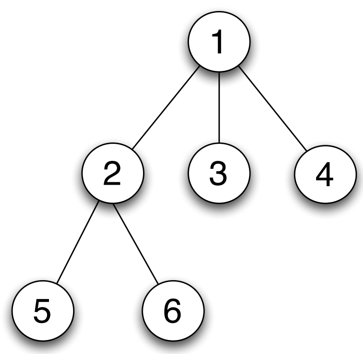
CIS 500 — Software Foundations

Midterm I

February 15, 2012

Answer key

1. (8 points) A *2-3 tree* is a tree data structure in which (1) every node is labeled with a value (drawn from some set `X`), and (2) every node has zero, two, or three children. For example, here is a 2-3 tree of numbers:



(a) Complete the following inductive definition of 2-3 trees:

```
Inductive ttree {X : Type} : Type :=
```

Answer:

```
| t_leaf : X -> ttree
| t_two : X -> ttree -> ttree -> ttree
| t_three : X -> ttree -> ttree -> ttree -> ttree.
```

(b) Write down a term of type `ttree nat` representing the tree shown above.

Answer:

```
t_three 1 (t_two 2 (t_leaf 5) (t_leaf 6)) (t_leaf 3) (t_leaf 4).
```

2. (6 points) Briefly explain the behavior of the `apply` and `apply... with...` tactics in Coq.

Answer: Invoking the tactic `apply H`*, where* `H` *is some hypothesis or previously proved theorem, matches the conclusion of* `H` *with the current goal and generates new subgoals for each premise of* `H`*. The* `apply...with...` *variant allows us to supply values for variables that appear in the premises of* `H` *but do not appear in its conclusion.*

3. (6 points) For each of the following types, define a function (using `Definition` or `Fixpoint`) with the given type.

(a) `nat -> list (list nat)`

Answer:

```
Definition foo1 (n:nat): list (list nat) := [n::nil].
```

(b) `forall X Y : Type, list X -> (X -> Y) -> list Y`

Answer:

```
Fixpoint foo3 (X Y:Type) (l: list X) (f:X->Y) : list Y:=
  match l with
  | []   => []
  | x::t => (f x)::(foo3 X Y t f)
  end.
```

4. (8 points) Write down the type of each of the following expressions. (For example, for the expression

```
fun (x y : nat) => beq_nat (x+y) 10
```

you'd write `nat -> nat -> bool`.) If an expression is not typeable, write "ill typed."

(a) `fun (x : nat) => x :: []`

Answer:

```
nat -> list nat
```

(b) `(2 :: 3 :: []) :: []`

Answer:

```
list (list nat)
```

(c)

```
fun (X : Type) (l : list X) =>
  match l with
    [] => []
  | h :: t => h
  end
```

Answer: Ill typed

(d)

```
fun (X Y Z : Type) (f : X->Y) (g : Y->Z) (a : X) =>
  g (f a)
```

Answer:

```
forall X Y Z : Type, (X->Y) -> (Y->Z) -> X -> Z
```

5. (12 points) In this question, we'll consider two different implementations of the same transformation on lists — one as an inductively defined relation and one as a `Fixpoint`.

(a) The relation `rdrop` is a three-place relation that holds between a number `n`, a list `xs`, and a list `xs'` if and only if `xs'` is the list obtained by dropping the first `n` elements of `xs`. For example, the following are all provable instances of `rdrop`.

```
rdrop 3 [1,2,3,4,5] [4,5]
rdrop 2 [5,4,3,2,1] [3,2,1]
rdrop 5 [1,2,3] []
```

Complete the following definition of `rdrop`.

```
Inductive rdrop {X : Type} : nat -> list X -> list X -> Prop :=
```

Answer:

```
  | d_zero : forall xs : list X, rdrop 0 xs xs
  | d_drop : forall (n : nat) (x : X) (xs ys : list X),
               rdrop n xs ys ->
               rdrop (S n) (x :: xs) ys
  | d_S_nil : forall n,  rdrop (S n) nil nil.
```

(b) Similarly, `fdrop` is a *function* that takes a number `n` and a list `xs` and returns the list consisting of all except the first `n` the elements of `xs`. For example:

```
fdrop 3 [1,2,3,4,5] = [4,5].
fdrop 2 [5,4,3,2,1] = [3,2,1].
fdrop 5 [1,2,3] = []
```

Complete the following `Fixpoint` definition of `fdrop`.

```
Fixpoint fdrop {X : Type} (n : nat) (xs : list X) : list X :=
```

Answer:

```
  match (n, xs) with
  | 0, _ => xs
  | S n', x :: xs' => fdrop n' xs'
  | S n', [] => []
  end.
```

One can also write this using nested matches:

```
Fixpoint fdrop {X : Type} (n : nat) (xs : list X) : list X :=
  match n with
  | 0 => xs
  | S n' =>
    match xs with
    | [] => []
    | x :: xs' => fdrop n' xs'
    end
  end.
```

6. (20 points) Recall the definition of `beq_nat`:

```
Fixpoint beq_nat (n m : nat) : bool :=
  match n with
  | O => match m with
         | O => true
         | S m' => false
         end
  | S n' => match m with
            | O => false
            | S m' => beq_nat n' m'
            end
  end.
```

Write out a careful informal proof of the following theorem, using the pedantic "template" style discussed in the notes. Make sure to state the induction hypothesis explicitly.

Theorem: For all natural numbers `n` and `m`, if `beq_nat n m = true` then `n = m`.

Proof: We show, by induction on `n`, that, for all `m`, if `true = beq_nat n m`, then `n = m`.

- Suppose `n = 0`. We must show, for all `m`, that if `true = beq_nat 0 m`, then `0 = m`. We proceed by cases on `m`.
  - If `m = 0`, we must show `0 = 0`, which holds by reflexivity.
  - If `m = S m'`, the hypothesis states that `true = beq_nat 0 (S m')`. But `beq_nat 0 (S m')` reduces to `false`, so this is absurd.
- Otherwise, `n = S n'`, and the induction hypothesis states that for all natural numbers `m'`, if `true = beq_nat n' m'`, then `n' = m'`. We must show that if `true = beq_nat (S n') m`, then `S n' = m`. We again proceed by cases on m.
  - If `m = 0`, the hypothesis states that `true = beq_nat (S n') 0`, which is absurd.
  - Otherwise `m = S m'`. Our hypothesis now states that `true = beq_nat (S n') (S m')`, which simplifies to `true = beq_nat n' m'`. We may therefore apply the induction hypothesis (instantiated at m') to conclude that `n' = m'`, which immediately implies that `S n' = S m'`. □

7. (10 points) Recall the inductive definitions of logical conjunction and the property `beautiful`:

```
Inductive and (P Q : Prop) : Prop :=
  conj : P -> Q -> (and P Q).

Notation "P /\ Q" := (and P Q) : type_scope.

Inductive beautiful : nat -> Prop :=
  b_0   : beautiful 0
| b_3   : beautiful 3
| b_5   : beautiful 5
| b_sum : forall n m, beautiful n -> beautiful m -> beautiful (n+m).
```

Suppose we have already proved the following theorem:

```
Theorem b1000: beautiful 1000.
```

Give a proof object for the following proposition. Show all parts of the proof object explicitly (i.e., do not use _ anywhere).

```
Definition b_facts : forall x,
                       beautiful x ->
                       (beautiful (1000 + x) /\ beautiful 3) :=
```

Answer:

```
fun x H =>
  conj (beautiful (1000 + x)) (beautiful 3) (b_sum 1000 x b1000 H) b_3.
```

8. (2 points) How many different proof objects are there for the proposition in the previous question?

Answer: Infinitely many.

9. (8 points) Recall the definition of existential quantification:

```
Inductive ex (X:Type) (P : X->Prop) : Prop :=
  ex_intro : forall (witness:X), P witness -> ex X P.
```

(a) Write a proposition capturing the claim "there is some number whose successor is beautiful."

Answer: `ex nat (fun n => beautiful (S n))`

Or: `exists n, beautiful (S n)`

(b) Give a proof object for this proposition.

Answer: `ex_intro nat (fun n => beautiful (S n)) 2 b_3`