Homework 3: Dependently Typed Interpreter

Due No Due Date **Points** None

This homework is ungraded. Feel free to ask and answer questions via Canvas, but don't post complete solutions.

Consider the following simple expression language, whose syntax is given by a (plain) Haskell/Idris data type.

```
data Term = Num Nat | Plus Term Term | Equal Term Term | Cond Term Term Term
```

By contemplating different approaches to defining the denotational semantics for this language, the effect of dependent types will become clearer.

(a) The first goal is to define the denotational semantics for this language in Idris (or Haskell) by using the following domain.

```
data Value = N Nat | B Bool
```

To this end, define the Idris function:

```
sem : Term -> Value
```

Note that the recursive call of `sem` is expressed in Idris using the `with` notation. As an example, here is how the case for `Plus` would look like. (In Haskell we would use a `case` expression.)

```
sem (Plus t u) with (sem t,sem u)
  | (N n,N m) = N (n+m)
```

Basically, this means to recursively compute `sem` for the two argument terms and then to pattern-match on the resulting value (which is a pair of `Value` elements).

Obviously, this domain is insufficient, since it can't represent errors. Therefore, the definition is necessarily partial. If you put the `%default total` directive into your module, Idris will complain about the partiality of the definition. To make the definition total, we can temporarily employ the following "hack". (This will be needed for other cases as well, and we will later get rid of it.)

```
error : Value
error = N 999

sem (Plus t u) with (sem t,sem u)
  | (N n,N m) = N (n+m)
  | _         = error
```

Play around with the definition and test it on these values.

```
n : Term Nat
n = Num 3 `Plus` Num 5

b : Term Bool
b = Equal (Num 8) n

c : Term Bool
c = Equal b b

err : Term Nat
err = n `Plus` b
```

(b) Re-implement `sem` by using the following GADT representation of the language. Note that `Term` is now a type constructor, parameterized by a `Ty` value, which is used to indicate the result type of the `Term` expression.

```
data Ty = Nat | Bool

data Term : Ty -> Type where
  Num   : Nat -> Term Nat
  Plus  : Term Nat -> Term Nat -> Term Nat
  Equal : Term t -> Term t -> Term Bool
  Cond  : Term Bool -> Term t -> Term t -> Term t
```

The `Term` type is now indexed by `Ty`, and the set of all terms is partitioned into `Term` expressions of type `Nat` and `Term` expressions of type `Bool`. Note that `Nat` and `Bool` in the definition of `Ty` are ***(data) constructors*** and ***NOT types***. If you find this confusing, you can use the following definition instead.

```
data Ty = TyNat | TyBool
```

The function `sem` will have the following type.

```
sem : Term t -> Value
```

Although you cannot anymore construct type-incorrect `Terms` (`err` will not compile and must be commented out), you still need the error value in the implementation of `sem`. Try to remove those cases, and it will lead Idris to complain about the function not being total.

Why is that?

(c) Now let's turn the `Value` type also into a dependent type.

```
data Value : Ty -> Type where
  N : Nat  -> Value Nat
  B : Bool -> Value Bool
```

Make sure you understand that the two occurrences of `Nat` in the type of the `N` constructor represent two very different things. What are they? (Same for `Bool` in the type of the `B` constructor.) Finally, define a version of `sem` that uses the dependent versions of both, `Term` and `Value`. Now you can remove the "error" cases, and Idris will successfully compile the program.

Why is that?