INFR08013 INFORMATICS 1 - FUNCTIONAL PROGRAMMING

Thursday 22\textsuperscript{nd} August 2013

14:30 to 16:30

INSTRUCTIONS TO CANDIDATES

1. Note that ALL QUESTIONS ARE COMPULSORY.

2. DIFFERENT QUESTIONS MAY HAVE DIFFERENT NUMBERS OF TOTAL MARKS. Take note of this in allocating time to questions.

3. This is an OPEN BOOK examination.

Convener: J Bradfield
External Examiner: A Preece

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY
1. (a) Write a function \( f :: [(\text{Int},\text{Int})] \rightarrow [\text{Int}] \) that takes a list of pairs of integers and returns a list containing the first element from each of the pairs in even-numbered positions and the second element from each of the pairs in odd-numbered positions, where numbering of list elements begins from 0. For example:

\[
\begin{align*}
   f [(1,2),(5,7),(3,8),(4,9)] &= [1,7,3,9] \\
   f [(1,2)] &= [1] \\
   f [] &= []
\end{align*}
\]

Use \textit{basic functions}, \textit{list comprehension}, and \textit{library functions}, but not recursion. Credit may be given for indicating how you have tested your function.

(12 marks)

(b) Write a second function \( g :: [(\text{Int},\text{Int})] \rightarrow [\text{Int}] \) that behaves like \( f \), this time using \textit{basic functions} and \textit{recursion}, but not list comprehension or other library functions. Credit may be given for indicating how you have tested your function.

(12 marks)
2. (a) Write a function \( p :: [\text{Int}] \rightarrow \text{Int} \) that computes the product of the results of multiplying each of the positive odd numbers in a list by three. If the list is empty or there are no positive odd numbers in the list, the function should produce 1. For example:

\[
\begin{align*}
p \ [1,6,-15,11,-9] &= 3 \times 1 \times 3 \times 11 = 99 \\
p \ [3,6,9,12,-9,9] &= 3 \times 3 \times 3 \times 9 \times 3 \times 9 = 6561 \\
p \ [] &= 1 \\
p \ [-1,4,-15] &= 1
\end{align*}
\]

Use basic functions, list comprehension, and library functions, but not recursion. Credit may be given for indicating how you have tested your function. [16 marks]

(b) Write a second function \( q :: [\text{Int}] \rightarrow \text{Int} \) that behaves like \( p \), this time using basic functions and recursion, but not list comprehension or library functions. Credit may be given for indicating how you have tested your function. [16 marks]

(c) Write a third function \( r :: [\text{Int}] \rightarrow \text{Int} \) that also behaves like \( p \), this time using the following higher-order library functions:

\[
\begin{align*}
\text{map} & \quad :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{filter} & \quad :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\
\text{foldr} & \quad :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b
\end{align*}
\]

Do not use recursion or list comprehension. Credit may be given for indicating how you have tested your function. [12 marks]
3. The following data type represents propositional formulas built from a single variable (X), constants true (T) and false (F), negation (Not) and bi-implication, also known as logical equivalence (↔, written using infix :<->:):

```
data Prop = X
    | F
    | T
    | Not Prop
    | Prop :<->: Prop
```

The template file includes a function (showProp :: Prop -> String) which converts formulas into a readable format and code that enables QuickCheck to generate arbitrary values of type Prop, to aid testing.

(a) Write a function eval :: Prop -> Bool -> Bool, which given a propositional formula and the value of the variable X returns the value of the formula. Recall the truth tables for negation and bi-implication:

<table>
<thead>
<tr>
<th>P</th>
<th>Not P</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P :&lt;-&gt;: Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

For example,

- `eval (Not T) True` = False
- `eval (Not X) False` = True
- `eval (Not X :<->: Not (Not X)) True` = False
- `eval (Not X :<->: Not (Not X)) False` = False
- `eval (Not (Not X :<->: F)) True` = False
- `eval (Not (Not X :<->: F)) False` = True

Credit may be given for indicating how you have tested your function. [16 marks]

(b) Write a function simplify :: Prop -> Prop that converts a propositional formula to an equivalent simpler formula by repeated use of the following laws:

- `Not T = F`
- `Not F = T`
- `Not (Not p) = p`
- `T :<->: p = p`
- `F :<->: p = Not p`
- `p :<->: T = p`
- `p :<->: F = Not p`
- `p :<->: p = T`

**QUESTION CONTINUES ON NEXT PAGE**
For example,

\[
\begin{align*}
\text{simplify (Not F)} & = T \\
\text{simplify (Not X <-> Not (X <-> T))} & = T \\
\text{simplify (Not (Not X <-> Not T))} & = \text{Not X} \\
\text{simplify (Not (F <-> Not (Not X)))} & = X
\end{align*}
\]

Credit may be given for indicating how you have tested your function. [16 marks]