Logic
Informatics 1 – Functional Programming: Tutorial 5

Due: The tutorial of week 7 (3/4 Nov.)

Please attempt the entire worksheet in advance of the tutorial, and bring with you all work, including (if a computer is involved) printouts of code and test results. Tutorials cannot function properly unless you do the work in advance.

You may work with others, but you must understand the work; you can't phone a friend during the exam.

Assessment is formative, meaning that marks from coursework do not contribute to the final mark. But coursework is not optional. If you do not do the coursework you are unlikely to pass the exams.

Attendance at tutorials is obligatory; please let your tutor know if you cannot attend.

Warmup

First you we will write some functions to act on input of the user-defined type Fruit. In the file `tutorial5.hs` you will find the following data declaration:

```
data Fruit = Apple String Bool
           | Orange String Int
```

An expression of type `Fruit` is either an `Apple String Bool` or an `Orange String Int`. We use a `String` to indicate the variety of the apple or orange, a `Bool` to describe whether an apple has a worm and an `Int` to count the number of segments in an orange. For example:

```
Apple "Granny Smith" False  -- a Granny Smith apple with no worm
Apple "Braeburn" True       -- a Braeburn apple with a worm
Orange "Sanguinello" 10     -- a Sanguinello orange with 10 segments
```

Exercises

1. Write a function `isBloodOrange :: Fruit -> Bool` which returns `True` for blood oranges and `False` for apples and other oranges. Blood orange varieties are: Tarocco, Moro and Sanguinello. For example:

   ```
   isBloodOrange(Orange "Moro" 12) == True
   isBloodOrange(Apple "Granny Smith" True) == False
   ```

2. Write a function `bloodOrangeSegments :: [Fruit] -> Int` which returns the total number of blood orange segments in a list of fruit.
3. Write a function `worms :: [Fruit] -> Int` which returns the number of apples that contain worms.

Logic

In the rest of this tutorial we will implement propositional logic in Haskell. In the file `tutorial5.hs` you will find the following type and data declarations:

```
type Name = String
data Prop = Var Name
          | F
          | T
          | Not Prop
          | Prop :|: Prop
          | Prop :&: Prop
```

The type `Prop` is a representation of propositional formulas. Propositional variables such as P and Q can be represented as `Var "P"` and `Var "Q"`. Furthermore, we have the Boolean constants `T` and `F` for 'true' and 'false', the unary connective `Not` for negation (not to be confused with the function `not :: Bool -> Bool`), and (infix) binary connectives `:|:` and `:&:` for disjunction ($\vee$) and conjunction (&). Another type defined by `tutorial5.hs` is:

```
type Env = [(Name, Bool)]
```

The type `Env` is used as an 'environment' in which to evaluate a proposition: it is a list of truth assignments for (the names of) propositional variables. Using these types, `tutorial5.hs` defines the following functions:

- `satisfiable :: Prop -> Bool` checks whether a formula is satisfiable — that is, whether there is some assignment of truth values to the variables in the formula that will make the whole formula true.

  ```
  *Main> satisfiable (Var "P" :&: Not (Var "P"))
  False
  *Main> satisfiable ((Var "P" :&: Not (Var "Q")) :&: (Var "Q" :|: Var "P"))
  True
  ```

- `eval :: Env -> Prop -> Bool` evaluates the given proposition in the given environment (assignment of truth values). For example:

  ```
  *Main> eval [("P", True), ("Q", False)] (Var "P" :|: Var "Q")
  True
  ```

- `showProp :: Prop -> String` converts a proposition into a readable string approximating the mathematical notation. For example:

  ```
  *Main> showProp (Not (Var "P") :&: Var "Q")
  "((~P)&Q)"
  ```

- `names :: Prop -> Names` returns all the variable names used in a proposition. Example:

  ```
  *Main> names (Not (Var "P") :&: Var "Q")
  ["P", "Q"]
  ```

- `envs :: Names -> [Env]` generates a list of all the possible truth assignments for the given list of variables. Example:

```
*Main> envs ["P", "Q"]
[[("P",False),("Q",False)],
 [("P",False),("Q",True)],
 [("P",True),("Q",False)],
 [("P",True),("Q",True)]   ]
```

- `table :: Prop -> IO ()` prints out a truth table.

```
*Main> table ((Var "P" :&: Not (Var "Q")) :&: (Var "Q" :|: Var "P"))
P Q | ((P&(~Q))&(Q|P))
- - | ----------------
F F |        F
F T |        F
T F |        T
T T |        F
```

- `fullTable :: Prop -> IO ()` prints out a truth table that includes the evaluation of the subformulas of the given proposition. (**Note:** `fullTable` uses the function `subformulas` that you will define in Exercise 8, so it doesn't work just yet.)

```
*Main> fullTable ((Var "P" :&: Not (Var "Q")) :&: (Var "Q" :|: Var "P"))
P Q | ((P&(~Q))&(Q|P)) (P&(~Q)) (~Q) (Q|P)
- - | ---------------- -------- ---- -----
F F |        F             F      T     F
F T |        F             F      F     T
T F |        T             T      T     T
T T |        F             F      F     T
```

Exercises

4. Write the following formulas as `Props` (call them `p1`, `p2` and `p3`). Then use `satisfiable` to check their satisfiability and `table` to print their truth tables.

   (a) $((P \vee Q) \,\&\, (P \,\&\, Q))$

   (b) $((P \vee Q) \,\&\, ((\neg P) \,\&\, (\neg Q)))$

   (c) $((P \,\&\, (Q \vee R)) \,\&\, (((\neg P) \vee (\neg Q)) \,\&\, ((\neg P) \vee (\neg R))))$

5. (a) A proposition is a tautology if it is always true, i.e. in every possible environment. Using `names`, `envs` and `eval`, write a function `tautology :: Prop -> Bool` which checks whether the given proposition is a tautology. Test it on the examples from Exercise (4) and on their negations.

   (b) Create two QuickCheck tests to verify that `tautology` is working correctly. Use the following facts as the basis for your test properties:

   For any property P,

   i. either P is a tautology, or $\neg P$ is satisfiable,

   ii. either P is not satisfiable, or $\neg P$ is not a tautology.

   Note: be careful to distinguish the negation for `Bools` (`not`) from that for `Props` (`Not`).

6. We will extend the datatype and functions for propositions in `tutorial5.hs` to handle the connectives $\to$ (implication) and $\leftrightarrow$ (bi-implication, or 'if and only if'). They will be implemented as the constructors `:->:` and `:<->:`. After you have implemented them, the truth tables for both should be as follows:

```
*Main> table (Var "P" :->: Var "Q")
P Q | (P->Q)
- - | ------
F F |   T
F T |   T
T F |   F
T T |   T
```

```
*Main> table (Var "P" :<->: Var "Q")
P Q | (P<->Q)
- - | -------
F F |    T
F T |    F
T F |    F
T T |    T
```

(a) Find the declaration of the datatype `Prop` in `tutorial5.hs` and extend it with the infix constructors `:->:` and `:<->:`.

(b) Find the printer (`showProp`), evaluator (`eval`), and name-extractor (`names`) functions and extend their definitions to cover the new constructors `:->:` and `:<->:`. Test your definitions by printing out the truth tables above.

(c) Define the following formulas as `Prop`s (call them `p4`, `p5`, and `p6`). Check their satisfiability and print their truth tables.

   i. $((P \to Q) \,\&\, (P \leftrightarrow Q))$
   ii. $((P \to Q) \,\&\, (P \,\&\, (\neg Q)))$
   iii. $((P \leftrightarrow Q) \,\&\, ((P \,\&\, (\neg Q)) \vee ((\neg P) \,\&\, Q)))$

(d) Below the 'exercises' section of `tutorial5.hs`, in the section called 'for QuickCheck', you can find a declaration that starts with:

```
instance Arbitrary Prop where
```

This tells QuickCheck how to generate arbitrary `Props` to conduct its tests. To make QuickCheck use the new constructors, uncomment the two lines in the middle of the definition:

```
-- , liftM2 (:->:) subform subform
-- , liftM2 (:<->:) subform' subform'
```

Now try your test properties from Exercise (5b) again.

7. Two formulas are *equivalent* if they always have the same truth values, regardless of the values of their propositional variables. In other words, formulas are equivalent if in any given environment they are either both true or both false.

   (a) Write a function `equivalent :: Prop -> Prop -> Bool` that returns `True` just when the two propositions are equivalent in this sense. For example:

```
*Main> equivalent (Var "P" :&: Var "Q") (Not (Not (Var "P") :|: Not (Var "Q")))
True
*Main> equivalent (Var "P") (Var "Q")
False
*Main> equivalent (Var "R" :|: Not (Var "R")) (Var "Q" :|: Not (Var "Q"))
True
```

   You can use `names` and `envs` to generate all relevant environments, and use `eval` to evaluate the two `Props`.

   (b) Write another version of equivalent, this time by combining the two arguments into a larger proposition and using `tautology` or `satisfiable` to evaluate it.

   (c) Write a QuickCheck test property to verify that the two versions of `equivalent` are equivalent.

The *subformulas* of a proposition are defined as follows:

- A propositional letter P or a constant **t** or **f** has itself as its only subformula.

- A proposition of the form $\neg P$ has as subfomulas itself and all the subformulas of P.
- A proposition of the form $P \,\&\, Q$, $P \vee Q$, $P \to Q$, or $P \leftrightarrow Q$ has as subformulas itself and all the subformulas of P and Q.

The function `fullTable :: Prop -> IO ()`, already defined in `tutorial5.hs`, prints out a truth table for a formula, with a column for each of its non-trivial subformulas.

Exercises

8. Add a definition for the function `subformulas :: Prop -> [Prop]` that returns all of the subformulas of a formula. For example:

```
*Main> map showProp (subformulas p2)
["((P|Q)&((~P)&(~Q)))","(P|Q)","P","Q","((~P)&(~Q))","(~P)","(~Q)"]
```

   (We need to use `map showProp` here in order to convert each proposition into a string; otherwise we could not easily view the results.)

   Test out `subformulas` and `fullTable` on each of the `Prop`s you defined earlier (`p1`–`p6`).

Optional material

Normal Forms

In this part of the tutorial we will put propositional formulas into several different normal forms. First, we will deal with negation normal form. As a reminder, a formula is in negation normal form if it consists of just the connectives $\vee$ and $\&$, unnegated propositional variables P and negated propositional variables $\neg P$, and the constants **t** and **f**. Thus, negation is only applied to propositional variables, and nothing else.

To transform a formula into negation normal form, you might want to use the following equivalences:

$$\begin{array}{rcl}
\neg(P \,\&\, Q) & \Leftrightarrow & (\neg P) \vee (\neg Q) \\
\neg(P \vee Q) & \Leftrightarrow & (\neg P) \,\&\, (\neg Q) \\
(P \to Q) & \Leftrightarrow & (\neg P) \vee Q \\
(P \leftrightarrow Q) & \Leftrightarrow & (P \to Q) \,\&\, (Q \to P) \\
\neg(\neg P) & \Leftrightarrow & P
\end{array}$$

Exercises

9. Write a function `isNNF` to test whether a `Prop` is in negation normal form.
10. Write a function `toNNF` that puts an arbitrary `Prop` into negation normal form. Use the test properties `prop_NNF1` and `prop_NNF2` to verify that your function is correct. **Hint:** don't be alarmed if you need many case distinctions.

Next, we will turn a formula into conjunctive normal form. This means the formula is a conjunction of clauses, and a clause is a disjunction of (possibly negated) propositional variables, called *atoms*.

You will need to pay special attention to the constants **t** and **f**. The `Prop`s `T` and `F` themselves are considered to be in conjunction normal form, but otherwise they should not occur in formulas in normal form. They can be eliminated using the following equivalences:

$$\begin{array}{rcccl}
(P \,\&\, \mathbf{t}) & \Leftrightarrow & (\mathbf{t} \,\&\, P) & \Leftrightarrow & P \\
(P \,\&\, \mathbf{f}) & \Leftrightarrow & (\mathbf{f} \,\&\, P) & \Leftrightarrow & \mathbf{f} \\
(P \vee \mathbf{t}) & \Leftrightarrow & (\mathbf{t} \vee P) & \Leftrightarrow & \mathbf{t} \\
(P \vee \mathbf{f}) & \Leftrightarrow & (\mathbf{f} \vee P) & \Leftrightarrow & P
\end{array}$$

Exercises

11. Write a function `isCNF` to test if a `Prop` is in conjunction normal form.

A common way of writing formulas in conjunctive normal form is as a list of lists, where the inner lists represent the clauses. Thus:

$$((A \vee B) \,\&\, ((C \vee D) \vee E)) \,\&\, G \quad \Leftrightarrow \quad \texttt{[[A,B],[C,D,E],[G]]}$$

Exercises

12. Think of how the constants `t` and `f` can be represented as lists of lists.

    Hint: a formula in conjunction normal form is true when *all* its clauses are true. A clause is true if *any* of its atoms is true.
13. Write a function `listsToCNF` to translate a list of lists of `Props` (which you may assume to be variables or negated variables) to a `Prop` in conjunction normal form.
14. Write a function `listsFromCNF` to write a formula in conjunction normal form as a list of lists.

Finally, we will convert an arbitrary `Prop` to a list of lists. You can use the following distributive law (check it first using your previous code):

$$P \vee (Q \,\&\, R) \quad \Leftrightarrow \quad (P \vee Q) \,\&\, (P \vee R)$$

Or, in a more generalized version:

$$(P_1 \,\&\, P_2 \,\&\, \ldots \,\&\, P_m) \vee (Q_1 \,\&\, Q_2 \,\&\, \ldots \,\&\, Q_n)$$

$$\Updownarrow$$

$$\begin{array}{c}
(P_1 \vee Q_1) \,\&\, (P_1 \vee Q_2) \,\&\, (P_1 \vee Q_3) \,\&\, \ldots \,\&\, (P_1 \vee Q_n) \,\&\\
(P_2 \vee Q_1) \,\&\, (P_2 \vee Q_2) \,\&\, (P_2 \vee Q_3) \,\&\, \ldots \,\&\, (P_2 \vee Q_n) \,\&\\
\vdots\\
(P_m \vee Q_1) \,\&\, (P_m \vee Q_2) \,\&\, (P_m \vee Q_3) \,\&\, \ldots \,\&\, (P_m \vee Q_n)
\end{array}$$

Exercises

15. Write a function `toCNFList` that turns a `Prop` into a list of lists of propositional variables and their negations, representing the formula in conjunction normal form. The output of `toCNFList` may contain empty clauses as long as `toCNF` produces a formula without `T` nor `F` as strict subformulas.

Note: transforming to conjunctive normal form is computationally expensive, especially for formulas with many bi-implications ($\leftrightarrow$). Be sure to test your code on small examples first before trying the test property `prop_CNF` with QuickCheck.