```
-- CS2102 F19 Exam #1
--Derek Johnson dej3tc
--10/10/19
/-
#1 [10 points]

Complete the following definitions to give examples of
literal values of specified types. Use lambda expressions
to complete the questions involving function types. Make
functions (lambda expressions) simple: we don't care what
the functions do, just that they are of the right types.
-/
open nat
def x := λ (n:ℕ), n
def n : ℕ := 1
def s : string := "CS"
def b : bool := tt
def f1 : ℕ → bool := λ (n: ℕ), tt
def f2 : (ℕ → ℕ) → bool := λ x,tt
def f3 : (ℕ → ℕ) → (ℕ → ℕ) := λ x,x
def t1 : Type := nat
def t2 : Type → Type := λ (α : Type), α

/-
#2 [10 points]

Complete the following recursive function definition
so that the function computes the sum of the natural
numbers from 0 to (and including) a given value, n.
-/

def sumto : ℕ → ℕ
| 0 := 0
| (nat.succ n') := nat.succ n' + sumto(n')
/-
#3. [5 points]

We have seen that we can write function *specifications*
in the language of predicate logic, and specifically in
the language of pure functional programming. We also know
we can, and generally do, write *implementations* in the
language of imperative programming (e.g., in Python or in
Java). Complete the following sentence by filling in the
blanks to explain the esssential tradeoff between function
specifications, in the langugae of predicate logic, and
function implementations written in imperative programming
languages, respectively.

Specifications are generally understandable but less efficient
while implementations are generally the opposite: less
understandable but more efficient.

-/
```

```
/-
# 4. [10 points]

Natural languages, such as English and Mandarin, are very
powerful, but they have some fundamental weaknesses when it
comes to writing and verifying precise specifications and
claims about properties of algorithms and programs. It is
for this reason that computer scientists often prefer to
write express such things using mathematical logic instead
of natural language.

Name three fundamental weaknesses of natural language when
it comes to carrying out such tasks. You may given one-word
answers if you wish.

A. Ambiguous
B. Not Machine Checkable
C. Too Verbose (in some situations)

-/


/-
#5. [10 points]

What logical proposition expresses the claim that a given
implementation, I, of a function of type ℕ → ℕ, is correct
with respect to a specification, S, of the same function?

Answer: For all natural numbers, implementation I will
hold to / be true for specification S.
-/

/-
#6. [10 points]

What Boolean functions do the following definitions define?
-/

def mystery1 : bool → bool → bool
| tt tt := ff
| ff ff := ff
| _ _ := tt

/-
Answer: Not Equal To
-/

def mystery2 : bool → bool → bool
| tt ff := ff
| _ _ := tt
```

```
/-
Answer: Not (a, not b)
-/

/-
#7. [10 points]

Define a function that takes a string, s, and a natural
number, n, and that returns value of type (list string)
in which s is repeated n times. Give you answer by
completing the following definition: fill in underscores
with the answers that are needed. Note that the list
namespace is not open by default, so prefix constructor
names with "list." as we do for the first (base) case.
-/

def repeat : string → ℕ → list string
| s nat.zero := list.nil
| s (nat.succ n') := list.cons s (repeat s n')

#eval repeat "hello" 3
/-
#8. [10 points]

Definea a polymorphic function that takes (1) a type, α,
(2) a value, s : α, and (3) a natural number, n, and that
returns a list in which the value, a, is repeated n times.
Make the type argument to this function implicit. Replace
underscores as necessary to give a complete answer. Note
again that the list namespace is not open, so use "fully
qualified" constructor names.
-/

def poly_repeat {α : Type} : α → ℕ → list α
| s nat.zero := list.nil
| s (nat.succ n') := list.cons s (poly_repeat s n')

#eval poly_repeat "hello" 3
/-
#9. [10 points]

Define a data type, an enumerated type, friend_or_foe,
with just two terms, one called friend, one called foe.
Then define a function called eval that takes two terms,
F1 and F2, of this type and returns a term of this type,
where the function implements the following table:

F1      F2      result
friend  friend  friend
friend  foe     foe
foe     friend  foe
foe     foe     friend
-/
```

```
-- Answer here
inductive friend_or_foe : Type
 | friend : friend_or_foe
 | foe : friend_or_foe

open friend_or_foe
def eval : friend_or_foe → friend_or_foe → friend_or_foe
| friend friend := friend
| friend foe := foe
| foe friend := foe
| foe foe := friend

/-
#10. [10 points]

We studied the higher-order function, map. In particular,
we implemented a version of it, which we called mmap, for
functions of type ℕ → ℕ, and for lists of type (list ℕ).
The function is reproduced next for your reference. Read
and recall how the function works, then continue on to the
questions that follow.
-/

def mmap : (ℕ → ℕ) → list ℕ → list ℕ
| f [] := []
| f (list.cons h t) := list.cons (f h) (mmap f t)

-- An example application of this function
#eval mmap (λ n, n + 1) [1, 2, 3, 4, 5]

/-
A. Write a polymorphic version of this function, called
pmap, that takes (1) two type arguments, α and β, (2) a
function of type α → β, and (3) a list of values of type
α, and that returns the list of values obtained by
applying the given function to each value in the given
list. Make α and β implicit arguments.
-/

-- Answer here
def pmap {α β : Type} : (α → β) → list α → list β
| f [] := []
| f (list.cons h t) := list.cons (f h) (pmap f t)

/-
B.

Use #eval to evaluate an application of pmap to a function
of type ℕ to bool and a non-empty list of natural numbers.
Use a lambda abstraction to give the function argument. It
does not matter to us what value the function returns.
-/

-- Answer here
```

```
def tttt:= λ (n:ℕ), tt

def list1 := [1,2,3,4,5]

#eval pmap (λ (n:ℕ), tt) [1,2,3,4,5]

/-
#11. [10 points]

Define a data type, prod3_nat, with one constructor,
triple, that takes three natural numbers as arguments,
yielding a term of type prod3_nat. Then write three
"projection functions", prod3_nat_fst, prod3_nat_snd,
and prod3_nat_thd, each of which takes a prod3_nat value
and returns its corresponding component element. Hint:
look to see how we defined the prod type, its pair
constructor, and its two projection functions.
-/
inductive prod3_nat : Type
| triple (a :ℕ ) (b:ℕ ) (c : ℕ ) : prod3_nat

def prod3_nat_fst : prod3_nat → ℕ
| (prod3_nat.triple a b c) := a

def prod3_nat_snd : prod3_nat → ℕ
| (prod3_nat.triple a b c) := b

def prod3_nat_thd : prod3_nat → ℕ
| (prod3_nat.triple a b c) := c

def ss := prod3_nat.triple 2 3 4
#eval prod3_nat_fst ss

/-
Extra credit. Define prod3 as a version
of prod3_nat that is polymorphic in each of
its three components; define polymorphic
projection functions; and then use them to
define a function, rotate_right, that takes
a triple, (a, b, c), and returns the triple
(c, a, b).  (Call your type arguments α,
β, and γ - alpha, beta, and gamma).
-/

inductive prod3 (α β γ : Type) : Type
| triple (a : α) (b : β) (y : γ) : prod3

def prod3_fst {α β γ : Type}: prod3 α β γ → α
| (prod3.triple a b c) := a

def prod3_snd {α β γ : Type} : prod3 α β γ → β
| (prod3.triple a b c) := b
```

```
def prod3_thd {α β γ : Type} : prod3 α β γ → γ
| (prod3.triple a b c) := c

def rotate_right {α β γ : Type} : prod3 α β γ → prod3 γ α β
| (prod3.triple a b c) := prod3.triple c a b
```