Programming Language Concepts, cs2104
Tutorial 1. Answers

Exercise 1. (Variables and Cells) Given:

```
      local X in
      X=23
      local X in
      X=44
      end
      {Browse X}
      end
```

The second uses a cell:

```
      local X in
      X={NewCell 23}
      X:=44
      {Browse @X}
      end
```

In the first, the identifier X refers to two different variables. In the second, X refers to a cell. What does Browse display in each fragment? Explain.

Answer.
First program: 23, since {Browse X} is visible to the outermost "local" statement.
Second program: 44, because X is a multiple-assignment variable (cell).

Exercise 2. (Accumulators) This exercise investigates how to use cells together with functions. Let us define a function {Accumulate N} that accumulates all its inputs, i.e., it adds together
all the arguments of all calls. Here is an example:

```
      {Browse {Accumulate 5}}
      {Browse {Accumulate 100}}
      {Browse {Accumulate 45}}
```

This should display 5, 105, and 150, assuming that the accumulator contains zero at the start. Here is a wrong way to write Accumulate:

```
      declare
      fun {Accumulate N}
      Acc in
        Acc={NewCell 0}
        Acc:=@Acc+N
        @Acc
      end
```

What is wrong with this definition? How would you correct it?

Answer.
It will display: 5 100 45
The reason is because the cell Acc is local to every call of Accumulate. A possible solution is to make the cell Acc visible for all calls of Accumulate.

```
      declare
      Acc={NewCell 0}
      fun {Accumulate N}
        Acc:=@Acc+N
        @Acc
      end
      {Browse {Accumulate 5}}
      {Browse {Accumulate 100}}
      {Browse {Accumulate 45}}
```

Exercise 3. (Values as trees) Given is the following declaration and assignment.

```
declare
      R=r(1:[a b c] 4:[d [e [f]]] z:q(g h [10 11 [12 13]]))
```

Draw the value as a tree. In the following you have to give an expression composed of dot, width, and label functions that return the desired value. For example, for the value g the expression is R.z.1 and for r it is {Label R}. If there is more than one possibility, give at least two expressions.

```
1. b        2. q              3. 12       4. nil
5. '|'            6. 3        7. h        8. 2
```

Answer.

1. the value b can be given by the expression: R.1.2.1
2. the value q can be given by the expression: {Label R.z}
3. the value 12 can be given by the expression: R.z.3.2.2.1.1
4. the value nil can be given by the expressions:
   a. R.1.2.2.2
   b. R.4.2.2
5. the value '|' can be given by the expressions:
   a. {Label R.1}
   b. {Label R.1.2}
6. the value 3 can be given by the expressions:
   a. {Width R}
   b. {Width R.z}
7. the value h can be given by the expression: R.z.2
8. the value 2 can be given by the expressions:
   a. {Width R.1}
   b. {Width R.1.2}

Exercise 4. (Traversing trees) We have seen a way to traverse in preorder (first the root, then the left child, followed by the right child). It is:

```
      declare
      Root=node(left:X1 right:X2 value:0)
      X1=node(left:X3 right:X4 value:1)
      X2=node(left:X5 right:X6 value:2)
      X3=node(left:nil right:nil value:3)
      X4=node(left:nil right:nil value:4)
      X5=node(left:nil right:nil value:5)
      X6=node(left:nil right:nil value:6)
      {Browse Root}
      proc {Preorder X}
         if X \= nil then {Browse X.value}
           if X.left \= nil then {Preorder X.left} end
           if X.right \= nil then {Preorder X.right} end
         end
      end
      {Preorder Root}
```

Design the other known strategies, namely traverse in inorder (first the left child, then the root, followed by the right child) and postorder (first the left child, then the right child, followed by the root).

Answer.

```
      proc {Inorder X}
           // pre X \= nil
         if X.left \= nil then {Inorder X.left} end
         {Browse X.value}
         if X.right \= nil then {Inorder X.right} end
         end
      end
      proc {Postorder X}
          // pre x \= nil
         if X.left \= nil then {Postorder X.left} end
         if X.right \= nil then {Postorder X.right} end
         {Browse X.value}
        end
```

Using case:

```
      proc {Inorder X}
         case X of nil then skip
           [] node(left:L value:V right:R) then
               {Inorder L} {Browse V} {Inorder R} end
      end
```

Exercise 5. (Pattern Matching for Head and Tail)
Give definitions for Head and Tail that use pattern matching.

Answer.

```
      declare
      fun {Head X|_}
         X
      end
      fun {Tail _|X}
         X
      end
      {Browse {Head [1 2 3]}}
      {Browse {Tail [1 2 3]}}
```

Exercise 6. (Length of a List) Try the version of Length as presented in the
lecture.
Since X from the pattern is not used in the right-hand side of the case, it can
be
replaced with the universal operator ("_" will not be bound to anything). Write
other
version, using the equality operator. Is there any more efficient way to
implement Length?

Answer.
Using universal operator ("_"):

```
      fun {Length Xs}
         case Xs of
               nil then 0
         [] _|Xr then 1+{Length Xr} end
      end
```

Using the equality operator:

```
      fun {Length L}
         if L==nil then 0
         else     1 + {Length {Tail L}} end
      end
```

==
=

Programming Language Concepts, cs2104
Tutorial 2. Answers

Exercise 1. (Finding an Element in a List) Give a definition of {Member Xs Y} that tests whether Y is an element of Xs. For this assignment you have to use the truth values true and false. The equality test (that is ==) returns truth values and a function returning truth values can be used as condition in an if-expression. For example, the call {Member [a b c] b} should return true, whereas {Member [a b c] d} should return false.

Answer.

```
declare
fun {Member Xs Y}
   case Xs of
      nil then false
   [] H|T then
      if H==Y then true
      else {Member T Y}
      end
   end
end
{Browse {Member [a b c] d}}
{Browse {Member [a b c] b}}
```

Exercise 2. (Taking and Dropping Elements) Write two functions {Take Xs N} and {Drop Xs N}. The call {Take Xs N} returns the first N elements of Xs whereas the call {Drop Xs N} returns Xs without its first N elements. For example, {Take [1 4 3 6 2] 3} returns [1 4 3] and {Drop [1 4 3 6 2] 3} returns [6 2].

Answer.
Solution 1.

```
declare
fun {Take Xs N}
  if N==0 then nil
  else if Xs \= nil then
        Xs.1|{Take Xs.2 N-1}
       else error
       end
  end
end
{Browse {Take [1 4 3 6 2] 3}}
{Browse {Take [1 4 3 6 2] 7}}
fun {Drop Xs N}
  if N==0 then Xs
  else if Xs \= nil then {Drop Xs.2 N-1}
       else error
       end
  end
end
{Browse {Drop [1 4 3 6 2] 4}}
{Browse {Drop [1 4 3 6 2] 6}}
```

Solution 2. The Take function is:

```
declare
fun {TakeAux Xs N I}
   case Xs of
      nil then if I =< N then error end
   [] H|T andthen I =< N then H|{TakeAux T N I+1}
```

```
      else nil
   end
end
fun {Take Xs N}
   {TakeAux Xs N 1}
end
{Browse {Take [1 4 3 6 2] 3}}
{Browse {Take [1 4 3 6 2] 7}}
```

The Drop function is:

```
declare
fun {DropAux Xs N I}
   case Xs of nil then nil
   [] H|T then
      if I < N then {DropAux T N I+1}
      else T
      end
   end
end
fun {Drop Xs N}
   {DropAux Xs N 1}
end
{Browse {Drop [1 4 3 6 2] 7}}
{Browse {Drop [1 4 3 6 2] 3}} % returns [6 2].
```

Exercise 3. (Zip and UnZip) Two important functions that convert pairlists to pairs of lists and vice versa are Zip and UnZip.
a) Implement a function Zip that takes a pair Xs#Ys of two lists Xs and Ys (of the same length) and returns a pairlist, where the first field of each pair is taken from Xs and the second from Ys. For example, {Zip [a b c]#[1 2 3]} returns the pairlist [a#1 b#2 c#3].
b) The function UnZip does the inverse, for example {UnZip [a#1 b#2 c#3]} returns [a b c]#[1 2 3]. Give a specification and implementation of UnZip.

Answer.
a) The first solution refers to the case when the lists have the same length:

```
declare
fun {Zip Xs#Ys}
   case Xs#Ys
   of nil#nil then nil
   [] (X|Xr)#(Y|Yr) then X#Y|{Zip Xr#Yr}
   end
end
{Browse {Zip [a b c]#[1 2 3]}}
```

a) The second solution covers the cases when the lists may have also different lengths:

```
declare
fun {Zip X}
   case X of nil#nil then nil
   [] X#nil then {Browse 'First list is too long'} X
   [] nil#X then {Browse 'Second list is too long'} X
   [] (H1|T1)#(H2|T2) then
      H1#H2|{Zip T1#T2}
   end
end
{Browse {Zip [a b c]#[1 2 3]}}
```

```
{Browse {Zip [a b c d]#[1 2 3]}}
{Browse {Zip [a b c]#[1 2 3 4]}}
```

b) A condensed solution:

```
declare
fun {UnZip XYs}
   case XYs
   of nil then nil#nil
   [] X#Y|XYr then
      Xr#Yr={UnZip XYr}
   in
      (X|Xr)#(Y|Yr)
   end
end
{Browse {UnZip [a#1 b#2 c#3]}}
```

b) An equivalent solution, where the code is explained in some details:

```
declare
fun {Unzip X}
   case X
   of nil then nil#nil
   [] (H1#H2|T) then
      Local Xr Yr      %Xr and Yr are local variables
      in
       Xr#Yr={Unzip T}  %when coming back from the recursion,
                        %Xr and Yr will be bound
       (H1|Xr)#(H2|Yr)  %construct the returned value
      end
   end
end
{Browse {Unzip [a#1 b#2 c#3]}}
```

Exercise 4. (Finding the Position of an Element in a List) Write a function {Position Xs Y} that returns the first position of Y in the list Xs. The positions in a list start with 1. For example, {Position [a b c] c} returns 3 and {Position [a b c b] b} returns 2.
Try two versions:
1) one that assumes that Y is an element of Xs and
2) one that returns 0, if Y does not occur in Xs.

Answer.
Solution 1. First version (assumes that element is included):

```
fun {Position Xs Y}
  case Xs of
    X|Xr then
      if X==Y then 1 else 1+{Position Xr Y} end
  end
end
```

Solution 2. Second version (not very efficient):

```
fun {Position Xs Y}
  case Xs
  of nil then 0
  [] X|Xr then
    if X==Y then 1
    else N={Position Xr Y} in
      if N==0 then 0 else N+1 end
    end
  end
end
```

Solution 3. We give an Oz program which embeds both versions, being a better solution because it is using a tail-recursive version with an accumulator:

```
declare
fun {PositionAux Xs Y Pos}
   case Xs of
      nil then 0
   [] H|T then
      if H==Y then Pos
      else {PositionAux T Y Pos+1}
      end
   end
end
fun {Position Xs Y}
   {PositionAux Xs Y 1}
end
{Browse {Position [a b c] c}}
{Browse {Position [a b c b] b}}
```

Exercise 5. (Arithmetic Expressions Evaluation) Suppose that you are given an arithmetic expression described by a tree constructed from tuples as follows:
1. An integer is described by a tuple int(N), where N is an integer.
2. An addition is described by a tuple add(X Y), where both X and Y are arithmetic expressions.
3. A multiplication is described by a tuple mul(X Y), where both X and Y are arithmetic expressions.
Implement a function Eval that takes an arithmetic expression and returns its value. For example, add(int(1) mul(int(3) int(4))) is an arithmetic expression and its evaluation returns 13.

Answer. (a complete solution has been done in Lecture 4, when dealing with exceptions)

```
fun {Eval X}
  case X
    of int(N) then N
    [] add(X Y) then {Eval X}+{Eval Y}
    [] mul(X Y) then {Eval X}*{Eval Y}
  end
end
{Browse {Eval add(int(1) mul(int(3) int(4)))}}
```

==
===========

Programming Language Concepts, cs2104
Tutorial 3. Answers

```
%Exercise1
declare
fun {Filter P Ls}
   case Ls of
```

```
      nil then nil
   [] H|T then if {P H} then H|{Filter P T}
             else {Filter P T} end
   end
end
{Browse {Filter (fun {$ N} if N mod 2==0 then N>=0 else false end) [1 ~2 3 ~4]}}

%Exercise2
declare
fun {FoldL Op Z L}
   case L of
      nil then Z
   [] H|T then {FoldL Op {Op Z H} T}
   end
end
fun {FoldR Op Z L}
   case L of
      nil then Z
   [] H|T then {Op H {FoldR Op Z T}}
   end
end
fun {Max2 Z H}
   if Z==neginf then H
   else if Z>H then Z else H end
   end
end
fun {MaxL L}
   {FoldL Max2 neginf L}
end
fun {MaxR L}
   {FoldR fun {$ H Z} {Max2 Z H} end neginf L}
end
{Browse {MaxL [2 4 ~6 100 40]}}
{Browse {MaxR [2 4 ~6 100 40]}}
{Browse {MaxL [~6 ~10]}}
{Browse {MaxR [~6 ~2]}}
{Browse {MaxL nil}}
{Browse {MaxR nil}}

% Exercise : Try implement Filter using FoldR!

%Exercise3
declare
fun {MapTuple T F}
   local
      W={Width T}
      NT={MakeTuple {Label T} W}
      in
   for I in 1..W do
      NT.I={F T.I}
   end
   NT
   end
end
{Browse {MapTuple a(1 2 3 4) fun {$ N} N*N end}}

%Exercise4
```

```
declare
fun {Fact N}
   if N==0 then 1 else N*{Fact N-1} end
end
fun {FactList N}
   if N==0 then nil
    else {Fact N}|{FactList N-1} end
end
fun {FactLTup N}
   if N==0 then 1#nil
   else case {FactLTup N-1} of
         F#L then local NF=N*F in
                    NF#(NF|L) end
      end
   end
end
{Browse {FactList 10}}
{Browse {FactLTup 10}.2}
```