6.S057 Fall 2026 Exam 1

SOLUTIONS

- You have **50 minutes** to complete this exam. There are **5 problems** that vary in length.
- Please **WAIT** until instructed to begin.
- The exam is **closed-book** and closed-notes.
- Write all answers in the boxes provided.
- You must write legibly: answers we cannot confidently read will not receive credit.
- If you have a question, please raise your hand.
- If you find yourself bogged down on one part of the exam, remember to keep going and work on other problems, and then come back.
- You **may not discuss** details of the exam with anyone other than course staff until exam grades have been assigned and released.

Good luck!

continue to next page

Problem 1

The following function computes integer quotient and remainder. Reasoning backward from the end of the function, in each numbered blank (1) through (4), fill in a predicate for what must be true at that point in the function in order for the postcondition to hold.

Following Dafny's own rules for such calculations, write each step without simplification or beautification so it is clear how your formulas are derived. However, you may use the abbreviation `POST` for the subexpression `a == q * b + r && r < b` in your answers.

```
method DivMod(a: nat, b: nat) returns (q: nat, r: nat)
  requires b > 0
  ensures a == q * b + r
  ensures r < b
{
    if a < b {
        q, r := 0, a;
    } else {
        var q', r' := DivMod(a / 2, b);
        q, r := 2 * q', 2 * r' + a % 2;

(4)     assert [ (r >= b && a == (q + 1) * b + (r - b) && (r - b) < b)
               || (r < b && a == q * b + r && r < b)                    ];

        if r >= b {

(3)         assert [ a == (q + 1) * b + (r - b) && (r - b) < b          ];

            q, r := q + 1, r - b;

(2)         assert [ a == q * b + r && r < b                            ];

        }

(1)     assert [ a == q * b + r && r < b                                ];

    }
    assert a == q * b + r && r < b;
}
```

Problem 2

Automatic induction is disabled. Complete the definition of `DecreasesForF` and the body of `LemmaAboutF` so that Dafny proves termination and correctness.

```
function F(n: nat): nat
    decreases DecreasesForF(n)
{
    if n <= 2 then
        0
    else if n % 2 == 0 then
        2 + F(n / 2)
    else
        2 + F(n + 1)
}

function DecreasesForF(n: nat): nat {
    if n % 2 == 0 then [ n ] else [ n + 2 ]
}

lemma {:induction false} LemmaAboutF(n: nat)
    ensures F(n) % 2 == 0
    decreases DecreasesForF(n)
{
    [
    if n > 2 {
        if n % 2 == 0 {
            LemmaAboutF(n / 2);
        } else {
            LemmaAboutF(n + 1);
        }
    }
    ]
}
```

Problem 3

Here is our friend `List`:

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

In class we saw this higher-order function called "fold-left," also known as "reduce," that produces an output of type `B` from an input `List<A>` by "folding" values of the list into the output from left to right:

```
function FoldL<A, B>(f: (B, A) -> B, acc: B, ls: List<A>): B {
  match ls
  case Nil => acc
  case Cons(x, ls') => FoldL(f, f(acc, x), ls')
}
```

We can define "fold-right" which goes from right to left:

```
function FoldR<A, B>(f: (A, B) -> B, acc: B, ls: List<A>): B {
  match ls
  case Nil => acc
  case Cons(x, ls') => f(x, FoldR(f, acc, ls'))
}
```

For example:

```
  var ls := Cons(x1, Cons(x2, Cons(x3, Nil))); // a.k.a. [x1, x2, x3]
  assert FoldL(f, e, ls) == f(f(f(e, x1), x2), x3);
  assert FoldR(f, e, ls) == f(x1, f(x2, f(x3, e)));
```

In what situations will fold-left and fold-right compute the same result? Looking at the signatures, we will need a function `f: (A, A) -> A`. Then we need two more properties:

1. we will need an associative `f`, and
2. we will need `e` to be an identity.

In the lemma `FoldLeftEqRight` on the following page, fill in those two properties as preconditions. Then, for each step of the `calc`, explain in a single phrase what justifies that calculation step.

Here are the fold functions again, for reference:

```
function FoldL<A, B>(f: (B, A) -> B, acc: B, ls: List<A>): B {
  match ls
  case Nil => acc
  case Cons(x, ls') => FoldL(f, f(acc, x), ls')
}
function FoldR<A, B>(f: (A, B) -> B, acc: B, ls: List<A>): B {
  match ls
  case Nil => acc
  case Cons(x, ls') => f(x, FoldR(f, acc, ls'))
}
```

Now the lemma:

```
lemma FoldLeftEqRight<A>(f: (A, A) -> A, e: A, ls: List<A>)
  requires  forall a, b, c ::  f(f(a, b), c) == f(a, f(b, c))
  requires  forall a ::  f(e, a) == f(a, e)
  ensures FoldL(f, e, ls) == FoldR(f, e, ls)
{
  match ls
  case Nil =>
  case Cons(x, ls') =>
    calc {
      FoldL(f, e, Cons(x, ls'));
      == // definition of FoldL(f, e, Cons(x, ls'))
      FoldL(f, f(e, x), ls');
      == //  identity precondition, f(e, x) == f(x, e)
      FoldL(f, f(x, e), ls');
      == { HelperLemma(f, x, e, ls'); } // defined on the next page
      f(x, FoldL(f, e, ls'));
      == //  induction hypothesis, FoldLeftEqRight(f, e, ls')
      f(x, FoldR(f, e, ls'));
      == //  definition of FoldR(f, e, Cons(x, ls'))
      FoldR(f, e, Cons(x, ls'));
    }
}
```

And for the helper lemma, fill in the precondition and justify each step of the `calc`.

```
lemma HelperLemma<A>(f: (A, A) -> A, a: A, b: A, ls: List<A>)
  requires  forall a, b, c ::  f(f(a, b), c) == f(a, f(b, c))

  ensures FoldL(f, f(a, b), ls) == f(a, FoldL(f, b, ls))
{
  match ls
  case Nil =>
  case Cons(y, ls') =>
    calc {
      FoldL(f, f(a, b), Cons(y, ls'));
      == // definition of FoldL(f, f(a, b), Cons(y, ls'))
      FoldL(f, f(f(a, b), y), ls');
      == //  associativity precondition, f(f(a, b), y) == f(a, f(b, y))
      FoldL(f, f(a, f(b, y)), ls');
      == //  induction hypothesis, HelperLemma(f, a, f(b, y), ls');
      f(a, FoldL(f, f(b, y), ls'));
      == //  definition of FoldL(f, b, Cons(y, ls'))
      f(a, FoldL(f, b, Cons(y, ls')));
    }
}
```

Explain in one sentence why your precondition for `HelperLemma` is satisfied when `FoldLeftEqRight` invokes it.

It is identical to a precondition of `FoldLeftEqRight`.

And explain in one sentence: how does Dafny prove termination of the recursive proofs of `FoldLeftEqRight` and `HelperLemma`?

In both lemmas we invoke the induction hypothesis with a `List ls'` that is structurally included in argument `ls`.

Problem 4

Given this function and associated lemma...

```
function Splice(s: seq<nat>, x: nat, y: nat): seq<nat>
    requires 0 <= x <= |s|
{
    s[..x] + [y] + s[x..]
}

lemma SpliceHasAdded(s: seq<nat>, x: nat, y: nat)
    requires 0 <= x <= |s|
    ensures exists i : nat :: 0 <= i <= |s| && Splice(s, x, y)[i] == y
```

... Dafny gets stuck trying to prove the lemma fully automatically. The trouble is that Dafny cannot figure out which `i` value exists. Indeed, there is not obviously a finite set of candidate expressions to try for `i`, so Dafny can't just run through them all.

Write an expression for a value of `i` that satisfies the postcondition:

```
x
```

Using that value, give a body of `SpliceHasAdded` such that Dafny accepts the lemma as verified. Your answer should be a single assertion with no further annotations:

```
assert Splice(s, x, y)[x] == y;
```

Problem 5

Here is the beginning of an implementation for `MinStack`, a stack data structure that supports finding the minimum value without scanning through all the values:

```
module MinStack {
    export
        provides Stack, Empty, IsEmpty, Push, Pop, Min
        reveals IsMin

    datatype Stack = Nil | Node(top: int, min: int, below: Stack)

    ghost predicate IsMin(m: int, xs: seq<int>) {
        m in xs && forall y :: y in xs ==> m <= y
    }

    function Empty(): (s: Stack)
    {
        Nil
    }

    predicate IsEmpty(s: Stack)
    {
        s == Nil
    }

    function Push(s: Stack, x: int): (s': Stack)
    {
        Node(x, if s == Nil || x < s.min then x else s.min, s)
    }

    function Pop(s: Stack): (pair: (int, Stack))
    requires !IsEmpty(s)
    {
        (s.top, s.below)
    }

    function Min(s: Stack): int
    requires !IsEmpty(s)
    {
        s.min
    }
}
```

Here are the same definitions in compressed form for reference:

```
datatype Stack = Nil | Node(top: int, min: int, below: Stack)
ghost predicate IsMin(m: int, xs: seq<int>) { m in xs && forall y :: y in xs ==> m <= y }
function Empty(): (s: Stack) { Nil }
predicate IsEmpty(s: Stack) { s == Nil }
function Push(s: Stack, x: int): (s': Stack)
                              { Node(x, if s == Nil || x < s.min then x else s.min, s) }
function Pop(s: Stack): (pair: (int, Stack)) requires !IsEmpty(s) { (s.top, s.below) }
function Min(s: Stack): int requires !IsEmpty(s) { s.min }
```

Write an abstraction function for `MinStack`:

```
ghost function Abstr(s: Stack): seq<int>
```

```
{
    match s
    case Nil => []
    case Node(x, _, rest) => [x] + Abstr(rest)
}
```

And write a representation invariant:

```
ghost predicate Valid(s: Stack)
```

```
{
    match s
    case Nil => true
    case Node(_, m, rest) => IsMin(m, Abstr(s)) && Valid(rest)
}
```

Depending on your definitions, if you want additional function(s) used by the representation invariant or abstraction function, define them here:

none

For each function listed below, in addition to its specification in the provided skeleton, add appropriate preconditions and postconditions (with the help of your abstraction function and rep invariant) so that clients will be able to use the functions effectively and prove correctness of their code.

```
function Empty(): (s: Stack)
```

```
    ensures Valid(s) && Abstr(s) == []
```

```
predicate IsEmpty(s: Stack)
```

```
    ensures IsEmpty(s) <==> Abstr(s) == []
```

```
function Push(s: Stack, x: int): (s': Stack)
```

```
    requires Valid(s)
    ensures Valid(s') && Abstr(s') == [x] + Abstr(s)
```

```
function Pop(s: Stack): (pair: (int, Stack))
requires !IsEmpty(s)
```

```
    requires Valid(s) && !IsEmpty(s)
    ensures var (x, s') := pair; Valid(s') && [x] + Abstr(s') == Abstr(s)
```

```
function Min(s: Stack): int
requires !IsEmpty(s)
```

```
    requires Valid(s) && !IsEmpty(s)
    ensures IsMin(Min(s), Abstr(s))
```