Programming Languages Assignment 2

Team Members :
1) Kunal Patil   kpatil2@binghamton.edu   B00533154
2) Abhijit Jachak   ajachak1@binghamton.edu   B00524304

---

1) Write a Haskell function **sublist lt** that computes all sublists of a list lt.
e.g. >sublist [1,2,3]
[[],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3]]

Answer -

Program written for Haskell function sublist lt is

-- Program--

```
sublist [] = [[]]
sublist (x:xs) = [x:tempList | tempList <- sublist xs] ++ sublist xs
```

Output for following program is

```
*Main> sublist [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

---

2) Define a function **replic lt** that replicates each element in **lt** into a list. If the element is in the kth position of **lt**, the resulting list contains k copies of the same element. **You must define this function using the higher-order function of "map"**
e.g. >replic [2,3,4,7,6]
[[2],[3,3],[4,4,4], [7,7,7,7], [6,6,6,6,6]]
Answer -

Program written for Haskell function replic lt is

-- Program--

```
replic = replicFunction . map return where
replicFunction [] = []
 replicFunction (x:xs) = x : replicFunction (map (\(x:xs) -> x:x:xs) xs)
```

Output for following program is

```
*Main> replic [1,2,3]
[[1],[2,2],[3,3,3]]
*Main> replic [2,3,4,7,6]
[[2],[3,3],[4,4,4],[7,7,7,7],[6,6,6,6,6]]
```

---

3) In class, we introduced function "reverse" to reverse the elements of a list, and function "head" as returning the first element of the list. Define a function "laste" to return the last element of a non-empty list based on "reverse", "head" and possible function composition operator (.). For instance
e.g. >laste [2,3,4,7,6]
6
Answer -

Program written for Haskell function laste lt is

```
laste = head.reverse
```

It satisfies requirement of "reverse","head" and function composition operator(.)

4) mystery n m
| n == m = [n]
| n < m = n:(mystery (n+1) m)
| n > m = n:(mystery (n-1) m)

Answer -

Above function mystery takes two integers n and m and creates a list recursively from n to m. Function checks three conditions if

1. n==m

in n==m then there is only one element in the list so function prints list containing [n]. For example,

*Main> mystery 5 5
[5]

As 5 == 5 is true, list containing [5] will be printed

2. n < m

When n < m then function will recursively call mystery function with (n+1) m and will add (n+1) to the list. That is if n and m are 5 and 10 respectively then, mystery function will be called for
mystery 6 10 [n] = [5, 6]
mystery 7 10 [n] = [5, 6, 7]
mystery 8 10 [n] = [5, 6,7, 8]
mystery 9 10 [n] = [5, 6, 7, 8, 9]
mystery 10 10 - At this point n==m hence [n] will be printed which will contain elements [5, 6, 7, 8, 9, 10]

3. n > m

When n > m then function will recursively call mystery function with (n-1) m and will add (n-1) to the list. That is if n and m are 5 and 1 respectively then, mystery function will be called for
mystery 4 1 [n] = [5, 4]
mystery 3 1 [n] = [5, 4, 3]
mystery 2 1 [n] = [5, 4,3, 2]
mystery 1 1 - At this point n==m hence [n] will be printed which will contain elements [5, 4, 3, 2, 1].

Thus mystery function is creating a range between n and m.

---

5) Given a Haskell program

let fun x = x + 1 in fun 3

Please encode the program into an expression so that “let” is not used at all.

Answer

foldr1(+)[1,(\x->x)3]

This command do not use "let". It is neither as simple as 4, 3+1, (\x -> x+1) 3 nor as complex as " let y = 1 in let fun x = x+y in fun 3".
Output when run from command line is as follows -

```
*Main> foldr1(+)[1,(\x->x)3]
4
```

6)

Given the following definition of the propositional formula:

```
data Formula
      = Atom Bool                          -- atomic formula
      | And Formula Formula                -- f /\ f
      | Or Formula Formula                 -- f \/ f
      | Not Formula                        -- not(f)
```

(1) Write a Haskell function **collect_atoms f** that computes all atomic formulas of a propositional formula f.
e.g. >collect_atoms (And (Or (Atom True) (Atom False)) (Not (Atom False)))
[Atom True, Atom False, Atom False]
(Note that your hug environment may not display the atomic formula as above if the display of a Formula (i.e. "show") is not defined. You do not need to worry about it. It is OK as long as your "collect_atoms" indeed returns a list as above.)
(2) Write a Haskell function **eval f** to evaluate term f according to standard definitions of propositional logic.
e.g. >eval (And (Or(Atom True) (Atom False)) (Not (Atom False)))
True

Answer -

Given -

```
data Formula
      = Atom Bool                          -- atomic formula
      | And Formula Formula                -- f /\ f
      | Or Formula Formula                 -- f \/ f
      | Not Formula                        -- not(f)
```

1.
In order to display results for collect_atoms f, we need to write a function called show Formula to display final results. Following piece of code writes collect_atoms f output.

```
instance Show Formula where
        show (Atom atomA) = "Atom " ++ show atomA
        show (And atomA atomB) = "And (" ++ show atomA ++ ") (" ++ show atomB ++ ")"
        show (Or atomA atomB) = "Or (" ++ show atomA ++ ") (" ++ show atomB ++ ")"
        show (Not atomA) = "Not (" ++ show atomA ++ ")"
```

Below Haskell function **collect_atoms f** that computes all atomic formulas of a propositional formula f

```
collect_atoms (Atom atom) = [Atom atom]
collect_atoms (And atomA atomB) = collect_atoms atomA ++ collect_atoms atomB
collect_atoms (Or atomA atomB) = collect_atoms atomA ++ collect_atoms atomB
collect_atoms (Not atomA) = collect_atoms atomA
```

Input :

```
collect_atoms (And (Or (Atom True) (Atom False)) (Not (Atom False)) )
```

Output is as follows :

```
*Main> collect_atoms (And (Or (Atom True) (Atom False)) (Not (Atom True)) )
[Atom True,Atom False,Atom True]
*Main>
```

2. Below Haskell function **eval f** to evaluate term f according to standard definitions of propositional logic.

```
eval (Atom atomA) = atomA
eval (And atomA atomB) = eval atomA && eval atomB
eval (Or atomA atomB) = eval atomA || eval atomB
eval (Not atomA) = not (eval atomA)
```

Input :

```
eval (And (Or (Atom True) (Atom False)) (Not (Atom False)) )
```

Output is as follows :

```
*Main> eval (And (Or (Atom True) (Atom False)) (Not (Atom False)) )
True
*Main>
```

Following assignment is tested successfully on **bingsuns**, Here is the screenshot of output of **assignment2.hs** file from **bingsuns** terminal.

Questions 1,2,3,5,6 are executed one by one.

```
bingsuns2% hugs
__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: September 2006 _________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs> :l assignment2.hs
Main> sublist [1,2,3]
[[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]
Main> replic [2,3,4,7,6]
[[2],[3,3],[4,4,4],[7,7,7,7],[6,6,6,6,6]]
Main> laste [2,3,4,7,6]
6
Main> foldr1(+)[1,(\x->x)3]
4
Main> collect_atoms (And (Or (Atom True) (Atom False)) (Not (Atom False)) )
[Atom True,Atom False,Atom False]
Main> eval (And (Or(Atom True) (Atom False)) (Not (Atom False)) )
True
Main> :quit
[Leaving Hugs]
bingsuns2%
```