CPSC 449: Sample questions and revision for the Final Exam

Robin Cockett

December 10, 2019

Haskell

1. Show (by hand) how Haskell evaluates:

   ```
   myappend [1,2] [3,4,5,6]
   ```

   given the code:

   ```
   myappend:: [a] -> [a] -> [a]
   myappend [] bs = bs
   myappend (a:as) bs = a:(myappend as bs)
   ```

2. Show (by hand) how Haskell evaluates:

   ```
   sfList [SS 3,SS 4]
   ```

   Given the code:

   ```
   data SF a = SS a | FF

   sfList:: [SF a] ->  SF [a]
   sfLIst [] = SS []
   sfList (FF:_) = FF
   sfList ((SS a):xs) =
           case (sfList xs) of
                    SS as -> SS (a:as)
                    FF -> FF
   ```

3. Show (by hand) how Haskell evaluates:

```
myOR [False,True,False]
```

   Given the code:

```
foldr:: (a -> b -> b) -> b  -> [a] -> b
foldr f b [] = b
foldr f b (x:xs) = f x (foldr f b xs)

myOR:: [Bool] -> Bool
myOR = foldr myor False

myor:: Bool -> Bool -> Bool
myor  False False = False
myor _ _ = True
```

4. Show (by hand) how Haskell evaluates:

```
unzip [(1,'a'),(2,'b')]
```

   given the code:

```
unzip:: [(a,b)] -> ([a],[b])
unzip [] = ([],[])
unzip ((a,b):xs) = case unzip xs of
          (as,bs) ->  (a:as,b:bs)
```

5. Answer the questions concerning Haskell syntax below:

   (a) Which of the types below are the same type:

      i. `a -> b -> c -> d`
      ii. `(a -> b) -> c -> d`
      iii. `a -> (b -> c) -> d`
      iv. `a -> b -> (c -> d)`
      v. `(a -> b) -> (c -> d)`

   (b) In the following terms what are the types of the functions `f` given that `x,y,z:: Integer`:

      i. `(f x) (y, z)`
      ii. `f x y z`
      iii. `f (x,y,z)`
      iv. `f (x,(y,z))`

(c) Give the types of the following terms (if indeed they type) and indicate which are equal:

   i. `"abcd"`
   ii. `[('a','b'),('c','d')]`
   iii. `('a':['b']):('c':['d'])`
   iv. `'a':('b':'c':'d':[])`
   v. `["ab","cd"]`

6. Given a list of items and a predicate write code to split the list into two lists: a list of item which satisfies the predicate and a list of items which does not:

```
split_list:: (a -> Bool) -> [a] -> ([a],[a])
```

7. In a merge sort one step is to merge two ordered lists of items into one ordered list. Write the code for this step

```
merge:: (Ord a) => [a] -> [a] -> [a]
```

8. Write the code to split a list into two lists such that the elements with odd index are in one list while the elements with even index are in the other list:

```
odd_even_split:: [a] -> ([a],[a])
```

   Can you write this using a `foldr`?

9. Write a function to determine whether a string is a substring of another string:

```
substring :: String -> String -> Bool
```

10. Write a function

```
grow :: String -> String
```

   which changes a string $a_1a_2a_3...$ to $a_1a_2a_2a_3a_3a_3...$ so grow “now!” == “noowww!!!!”.

11. Write a function to produce the list of all the sublists of a list.

12. Why is the following “naive” function for reversing a list $\mathcal{O}(n^2)$:

```
reverse: [a] -> [a]
reverse [] = []
reverse (a:as) = (reverse as) ++ [a]
```

   Give a “fast” $\mathcal{O}(n)$ version of reverse.

13. Why is the complexity of the following program $\mathcal{O}(\mathsf{Fib}(n))$ (assuming a positive input!):

```
fib 0 = 0
fib 1 = 1
fib n = (fib n) + (fib (n+1))
```

Give a $\mathcal{O}(n)$ version of `fib`.

14. A programmer writes the following code but fails to provide types:

```
data SF a = SS a | FF
    deriving (Show,Eq)

myhead [] = FF
myhead (a:as) = SS a

mytail [] = []
mytail (_:xs) = xs

myzip [] _ = []
myzip _ [] = []
myzip (a:as) (b:bs) = (a,b):(myzip as bs)

mystery xs =
   myhead [x|(x,y) <- myzip xs (mytail xs),x==y]
```

   (a) Provide the types for `myhead`, `mytail`, `myzip`, and `mystery`.
   (b) Explain what `mystery` does: what is the result of `mystery "abccdeffghii"`?
   (c) Can you rewrite the code using a `foldr`?

15. A programmer writes the following code but fails to provide a type:

```
mycode f g [] = ([],[])
mycode f g (a:as) = case (mycode f g as) of
         (xs,ys) -> ((f a):xs, (g a):yx)
```

   (a) What is the most general type of the code?
   (b) What is the result of evaluating the following

```
mycode (\a -> a+6)
          (\b -> b `mod` 2 \= 0)
             [3,7,10,2,9,17]
```

   (c) Rewrite `mycode` using a `foldr`.

Monads

1. Given the following list comprehension:

```
graph:: (Eq b) => (a ->b) -> [a] -> [b] -> [(a,b)]
graph f as bs = [ (x,y) | x <- as,y <- bs, f x == y]
```

   (a) What does `graph` do?

   (b) Rewrite the function using the "do" syntax:

   (c) Translate the list comprehension into basic Haskell using the translation scheme, $[\![\_]\!]$:

$$\begin{aligned} [\![[s|\,]]\!] &= [s] \\ [\![[s|x \leftarrow t, r]]\!] &= \mathsf{concat}(\mathsf{map}(\backslash x \rightarrow [\![[s|r]]\!])t) \\ [\![[s|p, r]]\!] &= \mathsf{if}\ p\ \mathsf{then}\ [\![[s|r]]\!]\ \mathsf{else}\ [] \end{aligned}$$

2. Given the list monad defined by:

```
     instance Monad [] where
          return x = [x]
          xs >>= f = concat (fmap f xs)
```

translate the following code in two stages (translate out the do syntax then translate the "return" and "sequencer") into basic Haskell:

```
ord_pairs xs ys = do x <- xs
                     y <-ys
                     compare x y
                     return (y-x)
     where
             compare x y | x < y = [()]
                         | otherwise = []
```

The translation scheme into the "sequencer" code is:

$$\begin{aligned} [\![do\{q\}]\!] &= q \\ [\![do\{x \leftarrow xs; rs\}]\!] &= xs \gg= (\backslash x \rightarrow [\![do\{rs\}]\!]) \\ [\![do\{q; rs\}]\!] &= q \gg= (\backslash\_ \rightarrow [\![do\{rs\}]\!]) \end{aligned}$$

3. Given the exception monad based on the "success or fail" datatype defined by:

```
     data SF a = SS a | FF

     instance Monad SF where
          return x = SS x
          FF >>= _ = FF
          SS x >>= f = f x
```

Translate the following code into basic Haskell syntax in two steps:

```
sflist:: [SF a] -> (SF [a])
sflist [] = SS []
sflist (a:as) = do a' <- a
                   as' <- sflist as
                   return (a':as')
```

Folding and trees

1. Given the following "search tree" datatype

   ```
             data STree a = Sn ( STree a) a (Stree a)
                          | Tp
                        deriving (Show, Eq)
   ```

   The fold for search trees is:

   ```
   foldST g t (Tp) = t
   foldST g t (Sn s1 a s2) = g (foldST g t s1) a (foldST g t s2)
   ```

   (a) What is the most general type of `foldST`:

   (b) Use `foldST` to write a program, `whgt::STree Int -> Int`, to find the "weighted height" of a search tree of integers. This is the greatest sum of integers on any path from the root to a tip of such a tree. Thus, for example:

   ```
   whgt (Sn(Sn Tp 5 Tp) 6 (Sn(Sn Tp 3 Tp) 1 (Sn Tp 2 Tp))) = 11
   ```

2. The `foldr` for lists (as above) is defined as:

   ```
   foldr f g [] = g
   foldr f g (x:xs) = f x (foldr f g xs)
   ```

   (a) What is the most general type of `foldr`?

   (b) Using `foldr` write the function

   ```
             group :: (a ->a -> Bool) -> [a] -> [[a]]
   ```

   which, given a predicate `p::a -> a -> Bool` and a list, breaks the list into a series of (maximal) sublists in which any two consecutive elements satisfy the predicate. For example, suppose that the predicate `nbr` determines whether two integers differ by at most one, then

   ```
   group nbr [2,1,3,4,5,5,4,7,4,3,3]=[[2,1],[3,4,5,5,4],[7],[4,3,3]]
   ```

3. Given the datatype of expressions:

   ```
   data Exp f  v = Opn f [Exp f v]
                 | Var v
   ```

   Expressions are trees with two sorts of nodes: a variable node and an operation node. The operation node has a function "value" `f` which is paired with a list of arguments.

   The fold for the expression type is:

```
foldExp:: (f -> [c] -> c) -> (v -> c) -> (Exp f v) -> c
foldExp g h (Var v) = h v
foldExp g h (Opn f args) = g f (map (foldExp g h) args)
```

(a) Using `foldr` on list write a function to calculate the minimum of a list of integers: `mins:  [Int] -> Int`.

(b) Similarly, using `foldr` on list write a function to calculate the maximum of a list of integers: `maxs:  [Int] -> Int`.

(c) Now, using the fold for expressions write a program `minmax::  Exp Bool Int -> Int`, which takes the maximum of the arguments when function value is `True` and takes the minimum of the arguments when the function value is `False`.

4. (harder!) A rose tree with values at the arguments of the nodes,

```
data Rose a = Rs [(a,Rose a)]
```

has its "tips" given by `Rs []` and may be regarded as a weighted tree (the weights are of type `a`) with the weights on the edges. The fold for this rose tree is:

```
foldRose g (Rs args)
       = g (map (\(a,arg) -> (a,foldRose g arg)) args)
```

(a) What is the type of `foldRose`?

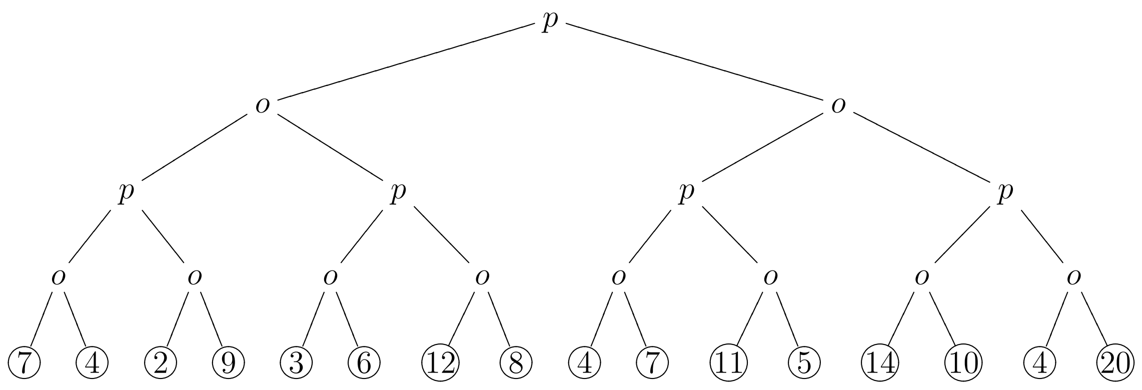
(b) Use the fold on rose trees to find the weighted width of a tree

$$\texttt{width} :: \texttt{RoseInt}- > \texttt{Int}.$$

The weighted width is the maximum sum of the edges on any path from tip to tip in the tree. For example:

$$\texttt{width}(\texttt{Rs}[(4,\texttt{Rs}[]),(1,\texttt{Rs}[]),(6,\texttt{Rs}[]),(3,\texttt{Rs}[])]) = 10.$$

5. Given the following minmax tree indicate where the $\alpha - \beta$ cut-offs occur for the maximizing player:



Prolog

1. What does the unification algorithm do? What is the "occurs check"? Why do many implementations of prolog avoid the occurs check?

2. Give the most general unifier for the following terms:

   (a) Unify $f(g(x, y), z)$ and $f(w, g(w, z))$

   (b) Unify $f(x, g(y, z))$ and $f(g(z, y), x)$

   (c) Unify $f(g(x, y), z)$ and $f(z, h(x, y))$

3. Hand evaluate `member(X, [a,b,c])` and explore all the backtracking behaviour, where:

   ```
   member(X,  [X|_]).
   member(X,[_|T]):- member(X,T).
   ```

   Give the different modes in which the `member` predicate can be used to produce a finite behaviour (the above is `member(-,+)`).

4. Hand evaluate `append(X,Y,[a,b])` and explore all the backtracking behaviour, where:

   ```
   append([], X,X).
   append[H|T],X,[H|Y]):- append(T,X,Y).
   ```

   Give the different modes in which the `append` predicate can be used to produce a finite behaviour (the above is `append(-,-,+)`).

5. Hand evaluate `insert(c,[a,b],Z)`.

   ```
   insert(X,Y,[X|Y]).
   insert(X,[H|T],[H|Y]):- insert(X,T,Y).
   ```

   Give the different modes in which the `insert` predicate can be used to produce a finite behaviour (the above is `insert(+,+,-)`).

6. Describe the behaviour of `reverse(X,Y)`:

   ```
   reverse([],[]).
   reverse([H|T],Y):- reverse(T,S), append(S,[H],Y).
   ```

   What happens when it is used in the mode `reverse(-,+)`? Can one improve this behaviour?

   This is a naive reverse with complexity $\mathcal{O}(n^2)$: write an efficient reverse and describe its behaviour when used in different modes.

7. Write a predicate `perm/2` which determines whether two lists are permutations of each other. What is its behaviour under different modes of use?

8. One way to sort a list is called a "permutation sort" and involves trying every permutation and checking whether the permutation is in order. Write a permutation sort for a list of integers. What is its complexity?

   Write an efficient sort predicate in Prolog (e.g. `quicksort(+,-)` or `mergesort(+,-)`).

9. Explain what a cut does in Prolog? In Prolog what is the meaning of `not(P(x))` and how is it computed? Give the code for `not`.

10. What does the following program do?

```
printer(L):-
    member(X,L),
    write(X),
    nl,
    false.
 printer(_).
```

11. Write a predicate, `equal_set(+,+)`, to determine whether two list are equal *as sets* (order and repetitions do not matter)

12. Write a predicate to sum every other (first, third, fifth, seventh, ...) element of a list.

13. Write a predicate which grows a list, `grow(+,-)`.

14. Write a predicate which, from a list of integers, extracts all the maximal sublists which are in ascending order, `group(+,-)`.

General knowledge

Who are the following people? When did they live? Where did they live? What did they do?

(a) Basile Bouchon

(b) Jean-Baptiste Falcon

(c) Jacques de Vaucanson

(d) Joseph Marie Jacquard

(e) Napoleon Bonaparte

(f) Charles Babbage

(g) Ada Lovelace

(h) Herman Hollerith

(i) Paul Otlet

(j) George Stibitz

(k) Konrad Zuse

(l) Alan Turing

(m) Alonzo Church

(n) Haskell Curry

(o) Greta Thunberg