Spring 2017

Final Exam • Two Parts: • Part 2: • 8 questions worth 38 marks • Open book / open Internet • On the computer •

Q1: Write a Haskell function (can be completed with pre-midterm concepts) •
Q2: Write a Haskell function that uses a fold •
Q3: Write a polymorphic Haskell function •
Q4: Write a Haskell function that operates on a data type that I have defined •
Q5: Write a Haskell function involving an infinite list •
Q6: Write a non-recursive Prolog rule •
Q7: Write a recursive Prolog rule involving lists and a little bit of math •
Q8: Write a recursive Prolog rule involving a little bit of math and a cut

```
--  Q1
okPassword :: String -> Bool
okPassword p = length p >= 8 &&
                 length (filter (\x -> 'A' <= x && x <= 'Z') p) > 0 &&
                 length (filter (\x -> 'a' <= x && x <= 'z') p) > 0 &&
                 length (filter (\x -> '0' <= x && x <= '9') p) > 0

--  Q2
pzn_help :: (Int, Int, Int) -> Int -> (Int, Int, Int)
pzn_help (p, z, n) x
  | x < 0  = (p, z, n+1)
  | x == 0 = (p, z+1, n)
  | x > 0  = (p+1, z, n)

pzn :: [Int] -> (Int, Int, Int)
pzn vals = foldl pzn_help (0, 0, 0) vals

--  Q3
closest :: (Ord a, Num a) => a -> [a] -> a
closest n [] = error "The list of values to search is empty"
closest n (x : []) = x
closest n (v1 : v2 : vs)
  | n <= v1 = v1
  | v1 < n && n < v2 && (n - v1) < (v2 - n) = v1
  | v1 < n && n < v2 = v2
  | otherwise = closest n (v2 : vs)

--  Q4
data TernaryTree = Internal String TernaryTree TernaryTree TernaryTree |
                   Leaf String deriving (Show, Eq)

countNodes :: TernaryTree -> (Int, Int)
countNodes (Leaf _) = (0, 1)
countNodes (Internal _ left mid right) = (1 + fst lc + fst mc + fst rc,
                                          snd lc + snd mc + snd rc)
  where
    lc = countNodes left
    mc = countNodes mid
    rc = countNodes right

--  Q5
--
random_list :: Int -> [Int]
random_list seed = [(next_seed - 1) `mod` 10 + 1] ++ random_list next_seed
  where next_seed = (7 * seed) `mod` 101
```

_____--

```
/*
*
** Final exam starter code for Q6 and Q8
*/

/*
** Starter code for Q6
*/
male(bob).
male(charles).
male(evan).
male(gordon).
female(alice).
female(diane).
female(fiona).
female(harriot).

likes(bob, pizza).
likes(evan, pizza).
likes(charles, alice).
likes(evan, fiona).
likes(gordon, fiona).
likes(fiona, charles).
likes(fiona, evan).
likes(charles, golf).
likes(evan, golf).
likes(gordon, golf).
likes(alice, shopping).
likes(diane, shopping).
likes(diane, bob).
likes(fiona, golf).
likes(harriot, golf).
likes(diane, steak).
likes(fiona, steak).

/*
** Starter code for Q8
*/

/*
** Define the precedence of the opearators.
*/
precedence(+, X) :- X = 1.
precedence(-, X) :- X = 1.
precedence(*, X) :- X = 2.
precedence(/, X) :- X = 2.
precedence(^, X) :- X = 3.

/*
** Succeed if and only if the parameter is a leaf node in the tree (or
** more accurately if it is not an internal node -- if something is passed
** that has nothing to do with the tree then this rule will succeed).
*/
isLeaf(X) :- X = term(_, _, _), !, fail.
isLeaf(_).

/*
** Build a tree for a list of numbers by placing an operator between each
** number (following appropriate rules for order of operations).
*/
build([H | T], Tree) :-
  buildHelper(H, T, Tree).
```

```
buildHelper(Current, [], Current).
buildHelper(Current, [H | T], Result) :-
  member(Op, [+, -, *, /, ^]),
  addNext(Current, Op, H, NewTree),
  buildHelper(NewTree, T, Result).

/*
** Add another operator and operand to the end of expression represented by
** the tree, ensuring that proper order of operations rules are respected.
*/
addNext(Initial, Operator, Operand, Result) :-
  isLeaf(Initial),
  Result = term(Operator, Initial, Operand).
addNext(Initial, Operator, Operand, Result) :-
  Initial = term(Op, Left, Right),
  precedence(Op, P1),
  precedence(Operator, P2),
  P1 >= P2,
  Result = term(Operator, term(Op, Left, Right), Operand).
addNext(Initial, Operator, Operand, Result) :-
  Initial = term(Op, Left, Right),
  precedence(Op, P1),
  precedence(Operator, P2),
  P1 < P2,
  Result = term(Op, Left, term(Operator, Right, Operand)).

/*
** Numerically evaluate a tree (compute the value the expression represents).
*/
eval(Tree, Result) :- isLeaf(Tree), Result = Tree.
eval(Tree, Result) :-
  (Tree = term(Op, Left, Right),
   eval(Left, LResult),
   eval(Right, RResult)),
  ((Op = (+), Result is LResult + RResult);
   (Op = (-), Result is LResult - RResult);
   (Op = (*), Result is LResult * RResult);
   (Op = (/), RResult =:= 0, !, fail);
   (Op = (/), Result is LResult / RResult);
   (Op = (^), Result is LResult ^ RResult)).

/*
** Convert the tree to a string so that it is easier to see what expression
** the tree actually represents.
*/
inOrder(Tree, String) :- isLeaf(Tree), number_codes(Tree, String), !.
inOrder(Tree, String) :-
  Tree = term(Op, Left, Right),
  inOrder(Left, LeftString),
  inOrder(Right, RightString),
  atom_codes(Op, MiddleString),
  append(LeftString, MiddleString, TempString),
  append(TempString, RightString, String).

/*
** Identify expressions that use the numbers in values and a selection of
** operators to equal the desired answer.
*/
numberPuzzle(Values, DesiredAnswer, Expression) :-
  build(Values, Tree),
  eval(Tree, TreeAns),
  TreeAns =:= DesiredAnswer,
  Expression = Tree.
```

```
**  Q6
*/
canBeFriends(X, Y) :-
  male(X),
  male(Y),
  X \= Y,
  likes(X, Z),
  likes(Y, Z),
  \+female(Z).
canBeFriends(X, Y) :-
  female(X),
  female(Y),
  X \= Y,
  likes(X, Z),
  likes(Y, Z),
  \+male(Z).
canBeFriends(X, Y) :-
  ((male(X), female(Y)) ; (female(X), male(Y))),
  \+likes(X, Y),
  \+likes(Y, X),
  likes(X, Z),
  likes(Y, Z).

/*
**  Q7
*/
elementAt([_ | T], Position, Result) :-
  Position > 0,
  PM1 is Position - 1,
  elementAt(T, PM1, Result).
elementAt([H | _], 0, H).
elementAt(Values, Position, Result) :-
  Position < -1,
  PP1 is Position + 1,
  append(AllButLast, [_], Values),
  elementAt(AllButLast, PP1, Result).
elementAt(Values, -1, Result) :-
  append(_, [X], Values),
  Result = X.

/*
**  Q8
*/
search(Values, Current, NoMatch) :-
  \+numberPuzzle(Values, Current, _),
  NoMatch = Current.
search(Values, Current, NoMatch) :-
  numberPuzzle(Values, Current, _),
  !,
  Next is Current + 1,
  search(Values, Next, NoMatch).

/*
** Final exam starter code for Q6 and Q8
*/

/*
** Starter code for Q6
*/
male(bob).
```

```
male(charles).
male(evan).
male(gordon).
female(alice).
female(diane).
female(fiona).
female(harriot).

likes(bob, pizza).
likes(evan, pizza).
likes(charles, alice).
likes(evan, fiona).
likes(gordon, fiona).
likes(fiona, charles).
likes(fiona, evan).
likes(charles, golf).
likes(evan, golf).
likes(gordon, golf).
likes(alice, shopping).
likes(diane, shopping).
likes(diane, bob).
likes(fiona, golf).
likes(harriot, golf).
likes(diane, steak).
likes(fiona, steak).

/*
** Starter code for Q8
*/

/*
** Define the precedence of the opearators.
*/
precedence(+, X) :- X = 1.
precedence(-, X) :- X = 1.
precedence(*, X) :- X = 2.
precedence(/, X) :- X = 2.
precedence(^, X) :- X = 3.

/*
** Succeed if and only if the parameter is a leaf node in the tree (or
** more accurately if it is not an internal node -- if something is passed
** that has nothing to do with the tree then this rule will succeed).
*/
isLeaf(X) :- X = term(_, _, _), !, fail.
isLeaf(_).

/*
** Build a tree for a list of numbers by placing an operator between each
** number (following appropriate rules for order of operations).
*/
build([H | T], Tree) :-
  buildHelper(H, T, Tree).

buildHelper(Current, [], Current).
buildHelper(Current, [H | T], Result) :-
  member(Op, [+, -, *, /, ^]),
  addNext(Current, Op, H, NewTree),
  buildHelper(NewTree, T, Result).

/*
** Add another operator and operand to the end of expression represented by
** the tree, ensuring that proper order of operations rules are respected.
```

```
*/
addNext(Initial, Operator, Operand, Result) :-
  isLeaf(Initial),
  Result = term(Operator, Initial, Operand).
addNext(Initial, Operator, Operand, Result) :-
  Initial = term(Op, Left, Right),
  precedence(Op, P1),
  precedence(Operator, P2),
  P1 >= P2,
  Result = term(Operator, term(Op, Left, Right), Operand).
addNext(Initial, Operator, Operand, Result) :-
  Initial = term(Op, Left, Right),
  precedence(Op, P1),
  precedence(Operator, P2),
  P1 < P2,
  Result = term(Op, Left, term(Operator, Right, Operand)).

/*
** Numerically evaluate a tree (compute the value the expression represents).
*/
eval(Tree, Result) :- isLeaf(Tree), Result = Tree.
eval(Tree, Result) :-
  (Tree = term(Op, Left, Right),
   eval(Left, LResult),
   eval(Right, RResult)),
  ((Op = (+), Result is LResult + RResult);
   (Op = (-), Result is LResult - RResult);
   (Op = (*), Result is LResult * RResult);
   (Op = (/), RResult =:= 0, !, fail);
   (Op = (/), Result is LResult / RResult);
   (Op = (^), Result is LResult ^ RResult)).

/*
** Convert the tree to a string so that it is easier to see what expression
** the tree actually represents.
*/
inOrder(Tree, String) :- isLeaf(Tree), number_codes(Tree, String), !.
inOrder(Tree, String) :-
  Tree = term(Op, Left, Right),
  inOrder(Left, LeftString),
  inOrder(Right, RightString),
  atom_codes(Op, MiddleString),
  append(LeftString, MiddleString, TempString),
  append(TempString, RightString, String).

/*
** Identify expressions that use the numbers in values and a selection of
** operators to equal the desired answer.
*/
numberPuzzle(Values, DesiredAnswer, Expression) :-
  build(Values, Tree),
  eval(Tree, TreeAns),
  TreeAns =:= DesiredAnswer,
  Expression = Tree.
```

Fall 2017:

Q1) Construct a list of some sort using lazy evaluation, 2 parts

Q2) Solve a problem using fold. Fold a function over a list (slightly easier than first midterm)

Q3) create a new type (3 parts)

- Build up a Type in a sequential type of way
- Last part write a small function involving new Type

Prolog: Knowledge Base will be provided with Facts

Q4) Create two rules involving the facts. Will involve lists, arithmetic and recursion. Section 8.20 in gProlog has some functions that he used in his solution...(list predicates)

- Use findall (let's you capture them in a list)
- No need for cuts

Q5) knowledge with facts and a rule that solves a small version of the problem

- Write 2nd version of rule that solves the other version of the problem
- Lists, recursion, and a bit more of arithmetic
- Base case is given, will have to make recursive case.

MC: Function composition, minimal coding, straight up recall

```
--------------------------------------------------------------------------------
--
--  Q1
--
--------------------------------------------------------------------------------
getSequence :: Int -> [Int]
getSequence 1 = [1]
getSequence n
  | n `rem` 2 == 0 = n : getSequence (n `div` 2)
  | otherwise      = n : getSequence (3 * n + 1)

findSequences :: Int -> Int -> [Int]
findSequences number target = take number (map head (filter (\x -> length x ==
target) (map getSequence [1..])))


--------------------------------------------------------------------------------
--
--  Q2
--
--------------------------------------------------------------------------------
getPreyLists :: [(String, String)] -> [(String, [String])]
getPreyLists values = foldl plHelper [] values

plHelper :: [(String, [String])] -> (String, String) -> [(String, [String])]
plHelper [] (pred, prey) = [(pred, [prey])]
plHelper ((pred, prey):rest) (npred, nprey)
  | pred == npred = (pred, prey ++ [nprey]) : rest
  | otherwise     = (pred, prey) : plHelper rest (npred, nprey)

--------------------------------------------------------------------------------
--
--  Q3
--
--------------------------------------------------------------------------------
```

```
data Eon = Hadean | Archean | Proterozoic | Phanerozoic deriving (Eq, Ord, Enum)

instance Show Eon where
  show Hadean = "Hadean (4.6 billion to 4.0 billion years ago)"
  show Archean = "Archean (4.0 billion to 2.5 billion years ago)"
  show Proterozoic = "Proterozoic (2.5 billion to 541 million years ago)"
  show Phanerozoic = "Phanerozoic (541 million years ago to present)"

yearToEon :: Int -> Eon
yearToEon year
  | -4600000000 <= year && year < -4000000000 = Hadean
  | -4000000000 <= year && year < -2500000000 = Archean
  | -2500000000 <= year && year <  -541000000 = Proterozoic
  |  -541000000 <= year && year <        2018 = Phanerozoic
  | otherwise                                 = error "Year outside of supported
range"
```

```
%
%  Q4:
%
%  This is the contents of aquatic.pro
%
eats(bird,prawn).
eats(bird,mussels).
eats(bird,crab).
eats(bird,limpets).
eats(bird,whelk).
eats(crab,mussels).
eats(crab,limpets).
eats(fish,prawn).
eats(limpets,seaweed).
eats(lobster,crab).
eats(lobster,mussels).
eats(lobster,limpets).
eats(lobster,whelk).
eats(mussels,phytoplankton).
eats(mussels,zooplankton).
eats(prawn,zooplankton).
eats(whelk,limpets).
eats(whelk,mussels).
eats(zooplankton,phytoplankton).

%
% An organism is a producer if it is eaten by something else and doesn't
% eat anything.
%
isProducer(X) :- eats(_, X), \+eats(X, _).

%
% Producers have height 0.  All other organisms have height equal to the
% maximum height of anything they eat, plus one.
%
height(X, H) :- isProducer(X), H=0, !.
height(X, H) :- findall(Y, eats(X, Y), EatList),
                maplist(height, EatList, Heights),
                max_list(Heights, MaxHeight),
                H is MaxHeight + 1.


%
%  Q5: All of the flight route information from Flights.pro is later in the
```

```
%      file.
%

%
%  This is the routeAfter rule given in the question
%
routeAfter(C1, C2, SoonestAllowed, DepartureTime, ArrivalTime, Route) :-
  flight(C1, C2, DepartureTime, Duration),
  DepartureTime >= SoonestAllowed,
  ArrivalTime is DepartureTime + Duration,
  Route = [fly(C1, C2)].

%
%  This is the recursive version of the rule that allows connecting
%  flights to be calculated
%
routeAfter(C1, C2, SoonestAllowed, DepartureTime, ArrivalTime, Route) :-
  % Find some flight departing from the desired city that is long enough
  % in the future and compute its arrival time
  flight(C1, X, DepartureTime, Duration),
  DepartureTime >= SoonestAllowed,
  FlightArrival is DepartureTime + Duration,

  % Find a route from where we currently are in city X to C2, ensuring that
  % the next flight leaves at least 60 minutes later
  RouteWhen is FlightArrival + 60,
  routeAfter(X, C2, RouteWhen, RouteDeparture, RouteArrival, RouteRoute),
  Layover is RouteDeparture - FlightArrival,
  ArrivalTime = RouteArrival,
  Route = [fly(C1, X), layover(Layover) | RouteRoute],

  % Ensure that we aren't using a city that is already in the route
  \+member(fly(_, C1), RouteRoute).


%
%  This is the contents of Flight.pro
%
flight(jfk, fll, 1180, 200).
flight(las, jfk, 649, 296).
flight(las, lax, 382, 83).
flight(mia, atl, 1140, 127).

.........................
```