```
Tutorial 4
==========

Exercise 1. (Free/Bound) Indicate which occurrences of variables are bound and
which ones are free in the following expressions. # marks the free vars.

1.    [\ x . z#  (x (\ x. y#(z#)) )]  x#


2.    (\ a b . c# d# a b) a# b# (\ c d . d c) (\ e f . f) e#

3.    [ (\ u v . \ w. w (\ x. x(u)) (v)) (v#) ]  (\ z. \ y. z(y))

Exercise 2. (Substitutions) Perform the following substitutions :

1     [ x  -> \ z . w  ] (\ y . x)
            = \ y. (\z. w)

2     [ x  -> \ z . w  ] (\ y . x  x)
            = (\ y . (\z.w)  (\z.w))

3     [ x  -> \ z . w  ] (\ y . x ((\ x . x))
            = (\ y . (\z.w) ((\ x . x))

4     [ x  -> \ z . w  ] (\ x . y)
            = (\x. y)

5     [ x  -> \ z . w  ] (\ w . x)
            = [ x  -> \ z . w  ] (\ u . x)
            = (\ u. (\z.w))

6     [ x  -> \ z . w  ] (\ z . x)
            = (\z. (\z. w))

7     [ x  -> \ z . w  ] (\ z . z x)
            = (\z. z (\z.w))

8     [ x  -> \ x . w  ] (\ z . z w)
             = (\z. z w)


Exercise 3. (Reduction) Reduce the following lambda expressions to their normal
form whenever possible.
1     P = (\ x . x (x y)) I  where I = \ u . u

          =  I (I y)
          =  I (y)
          =  y


2     Y = \ f. Q Q   where Q = (\ x . f( x x))

         =  \ f. (\ x . f( x x)) (\ x . f( x x))
         =  \ f. f( (\ x . f( x x)) (\ x . f( x x))))
         =  \ f. f( f( (\ x . f( x x)) (\ x . f( x x)) ))
         =  \ f. f( f( f( (\ x . f( x x)) (\ x . f( x x)) )) ))
         =    ....

3     L = (\ x. x x y)  (\ x. x x y)

         = (\ x. x x y) (\ x. x x y) y
         = ((\ x. x x y) (\ x. x x y) y)  y
         = (((\ x. x x y) (\ x. x x y) y)  y)  y
         = ((((\ x. x x y) (\ x. x x y) y)  y)  y)  y
         =   ....


4     (\ x. x L) M where M = (\ x . y)

          =  (\ x. x L) (\ x . y)
          =  (\ x . y) L
          =  y


Exercise 4. (Equivalence) Consider the lambda expressions in Q 3. Determine
whether the following pairs of expressions are equivalent or not.
1     L  and I
         NO since L is non-terminating but I is

2     P and (\ x . x L) M
         Both simplifies to y

3     \ a . y   and   M
         Yes, by alpha renaming

4     \ a . y  and    \ a . z
          No, since y and z are distinct free vars



Exercise 5. (Church boolean) Implement the following two boolean operators in
pure lambda calculus.
      not  - to negate a boolean value
      or    - find the disjunction of two Boolean values


Tutorial 6
==========

%Exercise1

You can represent a set polymorphically using Set(X)
where X is the type of the elements.

When designing an ADT library, the most important
part is to get the type declarations correct first:

 member :: Set(X), X --> Bool
 union  :: Set(X), Set(X) --> Set(X)
 intersect :: Set(X), Set(X) --> Set(X)

You need some constructors, eg.

 newSet :: () --> Set(X)
```

```
 insert :: X, Set(X) --> Set(X)
 singleton :: X --> Set(X)
```

You also need some destructors. e.g.

```
 chooseElem :: Set(X) --> X  // non-deterministic
 subtract ::  Set(X), X --> Set(X)
```

You may also need more query operations:

```
 size :: Set(X) --> Int
```

Once type specification is designed, you may proceed with implementation. For Set(X), you need to ensure that duplicates are ignored.

%Exercise2

The type below:

```
  isMember :: A -> [A] -> Boolean
```

is too general. It is not possible to support equality test for all type A. We need to restrict the type of A to the Eq type class, as follows (in Haskell):

```
  isMember :: Eq A => A -> [A] -> Boolean
```

The Eq class is typically defined as:

```
  class Eq A
    (==) :: A, A --> Bool
    (!=) :: A, A --> Bool
    a != b = not(a==b)
```

We can define List A to be an instance of Eq if we have Eq A, as follows:

```
  instance Eq A => Eq (List A)
    nil == nil       = true
    (a:as) == nil    = false
    nil  == (b:bs)   = false
    (a:as) == (b:bs) = if a==b then as==bs
                       else false
```

There are many ways to implement Set A as an ordered class. One way is to look at the cardinality, so that a set with more elements is considered bigger. Another way is to look at the elements of Set. Chose the largest element from both set to compare. If they are equal, proceed to the next largest element.

%Exercise3

Double 1 is incorrect.

Double 2 is inefficient as Append is linear complexity to the first argument.

Double 3 is implemented using a higher-order program with accumulating function-type parameter. This is correct but probably a bit inefficient due to the use of higher-order program. However, its complexity remains at O(n)

The best version of tail-recursive code is to use a procedure whereby the 2nd parameter denotes its result. In the recursion, we first build a constructor with head = 2*H but an undefined tail T before making a tail-recursive call. this corresponds to how you may implement a loop-version of the code.

```
declare
proc {Double4 Ls Res}
   case Ls of nil then Res=nil
   [] H|T then local R in
              Res=2*H|R
              {Double4 T R} end
   end
end
```

Thus the use of procedure and output parameter do add some flexibility/effectiveness to Oz programming.

Tutorial 7
===========

%Exercise1

```
local A B C in
   thread if A then B=true else B=false end end
   thread if B then C=false else C=true end end
   A=false
end
```

%Exercise2

```
local X Y Z in
   thread if X==1 then Y=2 else Z=2 end end
   thread if Y==1 then X=1 else Z=2 end end
   X=1
   {Browse X} {Browse Y} {Browse Z}
end

local X Y Z in
   thread if X==1 then Y=2 else Z=2 end end
   thread if Y==1 then X=1 else Z=2 end end
   X=2
   {Browse X} {Browse Y} {Browse Z}
end
```

%Exercise3
%Producer-driven
declare

```
proc {Produce N Xs Limit}
   if Limit>=0 then
      local Xr in
       Xs=N*N|Xr
       {Produce N+1 Xr Limit-1} end
   else Xs=nil
   end
end
fun {Consume Xs Min#Max}
   case Xs of X|Xr then
      {Consume Xr {Value.min Min X}#{Value.max Max X}}
   else Min#Max
   end
end
local Result Xs in
   thread {Produce 0 Xs 100} end
   thread Result={Consume Xs 0#0} end
   {Browse Result}
end

%Consumer-driven
declare
proc {Produce N Xs}
   case Xs of X|Xr then
      X=N*N
      {Produce N+1 Xr}
   else Xs=nil end
end
fun {Consume Xs Min#Max Limit}
%   {Delay 1000}
   if Limit>=0 then
      local X Xr in
       Xs=X|Xr
       {Consume Xr {Value.min Min X}#{Value.max Max X} Limit-1} end
   else Xs=nil Min#Max end
end

local Result Xs in
   thread {Produce 0 Xs} end
   thread Result={Consume Xs 0#0 100} end
   {Browse Result}
end

%Bounded-buffer
declare
proc {Buffer N ?Xs Ys}
   fun {Startup N ?Xs}
      if N==0 then Xs
      else Xr in Xs=_|Xr {Startup N-1 Xr} end
   end
   proc {AskLoop Ys ?Xs ?End}
      case Ys of Y|Yr then Xr End2 in
       Xs=Y|Xr % get element from buffer
       End=_|End2 % replenish the buffer
       {AskLoop Yr Xr End2}
      [] nil then End=nil
      end
   end
   End={Startup N Xs}
in
   {AskLoop Ys Xs End}
end

local Xs Ys Result in
   thread {Produce 0 Xs} end
   thread {Buffer 3 Xs Ys} end
   thread Result={Consume Ys 0#0 10} end
   {Browse Xs} {Browse Ys} {Browse Result}
end

%Exercise4
%Producer-driven
declare
proc {DataDriven  PAcc PState Term CAcc CState CResult}
   local
      proc {Produce Acc Xs}
   if {Term Acc} then
      local Xr NAcc X in
       X#NAcc = {PState Acc}
       Xs=X|Xr
       {Produce NAcc Xr} end
   else Xs=nil {Browse 'Term Producer'}
   end
 end
 fun {Consume Xs Acc} NAcc  in
   case Xs of X|Xr then
      NAcc = {CState X Acc}
      %{Browse X}
      {Consume Xr NAcc}
   else {Browse 'Term Consumer'} {CResult Acc}
   end
 end
 in
 local Result Xs in
   thread {Produce PAcc Xs} end
   thread Result={Consume Xs CAcc} end
   {Delay 1000} {Browse Result}
 end
 end
end
{DataDriven 0 (fun {$ N} N*N#N+1 end) (fun {$ N} N=<100 end)
   0#0 (fun {$ X Min#Max} {Value.min Min X}#{Value.max Max X} end)
 (fun {$ Acc} Acc end)}
```