Functional Programming: Practice Exam

This worksheet should give you an idea of what kinds of questions you can expect on the exam. Each question will be one of the following:

- A multiple choice question.
  - If a question asks "which of…", there are zero or more correct answers.
  - If a question asks "what is …", there is exactly one correct answer.
  - On the exam, this will also be specified on the answer sheet.
- A question about the type of something.
  - This may be the type of the goal.
- A question to put several things in the right order.
- A question that asks for a number or a yes/no answer.

Questions generally refer to the latest code snippet.

Warm-up

These are simple questions to get you used to the format. There will be a few of them on the exam, but but most will be harder.

I've included solutions to these questions at the end.

`P` implies `P`

```
example {P : Prop} : P → P := by
  intro h
  sorry
```

> **Question 1:** What is the goal immediately before the `sorry` tactic?

> **Question 2:** What is the type of `h`?

> **Question 3:** Which of the following can be used instead of `sorry` to complete the proof?
> A. `cases h`
> B. `intro h`
> C. `exact h`
> D. `assumption`

Again, but with a twist

```
example {P Q : Prop} : P → (P → Q) → Q := by
  sorry
```

> **Question 4:** You are given the following:
> 1. `apply f`
> 2. `exact p`
> 3. `intro f`
> 4. `intro p`
>
> Give the correct order to place these, replacing the `sorry`, so that the proof is complete.

> **Question 5:** Instead of `by sorry`, we can provide a term that completes the proof. Which of the following terms work?

A. `fun p => by apply p`
B. `fun p => fun f => f p`
C. `fun p f => p f`
D. `fun p f => f p`

Natural number or infinity

```
inductive NatOrInfinity where
  | finite (n : ℕ)
  | infinity
open NatOrInfinity

lemma eq_implies_finite_eq {n m : ℕ}
  : n = m → finite n = finite m := by
  sorry

lemma finite_eq_implies_eq {n m : ℕ}
  : finite n = finite m → n = m := by
  sorry
```

Note: when seeing the following two questions, you may worry that there is a trick, and the answer is "neither" because the proof fails in some minor way. This will not be the case: any incorrect proofs will be grossly incorrect.

Question 6: Which of the lemmas can be proven using the following proof?

```
intro h
cases h
rfl
```

A. Only `eq_implies_finite_eq`
B. Only `finite_eq_implies_eq`
C. Neither
D. Both

Question 7: Which of the lemmas can be proven using the following proof?

```
intro h
rewrite [h]
rfl
```

A. Only `eq_implies_finite_eq`
B. Only `finite_eq_implies_eq`
C. Neither
D. Both

Question 8: Which of the lemmas can be proven using the following proof?

```
exact (finite n = finite m)
```

A. Only `eq_implies_finite_eq`
B. Only `finite_eq_implies_eq`
C. Neither
D. Both

Question 9: Which of the following are provable?

A. `infinity = infinity`
B. `∀ a n, a = finite n ∨ a = infinity`
C. `¬ ∃ n, infinity = finite n`
D. `∀ n, infinity = finite n → n = 0`

Question 10: We want to define an order on `NatOrInfinity`. We define the following typeclass instance:

```
instance : Ord NatOrInfinity where
  compare
  | finite n, finite m => _
  | infinity, infinity => _
  | _, infinity => _
  | infinity, _ => _
```

There are four holes on the right-hand side of the `=>` here. Going top to bottom, arrange the following into which hole they should be placed in:

1. `.gt`
2. `.lt`
3. `.eq`
4. `compare n m`

General questions

These questions are not associated with any particular code.

Question 11: If we need something of type `A` and have `a : A` in context, we can often use `by assumption` to get `a`. What is the reason that this does not always work?

A. `by` is only allowed after `:=`.
B. `a` may have an inaccessible name (`a✝`).
C. `assumption` will only work if the goal type is in `Prop`.
D. We may have multiple values of type `A` in context, but need a specific one.

Question 12: Suppose `A` is a type. What is the difference between `A → ℕ` and `(a : A) → ℕ`?

A. `A → ℕ` is a function type, `(a : A) → ℕ` is a pair.
B. Both are function types, but `A → ℕ` takes any `A`, while `(a : A) → ℕ` only takes `x : A` if `x = a`.
C. Only the former is valid Lean.
D. Nothing, they are the same type.

Exam-level questions

These are questions of the difficulty you can expect on the exam.

Mapping lists

```
def myMap {α β : Type} (f : α → β) : List α → List β
  | [] => ?_1
  | x :: xs => ?_2 :: myMap ?_3 xs
```

This code has three holes, numbered 1 through 3.

Question 13: What is the type of each hole?

The implementation of `List.map` in the standard library follows the same structure. Recall that `id : {α : Type} → α → α` is the identity function.

```
example (xs : List α) : xs.map id = xs := by
  induction xs with
  | nil =>
```

> **Question 14:** Put the following lines into the correct order to complete the proof:
> 1. `| cons x xs IH =>`
> 2. `rewrite [IH]`
> 3. `rfl`
> 4. `rfl`
> 5. `unfold List.map`

> **Question 15:** What is the type of `IH` in the previous exercise?
> A. `∀ f, List.map f xs = f xs`
> B. `∀ xs, List.map id xs = xs`
> C. `List.map id xs = xs`
> D. `List.map id (x :: xs) = x :: xs`

```
example (f : α → β) (g : β → γ) (xs : List α)
  : xs.map (g ∘ f) = (xs.map f).map g := by
  sorry
```

> **Question 16:** Instead of `sorry`, which of these tactics will be a good first step in this proof?
> A. `induction xs`
> B. `induction (List α)`
> C. `rfl`
> D. `unfold List.map`

Primes

Our goal is to compute the first `n` prime numbers. Here's part of the code to do this:

```
def range (n m : ℕ) : List ℕ :=
  if n ≥ m
  then []
  else n :: range (n + 1) m

def primes : ℕ → List ℕ
  | 0 => []
  | n + 1 =>
      let ps := primes n
      ps ++ (?_1.filter (fun k => ps.all ?_2)).take 1
```

> **Question 17:** Which of the following ways of filling in hole 1 are type-correct?
> A. `range 2 (factorial n)` (assuming `factorial : ℕ → ℕ`)
> B. `range 2 infinity`
> C. `range 0 n`
> D. `range ps n`

> **Question 18:** What way of filling in hole 2 give the desired behaviour?
> A. `fun i => i % k == 0`

B. `fun i => i % k != 0`
C. `fun i => k % i != 0`
D. `fun i => 0 % k == i`

Suppose we fill in hole 1 with a sufficiently large range, so that it always contains a prime we haven't seen before, and fill in hole 2 with the correct answer from the last question.

Question 19: Which of the following may happen if we remove `.take 1` from the end of the line?
A. We get a type error.
B. The result may grow by more than one value at a time.
C. The result may contain the same prime multiple times.
D. The result may contain non-primes.

General questions

Question 20: Suppose `m` is a monad. Which of the following can be implemented?
A. `m (m a) → m a`
B. `m (a → b) → m a → m b`
C. `m (a → b)`
D. `m a → m (m a)`
E. `m a → a`
F. `(a → m b) → m a → b`

Answers to warm-up questions

Before reading these answers, I strongly suggest you open Lean and try to find the answer that way. I am mostly including these so that if you are unsure what a question means, you can check whether you understood correctly.

Question 1: `P`

Question 2: `P`

Question 3: C, D.

Question 4: 4, 3, 1, 2

Question 5: B, D

Question 6: D

Question 7: A

Question 8: C

Question 9: A, C, D

Question 10: 4, 3, 2, 1

Question 11: D

Question 12: D