COMP90048 Declarative Programming
Semester 1, 2018
Peter J. Stuckey

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

ANSWER

```
>my_sum [] = 0
>my_sum (x:xs) = x + my_sum xs

>my_product [] = 1
>my_product (x:xs) = x * my_product xs
```

These have similar structure to some_not_pos, all_pos, len (and many more):

```
f [] = BASE_CASE
f (x:xs) = SOME_FUNCTION x (f xs)
```

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

ANSWER

Here we use the "curried" style of definition, avoiding any explicit mention of the list, and making the definitions even more concise.
We have added "1" suffix to avoid name conflicts with previous versions.

```
>my_sum1 = foldr (+) 0
>my_product1 = foldr (*) 1
>all_pos1 = foldr ((&&).(>0)) True
>some_not_pos1 = foldr ((||).(<=0)) False
>len1 = foldr ((+).(const 1)) 0
```

An advantage is the definitions are very short and emphasise they are similar mathematically. A disadvantage is is they can be rather cryptic, especially if you are not used to this style. Its always important to use sensible names for functions and have comments describing what they compute.

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?

ANSWER
Yes.

```
>map' f xs = foldr ((:).f) [] xs
>filter' p xs = foldr (\x->if p x then (x:) else id) [] xs
```

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)?

ANSWER

Foldr and foldl both result in O(N^2) complexity. Essentially these are variations on insertion sort: we repeatedly merge a single element list with a sorted list, which is like inserting an element into a sorted list.

They produce the same result because merge is associative, with identity [].

With balanced_fold we get O(N log N) complexity (a version of merge sort).

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.

ANSWER
Here is the version with both improvements:

```
>balanced_fold' :: (e -> e -> e) -> e -> [e] -> e
>balanced_fold' f b xs = fst (bal_fold1 f b xs (length xs))
>
>bal_fold1 :: (e -> e -> e) -> e -> [e] -> Int -> (e, [e])
>bal_fold1 _ b xs 0 = (b, xs)
>bal_fold1 _ _ (x:xs) 1 = (x, xs)
>bal_fold1 f b xs len = -- len > 1
>    let
>        len1 = len `div` 2
>        len2 = len - len1
>        (value1, rest1) = bal_fold1 f b xs len1
>        (value2, rest2) = bal_fold1 f b rest1 len2
>    in
>        (f value1 value2, rest2)
```

For completeness, here is a version of merge sort which uses this.
Here we use map (with an operator section) instead of a separate
"to_single_els" function.

```
>mergesort :: (Ord a) => [a] -> [a]
>mergesort xs = balanced_fold' merge [] (map (:[]) 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
```

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.

ANSWER

```
>map_tree _ Empty = Empty
>map_tree f (Node l n r) = Node (map_tree f l) (f n) (map_tree f r)
```

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.

ANSWER
I'm already tired of writing that pattern, so I'll go directly to the higher order function :-)

```
>foldr_tree _ b Empty = b
>foldr_tree f b (Node l n r) = f (foldr_tree f b l) n (foldr_tree f b r)
```

Note that below we use "curried" functions - there is no reference to the tree. Type inference does a pretty good job, though it s not perfect because type classes are involved. For sum_tree and product_tree it infers
the elements are Integers (we should add a type declaration saying they are Nums). For max_tree it fails to infer a type, so we have to add a type declaration.

```
>height_tree = foldr_tree (node_max_plus_1) 0
>   where node_max_plus_1 l _ r = 1 + max l r
>size_tree = foldr_tree node_plus_1 0
>   where node_plus_1 l _ r = 1 + l + r
>sum_tree = foldr_tree (ternary (+)) 0
>product_tree = foldr_tree (ternary (*)) 1
>concat_tree = foldr_tree (ternary (++)) []
>max_tree :: Ord a => Tree a -> Maybe a
>max_tree = foldr_tree node_max Nothing
>   where node_max l n r = maybe_max (maybe_max l (Just n)) r
```

Ternary takes a binary function and turns it into a ternary one by applying it
twice (this pattern occurs three times above, so it is worth doing). Right
associativity is used (so ++ is faster). The type could actually be more general; x and y don't have to be the same type as z and the return type.

```
>ternary :: (a -> a -> a) -> a -> a -> a -> a
>ternary f x y z = f x (f y z)
```

Max wrapped up in Maybe:

```
>maybe_max :: (Ord a) => Maybe a -> Maybe a -> Maybe a
>maybe_max Nothing Nothing = Nothing
```

```
>maybe_max (Just x) Nothing = (Just x)
>maybe_max Nothing (Just x) = (Just x)
>maybe_max (Just x) (Just y) = Just (max x y)
```

A couple of very simple test cases:

```
>t1 = Node Empty 3 (Node Empty 4 Empty)
>t2 = Node Empty [3] (Node Empty [4,5] Empty)
```

Workshop exercise set 10.

You will need to log in to datura. You should start by creating a directory for this use in this workshop; you can name it e.g. ~/comp30020/workshop10.

The directory /home/subjects/comp30020/mercury_examples contains several small Mercury programs. Copy queens.m to your directory, take a good look at it, and compile it using mmc (mmc is installed in /usr/local/bin).

The simplest way to compile queens.m is with the command

```
    mmc queens.m
```

which creates an executable named "queen", and quite a few intermediate files
(you might like to browse some of these). There may be some warnings generated
when some intermediate C code is compiled. If you use the command

```
mmc --make queens
```

then most of the intermediate files will be put in a subdirectory Mercury, which makes it easier to clean them up. This way of compiling is particularly
advantageous if you have a multi-module program as it handles all the dependencies between modules for you.

Try executing queens. Do you understand the output? If not, read the comments in the program again.

Understanding Mercury execution
==========

The Mercury debugger allows you to step through the execution of a

Mercury program, printing parameters to calls, results etc. To understand
the queens program, it is best to use a simpler example. Change the definition
of data/1 so there are only four queens (on a four by four board):

```
data([1,2,3,4]).
```

Clean up the intermediate files with "mmc --make queens.realclean", and then
recompile the program with the --debug flag. (You should always remove the intermediate files before recompiling a program with different flags.)

Run the mdb (Mercury debugger) using the command:

```
mdb ./queens
```

With mdb, you can step through the execution just by hitting return.
Some extra commands that may be useful are

```
p (or print - prints some information about the current point)
format flat (makes future print commands print more details of
    data structures instead of truncating them as much)
p 1 (prints just the first argument, p 2 the second, p * all args)
f (or finish - skip to the end of the current sub-computation)
r (or retry - go back to the start of the current sub-computation)
```

It is particularly important to notice when predicates "EXIT" (succeed), since that is when you can print the output arguments. You also want to notice "CALL" events. When you get to them, you may want to skip to the end
of the call to see whether it succeeds (if it does, you can then print the values of the output arguments) or fails. If you then want to see in details what the call did, you can "retry" it.

Enhancing the queens program

Try implementing the following three (more or less orthogonal) enhancements.

(1) Make the number of queens (board size) a parameter which is read from standard input. It is easiest to use io.read for this.

(2) Make the output more easily understandable, by changing it from a list of
numbers into a printout of the chessboard, like this:

```
..Q..
....Q
.Q...
...Q.
Q....
```

The idea is that the solution for the N-queens problem is an N-by-N board
in which a "Q" in a given board position indicates the presence of a queen
on that position, and "." indicates an empty square.

(3) Print all the solutions. You can use the standard library predicate solutions/2; you will need ":- import_module solutions".

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.)

ANSWER

```
:- module mathjeopardy.
:- interface.
:- import_module io.

%% Note determinism cc_multi means the computation may generate more
%% than one solution, but Mercury will only produce the first and stop.
:- pred main(io::di, io::uo) is cc_multi.

:- implementation.
:- import_module list, string, char, int.

%% The main mathjeopardy program.  Parse the command line and try to
%% solve the puzzle.  If that succeeds, print out the expression and
%% exit, committing to the first solution; if not, say we can't solve
it
%% and exit.  See above for a description of the problem.

main(!IO) :-
    io.command_line_arguments(Args, !IO),
    ( map(string.to_int, Args, [Target|Inputs]) ->
        ( mathjeopardy(Target, Inputs, Expression) ->
        print_expression(Expression, 0, !IO),
        nl(!IO)
        ;   io.write_string("Could not find a solution\n", !IO),
        set_exit_status(1, !IO)
        )
    ;   usage(!IO),
        set_exit_status(1, !IO) % return an error status to the shell
    ).

:- pred usage(io::di, io::uo) is det.

%% Print out a helpful usage message.

usage(!IO) :-
```

```
    io.write_string("Usage:          mathjeopardy    target    number
number...\n", !IO),
    io.write_string(
"  mathjeopardy will try to find a mathematical expression that\n", !IO),
    io.write_string(
"    evaluates  to  the  specified  target  using  all  and  only  the
specified\n", !IO),
    io.write_string(
"      numbers  and  the  operations  of  addition,  subtraction,
multiplication,\n",
            !IO),
    io.write_string(
"  and division, and using parentheses freely.\n", !IO).

%% Our expression type.  We support only numbers and binary +, -, *,
and /

:- type expr --->
    number(int)
    ; plus(expr, expr)
    ; minus(expr, expr)
    ; times(expr, expr)
    ; divide(expr, expr)
    .

:- pred mathjeopardy(int::in, list(int)::in, expr::out) is nondet.

%% mathjeopardy(Target, Inputs, Expression)
%% Solve a mathjeopardy puzzle.  Target is the value we're trying to
%% produce, Inputs is a list of the numbers we can use, and Expression
%% is an arithmetic expression, using all and only the numbers on
%% Inputs, whose value is Target.

mathjeopardy(Target, Inputs, Expression) :-
    expr_of(Expression, Inputs, [], length(Inputs), Target).

%%  expr_of(Expr, Numbers0, Numbers, Maxdepth, Value)
%%  Expr is an expression with value Value involving all and only the
%%  numbers on the list Numbers0, with the exception of those on
%%  Nubmers.  The maximum depth of the expression tree is
%%  Maxdepth. This is needed to ensure the computation terminates,
%%  because without the MaxDepth argument, no input argument would get
%%  smaller in the recursive call.
```

```
:- pred expr_of(expr, list(int), list(int), int, int).
:- mode expr_of(out, in, in, in, out) is nondet.
:- mode expr_of(out, in, out, in, out) is nondet.

expr_of(number(Int), !Values, _, Int) :-
    list.delete(!.Values, Int, !:Values).
expr_of(Expr, !Values, Limit, Value) :-
    Limit > 0,
    expr_of(X, !Values, Limit-1, Xval),
    expr_of(Y, !Values, Limit-1, Yval),
    (   Expr = plus(X, Y), Value = Xval + Yval
    ;   Expr = minus(X, Y), Value = Xval - Yval
    ;   Expr = times(X, Y), Value = Xval * Yval
    ;   Expr = divide(X, Y),
        Yval \= 0, Xval mod Yval = 0, Value = Xval / Yval
    ).

%%  print_expression(Expr, Precedence, !IO)
%%  Print out expression Expr with minimal parentheses.  If the
%%  outermost function of Expr has precedence less than Precedence,
%%  wrap it in parentheses; otherwise don't.  Addition and subtraction
%%  have precedence 1; multiplication and division have precedence 2.

:- pred print_expression(expr::in, int::in, io::di, io::uo) is det.

print_expression(number(Int), _, !IO) :-
    write_int(Int, !IO).
print_expression(Expr, At_prec, !IO) :-
    (   Expr = plus(X, Y), Op = ('+'), Prec = 1, Comm = 1
    ;   Expr = minus(X, Y), Op = ('-'), Prec = 1, Comm = 0
    ;   Expr = times(X, Y), Op = ('*'), Prec = 2, Comm = 1
    ;   Expr = divide(X, Y), Op = ('/'), Prec = 2, Comm = 0
    ),
    ( At_prec > Prec ->
        write_char('(', !IO),
        print_op_expression(X, Op, Y, Prec, Comm, !IO),
        write_char(')', !IO)
    ;   print_op_expression(X, Op, Y, Prec, Comm, !IO)
    ).

%%  print_op_expression(Expr1, Op, Expr2, Prec, Comm, !IO)
%%  Print out an expression whose left operand is Expr1, whose operator
%%  is Op, and whose right operand is Expr2.  The precedence of the
%%  operation is Prec, and  Comm is 1 if the operation is commutative,
```

```
%%  otherwise it's 0.

:- pred print_op_expression(expr::in, char::in, expr::in, int::in,
int::in,
                io::di, io::uo) is det.

print_op_expression(X, Op, Y, Prec, Comm, !IO) :-
    print_expression(X, Prec, !IO),
    write_char(Op, !IO),
    print_expression(Y, Prec+(1-Comm), !IO).
```

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)
```

ANSWER

The class definition is as follows. Since there are two parameters (we want this to work for lists/cords/etc of any type) we need the -fglasgow-exts flag on ghc/ghci.

```
>class Appendlist c a where
>    empty ::  c a
>    cons ::   a -> c a -> c a
>    append ::  c a -> c a -> c a
>    is_empty ::  c a -> Bool
>    decons ::  c a -> (a, c a)
```

For lists the instance is quite simple. One slight problem is the simple definition of is_empty doesn't work (with the current version of the compiler) because == for lists requires the elements of the lists to be in Eq, even though in this case there are no elements to compare.

We don't want to constrain Appendlist so it only works with types in Eq.

```
>instance Appendlist [] a where
>    empty = []
>    cons = (:)
>    append = (++)
>--  is_empty = (==[])  -- compiler complains we need Eq a
>    is_empty = is_empty_list
>     decons = \(x:xs) -> (x, xs)  -- pattern matching with lambda is
supported
>--  decons = \xs -> (head xs, tail xs)  -- simpler use of lambda
>
>is_empty_list [] = True
>is_empty_list (_:_) = False
```

The code for the cord instance is a bit more complicated. The main advantage is that append is constant time (note we are using a data constructor as a function here).

```
>instance Appendlist Cord a where
>    empty = Nil
>    cons = cord_cons
>    append = Branch
>    is_empty = cord_is_empty
>    decons = cord_decons
>
>cord_cons :: a -> Cord a -> Cord a
>cord_cons x xs = Branch (Leaf x) xs
>
>cord_is_empty :: Cord a -> Bool
>cord_is_empty Nil = True
>cord_is_empty (Leaf _) = False
>cord_is_empty (Branch ls rs) = cord_is_empty ls && cord_is_empty rs
```

There are several ways we can code cord_decons. This one is quite neat. Ideally, a sequence of N calls to cord_decons, where the "tail" returned from one call is passed to the next call, should take O(N) time. We can't
always ensure this since there can be arbitrary numbers of branches containing just Nil. A weaker constraint is that a cord constructed with N operations should take O(N) time to deconstruct completely by a sequence of cord_decons calls. This requires some rearrangement of branches (see the last equation, for example).

```
>cord_decons :: Cord a -> (a, Cord a)
>cord_decons Nil = error "decons of empty cord"
>cord_decons (Leaf a) = (a, Nil)
>cord_decons (Branch Nil as) = cord_decons as
>cord_decons (Branch (Leaf a) as) = (a, as)
>cord_decons (Branch (Branch as bs) cs) =
>    cord_decons (Branch as (Branch bs cs))
```

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?

ANSWER

We can use a pair of lists: the elements to the left and right of the

cursor (the heads of the lists being the elements next to the cursor, so the left elements are in reverse order). A very simple coding is:

```
>dll_new = ([], [])
>dll_empty (ls, rs) = ls==[] && rs==[]
>dll_empty_l (ls, rs) = ls==[]
>dll_empty_r (ls, rs) = rs==[]
>dll_l (a:ls, rs) = a
>dll_r (ls, a:rs) = a
>dll_move_l (a:ls, rs) = (ls, a:rs)
>dll_move_r (ls, a:rs) = (a:ls, rs)
>dll_add_l a (ls, rs) = (a:ls, rs)
>dll_add_r a (ls, rs) = (ls, a:rs)
>dll_remove_l (a:ls, rs) = (ls, rs)
>dll_remove_r (ls, a:rs) = (ls, rs)
```

In Mercury we will use the following data type:

```
:- type dll(T)
    --->    dll(list(T), list(T)).
```

dll_new and dll_empty can be the same in Mercury. dll_new has mode `out' and dll_empty has (implied) mode `in'.

```
:- pred dll_empty(dll(T)).
:- mode dll_empty(out) is det.          % create an empty list
:- mode dll_empty(in) is semidet.       % check whether list is empty
dll_empty_l(dll([], [])).
```

dll_empty_l and dll_empty_r are pretty much the same as in Haskell, except we can use a semidet predicate instead of a function returning a Boolean:

```
:- pred dll_empty_l(dll(T)::in) is semidet.
dll_empty_l(dll([], _)).

:- pred dll_empty_r(dll(T)::in) is semidet.
dll_empty_r(dll(_, [])).
```

dll_move_l and dll_move_r can be combined, since the "reverse" of moving right is moving left (we can just use the second argument as input
and the first as output to get the "reverse" behaviour). We can also combine

dll_add_l and dll_remove_l as well as dll_add_r and dll_remove_r, since the "reverse" of moving right is moving left and the "reverse" of insertion is deletion (these versions of delete then also return the element deleted, so they also subsume dll_l and dll_r).

```
:- pred dll_move_l(dll(T), dll(T)).
:- mode dll_move_l(in, out) is semidet.     % move left
:- mode dll_move_l(out, in) is semidet.     % move right
dll_move_l(dll([A | Ls], Rs), dll(Ls, [A | Rs])).
```

For example, dll_move_l(dll([2,1],[3,4]), X) will use the in, out mode. A will unify with 2, Ls with [1] and Rs with [3,4], thus X will be unified with dll([1],[2,3,4]) (we have moved one position left). For dll_move_l(Y, dll([1],[2,3,4])) the out, in mode is used and we obtain
Y=dll([2,1],[3,4]) (of course, which is moving one position to the right).

```
:- pred dll_add_l(T, dll(T), dll(T)).
:- mode dll_add_l(in, in, out) is semidet.  % add to left
:- mode dll_add_l(out, out, in) is semidet. % remove from left
dll_add_l(A, dll(Ls, Rs), dll([A | Ls], Rs)).
```

The call dll_add_l(2, dll([1],[3,4]), X) binds X to dll([2,1],[3,4]) (the in, in, out mode, which adds an element). The call dll_add_l(Y, Z, dll([2,1],[3,4])) binds Y to 2 and Z to dll([1],[3,4]) (the out, out, in mode, which deletes the left element and returns both the element and the new list).

```
:- pred dll_add_r(T, dll(T), dll(T)).
:- mode dll_add_r(in, in, out) is semidet.  % add to right
:- mode dll_add_r(out, out, in) is semidet. % remove from right
dll_add_r(A, dll(Ls, Rs), dll(Ls, [A | Rs])).
```

There is no significant loss in combining the different modes for dll_empty/dll_new. However, for the others it means we cannot add appropriate error checking, so the predicates must be semidet. By separating the modes for dll_move_l/dll_move_r, for example, we can check whether the left (respectively, right) lists are nonempty, and call
error if they are not, allowing us to make the predicates det:

```
:- pred dll_move_l(dll(T), dll(T)).
:- mode dll_move_l(in, out) is det.     % move left
```

```
dll_move_l(dll([A | Ls], Rs), dll(Ls, [A | Rs])).
dll_move_l(dll([], _), _) :-
    error("dll_move_l with empty left list").

:- pred dll_move_r(dll(T), dll(T)).
:- mode dll_move_r(in, out) is det.     % move right
dll_move_r(dll(Ls, [A | Rs]), dll([A | Ls], Rs)).
dll_move_r(dll(_, []), _) :-
    error("dll_move_r with empty right list").
```

The same style of code is also preferable for dll_add_l, dll_add_r, dll_l and dll_r. By deciding that certain things are errors rather than leading to failure, we restrict the programming style, allowing the compiler to detect more errors. If these procedures were all semidet, then procedures wich call them would also have to be semidet unless the caller itself catches and handles the failure (e.g. by putting the call into the condition of an if-then-else). Calling error when the precondition
of a function is not met is also desirable in Haskell code. Routinely covering all patterns means that (with the appropriate compiler flags) you can be alerted when you *accidentally* don't cover all patterns.

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]
```

ANSWER

```
>ltree_map :: (a -> b) -> Ltree a -> Ltree b
>ltree_map f (LTLeaf a) = LTLeaf (f a)
>ltree_map f (LTBranch l r) = LTBranch (ltree_map f l) (ltree_map f r)

>cord_map :: (a -> b) -> Cord a -> Cord b
>cord_map f Nil = Nil
>cord_map f (Leaf a) = Leaf (f a)
>cord_map f (Branch l r) = Branch (cord_map f l) (cord_map f r)
```

For Mtrees we can use map (for lists) to process the list of subtrees.

```
>mtree_map :: (a -> b) -> Mtree a -> Mtree b
>mtree_map f (Mnode a mts) = Mnode (f a) (map (mtree_map f) mts)
```

Note that GHC is able to generate such code for you automatically (though its not in the Haskell98 standard). You need to have the -XDeriveFunctor command line option, and add "deriving Functor" to the type definitions, eg

```
data Ltree a = LTLeaf a | LTBranch (Ltree a) (Ltree a)
    deriving Functor
```

This will automatically derive the "fmap" function of the Functor typeclass. You can then use fmap for Ltrees (the same as ltree_map).

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.

ANSWER
In all these foldr versions, we have to make some decision on the order of the arguments for the different data constructors. Here we use the reverse of the order they are given above, so base cases are last, like foldr for lists and foldr_tree. Note the code is similar to the versions of map, but we use these extra arguments in place of data constructors on the right sides of equations.

```
>ltree_foldr :: (b -> b -> b) -> (a -> b) -> Ltree a -> b
>ltree_foldr _fb fl (LTLeaf a) = fl a
>ltree_foldr fb fl (LTBranch l r) =
>   fb (ltree_foldr fb fl l) (ltree_foldr fb fl r)

>cord_foldr :: (b -> b -> b) -> (a -> b) -> b -> Cord a -> b
>cord_foldr _fb _fl n Nil = n
>cord_foldr _fb fl _n (Leaf a) = fl a
>cord_foldr fb fl n (Branch l r) =
>   fb (cord_foldr fb fl n l) (cord_foldr fb fl n r)
```

For Mtrees we need arguments for Mnode, : and []. We can use foldr and map (for lists) to process the list of subtrees:

```
    foldr fc n (map (mtree_foldr fn fc n) mts)
```

but the map and fold can be combined:

```
>mtree_foldr :: (a -> c -> b) -> (b -> c -> c) -> c -> Mtree a -> b
>mtree_foldr fn fc n (Mnode a mts) =
>   fn a (foldr (fc.(mtree_foldr fn fc n)) n mts)
```

Note the most general type has three type variables: (1) the type of the
original tree elements, (2) the final result type, and (3) the type that a list of subtrees is converted into.

Some examples for testing:

```
>lt1 = LTBranch (LTLeaf 2) (LTLeaf 3)
>c1 = Branch (Leaf 2) (Branch (Leaf 3) Nil)
>mt1 = Mnode 2 [Mnode 3 []]

>ltree_sum = ltree_foldr (+) id
>cord_sum = cord_foldr (+) id 0
>mtree_sum = mtree_foldr (+) (+) 0
>mtree_id = mtree_foldr Mnode (:) []
```

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?

ANSWER
For the fold versions we use the higher order flip function which swaps the order of the arguments to (in this case) ++. This also swaps the relative efficiency: while the time taken by ++ depends on the length
of its first argument, the time taken by flip (++) depends on the length

of its second argument (since this becomes ++'s first argument).

```
>concat_rev1 = foldr (flip (++)) []  -- Inefficient
>concat_rev2 = foldl (flip (++)) []  -- Efficient
```

Foldl for lists is like using an accumulator; the following is efficient:

```
>concat_rev3 xss = concat_rev3_acc xss []

>concat_rev3_acc [] acc = acc
>concat_rev3_acc (xs:xss) acc = concat_rev3_acc xss (xs++acc)
```

Simple recursion without an accumulator is like foldr for lists; the following is inefficient:

```
>concat_rev4 [] = []
>concat_rev4 (xs:xss) = (concat_rev4 xss) ++ xs
```

Because cords are a tree structure, the foldr version will still do lots on inefficient concatenation (at least in the worst case). To get an efficient version we must use an accumulator but traverse the tree in a different order to that above. In fact, we need to traverse the tree from left to right (the above version traverses it from right to left). All we need to do is exchange a and b in the recursive calls in the last line of the definition.

```
>cord_to_list_rev :: Cord a -> [a]  -- Efficient
>cord_to_list_rev c = cord_to_list_acc c []

>cord_to_list_acc :: Cord a -> [a] -> [a]
>cord_to_list_acc Nil rest = rest
>cord_to_list_acc (Leaf x) rest = x:rest
>cord_to_list_acc (Branch a b) rest =
>    cord_to_list_acc b (cord_to_list_acc a rest)
```

The simple (inefficient) definition of cord_to_list is essentially the same as

```
cord_to_list = cord_foldr (++) (:[]) []
```

The efficient cord_to_list2 is like foldl in its use of an accumulator but also like foldr in that it traverses from right to left.

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.

ANSWER

One possible design is

```
>data Cord' a
>        = EmptyCord
>        | NonEmptyCord (CordNode a)

>data CordNode a
>        = Leaf' a
>        | Branch' (CordNode a) (CordNode a)
```

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).
```

ANSWER

There are obvious differences in syntax. Both use poor choices for variable names. The overall structure is similar, except where Haskell has nested expressions, Mercury has "flattened" conjunctions, which means
we need to invent additional variable names, such as CB. Similarly, the results in Haskell are on the RHS of the equal sign, whereas in Mercury they are arguments in the clause head. The types of the arguments of the Haskell functions and Mercury predicates can be inferred by the two language compilers, or we can declare them. In Mercury, the programmer also has to think about mode and determinism information. The append predicate can be run in several different modes, and the different modes have different determinisms. The evaluation in Mercury is strict whereas it is lazy in Haskell. In this example, Haskell's append can handle infinite lists; it can even append an infinite list in front of another list, provided the program needs to look at only a finite number of elements of the result. This added flexibility comes at a price: the Haskell code is much harder to implement efficiently (thus it tends to be slower than Mercury).

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.

ANSWER
There are many possible ways of solving this task (it is called isInFixOf in the Haskell library). As hinted at in the question, in general there can be several positions in the list which match the start of the sublist and we need to consider all of them. We could scan the list looking for the first element of the sublist, but the code ends up simpler if we just consider every possibly position for matching the entire sublist. We can use structural induction on the list as follows (the base case is slightly tricky - only the empty list is a sublist of the empty list):

```
>is_sublist as [] = as == []
>is_sublist as (b:bs) =
>   is_prefix as (b:bs) || is_sublist as bs
```

is_prefix (prefixOf in the list library) can also be coded using simple structural induction:

```
>is_prefix [] _ = True
>is_prefix (_:_) [] = False
>is_prefix (a:as) (b:bs) = if a == b then is_prefix as bs else False
```

A Mercury version which returns a Bool can be constructed in a very similar way. However, rather than returning a Bool it is generally simpler
to have a semidet predicate (success = True and failure = False). Using the same overall structure, we can get the following:

```
:- pred is_sublist(list(T)::in, list(T)::in) is semidet.

is_sublist([], _).
is_sublist([A | As], [B | Bs]) :-
    (
        is_prefix([A | As], [B | Bs])
    ;
        is_sublist([A | As], Bs)
    ).
```

```
:- pred is_prefix(list(T)::in, list(T)::in) is semidet.

is_prefix([], _).
is_prefix([A | As], [B | Bs]) :-
    A = B,
    is_prefix(As, Bs).
```

It turns out there is a simpler coding in Mercury. As is a substring of Bs if Bs is some postfix appended onto As appended onto some prefix. This can be expressed very simply with the append3 predicate given in lectures:

```
:- pred is_sublist(list(T)::in, list(T)::in) is semidet.

is_sublist(As, Bs) :-
    append3(_Prefix, As, _Suffix, Bs).
```

We can add the following mode declaration for append3:

```
:- mode append3(out, in, out, in) is nondet.

append3(A, B, C, ABC) :-
    append(A, B, AB), append(AB, C, ABC).
```

append3 is nondet in this mode: given values for B and ABC, there is no guarantee that there are values of A and C so that A ++ B ++ C = ABC. However, in some cases, there may be multiple such values A and C (if B occurs in ABC more than once). However, because is_sublist does not care about the values of A and C, it can still be semidet. We could also just use the (out, out, out, in) mode declaration given in the lectures, which implies the mode above.

This solution may be somewhat less efficient than the longer version, but it has the same algorithmic complexity (there are significantly more complicated algorithms which have better complexity). However, with a different mode declaration it can also return all sublists of a list.

The versions above only allow us to find out if the substring exists in the string. One possible more informative output would be the position in which it occurs (an integer). Another would be all positions (a list of integers). Another possibility, which is a useful general technique in parsing and pattern matching is to return the suffix part of the string

(the tail following the substring), or all such substrings. This allows us to easily search string Cs for substring As and later substring Bs (and return the suffix after Bs). The append3 coding can trivially be modified to return the suffix; it will return the suffix corresponding to each match on backtracking:

```
:- pred sublist(list(T)::in, list(T)::in, list(T)::out) is nondet.

sublist(As, Bs, Suffix) :-
    append3(_Prefix, As, Suffix, Bs).

:- pred sublist2(list(T)::in, list(T)::in, list(T)::in, list(T)::out)
    is nondet.

sublist2(As, Bs, Cs, Suffix) :-
    sublist(As, Cs, Mid),
    sublist(Bs, Mid, Suffix).
```

In Haskell we would need to return a list of solutions, where each solution
is the suffix string. An empty outer list corresponds to no solutions. The is_prefix function can be modified to return the suffix, if the match
succeeds. It it natural to use a Maybe type. However, for consistency we could also return a list of solutions, which is what we do here. This leads to an interesting comparison with Mercury code as well as the previous Haskell code (e.g. instead of disjunction we use ++ and instead of False we use []).

```
>sublist as [] = if as == [] then [[]] else []
>sublist as (b:bs) =
>       prefix as (b:bs) ++ sublist as bs

>prefix [] bs = [bs]
>prefix (_:_) [] = []
>prefix (a:as) (b:bs) = if a == b then prefix as bs else []
```

This style is also related to the list monad (not covered in this subject but you might want to look it up). For example, sublist2 can be coded as follows (>>= is like conjunction for nondet predicates).

```
>sublist2 as bs cs = sublist as cs >>= sublist bs
```

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?

ANSWER
It is best to hide the implementation of queues inside a module, which exports just the name of the type and the operations on that type. Here we just use lists and don't hide anything.

```
>queue_new = []
>queue_empty q = (q==[])
>queue_add a q = q ++ [a]
>queue_remove [] = error "remove from empty queue"
>queue_remove (a:q) = (a, q)

:- pred queue_new(list(T)::out) is det.
queue_new([]).

%% instead of returning a Bool we just succeed or fail
%% Note: this has identical logic to queue_new but the mode is different
:- pred queue_empty(list(T)::in) is semidet.
queue_empty([]).

:- pred queue_add(T::in, list(T)::in, list(T)::out) is det.
queue_add(A, Q0, Qs) :- append(Q0, [A], Q).

%% We could make queue_remove fail with an empty queue but it is better
%% to make it an error so the predicate can be det.
%% This way, you get better error checking and faster code.
:- pred queue_remove(list(T)::in, T::out, list(T)::out) is det.
queue_remove([], _, _) :-
    error("remove from empty queue").
queue_remove([A | Q], A, Q).
```

queue_add has O(N) complexity, where N is the queue length. Thus N queue operations, starting with queue_new, can have O(N^2) complexity overall, which is a significant disadvantage. If the order of elements was reversed,
queue_add would be O(1) but queue_remove would be O(N), and the complexity
problem would remain.

The Mercury implementation is very similar, except it is more natural to
have queue_empty succeed or fail rather than explicitly return a Boolean.

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?

ANSWER
With this representation, queue_add can add an element to the front of lr,
which is a constant time operation. queue_remove can *usually* remove the head
of lf, which is also constant time. However, we can run out of elements in lf
even though there some elements left in lr. We need to get the last element
of lr, which takes O(N), but the trick is to get the reverse of lr, which also
takes O(N), and use it as the next lf, making the next N queue_remove operations constant time. Overall, N queue operations, starting with queue_new,
have O(N) complexity, which means that overall, each of the N operations has O(1) *amortised* complexity.

```
>type Queue a = ([a], [a])
>queue_new' = ([], [])
>queue_empty' (lf, lr) = (lf==[] && lr==[])
>queue_add' a (lf, lr) = (lf, a:lr)
>queue_remove' (a:lf, lr) = (a, (lf, lr))
>queue_remove' ([], lr) =
>       let rlr = reverse lr in
>       case rlr of
```

```
>               [] ->
>                       error "remove from empty queue"
>               (a:as) ->
>                       (a, (as, []))
```

The Mercury implementation is similar but we use an explicit data constructor
(function symbol) to form pairs. We might as well give it a meaningful name
such as "queue".

```
:- type queue(T)
    --->    queue(list(T), list(T)).

:- pred queue_new(queue(T)::out) is det.
queue_new(queue([],[])).

%% Instead of returning a Bool, we just succeed or fail.
%% Note: this has identical logic to queue_new but the mode is different.
:- pred queue_empty(queue(T)::in) is semidet.
queue_empty(queue([],[])).

:- pred queue_add(T::in, queue(T)::in, queue(T)::out) is det.
queue_add(A, queue(Fs, Rs), queue(Fs, [A | Rs])).

%% We could make queue_remove fail with an empty queue but it is better
%% to make it an error so the predicate can be det.
%% Logically, we could put the error check in a separate clause
%% (the commented out one below) and simplify the logic of the last clause.
%% However, that would make it too hard for the Mercury compiler to check
%% that it really is det (it would need to figure out that the reverse of
%% a non-empty list is non-empty), so we have to use a structure more like
%% the Haskell code.
:- pred queue_remove(queue(T)::in, T::out, queue(T)::out) is det.
%% queue_remove(queue([], []), _, _) :- error("...").
queue_remove(queue([A | Fs], Rs), A, queue(Fs, Rs)).
queue_remove(queue([], Rs), A, queue(Fs, [])) :-
    reverse(Rs, RRs),
    (
        RRs = [],
```

```
    error("remove from empty queue")
;
    RRs = [A | Fs]
).
```