18/20

Some mistakes in Q3. proofs in Q1 are nice, but in a few places can be made shorter/easier. (see my solutions)

Interactive Theorem Proving Assignment 1
Martin Skilleter (u6679334)
15 March 2019

I collaborated with the following people while completing this assignment: Isabel Longbottom (u6702144), Aaron Bryce (u6677442), Andy Yin (u6104905) and Laurence Fletcher (u6672947).

Question 1

```
namespace hidden
inductive Natural
| zero : Natural
| succ : Natural -> Natural

open Natural

instance : has_zero Natural :=
{ zero := zero}

instance : has_one Natural :=
{ one := succ zero}

def add : Natural -> Natural -> Natural
| a zero := a
| a (succ b) := succ (add a b).

instance : has_add Natural :=
{ add := add }

def times : Natural -> Natural -> Natural
| a zero := zero
| a (succ b) := add (times a b) a

instance : has_mul Natural :=
{ mul := times}

def pow : Natural -> Natural -> Natural
| a zero := (succ zero)
| a (succ b) := times (pow a b) a

instance : has_pow Natural Natural :=
{ pow := pow}
```

consistent spacing!

Can you explain what is better/worse if you define

| (succ a) b = succ (add a b)

instead?

I'd remove the whole "show" line here: the calc is already completely explicit about what's being proved.

```
theorem add_associativity (a b c : Natural) : (a + b) + c = a + (b + c) :=
Natural.rec_on c
(show (a + b) + 0 = a + (b + 0), from calc
     (a + b) + 0 = a + b : rfl
             ... = a + (b + 0) : rfl
)
(assume c, assume ih : (a + b) + c = a + (b + c),
show (a + b) + (c + 1) = a + (b + (c + 1)), from calc
     (a + b) + (c + 1) = ((a + b) + c) + 1 : rfl
                   ... = (a + (b + c)) + 1 : by rw ih
                   ... = a + ((b + c) + 1) : rfl
                   ... = a + (b + (c + 1)) : rfl
)
```

use "λ"; assume is okay for propositional arguments.

```
lemma zero_commutativity (a : Natural) : a + 0 = 0 + a :=
Natural.rec_on a
(show zero + 0 = 0 + zero, from rfl
)
(assume a, assume ih : a + 0 = 0 + a,
show (a + 1) + 0 = 0 + (a + 1), from calc
     (a + 1) + 0 = a + 1 : rfl
             ... = (a + 0) +1 : rfl
             ... = (0 + a) + 1 : by rw ih
             ... = 0 + (a + 1) : rfl
)

lemma one_commutativity (a : Natural) : a + 1 = 1 + a :=
Natural.rec_on a
(show (0 + 1 : Natural) = 1 + 0, by rw zero_commutativity)
(assume a, assume ih : a + 1 = 1 + a,
show (a + 1) + 1 = 1 + (a + 1), from calc
     (a + 1) + 1 = (1 + a) + 1 : by rw ih
             ... = 1 + (a + 1) : rfl
)
```

```
theorem add_commutativity (a b : Natural) : a + b = b + a :=
Natural.rec_on b
(show (a + 0 : Natural) = 0 + a, by rw zero_commutativity)
(assume b, assume ih : a + b = b + a,
show a + (b + 1) = (b + 1) + a, from calc
     a + (b + 1) = (a + b) + 1 : by rw add_associativity
             ... = (b + a) + 1 : by rw ih
             ... = b + (a + 1) : by rw add_associativity
             ... = b + (1 + a) : by rw one_commutativity
             ... = (b + 1) + a : by rw add_associativity
)

-- Question 1a
theorem left_distributivity (a b c : Natural) : a * (b + c) = a * b + a * c :=
Natural.rec_on c
(show a * (b + 0) = a * b + a * 0, from calc
     a * (b + 0) = a * b : rfl
             ... = a * b + 0 : rfl
             ... = a * b + a * 0 : rfl
)
(assume c, assume ih : a * (b + c) = a * b + a * c,
show a * (b + (c + 1)) = a * b + a * (c + 1), from calc
     a * (b + (c + 1)) = a * ((b + c) + 1) : by rw add_associativity
                   ... = a * (b + c) + a : rfl
                   ... = (a * b + a * c) + a : by rw ih
                   ... = a * b + (a * c + a) : by rw add_associativity
                   ... = a * b + a * (c + 1) : rfl
)

lemma times_zero (a : Natural) : 0 * a = 0 :=
Natural.rec_on a
(show (0 * 0 : Natural) = 0, by refl)
(assume a, assume ih : 0 * a = 0,
show 0 * (a + 1) = 0, from calc
     0 * (a + 1) = 0 * a + 0 * 1 : by rw left_distributivity
             ... = 0 + 0 * 1 : by rw ih
             ... = 0 + 0 : rfl
             ... = 0 : rfl
)
```

```
theorem right_distributivity (a b c : Natural) : (a + b) * c = a * c + b * c :=
Natural.rec_on c
(show (a + b) * 0 = a * 0 + b * 0, by refl)
(assume c, assume ih : (a + b) * c = a * c + b * c,
show (a + b) * (c + 1) = a * (c + 1) + b * (c + 1), from calc
     (a + b) * (c + 1) = ((a + b) * c) + (a + b) : rfl
                   ... = (a * c + b * c) + (a + b) : by rw ih
                   ... = ((a * c + b * c) + a) + b : by rw ← add_associativity
                   ... = (a * c + (b * c + a)) + b : by rw ← add_associativity
                   ... = (a * c + (a + b * c)) + b : by rw add_commutativity (b
* c)
                   ... = ((a * c + a) + b * c) + b : by rw ← add_associativity
                   ... = (a * c + a) + (b * c + b) : by rw ← add_associativity
                   ... = a * (c + 1) + b * (c + 1) : by refl
)

lemma right_times_one (a : Natural) : a * 1 = a :=
(show a * 1 = a, from calc
     a * 1 = a * 0 + a : rfl
       ... = 0 + a : rfl
       ... = a + 0 : by rw add_commutativity
       ... = a : rfl
)

lemma left_times_one (a : Natural) : 1 * a = a :=
Natural.rec_on a
(show (1 * 0 : Natural) = 0, by refl)
(assume a, assume ih : 1 * a = a,
show 1 * (a + 1) = a + 1, from calc
     1 * (a + 1) = 1 * a + 1 : rfl
            ...  = a + 1 : by rw ih
)
```

```
-- Question 1b
theorem times_commutativity (a b : Natural) : a * b = b * a :=
Natural.rec_on b
(show a * 0 = 0 * a, from calc
     a * 0 = 0 : rfl
       ... = 0 * a : by rw times_zero
)
(assume b, assume ih : a * b = b * a,
show a * (b + 1) = (b + 1) * a, from calc
     a * (b + 1) = a * b + a : rfl
             ... = b * a + a : by rw ih
             ... = a + b * a : by rw add_commutativity
             ... = 1 * a + b * a : by rw left_times_one
             ... = (1 + b) * a : by rw right_distributivity
             ... = (b + 1) * a : by rw add_commutativity
)

-- Question 1c
theorem times_associativity (a b c : Natural) : (a * b) * c = a * (b * c) :=
Natural.rec_on c
(show (a * b) * 0 = a * (b * 0), from calc
     (a * b) * 0 = 0 : rfl
             ... = a * 0 : rfl
             ... = a * (b * 0) : rfl
)
(assume c, assume ih : (a * b) * c = a * (b * c),
show (a * b) * (c + 1) = a * (b * (c + 1)), from calc
     (a * b) * (c + 1) = (a * b) * c + a * b : rfl
                   ... = a * (b * c) + a * b : by rw ih
                   ... = a * ((b * c) + b) : by rw left_distributivity
                   ... = a * (b * (c + 1)) : rfl
)

end hidden
```

great.

Question 2

```
import data.finset
import data.real.basic
import data.real.cau_seq_completion
import data.nat.gcd
import analysis.exponential

noncomputable theory

open nat
open finset
open int
open real
open complex

example : decidable_linear_ordered_comm_group ℝ := by apply_instance

-- n is the length of the arithmetic sequence we want, a is the starting number
-- and k is the common difference (k ≠ 0 or else we can choose a to be prime
and we are done)
theorem green_tao : Prop :=
∀ (n : ℕ), ∃ (a k : ℕ), ∀ (i < n), k ≠ 0 ∧ prime (a + i * k)
```

Not wrong, but awkward to put this under the quantifier for i.

You could also use: k : ℕ+ (there's a type 'pnat')

```
-- The nth partial sum of a series Σf
def P_n (f : ℕ → ℝ) (n : ℕ) : ℝ :=
finset.sum (finset.Ico 1 (n+1)) f

-- The series with f inside the sum
def converges (f : ℕ → ℝ) (L : ℝ) : Prop :=
∀ (ε > 0), ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → complex.abs (P_n f n - L) < ε
```

not wrong, but unpleasant to have to introduce the complex numbers here, and it will interfere with proofs!

```
-- Apery's Theorem says that the sum from n=1 to ∞ of 1/n^3 has a limit and
that the limit is irrational
theorem apery's_theorem : Prop :=
∃ (L : ℝ), converges (λ n, 1/(n^3)) L ∧ ∀ (p q : ℤ), (p/q : ℝ) ≠ L

def find_prime_factors (n: ℕ) : list ℕ :=
((list.range (n+1)).filter prime).filter (∣ n)

def rad (n : ℕ) : ℕ :=
(find_prime_factors n).prod
```

perhaps define a 'coprime' predicate?

This warrants more explanation. You need to know $x^{1+\varepsilon}$ is monotone in ε, as well.

```
-- We use formulation 2 of the abc conjecture from
https://en.wikipedia.org/wiki/Abc_conjecture
-- Note that because 1/n → 0 as n → ∞, this is equivalent to the ε
formulation
theorem abc_conjecture : Prop :=
∀ (n : ℕ), ∃ (K : ℝ), ∀ (a b c : ℕ),
n > 0 ∧ ∀ (z : ℕ), (z ∣ a ∧ z ∣ b ∧ z ∣ c → z = 1) ∧ a + b = c → (c : ℝ)
< K*(rad a*b*c : ℝ)^(1+1/n)
```

uhoh! This is definitely in the wrong place. See my commit.

this takes place in $\mathbb{N}$, so $1/n = 0$.

```
-- We cite the paper "An Elementary Problem Equivalent to the Riemann
Hypothesis"
-- by Jeffrey Lagarias, which can be found at
http://www.math.lsa.umich.edu/~lagarias/doc/elementaryrh.pdf

-- We begin by defining the harmonic series
def H_n (n : ℕ) : ℝ := P_n (λm, 1/m) n

def sum_of_divisors (n : ℕ) : ℝ :=
((list.range (n+1)).filter (∣ n)).sum

theorem riemann_hypothesis : Prop :=
∀ (n : ℕ), sum_of_divisors n ≤ H_n n + exp (H_n n)*log (H_n n) ∧
          sum_of_divisors n = H_n n + exp (H_n n)*log (H_n n) ↔ n = 1
```

Interestingly, this really should be a 'def', not a 'theorem'.

Sadly, this one is false too.

1. there are some essential parentheses missing!
2. what about $n=0$?

(see my commit)

Question 3

```
import data.list

namespace hidden

variable {α : Type}

def len {α : Type} : list α → ℕ
| [] := 0
| (hd :: tl) := 1 + len tl

def concat {α : Type} : list (list α) → list α
| [] := []
| (hd :: tl) := hd ++ (concat tl)

def nonempty {α : Type} : list α → Prop
| [] := false
| (_ :: _) := true

-- If the 2nd of two lists being concatenated is non-empty then their
concatenation is non-empty
lemma nonempty_tail (L M : list α) (h : nonempty M) : nonempty (L ++ M) :=
begin
   cases L,
   {simp,
   exact h},
   {simp}
end
```

```
-- If the 1st of two lists being concatenated is non-empty then their
concatenation is non-empty
-- This is the lemma that is actually used in the proof of our theorem, but the
other lemma
-- is needed for this proof
lemma nonempty_head (L M : list α) (h : nonempty L) : nonempty (L ++ M) :=
begin
   cases M,
   {simp,
   exact h},
   {have h₂ : nonempty (M_hd :: M_tl), begin
       dsimp [nonempty],
       trivial,
   end,
   apply nonempty_tail,
   exact h₂,
   }
end
```

no need to start a nested begin... end block

```
-- Given a non-empty list of lists, all of the elements of which are also
non-empty, we show that the full concatenation is non-empty
theorem nonempty_concat_of_nonempty_is_nonempty
(L : list (list α)) (h : nonempty L) (w : ∀ (m ∈ L), nonempty m) : nonempty
(concat L) :=
begin
 cases L,
 {cases h},
 {dsimp [concat],
  apply nonempty_head,
  apply w,
  simp,
 }
end

end hidden
```

great.

Question 4

```
import data.finset
import data.nat.choose
import tactic.squeeze

open nat
open finset

-- We apply the add_pow theorem, which is actually the binomial expansion under
a pseudonym
theorem binomial_expansion_for_2 (n : ℕ) :
finset.sum (finset.range (n+1)) (λ (m : ℕ), choose n m) = 2^n :=
begin
   have h := add_pow 1 1 n,
   simp only [nat.one_pow, mul_one, one_mul, nat.cast_id, nat.pow_eq_pow] at h,
   simp only [h, eq_self_iff_true],
end
```

✓ fine.