1 Underyling Structures

The assignment specification gives the underlying data constructors listed here.

```
data Term = Tru |
   Fls |
   Zero |
   If Term Term Term |
   Succ Term |
   Pred Term |
   IsZero Term
   deriving Eq
```

2 Showing the data

The basic schema here is that values are lower case and keywords of functions and operators are listed as upper case. I made use Haskell's pattern matching to differentiate how the show function works on each data constructor found above

```
instance Show Term where
   show Tru = "true"
   show Fls = "false"
   show Zero = "zero"
   show (If t1 t2 t3) = "IF " ++ show t1 ++
      " THEN " ++ show t2 ++
      " ELSE " ++ show t3
   show (Succ t) = "SUCC " ++ show t
   show (Pred t) = "PRED " ++ show t
   show (IsZero t) = "ISZERO " ++ show t
```

3 Determining Values

Values in the the languages of Booleans and Natural numbers are any of True, False, and Numbers themselves. Therefore, any data constructor that corresponds to a boolean value or natural number returns true and anything else is false.

```
isValue :: Term → Bool
isValue Fls = True
isValue Tru = True
isValue t = isNumericValue t
```

Numeric values are either Zero or Succ applied to another numeric value. If the parameter is neither or is Succ applied to a non-numeric value it will return False.

```
isNumericValue :: Term → Bool
isNumericValue Zero = True
```

$isNumericValue\ (Succ\ t) = isNumericValue\ t$
$isNumericValue\ \_ = False$

4 Single Evaluation Statements

4.1 Values

Evaluation in one step for any value in the language is the value itself since there are no rewrite rules for values.

$eval1 :: Term \rightarrow Maybe\ Term$
$eval1\ Tru = Just\ Tru$
$eval1\ Fls = Just\ Fls$
$eval1\ Zero = Just\ Zero$
$eval1\ s@(Succ\ \_) = Just\ s$

4.2 Pred rewrites

Rewrite rules for one step of evaluation for Pred are as:

1. Pred applied to Zero is just Zero

   $eval1\ (Pred\ Zero) = Just\ Zero$

2. Pred applied to Succ of a value is the value

   $eval1\ (Pred\ (Succ\ t)) = Just\ t$

3. Pred applied to another term is the term rewritten with any rewrite rules for that term applied. This makes use of the monadic properties of the *Just* monad to propogate Nothing values if the evaluation of the term t evaluates to Nothing.

   $eval1\ (Pred\ t) = eval1\ t \gg\!= (Just \circ Pred)$

4.3 IsZero rewrites

Evaluation rules for one step of the IsZero term are:

1. IsZero applied to the Zero term evaluates to Tru.

   $eval1\ (IsZero\ Zero) = Just\ Tru$

2. IsZero applied to a non-zero term evaluates to Fls.

   $eval1\ (IsZero\ (Succ\ t)) = Just\ Fls$

3. IsZero applied to a term that can still be rewritten is IsZero applied to the rewritten term. This makes use of the monadic properties of the *Just* monad to propogate Nothing values if the evaluation of the term t evaluates to Nothing.

   $eval1\ (IsZero\ t) = eval1\ t \gg\!= (Just \circ IsZero)$

4.4 If-Then-Else rewrites

Evaluation rules for one step of the If-Then-Else term are:

1. If the boolean part is Tru, return the term in the *then* position

   ```
   eval1 (If Tru t2 _) = Just t2
   ```

2. If the boolean part is Fls, return the term in the *else* position

   ```
   eval1 (If Fls _ t3) = Just t3
   ```

3. However, if the boolean term has not been evaluated, evaluate it and return the proper term if statement with the corresponding boolean value or Nothing if the boolean term does not evaluate to a boolean value.

   ```
   eval1 (If t1 t2 t3) = case eval1 t1 of
     Just Tru → Just (If Tru t2 t3)
     Just Fls → Just (If Fls t2 t3)
     Just _ → Nothing
     Nothing → Nothing
   ```

5 Full Evaluation

Full evaluation of a term is the repetitive application of eval1 until no more rewrites can occur or until the term evaluates into value.

```
eval :: Term → Term
eval t = case eval1 t of
  Just t1 → if isValue t1
    then t1
    else eval t1
    Nothing → t
```