Declarative Programming

Workshop exercises set 1 (for workshops in week 2).

QUESTION 1
What are the most annoying limitations of a programming language you have used? It would be good if you posted something on this topic to the LMS discussion forum.

QUESTION 2
What are some useful Haskell resources on the web?

QUESTION 3
What will be printed by this C code fragment?

```
        #include <stdio.h>
        int f(int x, int y)
        {
                return 10 * x + y;
        }

        int main(void)
        {
                int i, j;

                i = 0;
                j = f(++i, ++i);
                printf("%d\n", j);
                return 0;
        }
```

What does this show about the impact of side effects in C?

QUESTION 4
Fix the formatting errors (due the offside rule) in the following code, making the minimal possible changes.

```
zero = len []
one
 = len
      []
two = len [1,2] three = len [1,2,3]
four = len [1,
     2,3,
         4]
 len []     = 0
 len (x:xs) = 1 + len xs
```

QUESTION 5
Implement a function to perform the logical XOR (exclusive or) operation. The XOR of two truth values is true if exactly one of them is true. You may approach this using pattern matching or using other logical operations.

QUESTION 6
Implement a function to append two lists in Haskell (this is the ++ function in the standard prelude). What is the type of this function?

QUESTION 7
Implement your own version of the 'reverse' function included in the Haskell Prelude. Do not use the existing 'reverse' function in your implementation.

Note you should call your function 'myReverse' to avoid shadowing the
existing function.

QUESTION 8
Implement a function 'getNthElem' which takes an integer 'n' and a list, and
returns the nth element of the list.

Declarative Programming

Workshop exercises set 2.

QUESTION 1
Give a high level description (not programming language specific)
of at least five different possible representations of playing cards
from a standard 52 card deck. Describe the advantages and disadvantages
of each representation.

The standard 52 card deck has 13 cards in each of four suits.
The suits are clubs, diamonds, hearts and spades, and the 13 ranks in
each suit are the 2, 3, 4, 5, 6, 7, 8, 9, 10, jack, queen, king and ace.
In this question, we ignore jokers.

QUESTION 2
Define a Haskell type for representing "font" tags in HTML. A font tag
can specify zero or more of the following: the size in points (e.g. 10),
the face (e.g. "courier") and the colour. The colour can be described
using a colour name (e.g., "red"), a six-digit hexidecimal number
(e.g. #02EA1F) or a RGB triple of numbers (e.g. rgb(255,100,0)).

Note: the font tag is the most widely misused of all HTML tags,
and in fact it is fundamentally misconceived. The font should be up to
the VIEWER of the web page, not the web page DESIGNER; if the designer
selects a small font, people with bad eyesight looking at the page
won't be able to read it. This is why the font tag is actually deprecated,
which means it is slated to disappear in a future version of the HTML standard.

QUESTION 3
Implement a function 'factorial' that computes the factorial of a given
integer. Include a type declaration.

QUESTION 4
Implement a function 'myElem' which returns True if a given item is present
in a given list. Include a type declaration.

QUESTION 5
Implement a function 'longestPrefix' which returns the longest common prefix
of two lists. ie: When applied to "extras" and "extreme", the function
should return "extr".

QUESTION 6
Without necessarily understanding the code, translate the following
C function into Haskell.

```
int mccarthy_91(int n)
{
    int c = 1;

    while (c != 0) {
        if (n > 100) {
            n = n - 10;
            c--;
        } else {
            n = n + 11;
            c++;
        }
    }

    return n;
}
```

QUESTION 7
Write a Haskell function which takes two integers, min and max, and returns a list of integers from min to max, inclusive. Note there are two different strategies to solve this problem: we can build up the list from min to max or backwards, from max to min. How does your Haskell code compare with a version in an imperative language such as C, and how would you reason about the correctness of a C version?

Declarative Programming

Workshop exercises set 3.

QUESTION 1
If you were working on a program that functioned as a web server, and thus its output was in the form of web pages, you could:

(a) have the program write out each part of the page as soon as it has decided what it should be;
(b) have the program generate the output in the form of a string, and then print the string;
(c) have the program generate the output in the form of a representation such as the HTML type of the previous questions, and then convert that to a string and then print the string.

Which of these approaches would you choose, and why?

QUESTION 2
Implement a function ftoc :: Double -> Double, which converts a temperature in Fahrenheit to Celsius. Recall that C = (5/9) * (F - 32).
What is the inferred type of the function if you comment out the type declaration? What does this tell you?

QUESTION 3
Implement a function quadRoots :: Double -> Double -> Double -> [Double], which computes the roots of the quadratic equation defined by
0 = a*x^2 + b*x + c, given a, b, and c. See
http://en.wikipedia.org/wiki/Quadratic_formula for the formula.
What is the inferred type of the function if you comment out the type declaration? What does this tell you?

QUESTION 4
Write a Haskell function to merge two sorted lists into a single sorted list

QUESTION 5
Write a Haskell version of the classic quicksort algorithm for lists.
(Note that while quicksort is a good algorithm for sorting arrays, it is not actually that good an algorithm for sorting lists; variations of merge sort generally perform better. However, that fact has no bearing on this exercise.)

QUESTION 6
Given the following type definition for binary search trees from lectures,

```
>data Tree k v = Leaf | Node k v (Tree k v) (Tree k v)
>        deriving (Eq, Show)
```

define a function

```
>same_shape :: Tree a b -> Tree c d -> Bool
```

which returns True if the two trees have the same shape: same arrangement of nodes and leaves, but possibly different keys and values in the nodes.

QUESTION 7
Consider the following type definitions, which allow us to represent expressions containing integers, variables "a" and "b", and operators for addition, subtraction, multiplication and division.

```
>data Expression
>       = Var Variable
>       | Num Integer
>       | Plus Expression Expression
>       | Minus Expression Expression
>       | Times Expression Expression
>       | Div Expression Expression

>data Variable = A | B
```

For example, we can define exp1 to be a representation of 2*a + b as follows:

```
>exp1 = Plus (Times (Num 2) (Var A)) (Var B)
```

Write a function eval :: Integer -> Integer -> Expression -> Integer which takes the values of a and b and an expression, and returns the value of the expression. For example eval 3 4 exp1 = 10.

Declarative Programming

Workshop exercises set 4.

QUESTION 1
Write a Haskell version of the tree sort algorithm, which inserts all the to-be-sorted data items into a binary search tree, then performs an inorder traversal to extract the items in sorted order. Use simple structural induction where possible.

QUESTION 2
Write a Haskell function to "transpose" a list of lists. You may assume that all lists are non-empty, and that the inner lists are all the same length. If you are given a list of N lists, each of length M, the result should be a list of M lists, each of length N. For example,

```
        transpose [[1,2],[4,4],[8,9]]
```

should return

```
        [[1,4,8],[2,4,9]]
```

QUESTION 3
Write a Haskell function which takes a list of numbers and returns a triple containing the length, the sum of the numbers, and the sum of the squares of the numbers. Try coding this with (1) three separate traversals of the list and (2) a single traversal of the list.

Declarative Programming

Workshop exercises set 5.

QUESTION 1
Define the function

```
    maybeApply :: (a -> b) -> Maybe a -> Maybe b
```

that yields Nothing when the input Maybe is Nothing, and
applies the supplied function to the content of the Maybe when it is
Just some content.

Try, for example, computing

```
    maybeApply (+1) (Just 41)
    maybeApply (+1) Nothing
```

This function is defined in the standard prelude as fmap.

QUESTION 2
Define the function

```
    zWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

that constructs a list of the result of applying the first argument to
corresponding elements of the two input lists. If the two list
arguments are different lengths, the extra elements of the longer one
are ignored. For example,

```
    zWith (-) [1,4,9,16] [1,2,3,4,5] = [0,2,6,12]
```

This function is defined in the standard library as 'zipWith'.

QUESTION 3
Define the function

```
    linearEqn :: Num a => a -> a -> [a] -> [a]
```

that constructs a list of the result of multiplying each element in the
third argument by the first argument, and then adding the second argument.
For example,

```
    linearEqn 2 1 [1,2,3] = [2*1+1, 2*2+1, 2*3+1] = [3,5,7]
```

Write the simplest defintion you can, remembering the material covered
recently.

QUESTION 4
The following function takes a number and returns a list containing the
positive and negative square roots of the input (assume non-zero input)

```
>sqrtPM :: (Floating a, Ord a) => a -> [a]
>sqrtPM x
>  | x  > 0    = let y = sqrt x in [y, -y]
>  | x == 0    = [0]
>  | otherwise = []
```

Using this function, define a function allSqrts that takes a list and
returns a list of all the positive and negative square roots of all
the numbers on the list. For example:

```
    allSqrts [1,4,9] = [1.0,-1.0,2.0,-2.0,3.0,-3.0]
```

Include a type declaration for your function.

QUESTION 5
Lectures have given the definitions of two higher order functions in the
Haskell prelude, filter and map:

```
        filter :: (a -> Bool) -> [a] -> [a]
        map :: (a -> b) -> [a] -> [b]
```

Filter returns those elements of its argument list for which the given function
returns True, while map applies the given function to every element of the
given list.

Suppose you have a list of numbers, and you want to (a) filter out all the
negative numbers, and (b) apply the sqrt function to all the remaining
integers.

(a) Write code to accomplish this task using filter and map.
(b) Write code to accomplish this task that does only one list traversal,
without any higher order functions.
(c) Transform (b) to (a).

Declarative Programming

Workshop exercises set 6.

QUESTION 1

Download the files borders.pl, cities.pl, countries.pl, and rivers.pl.
These files contain facts about the world circa 1980. Create a file
world.pl and insert the four lines:

```
:- ensure_loaded(borders).
:- ensure_loaded(cities).
:- ensure_loaded(countries).
:- ensure_loaded(rivers).
```

These lines will automatically load the four files when you load your
world.pl file.

Start up SWI Prolog and load your world.pl file.

This will define a predicate borders/2 (that notation means a
predicate named borders that takes two arguments) describing which
countries and oceans border which others.

Give a query to find what borders Australia (remember: Prolog symbols
are all lower case).

QUESTION 2

Give a query to find what shares a border with both France and Spain.

QUESTION 3

The files you have loaded also define a predicate country/8:

```
    country(Country,Region,Latitude,Longitude,Area,
            Population,Capital,Currency)
```

where Country is a country located in Region at the indicated
Latitude and Longitude, occupying the specified Area, occupied by the

specified Population, with the specified Capital city and using the specified Currency.

Give a query to find what countries share a border with both France and Spain. Remember, _ specifies a "don't care" variable.

QUESTION 4

Edit your world.pl file and define a predicate country/1 so that country(C) holds when C is any country. Reload your file and use your new country/1 predicate to find what countries share a border with both France and Spain. Note that you can type the goal "make." to Prolog to reload any changed files, much like ":reload (or ":r") in GHCi.

QUESTION 5

Edit your world.pl file again to define a predicate larger/2 so that larger(Country1, Country2) holds when the area of Country1 is larger than that of Country2. You can use the (infix) predicates < and > to compare numbers, but note that you must ensure that the arguments of a comparison are bound when the comparison is executed, so the goals that bind the values to be compared must appear before the comparison.

Which is bigger, Australia or China?

QUESTION 6

The predicate river/2 relates rivers, their countries, and the sea they drain into. river(River, Countries) holds when River is a river that flows through or into all of the countries on the list Countries.

The member/2 predicate is an SWI Prolog built-in that relates lists and their elements. member(Elt, Lst) holds when Elt is an element of Lst.

Write a predicate river_country(River, Country) that holds when River is a river, Country is a country, and River flows into and/or out of Country.

Also write a predicate country_region(Country, Region) that holds when Country is a country in region Region.

Give a query to find a river that flows between countries in different regions.

QUESTION 1

Implement the predicate list_of(Elt, List) such that every element of List is (equal to) Elt. What modes make sense for this predicate? What modes does it actually work in?

Hint: the structure of the code is very similar to that of proper_list/1 from the lecture notes.

QUESTION 2

Implement the predicate all_same(List) such that every element of List is identical. This should hold for empty and single element lists, as well.

QUESTION 3

Implement the predicate adjacent(E1, E2, List) such that E1 appears
immediately before E2 in List. Implement it by a single call to
append/3. What modes should and does this work in?

QUESTION 4

Reimplement the adjacent(E1, E2, List) predicate as a recursive
predicate that calls no other predicate but itself.

Hint: the structure of the code is very similar to that of member/2
from the lecture notes.

QUESTION 5

Implement the predicate before(E1, E2, List) such that E1 and E2 are
both elements of List, where E2 occurrs after E1 on List.

QUESTION 6

Suppose we wish to represent a set of integers as a binary tree. We
can use the atom empty to represent an empty tree or node, and
tree(L,N,R) to represent a node with label N (an integer), and left
and right subtrees L and R. Naturally, we want N to be strictly
larger than any label in L and strictly smaller than any in R. The
tree need not be balanced. For example,

```
        tree(tree(tree(empty, 1, empty),
                  2,
                  tree(empty, 3, tree(empty, 4, empty))),
             5,
             tree(tree(tree(empty,6,empty),
                       7,
                       tree(empty,8,empty)),
                  9,
                  tree(empty, 10, empty)))
```

is one possible representation of the set of numbers from 1 to 10. It
might be visualized as

```
                 5
         /--------/ \--------\
        2                     9
       / \                   / \
      1   3                 7   10
           \               / \
            4             6   8
```

Hint: Prolog's arithmetic comparison operators are <, >, =< (not <=),
and >=. You can also use = and \= for equality and disequality.

Write a predicate intset_member(N, Set) such that N is a member of
integer set Set. Do not search in parts of the tree where the sought
element cannot be. This only needs to work when N is bound to an
integer and Set is bound to an integer set represented as described
above. Later in the subject we will learn how to make this work in
other modes.

Hint: write one clause for the element being at the root of the
tree, one for it being in the left subtree, and one for the right subtree.

Write a predicate intset_insert(N, Set0, Set) such that Set is the

same as Set0, except that Set has N as a member. It doesn't matter
whether Set0 already has N in it, but Set must not have multiple
occurrences of N.

QUESTION 1

This definition of sumlist/2 is not tail recursive:

```
sumlist([], 0).
sumlist([N|Ns], Sum) :-
        sumlist(Ns, Sum0),
        Sum is N + Sum0.
```

Rewrite it to be tail recursive.

QUESTION 2

Given a binary tree defined to satisfy the predicate tree/1

```
tree(empty).
tree(node(Left,_,Right)) :-
        tree(Left),
        tree(Right).
```

write a predicate tree_list(Tree, List) that holds when List is a
list of all the elements of Tree, in left-to-right order. This code
need only work in the mode where the tree is input.

QUESTION 3

Revise the definition from the previous question not to use append/3
to construct the list. That is, ensure the cost of the operation is
proportional to the number of elements in the tree.

Hint: look at the approach taken to write a tail recursive reverse
predicate for inspiration.

QUESTION 4

Write a predicate list_tree(List, Tree) that holds when Tree is a
balanced tree whose elements are the elements of List, in the order
they appear in List. This need only work in the mode where List is a
proper list.

Hint: First divide the list into the first half, the middle element,
and the last half, then recursively construct a tree from the first
half and second half, and assemble these into the resulting tree.

Clarification: The main intended use of this predicate is to
build a balanced tree from a list. It should, as far as
reasonable, work in other modes. One complication is that it's
possible for there to be different trees, balanced in different
ways, with the same nodes in order. You don't need to handle
this possibility here (though it would make an interesting
challenge exercise). You just need to handle balanced trees
that could have been produced by this predicate.

Declarative Programming

Workshop exercises set 09.

QUESTION 1

Write a Prolog predicate same_elements(L1, L2) that holds when all elements of list L1 are in L2 and vice versa, though they may be in different orders and have different numbers of occurrences of the elements. This need only work in modes where both arguments are ground.

QUESTION 2

Rewrite your same_elements predicate to work in n log(n) time, where n is the length of the longer list.

QUESTION 3

Write a predicate times(W,X,Y,Z) that holds when all arguments are integers and W*X + Y = Z and 0 <= Y < |W| (where |W| denotes the absolute value of W). This implies that W \= 0, and that |W| <= |Z|. This predicate should work when W and at least one of X and Z are bound to integers, and should be deterministic as long as at least three of W, X, Y and Z are bound.

This predicate is similar to the built-in predicate plus/2, which can do addition and subtraction, depending on which arguments are bound when you call it. But times/4 handles multiplication and addition or division and modulus, depending on how you call it.

Hint: use the predicate integer/1 to check if an argument is bound to an integer.

Hint: the builtin between(X,Y,Z) holds when X <= Z <= Y, and work when X and Y are bound to integers. If Z is unbound, it will nondeterministically generate Z between X and Y inclusive.

Hint: use the function abs/1 (on the right side of is) to compute absolute value.

Hint: the expressions X div Y rounds down while X // Y rounds toward zero. Also, the expression X mod Y produces a negative result when Y is negative, while X rem Y produces a negative result when X is negative. So you want to use div together with mod and // together with rem.

Hint: the goal

```
    throw(error(instantiation_error, context(times/4,_)))
```

will throw an exception reporting that a call to times/4 had insufficiently instantiated arguments. Similarly, the goal

```
    throw(error(type_error(integer, X), context(times/4,_)))
```

will throw an exception reporting that X was expected to be an integer and was not in a call to times/4.

QUESTION 4

Write a program to solve the water containers problem.

You have two containers, one able to hold 5 litres and the other able to hold 3 litres. Your goal is to get exactly 4 litres of water into the 5 litre container.

You have a well with an unlimited supply of water.
For each ``turn,'' you are permitted to do one of the following:

    completely empty one container, putting the water back in the well

    completely fill one container from the well

    pour water from one container to the other just until the
    source container is empty or the receiving container is full.

In the last case, the original container is left with what it
originally had less whatever unfilled space the receiving container
originally had.

All containers begin empty.

Write a Prolog predicate containers(Moves) such that Moves is a list of
actions to take in order to obtain the desired state. Each action on
the list is of one of the following forms:

    fill(To), where To is the capacity of the container to fill from
    the well

    empty(From), where From is the capacity of the container to empty

    pour(From,To) where From is capacity of the container to pour from
    andTo is the capacity of the container to pour into

Hint: write a predicate that computes the effect of each of the
possible moves. Then write a predicate that nondeterministically
explores all the possible sequences of moves, computing their effect.

Hint: you will need to stop Prolog from repeatedly returning to the
same state, otherwise Prolog will search longer and longer sequences
of moves just pouring water back and forth between the two
containers. You can prevent this by keeping the list of states seen
so far in the search, and reject any more that returns to a state you
have seen before.

Declarative Programming

Workshop exercises set 10.

QUESTION 1
Recall the discussion of the Maybe monad in lectures, and the definitions
of maybe_head, maybe_sqrt and maybe_sqrt_of_head. In a similar style, write
Haskell code for the function

```
        maybe_tail :: [a] -> Maybe [a]
```

which returns the tail of a list if the list is not empty, and

```
        maybe_drop :: Int -> [a] -> Maybe [a]
```

which is like the prelude function drop ("drop n xs" drops the first n elements
of the list xs), but returns a Maybe type. If n is greater than the length
of xs, it should return Nothing (drop returns [] in this case), otherwise
it should return Just the resulting list.

Code two versions of maybe_drop. Both should use maybe_tail. One should
explicitly check for Nothing and the other should use >>=.

QUESTION 2
Given the tree data type defined below, write the Haskell function

```
   print_tree :: Show a => Tree a -> IO ()
```

which does an inorder traversal the tree, printing the contents of each node

on a separate line. What are the advantages and disadvantages of this approach
compared to traversing the tree and returning a string, and then printing
the string?

```
>data Tree a = Empty | Node (Tree a) a (Tree a)
```

QUESTION 3
Write a Haskell function

```
    str_to_num :: String -> Maybe Int
```

that converts a string containing nothing but digits to Just the number they
represent, and any other string to Nothing. Hint: the standard library module
Data.Char has a function isDigit that tests whether a character is a decimal
digit, and another function digitToInt that converts such characters to a
number between between 0 and 9.

QUESTION 4
Write two versions of a Haskell function that reads in a list of lines
containing numbers, and returns their sum. The function should read in lines
until it finds one that contains something other than a number.

The first version of the function should sum up the numbers as it read them in.
The second should collect the entire list of numbers before it starts summing
them up.

QUESTION 5
Write a Haskell main function that repeatedly reads in and executes commands
to implement a trivial phonebook program. The commands it should support are:

```
    print           prints the entire phone book
    add name num    adds num as the phone number for name
    delete name     delete the entry for name
    lookup name     print the entries that match name
    quit            exit the program
```

To keep things simple, only check the first letter of commands (so people
can abbreviate commands to a single letter). You may assume that a name is
a single word, and that it must match exactly. You can use the Haskell
prelude function words to split a single string into a list of words.
If you print a prompt and expect to read the command on the same line,
you need to do hFlush stdout to ensure the prompt is written before reading
the user command. To use this, you will need to import System.IO.

Declarative Programming

Workshop exercises set 11.

QUESTION 1
Write a function fibs :: Int -> [Integer] which returns a list containing
the first n numbers in the Fibonacci sequence: [0,1,1,2,3,5,8,...], where
the third and subsequent numbers are the sum of the two preceeding numbers
(0+1=1, 1+1=2, 1+2=3, 2+3=5, etc). We use Integer rather than Int because
the numbers grow exponentially and therefore overflow native Ints quite
quickly. Is the algorithmic complexity of your solution acceptable?

QUESTION 2
If we do pairwise addition of the elements of the Fibonacci sequence
and its tail, we get the tail of the tail of the sequence:

```
  0 1 1 2 3 5 8 ...     fibs
+ 1 1 2 3 5 8 ...       tail fibs
```

```
= 1 2 3 5 8 ...         tail (tail fibs)
```

Use this property to write a definition of allfibs :: [Integer] which is
the (infinite) Fibonacci sequence (Hint: the zipWith Prelude function
is useful). Define fibs in terms of allfibs. How efficient is this definition
of fibs compared to your previous one?

QUESTION 3
Consider the bottom-up merge sort implementation from workshop 2.

```
>mergesort xs = repeat_merge_all (merge_consec (to_single_els xs))
>
>to_single_els [] = []
>to_single_els (x:xs) = [x] : to_single_els xs
>
>merge [] ys = ys
>merge (x:xs) [] = x:xs
>merge (x:xs) (y:ys)
>       | x <= y = x : merge xs (y:ys)
>       | x >  y = y : merge (x:xs) ys
>
>merge_consec [] = []
>merge_consec [xs] = [xs]
>merge_consec (xs1:xs2:xss) = (merge xs1 xs2) : merge_consec xss
>
>repeat_merge_all [] = []
>repeat_merge_all [xs] = xs
>repeat_merge_all xss@(_:_:_) = repeat_merge_all (merge_consec xss)
```

With list xs of length n, what is the maximum additional space that is
needed at any one time, assuming strict evaluation, for evaluating
merge_consec (to_single_els xs)? What if lazy evaluation is used instead?

What is the maximum additional space is needed at any one time, assuming
strict evaluation, for evaluating mergesort xs? Can we do significantly
better than this?

QUESTION 4
Consider an interpreter for a language which produces a pair
containing the result of the computation plus some debugging information,
which is a string containing information about all assignment statements
and function calls. Compare the efficiency of the following:

a) Execution of the interpreter using strict evaluation and printing
   the debugging string.
b) Execution of the interpreter using lazy evaluation and printing
   the debugging string.
c) Execution of the interpreter using strict evaluation but not printing
   the debugging string.
d) Execution of the interpreter using lazy evaluation but not printing
   the debugging string.
e) Execution of a similar interpreter which doesn't produce the debugging
   string at all.

QUESTION 5
Here are some students' answers to one of the questions on a sample
mid-semester test, which were posted on the LMS (thanks to the authors).
The question asked for a Haskell function to print out Mtrees
with indentation showing the structure. Compare and contrast these
solutions. Can you come up with something better than all three?

```
>data Mtree a = Mnode a [Mtree a]
```

```
>print_mtree :: Show a => Mtree a -> IO()
>print_mtree tree = indent_mtree 0 tree
>    where
>        indent_mtree :: Show a => Int -> Mtree a -> IO()
>        indent_mtree i (Mnode val children) = do
>            putStrLn $ (replicate i ' ') ++ (show val)
>            foldl (>>) (return ()) (map (indent_mtree (i+1)) children)

>type Line = String
>
>print_mtree' :: Show a => Mtree a -> IO ()
>print_mtree' t =
>    let
>        toLines :: Show a => Mtree a -> [Line]
>        toLines (Mnode val cs) = show val : map (' ':) (concatMap (toLines) cs)
>    in  foldl (\acc str -> acc >> (putStrLn str)) (return ()) (toLines t)
>
>-- A clearer version
>print_mtree2 :: Show a => Mtree a ->  IO ()
>print_mtree2 t =
>    let
>        toLines :: Show a => Mtree a -> [IO ()]
>        toLines (Mnode val cs) =
>            print val : map (putChar ' ' >>) (concatMap (toLines) cs)
>    in  foldl1 (>>) (toLines t)
```
QUESTION 1

Write a Haskell implementation of the old Animals guessing game.
Start by prompting "Think of an animal. Hit return when ready." Wait
for the user to hit return, then ask: "Is it a penguin?" and wait for
a Y or N answer. If the answer is yes, print out that you guessed it
with 0 questions. If no, then ask them what their animal was, ask
them to enter a yes or no question that would distinguish their animal
from a penguin, and whether the answer is yes or no for their animal.

Then start over. This time start by asking them the question they
just entered, and depending on their answer, and the answer they said
to expect for their previous animal, ask them if their (new) animal is
their previous animal, or ask if it is a penguin. If that is correct,
print out that you guessed it with 1 question. If no, then ask them
what their animal was, ask them to enter a yes or no question that
would distinguish their animal from the one you just guessed, and
whether the answer is yes or no for their animal.

The game proceeds like this indefinitely. You should build up a
decision tree with questions at the nodes, and animals at the leaves.
For each animal they think of, you traverse the tree from the root
asking questions and following the branch selected by their answer
until you reach a leaf, then guess the animal at the leaf, and get
them to give you a question to extend the tree if you get it wrong.

QUESTION 1
Question about [a] vs [[a]] vs Maybe [a] vs [Maybe a] - which is
most appropriate when? Eg Matt Giuca's LMS posting:

Let's assume we're writing a "sports team signup sheet" program, where a
Team consists of a number of Players. Naturally, we would define it like
this:

```
type Team = [Player]
```

There is no immediate reason to have a Maybe type either inside or outside of the list. If you instead made it [Maybe Player], you would have to be explicitly checking each entry to see if it's Nothing or Just (as an aside, note that in Java you would always have to check for null -- it's a key advantage of Haskell/Mercury that null isn't allowed unless you explicitly declare something Maybe). If you wanted to print out the list, you would have to delete all the non-Nothing entries. And [], [Nothing] and [Nothing, Nothing, Nothing] would represent the same team -- so why allow this redundant data structure. Similarly, if you made it a Maybe [Player], now an "empty" team could be represented as "Nothing" or "Just []", so you would have special cases for the empty team all over the place.

But there are some reasons to combine a List and a Maybe. Consider that this program now allows a Team to be created, but it can't have Players in it until they have paid their signup fee. Now maybe it makes sense to define it as:

```
type Team = Maybe [Player]
```

A Team of Nothing does exist, but hasn't paid its signup fee, so the player list is more than just empty -- it isn't there at all. A Team of Just [] has paid their signup fee, but hasn't enrolled any Players yet. The important thing is that both "Nothing" and "Just []" have a distinct meaning, so the Maybe List type is justified (though there are probably better ways to represent this).

Alternatively, consider that the program now allows "undecided" signups -- a team can nominate that they intend to put a player in a particular place, but haven't decided on a person yet. Now maybe it makes sense to define:

```
type Team = [Maybe Player]
```

The empty Team, [], actually has no players. The Team [Nothing, Nothing, Nothing] has three player spots nominated, but have not appointed any specific people there yet. Still, length will tell us the planned team size. This is a useful representation.

The key is to consider, for every valid value of this data type, a) does it mean something sensible, and b) is it the only way to represent that meaning in this data type. (a) is highly desirable, (b) is less desirable but still good. Again, they aren't always possible.

QUESTION 2
A question that shows code that uses a variable instead of a constructor in a pattern, leading to a bug. Ask students to find the bug.

QUESTION 3
Heap implementation.

QUESTION 4
Write definitions (as simple as possible) of the Haskell functions mysum and myproduct, which return the sum and product of a list of numbers, respectively. These definitions have a similar structure; what other definitions you have seen have the same structure? Note: later we will consider a "higher order function" which allows you to write definitions of such functions much more concisely.
XXX do not use: will be discussed in lectures

QUESTION 5
Write a function called filter_map that does the jobs of filter and map

at the same time. The type of filter_map should be

```
        filter_map :: (a -> Maybe b) -> [a] -> [b]
```
XXX do not use: will be discussed in lectures

QUESTION 6
Use standard higher order functions and operator sections to write
single line definitions of sum, product, all_pos, some_not_pos and length
(see question in previous tutorial). What are the advantages and
disadvantages of such definitions.
XXX do not use: will be discussed in lectures

QUESTION 7
Foldr takes a list and "folds" it into a single value, but that value
could be any type, including a list. Can you implement map and filter
using foldr?

QUESTION 8
Consider the task of converting a list of single element lists into
a sorted list by repeatedly merging lists. This is a fold operation.
What is the algorithmic complexity if we use "foldr merge []"? What if
we use foldl instead of foldr? Is the result the same, and why or why not?
What is the complexity? What if we use balanced_fold (defined in lectures)?

QUESTION 9
There are two improvements we can make to the definition of the balanced_fold
function given in lectures.

The balanced_fold function given in lectures computes the length of not just
the original list, but of every one of the lists it is divided into,
even though the code that divides a list knows (or should know) how long
the resulting lists should be. The first improvement is therefore avoiding
the redundant length computations, and computing the length of just one list:
the original list.

The second improvement is to avoid the intermediate lists being created
at each level of recursion, when the list is split in two. An alternative
is for the first recursive call to return both the fold of the first
half (or n elements) of the list and the remainder of the list (which
is then used in the second recursive call). For example, using the scenario
from the previous question, doing a merge using balanced folds on the list
of lists [[4],[1],[6],[2],[8],[7],[3],[5]], the first recursive call could
return the pair ([1,2,4,6], [[8],[7],[3],[5]]), and then pass the list
of lists [[8],[7],[3],[5]] to the second call.

Write two versions of balanced_fold. The first should have the first of these
improvements, the second should have both.

You might want also want to code merge sort (for example) in this style,
and then generalise it to balanced_fold.

QUESTION 10
Consider the following tree data type:

```
>data Tree a = Empty | Node (Tree a) a (Tree a)
```

Define a higher order function

```
>map_tree :: (a->a) -> Tree a -> Tree a
```

which is the analogue of map for trees (rather than lists): it applies
a function to each element of the tree and produces a new tree containing

```
the results. The result tree is the same shape as the input tree, just
as the result list in map is the same length as the input list.

QUESTION 11
Given the Tree type above, write functions which take a Tree and compute

(a) the height,
(b) the number of nodes,
(c) the concatenation of the elements in the nodes (assuming that the
    values in tree nodes are in fact lists)
(d) the sum of the elements (assuming that values in the tree are Nums)
(e) the product of the elements in the nodes (assuming they are Nums), and
(f) Just the maximum of the elements in the nodes, or, Nothing if the tree
    is empty Nothing.

Before you start, it may be helpful to review the tree sort question
from the workshop for week 4.

These operations can be seen as folds on Trees. When you get tired of writing
the same pattern for traversing the Tree, write a higher order function

>foldr_tree :: (a->b->a->a) -> a -> Tree b -> a

which can be used to define the other functions. The first argument is
a function which is applied at each Node, after the subtrees have been
folded. The second argument is returned for Empty trees.

QUESTION 12

Write a Mercury program that takes at least one integer command line
arguments, and prints out an arithmetic expression whose value is the
first command line argument using all and only the command line
arguments following the first exactly once, and using any of the
operations of addition, subtraction, multiplication, and division.
Calculations may use parentheses as necessary.  All calculations
should be integer-valued, so all divisions must work out to an
integer.  Eg, 7/3 would not be allowed, but 6/3 would.  For example,

        ./mathjeopardy 24 12 6 3 2

might print

        12+6+3*2

and

        ./mathjeopardy 91 8 27 36 4 9 18

might print

        8+27+4*(18-36/9)

For starters, print the expression with parentheses everywhere.  When
you get that working, try to print as few parentheses as necessary.

(I'm calling this mathjeopardy because it's a bit like the game show
Jeopardy:  you are given the answer and have to come up with the question.)

QUESTION 13
Implement a Haskell class called Appendlist which supports the
following operations:
```

empty:	creates an empty list
cons:	adds an element to the front of a list
append:	appends two lists together
is_empty:	checks if a list is empty, returning a Bool
de_cons:	returns a pair containing the head (the first element) and the tail (the rest of the list), aborting if the list is empty

Implement two instances: one for (normal) lists and one for cords as defined in lectures:

```
>data Cord a = Nil | Leaf a | Branch (Cord a) (Cord a)
```

QUESTION 14
In a language such as C we sometimes use doubly linked lists, where each cell has pointers to the next cell and also to the previous cell. This allows us to move one element left or right in the list, insert or delete an element to the left or right of the "current position", or "cursor". By thinking of the current position as being *between* two elements (or to the left of the leftmost element or to the right of the rightmost element), we can naturally support empty lists (which are a problem if we need a "current element"). How can we support the following operations in constant time in Haskell?

dll_new:	creates a new (empty) doubly linked list
dll_empty:	tests if the list is empty
dll_empty_l:	tests there are no elements to the left of the cursor
dll_empty_r:	tests there are no elements to the right of the cursor
dll_l:	the element on the left
dll_r:	the element on the right
dll_move_l:	move the cursor one element left
dll_move_r:	move the cursor one element right
dll_add_l:	adds an element to the left of the cursor
dll_add_r:	adds an element to the right of the cursor
dll_remove_l:	removes the element to the left of the cursor
dll_remove_r:	removes the element to the right of the cursor

In Mercury we can have several modes for each predicate. Which of the operations above could potentially be combined into a single predicate by using multiple modes? Give the code (including the determinism in the declarations) and some sample calls which illustrate its use in different modes. Is it always a good idea to combine the different modes? Why or why not?

QUESTION 15
Define versions of map for the following data types:

```
>data Ltree a = LTLeaf a | LTBranch (Ltree a) (Ltree a)
>data Cord a = Nil | Leaf a | Branch (Cord a) (Cord a)
>data Mtree a = Mnode a [Mtree a]
```

QUESTION 16
In "foldr f b" we essentially replace [] by b and : by f. For example,

```
        foldr f (+) 0 [1, 2]
        = foldr f (+) 0 (1:2:[])
        = foldr f (+) 0 ((:) 1 ((:) 2 []))
        = ((+) 1 ((+) 2 0))
        = 1 + (2 + 0)
        = 3.
```

Similarly, foldr_tree f b (from the previous workshop) replaces Empty by b

and Node by f (which takes three arguments). Define versions of foldr
in this style for the types in the previous question.

Note that GHC supports the Foldable typeclass (Data.Foldable), but this
typeclass generalises foldr for lists in a different way, ignoring the
structure and just using the elements within the data type. This form
of foldr on a tree is equivalent to traversing the tree to get a list,
then applying foldr to the list. Here we want more general functions
which do not necessarily ignore the structure. For example, in the
previous workshop, foldr_tree Node Empty is the identity function over
Trees, which is not possible if we ignore the structure. Similarly,
height_tree requires the tree structure.

QUESTION 17
Recall the following functions from lectures, for concatenating a
list of lists and converting a cord into a list:

```
>concatr = foldr (++) []  -- Efficient
>concatl = foldl (++) []  -- Inefficient

>cord_to_list :: Cord a -> [a]  -- Inefficient
>cord_to_list Nil = []
>cord_to_list (Leaf x) = [x]
>cord_to_list (Branch a b) =
>    (cord_to_list a) ++ (cord_to_list b)

>cord_to_list2 :: Cord a -> [a]  -- Efficient
>cord_to_list2 c = cord_to_list' c []
>cord_to_list' :: Cord a -> [a] -> [a] -- Arg 2 is an accumulator
>cord_to_list' Nil rest = rest
>cord_to_list' (Leaf x) rest = x:rest
>cord_to_list' (Branch a b) rest =
>    cord_to_list' a (cord_to_list' b rest)
```

Recall the reason for the relative (in)efficiency is that ((a++b)++c)
has to copy the elements and cons cells of the list `a' twice, whereas
(a++(b++c)) only needs to do it once. The cost of ++ depends on the length
of its first argument but not its second argument.

Define four versions of concat_rev, which concatenates the lists in reverse
order (the inner lists are not reversed, for example concat_rev
[[1,2],[3]] = [3,1,2]) using (1) foldr, (2) foldl, (3) recursion without an
accumulator, and (4) recursion with an accumulator. Determine which versions
are efficient.

How would you code cord_to_list_rev, which converts a cord to a list,
but in the reverse order? Code it directly rather than creating an in-order
list and then reversing it.

What is the relationship between foldr for cords and the code above and
in what way is the code similar to both foldl and foldr?

QUESTION 18
The definition of the cord data type presented in lectures (see Q1 above)
can represent the same nonempty sequence of items in more than one way.
For example, the sequence [1, 2, 3] can be represented as

```
        Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3))
```

or as

```
        Branch (Branch (Leaf 1) (Leaf 2)) (Leaf 3)
```

Without this flexibility, the operation to concatenate two cords couldn't
be implemented in constant time. However, this type also has unnecessary
flexibility. For example, it can represent the empty sequence in more than

one way, including Nil, Branch Nil Nil and Branch Nil (Branch Nil Nil).
Design a version of the cord data type that has only one way to represent
the empty sequence.

QUESTION 19
Compare and contrast the following Haskell and Mercury code:

```
>append [] l = l
>append (j:k) l = j : append k l

>rev [] = []
>rev (a:bc) = append (rev bc) [a]

append([], L, L).
append([J | K], L, [J | KL]) :-
        append(K, L, KL).

rev([], []).
rev([A | BC], R) :-
        rev(BC, CB),
        append(CB, [A], R).
```

QUESTION 20
Write Haskell code which can check if one string (or, more generally,
a list of elements in the Eq type class) is a substring (sublist) of another.
The elements of the substring must occur next to each other in the bigger list,
so that for example, "bcd" is a substring of "abcde", but not of "abcfde".

Note that there are some tricky cases. For example, when searching for "bcd"
in "bcabcd", the first two characters of the substring match the initial part
of the string but the next character (a) prevents a match at that point;
nevertheless, there is a match later in the string.

Even more tricky are cases where you cannot skip characters that participated
in an ultimately failed match. Consider looking for "bcbcd" in the string
"bcbcbcd". When you look for a match starting at the first character,
the first four characters match, while the fifth doesn't. However, there
is a match starting at the third character of "bcbcbcd".

Your function could simply return a Boolean. What other information could
it return?

How would your overall design differ in Mercury? Write a Mercury predicate
that does the task in a manner appropriate for Mercury.

QUESTION 21
Write Haskell and Mercury implementations of queues, where a queue
is represented as a list, the head of the list being the head of the
queue. The code should support the following operations:

queue_new:	creates a new (empty) queue
queue_empty:	tests whether a queue is empty
queue_add:	adds a new element to the back of the queue
queue_remove:	returns a two-tuple containing (a) the element at the head of the queue and (b) the rest of the queue

What is the main disadvantage of this representation?
Would reversing the order of elements help?

QUESTION 22
Write Haskell and Mercury implementations of queues using a different
representation: a pair of lists (lf, lr), where lf ++ (reverse lr) gives the

single list representation above. Can this avoid the main disadvantage
of the representation used in the previous question?