CPSC 449: Assignment 3

Fall 2020

Consult the D2L site for the due date/time

1. [40%] A *well-formed formula (wff)* of Propositional Logic is one of the following:

   - a propositional variable (with a name of type **`String`**),
   - the negation of a wff (i.e., not),
   - the conjunction of two wffs (i.e., and), or
   - the disjunction of two wffs (i.e., or).

   Except for the above there are no other wffs.

   (a) [10%] Declare a Haskell algebraic type **`Formula`** to represent wffs.

   (b) [5%] Give a Haskell expression that constructs a **`Formula`** representation of the formula "$\neg(\neg p \wedge \neg q)$"

   (c) [5%] Define a function **`showFormula :: Formula -> String`** such that **`(showFormula f)`** returns a string representation of the formula **`f`**. You need this function for debugging the following part. **Hint:** Your implementation shall be primitively recursive. There is no need to do factorization or simplification. Print parentheses to disambiguate syntax.

   (d) [20%] A wff p is in ***Negation Normal Form (NNF)*** if and only if every negation occurring in p is applied to a propositional variable. For example, the following wffs are in NNF (x, y, and z are propositional variables):

   $$\neg x \qquad (x \wedge \neg y) \vee \neg z$$

   The following wffs are *not* in NNF:

   $$(\neg\neg x) \wedge \neg y \qquad x \wedge \neg(y \vee \neg z)$$

   It is a well-known fact in Propositional Logic that every wff can be converted to an equivalent wff in NNF. The idea is to apply the following rewriting rules repeatedly, until no more rewriting can be performed:

   $$\begin{aligned} \neg\neg p &\equiv p \\ \neg(p \wedge q) &\equiv (\neg p) \vee (\neg q) \\ \neg(p \vee q) &\equiv (\neg p) \wedge (\neg q) \end{aligned}$$

   Develop a Haskell function

```
rewrite :: Formula -> Formula
```

such that `(rewrite f)` returns a wff `f'` such that (i) `f'` is logically equivalent to `f`, and (ii) `f'` is in NNF. **Hint:** The function `rewrite` requires general recursion. While this assignment does not ask you to devise a termination proof, it is to your best interest to keep such a proof in the back of your mind when you formulate the general recursion. Again, there is no need to submit a termination proof. Any such submission will *not* be graded.

2. *[20%]* A ***polynomial with one variable*** is represented using the following algebraic type.

```
data Polynomial = PConst Integer |
                  PVar |
                  PAdd Polynomial Polynomial |
                  PMul Polynomial Polynomial
```

The ***degree*** of a `Polynomial` can be obtained by the function `degree :: Polynomial -> Integer`, which is defined by the following equations.

```
degree (PConst n)   = 0                                  (degree.1)
degree PVar         = 1                                  (degree.2)
degree (PAdd p1 p2) = max (degree p1) (degree p2)        (degree.3)
degree (PMul p1 p2) = (degree p1) + (degree p2)          (degree.4)
```

The ***first derivative*** of a `Polynomial` can be computed using the function `d :: Polynomial -> Polynomial`, the definition of which is given by the following equations.

```
d (PConst n)   = PConst 0                                (d.1)
d PVar         = PConst 1                                (d.2)
d (PAdd p1 p2) = PAdd (d p1) (d p2)                      (d.3)
d (PMul p1 p2) = PAdd (PMul p1 (d p2))
                      (PMul (d p1) p2)                   (d.4)
```

These four equations are the direct encoding of the well-known formulas in standard calculus textbooks.

$$\frac{dC}{dx} = 0 \qquad \frac{dx}{dx} = 1 \qquad \frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx} \qquad \frac{d(u \times v)}{dx} = u\frac{dv}{dx} + v\frac{du}{dx}$$

Prove, by structural induction, that the following inequality holds for all finite `Polynomial` p.

$$\texttt{degree}\ p \geq \texttt{degree (d}\ p\texttt{)} \tag{1}$$

Your demonstration shall follow the steps below.

(a) *[6%]* State the Principle of Structural Induction for `Polynomial`s.

(b) *[4%]* Prove the base case(s).

(c) *[10%]* Prove the induction step(s).

3. *[20%]* Use **`map`**, **`filter`**, and/or **`foldr`**/**`foldr1`** to implement the following Haskell functions. You shall demonstrate the use of anonymous functions (i.e., lambda abstractions) in the implementation of at least one of following functions.

   (a) *[6%]* **`lastElm :: [a] -> a`**. The function returns the last element of the list argument.

   (b) *[6%]* **`unanimous :: [a->Bool] -> a -> Bool`**. The function takes a list of predicates and an entity as arguments, then applies every predicate to the entity, and finally returns **`True`** if and only if every predicate in the list returns **`True`**. In the degenerate case, when the list argument is empty, then **`True`** is returned.

   (c) *[8%]* **`selectiveMap :: (a->Bool) -> (a->b) -> [a] -> [b]`**. The function takes three arguments: (i) a predicate for type-**`a`** values, (ii) a function that transforms a type-**`a`** value to a type-**`b`** value, and (iii) a list of type-**`a`** values. The function **`selectiveMap`** tests every element of the list argument using the given predicate. Those elements that satisfy the predicate will be transformed by the function argument. The transformed elements are then collected into a list, which is returned as the value of **`selectiveMap`**.

4. (a) [14%] **[Thompson]** exercise 10.9.

   (b) [6%] **[Thompson]** exercise 10.10. Name and type your function as follows.

   ```
   powerOfTwo :: Integer -> Integer
   ```