Dependent types homework

Exercise 1 (2 points)

During the seminar we defined pairs by declaring new data type:

```
data Pair : Type -> Type -> Type where
    MkPair : a -> b -> Pair a b
```

Use dependent pairs to create an equivalent definition but without using `data` keyword, i.e. provide implementations for following type annotations:

```
Pair' : Type -> Type -> Type
MkPair' : a -> b -> Pair' a b
```

Exercise 2 (10 points)

Recall the `Fin n` type we used to define safe vector indexing:

```
data Fin : Nat -> Type where
    FZ : Fin (S n)
    FS : Fin n -> Fin (S n)
```

However, usually we work with `Nat` type. Provide implementations for following functions:

```
fin2Nat : Fin n -> Nat -- 1 point
nat2Fin : Nat -> (n : Nat) -> Maybe (Fin n) -- 3 points
```

Last part of this exercises requires you to prove injectiveness of `fin2Nat`:

```
fin2NatInjective : (m : Fin k) -> (n : Fin k) ->
                   (fin2Nat m) = (fin2Nat n) -> m = n -- 6 points
```

You may find the following lemmas useful (they are avaliable as part of Idris prelude so you don't have to declare them):

```
succInjective : (n : Nat) -> (m : Nat) ->
                S n = S m -> n = m
succInjective _ _ Refl = Refl

cong : {a : Type} -> {b : Type} ->
       {x : a} -> {y : a} -> {f : a -> b} ->
       x = y -> f x = f y
cong Refl = Refl
```

Exercise 3 (10 points)

Fixed-point expressions (5 points)

We showed factorial function as an example of valid expression of our statically checked lambda calculus. However, in order to achieve that we used Idris-level recursion (which might loop infinitely - try compiling it with totality checking enabled). Extend STLC and its interpreter to support *fixed-point operator* and use it to define factorial function (this will make interpreter not total but a man's gotta do what a man's gotta do).

Let expressions (5 points)

Extend STLC and its interpreter to support *let* expressions (non-recursive).

Exercise 4 (12 points)

Implement red-black trees (just insertion operation) and use dependent types to ensure that tree invariants are mantained. We are only interested in static checking of tree structural invariants (i.e. each path has the same number of black nodes and no red node has red child) so you may skip ordering correctness of tree labels.

You are free to choose the way to achieve that but I suggest the approach discussed in this presentation: http://www.cis.upenn.edu/~sweirich/talks/icfp14.pdf.