TÖL212M Lokapróf
TÖL212M Final Exam
Spurningabók
Book of Questions

1. Engin hjálpargögn eru leyfileg, en prófið inniheldur hjálplegar upplýsingar fyrir aftan spurningarnar.
   No help materials are allowed, but the exam contains helpful information following the questions.
2. Skrifið svörin í svarabókina, ekki á önnur blöð og ekki á baksíður.
   Write the answers in the book of answers, not on other pages and not on the back sides.
3. Ef svarið kemst ekki fyrir á tilteknu svæði má skrifa á auðar síður aftast, en þá skalt þú láta vita af því með því að skrifa tilvísun í tiltekið svæði, til dæmis „framhald á blaðsíðu 36“.
   If the answer does not fit on the allocated space you may write on empty pages at the back, but then you should indicate this by referring to the relevant space, for example writing “continued on page 36”.
4. Forðist að skemma eða rífa þessar síður, þær þurfa að fara gegnum skanna. Skrifið skýrt með **dökku letri** og ekki skrifa í spássíur.
   Refrain from damaging or tearing these pages, they need to be passed through a scanner. Write clearly with **dark letters** and do not write on margins.
5. Baksíður **verða ekki skannaðar** og má nota fyrir krass. **Ekki verður tekið tillit til** svara sem skrifuð eru á baksíður.
   The backs of pages **will not be scanned** and can be used for scratch. Answers written on back pages **will be ignored**.

6. Prófið skiptist í **hluta**. Svarið **8** spurningum í heild og að minnsta kosti **tilteknum lágmarksfjölda** í hverjum hluta.
   The exam is divided into **parts**. Answer **8** questions in total and at least the **required minimum** in each part.
7. Ef þú svarar fleiri en **8** spurningum þá verður einkunn þín reiknuð sem **meðaltal allra svara** nema þú **krossir skýrt út** svör sem þú vilt ekki að gildi. Þú verður að krossa út allt svarið, ekki aðeins hluta þess.
   If you answer more than **8** questions then your grade will be computed as the **average of all answers** unless you **clearly cross out** answers you do not want to count. You must cross out the whole answer, not just part of it.
8. Munið að öll Dafny föll þurfa **notkunarlýsingu** með **requires/ensures**. Allar lykkjur þurfa **invariant** sem dugar til að rökstyðja.
   Remember that all Dafny functions need a **description** with **requires/ensures**. All loops need an **invariant** that is sufficient for supporting its correctness.
9. Munið að öll Java föll þurfa **notkunarlýsingu** með Notkun/Fyrir/Eftir. Allar lykkjur þurfa **fastayrðingu** sem dugar til að rökstyðja.
   Remember that all Java functions need a **description** with Usage/Pre/Post. All loops need an **invariant** that is sufficient to support its correctness.
10. Munið að nota viðeigandi **innfellingu** í öllum forritstexta.
    Remember to use the appropriate **indentation** in all code.
11. Ekki þarf að kalla á neinar hjálparsetningar í Dafny, jafnvel þótt vera megi að slíkt sé nauðsynlegt svo Dafny samþykki lausnina. Sama gildir um assert.
    You do not need to call any lemmas in Dafny even though that might be necessary for Dafny to accept your solution. The same applies to asserts.

Hluti I – Helmingunarleit o.fl.
Part I – Binary Search etc.

Svarið að minnsta kosti tveimur spurningum í þessum hluta – Munið að svara a.m.k. 8 spurningum í heild
Answer at least two questions in this part – Remember to answer at least 8 questions in total

1.

Skrifið fall í Dafny sem leitar með helmingunarleit í heiltalnarunu, a, sem raðað er í vaxandi röð, að fremsta sæti sem inniheldur tölu stærri en 2025. Ef ekkert slíkt sæti er til skal skila lengd rununnar. Fallið skal aðeins taka a sem viðfang. Notið lykkju, ekki endurkvæmni.
Write a function in Dafny that uses binary search in an integer sequence a, that is sorted in ascending order, to find the leftmost position containing a number that is greater than 2025. If no such position exists, return the length of the sequence. The function should only take a as an argument. Use a loop, not recursion.

2.

Skrifið endurkvæmt helmingunarleitarfall í Dafny sem leitar í svæði í heiltalnarunu sem raðað er í minnkandi röð að aftasta sæti innan svæðisins sem inniheldur tölu sem er stærri en 2025. Ef ekkert slíkt sæti er til innan svæðisins skal skila -1.
Write a recursive binary search function in Dafny that searches in a section of an integer sequence that is sorted in descending order for the rightmost position that contains a number greater than 2025. If no such position exists return -1.

3.

Skrifið endurkvæmt helmingunarleitarfall í Java sem hefur eftirfarandi lýsingu:
Write a recursive binary search function in Java with the following description:

```
// Notkun: int k = find(a,i,n,x);
// Fyrir:  0 <= i <= i+n <= a.length, x er heiltala,
//         a[i..i+n) er í vaxandi röð og inniheldur
//         engar endurtekningar.
// Eftir:  i <= k <= i+n.
//         Ef x er í a[i..i+n) þá er k sætið sem
//         inniheldur x, annars er k sætið þar sem x
//         ætti að vera til að halda röðun, þ.a. öll
//         fremri sæti innihalda gildi <x.
// Usage:  int k = find(a,i,n,x);
// Pre:    0 <= i <= i+n <= a.length, x is an int,
//         a[i..i+n) is in ascending order and
//         has no duplicate values.
// Post:   i <= k <= i+n.
//         If x is in a[i..i+n) then k is the position
//         that contains x, otherwise k is the position
//         which should contain x to maintain order,
//         such that all positions before contain
//         values <x.
static int find( int[] a, int i, int n, int x )
{
  …
}
```

Hluti II – Quicksort o.fl.
Part II – Quicksort etc.

Svarið að minnsta kosti einni spurningu í þessum hluta – Munið að svara a.m.k. 8 spurningum í heild
Answer at least one question in this part – Remember to answer at least 8 questions in total

4.

Gerið ráð fyrir að til sé Dafny fall með eftirfarandi lýsingu:
Assume a Dafny function exists with the following description:

```
method MiddlePartition( a: array<int>, i: int, j: int )
      returns( p: int )
   modifies a
   requires 0 <= i < j <= a.Length
   ensures i <= p < j
   ensures forall r | i <= r < p :: a[r] <= a[p]
   ensures forall r | p < r < j :: a[r] >= a[p]
   ensures a[p] == old(a[(i+j)/2])
   ensures multiset(a[i..j]) == old(multiset(a[i..j]))
   ensures a[..i] == old(a[..i])
   ensures a[j..] == old(a[j..])
```

Skrifið Quicksort fall sem notar þetta fall sem hjálparfall. Ekki forrita MiddlePartition fallið hér.
Write a Quicksort function that uses this function as a helper function. Do not implement the MiddlePartition function here.

5.

Forritið stofn fallsins MiddlePartition sem lýst er að ofan. Munið að allar lykkjur þurfa fastayrðingu.
Program the body of the MiddlePartition function described above. Remember that all loops need an invariant.

6.

Forritið eftirfarandi Dafny fall – Program the following Dafny function:

```
method Quickselect( a: array<int>, k: int )
    modifies a
    requires 0 <= k < a.Length
    ensures multiset(a[..]) == old(multiset(a[..]))
    ensures forall r | 0 <= r < k :: a[r] <= a[k]
    ensures forall r | k < r < a.Length :: a[r] >= a[k]
```

Þið megið nota MiddlePartition fallið sem hjálparfall. Sjáið til þess að meðaltímaflækjan fyrir slembin fylki sé ekki verri en $O(n)$ þar sem n er stærð fylkisins.
You may use the MiddlePartition function as a helper function. Make sure that the average time complexity for random arrays is not worse than $O(n)$ where n is the size of the array.

Hluti III – Tvíleitartré
Part III – Binary Search Trees

Svarið að minnsta kosti tveimur spurningum í þessum hluta – Munið að svara a.m.k. 8 spurningum í heild
Aftast í prófinu eru lýsingar á helstu föllum í skránni BST.dfy.
Answer at least two questions in this part – Remember to answer at least 8 questions in total
At the end of the exam there are descriptions of the most relevant functions in the file BST.dfy.

7.

Gerið ráð fyrir skilgreiningunni

```
datatype BST = BSTEmpty | BSTNode(BST,int,BST)
```

eins og í skránni okkar BST.dfy. Gerið einnig ráð fyrir föllunum IsTreePath, TreeSeq, PreSeq, MidSeq, PostSeq, PreSeqIncluding, o.s.frv.
Skrifið fall sem leitar í tvíleitartré og skilar tilvísun á fremsta hnút í milliröð sem inniheldur jákvæða tölu (tölu stærri en núll), eða skilar BSTEmpty ef slíkur hnútur finnst ekki í leitartrénu.
Munið að skrifa fulla lýsingu með requires/ensures og skrifa invariant fyrir lykkjuna ef þið notið lykkju.

Assume the definition

```
datatype BST = BSTEmpty | BSTNode(BST,int,BST)
```

as in our file BST.dfy. Also assume the functions IsTreePath, TreeSeq, PreSeq, MidSeq, PostSeq, PreSeqIncluding, etc. Write a function that searches in a binary search tree and returns a reference to the leftmost node that contains a positive number (a number greater than zero), or returns BSTEmpty is such a node does not exist in the tree. Remember to write a full description with requires/ensures and to write an invariant for your loops if you use loops.

8. **Leysið sama vandamál og að ofan, en núna í Java. Notið gamalkunna skilgreiningu á trjáhnútum, sem sjá má fyrir neðan. (Ég sleppi hér Notkun/Fyrir/Eftir til að spara pláss því þau eru gamalkunnug og augljós. Athugið að það þýðir ekki að nemendur megi sleppa að skrifa gamalkunnugar og augljósar lýsingar fyrir sína klasa.)**

```
public class BSTNode {
  private BSTNode left,right;
  private int val;
  public BSTNode( BST a, int x, BST b )
  { left=a; val=x; right=b; }
  public static left( BSTNode t ) { return t.left; }
  public static right( BSTNode t ) { return t.right; }
  public static rootValue( BSTNode t ) { return t.val; }
}
```

Solve the same problem as above, but this time in Java. Use our familiar definition of tree nodes, as seen above.
(I am skipping Usage/Pre/Post to save space because they are familiar and obvious. Note that this does not mean that students can skip writing familiar and obvious descriptions for their classes.)

9. **Skrifið fall í Dafny sem tekur eitt viðföng, sem er tvíleitartré t, og skilar true ef neikvæð tala er til í trénu en skilar false annars. Þið megið nota lykkju eða endurkvæmni, að því tilskildu að rökstuðningur sé réttur (þ.e. rétt requires, ensures og invariant).**
Write a function in Dafny that takes one argument, a binary search tree t, and returns true if a negative number exists in the tree and returns false otherwise. You may use a loop or recursion, given that the reasoning is correct (i.e. correct requires, ensures and invariant)

10. **Leysið sama vandamál og að ofan, en í Java.**

    Solve the same problem as above, but in Java.

Hluti IV – Ýmislegt
Part IV -- Miscellaneous

Svarið að minnsta kosti einni spurningu í þessum hluta – Munið að svara a.m.k. 8 spurningum í heild
Answer at least one question in this part – Remember to answer at least 8 questions in total

11.

Forritið stofninn á fallinu að neðan. Notið endurkvæmni en ekki lykkju.
Program the body of the function below. Use recursion and not a loop.

```
method Minimum( f: real->real
              , a: real
              , b: real
              , c: real
              , eps: real )
  returns( d: real, e: real )
    decreases ((c-a)/eps).Floor
    requires a < c
    requires b == (a+c)/2.0
    requires eps > 0.0
    requires f(b) <= f(a)
    requires f(b) <= f(c)
    ensures a <= d < e <= c
    ensures e-d < eps
    ensures f((d+e)/2.0) <= f(d)
    ensures f((d+e)/2.0) <= f(e)
```

Vísbending: Eftirfarandi hjálparsetningu má sanna í Dafny. Þið þurfið ekki að kalla á hana.
Hint: the following lemma can be proven in Dafny. You do not need to call it.

```
lemma BisectionTermination( x: real, y: real, eps: real )
    requires eps > 0.0
    requires x >= eps > 0.0
    requires y == x/2.0
    ensures (x/eps).Floor > (y/eps).Floor
```

12.

Forritið stofninn á fallinu að neðan. Notið lykkju en ekki endurkvæmni. Munið að setja decreases klausu í lykkjuna. Íhugið vísbendinguna í dæminu á undan.
Program the body of the function below. Use a loop and not recursion. Remember to put a decreases clause in the loop. Consider the hint in the previous question.

```
method Minimum( f: real->real
              , a: real
              , b: real
              , c: real
              , eps: real )
  returns( d: real, e: real )
    requires a < c
    requires b == (a+c)/2.0
    requires eps > 0.0
    requires f(b) <= f(a)
    requires f(b) <= f(c)
    ensures a <= d < e <= c
    ensures e-d < eps
    ensures f((d+e)/2.0) <= f(d)
    ensures f((d+e)/2.0) <= f(e)
```

Mikilvæg föll og tög í BST.dfy
Important functions and types in BST.dfy

```
// Skilgreining BST / Definition of BST:
datatype BST = BSTEmpty | BSTNode(BST,int,BST)
// Gildi af tagi BST eru tvíundartré.
// Values of type BST are binary trees.

// Skilgreining trjáslóða / The definition of tree paths:
newtype dir = x | 0 <= x <= 1
// Trjáslóðir eru af tagi seq<dir>.
// Tree paths are of type seq<dir>.

// Öll þau fyrirbæri sem hér er lýst má nota í
// röksemdafærslu, þ.e. í requires/ensures/invariant.
// Þau föll sem hafa Notkun/Fyrir/Eftir má nota í
// raunverulegum útreikningum en hin, sem hafa
// Notkun/Fyrir/Gildi, eru einungis nothæf í
// röksemdafærslu.

// All the items described here can be used in reasoning,
// i.e. in requires/ensures/invariant.
// Those functions that have Usage/Pre/Post can be used in
// real computations, but the others, that have
// Usage/Pre/Value, can only be used in reasoning.

// Notkun: var t := BSTEmpty;
// Fyrir:  Ekkert.
// Eftir:  t er tómt tvíundartré.

// Usage:  var t := BSTEmpty;
// Pre:    Nothing.
// Post:   t is an empty binary tree.

// Notkun: var t := BSTNode(p,x,q);
// Fyrir:  p og q eru tvíundartré.
// Eftir:  t er tvíundartré með x í rót,
//         með p sem vinstra undirtré og
//         með q sem hægra undirtré.

// Usage:  var t := BSTNode(p,x,q);
// Pre:    p and q are binary trees.
// Post:   t is a binary tree with x in the root,
//         with p as the left subree and
//         with q as the right subtree.
```

```
// Notkun: var x = RootValue(t);
// Fyrir:  t er tvíundartré, ekki tómt.
// Eftir:  x er gildið í rót t.

// Usage:  var x = RootValue(t);
// Pre:    t is a binary tree, not empty.
// Post:   x is the value in the root of t.

// Notkun: var l = Left(t);
// Fyrir:  t er tvíundartré, ekki tómt.
// Eftir:  l er vinstra undirtré t.

// Usage:  var l = Left(t);
// Pre:    t is a binary tree, not empty.
// Post:   l is the left subtree of t.

// Notkun: var r = Right(t);
// Fyrir:  t er tvíundartré, ekki tómt.
// Eftir:  r er hægra undirtré t.

// Usage:  var r = Right(t);
// Pre:    t is a binary tree, not empty.
// Post:   r is the right subtree of t.

// Notkun: TreeIsSorted(t)
// Fyrir:  t er tvíundartré.
// Gildi:  satt ef t er tvíleitartré, þ.e. í vaxandi
//         milliröð, ósatt annars.

// Usage:  TreeIsSorted(t)
// Pre:    t is a binary tree.
// Value:  true is t is a binary search tree, i.e. in
//         ascending order when traversed inorder,
//         otherwise false.

// Notkun: TreeSeq(t)
// Fyrir:  t er tvíundartré.
// Gildi:  Runa gildanna í t í milliröð,
//         af tagi seq<int>.

// Usage:  TreeSeq(t)
// Pre:    t is a binary tree.
// Value:  The sequence of the values in t when t is
//         traversed inorder, of type seq<int>.
```

```
// Notkun: IsTreePath(t,p)
// Fyrir:  t er tvíundartré, p er trjáslóð.
// Gildi:  Satt ef p er slóð innan t, annars ósatt.
// Ath.:   Trjáslóðin [] er slóð innan allra trjáa.
//         Ef p==[0]+q þá er p trjáslóð innan t ef t
//         er ekki tómt og q er trjáslóð innan Left(t).
//         Ef p==[1]+q þá er p trjáslóð innan t ef t
//         er ekki tómt og q er trjáslóð innan Right(t).

// Usage:  IsTreePath(t,p)
// Pre:    t is a binary tree, p is a tree path.
// Value:  true is p is a path within t, otherwise false.
// Note:   The path [] is a path within all trees.
//         If p==[0]+q then p is a path within t if t is not
//         empty and q is a path within Left(t).
//         If p==[1]+q then p is a path within t if t is not
//         empty and q is a path within Right(t).

// Notkun: Subtree(t,p)
// Fyrir:  t er tvíundartré, p er trjáslóð innan t.
// Gildi:  Undirtréð innan t sem p vísar á.
// Ath.:   Ef p er [] þá er skilagildið t, ef p==[0]+q
//         þá er það Subtree(Left(t),q) og ef p==[1]+q
//         þá er það Subtree(Right(t),q).

// Usage:  Subtree(t,p)
// Pre:    t is a binary tree, p is a tree path.
// Value:  The subtree within t that p refers to.
// Note:   If p is [] then the return value is t, if p==[0]+q
//         then it is Subtree(Left(t),q) and if p==[1]+q
//         then it is Subtree(Right(t),q).

// Notkun: PreSeq(t,p)
// Fyrir:  t er tvíundartré, p er trjáslóð innan t.
// Gildi:  Runa þeirra gilda í milliröð  innan t sem
//         eru fyrir framan undirtréð sem p vísar á.

// Usage:  PreSeq(t,p)
// Pre:    t is a binary tree, p is a tree path within t.
// Value:  The sequence of the values when traversed inorder
//         that are to the left of the subtree that p refers
//         to.
```

```
// Notkun: PostSeq(t,p)
// Fyrir:  t er tvíundartré, p er trjáslóð innan t.
// Gildi:  Runa þeirra gilda í milliröð  innan t sem
//         eru fyrir aftan undirtréð sem p vísar á.

// Usage:  PostSeq(t,p)
// Pre:    t is a binary tree, p is a tree path within t.
// Value:  The sequence of the values when traversed inorder
//         that are to the right of the subtree that p refers
//         to.

// Notkun: MidSeq(t,p)
// Fyrir:  t er tvíundartré, p er trjáslóð innan t.
// Gildi:  Runa gildanna í milliröð sem eru í
//         undirtrénu sem p vísar á. Sama og
//         TreeSeq(Subtree(t,p))

// Usage:  MidSeq(t,p)
// Pre:    t is a binary tree, p is a tree path within t.
// Value:  The sequence of the values inorder that are in
//         the subtree that p refers to. Same as
//         TreeSeq(Subtree(t,p))

// Fyrir sérhverja trjáslóð p innan t gildir:
// For each tree path p within t we have:
//   TreeSeq(t) == PreSeq(t,p)+MidSeq(t,p)+PostSeq(t,p)

// Notkun: PreSeqIncluding(t,p)
// Fyrir:  t er tvíundartré og p er trjáslóð innan t
//         sem vísar á ekki-tómt undirtré.
// Gildi:  Runa þeirra gilda í t, í milliröð, sem eru
//         í hnútunum fram til hnútsins sem p vísar á
//         að þeim hnút meðtöldum.

// Usage:  PreSeqIncluding(t,p)
// Pre:    t is a binary tree and p is a tree path within t
//         that refers to a non-empty subtree.
// Value:  The sequence of values in t, when traversed inorder
//         that are in nodes to the left of the node that p
//         refers to and also including that node.

// Notkun: PostSeqExcluding(t,p)
// Fyrir:  t er tvíundartré og p er trjáslóð innan t
//         sem vísar á ekki-tómt undirtré.
// Gildi:  Runa þeirra gilda í t, í milliröð, sem eru
//         í hnútunum fyrir aftan hnútinn sem p vísar
//         á.
```

```
// Usage:  PostSeqExcluding(t,p)
// Pre:    t is a binary tree and p is a tree path within t
//         that refers to a non-empty subtree.
// Value:  The sequence of values in t, when traversed inorder
//         that are in nodes to the right of the node that p
//         refers to and excluding that node.

// Notkun: PreSeqExcluding(t,p)
// Fyrir:  t er tvíundartré og p er trjáslóð innan t
//         sem vísar á ekki-tómt undirtré.
// Gildi:  Runa þeirra gilda í t, í milliröð, sem eru
//         í hnútunum fyrir framan hnútinn sem p
//         vísar á.

// Usage:  PreSeqExcluding(t,p)
// Pre:    t is a binary tree and p is a tree path within t
//         that refers to a non-empty subtree.
// Value:  The sequence of values in t, when traversed inorder
//         that are in nodes to the left of the node that p
//         refers to and excluding that node.

// Notkun: PostSeqIncluding(t,p)
// Fyrir:  t er tvíundartré og p er trjáslóð innan t
//         sem vísar á ekki-tómt undirtré.
// Gildi:  Runa þeirra gilda í t, í milliröð, sem eru
//         í hnútnum sem p vísar á eða fyrir aftan
//         hann.

// Usage:  PostSeqIncluding(t,p)
// Pre:    t is a binary tree and p is a tree path within t
//         that refers to a non-empty subtree.
// Value:  The sequence of values in t, when traversed inorder
//         that are in nodes to the right of the node that p
//         refers to and also including that node.


// Fyrir sérhverja trjáslóð p sem vísar á ekki-tómt undirtré
// innan t gildir:
// For each tree path p that refers to a non-empty subtree
// within t, we have:
//
// TreeSeq(t) == PreSeqExcluding(t,p)+PostSeqIncluding(t,p)
// TreeSeq(t) == PreSeqIncluding(t,p)+PostSeqExcluding(t,p)
// PreSeqExcluding(t,p) == PreSeq(t,p)+TreeSeq(Left(Subtree(t,p)))
// PreSeqIncluding(t,p) == PreSeqExcluding(t,p)+[RootValue(Subtree(t,p))]
// PostSeqExcluding(t,p) == TreeSeq(Right(Subtree(t,p)))+PostSeq(t,p)
// PostSeqIncluding(t,p) == [RootValue(Subtree(t,p))]+PostSeqExcluding(t,p)
```

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

Appendix A

Dafny Syntax Cheat Sheet

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

A.0. Declarations

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

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

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

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

A.0.0. Types and type declarations

Here are some example type declarations:

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

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

type OpaqueType

type TypeSynonym = int
```

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

Examples of types:

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

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

A.0.1. Member declarations

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

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

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

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

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

Declarations of fields and constants:

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

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

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

A.1. Statements

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

Declaration of local variables:

```
var x: X;
```

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

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

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

Assignments:

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

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

Dynamic allocation of objects and arrays:

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

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

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

Other primitive statements:

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

Some composite statements:

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

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

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

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

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

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

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

A.2. Expressions

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

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

Figure A.0. Operator binding powers.

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

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

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

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

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

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

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

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

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

Here are some literals and other expressions:

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

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

if E then E0 else E1

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

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

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

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

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

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