 1000 {
}
```

The `forall` statement executes the body simultaneously for all quantified values in the specified quantifier domain. You can find more details about quantifier domains here.

There are several variant uses of the `forall` statement and there are a number of restrictions. A `forall` statement can be classified as one of the following:

- *Assign* - the `forall` statement is used for simultaneous assignment. The target must be an array element or an object field.
- *Call* - The body consists of a single call to a ghost method without side effects
- *Proof* - The `forall` has `ensure` expressions which are effectively quantified or proved by the body (if present).

An *assign* `forall` statement performs simultaneous assignment. The left-hand sides must denote different l-values, unless the corresponding right-hand sides also coincide.

The following is an excerpt of an example given by Leino in *Developing Verified Programs with Dafny*. When the buffer holding the queue needs to be resized, the `forall` statement is used to simultaneously copy the old contents into the new buffer.

```
class SimpleQueue<Data(0)>
{
  ghost var Contents: seq<Data>
  var a: array<Data>  // Buffer holding contents of queue.
  var m: int          // Index head of queue.
  var n: int          // Index just past end of queue

  method Enqueue(d: Data)
    requires a.Length > 0
    requires 0 <= m <= n <= a.Length
    modifies this, this.a
    ensures Contents == old(Contents) + [d]
  {
    if n == a.Length {
      var b := a;
      if m == 0 { b := new Data[2 * a.Length]; }
      forall i | 0 <= i < n - m {
        b[i] := a[m + i];
      }
      a, m, n := b, 0, n - m;
    }
    a[n], n, Contents := d, n + 1, Contents + [d];
  }
}
```

Here is an example of a *call* `forall` statement and the callee. This is contained in the

`CloudMake-ConsistentBuilds.dfy` test in the Dafny repository.

```
method m() {
  forall cmd', deps', e' |
       Hash(Loc(cmd', deps', e')) == Hash(Loc(cmd, deps, e)) {
    HashProperty(cmd', deps', e', cmd, deps, e);
  }
}

lemma HashProperty(cmd: Expression, deps: Expression, ext: string,
    cmd': Expression, deps': Expression, ext': string)
  requires Hash(Loc(cmd, deps, ext)) == Hash(Loc(cmd', deps', ext'))
  ensures cmd == cmd' && deps == deps' && ext == ext'
```

The following example of a *proof* `forall` statement comes from the same file:

```
forall p | p in DomSt(stCombinedC.st) && p in DomSt(stExecC.st)
  ensures GetSt(p, stCombinedC.st) == GetSt(p, stExecC.st)
{
  assert DomSt(stCombinedC.st) <= DomSt(stExecC.st);
  assert stCombinedC.st == Restrict(DomSt(stCombinedC.st),
                                               stExecC.st);
}
```

More generally, the statement

```
forall x | P(x) { Lemma(x); }
```

is used to invoke `Lemma(x)` on all `x` for which `P(x)` holds. If `Lemma(x)` ensures `Q(x)`, then the forall statement establishes

```
forall x :: P(x) ==> Q(x).
```

The `forall` statement is also used extensively in the de-sugared forms of co-predicates and co-lemmas. See datatypes.

8.22. Modify Statement (grammar)

The effect of the `modify` statement is to say that some undetermined modifications have been made to any or all of the memory locations specified by the given frame expressions. In the following example, a value is assigned to field `x` followed by a `modify` statement that may modify any field in the object. After that we can no longer prove that the field `x` still has the value we assigned to it. The now unknown values still are values of their type (e.g. of the subset type or newtype).

```
class MyClass {
  var x: int
  method N()
    modifies this
  {
    x := 18;
    modify this;
    assert x == 18;  // error: cannot conclude this here
  }
}
```

Using `...` as the argument of the statement is deprecated.

The form of the `modify` statement which includes a block statement is also deprecated.

The havoc assignment also sets a variable or field to some arbitrary (but type-consistent) value. The difference is that the havoc assignment acts on one LHS variable or memory location; the modify statement acts on all the fields of an object.

8.23. Calc Statement (grammar)

See also: Verified Calculations.

The `calc` statement supports *calculational proofs* using a language feature called *program-oriented calculations* (poC). This feature was introduced and explained in the [*Verified Calculations*] paper by Leino and Polikarpova(Leino and Polikarpova 2013). Please see that paper for a more complete explanation of the `calc` statement. We here mention only the highlights.

Calculational proofs are proofs by stepwise formula manipulation as is taught in elementary algebra. The typical example is to prove an equality by starting with a left-hand-side and through a series of transformations morph it into the desired right-hand-side.

Non-syntactic rules further restrict hints to only ghost and side-effect free statements, as well as imposing a constraint that only chain-compatible operators can be used together in a calculation. The notion of chain-compatibility is quite intuitive for the operators supported by poC; for example, it is clear that "<" and ">" cannot be used within the same calculation, as there would be no relation to conclude between the first and the last line. See the [paper][Verified Calculations] for a more formal treatment of chain-compatibility.

Note that we allow a single occurrence of the intransitive operator "!=" to appear in a chain of equalities (that is, "!=" is chain-compatible with equality but not with any other operator, including itself). Calculations with fewer than two lines are allowed, but have no effect. If a step operator is omitted, it defaults to the calculation-wide operator, defined after the `calc` keyword. If that operator is omitted, it defaults to equality.

Here is an example using `calc` statements to prove an elementary algebraic identity. As it turns out, Dafny is able to prove this without the `calc` statements, but the example illustrates the syntax.

```
lemma docalc(x : int, y: int)
  ensures (x + y) * (x + y) == x * x + 2 * x * y + y * y
{
  calc {
    (x + y) * (x + y);
    ==
    // distributive law: (a + b) * c == a * c + b * c
    x * (x + y) + y * (x + y);
    ==
    // distributive law: a * (b + c) == a * b + a * c
    x * x + x * y + y * x + y * y;
    ==
    calc {
        y * x;
      ==
        x * y;
    }
    x * x + x * y + x * y + y * y;
    ==
    calc {
      x * y + x * y;
      ==
      // a = 1 * a
      1 * x * y + 1 * x * y;
      ==
      // Distributive law
      (1 + 1) * x * y;
      ==
      2 * x * y;
    }
    x * x + 2 * x * y + y * y;
  }
}
```

Here we started with `(x + y) * (x + y)` as the left-hand-side expressions and gradually transformed it using distributive, commutative and other laws into the desired right-hand-side.

The justification for the steps are given as comments or as nested `calc` statements that prove equality of some sub-parts of the expression.

The `==` operators show the relation between the previous expression and the next. Because of the transitivity of equality we can then conclude that the original left-hand-side is equal to the final expression.

We can avoid having to supply the relational operator between every pair of expressions by giving a default operator between the `calc` keyword and the opening brace as shown in this abbreviated version of the above calc statement:

```
lemma docalc(x : int, y: int)
  ensures (x + y) * (x + y) == x * x + 2 * x * y + y * y
{
  calc == {
    (x + y) * (x + y);
    x * (x + y) + y * (x + y);
    x * x + x * y + y * x + y * y;
    x * x + x * y + x * y + y * y;
    x * x + 2 * x * y + y * y;
  }
}
```

And since equality is the default operator, we could have omitted it after the `calc` keyword. The purpose of the block statements or the `calc` statements between the expressions is to provide hints to aid Dafny in proving that step. As shown in the example, comments can also be used to aid the human reader in cases where Dafny can prove the step automatically.

9. Expressions

Dafny expressions come in three flavors. - The bulk of expressions have no side-effects and can be used within methods, functions, and specifications, and in either compiled or ghost code. - Some expressions, called right-hand-side expressions, do have side-effects and may only be used in specific syntactic locations, such as the right-hand-side of update (assignment) statements; object allocation and method calls are two typical examples of right-hand-side expressions. Note that method calls are syntactically indistinguishable from function calls; both are Expressions (PrimaryExpressions with an ArgumentList suffix). However, method calls are semantically permitted only in right-hand-side expression locations. - Some expressions are allowed only in specifications and other ghost code, as listed here.

The grammar of Dafny expressions follows a hierarchy that reflects the precedence of Dafny operators. The following table shows the Dafny operators and their precedence in order of increasing binding power.

operator	precedence	description
`;`	0	That is LemmaCall; Expression
`<==>`	1	equivalence (if and only if)
`==>`	2	implication (implies)
`<==`	2	reverse implication (follows from)
`&&`, `&`	3	conjunction (and)
`||`, `|`	3	disjunction (or)
`==`	4	equality
`==#[k]`	4	prefix equality (coinductive)
`!=`	4	disequality
`!=#[k]`	4	prefix disequality (coinductive)
`<`	4	less than
`<=`	4	at most
`>=`	4	at least
`>`	4	greater than
`in`	4	collection membership
`!in`	4	collection non-membership
`!!`	4	disjointness
`<<`	5	left-shift
`>>`	5	right-shift

operator	precedence	description
`+`	6	addition (plus)
`-`	6	subtraction (minus)
`*`	7	multiplication (times)
`/`	7	division (divided by)
`%`	7	modulus (mod)
`|`	8	bit-wise or
`&`	8	bit-wise and
`^`	8	bit-wise exclusive-or (not equal)
`as` operation	9	type conversion
`is` operation	9	type test
`-`	10	arithmetic negation (unary minus)
`!`	10	logical negation, bit-wise complement
Primary Expressions	11	

9.1. Lemma-call expressions (grammar)

Examples:

```
var a := L(a,b); a*b
```

This expression has the form `S; E`. The type of the expression is the type of `E`. `S` must be a lemma call (though the grammar appears more lenient). The lemma introduces a fact necessary to establish properties of `E`.

Sometimes an expression will fail unless some relevant fact is known. In the following example the `F_Fails` function fails to verify because the `Fact(n)` divisor may be zero. But preceding the expression by a lemma that ensures that the denominator is not zero allows function `F_Succeeds` to succeed.

```
function Fact(n: nat): nat
{
  if n == 0 then 1 else n * Fact(n-1)
}

lemma L(n: nat)
  ensures 1 <= Fact(n)
{
}

function F_Fails(n: nat): int
{
  50 / Fact(n)  // error: possible division by zero
}

function F_Succeeds(n: nat): int
{
  L(n); // note, this is a lemma call in an expression
  50 / Fact(n)
}
```

One restriction is that a lemma call in this form is permitted only in situations in which the expression itself is not terminated by a semicolon.

A second restriction is that `E` is not always permitted to contain lambda expressions, such as in the expressions that are the body of a lambda expression itself, function, method and iterator specifications, and if and while statements with guarded alternatives.

A third restriction is that `E` is not always permitted to contain a bit-wise or (`|`) operator, because it would be ambiguous with the vertical bar used in comprehension expressions.

Note that the effect of the lemma call only extends to the succeeding expression `E` (which may be another `;` expression).

9.2. Equivalence Expressions (grammar)

Examples:

```
A
A <==> B
A <==> C ==> D <==> B
```

An Equivalence Expression that contains one or more `<==>`s is a boolean expression and all the operands must also be boolean expressions. In that case each `<==>` operator tests for logical equality which is the same as ordinary equality (but with a different precedence).

See Section 5.2.1.1 for an explanation of the `<==>` operator as compared with the `==` operator.

The `<==>` operator is commutative and associative: `A <==> B <==> C` and `(A <==> B) <==> C` and `A <==> (B <==> C)` and `C <==> B <==> A` are all equivalent and are all true iff an even number of operands are false.

9.3. Implies or Explies Expressions (grammar)

Examples:

```
A ==> B
A ==> B ==> C ==> D
B <== A
```

See Section 5.2.1.3 for an explanation of the `==>` and `<==` operators.

9.4. Logical Expressions (grammar)

Examples:

```
A && B
A || B
&& A && B && C
```

Note that the Dafny grammar allows a conjunction or disjunction to be *prefixed* with `&&` or `||` respectively. This form simply allows a parallel structure to be written:

```
method m(x: object?, y:object?, z: object?) {
  var b: bool :=
    && x != null
    && y != null
    && z != null
    ;
}
```

This is purely a syntactic convenience allowing easy edits such as reordering lines or commenting out lines without having to check that the infix operators are always where they should be.

Note also that `&&` and `||` cannot be mixed without using parentheses: `A && B || C` is not permitted. Write `(A && B) || C` or `A && (B || C)` instead.

See Section 5.2.1.2 for an explanation of the `&&` and `||` operators.

9.5. Relational Expressions (grammar)

Examples:

```
x == y
x != y
x < y
x >= y
x in y
x ! in y
x !! y
x ==#[k] y
```

The relation expressions compare two or more terms. As explained in the section about basic types, `==`, `!=`, `<`, `>`, `<=`, and `>=` are *chaining*.

The `in` and `!in` operators apply to collection types as explained in Section 5.5 and represent membership or non-membership respectively.

The `!!` represents disjointness for sets and multisets as explained in Section 5.5.1 and Section 5.5.2.

`x ==#[k] y` is the prefix equality operator that compares coinductive values for equality to a nesting level of k, as explained in the section about co-equality.

9.6. Bit Shifts (grammar)

Examples:

```
k << 5
j >> i
```

These operators are the left and right shift operators for bit-vector values. They take a bit-vector value and an `int`, shifting the bits by the given amount; the result has the same bit-vector type as the LHS. For the expression to be well-defined, the RHS value must be in the range 0 to the number of bits in the bit-vector type, inclusive.

The operations are left-associative: `a << i >> j` is `(a << i) >> j`.

9.7. Terms (grammar)

Examples:

```
x + y - z
```

`Terms` combine `Factors` by adding or subtracting. Addition has these meanings for different types:

- arithmetic addition for numeric types (Section 5.2.2])
- union for sets and multisets (Section 5.5.1 and Section 5.5.2)
- concatenation for sequences (Section 5.5.3)
- map merging for maps (Section 5.5.4)

Subtraction is

- arithmetic subtraction for numeric types
- set or multiset subtraction for sets and multisets
- domain subtraction for maps.

All addition operations are associative. Arithmetic addition and union are commutative. Subtraction is neither; it groups to the left as expected: `x - y -z` is `(x - y) -z`.

9.8. Factors (grammar)

Examples:

```
x * y
x / y
x % y
```

A `Factor` combines expressions using multiplication, division, or modulus. For numeric types these are explained in Section 5.2.2. As explained there, `/` and `%` on `int` values represent *Euclidean* integer division and modulus and not the typical C-like programming language operations.

Only `*` has a non-numeric application. It represents set or multiset intersection as explained in Section 5.5.1 and Section 5.5.2.

`*` is commutative and associative; `/` and `%` are neither but do group to the left.

9.9. Bit-vector Operations (grammar)

Examples:

```
x | y
x & y
x ^ y
```

These operations take two bit-vector values of the same type, returning a value of the same type. The operations perform bit-wise *or* (`|`), *and* (`&`), and *exclusive-or* (`^`). To perform bit-wise equality, use `^` and `!` (unary complement) together. (`==` is boolean equality of the whole bit-vector.)

These operations are associative and commutative but do not associate with each other. Use parentheses: `a & b | c` is illegal; use `(a & b) | c` or `a & (b | c)` instead.

Bit-vector operations are not allowed in some contexts. The `|` symbol is used both for bit-wise or and as the delimiter in a cardinality expression: an ambiguity arises if the expression E in `| E |` contains a `|`. This situation is easily remedied: just enclose E in parentheses, as in `|(E)|`. The only type-correct way this can happen is if the expression is a comprehension, as in `| set x: int :: x | 0x101 |`.

9.10. As (Conversion) and Is (type test) Expressions (grammar)

Examples:

```
e as MyClass
i as bv8
e is MyClass
```

The `as` expression converts the given LHS to the type stated on the RHS, with the result being of the given type. The following combinations of conversions are permitted:

- Any type to itself
- Any int or real based numeric type or bit-vector type to another int or real based numeric type or bit-vector type
- Any base type to a subset or newtype with that base

- Any subset or newtype to its base type or a subset or newtype of the same base
- Any type to a subset or newtype that has the type as its base
- Any trait to a class or trait that extends (perhaps recursively) that trait
- Any class or trait to a trait extended by that class or trait

Some of the conversions above are already implicitly allowed, without the `as` operation, such as from a subset type to its base. In any case, it must be able to be proved that the value of the given expression is a legal value of the given type. For example, `5 as MyType` is permitted (by the verifier) only if `5` is a legitimate value of`MyType` (which must be a numeric type).

The `as` operation is like a grammatical suffix or postfix operation. However, note that the unary operations bind more tightly than does `as`. That is `- 5 as nat` is `(- 5) as nat` (which fails), whereas `a * b as nat` is `a * (b as nat)`. On the other hand, `- a[4]` is `- (a[4])`.

The `is` expression is grammatically similar to the `as` expression, with the same binding power. The `is` expression is a type test that returns a `bool` value indicating whether the LHS expression is a legal value of the RHS type. The expression can be used to check whether a trait value is of a particular class type. That is, the expression in effect checks the allocated type of a trait.

The RHS type of an `is` expression can always be a supertype of the type of the LHS expression, in which case the result is trivally true. Other than that, the RHS must be based on a reference type and the LHS expression must be assignable to the RHS type. Furthermore, in order to be compilable, the RHS type must not be a subset type other than a non-null reference type, and the type parameters of the RHS must be uniquely determined from the type parameters of the LHS type. The last restriction is designed to make it possible to perform type tests without inspecting type parameters at run time. For example, consider the following types:

```
trait A { }
trait B<X> { }
class C<Y> extends B<Y> { }
class D<Y(==)> extends B<set<Y>> { }
class E extends B<int> { }
class F<Z> extends A { }
```

A LHS expression of type `B<set<int>>` can be used in a type test where the RHS is `B<set<int>>`, `C<set<int>>`, or `D<int>`, and a LHS expression of type `B<int>` can be used in a type test where the RHS is `B<int>`, `C<int>`, or `E`. Those are always allowed in compiled (and ghost) contexts. For an expression `a` of type `A`, the expression `a is F<int>` is a ghost expression; it can be used in ghost contexts, but not in compiled contexts.

For an expression `e` and type `t`, `e is t` is the condition determining whether `e as t` is well-defined (but, as noted above, is not always a legal expression).

The repertoire of types allowed in `is` tests may be expanded in the future.

9.11. Unary Expressions (grammar)

Examples:

```
-x
- - x
! x
```

A unary expression applies

- logical complement (`!` – Section 5.2.1),
- bit-wise complement (`!` – Section 5.2.3),
- numeric negation (`-` – Section 5.2.2), or
- bit-vector negation (`-` – Section 5.2.3)

to its operand.

9.12. Primary Expressions (grammar)

Examples:

```
true
34
M(i,j)
b.c.d
[1,2,3]
{2,3,4}
map[1 => 2, 3 => 4]
(i:int,j:int)=>i+j
if b then 4 else 5
```

After descending through all the binary and unary operators we arrive at the primary expressions, which are explained in subsequent sections. A number of these can be followed by 0 or more suffixes to select a component of the value.

9.13. Lambda expressions (grammar)

Examples:

```
x => -x
_ => true
(x,y) => x*y
(x:int, b:bool) => if b then x else -x
x requires x > 0 => x-1
```

See Section 7.4 for a description of specifications for lambda expressions.

In addition to named functions, Dafny supports expressions that define functions. These are called *lambda (expression)s* (some languages know them as *anonymous functions*). A lambda expression has the form:

```
( _params_ ) _specification_ => _body_
```

where *params* is a comma-delimited list of parameter declarations, each of which has the form `x` or `x: T`. The type `T` of a parameter can be omitted when it can be inferred. If the identifier `x` is not needed, it can be replaced by `_`. If *params* consists of a single parameter `x` (or `_`) without an explicit type, then the parentheses can be dropped; for example, the function that returns the successor of a given integer can be written as the following lambda expression:

```
x => x + 1
```

The *specification* is a list of clauses `requires E` or `reads W`, where `E` is a boolean expression and `W` is a frame expression.

body is an expression that defines the function's return value. The body must be well-formed for all possible values of the parameters that satisfy the precondition (just like the bodies of named functions and methods). In some cases, this means it is necessary to write explicit `requires` and `reads` clauses. For example, the lambda expression

```
x requires x != 0 => 100 / x
```

would not be well-formed if the `requires` clause were omitted, because of the possibility of division-by-zero.

In settings where functions cannot be partial and there are no restrictions on reading the heap, the *eta expansion* of a function `F: T -> U` (that is, the wrapping of `F` inside a lambda expression in such a way that the lambda expression is equivalent to `F`) would be written `x => F(x)`. In Dafny, eta expansion must also account for the precondition and reads set of the function, so the eta expansion of `F` looks like:

```
x requires F.requires(x) reads F.reads(x) => F(x)
```

9.14. Left-Hand-Side Expressions (grammar)

Examples:

```
x
a[k]
LibraryModule.F().x
old(o.f).x
```

A left-hand-side expression is only used on the left hand side of an Update statement or an Update with Failure Statement.

An LHS can be

- a simple identifier: `k`
- an expression with a dot suffix: `this.x`, `f(k).y`
- an expression with an array selection: `a[k]`, `f(a8)[6]`

9.15. Right-Hand-Side Expressions (grammar)

Examples:

```
new int[6]
new MyClass
new MyClass(x,y,z)
x+y+z
*
```

A Right-Hand-Side expression is an expression-like construct that may have side-effects. Consequently such expressions can only be used within certain statements within methods, and not as general expressions or within functions or specifications.

An RHS is either an array allocation, an object allocation, a havoc right-hand-side, a method call, or a simple expression, optionally followed by one or more attributes.

Right-hand-side expressions (that are not just regular expressions) appear in the following constructs:

- return statements,
- yield statements,
- update statements,
- update-with-failure statements, or
- variable declaration statements.

These are the only contexts in which arrays or objects may be allocated or in which havoc may be stipulated.

9.16. Array Allocation (grammar)

Examples:

```
new int[5,6]
new int[5][2,3,5,7,11]
new int[][2,3,5,7,11]
new int[5](i => i*i)
new int[2,3]((i,j) => i*j)
```

This right-hand-side expression allocates a new single or multi-dimensional array (cf. Section 5.10). The initialization portion is optional. One form is an explicit list of values, in which case the dimension is optional:

```
var a := new int[5];
var b := new int[5][2,3,5,7,11];
var c := new int[][2,3,5,7,11];
var d := new int[3][4,5,6,7]; // error
```

The comprehension form requires a dimension and uses a function of type `nat -> T` where `T` is the array element type:

```
var a := new int[5](i => i*i);
```

To allocate a multi-dimensional array, simply give the sizes of each dimension. For example,

```
var m := new real[640, 480];
```

allocates a 640-by-480 two-dimensional array of `real`s. The initialization portion cannot give a display of elements like in the one-dimensional case, but it can use an initialization function. A function used to initialize a n-dimensional array requires a function from n `nat`s to a `T`, where `T` is the element type of the array. Here is an example:

```
var diag := new int[30, 30]((i, j) => if i == j then 1 else 0);
```

Array allocation is permitted in ghost contexts. If any expression used to specify a dimension or initialization value is ghost, then the `new` allocation can only be used in ghost contexts. Because the elements of an array are non-ghost, an array allocated in a ghost context in effect cannot be changed after initialization.

9.17. Object Allocation (grammar)

Examples:

```
new MyClass
new MyClass.Init
new MyClass.Init(1,2,3)
```

This right-hand-side expression allocates a new object of a class type as explained in section Class Types.

9.18. Havoc Right-Hand-Side (grammar)

Examples:

```
*
```

A havoc right-hand-side is just a `*` character. It produces an arbitrary value of its associated type. The "assign-such-that" operator (`:|`) can be used to obtain a more constrained arbitrary value. See Section 8.5.

9.19. Constant Or Atomic Expressions (grammar)

Examples:

```
this
null
5
5.5
true
'a'
"dafny"
( e )
| s |
old(x)
```

```
allocated(x)
unchanged(x)
fresh(e)
assigned(x)
```

These expressions are never l-values. They include

- literal expressions
- parenthesized expressions
- `this` expressions
- fresh expressions
- allocated expressions
- unchanged expressions
- old expressions
- cardinality expressions
- assigned expressions

9.20. Literal Expressions (grammar}

Examples:

```
5
5.5
true
'a'
"dafny"
```

A literal expression is a null object reference or a boolean, integer, real, character or string literal.

9.21. `this` Expression (grammar)

Examples:

```
this
```

The `this` token denotes the current object in the context of a constructor, instance method, or instance function.

9.22. Old and Old@ Expressions (grammar)

Examples:

```
old(c)
old@L(c)
```

An *old expression* is used in postconditions or in the body of a method or in the body or specification of any two-state function or two-state lemma; an *old* expression with a label is used only in the body of a method at a point where the label dominates its use in the expression.

`old(e)` evaluates the argument using the value of the heap on entry to the method; `old@ident(e)` evaluates the argument using the value of the heap at the given statement label.

Note that **old** and **old@** only affect heap dereferences, like `o.f` and `a[i]`. In particular, neither form has any effect on the value returned for local variables or out-parameters (as they are not on the heap).[9] If the value of an entire expression at a particular point in the method body is needed later on in the method body, the clearest means is to declare a ghost variable, initializing it to the expression in question. If the argument of `old` is a local variable or out-parameter. Dafny issues a warning.

The argument of an `old` expression may not contain nested `old`, `fresh`, or `unchanged` expressions, nor two-state functions or two-state lemmas.

Here are some explanatory examples. All `assert` statements verify to be true.

```
class A {

  var value: int

  method m(i: int)
    requires i == 6
    requires value == 42
    modifies this
  {
    var j: int := 17;
    value := 43;
    label L:
    j := 18;
    value := 44;
    label M:
    assert old(i) == 6; // i is local, but can't be changed anyway
    assert old(j) == 18; // j is local and not affected by old
    assert old@L(j) == 18; // j is local and not affected by old
    assert old(value) == 42;
    assert old@L(value) == 43;
    assert old@M(value) == 44 && this.value == 44;
    // value is this.value; 'this' is the same
    // same reference in current and pre state but the
    // values stored in the heap as its fields are different;
    // '.value' evaluates to 42 in the pre-state, 43 at L,
    // and 44 in the current state
  }
}
```

[9]The semantics of `old` in Dafny differs from similar constructs in other specification languages like ACSL or JML.

```
class A {
  var value: int
  constructor ()
     ensures value == 10
  {
     value := 10;
  }
}

class B {
   var a: A
   constructor () { a := new A(); }

   method m()
     requires a.value == 11
     modifies this, this.a
   {
     label L:
     a.value := 12;
     label M:
     a := new A(); // Line X
     label N:
     a.value := 20;
     label P:

     assert old(a.value) == 11;
     assert old(a).value == 12; // this.a is from pre-state,
                                // but .value in current state
     assert old@L(a.value) == 11;
     assert old@L(a).value == 12; // same as above
     assert old@M(a.value) == 12; // .value in M state is 12
     assert old@M(a).value == 12;
     assert old@N(a.value) == 10; // this.a in N is the heap
                                  // reference at Line X
     assert old@N(a).value == 20; // .value in current state is 20
     assert old@P(a.value) == 20;
     assert old@P(a).value == 20;
  }
}
```

```
class A {
  var value: int
  constructor ()
     ensures value == 10
  {
     value := 10;
  }
}

class B {
   var a: A
   constructor () { a := new A(); }

   method m()
     requires a.value == 11
     modifies this, this.a
   {
     label L:
     a.value := 12;
     label M:
     a := new A(); // Line X
     label N:
     a.value := 20;
     label P:

     assert old(a.value) == 11;
     assert old(a).value == 12; // this.a is from pre-state,
                                // but .value in current state
     assert old@L(a.value) == 11;
     assert old@L(a).value == 12; // same as above
     assert old@M(a.value) == 12; // .value in M state is 12
     assert old@M(a).value == 12;
     assert old@N(a.value) == 10; // this.a in N is the heap
                                  // reference at Line X
     assert old@N(a).value == 20; // .value in current state is 20
     assert old@P(a.value) == 20;
     assert old@P(a).value == 20;
  }
}
```

The next example demonstrates the interaction between `old` and array elements.

```
class A {
  var z1: array<nat>
  var z2: array<nat>

  method mm()
    requires z1.Length > 10 && z1[0] == 7
    requires z2.Length > 10 && z2[0] == 17
    modifies z2
  {
    var a: array<nat> := z1;
    assert a[0] == 7;
    a := z2;
    assert a[0] == 17;
    assert old(a[0]) == 17; // a is local with value z2
    z2[0] := 27;
    assert old(a[0]) == 17; // a is local, with current value of
                            // z2; in pre-state z2[0] == 17
    assert old(a)[0] == 27; // a is local, so old(a) has no effect
  }
}
```

9.23. Fresh Expressions (grammar)

Examples:

```
fresh(e)
fresh@L(e)
```

`fresh(e)` returns a boolean value that is true if the objects denoted by expression `e` were all freshly allocated since the time of entry to the enclosing method, or since `label L:` in the variant `fresh@L(e)`. The argument is an object or set of objects. For example, consider this valid program:

```
class C { constructor() {} }
method f(c1: C) returns (r: C)
  ensures fresh(r)
{
  assert !fresh(c1);
  var c2 := new C();
  label AfterC2:
  var c3 := new C();
  assert fresh(c2) && fresh(c3);
  assert fresh({c2, c3});
  assert !fresh@AfterC2(c2) && fresh@AfterC2(c3);
  r := c2;
}
```

The `L` in the variant `fresh@L(e)` must denote a label that, in the enclosing method's control flow, dominates the expression. In this case, `fresh@L(e)` returns `true` if the objects denoted by `e` were all freshly allocated since control flow reached label `L`.

The argument of `fresh` must be either an `object` reference or a set or sequence of object references. In this case, `fresh(e)` (respectively `fresh@L(e)` with a label) is a synonym of `old(!allocated(e))` (respectively `old@L(!allocated(e))`)

9.24. Allocated Expressions (grammar)

Examples:

```
allocated(c)
allocated({c1,c2})
```

For any expression e, the expression allocated(e) evaluates to true in a state if the value of e is available in that state, meaning that it could in principle have been the value of a variable in that state.

For example, consider this valid program:

```
class C { constructor() {} }
datatype D = Nil | Cons(C, D)
method f() {
  var d1, d2 := Nil, Nil;
  var c1 := new C();
  label L1:
  var c2 := new C();
  label L2:
  assert old(allocated(d1) && allocated(d2));
  d1 := Cons(c1, Nil);
  assert old(!allocated(d1) && allocated(d2));
  d2 := Cons(c2, Nil);
  assert old(!allocated(d1) && !allocated(d2));
  assert allocated(d1) && allocated(d2);
  assert old@L1(allocated(d1) && !allocated(d2));
  assert old@L2(allocated(d1) && allocated(d2));
  d1 := Nil;
  assert old(allocated(d1) && !allocated(d2));
}
```

This can be useful when, for example, allocated(e) is evaluated in an old state. Like in the example, where d1 is a local variable holding a datatype value Cons(c1, Nil) where c1 is an object that was allocated in the enclosing method, then old(allocated(d)) is false.

If the expression e is of a reference type, then !old(allocated(e)) is the same as fresh(e).

9.25. Unchanged Expressions (grammar)

Examples:

```
unchanged(c)
unchanged([c1,c2])
unchanged@L(c)
```

The unchanged expression returns true if and only if every reference denoted by its arguments has the same value for all its fields in the old and current state. For example, if c is an object with two fields, x and y, then unchanged(c) is equivalent to

```
c.x == old(c.x) && c.y == old(c.y)
```

Each argument to `unchanged` can be a reference, a set of references, or a sequence of references, each optionally followed by a back-tick and field name. This form with a frame field expresses that just the field `f`, not necessarily all fields, has the same value in the old and current state. If there is such a frame field, all the references must have the same type, which must have a field of that name.

The optional `@`-label says to use the state at that label as the old-state instead of using the `old` state (the pre-state of the method). That is, using the example `c` from above, the expression `unchanged@Lbl(c)` is equivalent to

```
c.x == old@Lbl(c.x) && c.y == old@Lbl(c.y)
```

Each reference denoted by the arguments of `unchanged` must be non-null and must be allocated in the old-state of the expression.

9.26. Cardinality Expressions (grammar)

Examples:

```
|s|
|s[1..i]|
```

For a finite-collection expression `c`, `|c|` is the cardinality of `c`. For a finite set or sequence, the cardinality is the number of elements. For a multiset, the cardinality is the sum of the multiplicities of the elements. For a finite map, the cardinality is the cardinality of the domain of the map. Cardinality is not defined for infinite sets or infinite maps. For more information, see Section 5.5.

9.27. Parenthesized Expressions (grammar)

A parenthesized expression is a list of zero or more expressions enclosed in parentheses.

If there is exactly one expression enclosed then the value is just the value of that expression.

If there are zero or more than one, the result is a `tuple` value. See Section 5.13.

9.28. Sequence Display Expression (grammar)

Examples:

```
[1, 2, 3]
[1]
[]
seq(k, n => n+1)
```

A sequence display expression provides a way to construct a sequence with given values. For example

```
[1, 2, 3]
```

is a sequence with three elements in it.

```
seq(k, n => n+1)
```

is a sequence of k elements whose values are obtained by evaluating the second argument (a function, in this case a lambda expression) on the indices 0 up to k.

See this section for more information on sequences.

9.29. Set Display Expression (grammar)

Examples:

```
{}
{1,2,3}
iset{1,2,3,4}
multiset{1,2,2,3,3,3}
multiset(s)
```

A set display expression provides a way of constructing a set with given elements. If the keyword `iset` is present, then a potentially infinite set (with the finite set of given elements) is constructed.

For example

```
{1, 2, 3}
```

is a set with three elements in it. See Section 5.5.1 for more information on sets.

A multiset display expression provides a way of constructing a multiset with given elements and multiplicities. For example

```
multiset{1, 1, 2, 3}
```

is a multiset with three elements in it. The number 1 has a multiplicity of 2, and the numbers 2 and 3 each have a multiplicity of 1.

A multiset cast expression converts a set or a sequence into a multiset as shown here:

```
var s : set<int> := {1, 2, 3};
var ms : multiset<int> := multiset(s);
ms := ms + multiset{1};
var sq : seq<int> := [1, 1, 2, 3];
var ms2 : multiset<int> := multiset(sq);
assert ms == ms2;
```

Note that `multiset{1, 1}` is a multiset holding the value `1` with multiplicity 2, but in `multiset({1,1})` the multiplicity is 1, because the expression `{1,1}` is the set `{1}`, which is then converted to a multiset.

See Section 5.5.2 for more information on multisets.

9.30. Map Display Expression (grammar)

Examples:

```
map[]
map[1 := "a", 2 := "b"]
imap[1 := "a", 2 := "b"]
```

A map display expression builds a finite or potentially infinite map from explicit mappings. For example:

```
const m := map[1 := "a", 2 := "b"]
ghost const im := imap[1 := "a", 2 := "b"]
```

See Section 5.5.4 for more details on maps and imaps.

9.31. Endless Expression (grammar)

Endless expression gets it name from the fact that all its alternate productions have no terminating symbol to end them, but rather they all end with an arbitrary expression at the end. The various endless expression alternatives are described in the following subsections.

9.31.1. If Expression (grammar)

Examples:

```
if c then e1 else e2
if x: int :| P(x) then x else 0
```

An *if expression* is a conditional (ternary) expression. It first evaluates the condition expression that follows the `if`. If the condition evaluates to `true` then the expression following the `then` is evaluated and its value is the result of the expression. If the condition evaluates to `false` then the expression following the `else` is evaluated and that value is the result of the expression. It is important that only the selected expression is evaluated as the following example shows.

```
var k := 10 / x; // error, may divide by 0.
var m := if x != 0 then 10 / x else 1; // ok, guarded
```

The `if` expression also permits a binding form. In this case the condition of the `if` is an existential asking "does there exist a value satisfying the given predicate?". If not, the else branch is evaluated. But if so, then an (arbitrary) value that does satisfy the given predicate is bound to the given variable and that variable is in scope in the then-branch of the expression.

For example, in the code

```
predicate P(x: int) {
  x == 5 || x == -5
}
method main() {
  assert P(5);
  var y := if x: int :| P(x) then x else 0;
  assert y == 5 || y == -5;
}
```

`x` is given some value that satisfies `P(x)`, namely either `5` or `-5`. That value of `x` is the value of the expression in the `then` branch above; if there is no value satisfying `P(x)`, then `0` is returned. Note that if `x` is declared to be a `nat` in this example, then only the value `5` would be permissible.

This binding form of the `if` expression acts in the same way as the binding form of the `if` statement.

In the example given, the binder for `x` has no constraining range, so the expression is `ghost`; if a range is given, such as `var y := if x: int :| 0 <= x < 10 && P(x) then x else 0;`, then the `if` and `y` are no longer ghost, and `y` could be used, for example, in a `print` statement.

9.31.2. Case and Extended Patterns (grammar)

Patterns are used for (possibly nested) pattern matching on inductive, coinductive or base type values. They are used in match statements, match expressions, let expressions, and variable declarations. The match expressions and statements allow literals, symbolic constants, and disjunctive ("or") patterns.

When matching an inductive or coinductive value in a match statement or expression, the pattern must correspond to one of the following:

- (0) a case disjunction ("or-pattern")
- (1) bound variable (a simple identifier),
- (2) a constructor of the type of the value,
- (3) a literal of the correct type, or
- (4) a symbolic constant.

If the extended pattern is

- a sequence of `|`-separated sub-patterns, then the pattern matches values matched by any of the sub-patterns.
- a parentheses-enclosed possibly-empty list of patterns, then the pattern matches a tuple.
- an identifier followed by a parentheses-enclosed possibly-empty list of patterns, then the pattern matches a constructor.
- a literal, then the pattern matches exactly that literal.
- a simple identifier, then the pattern matches
  - a parameter-less constructor if there is one defined with the correct type and the given name, else
  - the value of a symbolic constant, if a name lookup finds a declaration for a constant with the given name (if the name is declared but with a non-matching type, a type resolution error will occur),
  - otherwise, the identifier is a new bound variable

Disjunctive patterns may not bind variables, and may not be nested inside other patterns.

Any patterns inside the parentheses of a constructor (or tuple) pattern are then matched against the arguments that were given to the constructor when the value was constructed. The number of patterns must match the number of parameters to the constructor (or the arity of the tuple).

When matching a value of base type, the pattern should either be a literal expression of the same type as the value, or a single identifier matching all values of this type.

Patterns may be nested. The bound variable identifiers contained in all the patterns must be distinct. They are bound to the corresponding values in the value being matched. (Thus, for example, one cannot repeat a bound variable to attempt to match a constructor that has two identical arguments.)

9.31.3. Match Expression (grammar)

A *match expression* is used to conditionally evaluate and select an expression depending on the value of an algebraic type, i.e. an inductive type, a coinductive type, or a base type.

All of the variables in the patterns must be distinct. If types for the identifiers are not given then types are inferred from the types of the constructor's parameters. If types are given then they must agree with the types of the corresponding parameters.

The expression following the `match` keyword is called the *selector*. A match expression is evaluated by first evaluating the selector. The patterns of each match alternative are then compared, in order, with the resulting value until a matching pattern is found, as described in the section on case bindings. If the constructor had parameters, then the actual values used to construct the selector value are bound to the identifiers in the identifier list. The expression to the right of the `=>` in the matched alternative is then evaluated in the environment enriched by this binding. The result of that evaluation is the result of the match expression.

Note that the braces enclosing the sequence of match alternatives may be omitted. Those braces are required if lemma or lambda expressions are used in the body of any match alternative; they may also be needed for disambiguation if there are nested match expressions.

9.31.4. Quantifier Expression (grammar)

Examples:

```
forall x: int :: x > 0
forall x: nat | x < 10 :: x*x < 100
exists x: int :: x * x == 25
```

A *quantifier expression* is a boolean expression that specifies that a given expression (the one following the `::`) is true for all (for **forall**) or some (for **exists**) combination of values of the quantified variables, namely those in the given quantifier domain. See Section 2.7.4 for more details on quantifier domains.

Here are some examples:

```
assert forall x : nat | x <= 5 :: x * x <= 25;
(forall n :: 2 <= n ==> (exists d :: n < d < 2*n))
assert forall x: nat | 0 <= x < |s|, y <- s[x] :: y < x;
```

The quantifier identifiers are *bound* within the scope of the expressions in the quantifier expression.

If types are not given for the quantified identifiers, then Dafny attempts to infer their types from the context of the expressions. It this is not possible, the program is in error.

9.31.5. Set Comprehension Expressions (grammar)

Examples:

```
const c1 := set x: nat | x < 100
const c2 := set x: nat | x < 100 :: x * x
const c3 := set x: nat, y: nat | x < y < 100 :: x * y
ghost const c4 := iset x: nat | x > 100
ghost const c5: iset<int> := iset s
const c6 := set x <- c3 :: x + 1
```

A set comprehension expression is an expression that yields a set (possibly infinite only if `iset` is used) that satisfies specified conditions. There are two basic forms.

If there is only one quantified variable, the optional `"::" Expression` need not be supplied, in which case it is as if it had been supplied and the expression consists solely of the quantified variable. That is,

```
set x : T | P(x)
```

is equivalent to

```
set x : T | P(x) :: x
```

For the full form

```
var S := set x1: T1 <- C1 | P1(x1),
             x2: T2 <- C2 | P2(x1, x2),
             ...
             :: Q(x1, x2, ...)
```

the elements of `S` will be all values resulting from evaluation of `Q(x1, x2, ...)` for all combinations of quantified variables `x1, x2, ...` (from their respective `C1, C2, ...` domains) such that all predicates `P1(x1), P2(x1, x2), ...` hold.

For example,

```
var S := set x:nat, y:nat | x < y < 3 :: (x, y)
```

yields `S == {(0, 1), (0, 2), (1, 2) }`

The types on the quantified variables are optional and if not given Dafny will attempt to infer them from the contexts in which they are used in the various expressions. The `<- C` domain expressions are also optional and default to `iset x: T` (i.e. all values of the variable's type), as are the `| P` expressions which default to `true`. See also Section 2.7.4 for more details on quantifier domains.

If a finite set was specified ("set" keyword used), Dafny must be able to prove that the result is finite otherwise the set comprehension expression will not be accepted.

Set comprehensions involving reference types such as

```
set o: object
```

are allowed in ghost expressions within methods, but not in ghost functions[10]. In particular, in ghost contexts, the check that the result is finite should allow any set comprehension where the bound variable is of a reference type. In non-ghost contexts, it is not allowed, because–even though the resulting set would be finite–it is not pleasant or practical to compute at run time.

The universe in which set comprehensions are evaluated is the set of all *allocated* objects, of the appropriate type and satisfying the given predicate. For example, given

```
class I {
  var i: int
}

method test() {
  ghost var m := set x: I :: 0 <= x.i <= 10;
}
```

the set `m` contains only those instances of `I` that have been allocated at the point in program execution that `test` is evaluated. This could be no instances, one per value of `x.i` in the stated range, multiple instances of `I` for each value of `x.i`, or any other combination.

9.31.6. Statements in an Expression (grammar)

Examples:

```
assert x != 0; 10/x
assert x != 0; assert y > 0; y/x
assume x != 0; 10/x
expect x != 0; 10/x
reveal M.f; M.f(x)
calc { x * 0; == 0; } x/1;
```

A `StmtInExpr` is a kind of statement that is allowed to precede an expression in order to ensure that the expression can be evaluated without error. For example:

```
assume x != 0; 10/x
```

`Assert`, `assume`, `expect`, `reveal` and `calc` statements can be used in this way.

[10]In order to be deterministic, the result of a function should only depend on the arguments and of the objects it reads, and Dafny does not provide a way to explicitly pass the entire heap as the argument to a function. See this post for more insights.

9.31.7. Let and Let or Fail Expression (grammar)

Examples:

```
var x := f(y); x*x
var x :- f(y); x*x
var x :| P(x); x*x
var (x, y) := T(); x + y   // T returns a tuple
var R(x,y) := T(); x + y   // T returns a datatype value R
```

A `let` expression allows binding of intermediate values to identifiers for use in an expression. The start of the `let` expression is signaled by the `var` keyword. They look much like a local variable declaration except the scope of the variable only extends to the enclosed expression.

For example:

```
var sum := x + y; sum * sum
```

In the simple case, the pattern is just an identifier with optional type (which if missing is inferred from the rhs).

The more complex case allows destructuring of constructor expressions. For example:

```
datatype Stuff = SCons(x: int, y: int) | Other
function GhostF(z: Stuff): int
  requires z.SCons?
{
  var SCons(u, v) := z; var sum := u + v; sum * sum
}
```

The Let expression has a failure variant that simply uses `:-` instead of `:=`. This Let-or-Fail expression also permits propagating failure results. However, in statements (Section 8.6), failure results in immediate return from the method; expressions do not have side effects or immediate return mechanisms. Rather, if the expression to the right of `:-` results in a failure value `V`, the overall expression returns `V.PropagateFailure()`; if there is no failure, the expression following the semicolon is returned. Note that these two possible return values must have the same type (or be implicitly convertible to the same type). Typically that means that `tmp.PropagateFailure()` is a failure value and `E` is a value-carrying success value, both of the same failure-compatible type, as described in Section 8.6.

The expression `:- V; E` is desugared into the *expression*

```
var tmp := V;
if tmp.IsFailure()
then tmp.PropagateFailure()
else E
```

The expression `var v :- V; E` is desugared into the *expression*

```
var tmp := V;
if tmp.IsFailure()
```

```
then tmp.PropagateFailure()
else var v := tmp.Extract(); E
```

If the RHS is a list of expressions then the desugaring is similar. `var v, v1 :- V, V1; E` becomes

```
var tmp := V;
if tmp.IsFailure()
then tmp.PropagateFailure()
else var v, v1 := tmp.Extract(), V1; E
```

So, if tmp is a failure value, then a corresponding failure value is propagated along; otherwise, the expression is evaluated as normal.

9.31.8. Map Comprehension Expression (grammar)

Examples:

```
map x : int | 0 <= x <= 10 :: x * x;
map x : int | 0 <= x <= 10 :: -x := x * x;
imap x : int | 10 < x :: x * x;
```

A *map comprehension expression* defines a finite or infinite map value by defining a domain and for each value in the domain, giving the mapped value using the expression following the "::". See Section 2.7.4 for more details on quantifier domains.

For example:

```
function square(x : int) : int { x * x }
method test()
{
  var m := map x : int | 0 <= x <= 10 :: x * x;
  ghost var im := imap x : int :: x * x;
  ghost var im2 := imap x : int :: square(x);
}
```

Dafny finite maps must be finite, so the domain must be constrained to be finite. But imaps may be infinite as the examples show. The last example shows creation of an infinite map that gives the same results as a function.

If the expression includes the `:=` token, that token separates domain values from range values. For example, in the following code

```
method test()
{
  var m := map x : int | 1 <= x <= 10 :: 2*x := 3*x;
}
```

`m` maps `2` to `3`, `4` to `6`, and so on.

9.32. Name Segment (grammar)

Examples:

```
I
I<int,C>
I#[k]
I#<int>[k]
```

A *name segment* names a Dafny entity by giving its declared name optionally followed by information to make the name more complete. For the simple case, it is just an identifier. Note that a name segment may be followed by suffixes, including the common '.' and further name segments.

If the identifier is for a generic entity, it is followed by a `GenericInstantiation` which provides actual types for the type parameters.

To reference a prefix predicate (see Section 5.14.3.5) or prefix lemma (see Section 5.14.3.6.3), the identifier must be the name of the greatest predicate or greatest lemma and it must be followed by a *hash call*.

9.33. Hash call (grammar)

A *hash call* is used to call the prefix for a greatest predicate or greatest lemma. In the non-generic case, just insert `"#[k]"` before the call argument list where k is the number of recursion levels.

In the case where the `greatest lemma` is generic, the generic type argument is given before. Here is an example:

```
codatatype Stream<T> = Nil | Cons(head: int, stuff: T,
                                  tail: Stream<T>)

function append(M: Stream, N: Stream): Stream
{
  match M
  case Nil => N
  case Cons(t, s, M') => Cons(t, s, append(M', N))
}

function zeros<T>(s : T): Stream<T>
{
  Cons(0, s, zeros(s))
}

function ones<T>(s: T): Stream<T>
{
  Cons(1, s, ones(s))
}
```

```
greatest predicate atmost(a: Stream, b: Stream)
{
  match a
  case Nil => true
  case Cons(h,s,t) => b.Cons? && h <= b.head && atmost(t, b.tail)
}

greatest lemma {:induction false} Theorem0<T>(s: T)
  ensures atmost(zeros(s), ones(s))
{
  // the following shows two equivalent ways to state the
  // coinductive hypothesis
  if (*) {
    Theorem0#<T>[_k-1](s);
  } else {
    Theorem0(s);
  }
}
```

where the `HashCall` is `"Theorem0#<T>[_k-1](s);"`. See Section 5.14.3.5 and Section 5.14.3.6.3.

9.34. Suffix (grammar)

A *suffix* describes ways of deriving a new value from the entity to which the suffix is appended. The several kinds of suffixes are described below.

9.34.1. Augmented Dot Suffix (grammar)

Examples: (expression with suffix)

```
a.b
(a).b<int>
a.b#[k]
a.b#<int>[k]
```

An augmented dot suffix consists of a simple dot suffix optionally followed by either

- a `GenericInstantiation` (for the case where the item selected by the `DotSuffix` is generic), or
- a `HashCall` for the case where we want to call a prefix predicate or prefix lemma. The result is the result of calling the prefix predicate or prefix lemma.

9.34.2. Datatype Update Suffix (grammar)

Examples: (expression with suffix)

```
a.(f := e1, g:= e2)
a.(0 := e1)
(e).(f := e1, g:= e2)
```

A *datatype update suffix* is used to produce a new datatype value that is the same as an old datatype value except that the value corresponding to a given destructor has the specified value. In a *member binding update*, the given identifier (or digit sequence) is the name of a destructor (i.e. the formal parameter name) for one of the constructors of the datatype. The expression to the right of the := is the new value for that formal.

All of the destructors in a datatype update suffix must be for the same constructor, and if they do not cover all of the destructors for that constructor then the datatype value being updated must have a value derived from that same constructor.

Here is an example:

```
module NewSyntax {
  datatype MyDataType = MyConstructor(myint:int, mybool:bool)
                    | MyOtherConstructor(otherbool:bool)
                    | MyNumericConstructor(42:int)

  method test(datum:MyDataType, x:int)
    returns (abc:MyDataType, def:MyDataType,
             ghi:MyDataType, jkl:MyDataType)
    requires datum.MyConstructor?
    ensures abc == datum.(myint := x + 2)
    ensures def == datum.(otherbool := !datum.mybool)  // error
    ensures ghi == datum.(myint := 2).(mybool := false)
    // Resolution error: no non_destructor in MyDataType
    //ensures jkl == datum.(non_destructor := 5) // error
    ensures jkl == datum.(42 := 7)
  {
    abc := MyConstructor(x + 2, datum.mybool);
    abc := datum.(myint := x + 2);
    def := MyOtherConstructor(!datum.mybool);
    ghi := MyConstructor(2, false);
    jkl := datum.(42 := 7); // error

    assert abc.(myint := abc.myint - 2) == datum.(myint := x);
  }
}
```

9.34.3. Subsequence Suffix (grammar)

Examples: (with leading expression)

```
a[lo .. hi ]
(e)[ lo .. ]
e[ .. hi ]
e[ .. ]
```

A subsequence suffix applied to a sequence produces a new sequence whose elements are taken from a contiguous part of the original sequence. For example, expression `s[lo..hi]` for sequence `s`, and integer-based numeric bounds `lo` and `hi` satisfying `0 <= lo <= hi <= |s|`. See the section about other sequence expressions for details.

A subsequence suffix applied to an array produces a *sequence* consisting of the values of the designated elements. A concise way of converting a whole array to a sequence is to write `a[..]`.

9.34.4. Subsequence Slices Suffix (grammar)

Examples: (with leading expression)

```
a[ 0 : 2 : 3 ]
a[ e1 : e2 : e3 ]
a[ 0 : 2 : ]
```

Applying a *subsequence slices suffix* to a sequence produces a sequence of subsequences of the original sequence. See the section about other sequence expressions for details.

9.34.5. Sequence Update Suffix (grammar)

Examples:

```
s[1 := 2, 3 := 4]
```

For a sequence `s` and expressions `i` and `v`, the expression `s[i := v]` is the same as the sequence `s` except that at index `i` it has value `v`.

If the type of `s` is `seq<T>`, then `v` must have type `T`. The index `i` can have any integer- or bit-vector-based type (this is one situation in which Dafny implements implicit conversion, as if an `as int` were appended to the index expression). The expression `s[i := v]` has the same type as `s`.

9.34.6. Selection Suffix (grammar)

Examples:

```
a[9]
a[i.j.k]
```

If a selection suffix has only one expression in it, it is a zero-based index that may be used to select a single element of a sequence or from a single-dimensional array.

If a selection suffix has more than one expression in it, then it is a list of indices to index into a multi-dimensional array. The rank of the array must be the same as the number of indices.

If the selection suffix is used with an array or a sequence, then each index expression can have any integer- or bit-vector-based type (this is one situation in which Dafny implements implicit conversion, as if an `as int` were appended to the index expression).

9.34.7. Argument List Suffix (grammar)

Examples:

```
()
(a)
(a, b)
```

An argument list suffix is a parenthesized list of expressions that are the arguments to pass to a method or function that is being called. Applying such a suffix causes the method or function to be called and the result is the result of the call.

Note that method calls may only appear in right-hand-side locations, whereas function calls may appear in expressions and specifications; this distinction can be made only during name and type resolution, not by the parser.

9.35. Expression Lists (grammar)

Examples:

```
                 // empty list
a
a, b
```

An expression list is a comma-separated sequence of expressions, used, for example, as actual araguments in a method or function call or in parallel assignment.

9.36. Parameter Bindings (grammar)

Examples:

```
a
a, b
a, optimize := b
```

Method calls, object-allocation calls (`new`), function calls, and datatype constructors can be called with both positional arguments and named arguments.

Formal parameters have three ways to indicate how they are to be passed in: - nameonly: the only way to give a specific argument value is to name the parameter - positional only: these are nameless parameters (which are allowed only for datatype constructor parameters) - either positional or by name: this is the most common parameter

A parameter is either required or optional: - required: a caller has to supply an argument - optional: the parameter has a default value that is used if a caller omits passing a specific argument

The syntax for giving a positional-only (i.e., nameless) parameter does not allow a default-value expression, so a positional-only parameter is always required.

At a call site, positional arguments are not allowed to follow named arguments. Therefore, if `x` is a nameonly parameter, then there is no way to supply the parameters after `x` by position. Thus, any parameter that follows `x` must either be passed by name or have a default value. That is, if a later (in the formal parameter declaration) parameter does not have a default value, it is effectively nameonly.

Positional arguments must be given before any named arguments. Positional arguments are passed to the formals in the corresponding position. Named arguments are passed to the formal of the given name. Named arguments can be given out of order from how the corresponding formal parameters are declared. A formal declared with the modifier `nameonly` is not allowed to be passed positionally. The list of bindings for a call must provide exactly one value for every required parameter and at most one value for each optional parameter, and must never name non-existent formals. Any optional parameter that is not given a value takes on the default value declared in the callee for that optional parameter.

9.37. Assigned Expressions

Examples:

```
assigned(x)
```

For any variable, constant, out-parameter, or object field `x`, the expression `assigned(x)` evaluates to `true` in a state if `x` is definitely assigned in that state.

See Section 12.6 for more details on definite assignment.

9.38. Termination Ordering Expressions

When proving that a loop or recursive callable terminates, Dafny automatically generates a proof obligation that the sequence of expressions listed in a `decreases` clause gets smaller (in the lexicographic termination ordering) with each iteration or recursive call. Normally, this proof obligation is purely internal. However, it can be written as a Dafny expression using the `decreases to` operator.

The Boolean expression `(a, ..., b decreases to a', ..., b')` encodes this ordering. For example, the following assertions are valid:

```
method M(x: int, y: int) {
  assert (1 decreases to 0);
  assert (true, false decreases to false, true);
  assert (x, y decreases to x - 1, y);
}
```

Conversely, the following assertion is invalid:

```
method M(x: int, y: int) {
  assert (x decreases to x + 1);
}
```

Currently, `decreases to` expressions must be written in parentheses to avoid parsing ambiguities.

9.39. Compile-Time Constants

In certain situations in Dafny it is helpful to know what the value of a constant is during program analysis, before verification or execution takes place. For example, a compiler can choose an optimized representation of a `newtype` that is a subset of `int` if it knows the range of possible values of the subset type: if the range is within 0 to less than 256, then an unsigned 8-bit representation can be used.

To continue this example, suppose a new type is defined as

```
const MAX := 47
newtype mytype = x | 0 <= x < MAX*4
```

In this case, we would prefer that Dafny recognize that `MAX*4` is known to be constant with a value of `188`. The kinds of expressions for which such an optimization is possible are called *compile-time constants*. Note that the representation of `mytype` makes no difference semantically, but can affect how compiled code is represented at run time. In addition, though, using a symbolic constant (which may well be used elsewhere as well) improves the self-documentation of the code.

In Dafny, the following expressions are compile-time constants[11], recursively (that is, the arguments of any operation must themselves be compile-time constants):

- int, bit-vector, real, boolean, char and string literals
- int operations: `+ - * / %` and unary `-` and comparisons `< <= > >= == !=`
- real operations: `+ - *` and unary `-` and comparisons `< <= > >= == !=`
- bool operations: `&& || ==> <== <==> == !=` and unary `!`
- bit-vector operations: `+ - * / % << >> & | ^` and unary `! -` and comparisons `< <= > >= == !=`
- char operations: `< <= > >= == !=`
- string operations: length: `|...|`, concatenation: `+`, comparisons `< <= == !=`, indexing `[]`
- conversions between: `int real char` bit-vector
- newtype operations: newtype arguments, but not newtype results
- symbolic values that are declared `const` and have an explicit initialization value that is a compile-time constant
- conditional (if-then-else) expressions
- parenthesized expressions

[11] This set of operations that are constant-folded may be enlarged in future versions of `dafny`.

9.40. List of specification expressions

The following is a list of expressions that can only appear in specification contexts or in ghost blocks.

- Fresh expressions
- Allocated expressions
- Unchanged expressions
- Old expressions
- Assigned expressions
- Assert and calc expressions
- Hash Calls
- Termination ordering expression

10. Refinement

Refinement is the process of replacing something somewhat abstract with something somewhat more concrete. For example, in one module one might declare a type name, with no definition, such as `type T`, and then in a refining module, provide a definition. One could prove general properties about the contents of an (abstract) module, and use that abstract module, and then later provide a more concrete implementation without having to redo all of the proofs.

Dafny supports *module refinement*, where one module is created from another, and in that process the new module may be made more concrete than the previous. More precisely, refinement takes the following form in Dafny. One module declares some program entities. A second module *refines* the first by declaring how to augment or replace (some of) those program entities. The first module is called the *refinement parent*; the second is the *refining* module; the result of combining the two (the original declarations and the augmentation directives) is the *assembled* module or *refinement result*.

Syntactically, the refinement parent is a normal module declaration. The refining module declares which module is its refinement parent with the `refines` clause:

```
module P { // refinement parent
}
module M refines P { // refining module
}
```

The refinement result is created as follows.

0) The refinement result is a module within the same enclosing module as the refining module, has the same name, and in fact replaces the refining module in their shared scope.

1) All the declarations (including import and export declarations) of the parent are copied into the refinement result. These declarations are *not* re-resolved. That is, the assignment of declarations and types to syntactic names is not changed. The refinement result may exist in a different enclosing module and with a different set of imports than the refinement parent, so that if names were reresolved, the result might be different (and possibly not semantically valid). This is why Dafny does not re-resolve the names in their new context.

2) All the declarations of the refining module that have different names than the declarations in the refinement parent are also copied into the refinement result. However, because the refining module is just a set of augmentation directives and may refer to names copied from the refinement parent, resolution of names and types of the declarations copied in this step is performed in the context of the full refinement result.

3) Where declarations in the parent and refinement module have the same name, the second refines the first and the combination, a refined declaration, is the result placed in the refinement result module, to the exclusion of the declarations with the same name from the parent and refinement modules.

The way the refinement result declarations are assembled depends on the kind of declaration; the rules are described in subsections below.

So that it is clear that refinement is taking place, refining declarations have some syntactic indicator that they are refining some parent declaration. Typically this is the presence of a ... token.

10.1. Export set declarations

A refining export set declaration begins with the syntax

```
"export" Ident ellipsis
```

but otherwise contains the same `provides`, `reveals` and `extends` sections, with the ellipsis indicating that it is a refining declaration.

The result declaration has the same name as the two input declarations and the unions of names from each of the `provides`, `reveals`, and `extends` sections, respectively.

An unnamed export set declaration from the parent is copied into the result module with the name of the parent module. The result module has a default export set according to the general rules for export sets, after all of the result module's export set declarations have been assembled.

10.2. Import declarations

Aliasing import declarations are not refined. The result module contains the union of the import declarations from the two input modules. There must be no names in common among them.

Abstract import declarations (declared with `:` instead of `=`, Section 4.6) are refined. The refinement parent contains the abstract import and the refining module contains a regular aliasing import for the same name. Dafny checks that the refining import *adheres* to the abstract import.

10.3. Sub-module declarations

With respect to refinement, a nested module behaves just like a top-level module. It may be declared abstract and it may be declared to `refine` some refinement parent. If the nested module is not refining anything and not being refined, then it is copied into the refinement result like any other declaration.

Here is some example code:

```
abstract module P {
  module A { const i := 5 }
  abstract module B { type T }
}

module X refines P {
  module B' refines P.B { type T = int }
  module C { const k := 6}
}

module M {
  import X
  method m() {
    var z: X.B'.T := X.A.i + X.C.k;
  }
}
```

The refinement result of `P` and `X` contains nested modules `A`, `B'`, and `C`. It is this refinement result that is imported into `M`. Hence the names `X.B'.T`, `X.A.i` and `X.C.k` are all valid.

10.4. Const declarations

Const declarations can be refined as in the following example.

```
module A {
  const ToDefine: int
  const ToDefineWithoutType: int
  const ToGhost: int := 1
}

module B refines A {
  const ToDefine: int := 2
  const ToDefineWithoutType ... := 3
  ghost const ToGhost: int
  const NewConst: int
}
```

Formally, a child `const` declaration may refine a `const` declaration from a parent module if

- the parent has no initialization,
- the child has the same type as the parent, and
- one or both of the following holds:
  - the child has an initializing expression
  - the child is declared `ghost` and the parent is not `ghost`.

A refining module can also introduce new `const` declarations that do not exist in the refinement parent.

10.5. Method declarations

Method declarations can be refined as in the following example.

```
abstract module A {
  method ToImplement(x: int) returns (r: int)
    ensures r > x

  method ToStrengthen(x: int) returns (r: int)

  method ToDeterminize(x: int) returns (r: int)
```

```
    ensures r >= x
  {
    var y :| y >= x;
    return y;
  }

}

module B refines A {
  method ToImplement(x: int) returns (r: int)
  {
    return x + 2;
  }

  method ToStrengthen ...
    ensures r == x*2
  {
    return x*2;
  }

  method ToDeterminize(x: int) returns (r: int)
  {
    return x;
  }
}
```

Formally, a child `method` definition may refine a parent `method` declaration or definition by performing one or more of the following operations:

- provide a body missing in the parent (as in `ToImplement`),
- strengthen the postcondition of the parent method by adding one or more `ensures` clauses (as in `ToStrengthen`),
- provide a more deterministic version of a non-deterministic parent body (as in `ToDeterminize`), or

The type signature of a child method must be the same as that of the parent method it refines. This can be ensured by providing an explicit type signature equivalent to that of the parent (with renaming of parameters allowed) or by using an ellipsis (`...`) to indicate copying of the parent type signature. The body of a child method must satisfy any ensures clauses from its parent in addition to any it adds.

A refined method is allowed only if it does not invalidate any parent lemmas that mention it.

A refining module can also introduce new `method` declarations or definitions that do not exist in the refinement parent.

10.6. Lemma declarations

As lemmas are (ghost) methods, the description of method refinement from the previous section also applies to lemma refinement.

A valid refinement is one that does not invalidate any proofs. A lemma from a refinement parent must still be valid for the refinement result of any method or lemma it mentions.

10.7. Function and predicate declarations

Function (and equivalently predicate) declarations can be refined as in the following example.

```
abstract module A {
  function F(x: int): (r: int)
    ensures r > x

  function G(x: int): (r: int)
    ensures r > x
  { x + 1 }
}

module B refines A {
  function F ...
  { x + 1 }

  function G ...
    ensures r == x + 1
}
```

Formally, a child `function` (or `predicate`) definition can refine a parent `function` (or `predicate`) declaration or definition to

- provide a body missing in the parent,
- strengthen the postcondition of the parent function by adding one or more `ensures` clauses.

The relation between the type signature of the parent and child function is the same as for methods and lemmas, as described in the previous section.

A refining module can also introduce new `function` declarations or definitions that do not exist in the refinement parent.

10.8. Class, trait and iterator declarations

Class, trait, and iterator declarations are refined as follows: - If a class (or trait or iterator, respectively) `C` in a refining parent contains a member that is not matched by a same-named member in the class `C` in the refining module, or vice-versa, then that class is copied as is to the refinement result. - When there are members with the same name in the class in the refinement parent and in the refining module, then the combination occurs according to the rules for that category of member.

Here is an example code snippet:

```
abstract module P {
  class C {
    function F(): int
      ensures F() > 0
  }
}

module X refines P {
  class C ... {
    function F...
      ensures F() > 0
    { 1 }
  }
}
```

10.9. Type declarations

Types can be refined in two ways:

- Turning an abstract type into a concrete type;
- Adding members to a datatype or a newtype.

For example, consider the following abstract module:

```
abstract module Parent {
  type T
  type B = bool
  type S = s: string | |s| > 0 witness "!"
  newtype Pos = n: nat | n > 0 witness 1
  datatype Bool = True | False
}
```

In this module, type `T` is opaque and hence can be refined with any type, including class types. Types `B`, `S`, `Pos`, and `Bool` are concrete and cannot be refined further, except (for `Pos` and `Bool`) by giving them additional members or attributes (or refining their existing members, if any). Hence, the following are valid refinements:

```
module ChildWithTrait refines Parent {
  trait T {}
}

module ChildWithClass refines Parent {
  class T {}
}

module ChildWithSynonymType refines Parent {
  type T = bool
}

module ChildWithSubsetType refines Parent {
  type T = s: seq<int> | s != [] witness [0]
}

module ChildWithDataType refines Parent {
  datatype T = True | False
}

abstract module ChildWithExtraMembers refines Parent {
  newtype Pos ... {
    method Print() { print this; }
  }

  datatype Bool ... {
    function AsDafnyBool() : bool { this.True? }
  }
}
```

(The last example is marked `abstract` because it leaves `T` opaque.)

Note that datatype constructors, codatatype destructors, and newtype definitions cannot be refined: it is not possible to add or remove `datatype` constructors, nor to change destructors of a `codatatype`, nor to change the base type, constraint, or witness of a `newtype`.

When a type takes arguments, its refinement must use the same type arguments with the same type constraints and the same variance.

When a type has type constraints, these type constraints must be preserved by refinement. This means that a type declaration `type T(!new)` cannot be refined by a `class T`, for example. Similarly, a `type T(00)` cannot be refined by a subset type with a `witness *` clause.

The refinement of an abstract type with body-less members can include both a definition for the type along with a body for the member, as in this example:

```
abstract module P {
  type T3 {
    function ToString(): string
  }
}

module X refines P {
  newtype T3 = i | 0 <= i < 10 {
    function ToString... { "" }
  }
}
```

Note that type refinements are not required to include the `...` indicator that they are refining a parent type.

10.10. Statements

The refinement syntax (`...`) in statements is deprecated.

11. Attributes

Dafny allows many of its entities to be annotated with *Attributes*. Attributes are declared between `{:` and `}` like this:

```
{:attributeName "argument", "second" + "argument", 57}
```

(White-space may follow but not precede the `:` in `{:`.)

In general an attribute may have any name the user chooses. It may be followed by a comma-separated list of expressions. These expressions will be resolved and type-checked in the context where the attribute appears.

Any Dafny entity may have a list of attributes. Dafny does not check that the attributes listed for an entity are appropriate for it (which means that misspellings may go silently unnoticed).

The grammar shows where the attribute annotations may appear:

```
Attribute = "{:" AttributeName [ Expressions ] "}"
```

Dafny has special processing for some attributes[12]. Of those, some apply only to the entity bearing the attribute, while others (inherited attributes) apply to the entity and its descendants (such as nested modules, types, or declarations). The attribute declaration closest to the entity overrides those further away.

For attributes with a single boolean expression argument, the attribute with no argument is interpreted as if it were true.

11.1. Attributes on top-level declarations

11.1.1. `{:autocontracts}`

Dynamic frames (Kassios 2006; Smans et al. 2008; Smans, Jacobs, and Piessens 2009; Leino 2009) are frame expressions that can vary dynamically during program execution. AutoContracts is an experimental feature that will fill much of the dynamic-frames boilerplate into a class.

From the user's perspective, what needs to be done is simply:

- mark the class with `{:autocontracts}`
- declare a function (or predicate) called `Valid()`

AutoContracts will then:

- Declare:

```
  ghost var Repr: set<object>
```

- For function/predicate `Valid()`, insert:

[12]All entities that Dafny translates to Boogie have their attributes passed on to Boogie except for the `{:axiom}` attribute (which conflicts with Boogie usage) and the `{:trigger}` attribute which is instead converted into a Boogie quantifier *trigger*. See Section 11 of (Leino 2008b).

```
  reads this, Repr
```

- Into body of `Valid()`, insert (at the beginning of the body):

```
  this in Repr && null !in Repr
```

- and also insert, for every array-valued field `A` declared in the class:

```
  && (A != null ==> A in Repr)
```

- and for every field `F` of a class type `T` where `T` has a field called `Repr`, also insert:

```
  (F != null ==> F in Repr && F.Repr <= Repr && this !in F.Repr)
```

Except, if A or F is declared with `{:autocontracts false}`, then the implication will not be added.

- For every constructor, add:

```
  modifies this
  ensures Valid() && fresh(Repr - {this})
```

- At the end of the body of the constructor, add:

```
  Repr := {this};
  if (A != null) { Repr := Repr + {A}; }
  if (F != null) { Repr := Repr + {F} + F.Repr; }
```

- For every method, add:

```
  requires Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr))
```

- At the end of the body of the method, add:

```
  if (A != null) { Repr := Repr + {A}; }
  if (F != null) { Repr := Repr + {F} + F.Repr; }
```

11.1.2. `{:nativeType}`

The `{:nativeType}` attribute is only recognized by a `newtype` declaration where the base type is an integral type or a real type. For example:

```
newtype {:nativeType "byte"} ubyte = x : int | 0 <= x < 256
newtype {:nativeType "byte"} bad_ubyte = x : int | 0 <= x < 257 // Fails
```

It can take one of the following forms:

- `{:nativeType}` - With no parameters it has no effect and the declaration will have its default behavior, which is to choose a native type that can hold any value satisfying the constraints, if possible, and otherwise to use BigInteger.

- `{:nativeType true}` - Also gives default behavior, but gives an error if the base type is not integral.
- `{:nativeType false}` - Inhibits using a native type. BigInteger is used.
- `{:nativeType "typename"}` - This form has an native integral type name as a string literal. Acceptable values are:
  - `"byte"` 8 bits, unsigned
  - `"sbyte"` 8 bits, signed
  - `"ushort"` 16 bits, unsigned
  - `"short"` 16 bits, signed
  - `"uint"` 32 bits, unsigned
  - `"int"` 32 bits, signed
  - `"number"` 53 bits, signed
  - `"ulong"` 64 bits, unsigned
  - `"long"` 64 bits, signed

  If the target compiler does not support a named native type X, then an error is generated. Also, if, after scrutinizing the constraint predicate, the compiler cannot confirm that the type's values will fit in X, an error is generated. The names given above do not have to match the names in the target compilation language, just the characteristics of that type.

11.1.3. `{:ignore}` (deprecated)

Ignore the declaration (after checking for duplicate names).

11.1.4. `{:extern}`

`{:extern}` is a target-language dependent modifier used

- to alter the `CompileName` of entities such as modules, classes, methods, etc.,
- to alter the `ReferenceName` of the entities,
- to decide how to define external abstract types,
- to decide whether to emit target code or not, and
- to decide whether a declaration is allowed not to have a body.

The `CompileName` is the name for the entity when translating to one of the target languages. The `ReferenceName` is the name used to refer to the entity in the target language. A common use case of `{:extern}` is to avoid name clashes with existing library functions.

`{:extern}` takes 0, 1, or 2 (possibly empty) string arguments:

- `{:extern}`: Dafny will use the Dafny-determined name as the `CompileName` and not affect the `ReferenceName`
- `{:extern s1}`: Dafny will use `s1` as the `CompileName`, and replaces the last portion of the `ReferenceName` by `s1`. When used on an abstract type, s1 is used as a hint as to how to declare that type when compiling.
- `{:extern s1, s2}` Dafny will use `s2` as the `CompileName`. Dafny will use a combination of `s1` and `s2` such as for example `s1.s2` as the `ReferenceName` It may also be the case that one of the arguments is simply ignored.

Dafny does not perform sanity checks on the arguments—it is the user's responsibility not to generate malformed target code.

For more detail on the use of `{:extern}`, see the corresponding section in the user's guide.

11.1.5. `{:disable-nonlinear-arithmetic}`

This attribute only applies to module declarations. It overrides the global option `--disable-nonlinear-arithmetic` for that specific module. The attribute can be given true or false to disable or enable nonlinear arithmetic. When no value is given, the default value is true.

11.2. Attributes on functions and methods

11.2.1. `{:abstemious}`

The `{:abstemious}` attribute is appropriate for functions on codatatypes. If appropriate to a function, the attribute can aid in proofs that the function is *productive*. See the section on abstemious functions for more description.

11.2.2. `{:autoReq}`

For a function declaration, if this attribute is set true at the nearest level, then its `requires` clause is strengthened sufficiently so that it may call the functions that it calls.

For following example

```
function f(x:int) : bool
  requires x > 3
{
  x > 7
}

// Should succeed thanks to auto_reqs
function {:autoReq} g(y:int, b:bool) : bool
{
  if b then f(y + 2) else f(2*y)
}
```

the `{:autoReq}` attribute causes Dafny to deduce a `requires` clause for g as if it had been declared

```
function f(x:int) : bool
  requires x > 3
{
  x > 7
}
function g(y:int, b:bool) : bool
  requires if b then y + 2 > 3 else 2 * y > 3
```

```
{
  if b then f(y + 2) else f(2*y)
}
```

11.2.3. `{:autoRevealDependencies k}`

When setting `--default-function-opacity` to `autoRevealDependencies`, the `{:autoRevealDependencies k}` attribute can be set on methods and functions to make sure that only function dependencies of depth `k` in the call-graph or less are revealed automatically. As special cases, one can also use `{:autoRevealDependencies false}` (or `{:autoRevealDependencies 0}`) to make sure that no dependencies are revealed, and `{:autoRevealDependencies true}` to make sure that all dependencies are revealed automatically.

For example, when the following code is run with `--default-function-opacity` set to `autoRevealDependencies`, the function `p()` should verify and `q()` should not.

```
   function t1() : bool { true }

   function t2() : bool { t1() }

   function {:autoRevealDependencies 1} p() : (r: bool)
     ensures r
   { t1() }

   function {:autoRevealDependencies 1} q() : (r: bool)
     ensures r
   { t2() }
```

11.2.4. `{:axiom}`

The `{:axiom}` attribute may be placed on a function or method. It means that the post-condition may be assumed to be true without proof. In that case also the body of the function or method may be omitted.

The `{:axiom}` attribute only prevents Dafny from verifying that the body matches the post-condition. Dafny still verifies the well-formedness of pre-conditions, of post-conditions, and of the body if provided. To prevent Dafny from running all these checks, one would use `{:verify false}`, which is not recommended.

The compiler will still emit code for an `{:axiom}`, if it is a `function`, a `method` or a `function by method` with a body.

11.2.5. `{:compile}`

The `{:compile}` attribute takes a boolean argument. It may be applied to any top-level declaration. If that argument is false, then that declaration will not be compiled at all. The difference with `{:extern}` is that `{:extern}` will still emit declaration code if necessary, whereas `{:compile false}` will just ignore the declaration for compilation purposes.

11.2.6. `{:concurrent}`

The `{:concurrent}` attribute indicates that the compiled code for a function or method may be executed concurrently. While Dafny is a sequential language and does not support any native concepts for spawning or controlling concurrent execution, it does support restricting the specification of declarations such that it is safe to execute them concurrently using integration with the target language environment.

Currently, the only way to satisfy this requirement is to ensure that the specification of the function or method includes the equivalent of `reads {}` and `modifies {}`. This ensures that the code does not read or write any shared mutable state, although it is free to read and write newly allocated objects.

11.2.7. `{:extern <name>}`

See `{:extern <name>}`.

11.2.8. `{:fuel X}`

The fuel attribute is used to specify how much "fuel" a function should have, i.e., how many times the verifier is permitted to unfold its definition. The `{:fuel}` annotation can be added to the function itself, in which case it will apply to all uses of that function, or it can be overridden within the scope of a module, function, method, iterator, calc, forall, while, assert, or assume. The general format is:

```
{:fuel functionName,lowFuel,highFuel}
```

When applied as an annotation to the function itself, omit functionName. If highFuel is omitted, it defaults to lowFuel + 1.

The default fuel setting for recursive functions is 1,2. Setting the fuel higher, say, to 3,4, will give more unfoldings, which may make some proofs go through with less programmer assistance (e.g., with fewer assert statements), but it may also increase verification time, so use it with care. Setting the fuel to 0,0 is similar to making the definition opaque, except when used with all literal arguments.

11.2.9. `{:id <string>}`

Assign a custom unique ID to a function or a method to be used for verification result caching.

11.2.10. `{:induction}`

The `{:induction}` attribute controls the application of proof by induction to two contexts. Given a list of variables on which induction might be applied, the `{:induction}` attribute selects a sub-list of those variables (in the same order) to which to apply induction.

Dafny issue 34 proposes to remove the restriction that the sub-list be in the same order, and would apply induction in the order given in the `{:induction}` attribute.

The two contexts are:

- A method, in which case the bound variables are all the in-parameters of the method.
- A quantifier expression, in which case the bound variables are the bound variables of the quantifier expression.

The form of the `{:induction}` attribute is one of the following:

- `{:induction}` – apply induction to all bound variables
- `{:induction false}` – suppress induction, that is, don't apply it to any bound variable
- `{:induction L}` where `L` is a list consisting entirely of bound variables – apply induction to the specified bound variables
- `{:induction X}` where `X` is anything else – treat the same as `{:induction}`, that is, apply induction to all bound variables. For this usage conventionally `X` is `true`.

Here is an example of using it on a quantifier expression:

```
datatype Unary = Zero | Succ(Unary)

function UnaryToNat(n: Unary): nat {
  match n
  case Zero => 0
  case Succ(p) => 1 + UnaryToNat(p)
}

function NatToUnary(n: nat): Unary {
  if n == 0 then Zero else Succ(NatToUnary(n - 1))
}

lemma Correspondence()
  ensures forall n: nat {:induction n} :: UnaryToNat(NatToUnary(n)) == n
{
}
```

11.2.11. {:only}

`method {:only} X() {}` or `function {:only} X() {}` temporarily disables the verification of all other non-`{:only}` members, e.g. other functions and methods, in the same file, even if they contain assertions with `{:only}`.

```
method {:only} TestVerified() {
  assert true;                  // Unchecked
  assert {:only} true by {      // Checked
    assert true;                // Checked
  }
  assert true;                  // Unchecked
}

method TestUnverified() {
  assert true;                  // Unchecked
  assert {:only} true by {       // Unchecked because of {:only} Test()
    assert true;                 // Unchecked
  }
  assert true;                   // Unchecked
}
```

{:only} can help focusing on a particular member, for example a lemma or a function, as it simply disables the verification of all other lemmas, methods and functions in the same file. It's equivalent to adding {:verify false} to all other declarations simulatenously on the same file. Since it's meant to be a temporary construct, it always emits a warning.

More information about the Boogie implementation of {:opaque} is here.

11.2.12. {:print}

This attribute declares that a method may have print effects, that is, it may use `print` statements and may call other methods that have print effects. The attribute can be applied to compiled methods, constructors, and iterators, and it gives an error if applied to functions or ghost methods. An overriding method is allowed to use a {:print} attribute only if the overridden method does. Print effects are enforced only with `--track-print-effects`.

11.2.13. {:priority}

{:priority N} assigns a positive priority 'N' to a method or function to control the order in which methods or functions are verified (default: N = 1).

11.2.14. {:resource_limit} and {:rlimit}

{:resource_limit N} limits the verifier resource usage to verify the method or function to N.

This is the per-method equivalent of the command-line flag `/rlimit:N` or `--resource-limit N`. If using {:isolate_assertions} as well, the limit will be set for each assertion.

The attribute {:rlimit N} is also available, and limits the verifier resource usage to verify the method or function to `N * 1000`. This version is deprecated, however.

To give orders of magnitude about resource usage, here is a list of examples indicating how many resources are used to verify each method:

- 8K resource usage

  ```
  method f() {
    assert true;
  }
  ```

- 10K resource usage using assertions that do not add assumptions:

  ```
  method f(a: bool, b: bool) {
    assert a: (a ==> b) <==> (!b ==> !a);
    assert b: (a ==> b) <==> (!b ==> !a);
  ```

```
  assert c: (a ==> b) <==> (!b ==> !a);
  assert d: (a ==> b) <==> (!b ==> !a);
}
```

- 40K total resource usage using `{:isolate_assertions}`

```
method {:isolate_assertions} f(a: bool, b: bool) {
  assert a: (a ==> b) <==> (!b ==> !a);
  assert b: (a ==> b) <==> (!b ==> !a);
  assert c: (a ==> b) <==> (!b ==> !a);
  assert d: (a ==> b) <==> (!b ==> !a);
}
```

- 37K total resource usage and thus fails with `out of resource`.

```
method {:rlimit 30} f(a: int, b: int, c: int) {
  assert ((1 + a*a)*c) / (1 + a*a) == c;
}
```

Note that, the default solver Z3 tends to overshoot by `7K` to `8K`, so if you put `{:rlimit 20}` in the last example, the total resource usage would be `27K`.

11.2.15. `{:selective_checking}`

Turn all assertions into assumptions except for the ones reachable from after the assertions marked with the attribute `{:start_checking_here}`. Thus, `assume {:start_checking_here} something;` becomes an inverse of `assume false;`: the first one disables all verification before it, and the second one disables all verification after.

11.2.16. `{:tailrecursion}`

This attribute is used on method or function declarations. It has a boolean argument.

If specified with a `false` value, it means the user specifically requested no tail recursion, so none is done.

If specified with a `true` value, or if no argument is specified, then tail recursive optimization will be attempted subject to the following conditions:

- It is an error if the method is a ghost method and tail recursion was explicitly requested.
- Only direct recursion is supported, not mutually recursive methods.
- If `{:tailrecursion true}` was specified but the code does not allow it, an error message is given.

If you have a stack overflow, it might be that you have a function on which automatic attempts of tail recursion failed, but for which efficient iteration can be implemented by hand. To do this, use a function by method and define the loop in the method yourself, proving that it implements the function.

Using a function by method to implement recursion can be tricky. It usually helps to look at the result of the function on two to three iterations, without simplification, and see what should be the first computation. For example, consider the following tail-recursion implementation:

```
datatype Result<V,E> = Success(value: V) | Failure(error: E)

function f(x: int): Result<int, string>

//  {:tailrecursion true}  Not possible here
function MakeTailRec(
  obj: seq<int>
): Result<seq<int>, string>
{
  if |obj| == 0 then Success([])
  else
    var tail := MakeTailRec(obj[1..]);
    var r := f(obj[0]);
    if r.Failure? then
      Failure(r.error)
    else if tail.Failure? then
      tail
    else
      Success([r.value] + tail.value)
} by method {
  var i: nat := |obj|;
  var tail := Success([]); // Base case
  while i != 0
    decreases i
    invariant tail == MakeTailRec(obj[i..])
  {
    i := i - 1;
    var r := f(obj[i]);
    if r.Failure? {
      tail := Failure(r.error);
    } else if tail.Success? {
      tail := Success([r.value] + tail.value);
    } else {
    }
  }
  return tail;
}
```

The rule of thumb to unroll a recursive call into a sequential one is to look at how the result would be computed if the operations were not simplified. For example, unrolling the function on `[1, 2, 3]` yields the result `Success([f(1).value] + ([f(2).value] + ([f(3).value] + [])))`. If you had to compute this expression manually, you'd start with `([f(3).value] + [])`, then add `[f(2).value]` to the left, then `[f(1).value]`. This is why the method loop iterates with the objects from the end, and why the intermediate invariants are all about proving `tail == MakeTailRec(obj[i..])`, which makes verification succeed easily because we replicate exactly the behavior of `MakeTailRec`. If we were not interested in the first error but the last one, a possible optimization would be, on the first error, to finish iterate with a ghost loop that is not executed.

Note that the function definition can be changed by computing the tail closer to where it's used or switching the order of computing `r` and `tail`, but the `by method` body can stay the same.

11.2.17. `{:test}`

This attribute indicates the target function or method is meant to be executed at runtime in order to test that the program is working as intended.

There are two different ways to dynamically test functionality in a test:

1. A test can optionally return a single value to indicate success or failure. If it does, this must be a *failure-compatible* type just as the update-with-failure statement requires. That is, the returned type must define a `IsFailure()` function method. If `IsFailure()` evaluates to `true` on the return value, the test will be marked a failure, and this return value used as the failure message.
2. Code in the control flow of the test can use `expect` statements to dynamically test if a boolean expression is true, and cause the test to halt if not (but not the overall testing process). The optional second argument to a failed `expect` statement will be used as the test failure message.

Note that the `expect` keyword can also be used to form "assign or halt" statements such as `var x :- expect CalculateX();`, which is a convenient way to invoke a method that may produce a failure within a test without having to return a value from the test.

There are also two different approaches to executing all tests in a program:

1. By default, the compiler will mark each compiled method as necessary so that a designated target language testing framework will discover and run it. This is currently only implemented for C#, using the xUnit `[Fact]` annotation.
2. If `dafny test` is used, Dafny will instead produce a main method that invokes each test and prints the results. This runner is currently very basic, but avoids introducing any additional target language dependencies in the compiled code.

A method marked `{:test}` may not have any input arguments. If there is an output value that does not have a failure-compatible type, that value is ignored. A method that does have input arguments can be wrapped in a test harness that supplies input arguments but has no inputs of its own and that checks any output values, perhaps with `expect` statements. The test harness is then the method marked with `{:test}`.

11.2.18. `{:timeLimit N}`

Set the time limit for verifying a given function or method.

11.2.19. `{:timeLimitMultiplier X}`

This attribute may be placed on a method or function declaration and has an integer argument. If `{:timeLimitMultiplier X}` was specified a `{:timeLimit Y}` attribute is passed on to Boogie where `Y` is `X` times either the default verification time limit for a function or method, or times the value specified by the Boogie `-timeLimit` command-line option.

11.2.20. `{:transparent}`

By default, the body of a function is transparent to its users. This can be overridden using the `--default-function-opacity` command line flag. If default function opacity is set to `opaque` or `autoRevealDependencies`, then this attribute can be used on functions to make them always non-opaque.

11.2.21. `{:verify false}`

Skip verification of a function or a method altogether, not even trying to verify the well-formedness of postconditions and preconditions. We discourage using this attribute and prefer `{:axiom}`, which performs these minimal checks while not checking that the body satisfies the postconditions.

If you simply want to temporarily disable all verification except on a single function or method, use the `{:only}` attribute on that function or method.

11.2.22. `{:vcs_max_cost N}`

Per-method version of the command-line option `/vcsMaxCost`.

The assertion batch of a method will not be split unless the cost of an assertion batch exceeds this number, defaults to 2000.0. In keep-going mode, only applies to the first round. If `{:isolate_assertions}` is set, then this parameter is useless.

11.2.23. `{:vcs_max_keep_going_splits N}`

Per-method version of the command-line option `/vcsMaxKeepGoingSplits`. If set to more than 1, activates the *keep going mode* where, after the first round of splitting, assertion batches that timed out are split into N assertion batches and retried until we succeed proving them, or there is only one single assertion that it timeouts (in which case an error is reported for that assertion). Defaults to 1. If `{:isolate_assertions}` is set, then this parameter is useless.

11.2.24. `{:vcs_max_splits N}`

Per-method version of the command-line option `/vcsMaxSplits`. Maximal number of assertion batches generated for this method. In keep-going mode, only applies to the first round. Defaults to 1. If `{:isolate_assertions}` is set, then this parameter is useless.

11.2.25. `{:isolate_assertions}`

Per-method version of the command-line option `/vcsSplitOnEveryAssert`

In the first and only verification round, this option will split the original assertion batch into one assertion batch per assertion. This is mostly helpful for debugging which assertion is taking the most time to prove, e.g. to profile them.

11.2.26. `{:synthesize}`

The `{:synthesize}` attribute must be used on methods that have no body and return one or more fresh objects. During compilation, the postconditions associated with such a method are translated to a series of API calls to the target languages's mocking framework. The object returned, therefore, behaves exactly as the postconditions specify. If there is a possibility that this behavior violates the specifications on the object's instance methods or hardcodes the values of its fields, the compiler will throw an error but the compilation will go through. Currently, this compilation pass is only supported in C# and requires adding the latest version of the Moq library to the .csproj file before generating the binary.

Not all Dafny postconditions can be successfully compiled - below is the grammar for postconditions that are supported (`S` is the start symbol, `EXPR` stands for an arbitrary Dafny expression, and `ID` stands for variable/method/type identifiers):

```
S         = FORALL
          | EQUALS
          | S && S
EQUALS    = ID.ID (ARGLIST) == EXPR // stubs a function call
          | ID.ID           == EXPR // stubs field access
          | EQUALS && EQUALS
FORALL    = forall BOUNDVARS :: EXPR ==> EQUALS
ARGLIST   = ID   // this can be one of the bound variables
          | EXPR // this expr may not reference any of the bound variables
          | ARGLIST, ARGLIST
BOUNDVARS = ID : ID
          | BOUNDVARS, BOUNDVARS
```

11.2.27. `{:options OPT0, OPT1, ... }`

This attribute applies only to modules. It configures Dafny as if `OPT0`, `OPT1`, … had been passed on the command line. Outside of the module, options revert to their previous values.

Only a small subset of Dafny's command line options is supported. Use the `/attrHelp` flag to see which ones.

11.3. Attributes on reads and modifies clauses

11.3.1. `{:assume_concurrent}`

This attribute is used to allow non-empty `reads` or `modifies` clauses on methods with the `{:concurrent}` attribute, which would otherwise reject them.

In some cases it is possible to know that Dafny code that reads or writes shared mutable state is in fact safe to use in a concurrent setting, especially when that state is exclusively ghost. Since the semantics of `{:concurrent}` aren't directly expressible in Dafny syntax, it isn't possible to express this assumption with an `assume {:axiom} ...` statement.

See also the `{:concurrent}` attribute.

11.4. Attributes on assertions, preconditions and postconditions

11.4.1. `{:only}`

`assert {:only} X;` temporarily transforms all other non-`{:only}` assertions in the surrounding declaration into assumptions.

```
method Test() {
  assert true;                  // Unchecked
  assert {:only} true by {      // Checked
    assert true;                // Checked
  }
  assert true;                  // Unchecked
  assert {:only "after"} true;  // Checked
  assert true;                  // Checked
  assert {:only "before"} true; // Checked
  assert true;                  // Unchecked
}
```

`{:only}` can help focusing on a particular proof or a particular branch, as it transforms not only other explicit assertions, but also other implicit assertions, and call requirements, into assumptions. Since it's meant to be a temporary construct, it always emits a warning. It also has two variants `assert {:only "before"}` and `assert {:only "after"}`. Here is precisely how Dafny determines what to verify or not. Each `{:only}` annotation defines a "verification interval" which is visual:

- `assert {:only} X [by {...} | ;]` sets a verification interval that starts at the keyword `assert` and ends either at the end of the proof `}` or the semicolon `;`, depending on which variant of `assert` is being used.
- `assert {:only} ...` inside another verification interval removes that verification interval and sets a new one.
- `assert {:only "before"} ...` inside another verification interval finishes that verification interval earlier at the end of this assertion. Outside a verification interval, it sets a verification interval from the beginning of the declaration to the end of this assertion, but only if there were no other verification intervals before.
- `assert {:only "after"} ...` inside another verification interval moves the start of that verification interval to the start of this new assert. Outside a verification interval, it sets a verification interval from the beginning of this `assert` to the end of the declaration.

The start of an asserted expression is used to determines if it's inside a verification interval or not. For example, in `assert B ==> (assert {:only "after"} true; C)`, `C` is actually the start of the asserted expression, so it is verified because it's after `assert {:only "after"} true`.

As soon as a declaration contains one `assert {:only}`, none of the postconditions are verified;

you'd need to make them explicit with assertions if you wanted to verify them at the same time.

You can also isolate the verification of a single member using a similar `{:only}` attribute.

11.4.2. `{:focus}`

`assert {:focus} X;` splits verification into two assertion batches. The first batch considers all assertions that are not on the block containing the `assert {:focus} X;` The second batch considers all assertions that are on the block containing the `assert {:focus} X;` and those that will *always* follow afterwards. Hence, it might also occasionally double-report errors. If you truly want a split on the batches, prefer `{:split_here}`.

Here are two examples illustrating how `{:focus}` works, where `--` in the comments stands for `Assumption`:

```
method doFocus1(x: bool) returns (y: int) {
  y := 1;                     // Batch 1    Batch 2
  assert y == 1;              // Assertion  --
  if x {
    if false {
      assert y >= 0;          // --         Assertion
      assert {:focus} y <= 2; // --         Assertion
      y := 2;
      assert y == 2;          // --         Assertion
    }
  } else {
    assert y == 1;            // Assertion  --
  }
  assert y == 1;              // Assertion  Assertion
  if !x {
    assert y >= 1;            // Assertion  Assertion
  } else {
    assert y <= 1;            // Assertion  Assertion
  }
}
```

And another one where the focused block is guarded with a `while`, resulting in remaining assertions not being part of the first assertion batch:

```
method doFocus2(x: bool) returns (y: int) {
  y := 1;                     // Batch 1    Batch 2
  assert y == 1;              // Assertion  --
  if x {
    while false {
      assert y >= 0;          // --         Assertion
      assert {:focus} y <= 2; // --         Assertion
      y := 2;
      assert y == 2;          // --         Assertion
    }
  } else {
    assert y == 1;            // Assertion  --
  }
  assert y == 1;              // Assertion  --
  if !x {
    assert y >= 1;            // Assertion  --
  } else {
    assert y <= 1;            // Assertion  --
  }
}
```

11.4.3. {:split_here}

`assert {:split_here} X;` splits verification into two assertion batches. It verifies the code leading to this point (excluded) in a first assertion batch, and the code leading from this point (included) to the next `{:split_here}` or until the end in a second assertion batch. It might help with timeouts.

Here is one example, where `--` in the comments stands for `Assumption`:

```
method doSplitHere(x: bool) returns (y: int) {
  y := 1;                     // Batch 1    Batch 2     Batch 3
  assert y >= 0;              // Assertion  --          --
  if x {
    assert y <= 1;            // Assertion  --          --
    assert {:split_here} true; // --        Assertion   --
    assert y <= 2;            // --         Assertion   --
    assert {:split_here} true; // --        --          Assertion
    if x {
      assert y == 1;          // --         --          Assertion
    } else {
      assert y >= 1;          // --         --          Assertion
    }
  } else {
    assert y <= 3;            // Assertion  --          --
  }
  assert y >= -1;             // Assertion  --          --
}
```

11.4.4. {:subsumption n}

Overrides the `/subsumption` command-line setting for this assertion. `{:subsumption 0}` checks an assertion but does not assume it after proving it. You can achieve the same effect using labelled assertions.

11.4.5. `{:error "errorMessage", "successMessage"}`

Provides a custom error message in case the assertion fails. As a hint, messages indicating what the user needs to do to fix the error are usually better than messages that indicate the error only. For example:

```
method Process(instances: int, price: int)
  requires {:error "There should be an even number of instances", "The number of instances is alway
  requires {:error "Could not prove that the price is positive", "The price is always positive"} pr
{
}
method Test()
{
  if * {
    Process(1, 0); // Error: There should be an even number of instances
  }
  if * {
    Process(2, -1); // Error: Could not prove that the price is positive
  }
  if * {
    Process(2, 5); // Success: The number of instances is always even
                   // Success: The price is always positive
  }
}
```

The success message is optional but is recommended if errorMessage is set.

11.4.6. `{:contradiction}`

Silences warnings about this assertion being involved in a proof using contradictory assumptions when `--warn-contradictory-assumptions` is enabled. This allows clear identification of intentional proofs by contradiction.

11.5. Attributes on variable declarations

11.5.1. `{:assumption}`

This attribute can only be placed on a local ghost bool variable of a method. Its declaration cannot have a rhs, but it is allowed to participate as the lhs of exactly one assignment of the form: `b := b && expr;`. Such a variable declaration translates in the Boogie output to a declaration followed by an `assume b` command. See (Leino and Wüstholz 2015), Section 3, for example uses of the `{:assumption}` attribute in Boogie.

11.6. Attributes on quantifier expressions (forall, exists)

11.6.1. `{:heapQuantifier}`

This attribute has been removed.

11.6.2. {:induction}

See {:induction} for functions and methods.

11.6.3. {:trigger}

Trigger attributes are used on quantifiers and comprehensions.

The verifier instantiates the body of a quantified expression only when it can find an expression that matches the provided trigger.

Here is an example:

```
predicate P(i: int)
predicate Q(i: int)

lemma {:axiom} PHoldEvenly()
  ensures  forall i {:trigger Q(i)} :: P(i) ==> P(i + 2) && Q(i)

lemma PHoldsForTwo()
  ensures forall i :: P(i) ==> P(i + 4)
{
  forall j: int
    ensures P(j) ==> P(j + 4)
  {
    if P(j) {
      assert P(j); // Trivial assertion

      PHoldEvenly();
      // Invoking the lemma assumes `forall i :: P(i) ==> P(i + 4)`,
      // but it's not instantiated yet

      // The verifier sees `Q(j)`, so it instantiates
      // `forall i :: P(i) ==> P(i + 4)` with `j`
      // and we get the axiom `P(j) ==> P(j + 2) && Q(j)`
      assert Q(j);     // hence it can prove `Q(j)`
      assert P(j + 2); //   and it can prove `P(j + 2)`
      assert P(j + 4); // But it cannot prove this
      // because it did not instantiate `forall i :: P(i) ==> P(i + 4)` with `j+2`
    }
  }
}
```

Here are ways one can prove `assert P(j + 4);`: * Add `assert Q(j + 2);` just before `assert P(j + 4);`, so that the verifier sees the trigger. * Change the trigger `{:trigger Q(i)}` to `{:trigger P(i)}` (replace the trigger) * Change the trigger `{:trigger Q(i)}` to `{:trigger Q(i)} {:trigger P(i)}` (add a trigger) * Remove `{:trigger Q(i)}` so that it will automatically determine all possible triggers thanks to the option `/autoTriggers:1` which is the default.

11.7. Deprecated attributes

These attributes have been deprecated or removed. They are no longer useful (or perhaps never were) or were experimental. They will likely be removed entirely sometime soon after the release of Dafny 4.

Removed: - :heapQuantifier - :dllimport - :handle

Deprecated: - :opaque : This attribute has been promoted to a first-class modifier for functions. Find more information here.

11.8. Other undocumented verification attributes

A scan of Dafny's sources shows it checks for the following attributes.

- `{:$}`
- `{:$renamed$}`
- `{:InlineAssume}`
- `{:PossiblyUnreachable}`
- `{:__dominator_enabled}`
- `{:__enabled}`
- `{:a##post##}`
- `{:absdomain}`
- `{:ah}`
- `{:assumption}`
- `{:assumption_variable_initialization}`
- `{:atomic}`
- `{:aux}`
- `{:both}`
- `{:bvbuiltin}`
- `{:candidate}`
- `{:captureState}`
- `{:checksum}`
- `{:constructor}`
- `{:datatype}`
- `{:do_not_predicate}`
- `{:entrypoint}`
- `{:existential}`
- `{:exitAssert}`
- `{:expand}`
- `{:extern}`
- `{:focus}`
- `{:hidden}`
- `{:ignore}`
- `{:inline}`
- `{:left}`
- `{:linear}`
- `{:linear_in}`

- {:linear_out}
- {:msg}
- {:name}
- {:originated_from_invariant}
- {:partition}
- {:positive}
- {:post}
- {:pre}
- {:precondition_previous_snapshot}
- {:qid}
- {:right}
- {:selective_checking}
- {:si_fcall}
- {:si_unique_call}
- {:sourcefile}
- {:sourceline}
- {:split_here}
- {:stage_active}
- {:stage_complete}
- {:staged_houdini_tag}
- {:start_checking_here}
- {:subsumption}
- {:template}
- {:terminates}
- {:upper}
- {:verified_under}
- {:weight}
- {:yields}

12. Advanced Topics

12.1. Type Parameter Completion

Generic types, like `A<T,U>`, consist of a *type constructor*, here `A`, and type parameters, here `T` and `U`. Type constructors are not first-class entities in Dafny, they are always used syntactically to construct type names; to do so, they must have the requisite number of type parameters, which must be either concrete types, type parameters, or a generic type instance.

However, those type parameters do not always have to be explicit; Dafny can often infer what they ought to be. For example, here is a fully parameterized function signature:

```
type List<T>
function Elements<T>(list: List<T>): set<T>
```

However, Dafny also accepts

```
type List<T>
function Elements(list: List): set
```

In the latter case, Dafny knows that the already defined types `set` and `List` each take one type parameter so it fills in `<T>` (using some unique type parameter name) and then determines that the function itself needs a type parameter `<T>` as well.

Dafny also accepts

```
type List<T>
function Elements<T>(list: List): set
```

In this case, the function already has a type parameter list. `List` and `set` are each known to need type parameters, so Dafny takes the first `n` parameters from the function signature and applies them to `List` and `set`, where `n` (here `1`) is the number needed by those type constructors.

It never hurts to simply write in all the type parameters, but that can reduce readability. Omitting them in cases where Dafny can intuit them makes a more compact definition.

This process is described in more detail with more examples in this paper: http://leino.science/papers/krml270.html.

12.2. Type Inference

Signatures of methods, functions, fields (except `const` fields with a RHS), and datatype constructors have to declare the types of their parameters. In other places, types can be omitted, in which case Dafny attempts to infer them. Type inference is "best effort" and may fail. If it fails to infer a type, the remedy is simply for the program to give the type explicitly.

Despite being just "best effort", the types of most local variables, bound variables, and the type parameters of calls are usually inferred without the need for a program to give the types explicitly. Here are some notes about type inference:

- With some exceptions, type inference is performed across a whole method body. In some cases, the information needed to infer a local variable's type may be found after the variable has been declared and used. For example, the nonsensical program

```
method M(n: nat) returns (y: int)
{
  var a, b;
  for i := 0 to n {
    if i % 2 == 0 {
      a := a + b;
    }
  }
  y := a;
}
```

uses `a` and `b` after their declarations. Still, their types are inferred to be `int`, because of the presence of the assignment `y := a;`.

A more useful example is this:

```
class Cell {
  var data: int
}

method LastFive(a: array<int>) returns (r: int)
{
  var u := null;
  for i := 0 to a.Length {
    if a[i] == 5 {
      u := new Cell;
      u.data := i;
    }
  }
  r := if u == null then a.Length else u.data;
}
```

Here, using only the assignment `u := null;` to infer the type of `u` would not be helpful. But Dafny looks past the initial assignment and infers the type of `u` to be `Cell?`.

- The primary example where type inference does not inspect the entire context before giving up on inference is when there is a member lookup. For example,

```
datatype List<T> = Nil | Cons(T, List<T>)

method Tutone() {
  assert forall pair :: pair.0 == 867 && pair.1 == 5309 ==> pair == (867, 5309); // error: mem
  assert forall pair: (int, int) :: pair.0 == 867 && pair.1 == 5309 ==> pair == (867, 5309);
}
```

In the first quantifier, type inference fails to infer the type of `pair` before it tries to look up the members `.0` and `.1`, which results in a "type of the receiver not fully determined" error.

The remedy is to provide the type of `pair` explicitly, as is done in the second quantifier.

(In the future, Dafny may do more type inference before giving up on the member lookup.)

- If type parameters cannot be inferred, then they can be given explicitly in angle brackets. For example, in

```
datatype Option<T> = None | Some(T)

method M() {
  var a: Option<int> := None;
  var b := None; // error: type is underspecified
  var c := Option<int>.None;
  var d := None;
  d := Some(400);
}
```

  the type of `b` cannot be inferred, because it is underspecified. However, the types of `c` and `d` are inferred to be `Option<int>`.

  Here is another example:

```
function EmptySet<T>(): set<T> {
  {}
}

method M() {
  var a := EmptySet(); // error: type is underspecified
  var b := EmptySet();
  b := b + {2, 3, 5};
  var c := EmptySet<int>();
}
```

  The type instantiation in the initial assignment to `a` cannot be inferred, because it is underspecified. However, the type instantiation in the initial assignment to `b` is inferred to be `int`, and the types of `b` and `c` are inferred to be `set<int>`.

- Even the element type of `new` is optional, if it can be inferred. For example, in

```
method NewArrays()
{
  var a := new int[3];
  var b: array<int> := new [3];
  var c := new [3];
  c[0] := 200;
  var d := new [3] [200, 800, 77];
  var e := new [] [200, 800, 77];
  var f := new [3](_ => 990);
}
```

the omitted types of local variables are all inferred as `array<int>` and the omitted element type of each `new` is inferred to be `int`.

- In the absence of any other information, integer-looking literals (like `5` and `7`) are inferred to have type `int` (and not, say, `bv128` or `ORDINAL`).
- Many of the types inferred can be inspected in the IDE.

12.3. Ghost Inference

After[13] type inference, Dafny revisits the program and makes a final decision about which statements are to be compiled, and which statements are ghost. The ghost statements form what is called the *ghost context* of expressions.

These statements are determined to be ghost:

- `assert`, `assume`, `reveal`, and `calc` statements.
- The body of the `by` of an `assert` statement.
- Calls to ghost methods, including lemmas.
- `if`, `match`, and `while` statements with condition expressions or alternatives containing ghost expressions. Their bodies are also ghost.
- `for` loops whose start expression contains ghost expressions.
- Variable declarations if they are explicitly ghost or if their respective right-hand side is a ghost expression.
- Assignments or update statement if all updated variables are ghost.
- `forall` statements, unless there is exactly one assignment to a non-ghost array in its body.

These statements always non-ghost:

- `expect` statements.
- `print` statements.

The following expressions are ghost, which is used in some of the tests above:

- All specification expressions
- All calls to functions and predicates marked as `ghost`
- All variables, constants and fields declared using the `ghost` keyword

Note that inferring ghostness can uncover other errors, such as updating non-ghost variables in ghost contexts. For example, if `f` is a ghost function, in the presence of the following code:

```
var x := 1;
if(f(x)) {
  x := 2;
}
```

Dafny will infer that the entire `if` is ghost because the condition uses a ghost function, and will then raise the error that it's not possible to update the non-ghost variable `x` in a ghost context.

[13]Ghost inference has to be performed after type inference, at least because it is not possible to determine if a member access `a.b` refers to a ghost variable until the type of `a` is determined.

12.4. Well-founded Functions and Extreme Predicates

Recursive functions are a core part of computer science and mathematics. Roughly speaking, when the definition of such a function spells out a terminating computation from given arguments, we may refer to it as a *well-founded function*. For example, the common factorial and Fibonacci functions are well-founded functions.

There are also other ways to define functions. An important case regards the definition of a boolean function as an extreme solution (that is, a least or greatest solution) to some equation. For computer scientists with interests in logic or programming languages, these *extreme predicates* are important because they describe the judgments that can be justified by a given set of inference rules (see, e.g., (Camilleri and Melham 1992; Winskel 1993; Leroy and Grall 2009; Pierce et al. 2015; Nipkow and Klein 2014)).

To benefit from machine-assisted reasoning, it is necessary not just to understand extreme predicates but also to have techniques for proving theorems about them. A foundation for this reasoning was developed by Paulin-Mohring (Paulin-Mohring 1993) and is the basis of the constructive logic supported by Coq (Bertot and Castéran 2004) as well as other proof assistants (Bove, Dybjer, and Norell 2009; Swamy et al. 2011). Essentially, the idea is to represent the knowledge that an extreme predicate holds by the proof term by which this knowledge was derived. For a predicate defined as the least solution, such proof terms are values of an inductive datatype (that is, finite proof trees), and for the greatest solution, a coinductive datatype (that is, possibly infinite proof trees). This means that one can use induction and coinduction when reasoning about these proof trees. These extreme predicates are known as, respectively, *least predicates* and *greatest predicates*. Support for extreme predicates is also available in the proof assistants Isabelle (Paulson 1994) and HOL (Harrison 1995).

Dafny supports both well-founded functions and extreme predicates. This section describes the difference in general terms, and then describes novel syntactic support in Dafny for defining and proving lemmas with extreme predicates. Although Dafny's verifier has at its core a first-order SMT solver, Dafny's logical encoding makes it possible to reason about fixpoints in an automated way.

The encoding for greatest predicates in Dafny was described previously (Leino and Moskal 2014b) and is here described in the section about datatypes.

12.4.1. Function Definitions

To define a function $f\colon X \to Y$ in terms of itself, one can write a general equation like

$$f = \mathcal{F}(f)$$

where $\mathcal{F}$ is a non-recursive function of type $(X \to Y) \to X \to Y$. Because it takes a function as an argument, $\mathcal{F}$ is referred to as a *functor* (or *functional*, but not to be confused by the category-theory notion of a functor). Throughout, assume that $\mathcal{F}(f)$ by itself is well defined, for example that it does not divide by zero. Also assume that f occurs only in fully applied calls in $\mathcal{F}(f)$; eta expansion can be applied to ensure this. If f is a `boolean` function, that is, if Y is the type of booleans, then f is called a *predicate*.

For example, the common Fibonacci function over the natural numbers can be defined by the equation

$$\mathit{fib} = \lambda n \bullet \ \mathbf{if}\, n < 2\, \mathbf{then}\, n\, \mathbf{else}\, \mathit{fib}(n-2) + \mathit{fib}(n-1)$$

With the understanding that the argument n is universally quantified, we can write this equation equivalently as

$$\mathit{fib}(n) = \mathbf{if}\, n < 2\, \mathbf{then}\, n\, \mathbf{else}\, \mathit{fib}(n-2)$$

The fact that the function being defined occurs on both sides of the equation causes concern that we might not be defining the function properly, leading to a logical inconsistency. In general, there could be many solutions to an equation like the general equation or there could be none. Let's consider two ways to make sure we're defining the function uniquely.

12.4.1.1. Well-founded Functions A standard way to ensure that the general equation has a unique solution in f is to make sure the recursion is well-founded, which roughly means that the recursion terminates. This is done by introducing any well-founded relation $\ll$ on the domain of f and making sure that the argument to each recursive call goes down in this ordering. More precisely, if we formulate the general equation as

$$f(x) = \mathcal{F}'(f)$$

then we want to check $E \ll x$ for each call $f(E)$ in $f(x) = \mathcal{F}'(f)$. When a function definition satisfies this *decrement condition*, then the function is said to be *well-founded*.

For example, to check the decrement condition for *fib* in the fib equation, we can pick $\ll$ to be the arithmetic less-than relation on natural numbers and check the following, for any n:

$$2 \leq n \implies n-2 \ll n \ \wedge\ n-1 \ll n$$

Note that we are entitled to use the antecedent $2 \leq n$ because that is the condition under which the else branch in the fib equation is evaluated.

A well-founded function is often thought of as "terminating" in the sense that the recursive *depth* in evaluating f on any given argument is finite. That is, there are no infinite descending chains of recursive calls. However, the evaluation of f on a given argument may fail to terminate, because its *width* may be infinite. For example, let P be some predicate defined on the ordinals and let $P_\downarrow$ be a predicate on the ordinals defined by the following equation:

$P_\downarrow = P(o) \ \wedge\ \forall p \bullet\ p \ll o \implies P_\downarrow(p)$

With $\ll$ as the usual ordering on ordinals, this equation satisfies the decrement condition, but evaluating $P\_ \downarrow (\omega)$ would require evaluating $P\_ \downarrow (n)$ for every natural number n. However, what we are concerned about here is to avoid mathematical inconsistencies, and that is indeed a consequence of the decrement condition.

12.4.1.2. Example with Well-founded Functions So that we can later see how inductive proofs are done in Dafny, let's prove that for any n, $\mathit{fib}(n)$ is even iff n is a multiple of 3. We split our task into two cases. If $n < 2$, then the property follows directly from the definition of *fib*. Otherwise, note that exactly one of the three numbers $n-2$, $n-1$, and n is a multiple of 3. If n is the multiple of 3, then by invoking the induction hypothesis on $n-2$ and $n-1$, we obtain that $\mathit{fib}(n-2)+\mathit{fib}(n-1)$ is the sum of two odd numbers, which is even. If $n-2$ or $n-1$ is a multiple of 3, then by invoking the induction hypothesis on $n-2$ and $n-1$, we obtain that $\mathit{fib}(n-2)+\mathit{fib}(n-1)$ is the sum of an even number and an odd number, which is odd. In this proof, we invoked the induction hypothesis on $n-2$ and on $n-1$. This is allowed, because both are smaller than n, and hence the invocations go down in the well-founded ordering on natural numbers.

12.4.1.3. Extreme Solutions We don't need to exclude the possibility of the general equation having multiple solutions—instead, we can just be clear about which one of them we want. Let's explore this, after a smidgen of lattice theory.

For any complete lattice $(Y, \leq)$ and any set X, we can by *pointwise extension* define a complete lattice $(X \to Y, \dot{\Rightarrow})$, where for any $f, g\colon X \to Y$,

$$f \dot{\Rightarrow} g \quad \equiv \quad \forall x \bullet\ f(x) \leq g(x)$$

In particular, if Y is the set of booleans ordered by implication (`false` $\leq$ `true`), then the set of predicates over any domain X forms a complete lattice. Tarski's Theorem (Tarski 1955) tells us that any monotonic function over a complete lattice has a least and a greatest fixpoint. In particular, this means that $\mathcal{F}$ has a least fixpoint and a greatest fixpoint, provided $\mathcal{F}$ is monotonic.

Speaking about the *set of solutions* in f to the general equation is the same as speaking about the *set of fixpoints* of functor $\mathcal{F}$. In particular, the least and greatest solutions to the general equation are the same as the least and greatest fixpoints of $\mathcal{F}$. In casual speak, it happens that we say "fixpoint of the general equation", or more grotesquely, "fixpoint of f" when we really mean "fixpoint of $\mathcal{F}$".

To conclude our little excursion into lattice theory, we have that, under the proviso of $\mathcal{F}$ being monotonic, the set of solutions in f to the general equation is nonempty, and among these solutions, there is in the $\dot{\Rightarrow}$ ordering a least solution (that is, a function that returns `false` more often than any other) and a greatest solution (that is, a function that returns `true` more often than any other).

When discussing extreme solutions, let's now restrict our attention to boolean functions (that is, with Y being the type of booleans). Functor $\mathcal{F}$ is monotonic if the calls to f in $\mathcal{F}'(f)$ are in *positive positions* (that is, under an even number of negations). Indeed, from now on, we will restrict our attention to such monotonic functors $\mathcal{F}$.

Here is a running example. Consider the following equation, where x ranges over the integers:

$$g(x) = (x = 0\ \vee\ g(x-2))$$

This equation has four solutions in g. With w ranging over the integers, they are:

$$\begin{array}{lcl} g(x) & \equiv & x \in \{w \mid 0 \leq w \ \wedge \ w \text{ even}\} \\ g(x) & \equiv & x \in \{w \mid w \text{ even}\} \\ g(x) & \equiv & x \in \{w \mid (0 \leq w \ \wedge \ w \text{ even}) \ \vee \ w \text{ odd}\} \\ g(x) & \equiv & x \in \{w \mid \mathit{true}\} \end{array}$$

The first of these is the least solution and the last is the greatest solution.

In the literature, the definition of an extreme predicate is often given as a set of *inference rules*. To designate the least solution, a single line separating the antecedent (on top) from conclusion (on bottom) is used:

$$\frac{}{g(0)} \qquad\qquad \frac{g(x-2)}{g(x)}$$

Through repeated applications of such rules, one can show that the predicate holds for a particular value. For example, the *derivation*, or *proof tree*, to the left in the proof tree figure shows that $g(6)$ holds. (In this simple example, the derivation is a rather degenerate proof "tree".) The use of these inference rules gives rise to a least solution, because proof trees are accepted only if they are *finite*.

When inference rules are to designate the greatest solution, a thick line is used:

$$\genfrac{}{}{1.5pt}{}{}{g(0)} \qquad\qquad \genfrac{}{}{1.5pt}{}{g(x-2)}{g(x)}$$

In this case, proof trees are allowed to be infinite. For example, the left-hand example below shows a finite proof tree that uses the inductive rules to establish $g(6)$. On the right is a partial depiction of an infinite proof tree that uses the coinductive rules to establish $g(1)$.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{g(0)}}{g(2)}}{g(4)}}{g(6)} \qquad\qquad \genfrac{}{}{1.5pt}{}{\genfrac{}{}{1.5pt}{}{\genfrac{}{}{1.5pt}{}{\genfrac{}{}{1.5pt}{}{\vdots}{g(-5)}}{g(-3)}}{g(-1)}}{g(1)}$$

Note that derivations may not be unique. For example, in the case of the greatest solution for g, there are two proof trees that establish $g(0)$: one is the finite proof tree that uses the left-hand rule of these coinductive rules once, the other is the infinite proof tree that keeps on using the right-hand rule of these coinductive rules.

12.4.2. Working with Extreme Predicates

In general, one cannot evaluate whether or not an extreme predicate holds for some input, because doing so may take an infinite number of steps. For example, following the recursive calls in the definition the EvenNat equation to try to evaluate $g(7)$ would never terminate. However, there are useful ways to establish that an extreme predicate holds and there are ways to make use of one once it has been established.

For any $\mathcal{F}$ as in the general equation, define two infinite series of well-founded functions, ${}^{\flat}\!f_k$ and ${}^{\sharp}\!f_k$ where k ranges over the natural numbers:

$$
{}^{\flat}\!f_k(x) = \left\{ \begin{array}{ll} \mathit{false} & \text{if } k = 0 \\ \mathcal{F}({}^{\flat}\!f_{k-1})(x) & \text{if } k > 0 \end{array} \right\}
$$

$$
{}^{\sharp}\!f_k(x) = \left\{ \begin{array}{ll} \mathit{true} & \text{if } k = 0 \\ \mathcal{F}({}^{\sharp}\!f_{k-1})(x) & \text{if } k > 0 \end{array} \right\}
$$

These functions are called the *iterates* of f, and we will also refer to them as the *prefix predicates* of f (or the *prefix predicate* of f, if we think of k as being a parameter). Alternatively, we can define ${}^{\flat}\!f_k$ and ${}^{\sharp}\!f_k$ without mentioning x: let $\bot$ denote the function that always returns `false`, let $\top$ denote the function that always returns `true`, and let a superscript on $\mathcal{F}$ denote exponentiation (for example, $\mathcal{F}^0(f) = f$ and $\mathcal{F}^2(f) = \mathcal{F}(\mathcal{F}(f))$). Then, the least approx definition and the greatest approx definition can be stated equivalently as ${}^{\flat}\!f_k = \mathcal{F}^k(\bot)$ and ${}^{\sharp}\!f_k = \mathcal{F}^k(\top)$.

For any solution f to the general equation, we have, for any k and ℓ such that $k \leq \ell$:

$$
{}^{\flat}\!f_k \quad \dot{\Rightarrow} \quad {}^{\flat}\!f_\ell \quad \dot{\Rightarrow} \quad f \quad \dot{\Rightarrow} \quad {}^{\sharp}\!f_\ell \quad \dot{\Rightarrow} \quad {}^{\sharp}\!f_k
$$

In other words, every ${}^{\flat}\!f\_k$ is a *pre-fixpoint* of f and every ${}^{\sharp}\!f\_k$ is a *post-fixpoint* of f. Next, define two functions, $f^{\downarrow}$ and $f^{\uparrow}$, in terms of the prefix predicates:

$$
f^{\downarrow}(x) \;=\; \exists k \bullet \; {}^{\flat}\!f_k(x)
$$

$$
f^{\uparrow}(x) \;=\; \forall k \bullet \; {}^{\sharp}\!f_k(x)
$$

By the prefix postfix result, we also have that $f^{\downarrow}$ is a pre-fixpoint of $\mathcal{F}$ and $f^{\uparrow}$ is a post-fixpoint of $\mathcal{F}$. The marvelous thing is that, if $\mathcal{F}$ is *continuous*, then $f^{\downarrow}$ and $f^{\uparrow}$ are the least and greatest fixpoints of $\mathcal{F}$. These equations let us do proofs by induction when dealing with extreme predicates. The extreme predicate section explains how to check for continuity.

Let's consider two examples, both involving function g in the EvenNat equation. As it turns out, g's defining functor is continuous, and therefore I will write $g^{\downarrow}$ and $g^{\uparrow}$ to denote the least and greatest solutions for g in the EvenNat equation.

12.4.2.1. Example with Least Solution The main technique for establishing that $g^{\downarrow}(x)$ holds for some x, that is, proving something of the form $Q \implies g^{\downarrow}(x)$, is to construct a proof tree like the one for $g(6)$ in the proof tree figure. For a proof in this direction, since we're just applying the defining equation, the fact that we're using a least solution for g never plays a role (as long as we limit ourselves to finite derivations).

The technique for going in the other direction, proving something *from* an established $g^{\downarrow}$ property, that is, showing something of the form $g^{\downarrow}(x) \implies R$, typically uses induction on the structure of the proof tree. When the antecedent of our proof obligation includes a predicate term $g^{\downarrow}(x)$, it is sound to imagine that we have been given a proof tree for $g^{\downarrow}(x)$. Such a proof tree would be a data structure—to be more precise, a term in an *inductive datatype*. Least solutions like $g^{\downarrow}$ have been given the name *least predicate*.

Let's prove $g^{\downarrow}(x) \implies 0 \le x \wedge x$ even. We split our task into two cases, corresponding to which of the two proof rules in the inductive rules was the last one applied to establish $g^{\downarrow}(x)$. If it was the left-hand rule, then $x = 0$, which makes it easy to establish the conclusion of our proof goal. If it was the right-hand rule, then we unfold the proof tree one level and obtain $g^{\downarrow}(x-2)$. Since the proof tree for $g^{\downarrow}(x-2)$ is smaller than where we started, we invoke the *induction hypothesis* and obtain $0 \le (x-2) \wedge (x-2)$ even, from which it is easy to establish the conclusion of our proof goal.

Here's how we do the proof formally using the least exists definition. We massage the general form of our proof goal:

$$
\begin{array}{ll}
 & f^{\uparrow}(x) \implies R \\
= & \qquad\qquad \text{(the least exists definition)} \\
 & (\exists k \bullet {}^{\flat}f_k(x)) \implies R \\
= & \qquad\qquad \text{distribute} \implies \text{over} \exists \text{ to the left} \\
 & \forall k \bullet ({}^{\flat}f_k(x) \implies R)
\end{array}
$$

The last line can be proved by induction over k. So, in our case, we prove ${}^{\flat}g\_k(x) \implies 0 \le x \wedge x$ even for every k. If $k = 0$, then ${}^{\flat}g\_k(x)$ is `false`, so our goal holds trivially. If $k > 0$, then ${}^{\flat}g\_k(x) = (x = 0 \vee {}^{\flat}g\_k - 1(x-2))$. Our goal holds easily for the first disjunct ($x = 0$). For the other disjunct, we apply the induction hypothesis (on the smaller $k-1$ and with $x-2$) and obtain $0 \le (x-2) \wedge (x-2)$ even, from which our proof goal follows.

12.4.2.2. Example with Greatest Solution We can think of a predicate $g^{\uparrow}(x)$ as being represented by a proof tree—in this case a term in a *coinductive datatype*, since the proof may be infinite. Greatest solutions like $g^{\uparrow}$ have been given the name *greatest predicate*. The main technique for proving something from a given proof tree, that is, to prove something of the form $g^{\uparrow}(x) \implies R$, is to destruct the proof. Since this is just unfolding the defining equation, the fact that we're using a greatest solution for g never plays a role (as long as we limit ourselves to a finite number of unfoldings).

To go in the other direction, to establish a predicate defined as a greatest solution, like $Q \implies g^{\uparrow}(x)$, we may need an infinite number of steps. For this purpose, we can use induction's dual, *coinduction*. Were it not for one little detail, coinduction is as simple as continuations in programming: the next part of the proof obligation is delegated to the *coinduction hypothesis*. The little detail is making sure that it is the "next" part we're passing on for the continuation, not the same part. This detail is called *productivity* and corresponds to the requirement in induction of making sure we're going down a well-founded relation when applying the induction hypothesis. There are many sources with more information, see for example the classic account by Jacobs and Rutten (Jacobs and Rutten 2011) or a new attempt by Kozen and Silva that aims to emphasize the simplicity, not the mystery, of coinduction (Kozen and Silva 2012).

Let's prove *true* $\implies g^{\uparrow}(x)$. The intuitive coinductive proof goes like this: According to the right-hand rule of these coinductive rules, $g^{\uparrow}(x)$ follows if we establish $g^{\uparrow}(x-2)$, and that's easy to do by invoking the coinduction hypothesis. The "little detail", productivity, is satisfied in this proof because we applied a rule in these coinductive rules before invoking the coinduction hypothesis.

For anyone who may have felt that the intuitive proof felt too easy, here is a formal proof using the greatest forall definition, which relies only on induction. We massage the general form of our proof goal:

$$\begin{array}{ll} & Q \implies f^{\uparrow}(x) \\ = & \qquad\qquad \text{(the greatest forall definition)} \\ & Q \implies \forall k \bullet\ {}^{\sharp}\!f_k(x) \\ = & \qquad\qquad \text{distribute} \implies \text{over } \forall \text{ to the right} \\ & \forall k \bullet\ Q \implies {}^{\sharp}\!f_k(x) \end{array}$$

The last line can be proved by induction over k. So, in our case, we prove *true* $\implies {}^{\sharp}\!g_k(x)$ for every k. If $k = 0$, then ${}^{\sharp}\!g\_k(x)$ is *true*, so our goal holds trivially. If $k > 0$, then ${}^{\sharp}\!g\_k(x) = (x = 0 \vee {}^{\sharp}\!g\_k - 1(x-2))$. We establish the second disjunct by applying the induction hypothesis (on the smaller $k-1$ and with $x-2$).

12.4.3. Other Techniques

Although this section has considered only well-founded functions and extreme predicates, it is worth mentioning that there are additional ways of making sure that the set of solutions to the general equation is nonempty. For example, if all calls to f in $\mathcal{F}'(f)$ are *tail-recursive calls*, then (under the assumption that Y is nonempty) the set of solutions is nonempty. To see this, consider an attempted evaluation of $f(x)$ that fails to determine a definite result value because of an infinite chain of calls that applies f to each value of some subset X' of X. Then, apparently, the value of f for any one of the values in X' is not determined by the equation, but picking any particular result value for these makes for a consistent definition. This was pointed out by Manolios and Moore (Manolios and Moore 2003). Functions can be underspecified in this way in the proof assistants ACL2 (Kaufmann, Manolios, and Moore 2000) and HOL (Krauss 2009).

12.5. Functions in Dafny

This section explains with examples the support in Dafny for well-founded functions, extreme predicates, and proofs regarding these, building on the concepts explained in the previous section.

12.5.1. Well-founded Functions in Dafny

Declarations of well-founded functions are unsurprising. For example, the Fibonacci function is declared as follows:

```
function fib(n: nat): nat
{
  if n < 2 then n else fib(n-2) + fib(n-1)
}
```

Dafny verifies that the body (given as an expression in curly braces) is well defined. This includes decrement checks for recursive (and mutually recursive) calls. Dafny predefines a well-founded relation on each type and extends it to lexicographic tuples of any (fixed) length. For example, the well-founded relation $x \ll y$ for integers is $x < y \ \wedge \ 0 \leq y$, the one for reals is $x \leq y - 1.0 \ \wedge \ 0.0 \leq y$ (this is the same ordering as for integers, if you read the integer relation as $x \leq y - 1 \ \wedge \ 0 \leq y$), the one for inductive datatypes is structural inclusion, and the one for coinductive datatypes is `false`.

Using a `decreases` clause, the programmer can specify the term in this predefined order. When a function definition omits a `decreases` clause, Dafny makes a simple guess. This guess (which can be inspected by hovering over the function name in the Dafny IDE) is very often correct, so users are rarely bothered to provide explicit `decreases` clauses.

If a function returns `bool`, one can drop the result type `: bool` and change the keyword `function` to `predicate`.

12.5.2. Proofs in Dafny

Dafny has `lemma` declarations, as described in Section 6.3.3: lemmas can have pre- and postcondition specifications and their body is a code block. Here is the lemma we stated and proved in the fib example in the previous section:

```
lemma FibProperty(n: nat)
  ensures fib(n) % 2 == 0 <==> n % 3 == 0
{
  if n < 2 {
  } else {
    FibProperty(n-2); FibProperty(n-1);
  }
}
function fib(n: nat): nat
{
  if n < 2 then n else fib(n-2) + fib(n-1)
}
```

The postcondition of this lemma (keyword `ensures`) gives the proof goal. As in any program-correctness logic (e.g., (Hoare 1969)), the postcondition must be established on every control path through the lemma's body. For `FibProperty`, I give the proof by an `if` statement, hence introducing a case split. The then branch is empty, because Dafny can prove the postcondition automatically in this case. The else branch performs two recursive calls to the lemma. These are the invocations of the induction hypothesis and they follow the usual program-correctness rules, namely: the precondition must hold at the call site, the call must terminate, and then the caller gets to assume the postcondition upon return. The "proof glue" needed to complete the proof is done automatically by Dafny.

Dafny features an aggregate statement using which it is possible to make (possibly infinitely) many calls at once. For example, the induction hypothesis can be called at once on all values `n'` smaller than `n`:

```
forall n' | 0 <= n' < n {
  FibProperty(n');
}
```

For our purposes, this corresponds to *strong induction*. More generally, the `forall` statement has the form

```
forall k | P(k)
  ensures Q(k)
{ Statements; }
```

Logically, this statement corresponds to *universal introduction*: the body proves that `Q(k)` holds for an arbitrary `k` such that `P(k)`, and the conclusion of the `forall` statement is then $\forall k \bullet P(k) \implies Q(k)$. When the body of the `forall` statement is a single call (or `calc` statement), the `ensures` clause is inferred and can be omitted, like in our `FibProperty` example.

Lemma `FibProperty` is simple enough that its whole body can be replaced by the one `forall` statement above. In fact, Dafny goes one step further: it automatically inserts such a `forall` statement at the beginning of every lemma (Leino 2012). Thus, `FibProperty` can be declared and proved simply by:

```
lemma FibProperty(n: nat)
  ensures fib(n) % 2 == 0 <==> n % 3 == 0
{ }
function fib(n: nat): nat
{
  if n < 2 then n else fib(n-2) + fib(n-1)
}
```

Going in the other direction from universal introduction is existential elimination, also known as Skolemization. Dafny has a statement for this, too: for any variable `x` and boolean expression `Q`, the *assign such that* statement `x :| Q;` says to assign to `x` a value such that `Q` will hold. A proof obligation when using this statement is to show that there exists an `x` such that `Q` holds. For example, if the fact
$existsk \bullet \ 100 \leq fib(k) < 200$ is known, then the statement `k :| 100 <= fib(k) < 200;` will assign to `k` some value (chosen arbitrarily) for which `fib(k)` falls in the given range.

12.5.3. Extreme Predicates in Dafny

The previous subsection explained that a `predicate` declaration introduces a well-founded predicate. The declarations for introducing extreme predicates are `least predicate` and `greatest predicate`. Here is the definition of the least and greatest solutions of g from above; let's call them `g` and `G`:

```
least predicate g[nat](x: int) { x == 0 || g(x-2) }
greatest predicate G[nat](x: int) { x == 0 || G(x-2) }
```

When Dafny receives either of these definitions, it automatically declares the corresponding prefix predicates. Instead of the names $^{\flat}g_k$ and $^{\sharp}g_k$ that I used above, Dafny names the prefix predicates `g#[k]` and `G#[k]`, respectively, that is, the name of the extreme predicate appended with `#`, and the subscript is given as an argument in square brackets. The definition of the prefix predicate derives from the body of the extreme predicate and follows the form in the least approx definition and the greatest approx definition. Using a faux-syntax for illustrative purposes, here are the prefix predicates that Dafny defines automatically from the extreme predicates `g` and `G`:

```
predicate g#[_k: nat](x: int) { _k != 0 && (x == 0 || g#[_k-1](x-2)) }
predicate G#[_k: nat](x: int) { _k != 0 ==> (x == 0 || G#[_k-1](x-2)) }
```

The Dafny verifier is aware of the connection between extreme predicates and their prefix predicates, the least exists definition and the greatest forall definition.

Remember that to be well defined, the defining functor of an extreme predicate must be monotonic, and for the least exists definition and the greatest forall definition to hold, the functor must be continuous. Dafny enforces the former of these by checking that recursive calls of extreme predicates are in positive positions. The continuity requirement comes down to checking that they are also in *continuous positions*: that recursive calls to least predicates are not inside unbounded universal quantifiers and that recursive calls to greatest predicates are not inside unbounded existential quantifiers (Milner 1982; Leino and Moskal 2014b).

12.5.4. Proofs about Extreme Predicates

From what has been presented so far, we can do the formal proofs for the example about the least solution and the example about the greatest solution. Here is the former:

```
least predicate g[nat](x: int) { x == 0 || g(x-2) }
greatest predicate G[nat](x: int) { x == 0 || G(x-2) }
lemma EvenNat(x: int)
  requires g(x)
  ensures 0 <= x && x % 2 == 0
{
  var k: nat :| g#[k](x);
  EvenNatAux(k, x);
}
lemma EvenNatAux(k: nat, x: int)
  requires g#[k](x)
  ensures 0 <= x && x % 2 == 0
```

```
{
  if x == 0 { } else { EvenNatAux(k-1, x-2); }
}
```

Lemma `EvenNat` states the property we wish to prove. From its precondition (keyword `requires`) and the least exists definition, we know there is some `k` that will make the condition in the assign-such-that statement true. Such a value is then assigned to `k` and passed to the auxiliary lemma, which promises to establish the proof goal. Given the condition `g#[k](x)`, the definition of `g#` lets us conclude `k != 0` as well as the disjunction `x == 0 || g#[k-1](x-2)`. The then branch considers the case of the first disjunct, from which the proof goal follows automatically. The else branch can then assume `g#[k-1](x-2)` and calls the induction hypothesis with those parameters. The proof glue that shows the proof goal for `x` to follow from the proof goal with `x-2` is done automatically.

Because Dafny automatically inserts the statement

```
forall k', x' | 0 <= k' < k && g#[k'](x') {
  EvenNatAux(k', x');
}
```

at the beginning of the body of `EvenNatAux`, the body can be left empty and Dafny completes the proof automatically.

Here is the Dafny program that gives the proof from the example of the greatest solution:

```
least predicate g[nat](x: int) { x == 0 || g(x-2) }
greatest predicate G[nat](x: int) { x == 0 || G(x-2) }
lemma Always(x: int)
  ensures G(x)
{ forall k: nat { AlwaysAux(k, x); } }
lemma AlwaysAux(k: nat, x: int)
  ensures G#[k](x)
{ }
```

While each of these proofs involves only basic proof rules, the setup feels a bit clumsy, even with the empty body of the auxiliary lemmas. Moreover, the proofs do not reflect the intuitive proofs described in the example of the least solution and the example of the greatest solution. These shortcomings are addressed in the next subsection.

12.5.5. Nicer Proofs of Extreme Predicates

The proofs we just saw follow standard forms: use Skolemization to convert the least predicate into a prefix predicate for some `k` and then do the proof inductively over `k`; respectively, by induction over `k`, prove the prefix predicate for every `k`, then use universal introduction to convert to the greatest predicate. With the declarations `least lemma` and `greatest lemma`, Dafny offers to set up the proofs in these standard forms. What is gained is not just fewer characters in the program text, but also a possible intuitive reading of the proofs. (Okay, to be fair, the reading is intuitive for simpler proofs; complicated proofs may or may not be intuitive.)

Somewhat analogous to the creation of prefix predicates from extreme predicates, Dafny automatically creates a *prefix lemma* `L#` from each "extreme lemma" `L`. The pre- and postconditions of a prefix lemma are copied from those of the extreme lemma, except for the following replacements: * for a least lemma, Dafny looks in the precondition to find calls (in positive, continuous positions) to least predicates `P(x)` and replaces these with `P#[_k](x)`; * for a greatest lemma, Dafny looks in the postcondition to find calls (in positive, continuous positions) to greatest predicates `P` (including equality among coinductive datatypes, which is a built-in greatest predicate) and replaces these with `P#[_k](x)`. In each case, these predicates `P` are the lemma's *focal predicates*.

The body of the extreme lemma is moved to the prefix lemma, but with replacing each recursive call `L(x)` with `L#[_k-1](x)` and replacing each occurrence of a call to a focal predicate `P(x)` with `P#[_k-1](x)`. The bodies of the extreme lemmas are then replaced as shown in the previous subsection. By construction, this new body correctly leads to the extreme lemma's postcondition.

Let us see what effect these rewrites have on how one can write proofs. Here are the proofs of our running example:

```
least predicate g(x: int) { x == 0 || g(x-2) }
greatest predicate G(x: int) { x == 0 || G(x-2) }
least lemma EvenNat(x: int)
  requires g(x)
  ensures 0 <= x && x % 2 == 0
{ if x == 0 { } else { EvenNat(x-2); } }
greatest lemma Always(x: int)
  ensures G(x)
{ Always(x-2); }
```

Both of these proofs follow the intuitive proofs given in the example of the least solution and the example of the greatest solution. Note that in these simple examples, the user is never bothered with either prefix predicates nor prefix lemmas—the proofs just look like "what you'd expect".

Since Dafny automatically inserts calls to the induction hypothesis at the beginning of each lemma, the bodies of the given extreme lemmas `EvenNat` and `Always` can be empty and Dafny still completes the proofs. Folks, it doesn't get any simpler than that!

12.6. Variable Initialization and Definite Assignment

The Dafny language semantics ensures that any use (read) of a variable (or constant, parameter, object field, or array element) gives a value of the variable's type. It is easy to see that this property holds for any variable that is declared with an initializing assignment. However, for many useful programs, it would be too strict to require an initializing assignment at the time a variable is declared. Instead, Dafny ensures the property through *auto-initialization* and rules for *definite assignment*.

As explained in section 5.3.1, each type in Dafny is one of the following:

- *auto-init type*: the type is nonempty and the compiler has some way to emit code that constructs a value

- *nonempty type*: the type is nonempty, but the compiler does not know how perform automatic initialization
- *possibly empty type*: the type is not known for sure to have a value

For a variable of an auto-init type, the compiler can initialize the variable automatically. This means that the variable can be used immediately after declaration, even if the program does not explicitly provide an initializing assignment.

In a ghost context, one can an imagine a "ghost" that initializes variables. Unlike the compiler, such a "ghost" does not need to emit code that constructs an initializing value; it suffices for the ghost to know that a value exists. Therefore, in a ghost context, a variable of a nonempty type can be used immediately after declaration.

Before a variable of a possibly empty type can be used, the program must initialize it. The variable need not be given a value when it is declared, but it must have a value by the time it is first used. Dafny uses the precision of the verifier to reason about the control flow between assignments and uses of variables, and it reports an error if it cannot assure itself that the variable has been given a value.

The elements of an array must be assured to have values already in the statement that allocates the array. This is achieved in any of the following four ways: - If the array is allocated to be empty (that is, one of its dimensions is requested to be 0), then the array allocation trivially satisfies the requirement. - If the element type of the array is an auto-init type, then nothing further is required by the program. - If the array allocation occurs in a ghost context and the element type is a nonempty type, then nothing further is required by the program. - Otherwise, the array allocation must provide an initialization display or an initialization function. See section 5.10 for information about array initialization.

The fields of a class must have values by the end of the first phase of each constructor (that is, at the explicit or implicit `new;` statement in the constructor). If a class has a compiled field that is not of an auto-init type, or if it has a ghost field of a possibly empty type, then the class is required to declare a(t least one) constructor.

The yield-parameters of an `iterator` turn into fields of the corresponding iterator class, but there is no syntactic place to give these initial values. Therefore, every compiled yield-parameter must be of auto-init types and every ghost yield-parameter must be of an auto-init or nonempty type.

For local variables and out-parameters, Dafny supports two definite-assignment modes: - A strict mode (the default, which is `--relax-definite-assignment=false`; or `/definiteAssignment:4` in the legacy CLI), in which local variables and out-parameters are always subject to definite-assignment rules, even for auto-initializable types. - A relaxed mode (enabled by the option `--relax-definite-assignment`; or `/definiteAssignment:1` in the legacy CLI), in which the auto-initialization (or, for ghost variables and parametes, nonemptiness) is sufficient to satisfy the definite assignment rules.

A program using the strict mode can still indicate that it is okay with an arbitrary value of a variable `x` by using an assignment statement `x := *;`, provided the type of `x` is an auto-init type (or, if `x` is ghost, a nonempty type). (If `x` is of a possibly nonempty type, then `x := *;` is still allowed, but it sets `x` to a value of its type only if the type actually contains a value. Therefore, when `x` is of a possibly empty type, `x := *;` does not count as a definite assignment to `x`.)

Note that auto-initialization is nondeterministic. Dafny only guarantees that each value it assigns to a variable of an auto-init type is *some* value of the type. Indeed, a variable may be auto-initialized to different values in different runs of the program or even at different times during the same run of the program. In other words, Dafny does not guarantee the "zero-equivalent value" initialization that some languages do. Along these lines, also note that the `witness` value provided in some subset-type declarations is not necessarily the value chosen by auto-initialization, though it does esstablish that the type is an auto-init type.

In some programs (for example, in some test programs), it is desirable to avoid nondeterminism. For that purpose, Dafny provides an `--enforce-determinism` option. It forbids use of any program statement that may have nondeterministic behavior and it disables auto-initialization. This mode enforces definite assignments everywhere, going beyond what the strict mode does by enforcing definite assignment also for fields and array elements. It also forbids the use of `iterator` declarations and `constructor`-less `class` declarations. It is up to a user's build process to ensure that `--enforce-determinism` is used consistently throughout the program. (In the legacy CLI, this mode is enabled by `/definiteAssignment:3`.)

This document, which is intended for developers of the Dafny tool itself, has more detail on auto-initialization and how it is implemented.

Finally, note that `--relax-definite-assignment=false` is the default in the command-based CLI, but, for backwards compatibility, the relaxed rules ('/definiteAssignment:1) are still the default in the legacy CLI.

12.7. Well-founded Orders

The well-founded order relations for a variety of built-in types in Dafny are given in the following table:

type of `X` and `x`	`x` strictly below `X`
`bool`	`X && !x`
`int`	`x < X && 0 <= X`
`real`	`x <= X - 1.0 && 0.0 <= X`
`set<T>`	`x` is a proper subset of `X`
`multiset<T>`	`x` is a proper multiset-subset of `X`
`seq<T>`	`x` is a consecutive proper sub-sequence of `X`
`map<K, V>`	`x.Keys` is a proper subset of `X.Keys`
inductive datatypes	`x` is structurally included in `X`
reference types	`x == null && X != null`
coinductive datatypes	`false`
type parameter	`false`
arrow types	`false`

Also, there are a few relations between the rows in the table above. For example, a datatype value `x` sitting inside a set that sits inside another datatype value `X` is considered to be strictly below `X`. Here's an illustration of that order, in a program that verifies:

```
datatype D = D(s: set<D>)

method TestD(dd: D) {
  var d := dd;
  while d != D({})
    decreases d
  {
    var x :| x in d.s;
    d := x;
  }
}
```

12.8. Quantifier instantiation rules

During verification, when Dafny knows that a universal quantifier is true, such as when verifying the body of a function that has the requires clause `forall x :: f(x) == 1`, it may instantiate the quantifier. Instantiation means Dafny will pick a value for all the variables of the quantifier, leading to a new expression, which it hopes to use to prove an assertion. In the above example, instantiating using `3` for `x` will lead to the expression `f(3) == 1`.

For each universal quantifier, Dafny generates rules to determine which instantiations are worthwhile doing. We call these rules triggers, a term that originates from SMT solvers. If Dafny can not generate triggers for a specific quantifier, it falls back to a set of generic rules. However, this is likely to be problematic, since the generic rules can cause many useless instantiations, leading to verification timing out or failing to proof a valid assertion. When the generic rules are used, Dafny emits a warning telling the user no triggers were found for the quantifier, indicating the Dafny program should be changed so Dafny can find triggers for this quantifier.

Here follows the approach Dafny uses to generate triggers based on a quantifier. Dafny finds terms in the quantifier body where a quantified variable is used in an operation, such as in a function application `P(x)`, array access `a[x]`, member accesses `x.someField`, or set membership tests `x in S`. To find a trigger, Dafny must find a set of such terms so that each quantified variable is used. You can investigate which triggers Dafny finds by hovering over quantifiers in the IDE and looking for 'Selected triggers', or by using the options `--show-tooltips` when using the LCI.

There are particular expressions which, for technical reasons, Dafny can not use as part of a trigger. Among others, these expression include: match, let, arithmetic operations and logical connectives. For example, in the quantifier `forall x :: x in S   f(x) > f(x+1)`, Dafny will use `x in S` and `f(x)` as trigger terms, but will not use `x+1` or any terms that contain it. You can investigate which triggers Dafny can not use by hovering over quantifiers in the IDE and looking for 'Rejected triggers', or by using the options `--show-tooltips` when using the LCI.

Besides not finding triggers, another problematic situation is when Dafny was able to generate triggers, but believes the triggers it found may still cause useless instantiations because they create matching loops. Dafny emits a warning when this happens, indicating the Dafny program should be changed so Dafny can find triggers for this quantifier that do not cause matching loops.

To understand matching loops, one needs to understand how triggers are used. During a single verification run, such as verifying a method or function, Dafny maintains a set of expressions which it believes to be true, which we call the ground terms. For example, in the body of a method, Dafny knows the requires clauses of that method hold, so the expressions in those will be ground terms. When Dafny steps through the statements of the body, the set of ground terms grows. For example, when an assignment `var x := 3` is evaluated, a ground term `x == 3` will be added. Given a universal quantifier that's a ground term, Dafny will try to pattern match its triggers on sub-expressions of other ground terms. If the pattern matches, that sub-expression is used to instantiate the quantifier.

Dafny makes sure not to perform the exact same instantiation twice. However, if an instantiation leads to a new term that also matches the trigger, but is different from the term used for the instantiation, the quantifier may be instantiated too often, an event we call a matching loop. For example, given the ground terms `f(3)` and `forall x {f(x)} :: f(x) + f(f(x))`, where `{f(x)}` indicates the trigger for the quantifier, Dafny may instantiate the quantifier using `3` for `x`. This creates a new ground term `f(3) + f(f(3))`, of which the right hand side again matches the trigger, allowing Dafny to instantiate the quantifier again using `f(3)` for `x`, and again and again, leading to an unbounded amount of instantiations.

Even existential quantifiers need triggers. This is because when Dafny determines an existential quantifier is false, for example in the body of a method that has `requires !exists x :: f(x) == 2`, Dafny will use a logical rewrite rule to change this existential into a universal quantifier, so it becomes `requires forall x :: f(x) != 2`. Before verification, Dafny can not determine whether quantifiers will be determined to be true or false, so it must assume any quantifier may turn into a universal quantifier, and thus they all need triggers. Besides quantifiers, comprehensions such as set and map comprehensions also need triggers, since these are modeled using universal quantifiers.

Dafny may report 'Quantifier was split into X parts'. This occurs when Dafny determines it can only generate good triggers for a quantifier by splitting it into multiple smaller quantifiers, whose aggregation is logically equivalent to the original one. To maintain logical equivalence, Dafny may have to generate more triggers than if the split had been done manually in the Dafny source file. An example is the expression `forall x :: P(x) && (Q(x) = P(x+1))`, which Dafny will split into

```
forall x {P(x)} {Q(x)} :: P(x) &&
forall x {(Q(x)} :: Q(x) = P(x+1)
```

Note the trigger `{Q(x)}` in the first quantifier, which was added to maintain equivalence with the original quantifier. If the quantifier had been split in source, only the trigger `{P(x)}` would have been added for `forall x :: P(x)`.

13. Dafny User's Guide

Most of this document describes the Dafny programming language. This section describes the `dafny` tool, a combined verifier and compiler that implements the Dafny language.

The development of the Dafny language and tool is a GitHub project at https://github.com/dafny-lang/dafny. The project is open source, with collaborators from various organizations; additional contributors are welcome. The software itself is licensed under the MIT license.

13.1. Introduction

The `dafny` tool implements the following primary capabilities, implemented as various *commands* within the `dafny` tool:

- checking that the input files represent a valid Dafny program (i.e., syntax, grammar and name and type resolution);
- verifying that the program meets its specifications, by translating the program to verification conditions and checking those with Boogie and an SMT solver, typically Z3;
- compiling the program to a target language, such as C#, Java, Javascript, Go (and others in development);
- running the executable produced by the compiler.

In addition there are a variety of other capabilities, such as formatting files, also implemented as commands; more such commands are expected in the future.

13.2. Installing Dafny

13.2.1. Command-line tools

The instructions for installing `dafny` and the required dependencies and environment are described on the Dafny wiki: https://github.com/dafny-lang/dafny/wiki/INSTALL. They are not repeated here to avoid replicating information that easily becomes inconsistent and out of date. The dafny tool can also be installed using `dotnet tool install --global dafny` (presuming that `dotnet` is already installed on your system).

Most users will find it most convenient to install the pre-built Dafny binaries available on the project release site or using the `dotnet` CLI. As is typical for Open Source projects, dafny can also be built directly from the source files maintained in the github project.

Current and past Dafny binary releases can be found at https://github.com/dafny-lang/dafny/releases for each supported platform. Each release is a .zip file with a name combining the release name and the platform. Current platforms are Windows 11, Ubuntu 20 and later, and MacOS 10.14 and later.

The dafny tool is distributed as a standalone executable. A compatible version of the required Z3 solver is included in the release. There are additional dependencies that are needed to compile dafny to particular target languages, as described in the release instructions. A development environment to *build* dafny from source requires additional dependencies, described here.

13.2.2. IDEs for Dafny

Dafny source files are text files and can of course be edited with any text editor. However, some tools provide syntax-aware features:

- VSCode, a cross-platform editor for many programming languages has an extension for Dafny. VSCode is available here and the Dafny extension can be installed from within VSCode. The extension provides syntax highlighting, in-line parser, type and verification errors, code navigation, counter-example display and gutter highlights.
- There is a Dafny mode for Emacs.
- An old Visual Studio plugin is no longer supported

Information about installing IDE extensions for Dafny is found on the Dafny INSTALL page in the wiki.

More information about using VSCode IDE is here.

13.3. Dafny Programs and Files

A Dafny program is a set of modules. Modules can refer to other modules, such as through `import` declarations or `refines` clauses. A Dafny program consists of all the modules needed so that all module references are resolved. Dafny programs are contained in files that have a `.dfy` suffix. Such files each hold some number of top-level declarations. Thus a full program may be distributed among multiple files. To apply the `dafny` tool to a Dafny program, the `dafny` tool must be given all the files making up a complete program (or, possibly, more than one program at a time). This can be effected either by listing all of the files by name on the command-line or by using `include` directives within a file to stipulate what other files contain modules that the given files need. Thus the complete set of modules are all the modules in all the files listed on the command-line or referenced, recursively, by `include` directives within those files. It does not matter if files are repeated either as includes or on the command-line.[14]

All files recursively included are always parsed and type-checked. However, which files are verified, built, run, or processed by other dafny commands depends on the individual command. These commands are described in Section 13.6.1.

13.3.1. Dafny Verification Artifacts: the Library Backend and .doo Files

As of Dafny 4.1, `dafny` now supports outputting a single file containing a fully-verified program along with metadata about how it was verified. Such files use the extension `.doo`, for Dafny Output Object, and can be used as input anywhere a `.dfy` file can be.

`.doo` files are produced by an additional backend called the "Dafny Library" backend, identified with the name `lib` on the command line. For example, to build multiple Dafny files into a single build artifact for shared reuse, the command would look something like:

[14]Files may be included more than once or both included and listed on the command line. Duplicate inclusions are detected and each file processed only once. For the purpose of detecting duplicates, file names are considered equal if they have the same absolute path, compared as case-sensitive strings (regardless of whether the underlying file-system is case sensitive). Using symbolic links may make the same file have a different absolute path; this will generally cause duplicate declaration errors.

```
dafny build -t:lib A.dfy B.dfy C.dfy --output:MyLib.doo
```

The Dafny code contained in a `.doo` file is not re-verified when passed back to the `dafny` tool, just as included files and those passed with the `--library` option are not. Using `.doo` files provides a guarantee that the Dafny code was in fact verified, however, and therefore offers protection against build system mistakes. `.doo` files are therefore ideal for sharing Dafny libraries between projects.

`.doo` files also contain metadata about the version of Dafny used to verify them and the values of relevant options that affect the sound separate verification and compilation of Dafny code, such as `--unicode-char`. This means attempting to use a library that was built with options that are not compatible with the currently executing command options will lead to errors. This also includes attempting to use a `.doo` file built with a different version of Dafny, although this restriction may be lifted in the future.

A `.doo` file is a compressed archive of multiple files, similar to the `.jar` file format for Java packages. The exact file format is internal and may evolve over time to support additional features.

Note that the library backend only supports the newer command-style CLI interface.

13.3.2. Dafny Translation Artifacts: .dtr Files

Some options, such as `--outer-module` or `--optimize-erasable-datatype-wrapper`, affect what target language code the same Dafny code is translated to. In order to translate Dafny libaries separately from their consuming codebases, the translation process for consuming code needs to be aware of what options were used when translating the library.

For example, if a library defines a `Foo()` function in an `A` module, but `--outer-module org.coolstuff.foolibrary.dafnyinternal` is specified when translating the library to Java, then a reference to `A.Foo()` in a consuming Dafny project needs to be translated to `org.coolstuff.foolibrary.dafnyinternal.A.Foo()`, independently of what value of `--outer-module` is used for the consuming project.

To meet this need, `dafny translate` also outputs a `<program-name>-<target id>.dtr` Dafny Translation Record file. Like `.doo` files, `.dtr` files record all the relevant options that were used, in this case relevant to translation rather than verification. These files can be provided to future calls to `dafny translate` using the `--translation-record` option, in order to provide the details of how various libraries provided with the `--library` flag were translated.

Currently `--outer-module` is the only option recorded in `.dtr` files, but more relevant options will be added in the future. A later version of Dafny will also require `.dtr` files that cover all modules that are defined in `--library` options, to support checking that all relevant options are compatible.

13.4. Dafny Standard Libraries

As of Dafny 4.4, the `dafny` tool includes standard libraries that any Dafny code base can depend on. For now they are only available when the `--standard-libraries` option is provided, but they will likely be available by default in the next major version of Dafny.

See https://github.com/dafny-lang/dafny/blob/master/Source/DafnyStandardLibraries/README.md for details.

13.5. Dafny Code Style

There are coding style conventions for Dafny code, recorded here. Most significantly, code is written without tabs and with a 2 space indentation. Following code style conventions improves readability but does not alter program semantics.

13.6. Using Dafny From the Command Line

`dafny` is a conventional command-line tool, operating just like other command-line tools in Windows and Unix-like systems. In general, the format of a command-line is determined by the shell program that is executing the command-line (.e.g., bash, the windows shell, COMMAND, etc.), but is expected to be a series of space-separated "words", each representing a command, option, option argument, file, or folder.

13.6.1. dafny commands

As of v3.9.0, `dafny` uses a command-style command-line (like `git` for example); prior to v3.9.0, the command line consisted only of options and files. It is expected that additional commands will be added in the future. Each command may have its own subcommands and its own options, in addition to generally applicable options. Thus the format of the command-line is a command name, followed by options and files: `dafny <command> <options> <files>`; the command-name must be the first command-line argument.

The command-line `dafny --help` or `dafny -h` lists all the available commands.

The command-line `dafny <command> --help` (or `-h` or `-?`) gives help information for that particular <command>, including the list of options. Some options for a particular command are intended only for internal tool development; those are shown using the `--help-internal` option instead of `--help`.

Also, the command-style command-line has modernized the syntax of options; they are now POSIX-compliant. Like many other tools, options now typically begin with a double hyphen, with some options having a single-hyphen short form, such as `--help` and `-h`.

If no <command> is given, then the command-line is presumed to use old-style syntax, so any previously written command-line will still be valid.

`dafny` recognizes the commands described in the following subsections. The most commonly used are `dafny verify`, `dafny build`, and `dafny run`.

The command-line also expects the following: - Files are designated by absolute paths or paths relative to the current working directory. A command-line argument not matching a known option is considered a filepath, and likely one with an unsupported suffix, provoking an error message. - Files containing dafny code must have a `.dfy` suffix. - There must be at least one `.dfy` file (except when using `--stdin` or in the case of `dafny format`, see the Dafny format section) -

The command-line may contain other kinds of files appropriate to the language that the Dafny files are being compiled to. The kind of file is determined by its suffix. - Escape characters are determined by the shell executing the command-line. - Per POSIX convention, the option `--` means that all subsequent command-line arguments are not options to the dafny tool; they are either files or arguments to the `dafny run` command. - If an option is repeated (e.g., with a different argument), then the later instance on the command-line supersedes the earlier instance, with just a few options accumulating arguments. - If an option takes an argument, the option name is followed by a `:` or `=` or whitespace and then by the argument value; if the argument itself contains white space, the argument must be enclosed in quotes. It is recommended to use the `:` or `=` style to avoid misinterpretation or separation of a value from its option. - Boolean options can take the values `true` and `false` (or any case-insensitive version of those words). For example, the value of `--no-verify` is by default `false` (that is, do verification). It can be explicitly set to true (no verification) using `--no-verify`, `--no-verify:true`, `--no-verify=true`, `--noverify true`; it can be explicitly set false (do verification) using `--no-verify:false` or `--no-verify=false` or `--no-verify false`. - There is a potential ambiguity when the form `--option value` is used if the value is optional (such as for boolean values). In such a case an argument afer an option (that does not have an argument given with `:` or `=`) is interpreted as the value if it is indeed a valid value for that option. However, better style advises always using a ':' or '=' to set option values. No valid option values in dafny look like filenames or begin with `--`.

13.6.1.1. Options that are not associated with a command

A few options are not part of a command. In these cases any single-hyphen spelling also permits a spelling beginning with '/'. - `dafny --help` or `dafny -h` lists all the available commands - `dafny -?` or `dafny -help` list all legacy options - `dafny --version` (or `-version`) prints out the number of the version this build of dafny implements

13.6.1.2. `dafny resolve`

The `dafny resolve` command checks the command-line and then parses and typechecks the given files and any included files.

The set of files considered by `dafny` are those listed on the command-line, including those named in a `--library` option, and all files that are named, recursively, in `include` directives in files in the set being considered by the tool.

The set of files presented to an invocation of the `dafny` tool must contain all the declarations needed to resolve all names and types, else name or type resolution errors will be emitted.

`dafny` can parse and verify sets of files that do not form a complete program because they are missing the implementations of some constructs such as functions, lemmas, and loop bodies.[15] However, `dafny` will need all implementations in order to compile a working executable.

The options relevant to this command are - those relevant to the command-line itself - `--allow-warnings` — return a success exit code, even when there are warnings

- those that affect dafny` as a whole, such as

[15]Unlike some languages, Dafny does not allow separation of declaration and implementation of methods, functions and types in separate files, nor, for that matter, separation of specification and declaration. Implementations can be omitted simply by leaving them out of the declaration (or a lemma, for example). However, a combination of `traits` and `classes` can achieve a separation of interface and specification from implementation.

  - `--cores` — set the number of cores dafny should use
  - `--show-snippets` — emit a line or so of source code along with an error message
  - `--library` — include this file in the program, but do not verify or compile it (multiple such library files can be listed using multiple instances of the `--library` option)
  - `--stdin` – read from standard input
- those that affect the syntax of Dafny, such as
  - `--prelude`
  - `--unicode-char`
  - `--function-syntax`
  - `--quantifier-syntax`
  - `--track-print-effects`
  - `--warn-shadowing`
  - `--warn-missing-constructor-parentheses`

13.6.1.3. `dafny verify` The `dafny verify` command performs the `dafny resolve` checks and then attempts to verify each declaration in the program.

A guide to controlling and aiding the verification process is given in a later section.

To be considered *verified* all the methods in all the files in a program must be verified, with consistent sets of options, and with no unproven assumptions (see `dafny audit` for a tool to help identify such assumptions).

Dafny works *modularly*, meaning that each method is considered by itself, using only the specifications of other methods. So, when using the dafny tool, you can verify the program all at once or one file at a time or groups of files at a time. On a large program, verifying all files at once can take quite a while, with little feedback as to progress, though it does save a small amount of work by parsing all files just once. But, one way or another, to have a complete verification, all implementations of all methods and functions must eventually be verified.

- By default, only those files listed on the command-line are verified in a given invocation of the `dafny` tool.
- The option `--verify-included-files` (`-verifyAllModules` in legacy mode) forces the contents of all non-library files to be verified, whether they are listed on the command-line or recursively included by files on the command-line.
- The `--library` option marks files that are excluded from `--verify-included-files`. Such a file may also, but need not, be the target of an `include` directive in some file of the program; in any case, such files are included in the program but not in the set of files verified (or compiled). The intent of this option is to mark files that should be considered as libraries that are independently verified prior to being released for shared use.
- Verifying files individually is equivalent to verifying them in groups, presuming no other changes. It is also permitted to verify completely disjoint files or programs together in a single run of `dafny`.

Various options control the verification process, in addition to all those described for `dafny resolve`.

- What is verified
  - `--verify-included-files` (when enabled, all included files are verified, except library files, otherwise just those files on the command-line)
  - `--relax-definite-assignment`
  - `--track-print-effects`
  - `--disable-nonlinear-arithmetic`
  - `--filter-symbol`
- Control of the proof engine
  - `--manual-lemma-induction`
  - `--verification-time-limit`
  - `--boogie`
  - `--solver-path`

13.6.1.4. `dafny translate <language>`

The `dafny translate` command translates Dafny source code to source code for another target programming language. The command always performs the actions of `dafny resolve` and, unless the `--no-verify` option is specified, does the actions of `dafny verify`. The language is designated by a subcommand argument, rather than an option, and is required. The current set of supported target languages is - cs (C#) - java (Java) - js (JavaScript) - py (Python) - go (Go) - cpp (C++ – but only limited support)

In addition to generating the target source code, `dafny` may generate build artifacts to assist in compiling the generated code. The specific files generated depend on the target programming language. More detail is given in the section on compilation.

The `dafny` tool intends that the compiled program in the target language be a semantically faithful rendering of the (verified) Dafny program. However, resource and language limitations make this not always possible. For example, though Dafny can express and reason about arrays of unbounded size, not all target programming languages can represent arrays larger than the maximum signed 32-bit integer.

Various options control the translation process, in addition to all those described for `dafny resolve` and `dafny verify`.

- General options:
  - `--no-verify` — turns off all attempts to verify the program
  - `--verbose` — print information about generated files
- The translation results
  - `--output` (or `-o`) — location of the generated file(s) (this specifies a file path and name; a folder location for artifacts is derived from this name)
  - `--include-runtime` — include the Dafny runtime for the target language in the generated artifacts
  - `--optimize-erasable-datatype-wrapper`
  - `--enforce-determinism`
  - `--test-assumptions` — (experimental) inserts runtime checks for unverified assumptions when they are compilable

13.6.1.5. `dafny build`

The `dafny build` command runs `dafny translate` and then compiles the result into an executable artifact for the target platform, such as a `.exe` or `.dll` or executable `.jar`, or just the source code for an interpreted language. If the Dafny program does not have a Main entry point, then the build command creates a library, such as a `.dll` or `.jar`. As with `dafny translate`, all the previous phases are also executed, including verification (unless `--no-verify` is a command-line option). By default, the generated file is in the same directory and has the same name with a different extension as the first .dfy file on the command line. This location and name can be set by the `--output` option.

The location of the `Main` entry point is described [here](#sec-user-guide-main}.

There are no additional options for `dafny build` beyond those for `dafny translate` and the previous compiler phases.

Note that `dafny build` may do optimizations that `dafny run` does not.

Details for specific target platforms are described in Section 25.7.

13.6.1.6. `dafny run`

The `dafny run` command compiles the Dafny program and then runs the resulting executable. Note that `dafny run` is engineered to quickly compile and launch the program; `dafny build` may take more time to do optimizations of the build artifacts.

The form of the `dafny run` command-line is slightly different than for other commands. - It permits just one `.dfy` file, which must be the file containing the `Main` entry point; the location of the `Main` entry point is described [here](#sec-user-guide-main}. - Other files are included in the program either by `include` directives within that one file or by the `--input` option on the command-line. - Anything that is not an option and is not that one dfy file is an argument to the program being run (and not to dafny itself). - If the `--` option is used, then anything after that option is a command-line argument to the program being run.

During development, users must use `dafny run --allow-warnings` if they want to run their Dafny code when it contains warnings.

Here are some examples: - `dafny run A.dfy` – builds and runs the Main program in `A.dfy` with no command-line arguments - `dafny run A.dfy --no-verify` – builds the Main program in `A.dfy` using the `--no-verify` option, and then runs the program with no command-line arguments - `dafny run A.dfy -- --no-verify` – builds the Main program in `A.dfy` (*not* using the `--no-verify` option), and then runs the program with one command-line argument, namely `--no-verify` - `dafny run A.dfy 1 2 3 B.dfy` – builds the Main program in `A.dfy` and then runs it with the four command-line arguments `1 2 3 B.dfy` - `dafny run A.dfy 1 2 3 --input B.dfy` – builds the Main program in `A.dfy` and `B.dfy`, and then runs it with the three command-line arguments `1 2 3` - `dafny run A.dfy 1 2 -- 3 -quiet` – builds the Main program in `A.dfy` and then runs it with the four command-line arguments `1 2 3 -quiet`

Each time `dafny run` is invoked, the input Dafny program is compiled before it is executed. If a Dafny program should be run more than once, it can be faster to use `dafny build`, which enables compiling a Dafny program once and then running it multiple times.

Note: `dafny run` will typically produce the same results as the executables produced by `dafny build`. The only expected differences are these: - performance — `dafny run` may not optimize as much as `dafny build` - target-language-specific configuration issues — e.g. encoding issues: `dafny run` sets language-specific flags to request UTF-8 output for the `print` statement in all languages, whereas `dafny build` leaves language-specific runtime configuration to the user.

13.6.1.7. `dafny server`

The `dafny server` command starts the Dafny Language Server, which is an LSP-compliant implementation of Dafny. The Dafny VSCode extension uses this LSP implementation, which in turn uses the same core Dafny implementation as the command-line tool.

The Dafny Language Server is described in more detail here.

13.6.1.8. `dafny audit`

The `dafny audit` command reports issues in the Dafny code that might limit the soundness claims of verification.

This command is under development.

The command executes the `dafny resolve` phase (accepting its options) and has the following additional options:

- `--report-file:<report-file>` — spcifies the path where the audit report file will be stored. Without this option, the report will be issued as standard warnings, written to standard-out.
- `--report-format:<format>` — specifies the file format to use for the audit report. Supported options include:
  - 'txt, 'text': plain text in the format of warnings
  - 'html': standalone HTML ('html')
  - 'md', 'markdown', 'md-table', 'markdown-table': a Markdown table
  - 'md-ietf', 'markdown-ietf': an IETF-language document in Markdown format
  - The default is to infer the format from the filename extension
- `--compare-report` — compare the report that would have been generated with the existing file given by –report-file, and fail if they differ.

The command emits exit codes of - 1 for command-line errors - 2 for parsing, type-checking or serious errors in running the auditor (e.g. failure to write a report or when report comparison fails) - 0 for normal operation, including operation that identifies audit findings

It also takes the `--verbose` option, which then gives information about the files being formatted.

The `dafny audit` command currently reports the following:

- Any declaration marked with the `{:axiom}` attribute. This is typically used to mark that a lemma with no body (and is therefore assumed to always be true) is intended as an axiom. The key purpose of the `audit` command is to ensure that all assumptions are intentional and acknowledged. To improve assurance, however, try to provide a proof.

- Any declaration marked with the `{:verify false}` attribute, which tells the verifier to skip verifying this declaration. Removing the attribute and providing a proof will improve assurance.

- Any declaration marked with the `{:extern}` attribute that has at least one `requires` or `ensures` clause. If code implemented externally, and called from Dafny, has an `ensures` clause, Dafny assumes that it satisfies that clause. Since Dafny cannot prove properties about code written in other languages, adding tests to provide evidence that any `ensures` clauses do hold can improve assurance. The same considerations apply to `requires` clauses on Dafny code intended to be called from external code.
- Any definition with an `assume` statement in its body. To improve assurance, attempt to convert it to an `assert` statement and prove that it holds. Such a definition will not be compilable unless the statement is also marked with `{:axiom}`. Alternatively, converting it to an `expect` statement will cause it to be checked at runtime.
- Any method marked with `decreases *`. Such a method may not terminate. Although this cannot cause an unsound proof, in the logic of Dafny, it's generally important that any non-termination be intentional.
- Any `forall` statement without a body. This is equivalent to an assumption of its conclusion. To improve assurance, provide a body that proves the conclusion.
- Any loop without a body. This is equivalent to an assumption of any loop invariants in the code after the loop. To improve assurance, provide a body that establishes any stated invariants.
- Any declaration with no body and at least one `ensures` clause. Any code that calls this declaration will assume that all `ensures` clauses are true after it returns. To improve assurance, provide a body that proves that any `ensures` clauses hold.

13.6.1.9. `dafny format` Dafny supports a formatter, which for now only changes the indentation of lines in a Dafny file, so that it conforms to the idiomatic Dafny code formatting style. For the formatter to work, the file should be parsed correctly by Dafny.

There are four ways to use the formatter:

- `dafny format <one or more .dfy files or folders>` formats the given Dafny files and the Dafny files in the folders, recursively, altering the files in place. For example, `dafny format .` formats all the Dafny files recursively in the current folder.
- `dafny format --print <files and/or folders>` formats each file but instead of altering the files, output the formatted content to stdout
- `dafny format --check <files and/or folders>` does not alter files. It will print a message concerning which files need formatting and return a non-zero exit code if any files would be changed by formatting.

You can also use `--stdin` instead of providing a file, to format a full Dafny file from the standard input. Input files can be named along with `--stdin`, in which case both the files and the content of the stdin are formatted.

Each version of `dafny format` returns a non-zero return code if there are any command-line or parsing errors or if –check is stipulated and at least one file is not the same as its formatted version.

`dafny format` does not necessarily report name or type resolution errors and does not attempt verification.

13.6.1.10. `dafny test` This command (verifies and compiles the program and) runs every method in the program that is annotated with the `{:test}` attribute. Verification can be disabled using the `--no-verify` option. `dafny test` also accepts all other options of the `dafny build` command. In particular, it accepts the `--target` option that specifies the programming language used in the build and execution phases.

`dafny test` also accepts these options:

- `-spill-translation` - (default disabled) when enabled the compilation artifacts are retained
- `--output` - gives the folder and filename root for compilation artifacts
- `--methods-to-test` - the value is a (.NET) regular expression that is matched against the fully qualified name of the method; only those methods that match are tested
- `--coverage-report` - the value is a directory in which Dafny will save an html coverage report highlighting parts of the program that execution of the tests covered.

The order in which the tests are run is not specified.

For example, this code (as the file `t.dfy`)

```
method {:test} m() {
  mm();
  print "Hi!\n";
}

method mm() {
  print "mm\n";
}

module M {
  method {:test} q() {
    print 42, "\n";
  }
}

class A {
  static method {:test} t() { print "T\n"; }
}
```

and this command-line

```
dafny test --no-verify t.dfy
```

produce this output text:

```
M.q: 42
PASSED
A.t: T
PASSED
m: mm
Hi!
PASSED
```

and this command-line

```
dafny test --no-verify --methods-to-test='m' t.dfy
```

produces this output text:

```
m: mm
Hi!
PASSED
```

13.6.1.11. `dafny doc` [Experimental] The `dafny doc` command generates HTML documentation pages describing the contents of each module in a set of files, using the documentation comments in the source files. This command is experimental; user feedback and contributor PRs on the layout of information and the navigation are welcome.

- The format of the documentation comments is described here.
- The `dafny doc` command accepts either files or folders as command-line arguments. A folder represents all the `.dfy` files contained recursively in that folder. A file that is a `.toml` project file represents all the files and options listed in the project file.
- The command first parses and resolves the given files; it only proceeds to produce documentation if type resolution is successful (on all files). All the command-line options relevant to `dafny resolve` are available for `dafny doc`.
- The value of the `--output` option is a folder in which all the generated files will be placed. The default location is `./docs`. The folder is created if it does not already exist. Any existing content of the folder is overwritten.
- If `--verbose` is enabled, a list of the generated files is emitted to stdout.
- The output files contain information stating the source .dfy file in which the module is declared. The `--file-name` option controls the form of the filename in that information:
  - –file-name:none – no source file information is emitted
  - –file-name:name – (default) just the file name is emitted (e.g., `Test.dfy`)
  - –file-name:absolute – an absolute full path is emitted
  - –file-name:relative= – a file name relative to the given prefix is emitted
- If `--modify-time` is enabled, then the generated files contain information stating the last modified time of the source of the module being documented.
- The `--program-name` option states text that will be included in the heading of the TOC and index pages

The output files are HTML files, all contained in the given folder, one per module plus an `index.html` file giving an overall table of contents and a `nameindex.html` file containing an alphabetical by name list of all the declarations in all the modules. The documentation for the root module is in `_.html`.

13.6.1.12. `dafny generate-tests`

This *experimental* command allows generating tests from Dafny programs. The tests provide complete coverage of the implementation and one can execute them using the `dafny test` command. Dafny can target different notions of coverage while generating tests, with basic-block coverage being the recommended setting. Basic blocks are extracted from the Boogie representation of the Dafny program, with one basic block corresponding to a statement or a non-short-circuiting subexpression in the Dafny code. The underlying implementation uses the verifier to reason about the reachability of different basic blocks in the program and infers necessary test inputs from counterexamples.

For example, this code (as the file `program.dfy`)

```
module M {
  function {:testEntry} Min(a: int, b: int): int {
    if a < b then a else b
  }
}
```

and this command-line

```
dafny generate-tests Block program.dfy
```

produce two tests:

```
include "program.dfy"
module UnitTests {
  import M
  method {:test} Test0() {
    var r0 := M.Min(0, 0);
  }
  method {:test} Test1() {
    var r0 := M.Min(0, 1);
  }
}
```

The two tests together cover every basic block within the `Min` function in the input program. Note that the `Min` function is annotated with the `{:testEntry}` attribute. This attribute marks `Min` as the entry point for all generated tests, and there must always be at least one method or function so annotated. Another requirement is that any top-level declaration that is not itself a module (such as class, method, function, etc.) must be a member of an explicitly named module, which is called `M` in the example above.

This command is under development and not yet fully functional.

13.6.1.13. `Inlining`

By default, when asked to generate tests, Dafny will produce *unit tests*, which guarantee coverage of basic blocks within the method they call but not within any of its callees. By contrast, system-level tests can guarantee coverage of a large part of the program while at the same time using a single method as an entry point. In order to prompt Dafny to generate system-level tests, one must use the `{:testInline}` attribute.

For example, this code (as the file `program.dfy`)

```
module M {
  function {:testInline} Min(a: int, b: int): int {
    if a < b then a else b
  }
  method {:testEntry} Max(a: int, b: int) returns (c: int)
    // the tests convert the postcondition below into runtime check:
    ensures c == if a > b then a else b
  {
    return -Min(-a, -b);
  }
}
```

and this command-line

```
dafny generate-tests Block program.dfy
```

produce two tests:

```
include "program.dfy"
module UnitTests {
  import M
  method {:test} Test0() {
    var r0 := M.Max(7719, 7720);
    expect r0 == if 7719 > 7720 then 7719 else 7720;
  }
  method {:test} Test1() {
    var r0 := M.Max(1, 0);
    expect r0 == if 1 > 0 then 1 else 0;
  }
}
```

Without the use of the `{:testInline}` attribute in the example above, Dafny will only generate a single test because there is only one basic-block within the `Max` method itself – all the branching occurs withing the `Min` function. Note also that Dafny automatically converts all non-ghost postconditions on the method under tests into `expect` statements, which the compiler translates to runtime checks in the target language of choice.

When the inlined method or function is recursive, it might be necessary to unroll the recursion several times to get adequate code coverage. The depth of recursion unrolling should be provided as an integer argument to the `{:testInline}` attribute. For example, in the program below, the function `Mod3` is annotated with `{:testInline 2}` and will, therefore, be unrolled twice during test generation. The function naively implements division by repeatedly and recursively subtracting `3` from its argument, and it returns the remainder of the division, which is one of the three base cases. Because the `TestEntry` method calls `Mod3` with an argument that is guaranteed to be at least 3, the base case will never occur on first iteration, and the function must be unrolled at least twice for Dafny to generate tests covering any of the base cases:

```
module M {
  function {:testInline 2} Mod3 (n: nat): nat
    decreases n
  {
    if n == 0 then 0 else
    if n == 1 then 1 else
    if n == 2 then 2 else
    Mod3(n-3)
  }
  method {:testEntry} TestEntry(n: nat) returns (r: nat)
    requires n >= 3
  {
    r := Mod3(n);
  }
}
```

13.6.1.14. `Command Line Options` Test generation supports a number of command-line options that control its behavior.

The first argument to appear after the `generate-test` command specifies the coverage criteria Dafny will attempt to satisfy. Of these, we recommend basic-block coverage (specified with keyword `Block`), which is also the coverage criteria used throughout the relevant parts of this reference manual. The alternatives are path coverage (`Path`) and block coverage after inlining (`InlinedBlock`). Path coverage provides the most diverse set of tests but it is also the most expensive in terms of time it takes to produce these tests. Block coverage after inlining is a call-graph sensitive version of block coverage - it takes into account every block in a given method for every path through the call-graph to that method.

The following is a list of command-line-options supported by Dafny during test generation:

- `--verification-time-limit` - the value is an integer that sets a timeout for generating a single test. The default is 20 seconds.
- `--length-limit` - the value is an integer that is used to limit the lenghts or all sequences and sizes of all maps and sets that test generation will consider as valid test inputs. This can sometimes be necessary to prevent test generation from creating unwieldy tests with excessively long strings or large maps. This option is disabled by default
- `--coverage-report` - the value is a directory in which Dafny will save an html coverage report highlighting parts of the program that the generated tests are expected to cover.
- `--print-bpl` - the value is the name of the file to which Dafny will save the Boogie code used for generating tests. This options is mostly useful for debugging test generation functionality itself.
- `--force-prune` - this flag enables axiom pruning, a feature which might significantly speed up test generation but can also reduce coverage or cause Dafny to produce tests that do not satisfy the preconditions.

Dafny will also automatically enforce the following options during test generation: `--enforce-determinism`, `/typeEncoding:p` (an option passed on to Boogie).

13.6.1.15. `dafny find-dead-code`

This *experimental* command finds dead code in a program, that is, basic-blocks within a method that are not reachable by any inputs that satisfy the method's preconditions. The underlying implementation is identical to that of `dafny generate-tests` command and can be controlled by the same command line options and method attributes.

For example, this code (as the file `program.dfy`)

```
module M {
  function {:testEntry} DescribeProduct(a: int): string {
    if a * a < 0
    then "Product is negative"
    else "Product is nonnegative"
  }
}
```

and this command-line

```
dafny find-dead-code program.dfy
```

produce this output:

```
program.dfy(5,9) is reachable.
program.dfy(3,4):initialstate is reachable.
program.dfy.dfy(5,9)#elseBranch is reachable.
program.dfy.dfy(4,9)#thenBranch is potentially unreachable.
Out of 4 basic blocks, 3 are reachable.
```

Dafny reports that the then branch of the condition is potentially unreachable because the verifier proves that no input can reach it. In this case, this is to be expected, since the product of two numbers can never be negative. In practice, `find-dead-code` command can produce both false positives (if the reachability query times out) and false negatives (if the verifier cannot prove true unreachability), so the results of such a report should always be reviewed.

This command is under development and not yet fully functional.

13.6.1.16. `dafny measure-complexity`

This *experimental* command reports complexity metrics of a program.

This command is under development and not yet functional.

13.6.1.17. Plugins

This execution mode is not a command, per se, but rather a command-line option that enables executing plugins to the dafny tool. Plugins may be either standalone tools or be additions to existing commands.

The form of the command-line is `dafny --plugin:<path-to-one-assembly[,argument]*>` or `dafny <command> --plugin:<path-to-one-assembly[,argument]*>` where the argument to

`--plugin` gives the path to the compiled assembly of the plugin and the arguments to be provided to the plugin.

More on writing and building plugins can be found in this section.

13.6.1.18. Legacy operation

Prior to implementing the command-based CLI, the `dafny` command-line simply took files and options and the arguments to options. That legacy mode of operation is still supported, though discouraged. The command `dafny -?` produces the list of legacy options. In particular, the common commands like `dafny verify` and `dafny build` are accomplished with combinations of options like `-compile`, `-compileTarget` and `-spillTargetCode`.

Users are encouraged to migrate to the command-based style of command-lines and the double-hyphen options.

13.6.2. In-tool help

As is typical for command-line tools, `dafny` provides in-tool help through the `-h` and `--help` options: - `dafny -h`, `dafny --help` list the commands available in the `dafny` tool - `dafny -?` lists all the (legacy) options implemented in `dafny` - `dafny <command> -h`, `dafny <command> --help`, `dafny <command> -?` list the options available for that command

13.6.3. dafny exit codes

The basic resolve, verify, translate, build, run and commands of dafny terminate with these exit codes.

- 0 – success
- 1 – invalid command-line arguments
- 2 – syntax, parse, or name or type resolution errors
- 3 – compilation errors
- 4 – verification errors

Errors in earlier phases of processing typically hide later errors. For example, if a program has parsing errors, verification or compilation will not be attempted.

Other dafny commands may have their own conventions for exit codes. However in all cases, an exit code of 0 indicates successful completion of the command's task and small positive integer values indicate errors of some sort.

13.6.4. dafny output

Most output from `dafny` is directed to the standard output of the shell invoking the tool, though some goes to standard error. - Command-line errors: these are produced by the dotnet CommandLineOptions package are directed to **standard-error** - Other errors: parsing, typechecking, verification and compilation errors are directed to **standard-out** - Non-error progress information also is output to **standard-out** - Dafny `print` statements, when executed, send output to **standard-out** - Dafny `expect` statements (when they fail) send a message to **standard-out**. - Dafny I/O libraries send output explicitly to either **standard-out or standard-error**

13.6.5. Project files

Commands on the Dafny CLI that can be passed a Dafny file can also be passed a Dafny project file. Such a project file may define which Dafny files the project contains and which Dafny options it uses. The project file must be a TOML file named `dfyconfig.toml` for it to work on both the CLI and in the Dafny IDE, although the CLI will accept any `.toml` file. Here's an example of a Dafny project file:

```toml
includes = ["src/**/*.dfy"]
excludes = ["**/ignore.dfy"]

base = ["../commonOptions.dfyconfig.toml"]

[options]
enforce-determinism = true
warn-shadowing = true
default-function-opacity = "Opaque"
```

- At most one `.toml` file may be named on the command-line; when using the command-line no `.toml` file is used by default.
- In the `includes` and `excludes` lists, the file paths may have wildcards, including `**` to mean any number of directory levels; filepaths are relative to the location of the `.toml` file in which they are named.
- Dafny will process the union of (a) the files on the command-line and (b) the files designated in the `.toml` file, which are those specified by the `includes`, omitting those specified by the `excludes`. The `excludes` does not remove any files that are listed explicitly on the command-line.
- Under the section `[options]`, any options from the Dafny CLI can be specified using the option's name without the `--` prefix.
- When executing a `dafny` command using a project file, any options specified in the file that can be applied to the command, will be. Options that can't be applied are ignored; options that are invalid for any dafny command trigger warnings.
- Options specified on the command-line take precedence over any specified in the project file, no matter the order of items on the command-line.
- When using a Dafny IDE based on the `dafny server` command, the IDE will search for project files by traversing up the file tree looking for the closest `dfyconfig.toml` file to the dfy being parsed that it can find. Options from the project file will override options passed to `dafny server`.
- The field 'base' can be used to let one project file inherit options from another. If an option is specified in both, then the value specified in the inheriting project is used. Includes from the inheritor override excludes from the base.

It's not possible to use Dafny project files in combination with the legacy CLI UI.

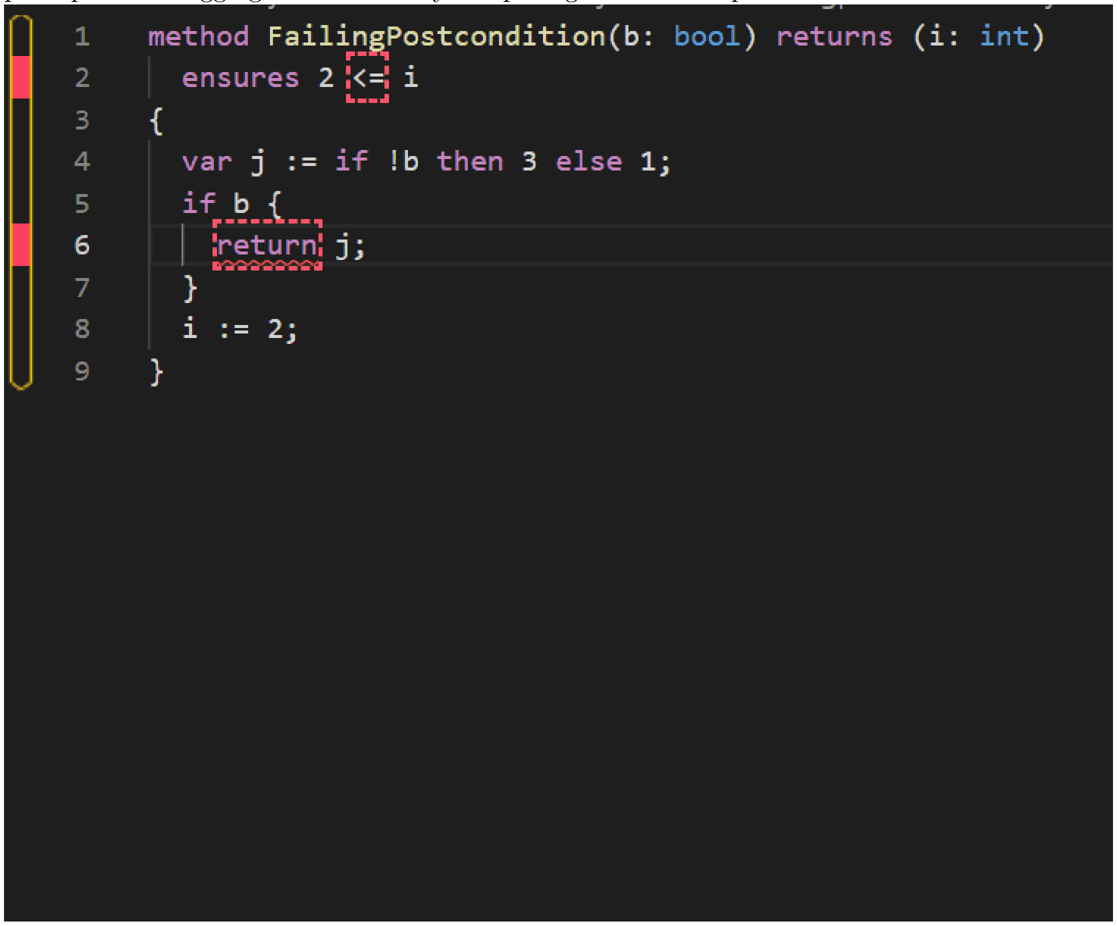
13.7. Verification

In this section, we suggest a methodology to figure out why a single assertion might not hold, we propose techniques to deal with assertions that slow a proof down, we explain how to verify assertions in parallel or in a focused way, and we also give some more examples of useful options and attributes to control verification.

13.7.1. Verification debugging when verification fails

Let's assume one assertion is failing ("assertion might not hold" or "postcondition might not hold"). What should you do next?

The following section is textual description of the animation below, which illustrates the principle of debugging an assertion by computing the weakest precondition:



```
method FailingPostcondition(b: bool) returns (i: int)
  ensures 2 <= i
{
  var j := if !b then 3 else 1;
  if b {
    return j;
  }
  i := 2;
}
```

13.7.1.1. Failing postconditions Let's look at an example of a failing postcondition.

```
method FailingPostcondition(b: bool) returns (i: int)
  ensures 2 <= i
{
  var j := if !b then 3 else 1;
  if b {
    return j;
  }//^^^^^^^ a postcondition might not hold on this return path.
  i := 2;
}
```

One first thing you can do is replace the statement `return j;` by two statements `i := j; return;` to better understand what is wrong:

```
method FailingPostcondition(b: bool) returns (i: int)
  ensures 2 <= i
{
  var j := if !b then 3 else 1;
  if b {
    i := j;
    return;
  }//^^^^^^^ a postcondition might not hold on this return path.
  i := 2;
}
```

Now, you can assert the postcondition just before the return:

```
method FailingPostcondition(b: bool) returns (i: int)
  ensures 2 <= i
{
  var j := if !b then 3 else 1;
  if b {
    i := j;
    assert 2 <= i; // This assertion might not hold
    return;
  }
  i := 2;
}
```

That's it! Now the postcondition is not failing anymore, but the `assert` contains the error! you can now move to the next section to find out how to debug this `assert`.

13.7.1.2. Failing asserts In the previous section, we arrived at the point where we have a failing assertion:

```
method FailingPostcondition(b: bool) returns (i: int)
  ensures 2 <= i
{
  var j := if !b then 3 else 1;
  if b {
    i := j;
    assert 2 <= i; // This assertion might not hold
    return;
  }
  i := 2;
}
```

To debug why this assert might not hold, we need to *move this assert up*, which is similar to *computing the weakest precondition*. For example, if we have `x := Y; assert F;` and the `assert F;` might not hold, the weakest precondition for it to hold before `x := Y;` can be written as the assertion `assert F[x:= Y];`, where we replace every occurence of `x` in `F` into `Y`. Let's do it in our example:

```
method FailingPostcondition(b: bool) returns (i: int)
  ensures 2 <= i
{
  var j := if !b then 3 else 1;
  if b {
    assert 2 <= j; // This assertion might not hold
    i := j;
    assert 2 <= i;
    return;
  }
  i := 2;
}
```

Yay! The assertion `assert 2 <= i;` is not proven wrong, which means that if we manage to prove `assert 2 <= j;`, it will work. Now, this assert should hold only if we are in this branch, so to *move the assert up*, we need to guard it. Just before the `if`, we can add the weakest precondition `assert b ==> (2 <= j)`:

```
method FailingPostcondition(b: bool) returns (i: int)
  ensures 2 <= i
{
  var j := if !b then 3 else 1;
  assert b  ==>  2 <= j;  // This assertion might not hold
  if b {
    assert 2 <= j;
    i := j;
    assert 2 <= i;
    return;
  }
  i := 2;
}
```

Again, now the error is only on the topmost assert, which means that we are making progress. Now, either the error is obvious, or we can one more time replace `j` by its value and create the assert `assert b ==> ((if !b then 3 else 1) >= 2);`

```
method FailingPostcondition(b: bool) returns (i: int)
  ensures 2 <= i
{
  assert b  ==>  2 <= (if !b then 3 else 1);  // This assertion might not hold
  var j := if !b then 3 else 1;
  assert b  ==>  2 <= j;
  if b {
    assert 2 <= j;
    i := j;
    assert 2 <= i;
    return;
  }
  i := 2;
}
```

At this point, this is pure logic. We can simplify the assumption:

```
b ==>  2 <= (if !b then 3 else 1)
!b ||  (if !b then 2 <= 3 else 2 <= 1)
!b ||  (if !b then true else false)
!b || !b;
!b;
```

Now we can understand what went wrong: When b is true, all of these formulas above are false, this is why the `dafny` verifier was not able to prove them. In the next section, we will explain how to “move asserts up” in certain useful patterns.

13.7.1.3. Failing asserts cases This list is not exhaustive but can definitely be useful to provide the next step to figure out why Dafny could not prove an assertion.

Failing assert	Suggested rewriting
`x := Y;assert P;`	`assert P[x := Y];x := Y;assert P;`
`if B { assert P; ...}`	`assert B ==> P;if B { assert P; ...}`
`if B { ...} else { assert P; ...}`	`assert !B ==> P;if B { ...} else { assert P; ...}`
`if X { ...} else { ...}assert A;`	`if X { ... assert A;} else { ... assert A;}assert A;`
`assert forall x :: Q(x);`	`forall x ensures Q(x){ assert Q(x);};assert forall x :: Q(x);`
`assert forall x :: P(x) ==> Q(x);`	`forall x | P(x) ensures Q(x){ assert Q(x);};assert forall x :: P(x) ==> Q(x);`

Failing assert	Suggested rewriting
`assert exists x | P(x) :: Q(x);assert exists x | P(x) :: Q'(x);`	`if x :| P(x) { assert Q(x); assert Q'(x);} else { assert false;}`
`assert exists x :: P(x);`	`assert P(x0);assert exists x :: P(x);`for a given expression `x0`.
`ensures exists i :: P(i);`	`returns (j: int)ensures P(j) ensures exists i :: P(i)`in a lemma, so that the `j` can be computed explicitly.
`assert A == B;callLemma(x);assert B == C;`	`calc == { A; B; { callLemma(x); } C;};assert A == B;`where the `calc` statement can be used to make intermediate computation steps explicit. Works with `<`, `>`, `<=`, `>=`, `==>`, `<==` and `<==>` for example.
`assert A ==> B;`	`if A { assert B;};assert A ==> B;`
`assert A && B;`	`assert A;assert B;assert A && B;`
`assert P(x);`where `P` is an `{:opaque}` predicate	`reveal P();assert P(x);`
`assert P(x);`where `P` is an `{:opaque}` predicate	`assert P(x) by { reveal P();}`
`assert P(x);`where `P` is not an `{:opaque}` predicate with a lot of `&&` in its body and is assumed	Make `P` `{:opaque}` so that if it's assumed, it can be proven more easily. You can always reveal it when needed.
`ensures P ==> Q` on a lemma	`requires P ensures Q` to avoid accidentally calling the lemma on inputs that do not satisfy `P`
`seq(size, i => P)`	`seq(size, i requires 0 <= i < size => P);`
`assert forall x :: G(i) ==> R(i);`	`assert G(i0);assert R(i0);assert forall i :: G(i) ==> R(i);` with a guess of the `i0` that makes the second assert to fail.
`assert forall i | 0 < i <= m :: P(i);`	`assert forall i | 0 < i < m :: P(i);assert forall i | i == m :: P(i);assert forall i | 0 < i <= m :: P(i);`
`assert forall i | i == m :: P(m);`	`assert P(m);assert forall i | i == m :: P(i);`
`method m(i) returns (j: T) requires A(i) ensures B(i, j){ ...}method n() { ... var x := m(a); assert P(x);`	`method m(i) returns (j: T) requires A(i) ensures B(i, j){ ...}method n() { ... assert A(k); assert forall x :: B(k, x) ==> P(x); var x := m(k); assert P(x);`
`method m_mod(i) returns (j: T) requires A(i) modifies this, i ensures B(i, j){ ...}method n_mod() { ... var x := m_mod(a); assert P(x);`	`method m_mod(i) returns (j: T) requires A(i) modifies this, i ensures B(i, j){ ...}method n_mod() { ... assert A(k); modify this, i; // Temporarily var x: T;     // Temporarily assume B(k, x); //  var x := m_mod(k); assert P(x);`

`modify x, y;assert P(x, y, z);`	`assert x != z && y != z;modify x, y;assert P(x, y, z);`

13.7.1.4. Counterexamples

When verification fails, we can rerun Dafny with `--extract-counterexample` flag to get a counterexample that can potentially explain the proof failure. Note that Danfy cannot guarantee that the counterexample it reports provably violates the assertion it was generated for (see [16]) The counterexample takes the form of assumptions that can be inserted into the code to describe the potential conditions under which the given assertion is violated. This output should be inspected manually and treated as a hint.

13.7.2. Verification debugging when verification is slow

In this section, we describe techniques to apply in the case when verification is slower than expected, does not terminate, or times out.

Additional detail is available in the verification optimization guide.

13.7.2.1. `assume false;`

Assuming `false` is an empirical way to short-circuit the verifier and usually stop verification at a given point,[17] and since the final compilation steps do not accept this command, it is safe to use it during development. Another similar command, `assert false;`, would also short-circuit the verifier, but it would still make the verifier try to prove `false`, which can also lead to timeouts.

Thus, let us say a program of this shape takes forever to verify.

```
method NotTerminating(b: bool) {
   assert X;
   if b {
     assert Y;
   } else {
     assert Z;
     assert P;
   }
}
```

What we can first do is add an `assume false` at the beginning of the method:

[16] The formula sent to the underlying SMT solver is the negation of the formula that the verifier wants to prove - also called a VC or verification condition. Hence, if the SMT solver returns "unsat", it means that the SMT formula is always false, meaning the verifier's formula is always true. On the other side, if the SMT solver returns "sat", it means that the SMT formula can be made true with a special variable assignment, which means that the verifier's formula is false under that same variable assignment, meaning it's a counter-example for the verifier. In practice and because of quantifiers, the SMT solver will usually return "unknown" instead of "sat", but will still provide a variable assignment that it couldn't prove that it does not make the formula true. `dafny` reports it as a "counter-example" but it might not be a real counter-example, only provide hints about what `dafny` knows.

[17] `assume false` tells the `dafny` verifier "Assume everything is true from this point of the program". The reason is that, 'false' proves anything. For example, `false ==> A` is always true because it is equivalent to `!false || A`, which reduces to `true || A`, which reduces to `true`.

```
method NotTerminating() {
   assume false; // Will never compile, but everything verifies instantly
   assert X;
   if b {
     assert Y;
   } else {
     assert Z;
     assert P;
   }
   assert W;
}
```

This verifies instantly. This gives us a strategy to bisect, or do binary search to find the assertion that slows everything down. Now, we move the `assume false;` below the next assertion:

```
method NotTerminating() {
   assert X;
   assume false;
   if b {
     assert Y;
   } else {
     assert Z;
     assert P;
   }
   assert W;
}
```

If verification is slow again, we can use techniques seen before to decompose the assertion and find which component is slow to prove.

If verification is fast, that's the sign that `X` is probably not the problem,. We now move the `assume false;` after the if/then block:

```
method NotTerminating() {
   assert X;
   if b {
     assert Y;
   } else {
     assert Z;
     assert P;
   }
   assume false;
   assert W;
}
```

Now, if verification is fast, we know that `assert W;` is the problem. If it is slow, we know that one of the two branches of the `if` is the problem. The next step is to put an `assume false;` at the end of the `then` branch, and an `assume false` at the beginning of the else branch:

```
method NotTerminating() {
   assert X;
   if b {
     assert Y;
     assume false;
   } else {
     assume false;
     assert Z;
     assert P;
   }
   assert W;
}
```

Now, if verification is slow, it means that `assert Y;` is the problem. If verification is fast, it means that the problem lies in the `else` branch. One trick to ensure we measure the verification time of the `else` branch and not the then branch is to move the first `assume false;` to the top of the then branch, along with a comment indicating that we are short-circuiting it for now. Then, we can move the second `assume false;` down and identify which of the two assertions makes the verifier slow.

```
method NotTerminating() {
   assert X;
   if b {
     assume false; // Short-circuit because this branch is verified anyway
     assert Y;
   } else {
     assert Z;
     assume false;
     assert P;
   }
   assert W;
}
```

If verification is fast, which of the two assertions `assert Z;` or `assert P;` causes the slowdown?[18]

We now hope you know enough of `assume false;` to locate assertions that make verification slow. Next, we will describe some other strategies at the assertion level to figure out what happens and perhaps fix it.

13.7.2.2. `assert ... by {}` If an assertion `assert X;` is slow, it is possible that calling a lemma or invoking other assertions can help to prove it: The postcondition of this lemma, or the added assertions, could help the `dafny` verifier figure out faster how to prove the result.

```
  assert SOMETHING_HELPING_TO_PROVE_LEMMA_PRECONDITION;
  LEMMA();
  assert X;
```

[18] `assert P;`.

```
...
lemma ()
  requires LEMMA_PRECONDITION
  ensures X { ... }
```

However, this approach has the problem that it exposes the asserted expressions and lemma postconditions not only for the assertion we want to prove faster, but also for every assertion that appears afterwards. This can result in slowdowns[19]. A good practice consists of wrapping the intermediate verification steps in an `assert ... by {}`, like this:

```
  assert X by {
    assert SOMETHING_HELPING_TO_PROVE_LEMMA_PRECONDITION;
    LEMMA();
  }
```

Now, only `X` is available for the `dafny` verifier to prove the rest of the method.

13.7.2.3. Labeling and revealing assertions Another way to prevent assertions or preconditions from cluttering the verifier[20] is to label and reveal them. Labeling an assertion has the effect of "hiding" its result, until there is a "reveal" calling that label.

The example of the previous section could be written like this.

```
  assert p: SOMETHING_HELPING_TO_PROVE_LEMMA_PRECONDITION;
  // p is not available here.
  assert X by {
    reveal p;
    LEMMA();
  }
```

Similarly, if a precondition is only needed to prove a specific result in a method, one can label and reveal the precondition, like this:

```
method Slow(i: int, j: int)
  requires greater: i > j {

  assert i >= j by {
    reveal greater;
  }
}
```

Labelled assert statements are available both in expressions and statements. Assertion labels are not accessible outside of the block which the assert statement is in. If you need to access

[19]By default, the expression of an assertion or a precondition is added to the knowledge base of the `dafny` verifier for further assertions or postconditions. However, this is not always desirable, because if the verifier has too much knowledge, it might get lost trying to prove something in the wrong direction.

[20]By default, the expression of an assertion or a precondition is added to the knowledge base of the `dafny` verifier for further assertions or postconditions. However, this is not always desirable, because if the verifier has too much knowledge, it might get lost trying to prove something in the wrong direction.

an assertion label outside of the enclosing expression or statement, you need to lift the labelled statement at the right place manually, e.g. rewrite

```
ghost predicate P(i: int)

method TestMethod(x: bool)
  requires r: x <==> P(1)
{
  if x {
    assert a: P(1) by { reveal r; }
  }
  assert x ==> P(1) by { reveal a; } // Error, a is not accessible
}
```

to

```
ghost predicate P(i: int)

method TestMethod(x: bool)
  requires r: x <==> P(1)
{
  assert a: x ==> P(1) by {
    if x {
      assert P(1) by { reveal r; } // Proved without revealing the precondition
    }
  }
  assert x ==> P(1) by { reveal a; } // Now a is accessible
}
```

To lift assertions, please refer to the techniques described in Verification Debugging.

13.7.2.4. Non-opaque `function method` Functions are normally used for specifications, but their functional syntax is sometimes also desirable to write application code. However, doing so naively results in the body of a `function method Fun()` be available for every caller, which can cause the verifier to time out or get extremely slow[21]. A solution for that is to add the attribute `{:opaque}` right between `function method` and `Fun()`, and use `reveal Fun();` in the calling functions or methods when needed.

13.7.2.5. Conversion to and from bitvectors Bitvectors and natural integers are very similar, but they are not treated the same by the `dafny` verifier. As such, conversion from `bv8` to an `int` and vice-versa is not straightforward, and can result in slowdowns.

There are two solutions to this for now. First, one can define a subset type instead of using the built-in type `bv8`:

[21]By default, the expression of an assertion or a precondition is added to the knowledge base of the `dafny` verifier for further assertions or postconditions. However, this is not always desirable, because if the verifier has too much knowledge, it might get lost trying to prove something in the wrong direction.

```
type byte = x | 0 <= x < 256
```

One of the problems of this approach is that additions, substractions and multiplications do not enforce the result to be in the same bounds, so it would have to be checked, and possibly truncated with modulos. For example:

```
type byte = x | 0 <= x < 256
method m() {
  var a: byte := 250;
  var b: byte := 200;
  var c := b - a;               // inferred to be an 'int', its value will be 50.
  var d := a + b;               // inferred to be an 'int', its value will be 450.
  var e := (a + b) % 256;       // still inferred to be an 'int'...
  var f: byte := (a + b) % 256; // OK
}
```

A better solution consists of creating a newtype that will have the ability to check bounds of arithmetic expressions, and can actually be compiled to bitvectors as well.

```
newtype {:nativeType "short"} byte = x | 0 <= x < 256
method m() {
  var a: byte := 250;
  var b: byte := 200;
  var c := b - a; // OK, inferred to be a byte
  var d := a + b; // Error: cannot prove that the result of a + b is of type `byte`.
  var f := ((a as int + b as int) % 256) as byte; // OK
}
```

One might consider refactoring this code into separate functions if used over and over.

13.7.2.6. Nested loops In the case of nested loops, the verifier might timeout sometimes because of inadequate or too much available information[22]. One way to mitigate this problem, when it happens, is to isolate the inner loop by refactoring it into a separate method, with suitable pre and postconditions that will usually assume and prove the invariant again. For example,

```
while X
   invariant Y
 {
   while X'
     invariant Y'
   {

   }
```

[22]By default, the expression of an assertion or a precondition is added to the knowledge base of the dafny verifier for further assertions or postconditions. However, this is not always desirable, because if the verifier has too much knowledge, it might get lost trying to prove something in the wrong direction.

```
 }
```

could be refactored as this:

```
`while X
   invariant Y
 {
   innerLoop();
 }
...
method innerLoop()
  require Y'
  ensures Y'
```

In the next section, when everything can be proven in a timely manner, we explain another strategy to decrease proof time by parallelizing it if needed, and making the verifier focus on certain parts.

13.7.3. Assertion batches, well-formedness, correctness

To understand how to control verification, it is first useful to understand how `dafny` verifies functions and methods.

For every method (or function, constructor, etc.), `dafny` extracts *assertions*. Assertions can roughly be sorted into two kinds: Well-formedness and correctness.

- *Well-formedness* assertions: All the implicit requirements of native operation calls (such as indexing and asserting that divisions are nonzero), `requires` clauses of function calls, explicit assertion expressions and `decreases` clauses at function call sites generate well-formedness assertions.
  An expression is said to be *well-formed* in a context if all well-formedness assertions can be proven in that context.
- *Correctness* assertions: All remaining assertions and clauses

For example, given the following statements:

```
if b {
  assert a*a != 0;
}
c := (assert b ==> a != 0; if b then 3/a else f(a));
assert c != 5/a;
```

Dafny performs the following checks:

```
var c: int;
if b {
  assert a*a != 0;   // Correctness
}
assert b ==> a != 0; // Well-formedness
```

```
if b {
  assert a != 0;      // Well-formedness
} else {
  assert f.requires(a); // Well-formedness
}
c := if b then 3/a else f(a);
assert a != 0;        // Well-formedness
assert c != 5/a;      // Correctness
```

Well-formedness is proved at the same time as correctness, except for well-formedness of requires and ensures clauses which is proved separatedly from the well-formedness and correctness of the rest of the method/function. For the rest of this section, we don't differentiate between well-formedness assertions and correctness assertions.

We can also classify the assertions extracted by Dafny in a few categories:

Integer assertions:

- Every division yields an *assertion* that the divisor is never zero.
- Every bounded number operation yields an *assertion* that the result will be within the same bounds (no overflow, no underflows).
- Every conversion yields an *assertion* that conversion is compatible.
- Every bitvector shift yields an *assertion* that the shift amount is never negative, and that the shift amount is within the width of the value.

Object assertions:

- Every object property access yields an *assertion* that the object is not null.
- Every assignment `o.f := E;` yields an *assertion* that `o` is among the set of objects of the `modifies` clause of the enclosing loop or method.
- Every read `o.f` yields an *assertion* that `o` is among the set of objects of the `reads` clause of the enclosing function or predicate.
- Every array access `a[x]` yields the assertion that `0 <= x < a.Length`.
- Every sequence access `a[x]` yields an *assertion*, that `0 <= x < |a|`, because sequences are never null.
- Every datatype update expression and datatype destruction yields an *assertion* that the object has the given property.
- Every method overriding a `trait` yields an *assertion* that any postcondition it provides implies the postcondition of its parent trait, and an *assertion* that any precondition it provides is implied by the precondition of its parent trait.

Other assertions:

- Every value whose type is assigned to a subset type yields an *assertion* that it satisfies the subset type constraint.
- Every non-empty subset type yields an *assertion* that its witness satisfies the constraint.
- Every Assign-such-that operator `x :| P(x)` yields an *assertion* that `exists x :: P(x)`. In case `x :| P(x); Body(x)` appears in an expression and `x` is non-ghost, it also yields `forall x, y | P(x) && P(y) :: Body(x) == Body(y)`.

- Every recursive function yields an *assertion* that it terminates.
- Every match expression or alternative if statement yields an *assertion* that all cases are covered.
- Every call to a function or method with a `requires` clause yields *one assertion per requires clause*[23] (special cases such as sequence indexing come with a special `requires` clause that the index is within bounds).

Specification assertions:

- Any explicit `assert` statement is *an assertion*[24].
- A consecutive pair of lines in a `calc` statement forms *an assertion* that the expressions are related according to the common operator.
- Every `ensures` clause yields an *assertion* at the end of the method and on every return, and on `forall` statements.
- Every `invariant` clause yields an *assertion* that it holds before the loop and an *assertion* that it holds at the end of the loop.
- Every `decreases` clause yields an *assertion* at either a call site or at the end of a while loop.
- Every `yield ensures` clause on an iterator yields *assertions* that the clause holds at every yielding point.
- Every `yield requires` clause on an iterator yields *assertions* that the clause holds at every point when the iterator is called.

It is useful to mentally visualize all these assertions as a list that roughly follows the order in the code, except for `ensures` or `decreases` that generate assertions that seem earlier in the code but, for verification purposes, would be placed later. In this list, each assertion depends on other assertions, statements and expressions that appear earlier in the control flow[25].

The fundamental unit of verification in `dafny` is an *assertion batch*, which consists of one or more assertions from this "list", along with all the remaining assertions turned into assumptions. To reduce overhead, by default `dafny` collects all the assertions in the body of a given method into a single assertion batch that it sends to the verifier, which tries to prove it correct.

- If the verifier says it is correct,[26] it means that all the assertions hold.

[23]`dafny` actually breaks things down further. For example, a precondition `requires A && B` or an assert statement `assert A && B;` turns into two assertions, more or less like `requires A requires B` and `assert A; assert B;`.

[24]`dafny` actually breaks things down further. For example, a precondition `requires A && B` or an assert statement `assert A && B;` turns into two assertions, more or less like `requires A requires B` and `assert A; assert B;`.

[25]All the complexities of the execution paths (if-then-else, loops, goto, break....) are, down the road and for verification purposes, cleverly encoded with variables recording the paths and guarding assumptions made on each path. In practice, a second clever encoding of variables enables grouping many assertions together, and recovers which assertion is failing based on the value of variables that the SMT solver returns.

[26]The formula sent to the underlying SMT solver is the negation of the formula that the verifier wants to prove - also called a VC or verification condition. Hence, if the SMT solver returns "unsat", it means that the SMT formula is always false, meaning the verifier's formula is always true. On the other side, if the SMT solver returns "sat", it means that the SMT formula can be made true with a special variable assignment, which means that the verifier's formula is false under that same variable assignment, meaning it's a counter-example for the verifier. In practice and because of quantifiers, the SMT solver will usually return "unknown" instead of "sat", but will still provide a variable assignment that it couldn't prove that it does not make the formula true. `dafny` reports it as

- If the verifier returns a counterexample, this counterexample is used to determine both the failing assertion and the failing path. In order to retrieve additional failing assertions, `dafny` will again query the verifier after turning previously failed assertions into assumptions.[27] [28]
- If the verifier returns `unknown` or times out, or even preemptively for difficult assertions or to reduce the chance that the verifier will 'be confused' by the many assertions in a large batch, `dafny` may partition the assertions into smaller batches[29]. An extreme case is the use of the `/vcsSplitOnEveryAssert` command-line option or the `{:isolate_assertions}` attribute, which causes `dafny` to make one batch for each assertion.

13.7.3.1. Controlling assertion batches

Here is how you can control how `dafny` partitions assertions into batches.

- `{:focus}` on an assert generates a separate assertion batch for the assertions of the enclosing block.
- `{:split_here}` on an assert generates a separate assertion batch for assertions after this point.
- `{:isolate_assertions}` on a function or a method generates one assertion batch per assertion

We discourage the use of the following *heuristics attributes* to partition assertions into batches. The effect of these attributes may vary, because they are low-level attributes and tune low-level heuristics, and will result in splits that could be manually controlled anyway. * `{:vcs_max_cost N}` on a function or method enables splitting the assertion batch until the "cost" of each batch is below N. Usually, you would set `{:vcs_max_cost 0}` and `{:vcs_max_splits N}` to ensure it generates N assertion batches. * `{:vcs_max_keep_going_splits N}` where $N > 1$ on a method dynamically splits the initial assertion batch up to N components if the verifier is stuck the first time.

13.7.4. Command-line options and other attributes to control verification

There are many great options that control various aspects of verifying dafny programs. Here we mention only a few:

- Control of output: `/dprint`, `/rprint`, `/stats`, `/compileVerbose`
- Whether to print warnings: `/proverWarnings`
- Control of time: `/timeLimit`
- Control of resources: `/rLimit` and `{:rlimit}`

a "counter-example" but it might not be a real counter-example, only provide hints about what `dafny` knows.

[27]This post gives an overview of how assertions are turned into assumptions for verification purposes.

[28]Caveat about assertion and assumption: One big difference between an "assertion transformed in an assumption" and the original "assertion" is that the original "assertion" can unroll functions twice, whereas the "assumed assertion" can unroll them only once. Hence, `dafny` can still continue to analyze assertions after a failing assertion without automatically proving "false" (which would make all further assertions vacuous).

[29]To create a smaller batch, `dafny` duplicates the assertion batch, and arbitrarily transforms the clones of an assertion into assumptions except in exactly one batch, so that each assertion is verified only in one batch. This results in "easier" formulas for the verifier because it has less to prove, but it takes more overhead because every verification instance have a common set of axioms and there is no knowledge sharing between instances because they run independently.

- Control of the prover used: `/prover`
- Control of how many times to *unroll* functions: `{:fuel}`

You can search for them in this file as some of them are still documented in raw text format.

13.7.5. Analyzing proof dependencies

When Dafny successfully verifies a particular definition, it can ask the solver for information about what parts of the program were actually used in completing the proof. The program components that can potentially form part of a proof include:

- `assert` statements (and the implicit assumption that they hold in subsequent code),
- implicit assertions (such as array or sequence bounds checks),
- `assume` statements,
- `ensures` clauses,
- `requires` clauses,
- function definitions,
- method calls, and
- assignment statements.

Understanding what portions of the program the proof depended on can help identify mistakes, and to better understand the structure of your proof (which can help when optimizing it, among other things). In particular, there are two key dependency structures that tend to indicate mistakes, both focused on what parts of the program were *not* included in the proof.

- Redundant assumptions. In some cases, a proof can be completed without the need of certain `assume` statements or `requires` clauses. This situation might represent a mistake, and when the mistake is corrected those program elements may become required. However, they may also simply be redundant, and the program will become simpler if they're removed. Dafny will report assumptions of this form when verifying with the flag `--warn-redundant-assumptions`. Note that `assert` statements may be warned about, as well, indicating that the fact proved by the assertion wasn't needed to prove anything else in the program.

- Contradictory assumptions. If the combination of all assumptions in scope at a particular program point is contradictory, anything can be proved at that point. This indicates the serious situation that, unless done on purpose in a proof by contradiction, your proof may be entirely vacuous. It therefore may not say what you intended, giving you a false sense of confidence. The `--warn-contradictory-assumptions` flag instructs Dafny to warn about any assertion that was proved through the use of contradictions between assumptions. If a particular `assert` statement is part of an intentional proof by contradiction, annotating it with the `{:contradiction}` attribute will silence this warning.

These options can be specified in `dfyconfig.toml`, and this is typically the most convenient way to use them with the IDE.

More detailed information is available using either the `--log-format text` or `--verification-coverage-report` option to `dafny verify`. The former will include a list of proof dependencies (including source location and description) alongside every assertion batch in the generated log whenever one of the two warning options above is also included. The latter will produce a highlighted HTML version of your source code, in the same format used by `dafny test --coverage-report` and `dafny generate-tests --verification-coverage-report`, indicating which parts of the program were used, not used, or partly used in the verification of the entire program.

13.7.6. Debugging brittle verification

When evolving a Dafny codebase, it can sometimes occur that a proof obligation succeeds at first only for the prover to time out or report a potential error after minor, valid changes. We refer to such a proof obligation as *brittle*. This is ultimately due to decidability limitations in the form of automated reasoning that Dafny uses. The Z3 SMT solver that Dafny depends on attempts to efficiently search for proofs, but does so using both incomplete heuristics and a degree of randomness, with the result that it can sometimes fail to find a proof even when one exists (or continue searching forever).

Dafny provides some features to mitigate this issue, primarily focused on early detection. The philosophy is that, if Dafny programmers are alerted to proofs that show early signs of brittleness, before they are obviously so, they can refactor the proofs to make them less brittle before further development becomes difficult.

The mechanism for early detection focuses on measuring the resources used to discharge a proof obligation (either using duration or a more deterministic “resource count” metric available from Z3). Dafny can re-run a given proof attempt multiple times after automatically making minor changes to the structure of the input or to the random choices made by the solver. If the resources used during these attempts (or the ability to find a proof at all) vary widely, we use this as a proxy metric indicating that the proof may be brittle.

13.7.6.1. Measuring proof brittleness To measure the brittleness of your proofs, start by using the `dafny measure-complexity` command with the `--iterations N` flag to instruct Dafny to attempt each proof goal `N` times, using a different random seed each time. The random seed used for each attempt is derived from the global random seed `S` specified with `-randomSeed:S`, which defaults to `0`. The random seed affects the structure of the SMT queries sent to the solver, changing the ordering of SMT commands, the variable names used, and the random seed the solver itself uses when making decisions that can be arbitary.

For most use cases, it also makes sense to specify the `--log-format csv` flag, to log verification cost statistics to a CSV file. By default, the resulting CSV files will be created in the `TestResults` folder of the current directory.

Once Dafny has completed, the `dafny-reportgenerator` tool is a convenient way to process the output. It allows you to specify several limits on statistics computed from the elapsed time or solver resource use of each proof goal, returning an error code when it detects violations of these limits. You can find documentation on the full set of options for `dafny-reportgenerator` in its `README.md` file.

In general, we recommend something like the following:

```
dafny-reportgenerator --max-resource-cv-pct 10 TestResults/*.csv
```

This bounds the coefficient of variation of the solver resource count at 10% (0.10). We recommend a limit of less than 20%, perhaps even as low as 5%. However, when beginning to analyze a new project, it may be necessary to set limits as high as a few hundred percent and incrementally ratchet down the limit over time.

When first analyzing proof brittleness, you may also find that certain proof goals succeed on some iterations and fail on others. If your aim is first to ensure that brittleness doesn't worsen and then to start reducing it, integrating `dafny-reportgenerator` into CI and using the `--allow-different-outcomes` flag may be appropriate. Then, once you've improved brittleness sufficiently, you can likely remove that flag (and likely have significantly lower limits on other metrics).

13.7.6.2. Improving proof brittleness

Reducing proof brittleness is typically closely related to improving performance overall. As such, techniques for debugging slow verification are typically useful for debugging brittle proofs, as well. See also the verification optimization guide.

13.8. Compilation

The `dafny` tool can compile a Dafny program to one of several target languages. Details and idiosyncrasies of each of these are described in the following subsections. In general note the following:

- The compiled code originating from `dafny` can be combined with other source and binary code, but only the `dafny`-originated code is verified.
- Output file or folder names can be set using `--output`.
- Code generated by `dafny` requires a Dafny-specific runtime library. By default the runtime is included in the generated code. However for `dafny translate` it is not included by default and must be explicitly requested using `--include-runtime`. All runtime libraries are part of the Binary (`./DafnyRuntime.*`) and Source (`./Source/DafnyRuntime/DafnyRuntime.*`) releases.
- Names in Dafny are written out as names in the target language. In some cases this can result in naming conflicts. Thus if a Dafny program is intended to be compiled to a target language X, you should avoid using Dafny identifiers that are not legal identifiers in X or that conflict with reserved words in X.

To be compilable to an executable program, a Dafny program must contain a `Main` entry point, as described here.

13.8.1.1 Built-in declarations

Dafny includes several built-in types such as tuples, arrays, arrows (functions), and the `nat` subset type. The supporting target language code for these declarations could be emitted on-demand, but these could then become multiple definitions of the same symbols when compiling multiple components separately. Instead, all such built-ins up to a pre-configured maximum size are included in most of the runtime libraries. This means that when compiling to certain target languages, the use of such built-ins above these maximum sizes, such as tuples with more than 20 elements, is not supported. See the Supported features by target language table for the details on these limits.

13.8.2. `extern` declarations

A Dafny declaration can be marked with the `{:extern}` attribute to indicate that it refers to an external definition that is already present in the language that the Dafny code will be compiled to (or will be present by the time the final target-language program is compiled or run).

Because the `{:extern}` attribute controls interaction with code written in one of many languages, it has some language-specific behavior, documented in the following sections. However, some aspects are target-language independent and documented here.

The attribute can also take several forms, each defining a different relationship between a Dafny name and a target language name. In the form `{:extern}`, the name of the external definition is assumed to be the name of the Dafny declaration after some target-specific name mangling. However, because naming conventions (and the set of allowed identifiers) vary between languages, Dafny allows additional forms for the `{:extern}` attribute.

The form `{:extern <s1>}` instructs `dafny` to compile references to most declarations using the name `s1` instead of the Dafny name. For abstract types, however, `s1` is sometimes used as a hint as to how to declare that type when compiling. This hint is interpreted differently by each compiler.

Finally, the form `{:extern <s1>, <s2>}` instructs `dafny` to use `s2` as the direct name of the declaration. `dafny` will typically use a combination of `s1` and `s2`, such as `s1.s2`, to reference the declaration. It may also be the case that one of the arguments is simply ignored, depending on the target language.

The recommended style is to prefer `{:extern}` when possible, and use similar names across languages. This is usually feasible because existing external code is expected to have the same interface as the code that `dafny` would generate for a declaration of that form. Because many Dafny types compile down to custom types defined in the Dafny runtime library, it's typically necessary to write wrappers by hand that encapsulate existing external code using a compatible interface, and those wrappers can have names chosen for compatibility. For example, retrieving the list of command line arguments when compiling to C# requires a wrapper such as the following:

```
using icharseq = Dafny.ISequence<char>;
using charseq = Dafny.Sequence<char>;

namespace Externs_Compile {
  public partial class __default {
    public static Dafny.ISequence<icharseq> GetCommandLineArgs() {
      var dafnyArgs = Environment
                      .GetCommandLineArgs()
                      .Select(charseq.FromString);
      return Dafny.Sequence<icharseq>.FromArray(dafnyArgs.ToArray());
    }
}
```

This serves as an example of implementing an extern, but was only necessary to retrieve command line arguments historically, as `dafny` now supports capturing these arguments via a main method that accepts a `seq<string>` (see the section on the Main method).

Note that `dafny` does not check the arguments to `{:extern}`, so it is the user's responsibility to ensure that the provided names result in code that is well-formed in the target language.

Also note that the interface the external code needs to implement may be affected by compilation flags. In this case, if `--unicode-char:true` is provided, `dafny` will compile its `char` type to the `Dafny.Rune` C# type instead, so the references to the C# type `char` above would need to be changed accordingly. The reference to `charseq.FromString` would in turn need to be changed to `charseq.UnicodeFromString` to return the correct type.

Most declarations, including those for modules, classes, traits, member variables, constructors, methods, function methods, and abstract types, can be marked with `{:extern}`.

Marking a module with `{:extern}` indicates that the declarations contained within can be found within the given module, namespace, package, or similar construct within the target language. Some members of the Dafny module may contain definitions, in which case code for those definitions will be generated. Whether this results in valid target code may depend on some target language support for something resembling "partial" modules, where different subsets of the contents are defined in different places.

The story for classes is similar. Code for a class will be generated if any of its members are not `{:extern}`. Depending on the target language, making either all or none of the members `{:extern}` may be the only options that result in valid target code. Traits with `{:extern}` can refer to existing traits or interfaces in the target language, or can refer to the interfaces of existing classes.

Member variables marked with `{:extern}` refer to fields or properties in existing target-language code. Constructors, methods, and functions refer to the equivalent concepts in the target language. They can have contracts, which are then assumed to hold for the existing target-language code. They can also have bodies, but the bodies will not be compiled in the presence of the `{:extern}` attribute. Bodies can still be used for reasoning, however, so may be valuable in some cases, especially for function methods.

Types marked with `{:extern}` must be opaque. The name argument, if any, usually refers to the type name in the target language, but some compilers treat it differently.

Detailed description of the `dafny build` and `dafny run` commands and the `--input` option (needed when `dafny run` has more than one input file) is contained in the section on command-line structure.

13.8.3. Replaceable modules

To enable easily customising runtime behavior across an entire Dafny program, Dafny has placeholder modules. Here follows an example:

```
replaceable module Foo {
  method Bar() returns (i: int)
    ensures i >= 2
}

method Main() {
  var x := Foo.Bar();
  print x;
}
// At this point, the program can be verified but not run.

module ConcreteFoo replaces Foo {
  method Bar() returns (i: int) {
    return 3; // Main will print 3.
  }
}
// ConcreteFoo can be swapped out for different replacements of Foo, to customize runtime behavior
```

When replacing a replaceable module, the same rules apply as when refining an abstract module. However, unlike an abstract module, a placeholder module can be used as if it is a concrete module. When executing code, using for example `dafny run` or `dafny translate`, any program that contains a placeholder module must also contain a replacement of this placeholder. When using `dafny verify`, placeholder modules do not have to be replaced.

Replaceable modules are particularly useful for defining behavior that depends on which target language Dafny is translated to.

13.8.4. C#

For a simple Dafny-only program, the translation step converts a `A.dfy` file into `A.cs`; the build step then produces a `A.dll`, which can be used as a library or as an executable (run using `dotnet A.dll`).

It is also possible to run the dafny files as part of a `csproj` project, with these steps: - create a dotnet project file with the command `dotnet new console` - delete the `Program.cs` file - build the dafny program: `dafny build A.dfy` - run the built program `dotnet A.dll`

The last two steps can be combined: `dafny run A.dfy`

Note that all input `.dfy` files and any needed runtime library code are combined into a single `.cs` file, which is then compiled by `dotnet` to a `.dll`.

Examples of how to integrate C# libraries and source code with Dafny source code are contained in this separate document.

13.8.5. Java

The Dafny-to-Java compiler translation phase writes out the translated files of a file *A*`.dfy` to a directory *A*`-java`. The build phase writes out a library or executable jar file. The `--output` option (`-out` in the legacy CLI) can be used to choose a different jar file path and name and correspondingly different directory for .java and .class files.

The compiler produces a single wrapper method that then calls classes in relevant other `.java` files. Because Java files must be named consistent with the class they contain, but Dafny files do not, there may be no relation between the Java file names and the Dafny file names. However, the wrapper class that contains the Java `main` method is named for the first `.dfy` file on the command-line.

The step of compiling Java files (using `javac`) requires the Dafny runtime library. That library is automatically included if dafny is doing the compilation, but not if dafny is only doing translation.

Examples of how to integrate Java source code and libraries with Dafny source are contained in this separate document.

13.8.6. Javascript

The Dafny-to-Javascript compiler translates all the given `.dfy` files into a single `.js` file, which can then be run using `node`. (Javascript has no compilation step). The build and run steps are simply - `dafny build --target:js A.dfy` - `node A.js`

Or, in one step, - `dafny run A.dfy`

Examples of how to integrate Javascript libraries and source code with Dafny source are contained in this separate document.

13.8.7. Go

The Dafny-to-Go compiler translates all the given `.dfy` files into a single `.go` file in `A-go/src/A.go`; the output folder can be specified with the `-out` option. For an input file `A.dfy` the default output folder is `A-go`. Then, Dafny compiles this program and creates an `A.exe` executable in the same folder as `A.dfy`. Some system runtime code is also placed in `A-go/src`. The build and run steps are - `dafny build --target:go A.dfy` - `./A`

The uncompiled code can be compiled and run by `go` itself using - ``(cd A-go; GO111MODULE=auto GOPATH=`pwd` go run src/A.go)``

The one-step process is - `dafny run --target:go A.dfy`

The `GO111MODULE` variable is used because Dafny translates to pre-module Go code. When the implementation changes to current Go, the above command-line will change, though the `./A` alternative will still be supported.

Examples of how to integrate Go source code and libraries with Dafny source are contained in this separate document.

13.8.8. Python

The Dafny-to-Python compiler is still under development. However, simple Dafny programs can be built and run as follows. The Dafny-to-Python compiler translates the `.dfy` files into a single `.py` file along with supporting runtime library code, all placed in the output location (`A-py` for an input file A.dfy, by default).

The build and run steps are - `dafny build --target:py A.dfy` - `python A-py/A.py`

In one step: - `dafny run --target:py A.dfy`

Examples of how to integrate Python libraries and source code with Dafny source are contained in this separate document.

13.8.9. C++

The C++ backend was written assuming that it would primarily support writing C/C++ style code in Dafny, which leads to some limitations in the current implementation.

- The C++ compiler does not support BigIntegers, so do not use `int`, or raw instances of `arr.Length`, or sequence length, etc. in executable code. You can however, use `arr.Length as uint64` if you can prove your array is an appropriate size. The compiler will report inappropriate integer use.
- The C++ compiler does not support advanced Dafny features like traits or coinductive types.
- There is very limited support for higher order functions even for array initialization. Use extern definitions like newArrayFill (see extern.dfy) or similar. See also the example in [`functions.dfy`] (https://github.com/dafny-lang/dafny/blob/master/Test/c++/functions.dfy).
- The current backend also assumes the use of C++17 in order to cleanly and performantly implement datatypes.

13.8.10. Supported features by target language

Some Dafny features are not supported by every target language. The table below shows which features are supported by each backend. An empty cell indicates that a feature is not supported, while an X indicates that it is.

Feature	C#	JavaScript	Go	Java	Python	C++	Dafny Library (.doo)
Unbounded integers	X	X	X	X	X		X
Real numbers	X	X	X	X	X		X
Ordinals	X	X	X	X	X		X
Function values	X	X	X	X	X		X
Iterators	X	X	X		X		X

Feature	C#	JavaScript	Go	Java	Python	C++	Dafny Library (.doo)
Collections with trait element types	X	X	X	X	X		X
External module names with only underscores	X	X		X	X	X	X
Co-inductive datatypes	X	X	X	X	X		X
Multisets	X	X	X	X	X		X
Runtime type descriptors	X	X	X	X	X		X
Multi-dimensional arrays	X	X	X	X	X		X
Map comprehensions	X	X	X	X	X		X
Traits	X	X	X	X	X		X
Let-such-that expressions	X	X	X	X	X		X
Non-native numeric newtypes	X	X	X	X	X		X
Method synthesis	X						X
External classes	X	X	X	X	X		X
Instantiating the `object` type	X	X	X	X	X		X

Feature	C#	JavaScript	Go	Java	Python	C++	Dafny Library (.doo)
`forall` statements that cannot be sequentialized[30]	X	X	X	X	X		X
Taking an array's length	X	X	X	X	X		X
`m.Items` when `m` is a map	X	X	X	X	X		X
The /runAllTests option	X	X	X	X	X		X
Integer range constraints in quantifiers (e.g. `a <= x <= b`)	X	X	X	X	X	X	X
Exact value constraints in quantifiers (`x == C`)	X	X	X	X	X		X

[30] 'Sequentializing' a `forall` statement refers to compiling it directly to a series of nested loops with the statement's body directly inside. The alternative, default compilation strategy is to calculate the quantified variable bindings separately as a collection of tuples, and then execute the statement's body for each tuple. Not all `forall` statements can be sequentialized.

Feature	C#	JavaScript	Go	Java	Python	C++	Dafny Library (.doo)
Sequence displays of characters[31]	X	X	X	X	X		X
Type test expressions (`x is T`)	X	X	X	X	X		X
Type test expressions on subset types							X
Quantifiers	X	X	X	X	X		X
Bitvector RotateLeft/RotateRight functions	X	X	X	X	X		X
`for` loops	X	X	X	X	X	X	X
`continue` statements	X	X	X	X	X	X	X
Assign-such-that statements with potentially infinite bounds[32]	X	X	X	X	X	X	X
Sequence update expressions	X	X	X	X	X	X	X

[31] This refers to an expression such as `['H', 'e', 'l', 'l', 'o']`, as opposed to a string literal such as `"Hello"`.

[32] This refers to assign-such-that statements with multiple variables, and where at least one variable has potentially infinite bounds. For example, the implementation of the statement `var x: nat, y: nat :| 0 < x && 0 < y && x*x == y*y*y + 1;` needs to avoid the naive approach of iterating all possible values of `x` and `y` in a nested loop.

Feature	C#	JavaScript	Go	Java	Python	C++	Dafny Library (.doo)
Sequence constructions with non-lambda initializers[33]	X	X	X	X	X	X	X
Externally-implemented constructors	X			X	X	X	X
Auto-initialization of tuple variables	X	X	X	X	X	X	X
Subtype constraints in quantifiers	X	X	X	X	X	X	X
Tuples with more than 20 arguments		X	X		X	X	X
Arrays with more than 16 dimensions		X	X		X	X	X
Arrow types with more than 16 arguments		X	X		X	X	X

[33]Sequence construction expressions often use a direct lambda expression, as in `seq(10, x => x * x)`, but they can also be used with arbitrary function values, as in `seq(10, squareFn)`.

Feature	C#	JavaScript	Go	Java	Python	C++	Dafny Library (.doo)
Unicode chars	X	X	X	X	X		X
Converting values to strings	X	X	X	X	X		X
Legacy CLI without commands	X	X	X	X	X	X	
Separate compilation	X		X	X	X	X	X
All built-in types in runtime library	X	X	X	X	X		X
Execution coverage report	X						

13.9. Dafny Command Line Options

There are many command-line options to the `dafny` tool. The most current documentation of the options is within the tool itself, using the `-?` or `--help` or `-h` options.

Remember that options are typically stated with either a leading `--`.

Legacy options begin with either '-' or '/'; however they are being migrated to the POSIX-compliant `--` form as needed.

13.9.1. Help and version information

These options emit general information about commands, options and attributes. When present, the dafny program will terminates after emitting the requested information but without processig any files.

- `--help`, `-h` - shows the various commands (which have help information under them as `dafny <command> -h`
- `--version` - show the version of the build

Legacy options:

- `-?` - print out the legacy list of command-line options and terminate. All of these options are also described in this and the following sections.
- `-attrHelp` - print out the current list of supported attribute declarations and terminate.
- `-env:<n>` - print the command-line arguments supplied to the program. The value of `<n>` can be one of the following.
  - `0` - never print command-line arguments.
  - `1` (default) - print them to Boogie (`.bpl`) files and prover logs.
  - `2` - operate like with option `1` but also print to standard output.
- `-wait` - wait for the user to press `Enter` before terminating after a successful execution.

13.9.2. Controlling input

These options control how Dafny processes its input.

- `-stdin` - read standard input and treat it as Dafny source code, instead of reading from a file.
- `--library:<files>` - treat the given files as *library* code, namely, skip these files (and any files recursively included) during verification; the value may be a comma-separated-list of files or folders; folders are expanded into a list of all .dfy files contained, recursively, in those folders
- `--prelude:<file>` (was `-dprelude`) - select an alternative Dafny prelude file. This file contains Boogie definitions (including many axioms) required by the translator from Dafny to Boogie. Using an alternative prelude is primarily useful if you're extending the Dafny language or changing how Dafny constructs are modeled. The default prelude is here.

13.9.3. Controlling plugins

Dafny has a plugin capability. A plugin has access to an AST of the dafny input files after all parsing and resolution are performed (but not verification) and also to the command-line options.

This facility is still *experimental* and very much in flux, particularly the form of the AST. The best guides to writing a new plugin are (a) the documentation in the section of this manual on plugins and (b) example plugins in the `src/Tools` folder of the `dafny-lang/compiler-bootstrap` repo.

The value of the option `--plugin` is a path to a dotnet dll that contains the compiled plugin.

13.9.4. Controlling output

These options instruct Dafny to print various information about your program during processing, including variations of the original source code (which can be helpful for debugging).

- `--use-basename-for-filename` - when enabled, just the filename without the directory path is used in error messages; this make error message shorter and not tied to the local environment (which is a help in testing)

- `--output`, `-o` - location of output files [translate, build]
- `--show-snippets` - include with an error message some of the source code text in the neighborhood of the error; the error location (file, line, column) is always given
- `--solver-log <file>` - [verification only] the file in which to place the SMT text sent to the solver
- `--log-format <configuration>` - [verification only] (was `-verificationLogger:<configuration string>`) log verification results to the given test result logger. The currently supported loggers are `trx`, `csv`, and `text`. These are the XML-based formats commonly used for test results for .NET languages, a custom CSV schema, and a textual format meant for human consumption, respectively. You can provide configuration using the same string format as when using the `--logger` option for dotnet test, such as:

  ```
      -verificationLogger:trx;LogFileName=<...>
  ```

  The exact mapping of verification concepts to these formats is experimental and subject to change!

  The `trx` and `csv` loggers automatically choose an output file name by default, and print the name of this file to the console. The `text` logger prints its output to the console by default, but can send output to a file given the `LogFileName` option.

  The `text` logger also includes a more detailed breakdown of what assertions appear in each assertion batch. When combined with the `-vcsSplitOnEveryAssert` option, it will provide approximate time and resource use costs for each assertion, allowing identification of especially expensive assertions.

Legacy options:

- `-stats` - print various statistics about the Dafny files supplied on the command line. The statistics include the number of total functions, recursive functions, total methods, ghost methods, classes, and modules. They also include the maximum call graph width and the maximum module height.
- `-dprint:<file>` - print the Dafny program after parsing (use `-` for `<file>` to print to the console).
- `-rprint:<file>` - print the Dafny program after type resolution (use `-` for `<file>` to print to the console).
- `-printMode:<Everything|DllEmbed|NoIncludes|NoGhost>` - select what to include in the output requested by `-dprint` or `-rprint`. The argument can be one of the following.
  - `Everything` (default) - include everything.
  - `DllEmbed`- print the source that will be included in a compiled DLL.
  - `NoIncludes` - disable printing of methods incorporated via the include mechanism that have the `{:verify false}` attribute, as well as datatypes and fields included from other files.

  - NoGhost - disables printing of functions, ghost methods, and proof statements in implementation methods. Also disable anything NoIncludes disables.

- -printIncludes:<None|Immediate|Transitive> - select what information from included files to incorporate into the output selected by -dprint or -rprint. The argument can be one of the following.
  - None (default) - don't print anything from included files.
  - Immediate - print files directly included by files specified on the command line. Exit after printing.
  - Transitive - print files transitively included by files specified on the command line. Exit after printing.

- -view:<view1, view2> - this option limits what is printed by /rprint for a module to the names that are part of the given export set; the option argument is a comma-separated list of fully-qualified export set names.

- -funcCallGraph - print out the function call graph. Each line has the format func,mod=callee*, where func is the name of a function, mod is the name of its containing module, and callee* is a space-separated list of the functions that func calls.

- --show-snippets (was -showSnippets:<n>) - show a source code snippet for each Dafny message. The legacy option was -showSnippets with values 0 and 1 for false and true.

- -printTooltips - dump additional positional information (displayed as mouse-over tooltips by LSP clients) to standard output as Info messages.

- -pmtrace - print debugging information from the pattern-match compiler.

- -titrace - print debugging information during the type inference process.

- -diagnosticsFormat:<text|json> - control how to report errors, warnings, and info messages. <fmt> may be one of the following:
  - text (default): Report diagnostics in human-readable format.
  - json: Report diagnostics in JSON format, one object per diagnostic, one diagnostic per line. Info-level messages are only included with -printTooltips. End positions are only included with -showSnippets:1. Diagnostics are the following format (but without newlines):

```
{
  "location": {
    "filename": "xyz.dfy",
    "range": { // Start and (optional) end of diagnostic
      "start": {
        "pos": 83, // 0-based character offset in input
        "line": 6, // 1-based line number
        "character": 0 // 0-based column number
      },
      "end": { "pos": 86, "line": 6, "character": 3 }
    }
  },
  "severity": 2, // 1: error; 2: warning; 4: info
  "message": "module-level const declarations are always non-instance ...",
  "source": "Parser",
  "relatedInformation": [ // Additional messages, if any
    {
      "location": { ... }, // Like above
      "message": "...",
    }
  ]
}
```

13.9.5. Controlling language features

These options allow some Dafny language features to be enabled or disabled. Some of these options exist for backward compatibility with older versions of Dafny.

- `--default-function-opacity:<transparent|autoRevealDependencies|opaque>` - Change the default opacity of functions.
  - `transparent` (default) means functions are transparent, can be manually made opaque and then revealed.
  - `autoRevealDependencies` makes all functions not explicitly labelled as opaque to be opaque but reveals them automatically in scopes which do not have `{:autoRevealDependencies false}`.
  - `opaque` means functions are always opaque so the opaque keyword is not needed, and functions must be revealed everywhere needed for a proof.

- `--function-syntax` (value '3' or '4') - permits a choice of using the Dafny 3 syntax (`function` and `function method`) or the Dafny 4 syntax (`ghost function` and `function`)

- `--quantifier-syntax` (value '3' or '4') - permits a choice between the Dafny 3 and Dafny 4 syntax for quantifiers

- `--unicode-char` - if false, the `char` type represents any UTF-16 code unit, that is, any 16-bit value, including surrogate code points and allows `\uXXXX` escapes in string and character literals. If true, `char` represnts any Unicode scalar value, that is, any Unicode code point excluding surrogates and allows `\U{X..X}` escapes in string and character literals. The default is false for Dafny version 3 and true for version 4. The legacy option was `-unicodeChar:<n>` with values 0 and 1 for false and true above.

Legacy options:

- `-noIncludes` - ignore `include` directives in the program.

- `-noExterns` - ignore `extern` attributes in the program.

- `--function-syntax:<version>` (was `-functionSyntax:<version>`) - select what function syntax to recognize. The syntax for functions is changing from Dafny version 3 to version 4. This switch gives early access to the new syntax, and also provides a mode to help with migration. The valid arguments include the following.
  - `3` (default) - compiled functions are written `function method` and `predicate method`. Ghost functions are written `function` and `predicate`.
  - `4` - compiled functions are written `function` and `predicate`. Ghost functions are written `ghost function` and `ghost predicate`.
  - `migration3to4` - compiled functions are written `function method` and `predicate method`. Ghost functions are written `ghost function` and `ghost predicate`. To migrate from version 3 to version 4, use this flag on your version 3 program to flag all occurrences of `function` and `predicate` as parsing errors. These are ghost functions, so change those into the new syntax `ghost function` and `ghost predicate`. Then, start using
    `-functionSyntax:4`. This will flag all occurrences of `function method` and `predicate method` as parsing errors. So, change those to just `function` and `predicate`. As a result, your program will use version 4 syntax and have the same meaning as your previous version 3 program.
  - `experimentalDefaultGhost` - like `migration3to4`, but allow `function` and `predicate` as alternatives to declaring ghost functions and predicates, respectively
  - `experimentalDefaultCompiled` - like `migration3to4`, but allow `function` and `predicate` as alternatives to declaring compiled functions and predicates, respectively
  - `experimentalPredicateAlwaysGhost` - compiled functions are written `function`. Ghost functions are written `ghost function`. Predicates are always ghost and are written `predicate`.

  This option can also be set locally (at the module level) using the `:options` attribute:

```
module {:options "--function-syntax:4"} M {
  predicate CompiledPredicate() { true }
}
```

- `--quantifier-syntax:<version>` (was `-quantifierSyntax:<version>`) - select what quantifier syntax to recognize. The syntax for quantification domains is changing from Dafny version 3 to version 4, more specifically where quantifier ranges (`| <Range>`) are allowed. This switch gives early access to the new syntax.
  - `3` (default) - Ranges are only allowed after all quantified variables are declared. (e.g. `set x, y | 0 <= x < |s| && y in s[x] && 0 <= y :: y`)
  - `4` - Ranges are allowed after each quantified variable declaration. (e.g. `set x | 0 <= x < |s|, y <- s[x] | 0 <= y :: y`)

  Note that quantifier variable domains (`<- <Domain>`) are available in both syntax versions.

- `-disableScopes` - treat all export sets as `export reveal *` to never hide function bodies or type definitions during translation.
- `-allowsGlobals` - allow the implicit class `_default` to contain fields, instance functions, and instance methods. These class members are declared at the module scope, outside of explicit classes. This command-line option is provided to simplify a transition from the behavior in the language prior to version 1.9.3, from which point onward all functions and methods declared at the module scope are implicitly static and field declarations are not allowed at the module scope.

13.9.6. Controlling warnings

These options control what warnings Dafny produces, and whether to treat warnings as errors.

- `--warn-as-errors` (was `-warningsAsErrors`) - treat warnings as errors.
- `--warn-shadowing` (was `-warnShadowing`) - emit a warning if the name of a declared variable caused another variable to be shadowed.
- `--warn-missing-constructor-parentheses` - warn if a constructor name in a pattern might be misinterpreted

Legacy options

- `-deprecation:<n>` - control warnings about deprecated features. The value of `<n>` can be any of the following.
  - `0` - don't issue any warnings.
  - `1` (default) - issue warnings.
  - `2` - issue warnings and advise about alternate syntax.

13.9.7. Controlling verification

These options control how Dafny verifies the input program, including how much it verifies, what techniques it uses to perform verification, and what information it produces about the verification process.

- `--no-verify` - turns off verification (for translate, build, run commands)
- `--verify-included-files` (was `-verifyAllModules`) - verify modules that come from include directives.

  By default, Dafny only verifies files explicitly listed on the command line: if `a.dfy` includes `b.dfy`, a call to `Dafny a.dfy` will detect and report verification errors from `a.dfy` but not from `b.dfy`.

  With this option, Dafny will instead verify everything: all input modules and all their transitive dependencies. This way `Dafny a.dfy` will verify `a.dfy` and all files that it includes (here `b.dfy`), as well all files that these files include, etc.

Running Dafny with this option on the file containing your main result is a good way to ensure that all its dependencies verify.

- `--track-print-effects` - If true, a compiled method, constructor, or iterator is allowed to have print effects only if it is marked with {{:print}}. (default false) The legacy option was `-trackPrintEffects:<n>`) with values 0 or 1 for false and true.
- `--relax-definite-assignment` - control the rules governing definite assignment, the property that every variable is eventually assigned a value before it is used.
  - if false (default), enforce definite-assignment for all non-yield-parameter variables and fields, regardless of their types
  - if false and `--enforce-determinism` is true, then also performs checks in the compiler that no nondeterministic statements are used
  - if true, enforce definite-assignment rules for compiled variables and fields whose types do not support auto-initialization and for ghost variables and fields whose type is possibly empty.
- `--disable-nonlinear-arithmetic` (was `-noNLarith`) - reduce Z3's knowledge of non-linear arithmetic (the operators `*`, `/`, and `%`). Enabling this option will typically require more manual work to complete proofs (by explicitly applying lemmas about non-linear operators), but will also result in more predictable behavior, since Z3 can sometimes get stuck down an unproductive path while attempting to prove things about those operators. (This option will perhaps be replaced by `-arith` in the future. For now, it takes precedence over `-arith`.)

  The behavior of `disable-nonlinear-arithmetic` can be turned on and off on a per-module basis by placing the attribute `{:disable-nonlinear-arithmetic}` after the module keyword. The attribute optionally takes the value `false` to enable nonlinear arithmetic.
- `--manual-lemma-induction` - diables automatic inducntion for lemmas
- `--isolate-assertions` - verify assertions individually
- `--extract-counterexample` - if verification fails, report a potential counterexample as a set of assumptions that can be inserted into the code. Note that Danfy cannot guarantee that the counterexample it reports provably violates the assertion or that the assumptions are not mutually inconsistent (see [34]), so this output should be inspected manually and treated as a hint.

Controlling the proof engine:

- `--cores:<n>` - sets the number or percent of the available cores to be used for verification

[34]The formula sent to the underlying SMT solver is the negation of the formula that the verifier wants to prove - also called a VC or verification condition. Hence, if the SMT solver returns "unsat", it means that the SMT formula is always false, meaning the verifier's formula is always true. On the other side, if the SMT solver returns "sat", it means that the SMT formula can be made true with a special variable assignment, which means that the verifier's formula is false under that same variable assignment, meaning it's a counter-example for the verifier. In practice and because of quantifiers, the SMT solver will usually return "unknown" instead of "sat", but will still provide a variable assignment that it couldn't prove that it does not make the formula true. `dafny` reports it as a "counter-example" but it might not be a real counter-example, only provide hints about what `dafny` knows.

- `--verification-time-limit <seconds>` - imposes a time limit on each verification attempt
- `--verification-error-limit <number>` - limits the number of verification errors reported (0 is no limit)
- `--resource-limit` - states a resource limit (to be used by the backend solver)

Legacy options:

- `-dafnyVerify:<n>` [discouraged] - turn verification of the program on or off. The value of `<n>` can be any of the following.
  - `0` - stop after type checking.
  - `1` - continue on to verification and compilation.
- `-separateModuleOutput` - output verification results for each module separately, rather than aggregating them after they are all finished.
- `-mimicVerificationOf:<dafny version>` - let `dafny` attempt to mimic the verification behavior of a previous version of `dafny`. This can be useful during migration to a newer version of `dafny` when a Dafny program has proofs, such as methods or lemmas, that are highly variable in the sense that their verification may become slower or fail altogether after logically irrelevant changes are made in the verification input.

  Accepted versions are: `3.3`. Note that falling back on the behavior of version 3.3 turns off features that prevent certain classes of verification variability.
- `-noCheating:<n>` - control whether certain assumptions are allowed. The value of `<n>` can be one of the following.
  - `0` (default) - allow `assume` statements and free invariants.
  - `1` - treat all assumptions as `assert` statements, and drop free invariants.
- `-induction:<n>` - control the behavior of induction. The value of `<n>` can be one of the following.
  - `0` - never do induction, not even when attributes request it.
  - `1` - apply induction only when attributes request it.
  - `2` - apply induction as requested (by attributes) and also for heuristically chosen quantifiers.
  - `3` - apply induction as requested, and for heuristically chosen quantifiers and lemmas.
  - `4` (default) - apply induction as requested, and for all lemmas.
- `-inductionHeuristic:<n>` - control the heuristics used for induction. The value of `<n>` can be one of the following.
  - `0` - use the least discriminating induction heuristic (that is, lean toward applying induction more often).

  - `1`, `2`, `3`, `4`, `5` - use an intermediate heuristic, ordered as follows as far as how discriminating they are: $0 < 1 < 2 < (3,4) < 5 < 6$.
  - `6` (default) - use the most discriminating induction heuristic.

- `-allocated:<n>` - specify defaults for where Dafny should assert and assume `allocated(x)` for various parameters `x`, local variables `x`, bound variables `x`, etc. Lower `<n>` may require more manual `allocated(x)` annotations and thus may be more difficult to use. The value of `<n>` can be one of the following.
  - `0` - never assume or assert `allocated(x)` by default.
  - `1` - assume `allocated(x)` only for non-ghost variables and fields. (These assumptions are free, since non-ghost variables always contain allocated values at run-time.) This option may speed up verification relative to `-allocated:2`.
  - `2` - assert/assume `allocated(x)` on all variables, even bound variables in quantifiers. This option is the easiest to use for code that uses the heap heavily.
  - `3` - (default) make frugal use of heap parameters.
  - `4` - like `3` but add `allocated` antecedents when ranges don't imply allocatedness.

  Warning: this option should be chosen consistently across an entire project; it would be unsound to use different defaults for different files or modules within a project. Furthermore, modes `-allocated:0` and `-allocated:1` let functions depend on the allocation state, which is not sound in general.

- `-noAutoReq` - ignore `autoReq` attributes, and therefore do not automatically generate `requires` clauses.
- `-autoReqPrint:<file>` - print the requires clauses that were automatically generated by `autoReq` to the given `<file>`.
- `-arith:<n>` - control how arithmetic is modeled during verification. This is an experimental switch, and its options may change. The value of `<n>` can be one of the following.
  - `0` - use Boogie/Z3 built-ins for all arithmetic operations.
  - `1` (default) - like `0`, but introduce symbolic synonyms for `*`, `/`, and `%`, and allow these operators to be used in triggers.
  - `2` - like `1`, but also introduce symbolic synonyms for `+` and `-`.
  - `3` - turn off non-linear arithmetic in the SMT solver. Still use Boogie/Z3 built-in symbols for all arithmetic operations.
  - `4` - like `3`, but introduce symbolic synonyms for `*`, `/`, and `%`, and allow these operators to be used in triggers.
  - `5` - like `4`, but also introduce symbolic synonyms for `+` and `-`.
  - `6` - like `5`, and introduce axioms that distribute `+` over `*`.
  - `7` - like `6`, and introduce facts about the associativity of literal arguments over `*`.

    - `8` - like `7`, and introduce axioms for the connection between `*`, `/`, and `%`.
    - `9` - like `8`, and introduce axioms for sign of multiplication.
    - `10` - like `9`, and introduce axioms for commutativity and associativity of `*`.

- `-autoTriggers:<n>` - control automatic generation of `{:trigger}` annotations. See triggers. The value of `<n>` can be one of the following.
    - `0` - do not generate `{:trigger}` annotations for user-level quantifiers.
    - `1` (default) - add a `{:trigger}` annotation to each user-level quantifier. Existing annotations are preserved.

- `-rewriteFocalPredicates:<n>` - control rewriting of predicates in the body of prefix lemmas. See the section about nicer extreme proofs. The value of `<n>` can be one of the following.
    - `0` - don't rewrite predicates in the body of prefix lemmas.
    - `1` (default) - in the body of prefix lemmas, rewrite any use of a focal predicate `P` to `P#[_k-1]`.

13.9.8. Controlling compilation

These options control what code gets compiled, what target language is used, how compilation proceeds, and whether the compiled program is immediately executed.

- `--target:<s>` or `-t:<s>` (was `-compileTarget:<s>`) - set the target programming language for the compiler. The value of `<s>` can be one of the following.
    - `cs` - C# . Produces a .dll file that can be run using `dotnet`. For example, `dafny Hello.dfy` will produce `Hello.dll` and `Hello.runtimeconfig.json`. The dll can be run using `dotnet Hello.dll`.
    - `go` - Go. The default output of `dafny Hello.dfy -compileTarget:go` is in the `Hello-go` folder. It is run using ``GOPATH=`pwd`/Hello-go/ GO111MODULE=auto go run Hello-go/src/Hello.go``
    - `js` - Javascript. The default output of `dafny Hello.dfy -compileTarget:js` is the file `Hello.js`, which can be run using `node Hello.js`. (You must have `bignumber.js` installed.)
    - `java` - Java. The default output of `dafny Hello.dfy -compileTarget:java` is in the `Hello-java` folder. The compiled program can be run using `java -cp Hello-java:Hello-java/DafnyRuntime.jar Hello`.
    - `py` - Python. The default output of `dafny Hello.dfy -compileTarget:py` is in the `Hello-py` folder. The compiled program can be run using `python Hello-py/Hello.py`, where `python` is Python version 3.

  - `cpp` - C++. The default output of `dafny Hello.dfy -compileTarget:cpp` is `Hello.exe` and other files written to the current folder. The compiled program can be run using `./Hello.exe`.

- `--input <file>` - designates files to be include in the compilation in addition to the main file in `dafny run`; these may be non-.dfy files; this option may be specified more than once
- `--output:<file>` or `-o:<file>` (was `-out:<file>`) - set the name to use for compiled code files.

By default, `dafny` reuses the name of the Dafny file being compiled. Compilers that generate a single file use the file name as-is (e.g. the C# backend will generate `<file>.dll` and optionally `<file>.cs` with `-spillTargetCode`). Compilers that generate multiple files use the file name as a directory name (e.g. the Java backend will generate files in directory `<file>-java/`). Any file extension is ignored, so `-out:<file>` is the same as `-out:<file>.<ext>` if `<file>` contains no periods.

- `--include-runtime` - include the runtime library for the target language in the generated artifacts. This is true by default for build and run, but false by default for translate. The legacy option `-useRuntimeLib` had the opposite effect: when enabled, the compiled assembly referred to the pre-built `DafnyRuntime.dll` in the compiled assembly rather than including `DafnyRuntime.cs` in the build process.

Legacy options:

- `-compile:<n>` - [obsolete - use `dafny build` or `dafny run`] control whether compilation happens. The value of `<n>` can be one of the following. Note that if the program is compiled, it will be compiled to the target language determined by the `-compileTarget` option, which is C# by default.
  - `0` - do not compile the program
  - `1` (default) - upon successful verification, compile the program to the target language.
  - `2` - always compile, regardless of verification success.
  - `3` - if verification is successful, compile the program (like option `1`), and then if there is a `Main` method, attempt to run the program.
  - `4` - always compile (like option `2`), and then if there is a `Main` method, attempt to run the program.
- `-spillTargetCode:<n>` - [obsolete - use `dafny translate`) control whether to write out compiled code in the target language (instead of just holding it in internal temporary memory). The value of `<n>` can be one of the following.
  - `0` (default) - don't make any extra effort to write the textual target program (but still compile it, if `-compile` indicates to do so).
  - `1` - write it out to the target language, if it is being compiled.
  - `2` - write the compiled program if it passes verification, regardless of the `-compile` setting.

  - `3` - write the compiled program regardless of verification success and the `-compile` setting.

Note that some compiler targets may (always or in some situations) write out the textual target program as part of compilation, in which case `-spillTargetCode:0` behaves the same way as `-spillTargetCode:1`.

- `-Main:<name>` - specify the (fully-qualified) name of the method to use as the executable entry point. The default is the method with the `{:main}` attribute, or else the method named `Main`.
- `-compileVerbose:<n>` - control whether to write out compilation progress information. The value of `<n>` can be one of the following.
  - `0` - do not print any information (silent mode)
  - `1` (default) - print information such as the files being created by the compiler
- `-coverage:<file>` - emit branch-coverage calls and outputs into `<file>`, including a legend that gives a description of each source-location identifier used in the branch-coverage calls. (Use `-` as `<file>` to print to the console.)
- `-optimize` - produce optimized C# code by passing the `/optimize` flag to the `csc` executable.
- `-optimizeResolution:<n>` - control optimization of method target resolution. The value of `<n>` can be one of the following.
  - `0` - resolve and translate all methods.
  - `1` - translate methods only in the call graph of the current verification target.
  - `2` (default) - as in `1`, but resolve only methods that are defined in the current verification target file, not in included files.
- `-testContracts:<mode>` - test certain function and method contracts at runtime. This works by generating a wrapper for each function or method to be tested that includes a sequence of `expect` statements for each requires clause, a call to the original, and sequence of `expect` statements for each `ensures` clause. This is particularly useful for code marked with the `{:extern}` attribute and implemented in the target language instead of Dafny. Having runtime checks of the contracts on such code makes it possible to gather evidence that the target-language code satisfies the assumptions made of it during Dafny verification through mechanisms ranging from manual tests through fuzzing to full verification. For the latter two use cases, having checks for `requires` clauses can be helpful, even if the Dafny calling code will never violate them.

  The `<mode>` parameter can currently be one of the following.
  - `Externs` - insert dynamic checks when calling any function or method marked with the `{:extern}` attribute, wherever the call occurs.
  - `TestedExterns` - insert dynamic checks when calling any function or method marked with the `{:extern}` attribute directly from a function or method marked with the

`{:test}` attribute.

13.9.9. Controlling Boogie

Dafny builds on top of Boogie, a general-purpose intermediate language for verification. Options supported by Boogie on its own are also supported by Dafny. Some of the Boogie options most relevant to Dafny users include the following. We use the term "procedure" below to refer to a Dafny function, lemma, method, or predicate, following Boogie terminology.

- `--solver-path` - specifies a custom SMT solver to use
- `--solver-plugin` - specifies a plugin to use as the SMT solver, instead of an external pdafny translaterocess
- `--boogie` - arguments to send to boogie

Legacy options:

- `-proc:<name>` - verify only the procedure named `<name>`. The name can include `*` to indicate arbitrary sequences of characters.
- `-trace` - print extra information during verification, including timing, resource use, and outcome for each procedure incrementally, as verification finishes.
- `-randomSeed:<n>` - turn on randomization of the input that Boogie passes to the SMT solver and turn on randomization in the SMT solver itself.

  Certain Boogie inputs cause proof variability in the sense that changes to the input that preserve its meaning may cause the output to change. The `-randomSeed` option simulates meaning-preserving changes to the input without requiring the user to actually make those changes.

  The `-randomSeed` option is implemented by renaming variables and reordering declarations in the input, and by setting solver options that have similar effects.
- `-randomSeedIterations:<n>` - attempt to prove each VC `<n>` times with `<n>` random seeds. If `-randomSeed` has been provided, each proof attempt will use a new random seed derived from this original seed. If not, it will implicitly use `-randomSeed:0` to ensure a difference between iterations. This option can be very useful for identifying input programs for which verification is highly variable. If the verification times or solver resource counts associated with each proof attempt vary widely for a given procedure, small changes to that procedure might be more likely to cause proofs to fail in the future.
- `-vcsSplitOnEveryAssert` - prove each (explicit or implicit) assertion in each procedure separately. See also the attribute `{:isolate_assertions}` for restricting this option on specific procedures. By default, Boogie attempts to prove that every assertion in a given procedure holds all at once, in a single query to an SMT solver. This usually performs well, but sometimes causes the solver to take longer. If a proof that you believe should succeed is timing out, using this option can sometimes help.
- `-timeLimit:<n>` - spend at most `<n>` seconds attempting to prove any single SMT query. This setting can also be set per method using the attribute `{:timeLimit n}`.

- `-rlimit:<n>` - set the maximum solver resource count to use while proving a single SMT query. This can be a more deterministic approach than setting a time limit. To choose an appropriate value, please refer to the documentation of the attribute `{:rlimit}` that can be applied per procedure.
- `-print:<file>` - print the translation of the Dafny file to a Boogie file.

If you have Boogie installed locally, you can run the printed Boogie file with the following script:

```
DOTNET=$(which dotnet)

BOOGIE_ROOT="path/to/boogie/Source"
BOOGIE="$BOOGIE_ROOT/BoogieDriver/bin/Debug/net6.0/BoogieDriver.dll"

if [[ ! -x "$DOTNET" ]]; then
    echo "Error: Dafny requires .NET Core to run on non-Windows systems."
    exit 1
fi

#Uncomment if you prefer to use the executable instead of the DLL
#BOOGIE=$(which boogie)

BOOGIE_OPTIONS="/infer:j"
PROVER_OPTIONS="\
  /proverOpt:O:auto_config=false \
  /proverOpt:O:type_check=true \
  /proverOpt:O:smt.case_split=3 \
  /proverOpt:O:smt.qi.eager_threshold=100 \
  /proverOpt:O:smt.delay_units=true \
  /proverOpt:O:smt.arith.solver=2 \
  "

"$DOTNET" "$BOOGIE" $BOOGIE_OPTIONS $PROVER_OPTIONS "$@"
#Uncomment if you want to use the executable instead of the DLL
#"$BOOGIE" $BOOGIE_OPTIONS $PROVER_OPTIONS "$@"
```

13.9.10. Controlling the prover

Much of controlling the prover is accomplished by controlling verification condition generation (25.9.7) or Boogie (Section 13.9.9). The following options are also commonly used:

- `--verification-error-limit:<n>` - limits the number of verification errors reported per procedure. Default is 5; 0 means as many as possible; a small positive number runs faster but a large positive number reports more errors per run
- `--verification-time-limit:<n>` (was `-timeLimit:<n>`) - limits the number of seconds spent trying to verify each procedure.

13.9.11. Controlling test generation

Dafny is capable of generating unit (runtime) tests. It does so by asking the prover to solve for values of inputs to a method that cause the program to execute specific blocks or paths. A detailed description of how to do this is given in a separate document.

14. Dafny VSCode extension and the Dafny Language Server

14.1. Dafny functionality within VSCode

There is a language server for Dafny, which implements the Language Server Protocol. This server is used by the Dafny VSCode Extension; it currently offers the following features: - Quick syntax highlighting - As-you-type parsing, resolution and verification diagnostics - Support for Dafny plugins - Expanded explanations (in addition to the error message) for selected errors (and more being added), shown by hovering - Quick fixes for selected errors (and more being added) - Limited support for symbol completion - Limited support for code navigation - Counter-example display - Highlighting of ghost statements - Gutter highlights - A variety of Preference settings

Most of the Dafny functionality is simply there when editing a .dfy file with VSCode that has the Dafny extension installed. Some actions are available through added menu items. The Dafny functionality within VSCode can be found in these locations:

- The preferences are under the menu Code->Preferences->Settings->Dafny extension configuration. There are two sections of settings.
- A hover over an error location will bring up a hover popup, which will show expanded error information and any quick fix options that are available.
- Within a .dfy editor, a right-click brings up a context menu, which has a menu item 'Dafny'. Under it are actions to Build or Run a program, to turn on or off counterexample display, find definitions, and the like.

14.2. Gutter highlights

Feedback on a program is show visually as underlining with squiggles within the text and as various markings in various colors in the *gutter* down the left side of an editor window.

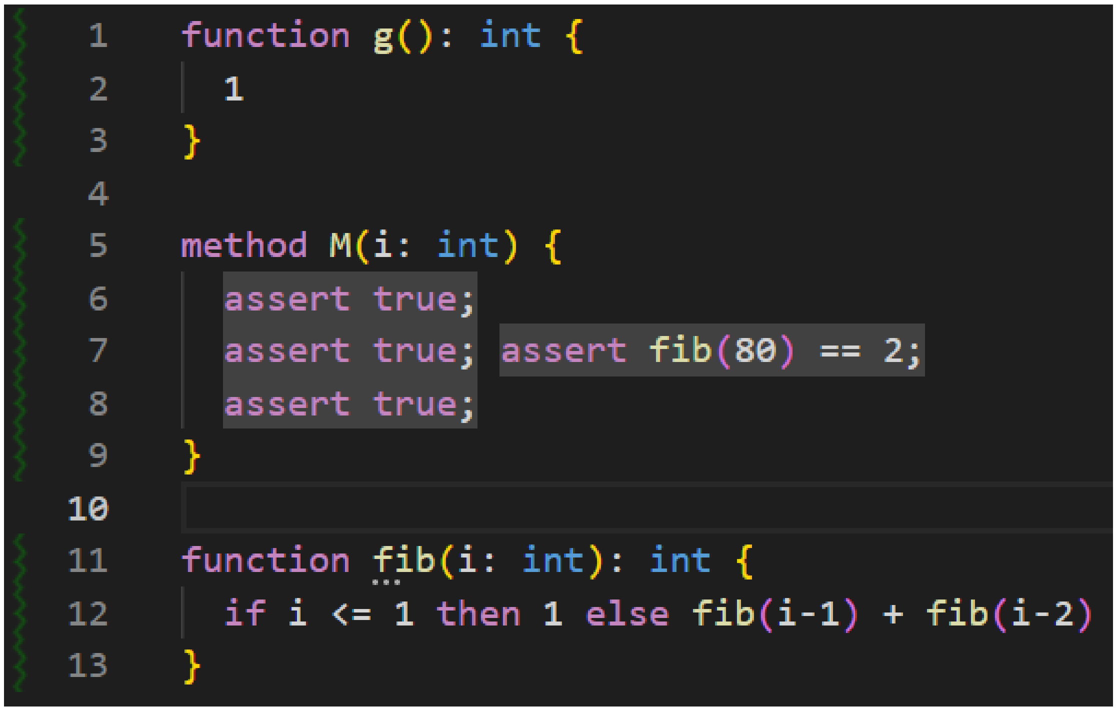
The first time a file is loaded, the gutter will highlight in a transparent squiggly green line all the methods that need to be verified, like this:

When the file is saved (in verification on save), or whenever the Dafny verifier is ready (in verification on change), it will start to verify methods. That line will turn into a thin green rectangle on methods that have been verified, and display an animated less transparent green squiggly line on methods that are being actively verified:

When the verification finishes, if a method, a function, a constant with default initialization or a subset type with a witness has some verification errors in it, the editor will display two yellow vertical rails indicating an error context.

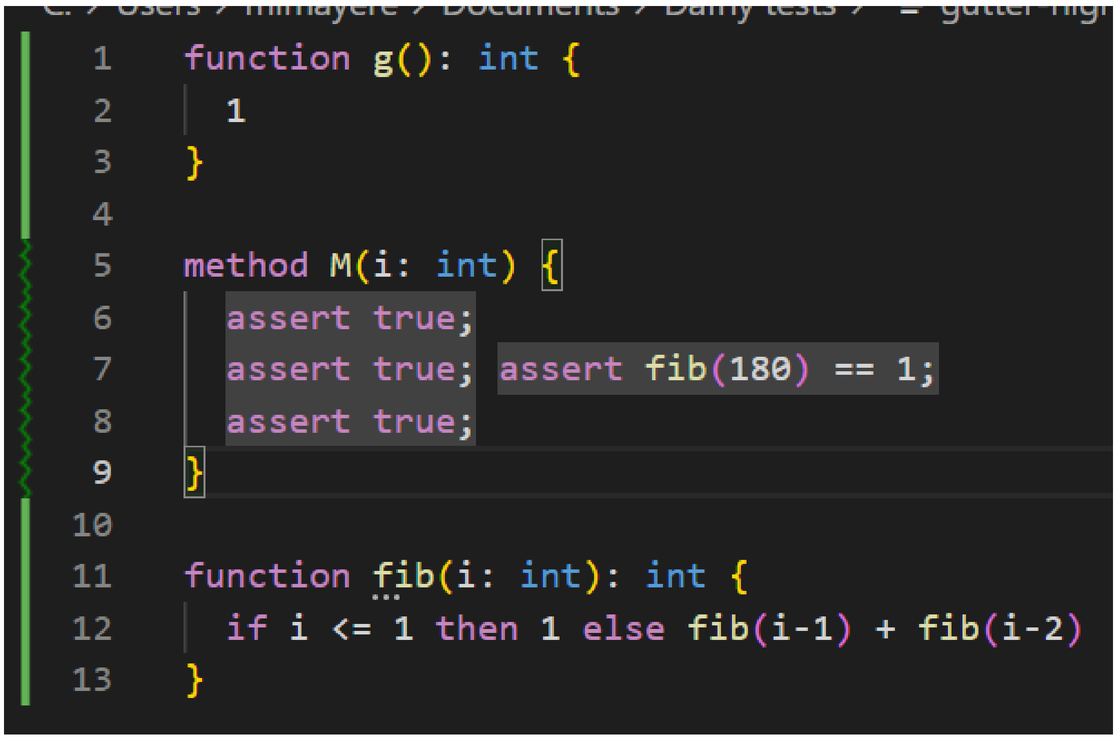
Inside this context, if there is a failing assertion on a line, it will fill the gap between the vertical yellow bars with a red rectangle, even if there might be other assertions that are verified on the line. If there is no error on a line, but there is at least one assertion that verified, it will display a green disk with a white checkmark on it, which can be used to check progress in a proof search.

As soon as a line is changed, the gutter icons turn transparent and squiggly, to indicate their obsolescence.



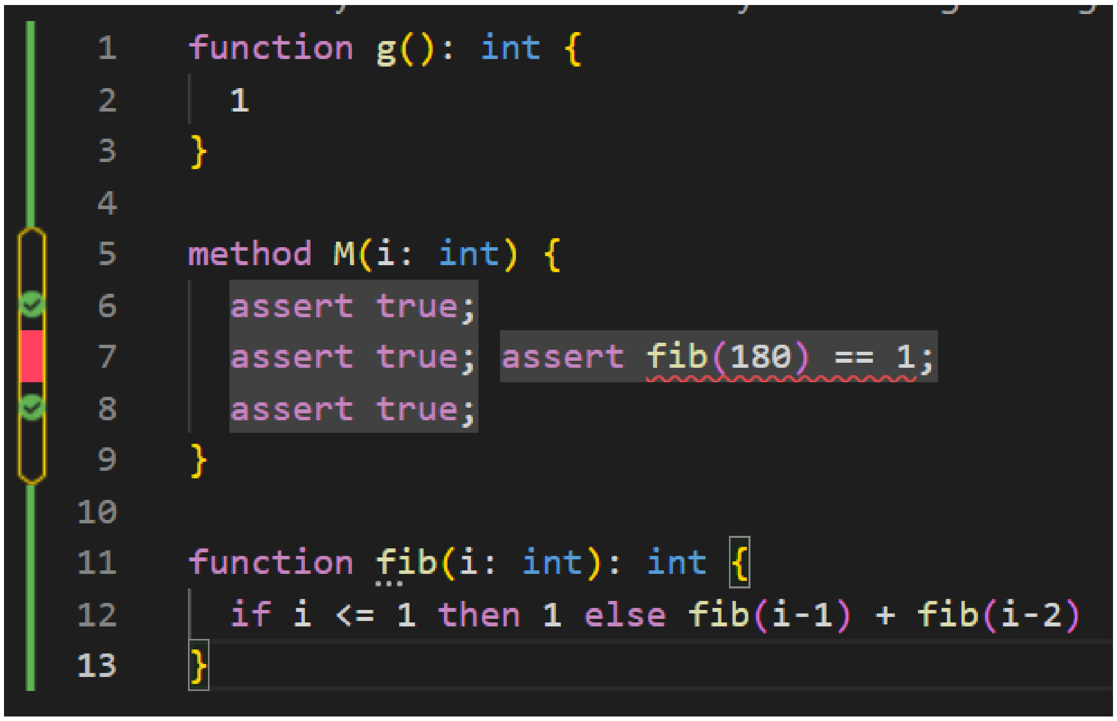
```
function g(): int {
  1
}

method M(i: int) {
  assert true;
  assert true; assert fib(80) == 2;
  assert true;
}

function fib(i: int): int {
  if i <= 1 then 1 else fib(i-1) + fib(i-2)
}
```

Figure 1: image



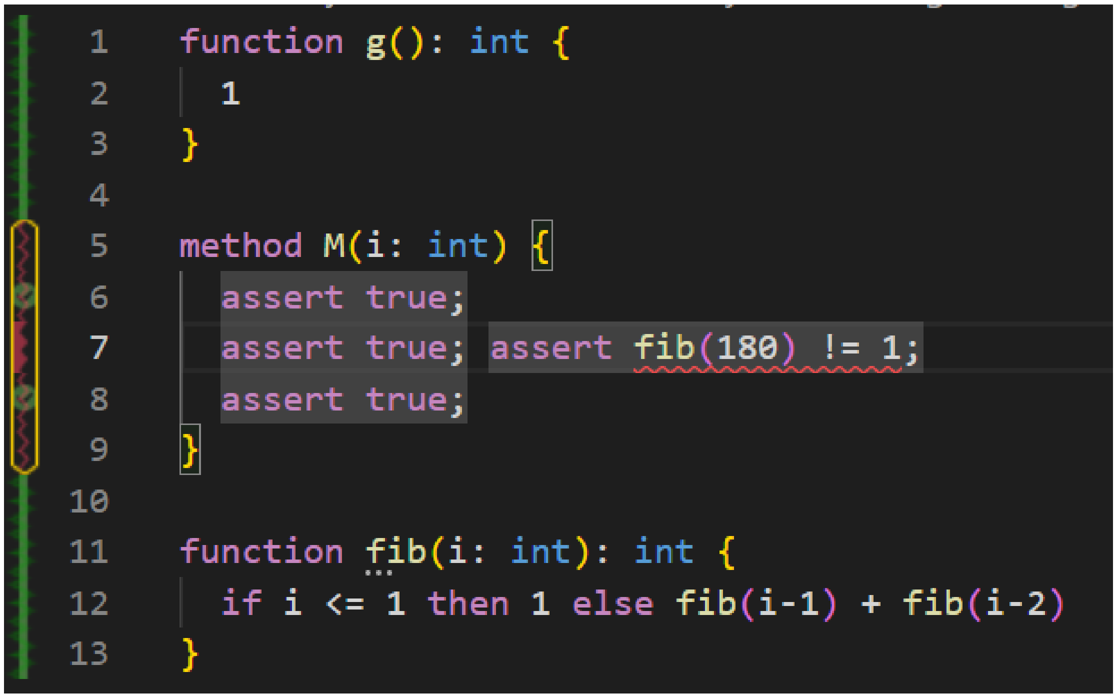
```
function g(): int {
  1
}

method M(i: int) {
  assert true;
  assert true; assert fib(180) == 1;
  assert true;
}

function fib(i: int): int {
  if i <= 1 then 1 else fib(i-1) + fib(i-2)
}
```

Figure 2: image



```
function g(): int {
  1
}

method M(i: int) {
  assert true;
  assert true; assert fib(180) == 1;
  assert true;
}

function fib(i: int): int {
  if i <= 1 then 1 else fib(i-1) + fib(i-2)
}
```

Figure 3: image

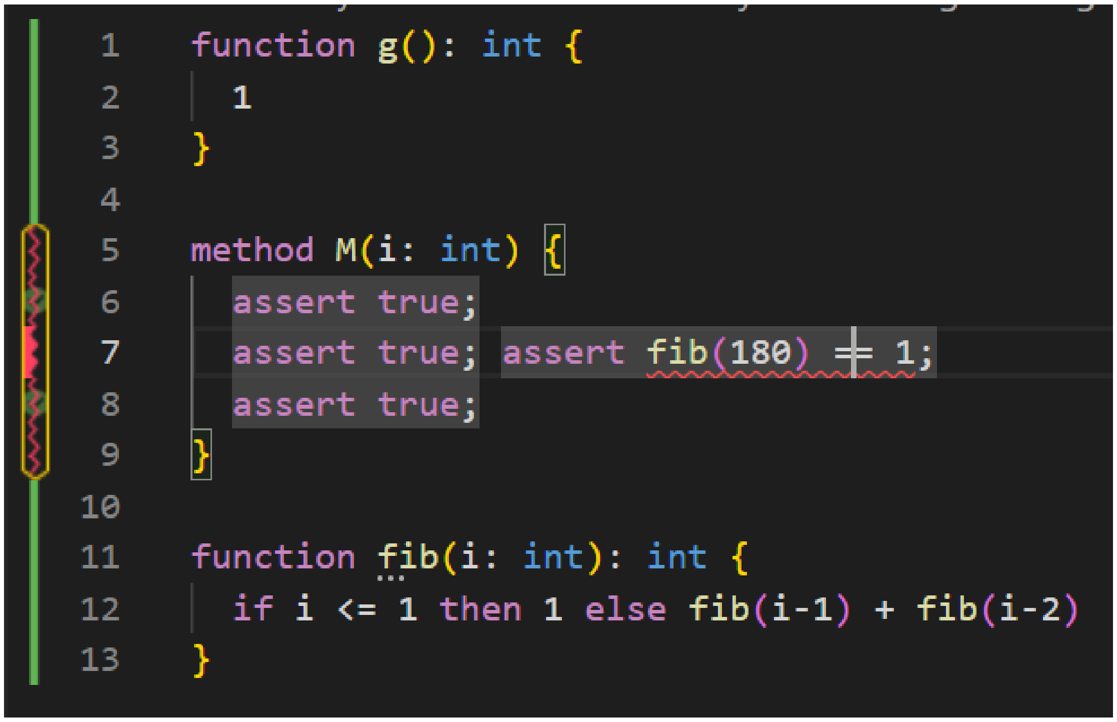


```
function g(): int {
  1
}

method M(i: int) {
  assert true;
  assert true; assert fib(180) != 1;
  assert true;
}

function fib(i: int): int {
  if i <= 1 then 1 else fib(i-1) + fib(i-2)
}
```

Figure 4: image

The red error rectangles occupy only half the horizontal space, to visualise their possible obsolescence.

When the file is saved (in verification on save), or as soon as possible otherwise, these squiggly icons will be animated while the Dafny verifier inspect the area.



```
function g(): int {
  1
}

method M(i: int) {
  assert true;
  assert true; assert fib(180) == 1;
  assert true;
}

function fib(i: int): int {
  if i <= 1 then 1 else fib(i-1) + fib(i-2)
}
```

Figure 5: image

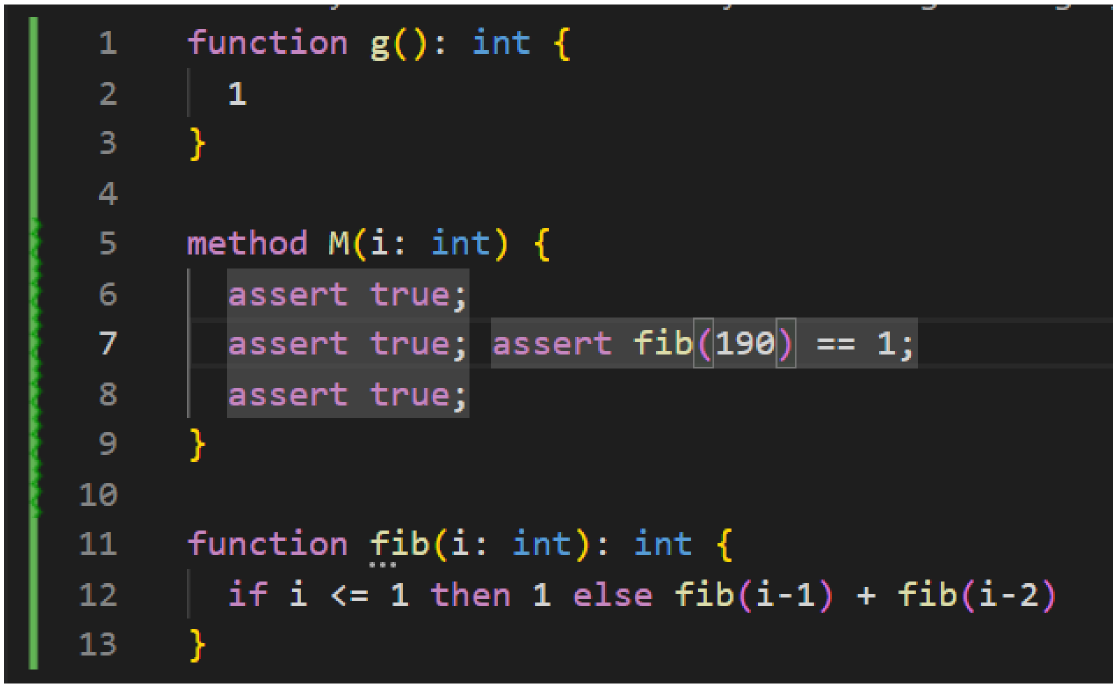
If the method was verifying before a change, instead of two yellow vertical bars with a red squiggly line, the gutter icons display an animated squiggly but more firm green line, thereby indicating that the method used to verify, but Dafny is still re-verifying it.

If there is a parse or resolution error, the previous gutter icons turn gray and a red triangle indicates the position of the parse or resolution error.

14.3. The Dafny Server

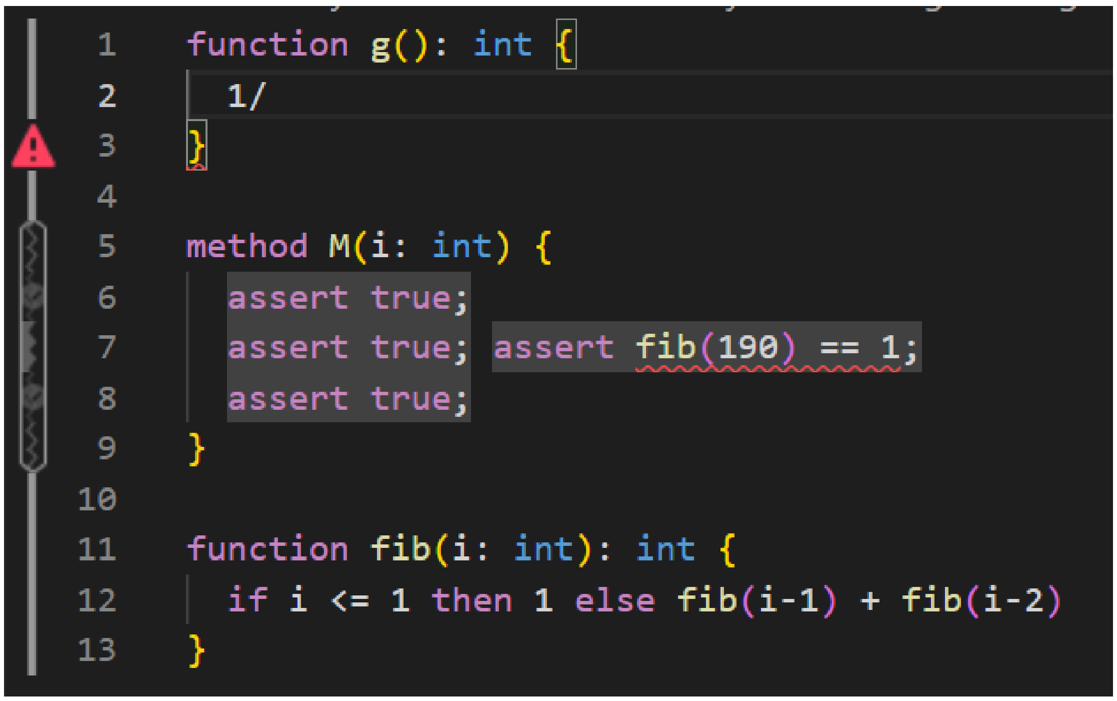
Before Dafny implemented the official Language Server Protocol, it implemented its own protocol for Emacs, which resulted in a project called DafnyServer. While the latest Dafny releases still contain a working DafnyServer binary, this component has been feature frozen since 2022, and it may not support features that were added to Dafny after that time. We do not recommend using it.

The Dafny Server has integration tests that serve as the basis of the documentation.



```
function g(): int {
  1
}

method M(i: int) {
  assert true;
  assert true; assert fib(190) == 1;
  assert true;
}

function fib(i: int): int {
  if i <= 1 then 1 else fib(i-1) + fib(i-2)
}
```

Figure 6: image



```
function g(): int {
  1/
}

method M(i: int) {
  assert true;
  assert true; assert fib(190) == 1;
  assert true;
}

function fib(i: int): int {
  if i <= 1 then 1 else fib(i-1) + fib(i-2)
}
```

Figure 7: image

The server is essentially a REPL, which produces output in the same format as the Dafny CLI; clients thus do not need to understand the internals of Dafny's caching. A typical editing session proceeds as follows:

- When a new Dafny file is opened, the editor starts a new instance of the Dafny server. The cache is blank at that point.
- The editor sends a copy of the buffer for initial verification. This takes some time, after which the server returns a list of errors.
- The user makes modifications; the editor periodically sends a new copy of the buffer's contents to the Dafny server, which quickly returns an updated list of errors.

The client-server protocol is sequential, uses JSON, and works over ASCII pipes by base64-encoding utf-8 queries. It defines one type of query, and two types of responses:

Queries are of the following form:

```
verify
<base64 encoded JSON payload>
[[DAFNY-CLIENT: EOM]]
```

Responses are of the following form:

```
<list of errors and usual output, as produced by the Dafny CLI>
[SUCCESS] [[DAFNY-SERVER: EOM]]
```

or

```
<error message>
[FAILURE] [[DAFNY-SERVER: EOM]]
```

The JSON payload is an utf-8 encoded string resulting of the serialization of a dictionary with 4 fields: * args: An array of Dafny arguments, as passed to the Dafny CLI * source: A Dafny program, or the path to a Dafny source file. * sourceIsFile: A boolean indicating whether the 'source' argument is a Dafny program or the path to one. * filename: The name of the original source file, to be used in error messages

For small files, embedding the Dafny source directly into a message is convenient; for larger files, however, it is generally better for performance to write the source snapshot to a separate file, and to pass that to Dafny by setting the 'sourceIsFile' flag to true.

For example, if you compile and run `DafnyServer.exe`, you could paste the following command:

```
verify
eyJhcmdzIjpbIi9jb21waWxlOjAiLCIvcHJpbnRUb29sdGlwcyIsIi90aW1lTGltaXQ6MjAiXSwi
ZmlsZW5hbWUiOiJ0cmFuc2NyaXB0Iiwic291cmNlIjoibWV0aG9kIEEoYTppbnQpIHJldHVybnMg
KGI6IGludCkge1xuICBiIDo9IGE7XG4gIGFzc2VydCBmYWxzZTtcbn1cbiIsInNvdXJjZUlzRmls
ZSI6ZmFsc2V9
[[DAFNY-CLIENT: EOM]]
```

The interpreter sees the command `verify`, and then starts reading every line until it sees `[[DAFNY-CLIENT: EOM]]` The payload is a base64 encoded string that you could encode or decode using JavaScript's `atob` and `btoa` function. For example, the payload above was generated using the following code:

```
btoa(JSON.stringify({
  "args": [
    "/compile:0",
    "/printTooltips",
    "/timeLimit:20"
   ],
   "filename":"transcript",
   "source":
`method A(a:int) returns (b: int) {
   b := a;
   assert false;
}
`,"sourceIsFile": false}))
=== "eyJhcmdzIjpbIi9jb21waWxlOjAiLCIvcHJpbnRUb29sdGlwcyIsIi90aW1lTGltaXQ6MjAiXSwiZmlsZW5hbWUiOiJ0c
```

Thus to decode such output, you'd manually use `JSON.parse(atob(payload))`.

15. Plugins to Dafny

Dafny has a plugin architecture that permits users to build tools for the Dafny language without having to replicate parsing and name/type resolution of Dafny programs. Such a tool might just do some analysis on the Dafny program, without concern for verifying or compiling the program. Or it might modify the program (actually, modify the program's AST) and then continue on with verification and compilation with the core Dafny tool. A user plugin might also be used in the Language Server and thereby be available in the VSCode (or other) IDE.

This is an experimental aspect of Dafny. *The plugin API directly exposes the Dafny AST, which is constantly evolving. Hence, always recompile your plugin against the binary of Dafny that will be importing your plugin.*

Plugins are libraries linked to a `Dafny.dll` of the same version as the Language Server. A plugin typically defines:

- Zero or one class extending `Microsoft.Dafny.Plugins.PluginConfiguration`, which receives plugins arguments in its method `ParseArguments`, and
  1. Can return a list of `Microsoft.Dafny.Plugins.Rewriter`s when its method `GetRewriters()` is called by Dafny,
  2. Can return a list of `Microsoft.Dafny.Plugins.Compiler`s when its method `GetCompilers()` is called by Dafny,
  3. If the configuration extends the subclass `Microsoft.Dafny.LanguageServer.Plugins.PluginConfigurati`
     1. Can return a list of `Microsoft.Dafny.LanguageServer.Plugins.DafnyCodeActionProvider`s when its method `GetDafnyCodeActionProviders()` is called by the Dafny Language Server.
     2. Can return a modified version of `OmniSharp.Extensions.LanguageServer.Server.LanguageServerOp` when its method `WithPluginHandlers()` is called by the Dafny Language Server.
- Zero or more classes extending `Microsoft.Dafny.Plugins.Rewriter`. If a configuration class is provided, it is responsible for instantiating them and returning them in `GetRewriters()`. If no configuration class is provided, an automatic configuration will load every defined `Rewriter` automatically.
- Zero or more classes extending `Microsoft.Dafny.Plugins.Compiler`. If a configuration class is provided, it is responsible for instantiating them and returning them in `GetCompilers()`. If no configuration class is provided, an automatic configuration will load every defined `Compiler` automatically.
- Zero or more classes extending `Microsoft.Dafny.LanguageServer.Plugins.DafnyCodeActionProvider`. Only a configuration class of type `Microsoft.Dafny.LanguageServer.Plugins.PluginConfiguration` can be responsible for instantiating them and returning them in `GetDafnyCodeActionProviders()`.

The most important methods of the class `Rewriter` that plugins override are

- (experimental) `PreResolve(ModuleDefinition)`: Here you can optionally modify the AST before it is resolved.
- `PostResolve(ModuleDefinition)`: This method is repeatedly called with every resolved and type-checked module, before verification. Plugins override this method typically to report additional diagnostics.
- `PostResolve(Program)`: This method is called once after all `PostResolve(ModuleDefinition)`

have been called.

Plugins are typically used to report additional diagnostics such as unsupported constructs for specific compilers (through the methods `Èrror(...)` and `Warning(...)` of the field `Reporter` of the class `Rewriter`)

Note that all plugin errors should use the original program's expressions' token and NOT `Token.NoToken`, else no error will be displayed in the IDE.

15.1. Language Server plugin tutorial

In this section, we will create a plugin that enhances the functionality of the Language Server. We will start by showing the steps needed to create a plugin, followed by an example implementation that demonstrates how to provide more code actions and add custom request handlers.

15.1.1. Create plugin project

Assuming the Dafny source code is installed in the folder `dafny/` start by creating an empty folder next to it, e.g. `PluginTutorial/`

```
mkdir PluginTutorial
cd PluginTutorial
```

Then, create a dotnet class project

```
dotnet new classlib
```

It will create a file `Class1.cs` that you can rename

```
mv Class1.cs MyPlugin.cs
```

Open the newly created file `PluginTutorial.csproj`, and add the following after `</PropertyGroup>`:

```
  <ItemGroup>
    <ProjectReference Include="../dafny/source/DafnyLanguageServer/DafnyLanguageServer.csproj" />
  </ItemGroup>
```

15.1.2. Implement plugin

15.1.2.1. Code actions plugin This code action plugin will add a code action that allows you to place a dummy comment in front of the first method name, only if the selection is on the line of the method.

Open the file `MyPlugin.cs`, remove everything, and write the imports and a namespace:

```
using Microsoft.Dafny;
using Microsoft.Dafny.LanguageServer.Plugins;
using Microsoft.Boogie;
using Microsoft.Dafny.LanguageServer.Language;
using System.Linq;
using Range = OmniSharp.Extensions.LanguageServer.Protocol.Models.Range;

namespace MyPlugin;
```

After that, add a `PluginConfiguration` that will expose all the quickfixers of your plugin. This class will be discovered and instantiated automatically by Dafny.

```
public class TestConfiguration : PluginConfiguration {
  public override DafnyCodeActionProvider[] GetDafnyCodeActionProviders() {
    return new DafnyCodeActionProvider[] { new AddCommentDafnyCodeActionProvider() };
  }
}
```

Note that you could also override the methods `GetRewriters()` and `GetCompilers()` for other purposes, but this is out of scope for this tutorial.

Then, we need to create the quickFixer `AddCommentDafnyCodeActionProvider` itself:

```
public class AddCommentDafnyCodeActionProvider : DafnyCodeActionProvider {
  public override IEnumerable<DafnyCodeAction> GetDafnyCodeActions(IDafnyCodeActionInput input, Ra
    return new DafnyCodeAction[] { };
  }
}
```

For now, this quick fixer returns nothing. `input` is the program state, and `selection` is where the caret is. We replace the return statement with a conditional that tests whether the selection is on the first line:

```
    var firstTokenRange = input.Program?.GetFirstTopLevelToken()?.GetLspRange();
    if(firstTokenRange != null && firstTokenRange.Start.Line == selection.Start.Line) {
      return new DafnyCodeAction[] {
        // TODO
      };
    } else {
      return new DafnyCodeAction[] { };
    }
```

Every quick fix consists of a title (provided immediately), and zero or more `DafnyCodeActionEdit` (computed lazily). A `DafnyCodeActionEdit` has a `Range` to remove and some `string` to insert instead. All `DafnyCodeActionEdits` of the same `DafnyCodeAction` are applied at the same time if selected.

To create a `DafnyCodeAction`, we can either use the easy-to-use `InstantDafnyCodeAction`, which accepts a title and an array of edits:

```
  return new DafnyCodeAction[] {
    new InstantDafnyCodeAction("Insert comment", new DafnyCodeActionEdit[] {
      new DafnyCodeActionEdit(firstTokenRange.GetStartRange(), "/*First comment*/")
    })
  };
```

or we can implement our custom inherited class of `DafnyCodeAction`:

```
public class CustomDafnyCodeAction: DafnyCodeAction {
  public Range whereToInsert;

  public CustomDafnyCodeAction(Range whereToInsert): base("Insert comment") {
    this.whereToInsert = whereToInsert;
  }
  public override DafnyCodeActionEdit[] GetEdits() {
    return new DafnyCodeActionEdit[] {
      new DafnyCodeActionEdit(whereToInsert.GetStartRange(), "/*A comment*/")
    };
  }
}
```

In that case, we could return:

```
  return new DafnyCodeAction[] {
    new CustomDafnyCodeAction(firstTokenRange)
  };
```

15.1.2.2. Request handler plugin This request handler plugin enhances the Language Server to support a request with a `TextDocumentIdentifier` as parameter, which will return a `bool` value denoting whether the provided `DocumentUri` has any `LoopStmt`'s in it.

Open the file `MyPlugin.cs`, remove everything, and write the imports and a namespace:

```
using OmniSharp.Extensions.JsonRpc;
using OmniSharp.Extensions.LanguageServer.Server;
using OmniSharp.Extensions.LanguageServer.Protocol.Models;
using Microsoft.Dafny.LanguageServer.Plugins;
using Microsoft.Dafny.LanguageServer.Workspace;
using MediatR;
using Microsoft.Dafny;

namespace MyPlugin;
```

After that, add a `PluginConfiguration` that will add all the request handlers of your plugin. This class will be discovered and instantiated automatically by Dafny.

```
public class TestConfiguration : PluginConfiguration {
  public override LanguageServerOptions WithPluginHandlers(LanguageServerOptions options) {
    return options.WithHandler<DummyHandler>();
  }
}
```

Then, we need to create the request handler `DummyHandler` itself:

```
[Parallel]
[Method("dafny/request/dummy", Direction.ClientToServer)]
public record DummyParams : TextDocumentIdentifier, IRequest<bool>;

public class DummyHandler : IJsonRpcRequestHandler<DummyParams, bool> {
  private readonly IProjectDatabase projects;
  public DummyHandler(IProjectDatabase projects) {
    this.projects = projects;
  }
  public async Task<bool> Handle(DummyParams request, CancellationToken cancellationToken) {
    var state = await projects.GetParsedDocumentNormalizeUri(request);
    if (state == null) {
      return false;
    }
    return state.Program.Descendants().OfType<LoopStmt>().Any();
  }
}
```

For more advanced example implementations of request handlers, look at `dafny/Source/DafnyLanguageServer/Handle`

15.1.3. Building plugin

That's it! Now, build your library while inside your folder:

```
> dotnet build
```

This will create the file `PluginTutorial/bin/Debug/net6.0/PluginTutorial.dll`. Now, open VSCode, open Dafny settings, and enter the absolute path to this DLL in the plugins section. Restart VSCode, and it should work!

17. Dafny Grammar

The Dafny grammar has a traditional structure: a scanner tokenizes the textual input into a sequence of tokens; the parser consumes the tokens to produce an AST. The AST is then passed on for name and type resolution and further processing.

Dafny uses the Coco/R lexer and parser generator for its lexer and parser (http://www.ssw.uni-linz.ac.at/Research/Projects/Coco)(Mössenböck, Löberbauer, and Wöß 2013). See the Coco/R Reference manual for details. The Dafny input file to Coco/R is the `Dafny.atg` file in the source tree.

The grammar is an *attributed extended BNF* grammar. The *attributed* adjective indicates that the BNF productions are parameterized by boolean parameters that control variations of the production rules, such as whether a particular alternative is permitted or not. Using such attributes allows combining non-terminals with quite similar production rules, making a simpler, more compact and more readable grammer.

The grammar rules presented here replicate those in the source code, but omit semantic actions, error recovery markers, and conflict resolution syntax. Some uses of the attribute parameters are described informally.

The names of character sets and tokens start with a lower case letter; the names of grammar non-terminals start with an upper-case letter.

17.1. Dafny Syntax

This section gives the definitions of Dafny tokens.

17.1.1. Classes of characters

These definitions define some names as representing subsets of the set of characters. Here,

- double quotes enclose the set of characters constituting the class,
- single quotes enclose a single character (perhaps an escaped representation using \),
- the binary `+` indicates set union,
- binary `-` indicates set difference, and
- `ANY` indicates the set of all (unicode) characters.

```
letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

digit = "0123456789"
posDigit = "123456789"
posDigitFrom2 = "23456789"

hexdigit = "0123456789ABCDEFabcdef"

special = "'_?"

cr        = '\r'
```

```
lf        = '\n'

tab       = '\t'

space     = ' '

nondigitIdChar = letter + special

idchar = nondigitIdChar + digit

nonidchar = ANY - idchar

charChar = ANY - '\'' - '\\' - cr - lf

stringChar = ANY - '"' - '\\' - cr - lf

verbatimStringChar = ANY - '"'
```

A `nonidchar` is any character except those that can be used in an identifier. Here the scanner generator will interpret `ANY` as any unicode character. However, `nonidchar` is used only to mark the end of the `!in` token; in this context any character other than whitespace or printable ASCII will trigger a subsequent scanning or parsing error.

17.1.2. Definitions of tokens

These definitions use

- double-quotes to indicate a verbatim string (with no escaping of characters)
- `'"'` to indicate a literal double-quote character
- vertical bar to indicate alternatives
- square brackets to indicate an optional part
- curly braces to indicate 0-or-more repetitions
- parentheses to indicate grouping
- a `-` sign to indicate set difference: any character sequence matched by the left operand except character sequences matched by the right operand
- a sequence of any of the above to indicate concatenation without whitespace

```
reservedword =
    "abstract" | "allocated" | "as" | "assert" | "assume" |
    "bool" | "break" | "by" |
    "calc" | "case" | "char" | "class" | "codatatype" |
    "const" | "constructor" | "continue" |
    "datatype" | "decreases" |
    "else" | "ensures" | "exists" | "expect" | "export" | "extends" |
    "false" | "for" | "forall" | "fresh" | "function" | "ghost" |
    "if" | "imap" | "import" | "in" | "include" |
    "int" | "invariant" | "is" | "iset" | "iterator" |
    "label" | "lemma" | "map" | "match" | "method" |
    "modifies" | "modify" | "module" | "multiset" |
    "nameonly" | "nat" | "new" | "newtype" | "null" |
    "object" | "object?" | "old" | "opaque" | "opened" | "ORDINAL"
    "predicate" | "print" | "provides" |
    "reads" | "real" | "refines" | "requires" | "return" |
    "returns" | "reveal" | "reveals" |
    "seq" | "set" | "static" | "string" |
    "then" | "this" | "trait" | "true" | "twostate" | "type" |
    "unchanged" | "var" | "while" | "witness" |
    "yield" | "yields" |
    arrayToken | bvToken

arrayToken = "array" [ posDigitFrom2 | posDigit digit { digit }]["?"]

bvToken = "bv" ( 0 | posDigit { digit } )

ident = nondigitIdChar { idchar } - charToken - reservedword

digits = digit {["_"] digit}

hexdigits = "0x" hexdigit {["_"] hexdigit}

decimaldigits = digit {["_"] digit} '.' digit {["_"] digit}

escapedChar =
    ( "\'" | "\"" | "\\" | "\0" | "\n" | "\r" | "\t"
      | "\u" hexdigit hexdigit hexdigit hexdigit
      | "\U{" hexdigit { hexdigit } "}"
    )

charToken = "'" ( charChar | escapedChar ) "'"

stringToken =
    '"' { stringChar | escapedChar }  '"'
  | "@" '"' { verbatimStringChar | '"' '"' } '"'

ellipsis = "..."
```

There are a few words that have a special meaning in certain contexts, but are not reserved words and can be used as identifiers outside of those contexts:

- `least` and `greatest` are recognized as adjectives to the keyword `predicate` (cf. Section 12.4).
- `older` is a modifier for parameters of non-extreme predicates (cf. Section 6.4.6).

The `\uXXXX` form of an `escapedChar` is only used when the option `--unicode-char=false` is set (which is the default for Dafny 3.x); the `\U{XXXXXX}` form of an `escapedChar` is only used when the option `--unicode-char=true` is set (which is the default for Dafny 4.x).

17.2. Dafny Grammar productions

The grammar productions are presented in the following Extended BNF syntax:

- identifiers starting with a lower case letter denote terminal symbols (tokens) as defined in the previous subsection
- identifiers starting with an upper case letter denote nonterminal symbols
- strings (a sequence of characters enclosed by double quote characters) denote the sequence of enclosed characters
- `=` separates the sides of a production, e.g. `A = a b c`
- `|` separates alternatives, e.g. `a b | c | d e` means `a b` or `c` or `d e`
- `( )` groups alternatives, e.g. `(a | b) c` means `a c` or `b c`
- `[ ]` option, e.g. `[a] b` means `a b` or `b`
- `{ }` iteration (0 or more times), e.g. `{a} b` means `b` or `a b` or `a a b` or …
- We allow `|` inside `[ ]` and `{ }`. So `[a | b]` is short for `[(a | b)]` and `{a | b}` is short for `{(a | b)}`.
- `//` in a line introduces a comment that extends to the end-of-the line, but does not terminate the production
- The first production defines the name of the grammar, in this case `Dafny`.

In addition to the Coco rules, for the sake of readability we have adopted these additional conventions.

- We allow `-` to be used. `a - b` means it matches if it matches `a` but not `b`.
- We omit the `.` that marks the end of a CoCo/R production.
- we omit deprecated features.

To aid in explaining the grammar we have added some additional productions that are not present in the original grammar. We name these with a trailing underscore. Inlining these where they are referenced will reconstruct the original grammar.

17.2.1. Programs

(discussion)

```
Dafny = { IncludeDirective_ } { TopDecl(isTopLevel:true, isAbstract: false) } EOF
```

17.2.1.1. Include directives (discussion)

```
IncludeDirective_ = "include" stringToken
```

17.2.1.2. Top-level declarations (discussion)

```
TopDecl(isTopLevel, isAbstract) =
  { DeclModifier }
  ( SubModuleDecl(isTopLevel)
  | ClassDecl
  | DatatypeDecl
  | NewtypeDecl
  | SynonymTypeDecl  // includes abstract types
  | IteratorDecl
  | TraitDecl
  | ClassMemberDecl(allowConstructors: false, isValueType: true, moduleLevelDecl: true)
  )
```

17.2.1.3. Declaration modifiers (discussion)

```
DeclModifier = ( "abstract" | "ghost" | "static" | "opaque" )
```

17.2.2. Modules

```
SubModuleDecl(isTopLevel) = ( ModuleDefinition | ModuleImport | ModuleExport )
```

Module export declarations are not permitted if `isTopLevel` is true.

17.2.2.1. Module Definitions (discussion)

```
ModuleDefinition(isTopLevel) =
  "module" { Attribute } ModuleQualifiedName
  [ "refines" ModuleQualifiedName ]
  "{" { TopDecl(isTopLevel:false, isAbstract) } "}"
```

The `isAbstract` argument is true if the preceding `DeclModifiers` include “abstract”.

17.2.2.2. Module Imports (discussion)

```
ModuleImport =
  "import"
  [ "opened" ]
  ( QualifiedModuleExport
  | ModuleName "=" QualifiedModuleExport
  | ModuleName ":" QualifiedModuleExport
  )

QualifiedModuleExport =
    ModuleQualifiedName [ "`" ModuleExportSuffix ]

ModuleExportSuffix =
  ( ExportId
  | "{" ExportId { "," ExportId } "}"
  )
```

17.2.2.3. Module Export Definitions (discussion)

```
ModuleExport =
  "export"
  [ ExportId ]
  [ "..." ]
  {
    "extends"  ExportId { "," ExportId }
  | "provides" ( ExportSignature { "," ExportSignature } | "*" )
  | "reveals"  ( ExportSignature { "," ExportSignature } | "*" )
  }

ExportSignature = TypeNameOrCtorSuffix [ "." TypeNameOrCtorSuffix ]
```

17.2.3. Types

(discussion)

```
Type = DomainType_ | ArrowType_

DomainType_ =
  ( BoolType_ | CharType_ | IntType_ | RealType_
  | OrdinalType_ | BitVectorType_ | ObjectType_
  | FiniteSetType_ | InfiniteSetType_
  | MultisetType_
  | FiniteMapType_ | InfiniteMapType_
  | SequenceType_
  | NatType_
  | StringType_
  | ArrayType_
  | TupleType
  | NamedType
  )

NamedType = NameSegmentForTypeName { "." NameSegmentForTypeName }

NameSegmentForTypeName = Ident [ GenericInstantiation ]
```

17.2.3.1. Basic types (discussion)

```
BoolType_ = "bool"
IntType_ = "int"
RealType_ = "real"
BitVectorType_ = bvToken
OrdinalType_ = "ORDINAL"
CharType_ = "char"
```

17.2.3.2. Generic instantiation (discussion)

```
GenericInstantiation = "<" Type { "," Type } ">"
```

17.2.3.3. Type parameter (discussion)

```
GenericParameters(allowVariance) =
  "<" [ Variance ] TypeVariableName { TypeParameterCharacteristics }
  { "," [ Variance ] TypeVariableName { TypeParameterCharacteristics } }
  ">"

// The optional Variance indicator is permitted only if allowVariance is true
Variance = ( "*" | "+" | "!" | "-" )

TypeParameterCharacteristics = "(" TPCharOption { "," TPCharOption } ")"

TPCharOption = ( "==" | "0" | "00" | "!" "new" )
```

17.2.3.4. Collection types (discussion)

```
FiniteSetType_ = "set" [ GenericInstantiation ]

InfiniteSetType_ = "iset" [ GenericInstantiation ]

MultisetType_ = "multiset" [ GenericInstantiation ]

SequenceType_ = "seq" [ GenericInstantiation ]

StringType_ = "string"

FiniteMapType_ = "map" [ GenericInstantiation ]

InfiniteMapType_ = "imap" [ GenericInstantiation ]
```

17.2.3.5. Type definitions (discussion)

```
SynonymTypeDecl =
  SynonymTypeDecl_ | OpaqueTypeDecl_ | SubsetTypeDecl_

SynonymTypeName = NoUSIdent

SynonymTypeDecl_ =
  "type" { Attribute } SynonymTypeName
   { TypeParameterCharacteristics }
   [ GenericParameters ]
   "=" Type
```

```
OpaqueTypeDecl_ =
  "type" { Attribute } SynonymTypeName
   { TypeParameterCharacteristics }
   [ GenericParameters ]
   [ TypeMembers ]

TypeMembers =
  "{"
  {
    { DeclModifier }
    ClassMemberDecl(allowConstructors: false,
                    isValueType: true,
                    moduleLevelDecl: false,
                    isWithinAbstractModule: module.IsAbstract)
  }
  "}"

SubsetTypeDecl_ =
  "type"
  { Attribute }
  SynonymTypeName [ GenericParameters ]
  "="
  LocalIdentTypeOptional
  "|"
  Expression(allowLemma: false, allowLambda: true)
  [ "ghost" "witness" Expression(allowLemma: false, allowLambda: true)
  | "witness" Expression((allowLemma: false, allowLambda: true)
  | "witness" "*"
  ]

NatType_ = "nat"

NewtypeDecl = "newtype" { Attribute } NewtypeName "="
  [ ellipsis ]
  ( LocalIdentTypeOptional
    "|"
    Expression(allowLemma: false, allowLambda: true)
    [ "ghost" "witness" Expression(allowLemma: false, allowLambda: true)
    | "witness" Expression((allowLemma: false, allowLambda: true)
    | "witness" "*"
    ]
  | Type
  )
  [ TypeMembers ]
```

17.2.3.6. Class type (discussion)

```
ClassDecl = "class" { Attribute } ClassName [ GenericParameters ]
  ["extends" Type {"," Type} | ellipsis ]
  "{" { { DeclModifier }
        ClassMemberDecl(modifiers,
                        allowConstructors: true,
                        isValueType: false,
                        moduleLevelDecl: false)
      }
  "}"

ClassMemberDecl(modifiers, allowConstructors, isValueType, moduleLevelDecl) =
  ( FieldDecl(isValueType) // allowed iff moduleLevelDecl is false
  | ConstantFieldDecl(moduleLevelDecl)
  | FunctionDecl(isWithinAbstractModule)
  | MethodDecl(modifiers, allowConstructors)
  )
```

17.2.3.7. Trait types (discussion)

```
TraitDecl =
  "trait" { Attribute } ClassName [ GenericParameters ]
  [ "extends" Type { "," Type } | ellipsis ]
  "{"
   { { DeclModifier } ClassMemberDecl(allowConstructors: true,
                                      isValueType: false,
                                      moduleLevelDecl: false,
                                      isWithinAbstractModule: false) }
  "}"
```

17.2.3.8. Object type (discussion)

```
ObjectType_ = "object" | "object?"
```

17.2.3.9. Array types (discussion)

```
ArrayType_ = arrayToken [ GenericInstantiation ]
```

17.2.3.10. Iterator types (discussion)

```
IteratorDecl = "iterator" { Attribute } IteratorName
  ( [ GenericParameters ]
    Formals(allowGhostKeyword: true, allowNewKeyword: false,
                                 allowOlderKeyword: false)
    [ "yields" Formals(allowGhostKeyword: true, allowNewKeyword: false,
                                            allowOlderKeyword: false) ]
  | ellipsis
  )
  IteratorSpec
  [ BlockStmt ]
```

17.2.3.11. Arrow types (discussion)

```
ArrowType_ = ( DomainType_ "~>" Type
             | DomainType_ "-->" Type
             | DomainType_ "->" Type
             )
```

17.2.3.12. Algebraic datatypes (discussion)

```
DatatypeDecl =
  ( "datatype" | "codatatype" )
  { Attribute }
  DatatypeName [ GenericParameters ]
  "="
  [ ellipsis ]
  [ "|" ] DatatypeMemberDecl
  { "|" DatatypeMemberDecl }
  [ TypeMembers ]

DatatypeMemberDecl =
  { Attribute } DatatypeMemberName [ FormalsOptionalIds ]
```

17.2.4. Type member declarations

(discussion)

17.2.4.1. Fields (discussion)

```
FieldDecl(isValueType) =
  "var" { Attribute } FIdentType { "," FIdentType }
```

A `FieldDecl` is not permitted if `isValueType` is true.

17.2.4.2. Constant fields (discussion)

```
ConstantFieldDecl(moduleLevelDecl) =
  "const" { Attribute } CIdentType [ ellipsis ]
   [ ":=" Expression(allowLemma: false, allowLambda:true) ]
```

If `moduleLevelDecl` is true, then the `static` modifier is not permitted (the constant field is static implicitly).

17.2.4.3. Method declarations (discussion)

```
MethodDecl(isGhost, allowConstructors, isWithinAbstractModule) =
  MethodKeyword_ { Attribute } [ MethodFunctionName ]
  ( MethodSignature_(isGhost, isExtreme: true iff this is a least
                                  or greatest lemma declaration)
  | ellipsis
  )
  MethodSpec(isConstructor: true iff this is a constructor declaration)
  [ BlockStmt ]

MethodKeyword_ = ( "method"
                 | "constructor"
                 | "lemma"
                 | "twostate" "lemma"
                 | "least" "lemma"
                 | "greatest" "lemma"
                 )


MethodSignature_(isGhost, isExtreme) =
  [ GenericParameters ]
  [ KType ]    // permitted only if isExtreme == true
  Formals(allowGhostKeyword: !isGhost, allowNewKeyword: isTwostateLemma,
          allowOlderKeyword: false, allowDefault: true)
  [ "returns" Formals(allowGhostKeyword: !isGhost, allowNewKeyword: false,
                      allowOlderKeyword: false, allowDefault: false) ]

KType = "[" ( "nat" | "ORDINAL" ) "]"

Formals(allowGhostKeyword, allowNewKeyword, allowOlderKeyword, allowDefault) =
  "(" [ { Attribute } GIdentType(allowGhostKeyword, allowNewKeyword, allowOlderKeyword,
                   allowNameOnlyKeyword: true, allowDefault)
        { "," { Attribute } GIdentType(allowGhostKeyword, allowNewKeyword, allowOlderKeyword,
                         allowNameOnlyKeyword: true, allowDefault) }
      ]
  ")"
```

If `isWithinAbstractModule` is false, then the method must have a body for the program that contains the declaration to be compiled.

The `KType` may be specified only for least and greatest lemmas.

17.2.4.4. Function declarations (discussion)

```
FunctionDecl(isWithinAbstractModule) =
  ( [ "twostate" ] "function" [ "method" ] { Attribute }
    MethodFunctionName
    FunctionSignatureOrEllipsis_(allowGhostKeyword:
                                          ("method" present),
                                 allowNewKeyword:
                                          "twostate" present)
  | "predicate" [ "method" ] { Attribute }
    MethodFunctionName
    PredicateSignatureOrEllipsis_(allowGhostKeyword:
                                          ("method" present),
                                  allowNewKeyword:
                                          "twostate" present,
                                  allowOlderKeyword: true)
  | ( "least" | "greatest" ) "predicate" { Attribute }
    MethodFunctionName
    PredicateSignatureOrEllipsis_(allowGhostKeyword: false,
                         allowNewKeyword: "twostate" present,
                         allowOlderKeyword: false))
  )
  FunctionSpec
  [ FunctionBody ]

FunctionSignatureOrEllipsis_(allowGhostKeyword) =
  FunctionSignature_(allowGhostKeyword) | ellipsis

FunctionSignature_(allowGhostKeyword, allowNewKeyword) =
  [ GenericParameters ]
  Formals(allowGhostKeyword, allowNewKeyword, allowOlderKeyword: true,
          allowDefault: true)
  ":"
  ( Type
  | "(" GIdentType(allowGhostKeyword: false,
                   allowNewKeyword: false,
                   allowOlderKeyword: false,
                   allowNameOnlyKeyword: false,
                   allowDefault: false)
    ")"
  )

PredicateSignatureOrEllipsis_(allowGhostKeyword, allowNewKeyword,
                              allowOlderKeyword) =
    PredicateSignature_(allowGhostKeyword, allowNewKeyword, allowOlderKeyword)
  | ellipsis

PredicateSignature_(allowGhostKeyword, allowNewKeyword, allowOlderKeyword) =
  [ GenericParameters ]
  [ KType ]
  Formals(allowGhostKeyword, allowNewKeyword, allowOlderKeyword,
          allowDefault: true)
  [
    ":"
    ( Type
    | "(" Ident ":" "bool" ")"
    )
  ]

FunctionBody = "{" Expression(allowLemma: true, allowLambda: true)
               "}" [ "by" "method" BlockStmt ]
```

17.2.5. Specifications

17.2.5.1. Method specifications (discussion)

```
MethodSpec =
  { ModifiesClause(allowLambda: false)
  | RequiresClause(allowLabel: true)
  | EnsuresClause(allowLambda: false)
  | DecreasesClause(allowWildcard: true, allowLambda: false)
  }
```

17.2.5.2. Function specifications (discussion)

```
FunctionSpec =
  { RequiresClause(allowLabel: true)
  | ReadsClause(allowLemma: false, allowLambda: false, allowWild: true)
  | EnsuresClause(allowLambda: false)
  | DecreasesClause(allowWildcard: false, allowLambda: false)
  }
```

17.2.5.3. Lambda function specifications (discussion)

```
LambdaSpec =
  { ReadsClause(allowLemma: true, allowLambda: false, allowWild: true)
  | "requires" Expression(allowLemma: false, allowLambda: false)
  }
```

17.2.5.4. Iterator specifications (discussion)

```
IteratorSpec =
  { ReadsClause(allowLemma: false, allowLambda: false,
                              allowWild: false)
  | ModifiesClause(allowLambda: false)
  | [ "yield" ] RequiresClause(allowLabel: !isYield)
  | [ "yield" ] EnsuresClause(allowLambda: false)
  | DecreasesClause(allowWildcard: false, allowLambda: false)
  }
```

17.2.5.5. Loop specifications (discussion)

```
LoopSpec =
  { InvariantClause_
  | DecreasesClause(allowWildcard: true, allowLambda: true)
  | ModifiesClause(allowLambda: true)
  }
```

17.2.5.6. Requires clauses (discussion)

```
RequiresClause(allowLabel) =
  "requires" { Attribute }
  [ LabelName ":" ]  // Label allowed only if allowLabel is true
  Expression(allowLemma: false, allowLambda: false)
```

17.2.5.7. Ensures clauses (discussion)

```
EnsuresClause(allowLambda) =
  "ensures" { Attribute } Expression(allowLemma: false, allowLambda)
```

17.2.5.8. Decreases clauses (discussion)

```
DecreasesClause(allowWildcard, allowLambda) =
  "decreases" { Attribute } DecreasesList(allowWildcard, allowLambda)

DecreasesList(allowWildcard, allowLambda) =
  PossiblyWildExpression(allowLambda, allowWildcard)
  { "," PossiblyWildExpression(allowLambda, allowWildcard) }

PossiblyWildExpression(allowLambda, allowWild) =
  ( "*"  // if allowWild is false, using '*' provokes an error
  | Expression(allowLemma: false, allowLambda)
  )
```

17.2.5.9. Modifies clauses (discussion)

```
ModifiesClause(allowLambda) =
  "modifies" { Attribute }
  FrameExpression(allowLemma: false, allowLambda)
  { "," FrameExpression(allowLemma: false, allowLambda) }
```

17.2.5.10. Invariant clauses (discussion)

```
InvariantClause_ =
  "invariant" { Attribute }
  Expression(allowLemma: false, allowLambda: true)
```

17.2.5.11. Reads clauses (discussion)

```
ReadsClause(allowLemma, allowLambda, allowWild) =
  "reads" { Attribute }
  PossiblyWildFrameExpression(allowLemma, allowLambda, allowWild)
  { "," PossiblyWildFrameExpression(allowLemma, allowLambda, allowWild) }
```

17.2.5.12. Frame expressions (discussion)

```
FrameExpression(allowLemma, allowLambda) =
  ( Expression(allowLemma, allowLambda) [ FrameField ]
  | FrameField
  )

FrameField = "`" IdentOrDigits

PossiblyWildFrameExpression(allowLemma, allowLambda, allowWild) =
  ( "*"  // error if !allowWild and '*'
  | FrameExpression(allowLemma, allowLambda)
  )
```

17.2.6. Statements

17.2.6.1. Labeled statement (discussion)

```
Stmt = { "label" LabelName ":" } NonLabeledStmt
```

17.2.6.2. Non-Labeled statement (discussion)

```
NonLabeledStmt =
  ( AssertStmt | AssumeStmt | BlockStmt | BreakStmt
  | CalcStmt | ExpectStmt | ForallStmt | IfStmt
  | MatchStmt | ModifyStmt
  | PrintStmt | ReturnStmt | RevealStmt
  | UpdateStmt | UpdateFailureStmt
  | VarDeclStatement | WhileStmt | ForLoopStmt | YieldStmt
  )
```

17.2.6.3. Break and continue statements (discussion)

```
BreakStmt =
  ( "break" LabelName ";"
  | "continue" LabelName ";"
  | { "break" } "break" ";"
  | { "break" } "continue" ";"
  )
```

17.2.6.4. Block statement (discussion)

```
BlockStmt = "{" { Stmt } "}"
```

17.2.6.5. Return statement (discussion)

```
ReturnStmt = "return" [ Rhs { "," Rhs } ] ";"
```

17.2.6.6. Yield statement (discussion)

```
YieldStmt = "yield" [ Rhs { "," Rhs } ] ";"
```

17.2.6.7. Update and call statement (discussion)

```
UpdateStmt =
  Lhs
  ( {Attribute} ";"
  |
    { "," Lhs }
    ( ":=" Rhs { "," Rhs }
    | ":|" [ "assume" ]
               Expression(allowLemma: false, allowLambda: true)
    )
    ";"
  )
```

17.2.6.8. Update with failure statement (discussion)

```
UpdateFailureStmt  =
  [ Lhs { "," Lhs } ]
  ":-"
  [ "expect"  | "assert" | "assume" ]
  Expression(allowLemma: false, allowLambda: false)
  { "," Rhs }
  ";"
```

17.2.6.9. Variable declaration statement (discussion)

```
VarDeclStatement =
  [ "ghost" ] "var" { Attribute }
  (
    LocalIdentTypeOptional
    { "," { Attribute } LocalIdentTypeOptional }
    [ ":="
      Rhs { "," Rhs }
    | ":-"
      [ "expect" | "assert" | "assume" ]
      Expression(allowLemma: false, allowLambda: false)
      { "," Rhs }
    | { Attribute }
      ":|"
      [ "assume" ] Expression(allowLemma: false, allowLambda: true)
    ]
  |
    CasePatternLocal
    ( ":=" | { Attribute } ":|" )
    Expression(allowLemma: false, allowLambda: true)
  )
  ";"

CasePatternLocal =
  ( [ Ident ] "(" CasePatternLocal { "," CasePatternLocal } ")"
  | LocalIdentTypeOptional
  )
```

17.2.6.10. Guards (discussion)

```
Guard = ( "*"
        | "(" "*" ")"
        | Expression(allowLemma: true, allowLambda: true)
        )
```

17.2.6.11. Binding guards (discussion)

```
BindingGuard(allowLambda) =
  IdentTypeOptional { "," IdentTypeOptional }
  { Attribute }
  ":|"
  Expression(allowLemma: true, allowLambda)
```

17.2.6.12. If statement (discussion)

```
IfStmt = "if"
  ( AlternativeBlock(allowBindingGuards: true)
  |
    ( BindingGuard(allowLambda: true)
    | Guard
    )
    BlockStmt [ "else" ( IfStmt | BlockStmt ) ]
  )

AlternativeBlock(allowBindingGuards) =
  ( { AlternativeBlockCase(allowBindingGuards) }
  | "{" { AlternativeBlockCase(allowBindingGuards) } "}"
  )

AlternativeBlockCase(allowBindingGuards) =
  { "case"
    (
    BindingGuard(allowLambda: false) //permitted iff allowBindingGuards == true
    | Expression(allowLemma: true, allowLambda: false)
    ) "=>" { Stmt }
  }
```

17.2.6.13. While Statement (discussion)

```
WhileStmt =
  "while"
  ( LoopSpec
    AlternativeBlock(allowBindingGuards: false)
  | Guard
    LoopSpec
    ( BlockStmt
    |           // no body
    )
  )
```

17.2.6.14. For statement (discussion)

```
ForLoopStmt =
  "for" IdentTypeOptional ":="
  Expression(allowLemma: false, allowLambda: false)
  ( "to" | "downto" )
  ( "*" | Expression(allowLemma: false, allowLambda: false)
  )
  LoopSpec
  ( BlockStmt
  |           // no body
  )
```

17.2.6.15. Match statement (discussion)

```
MatchStmt =
  "match"
  Expression(allowLemma: true, allowLambda: true)
  ( "{" { CaseStmt } "}"
  | { CaseStmt }
  )

CaseStmt = "case" ExtendedPattern "=>" { Stmt }
```

17.2.6.16. Assert statement (discussion)

```
AssertStmt =
  "assert"
  { Attribute }
  [ LabelName ":" ]
  Expression(allowLemma: false, allowLambda: true)
  ( ";"
  | "by" BlockStmt
  )
```

17.2.6.17. Assume statement (discussion)

```
AssumeStmt =
  "assume"
  { Attribute }
  Expression(allowLemma: false, allowLambda: true)
  ";"
```

17.2.6.18. Expect statement (discussion)

```
ExpectStmt =
  "expect"
  { Attribute }
  Expression(allowLemma: false, allowLambda: true)
  [ "," Expression(allowLemma: false, allowLambda: true) ]
  ";"
```

17.2.6.19. Print statement (discussion)

```
PrintStmt =
  "print"
  Expression(allowLemma: false, allowLambda: true)
  { "," Expression(allowLemma: false, allowLambda: true) }
  ";"
```

17.2.6.20. Reveal statement (discussion)

```
RevealStmt =
  "reveal"
  Expression(allowLemma: false, allowLambda: true)
  { "," Expression(allowLemma: false, allowLambda: true) }
  ";"
```

17.2.6.21. Forall statement (discussion)

```
ForallStmt =
  "forall"
  ( "(" [ QuantifierDomain ] ")"
  | [ QuantifierDomain ]
  )
  { EnsuresClause(allowLambda: true) }
  [ BlockStmt ]
```

17.2.6.22. Modify statement (discussion)

```
ModifyStmt =
  "modify"
  { Attribute }
  FrameExpression(allowLemma: false, allowLambda: true)
  { "," FrameExpression(allowLemma: false, allowLambda: true) }
  ";"
```

17.2.6.23. Calc statement (discussion)

```
CalcStmt = "calc" { Attribute } [ CalcOp ] "{" CalcBody_ "}"

CalcBody_ = { CalcLine_ [ CalcOp ] Hints_ }

CalcLine_ = Expression(allowLemma: false, allowLambda: true) ";"

Hints_ = { ( BlockStmt | CalcStmt ) }

CalcOp =
  ( "==" [ "#" "["
           Expression(allowLemma: true, allowLambda: true) "]" ]
  | "<" | ">"
  | "!=" | "<=" | ">="
  | "<==>" | "==>" | "<=="
  )
```

17.2.7. Expressions

17.2.7.1. Top-level expression (discussion)

```
Expression(allowLemma, allowLambda, allowBitwiseOps = true) =
  EquivExpression(allowLemma, allowLambda, allowBitwiseOps)
  [ ";" Expression(allowLemma, allowLambda, allowBitwiseOps) ]
```

The "allowLemma" argument says whether or not the expression to be parsed is allowed to have the form `S;E` where `S` is a call to a lemma. "allowLemma" should be passed in as "false" whenever the expression to be parsed sits in a context that itself is terminated by a semi-colon.

The "allowLambda" says whether or not the expression to be parsed is allowed to be a lambda expression. More precisely, an identifier or parenthesized, comma-delimited list of identifiers is allowed to continue as a lambda expression (that is, continue with a **`reads`**, **`requires`**, or `=>`) only if "allowLambda" is true. This affects function/method/iterator specifications, if/while statements with guarded alternatives, and expressions in the specification of a lambda expression itself.

17.2.7.2. Equivalence expression (discussion)

```
EquivExpression(allowLemma, allowLambda, allowBitwiseOps) =
  ImpliesExpliesExpression(allowLemma, allowLambda, allowBitwiseOps)
  { "<==>" ImpliesExpliesExpression(allowLemma, allowLambda, allowBitwiseOps) }
```

17.2.7.3. Implies expression (discussion)

```
ImpliesExpliesExpression(allowLemma, allowLambda, allowBitwiseOps) =
  LogicalExpression(allowLemma, allowLambda)
  [ (  "==>" ImpliesExpression(allowLemma, allowLambda, allowBitwiseOps)
    | "<==" LogicalExpression(allowLemma, allowLambda, allowBitwiseOps)
            { "<==" LogicalExpression(allowLemma, allowLambda, allowBitwiseOps) }
    )
  ]

ImpliesExpression(allowLemma, allowLambda, allowBitwiseOps) =
  LogicalExpression(allowLemma, allowLambda, allowBitwiseOps)
  [  "==>" ImpliesExpression(allowLemma, allowLambda, allowBitwiseOps) ]
```

17.2.7.4. Logical expression (discussion)

```
LogicalExpression(allowLemma, allowLambda, allowBitwiseOps) =
  [ "&&" | "||" ]
  RelationalExpression(allowLemma, allowLambda, allowBitwiseOps)
  { ( "&&" | "||" )
    RelationalExpression(allowLemma, allowLambda, allowBitwiseOps)
  }
```

17.2.7.5. Relational expression (discussion)

```
RelationalExpression(allowLemma, allowLambda, allowBitwiseOps) =
  ShiftTerm(allowLemma, allowLambda, allowBitwiseOps)
  { RelOp ShiftTerm(allowLemma, allowLambda, allowBitwiseOps) }

RelOp =
  ( "=="
    [ "#" "[" Expression(allowLemma: true, allowLambda: true) "]" ]
  | "!="
    [ "#" "[" Expression(allowLemma: true, allowLambda: true) "]" ]
  | "<" | ">" | "<=" | ">="
  | "in"
  | "!in"
  | "!!"
  )
```

17.2.7.6. Bit-shift expression (discussion)

```
ShiftTerm(allowLemma, allowLambda, allowBitwiseOps) =
  Term(allowLemma, allowLambda, allowBitwiseOps)
  { ShiftOp Term(allowLemma, allowLambda, allowBitwiseOps) }

ShiftOp = ( "<<" | ">>" )
```

17.2.7.7. Term (addition operations) (discussion)

```
Term(allowLemma, allowLambda, allowBitwiseOps) =
  Factor(allowLemma, allowLambda, allowBitwiseOps)
  { AddOp Factor(allowLemma, allowLambda, allowBitwiseOps) }

AddOp = ( "+" | "-" )
```

17.2.7.8. Factor (multiplication operations) (discussion)

```
Factor(allowLemma, allowLambda, allowBitwiseOps) =
  BitvectorFactor(allowLemma, allowLambda, allowBitwiseOps)
  { MulOp BitvectorFactor(allowLemma, allowLambda, allowBitwiseOps) }

MulOp = ( "*" | "/" | "%" )
```

17.2.7.9. Bit-vector expression (discussion)

```
BitvectorFactor(allowLemma, allowLambda, allowBitwiseOps) =
  AsExpression(allowLemma, allowLambda, allowBitwiseOps)
  { BVOp AsExpression(allowLemma, allowLambda, allowBitwiseOps) }
```

```
BVOp = ( "|" | "&" | "^" )
```

If `allowBitwiseOps` is false, it is an error to have a bitvector operation.

17.2.7.10. As/Is expression (discussion)

```
AsExpression(allowLemma, allowLambda, allowBitwiseOps) =
  UnaryExpression(allowLemma, allowLambda, allowBitwiseOps)
  { ( "as" | "is" ) Type }
```

17.2.7.11. Unary expression (discussion)

```
UnaryExpression(allowLemma, allowLambda, allowBitwiseOps) =
  ( "-" UnaryExpression(allowLemma, allowLambda, allowBitwiseOps)
  | "!" UnaryExpression(allowLemma, allowLambda, allowBitwiseOps)
  | PrimaryExpression(allowLemma, allowLambda, allowBitwiseOps)
  )
```

17.2.7.12. Primary expression (discussion)

```
PrimaryExpression(allowLemma, allowLambda, allowBitwiseOps) =
  ( NameSegment { Suffix }
  | LambdaExpression(allowLemma, allowBitwiseOps)
  | MapDisplayExpr { Suffix }
  | SeqDisplayExpr { Suffix }
  | SetDisplayExpr { Suffix }
  | EndlessExpression(allowLemma, allowLambda, allowBitwiseOps)
  | ConstAtomExpression { Suffix }
  )
```

17.2.7.13. Lambda expression (discussion)

```
LambdaExpression(allowLemma, allowBitwiseOps) =
  ( WildIdent
  | "(" [ IdentTypeOptional { "," IdentTypeOptional } ] ")"
  )
  LambdaSpec
  "=>"
  Expression(allowLemma, allowLambda: true, allowBitwiseOps)
```

17.2.7.14. Left-hand-side expression (discussion) {

```
Lhs =
  ( NameSegment { Suffix }
  | ConstAtomExpression Suffix { Suffix }
  )
```

17.2.7.15. Right-hand-side expression (discussion)

```
Rhs =
    ArrayAllocation
  | ObjectAllocation_
  | Expression(allowLemma: false, allowLambda: true, allowBitwiseOps: true)
  | HavocRhs_
  )
  { Attribute }
```

17.2.7.16. Array allocation right-hand-side expression (discussion)

```
ArrayAllocation_ =
  "new" [ Type ] "[" [ Expressions ] "]"
  [ "(" Expression(allowLemma: true, allowLambda: true) ")"
  | "[" [ Expressions ] "]"
  ]
```

17.2.7.17. Object allocation right-hand-side expression (discussion)

```
ObjectAllocation_ = "new" Type [ "." TypeNameOrCtorSuffix ]
                               [ "(" [ Bindings ] ")" ]
```

17.2.7.18. Havoc right-hand-side expression (discussion)

```
HavocRhs_ = "*"
```

17.2.7.19. Atomic expressions (discussion)

```
ConstAtomExpression =
  ( LiteralExpression
  | ThisExpression_
  | FreshExpression_
  | AllocatedExpression_
  | UnchangedExpression_
  | OldExpression_
  | CardinalityExpression_
  | ParensExpression
  )
```

17.2.7.20. Literal expressions (discussion)

```
LiteralExpression =
 ( "false" | "true" | "null" | Nat | Dec |
   charToken | stringToken )

Nat = ( digits | hexdigits )
```

```
Dec = decimaldigits
```

17.2.7.21. This expression (discussion)

```
ThisExpression_ = "this"
```

17.2.7.22. Old and Old@ Expressions (discussion)

```
OldExpression_ =
  "old" [ "@" LabelName ]
  "(" Expression(allowLemma: true, allowLambda: true) ")"
```

17.2.7.23. Fresh Expressions (discussion)

```
FreshExpression_ =
  "fresh" [ "@" LabelName ]
  "(" Expression(allowLemma: true, allowLambda: true) ")"
```

17.2.7.24. Allocated Expressions (discussion)

```
AllocatedExpression_ =
  "allocated" "(" Expression(allowLemma: true, allowLambda: true) ")"
```

17.2.7.25. Unchanged Expressions (discussion)

```
UnchangedExpression_ =
  "unchanged" [ "@" LabelName ]
  "(" FrameExpression(allowLemma: true, allowLambda: true)
      { "," FrameExpression(allowLemma: true, allowLambda: true) }
  ")"
```

17.2.7.26. Cardinality Expressions (discussion)

```
CardinalityExpression_ =
  "|" Expression(allowLemma: true, allowLambda: true) "|"
```

17.2.7.27. Parenthesized Expression (discussion)

```
ParensExpression =
  "(" [ TupleArgs ] ")"

TupleArgs =
  [ "ghost" ]
  ActualBinding(isGhost) // argument is true iff the ghost modifier is present
  { ","
    [ "ghost" ]
    ActualBinding(isGhost) // argument is true iff the ghost modifier is present
  }
```

17.2.7.28. Sequence Display Expression (discussion)

```
SeqDisplayExpr =
  ( "[" [ Expressions ] "]"
  | "seq" [ GenericInstantiation ]
    "(" Expression(allowLemma: true, allowLambda: true)
    "," Expression(allowLemma: true, allowLambda: true)
    ")"
  )
```

17.2.7.29. Set Display Expression (discussion)

```
SetDisplayExpr =
  ( [ "iset" | "multiset" ] "{" [ Expressions ] "}"
  | "multiset" "(" Expression(allowLemma: true,
                              allowLambda: true) ")"
  )
```

17.2.7.30. Map Display Expression (discussion)

```
MapDisplayExpr =
  ("map" | "imap" ) "[" [ MapLiteralExpressions ] "]"

MapLiteralExpressions =
  Expression(allowLemma: true, allowLambda: true)
  ":="
  Expression(allowLemma: true, allowLambda: true)
  { ","
    Expression(allowLemma: true, allowLambda: true)
    ":="
    Expression(allowLemma: true, allowLambda: true)
  }
```

17.2.7.31. Endless Expression (discussion)

```
EndlessExpression(allowLemma, allowLambda, allowBitwiseOps) =
  ( IfExpression(allowLemma, allowLambda, allowBitwiseOps)
  | MatchExpression(allowLemma, allowLambda, allowBitwiseOps)
  | QuantifierExpression(allowLemma, allowLambda)
  | SetComprehensionExpr(allowLemma, allowLambda, allowBitwiseOps)
  | StmtInExpr
    Expression(allowLemma, allowLambda, allowBitwiseOps)
  | LetExpression(allowLemma, allowLambda, allowBitwiseOps)
  | MapComprehensionExpr(allowLemma, allowLambda, allowBitwiseOps)
  )
```

17.2.7.32. If expression (discussion)

```
IfExpression(allowLemma, allowLambda, allowBitwiseOps) =
    "if" ( BindingGuard(allowLambda: true)
         | Expression(allowLemma: true, allowLambda: true, allowBitwiseOps: true)
         )
    "then" Expression(allowLemma: true, allowLambda: true, allowBitwiseOps: true)
    "else" Expression(allowLemma, allowLambda, allowBitwiseOps)
```

17.2.7.33. Match Expression (discussion)

```
MatchExpression(allowLemma, allowLambda, allowBitwiseOps) =
  "match"
  Expression(allowLemma, allowLambda, allowBitwiseOps)
  ( "{" { CaseExpression(allowLemma: true, allowLambda, allowBitwiseOps: true) } "}"
  | { CaseExpression(allowLemma, allowLambda, allowBitwiseOps) }
  )

CaseExpression(allowLemma, allowLambda, allowBitwiseOps) =
  "case" { Attribute } ExtendedPattern "=>" Expression(allowLemma, allowLambda, allowBitwiseOps)
```

17.2.7.34. Case and Extended Patterns (discussion)

```
CasePattern =
  ( IdentTypeOptional
  | [Ident] "(" [ CasePattern { "," CasePattern } ] ")"
  )

SingleExtendedPattern =
  ( PossiblyNegatedLiteralExpression
  | IdentTypeOptional
  | [ Ident ] "(" [ SingleExtendedPattern { "," SingleExtendedPattern } ] ")"
  )

ExtendedPattern =
  ( [ "|" ] SingleExtendedPattern { "|" SingleExtendedPattern } )

PossiblyNegatedLiteralExpression =
  ( "-" ( Nat | Dec )
  | LiteralExpression
  )
```

17.2.7.35. Quantifier expression (discussion)

```
QuantifierExpression(allowLemma, allowLambda) =
  ( "forall" | "exists" ) QuantifierDomain "::"
  Expression(allowLemma, allowLambda)
```

17.2.7.36. Set Comprehension Expressions (discussion)

```
SetComprehensionExpr(allowLemma, allowLambda) =
  [ "set" | "iset" ]
  QuantifierDomain(allowLemma, allowLambda)
  [ "::" Expression(allowLemma, allowLambda) ]
```

17.2.7.37. Map Comprehension Expression (discussion)

```
MapComprehensionExpr(allowLemma, allowLambda) =
  ( "map" | "imap" )
  QuantifierDomain(allowLemma, allowLambda)
  "::"
  Expression(allowLemma, allowLambda)
  [ ":=" Expression(allowLemma, allowLambda) ]
```

17.2.7.38. Statements in an Expression (discussion)

```
StmtInExpr = ( AssertStmt | AssumeStmt | ExpectStmt
             | RevealStmt | CalcStmt
             )
```

17.2.7.39. Let and Let or Fail Expression (discussion)

```
LetExpression(allowLemma, allowLambda) =
  (
    [ "ghost" ] "var" CasePattern { "," CasePattern }
    ( ":=" | ":-" | { Attribute } ":|" )
    Expression(allowLemma: false, allowLambda: true)
    { "," Expression(allowLemma: false, allowLambda: true) }
  |
    ":-"
    Expression(allowLemma: false, allowLambda: true)
  )
  ";"
  Expression(allowLemma, allowLambda)
```

17.2.7.40. Name Segment (discussion)

```
NameSegment = Ident [ GenericInstantiation | HashCall ]
```

17.2.7.41. Hash Call (discussion)

```
HashCall = "#" [ GenericInstantiation ]
  "[" Expression(allowLemma: true, allowLambda: true) "]"
  "(" [ Bindings ] ")"
```

17.2.7.42. Suffix (discussion)

```
Suffix =
  ( AugmentedDotSuffix_
  | DatatypeUpdateSuffix_
  | SubsequenceSuffix_
  | SlicesByLengthSuffix_
  | SequenceUpdateSuffix_
  | SelectionSuffix_
  | ArgumentListSuffix_
  )
```

17.2.7.43. Augmented Dot Suffix (discussion)

```
AugmentedDotSuffix_ = "." DotSuffix
                      [ GenericInstantiation | HashCall ]
```

17.2.7.44. Datatype Update Suffix (discussion)

```
DatatypeUpdateSuffix_ =
  "." "(" MemberBindingUpdate { "," MemberBindingUpdate } ")"

MemberBindingUpdate =
  ( ident | digits )
  ":=" Expression(allowLemma: true, allowLambda: true)
```

17.2.7.45. Subsequence Suffix (discussion)

```
SubsequenceSuffix_ =
  "[" [ Expression(allowLemma: true, allowLambda: true) ]
      ".." [ Expression(allowLemma: true, allowLambda: true) ]
  "]"
```

17.2.7.46. Subsequence Slices Suffix (discussion)

```
SlicesByLengthSuffix_ =
  "[" Expression(allowLemma: true, allowLambda: true) ":"
      [
        Expression(allowLemma: true, allowLambda: true)
        { ":" Expression(allowLemma: true, allowLambda: true) }
        [ ":" ]
      ]
  "]"
```

17.2.7.47. Sequence Update Suffix (discussion)

```
SequenceUpdateSuffix_ =
  "[" Expression(allowLemma: true, allowLambda: true)
      ":=" Expression(allowLemma: true, allowLambda: true)
  "]"
```

17.2.7.48. Selection Suffix (discussion)

```
SelectionSuffix_ =
  "[" Expression(allowLemma: true, allowLambda: true)
      { "," Expression(allowLemma: true, allowLambda: true) }
  "]"
```

17.2.7.49. Argument List Suffix (discussion)

```
ArgumentListSuffix_ = "(" [ Expressions ] ")"
```

17.2.7.50. Expression Lists (discussion)

```
Expressions =
  Expression(allowLemma: true, allowLambda: true)
  { "," Expression(allowLemma: true, allowLambda: true) }
```

17.2.7.51. Parameter Bindings (discussion)

```
ActualBindings =
  ActualBinding
  { "," ActualBinding }

ActualBinding(isGhost = false) =
  [ NoUSIdentOrDigits ":=" ]
  Expression(allowLemma: true, allowLambda: true)
```

```

QuantifierDomain(allowLemma, allowLambda) =
  QuantifierVarDecl(allowLemma, allowLambda)
  { "," QuantifierVarDecl(allowLemma, allowLambda) }

QuantifierVarDecl(allowLemma, allowLambda) =
  IdentTypeOptional
  [ "<-" Expression(allowLemma, allowLambda) ]
  { Attribute }
  [ | Expression(allowLemma, allowLambda) ]
```

17.2.7.52. Quantifier domains

```

Ident = ident

DotSuffix = ( ident | digits | "requires" | "reads" )

NoUSIdent = ident - "_" { idchar }

WildIdent = NoUSIdent | "_"

IdentOrDigits = Ident | digits
NoUSIdentOrDigits = NoUSIdent | digits
ModuleName = NoUSIdent
ClassName = NoUSIdent    // also traits
DatatypeName = NoUSIdent
DatatypeMemberName = NoUSIdentOrDigits
NewtypeName = NoUSIdent
SynonymTypeName = NoUSIdent
IteratorName = NoUSIdent
TypeVariableName = NoUSIdent
MethodFunctionName = NoUSIdentOrDigits
LabelName = NoUSIdentOrDigits
AttributeName = NoUSIdent
ExportId = NoUSIdentOrDigits
TypeNameOrCtorSuffix = NoUSIdentOrDigits

ModuleQualifiedName = ModuleName { "." ModuleName }

IdentType = WildIdent ":" Type

FIdentType = NoUSIdentOrDigits ":" Type

CIdentType = NoUSIdentOrDigits [ ":" Type ]

GIdentType(allowGhostKeyword, allowNewKeyword, allowOlderKeyword, allowNameOnlyKeyword, allowDefaul
  { "ghost" | "new" | "nameonly" | "older" } IdentType
  [ ":=" Expression(allowLemma: true, allowLambda: true) ]

LocalIdentTypeOptional = WildIdent [ ":" Type ]
```

```
IdentTypeOptional = WildIdent [ ":" Type ]

TypeIdentOptional =
  { Attribute }
  { "ghost" | "nameonly" } [ NoUSIdentOrDigits ":" ] Type
  [ ":=" Expression(allowLemma: true, allowLambda: true) ]

FormalsOptionalIds = "(" [ TypeIdentOptional
                           { "," TypeIdentOptional } ] ")"
```

17.2.7.53. Basic name and type combinations

18. Testing syntax rendering

Sample math B: $a \to b$ or

$$a \to \pi$$

or (a) or [a →]

Colors

```
integer literal:  10
hex literal:      0xDEAD
real literal:     1.1
boolean literal:  true false
char literal:     'c'
string literal:   "abc"
verbatim string:  @"abc"
ident:            ijk
type:             int
generic type:     map<int,T>
operator:         <=
punctuation:      { }
keyword:          while
spec:             requires
comment:          // comment
attribute         {: name }
error:            $
```

Syntax color tests:

```
integer: 0 00 20 01 0_1
float:   .0 1.0 1. 0_1.1_0
bad:    0_
hex:    0x10_abcdefABCDEF
string:   "string \n \t \r \0" "a\"b" "'" "\'" ""
string:   "!@#$%^&*()_-+={}[]|:;\\<>,.?/~`"
string:   "\u1234 "
string:   "     " : "\0\n\r\t\'\"\\"
notstring: "abcde
notstring: "\u123  " : "x\Zz" : "x\ux"
vstring:  @"" @"a" @"""" @"'\" @"\u"
vstring:  @"xx""y y""zz "
vstring:  @" " @"       "
vstring:  @"x
x"
bad:       @!
char:    'a' '\n' '\'' '"' '\"' ' ' '\\'
```

```
char:    '\0' '\r' '\t'  '\u1234'
badchar:  $ `
ids:  '\u123'   '\Z'  '\u'  '\u2222Z'
ids:  '\u123ZZZ'     '\u2222Z'
ids: 'a : a' : 'ab' :  'a'b' : 'a''b'
ids:  a_b _ab ab? _0
id-label:  a@label
literal:  true false null
op:      - ! ~ x  -!~x
op:      a + b - c * d / e % f a+b-c*d/e%f
op:      <= >= < > == != b&&c || ==> <==> <==
op:      !=# !! in !in
op:      !in   !iné
not op:  !inx
punc:    . , :: | :| := ( ) [ ] { }
types:   int real string char bool nat ORDINAL
types:   object object?
types:   bv1 bv10 bv0
types:   array array2 array20 array10
types:   array? array2? array20? array10?
ids:     array1 array0 array02 bv02 bv_1
ids:     intx natx int0 int_ int? bv1_ bv1x array2x
types:   seq<int>  set < bool >
types:   map<bool,bool>  imap < bool , bool >
types:   seq<Node> seq< Node >
types:   seq<set< real> >
types:   map<set<int>,seq<bool>>
types:   G<A,int> G<G<A>,G<bool>>
types:   seq map imap set iset multiset
ids:     seqx mapx
no arg:  seq < >  seq < , >  seq <bool , , bool >  seq<bool,>
keywords: if while assert assume
spec:    requires  reads modifies
attribute:  {: MyAttribute "asd", 34 }
attribute:  {: MyAttribute }
comment:  // comment
comment:  /* comment */ after
comment:  // comment /* asd */ dfg
comment:  /* comment /* embedded */ tail */ after
comment:  /* comment // embedded */ after
comment: /* comment
   /* inner comment
   */
   outer comment
   */ after
```

```
    more after
```

19. References

Barnett, Mike, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs." In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, edited by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, 4111:364–87. Lncs. Springer.

Bertot, Yves, and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer.

Bove, Ana, Peter Dybjer, and Ulf Norell. 2009. "A Brief Overview of Agda — a Functional Language with Dependent Types." In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, edited by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, 5674:73–78. Lecture Notes in Computer Science. Springer.

Camilleri, Juanito, and Tom Melham. 1992. "Reasoning with Inductively Defined Relations in the HOL Theorem Prover." 265. University of Cambridge Computer Laboratory.

Harrison, John. 1995. "Inductive Definitions: Automation and Application." In *TPHOLs 1995*, edited by E. Thomas Schubert, Phillip J. Windley, and Jim Alves-Foss, 971:200–213. LNCS. Springer.

Hoare, C. A. R. 1969. "An Axiomatic Basis for Computer Programming." *Cacm* 12 (10): 576–80, 583.

Jacobs, Bart, and Jan Rutten. 2011. "An Introduction to (Co)algebra and (Co)induction." In *Advanced Topics in Bisimulation and Coinduction*, edited by Davide Sangiorgi and Jan Rutten, 38–99. Cambridge Tracts in Theoretical Computer Science 52. Cambridge University Press.

Kassios, Ioannis T. 2006. "Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions." In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, edited by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, 4085:268–83. Lncs. Springer.

Kaufmann, Matt, Panagiotis Manolios, and J Strother Moore. 2000. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers.

Koenig, Jason, and K. Rustan M. Leino. 2012. "Getting Started with Dafny: A Guide." In *Software Safety and Security: Tools for Analysis and Verification*, edited by Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann, 33:152–81. NATO Science for Peace and Security Series d: Information and Communication Security. IOS Press.

Kozen, Dexter, and Alexandra Silva. 2012. "Practical Coinduction." http://hdl.handle.net/1813/30510. Comp.; Inf. Science, Cornell Univ.

Krauss, Alexander. 2009. "Automating Recursive Definitions and Termination Proofs in Higher-Order Logic." PhD thesis, Technische Universität München.

Leino, K. Rustan M. 2008a. "Main Microsoft Research Dafny Web Page."

———. 2008b. "This Is Boogie 2." Manuscript KRML 178.

———. 2009. "Dynamic-Frame Specifications in Dafny." JML seminar, Dagstuhl, Germany.

———. 2010. "Dafny: An Automatic Program Verifier for Functional Correctness." In *LPAR-16*, edited by Edmund M. Clarke and Andrei Voronkov, 6355:348–70. Lncs. Springer.

———. 2012. "Automating Induction with an SMT Solver." In *Verification, Model Checking, and Abstract Interpretation — 13th International Conference, VMCAI 2012*, edited by Viktor

Kuncak and Andrey Rybalchenko, 7148:315–31. Lncs. Springer.

Leino, K. Rustan M., and Michal Moskal. 2014a. "Co-Induction Simply: Automatic Co-Inductive Proofs in a Program Verifier." Manuscript KRML 230.

Leino, K. Rustan M., and Michał Moskal. 2014b. "Co-Induction Simply — Automatic Co-Inductive Proofs in a Program Verifier." In *FM 2014*, 8442:382–98. LNCS. Springer.

Leino, K. Rustan M., and Nadia Polikarpova. 2013. "Verified Calculations." Manuscript KRML 231.

Leino, K. Rustan M., and Philipp Rümmer. 2010. "A Polymorphic Intermediate Verification Language: Design and Logical Encoding." In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010*, edited by Javier Esparza and Rupak Majumdar, 6015:312–27. Lncs. Springer.

Leino, K. Rustan M., and Valentin Wüstholz. 2015. "Fine-Grained Caching of Verification Results." In *Computer Aided Verification (CAV)*, edited by Daniel Kroening and Corina S. Pasareanu, 9206:380–97. Lecture Notes in Computer Science. Springer.

Leroy, Xavier, and Hervé Grall. 2009. "Coinductive Big-Step Operational Semantics." *Information and Computation* 207 (2): 284–304.

Manolios, Panagiotis, and J Strother Moore. 2003. "Partial Functions in ACL2." *Journal of Automated Reasoning* 31 (2): 107–27.

Milner, Robin. 1982. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc.

Mössenböck, Hanspeter, Markus Löberbauer, and Albrecht Wöß. 2013. "The Compiler Generator Coco/r." Open source from University of Linz.

Moura, Leonardo de, and Nikolaj Bjørner. 2008. "Z3: An Efficient SMT Solver." In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, edited by C. R. Ramakrishnan and Jakob Rehof, 4963:337–40. Lncs. Springer.

Nipkow, Tobias, and Gerwin Klein. 2014. *Concrete Semantics with Isabelle/HOL*. Springer.

Paulin-Mohring, Christine. 1993. "Inductive Definitions in the System Coq — Rules and Properties." In *TLCA '93*, 664:328–45. LNCS. Springer.

Paulson, Lawrence C. 1994. "A Fixedpoint Approach to Implementing (Co)inductive Definitions." In *CADE-12*, edited by Alan Bundy, 814:148–61. LNCS. Springer.

Pierce, Benjamin C., Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2015. *Software Foundations*. Version 3.2. http://www.cis.upenn.edu/~bcpierce/sf.

Smans, Jan, Bart Jacobs, and Frank Piessens. 2009. "Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic." In *ECOOP 2009 — Object-Oriented Programming, 23rd European Conference*, edited by Sophia Drossopoulou, 5653:148–72. Lncs. Springer.

Smans, Jan, Bart Jacobs, Frank Piessens, and Wolfram Schulte. 2008. "Automatic Verifier for Java-Like Programs Based on Dynamic Frames." In *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008*, edited by José Luiz Fiadeiro and Paola Inverardi, 4961:261–75. Lncs. Springer.

Swamy, Nikhil, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. "Secure Distributed Programming with Value-Dependent Types." In *ICFP 2011*, 266–78. ACM.

Tarski, Alfred. 1955. "A Lattice-Theoretical Fixpoint Theorem and Its Applications." *Pacific Journal of Mathematics* 5: 285–309.

Winskel, Glynn. 1993. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press.