Lab 8
Part 1: Conceptual

Q1 Laziness and concurrency
This exercise looks closer at the concurrent behavior of lazy execution. Execute the following:

```
fun lazy {MakeX} {Browse x} {Delay 3000} 1 end
fun lazy {MakeY} {Browse y} {Delay 6000} 2 end
fun lazy {MakeZ} {Browse z} {Delay 9000} 3 end
X={MakeX}
Y={MakeY}
Z={MakeZ}
{Browse thread (X+Y) end + Z}
```

This displays x and y immediately, z after 6 seconds, and the result 6 after 15 seconds.
Explain this behavior. What happens if (X+Y)+Z is replaced by X+(Y+Z) or by thread X+Y end + Z?
Which form gives the final result the quickest?
How would you program the addition of n integers i1, ..., in, given that integer ij only
appears after tj milliseconds, so that the final result appears the quickest?

The X and Y are needed at the same time, so they are bound by the max(X,Y) = 6 seconds, and when the value for X+Y is created, Z can is needed, adding another 9 seconds. Switching to X+(Y+Z) will then have max(Y,Z) = 9 added to 3 from the X. Using a thread will allow both (X+Y) and Z to be called at the same time, giving a 9 second execution.

(y+z)+x The idea is that you will have to wait twice, and one of those waits will be 9 seconds because it is larger than the other two, so you should put Y with the Z, so that your other wait will be the shortest (X).

If you tried something like (Z+Z) + (Y+X) you will notice that the left addition is evaluated first, before the right. Thus, you can only ever have two variables needed at a time. That said, you should order your variables in pairs starting in decreasing order from greatest to smallest. -> Tj1 Tj2 … where Tji > Tj(i-1). (((TJ1 + Tj2) + (Tj3 + Tj4)) + (Tj5 + Tj6)) …

Q2 Laziness and monolithic functions.
Consider the following two definitions of lazy list reversal:

```
fun lazy {Reverse1 S}
  fun {Rev S R}
    case S of nil then R
    [] X|S2 then {Rev S2 X|R} end
    end
in {Rev S nil} end

fun lazy {Reverse2 S}
  fun lazy {Rev S R}
    case S of nil then R
    [] X|S2 then {Rev S2 X|R} end
  end
in {Rev S nil} end
```

What is the difference in behavior between {Reverse1 [a b c]} and {Reverse2 [a b c]}?
Do the two definitions calculate the same result? Do they have the same lazy behavior? Explain your answer in each case. Finally, compare the execution efficiency of the two definitions. Which definition would you use in a lazy program? (Generate a very long list e.g. size 10000 and run both reverse functions on the two lists, timing with your phone)

Both Reverse1 and Reverse2 will have the same lazy behavior, meaning that when an element of the list being reversed is needed, the whole list will have to be reversed. This is because reverse is not an productive function, like append for example. In the execution of reverse, only the base case will return a value, whereas the other case is simply another recursive call, from which a single element cannot be extracted.
It is expected that Reverse2 would be less efficient than Reveser1. In both case, when an element is needed, the entire list must be reversed, so the inner function Rev will be executed proportional to the length of the input list. However, in Reverse2, Rev is declared as a lazy function, so every time Rev is called, a ByNeed Trigger is activated.

Q3 Concurrency and exceptions.
Consider the following control abstraction that implements try–finally:

```
proc {TryFinally S1 S2}
  B Y in
    try {S1} B=false catch X then B=true Y=X end
    {S2}
    if B then raise Y end end
end
```

Using the abstract machine semantics as a guide, determine the different possible results of the following program:

```
local U=1 V=2 in
  {TryFinally
   proc {$}
    thread
      {TryFinally proc {$} U=V end
                  proc {$} {Browse bing} end}
    end
   end
   proc {$} {Browse bong} end}
end
```

How many different results are possible?
How many different executions are possible?

Possible outputs are “Bing Bong” or “Bong Bing”

With some substitutions, we have:

```
local U=1 V=2 in
local B Y in
    try {S1}
      B=false catch X then B=true Y=X end %P1.1
    {Browse bong}
    if B then raise Y end end
end
```

with S1 representing its own stack, equivalent to:

```
local B Y in
    try u=v                                 %P2.1
      B=false catch X then B=true Y=X end
    {Browse bing}
    if B then raise Y end end
end
```

```
end
```

The program begins to branch off into several possibilities after the execution of {S1} in the first segment, where at this point we have two stacks with statements ready to be executed. There is a decision between B = false in P1.1 or u = v in P2.1, and if P2.1 is executed first, then P1.1 will not be executed, because an exception will be raised in the try block. Notice that the reason "bong" can be displayed first is that P1.1 can be executed immediately after S1 is threaded, allowing the first code segment to leave the try block, and immediately enter into the browse of "bong".

Part 2: A new way to write streams

Q1 Programmed triggers using higher-order programming. Programmed triggers can be implemented by using higher-order programming instead of concurrency and dataflow variables. The producer passes a zero-argument function F to the consumer. Whenever the consumer needs an element, it calls the function. This returns a pair X#F2 where X is the next stream element and F2 is a function that has the same behavior as F. A key concept for this question is how to return 0 argument functions. For example, the functin that returns the value 3 can be written as F = fun {$} 3 end such that {F} will return the value 3.

(a) write a generator for the numbers 0 1 2 3 ..., where the generator returns a pair V#F, V being the next value in the stream and F being the function that returns the next V1#F1 pair.
example with generator G1... {G1} -> 0#G2 {G2} -> 1#G3 {G3} -> 2#G4a)

```
fun {Generate N}
  fun {$} N#{Generate N+1} end
end
```

(b) write a function that displays the first N values from the stream in part a

```
proc {Display X N}
  fun {Loop X N}
    if N == 0 then nil
    else case {X} of V#F then V|{Loop F N-1} end
    end
  end
in
  {Browse {Loop X N}}
end
```

(c) write a function that takes the stream from a as input, and returns a stream with the numbers multiplied by some number N
e.g. N = 3 ... the stream would be 0 3 6 9 ...

```
fun {Times X N}
  fun {$}
    case {X} of V#F then N*V#{Times F N} end
  end
end
```

(d) write a function that takes a stream as input, and adds the number N to the front of the stream.
e.g. the stream 1 2 3 4 ... with N = 5 would return 5 1 2 3 4 ...

```
fun {Front X N}
  fun {$}
    N#X
  end
end
```

(e) write a function that merges two streams into a single stream, where the output is the zip of the two streams
e.g. S1 = 1 2 3 4 ... S2 = a b c d .. output = 1 a 2 b 3 c ...

```
fun {Zip X Y}
  fun {MergeH X Y B}
    if B == 0 then
      case {X} of V#F then fun {$} V#{MergeH F Y 1} end end
    else
      case {Y} of V#F then fun {$} V#{MergeH X F 0} end end
    end
  end
in
  {MergeH X Y 0}
end

alternative approach

  fun {Zip2 X Y}
      fun {$} case {X} of V#F then V#{Zip2 Y F} end end
  end
```

Q2 Hamming Problem
Convert the solution of the hamming problem for primes 2,3,5 given in the book section 4.5.6 from an implementation using lazy generators, to an implementation using the generators described in part two that produce value function pairs. Note that you will still be needing data flow variables.
Hint -> Merge will take in generators, and return a generator (function that returns a value function pair)
Hint -> H will be a generator, where the first call {H} will return the pair 1#(some function)

```
declare
fun {Times N H}
  fun {$} case {H} of X#H2 then N*X#{Times N H2} end end
end

fun {Merge Xs Ys}
  fun {$} (case {Xs}#{Ys} of (X#Xr)#(Y#Yr) then
    if X < Y then X # {Merge Xr fun {$} Y#Yr end}
    elseif X > Y then Y # {Merge fun {$} X#Xr end Yr}
    else X # {Merge Xr Yr}
    end
          end)
     end
end

H = fun {$} 1 # {Merge {Times 2 H} {Merge {Times 3 H} {Times 5 H}}} end
```

Lab7
Part 1: Conceptual

Q1 Thread semantics.
Consider the following variation of the statement used in Section 4.1.3 to illustrate thread semantics:

```
local B in
  thread          % S1
    B=true        % T1
  end
  thread          % S2
    B=false       % T2
  end
  if B then       % S3
    {Browse yes} % S3.1
  end
end
```

For this exercise, do the following:
(a) Enumerate all possible executions of this statement.
(b) Some of these executions cause the program to terminate abnormally.
Make a small change to the program to avoid these abnormal terminations.

[S1 T1 S2 T2]
[S1 T1 S2 S3 T2]
[S1 T1 S2 S3 S3.1]
[S1 S2 T1 T2]
[S1 S2 T1 S3 T2]
[S1 S2 T1 S3 S3.1]
[S1 S2 T2 T1]
[S1 S2 T2 S3]

Abnormal terminations occur when both T1 and T2 occur in the sequence, before the main program thread terminates.

Q2 Concurrent Fibonacci.
Consider the following sequential definition of the Fibonacci function:

```
fun {Fib X}
  if X=<2 then 1
  else
    {Fib X-1}+{Fib X-2}
  end
end
```

and compare it with the concurrent definition given in Section 4.2.3.
Run both on the Mozart system and compare their performance. How much faster is the sequential definition?
Use the following inputs - 3,5,10,15,20,25,26,27,28
(Give your inputs, run time, and thread count in your experimentation)
Note - Page 255 describes how to use the Oz Panel to view the number of threads created!
How many threads are created by the concurrent call {Fib N} as a function of N?

Thread creation can be modeled with the recurrence relation:
$T(1) = 0$
$T(2) = 0$

$T(n) = 1 + T(n-1) + T(N-2)$

Q3 Order-determining concurrency.
Explain what happens when executing the following:

```
declare A B C D in
thread D=C+1 end
thread C=B+1 end
thread A=1 end
thread B=A+1 end
{Browse D}
```

In what order are the threads created?
In what order are the additions done?
What is the final result?
Compare with the following:

```
declare A B C D in
A=1
B=A+1
C=B+1
D=C+1
{Browse D}
```

Here there is only one thread. In what order are the additions done? What is the final result?
What do you conclude?

The threads are created in sequential order, because the main program can only executes statements from its stack using the Pop() operation, meaning that the first statement on the stack is executed. The additions are done in the exact order as the non-threaded program.

Q4 Thread Efficiency.
Take the nested flatten question from lab 5 (which is a non-iterative function)

```
fun {Flatten Xs}
  proc {FlattenD Xs ?Ds}
    case Xs
    of nil then Y in Ds=Y#Y
    [] X|Xr andthen {IsList X} then Y1 Y2 Y4 in
      Ds=Y1#Y4
      {FlattenD X Y1#Y2}   % ***************** A *********************
      {FlattenD Xr Y2#Y4}
    [] X|Xr then Y1 Y2 in
      Ds=(X|Y1)#Y2
      {FlattenD Xr Y1#Y2}
    end
  end Ys
  in {FlattenD Xs Ys#nil} Ys
end
```

If we replace statement A with
```
 thread {FlattenD X Y1#Y2} end
```
what will happen to the stack size as the program executes?
Would you consider this function iterative?
Do you think threading will make this function more effecient?

In both cases, with and without threading, the same amount of statements will be executed for a given example. Since the original definition of Flatten is non-tail-recursive, a single stack will grow in size as the algorithm recursively moves down the nesting levels of a list. However, for the threaded approach, the original stack size will not grow, because nested lists will be handled in a thread. What this means is that every new nesting level will be thrown out for a newly created stack to handle. For efficiency, it should be more difficult for a system to maintain multiple stacks, then maintain one large stack, because each new stack will incur some overhead for creation and maintenance.

Part 2: Streams

Q1 Producers, Filters, and Consumers

```
fun {Generate N Limit}
  if N<Limit then
    N|{Generate N+1 Limit}
  else nil
  end
end
```

(a) Using the above generator on a list from [0 1 2 ... 100] and threading, write functions that
filter out all odd numbers
filter out all multiples of 4
filter out all numbers greater than 7 and less than 77

```
fun {FOdd N}
  if N == nil then nil
  elseif (N.1 mod 2) == 0 then
      N.1|{FOdd N.2}
  else
    {FOdd N.2}
  end
end

fun {FMul4 N}
  if N == nil then nil
  elseif (N.1 mod 4) \= 0 then
      N.1|{FMul4 N.2}
  else
    {FMul4 N.2}
  end
end

fun {F3 N}
  if N == nil then nil
  elseif {And (N.1 > 7) (N.1<77)} then
      {F3 N.2}
  else
    N.1|{F3 N.2}
  end
end
```

(a') Place the generator, three filters, and reader all in separate threads (where the reader simply displays elements one at a time)
The stream will look like the following:
[Generator]->[remove odds]->[remove multiples of 4]->[remove numbers (7...77)]->[Display element]

```
fun {Generate N Limit}
  if N<Limit then
```

```
    {Delay 100}
    N|{Generate N+1 Limit}
  else nil
  end
end
```

Use the above generator so there is a pause between elements being created
Describe the flow of the first 9 elements as they move through the chain. What threads are awakened, and when?

```
declare Xs Ys Zs Os
thread Xs = {Generate 0 100} end  %T1
thread Ys = {FOdd Xs} end         %T2
thread Zs = {FMul4 Ys} end        %T3
thread Os = {F3 Zs} end           %T4
```

Upon creation, T1 is continuously awake, producing values in its stream until it reaches 100. T2 is awake whenever a value from T1 has been added to Xs, which is every $1/10^{th}$ of a second due to the Delay. T3 will only wake up if a value moves from T2 and is added to Ys, and likewise T4 will only awaken if a value is added to Zs from within T3. Values move through the filters sequentially, where 0 passes through to Os, but 1 will be filtered within T2 (along with every other number). The remaining numbers will be handled similarly.

(b) Using the above generator on a list from [0 1 2 ... 100] and threading, write consumers that
return the list of sums of every pair of consecutive integers, i.e. [0+1 2+3 4+5 ...] = [1 5 9 ...]
return the sum of all odd numbers (you will need a filter and fold operation)

```
fun {SumP N}
  case N of nil then nil
  [] [X] then X
  [] X|Y|T then (X+Y)|{SumP T}
  end
end

fun {Feven N}
  if N == nil then nil
  elseif (N.1 mod 2) == 1 then
      N.1|{Feven N.2}
  else
    {Feven N.2}
  end
end

fun {Con N}
  case N of nil then 0
  [] H|T then if  H mod 2 == 1 then H+{Con T} else {Con T} end
  end
end
```

Q2 Prime number filter
Using the above generator on a list from [0 1 2 ... 1000] filter out all prime numbers
The filter works as follows:
Maintain a list of primes, with the inital singleton list [2]
At each value n, check if n is divisible by any of the primes in your list from [2 .. m] (where $m^2 < n$)
- if n is divisible by at least one of these primes, keep it in the stream
- otherwise, n is prime, so it will be added to the list of primes, and removed from the stream

```
{IsDiv X Ps}
  case Ps of nil then false
```

```
  []  P|Ps then if P*P > X false
    elseif X mod P == 0 then true
    else {IsDiv X Ps}
    end
  end
end

{PFil N Ps}
  case N of nil then nil
  [] H|T then
    if H == 0 then H|{PFil T Ps}
    elseif H == 1 then H|{PFil T Ps}
    elseif H == 2 then H|{PFil T Ps}
    elseif {IsDiv H Ps} then H|{PFil T Ps}
    else {PFil T {Append Ps [H]}}
    end
  end
end
```

called {Pfil N [2]} where the primes list is initialized with the first prime 2 and there are base cases for 0,1,2

Q3 Digital logic simulation.
In this exercise we will design a circuit to add n- bit numbers and simulate it using the technique of Section 4.3.5. Given two n-bit binary numbers, (xn−1...x0)2 and (yn−1...y0)2. We will build a circuit to add these numbers by using a chain of full adders, similar to doing long addition by hand. The idea is to add each pair of bits separately, passing the carry to the next pair. We start with the low-order bits x0 and y0. Feed them to a full adder with the third input z = 0. This gives a sum bit s0 and a carry c0. Now feed x1, y1, and c0 to a second full adder. This gives a new sum s1 and carry c1. Continue this for all n bits. The final sum is (sn−1...s0)2. For this exercise, program the addition circuit using full adders. Verify that it works correctly by feeding it several additions.
GateMaker, FullAdder, and all the accompanying binary operatins can be found on pages 273,274 as well as a diagram of the full adder.

Code remains unchanged from the book, except for a base case in GateMaker

```
-- case Xs#Ys of nil#nil then nil
```

```
declare Z Zr S
Z = 0|Zr
{FullAdder {Reverse [1 0 0]} {Reverse [1 1 0]} Z Zr S}
{Browse {Reverse S}}
```

What is happening here is that the third argument for full adder, Z is the carry values for an addition of two numbers, which starts at 0. These carry values are calculated at each step of the addition, and stored in the fourth argument of FullAdder, for which we pass the tail of Z, namely Zr. This then computes carry values, and stores them in the tail of Z, so that they can be used in the next step of the addition. The first and second arguments are reversed because we are simulating addition from right to left. So FullAdder performs the left to right addition, and we reverse the Sum before browsing it.

Lan6
Part 1: Control Flow

For each question, come up with three operations, and test these operations on lists, displaying the input and output in a comment.

Q1 Binary Fold
The function BFold L F takes a list of integers L and a binary operation on integers F, and returns the binary fold of the F applied to L, where the binary fold is defined as follows:
• BFold where L contains a single element returns that element
• BFold where L contains two or more elements returns BFold of Bmap L F
  • Bmap applies F to successive pairs of a list as follows:
    • Bmap of a list with two or more elements, e.g. X|Y|Ls returns {F X Y} | {Bmap Ls F}
    • Bmap of a list with a single or no element, returns the list, i.e. Bmap [X] F returns [X] and Bmap nil F returns nil

```
fun {BMap L F}
  case L of nil then nil
  [] [X] then [X]
  [] X|Y|T then {F X Y}|{BMap T F}
  end
end

fun {BFold L F}
  case L of nil then 0
  [] [X] then X
  else {BFold {BMap L F} F}
  end
end
```

Q2 Nested Fold
The function NFoldL L FZs takes a nested list L and a list of binary operators, value pairs. If FZs is ordered as [F1#ZF1 F2#ZF2 F3#ZF3 ...], you will use the function Fi at the nested depth i, performing the right associative fold operation, with the second value of each pair being the initial value of the folds.
e.g.)
{ NFold [1 2 [2 3] [1 [2 3]]] [F#ZF G#ZG H#ZH] }
F 1 (F 2 (F (G 2 (G 3 ZG)) (F (G 1 (G (H 2 (H 3 ZH)) ZG)) ZF)))

You will raise an error if the nesting depth d is greater than the length of FZs (i.e. There are not enough functions in FZs to match each level of nesting in L)

```
fun {NFold L Fs}
  if Fs == nil then raise illegalFunctionsLength end end
  case L
  of nil then Fs.1.2
  [] X|Xr andthen {IsList X} then
    {Fs.1.1 {NFold X Fs.2} {NFold Xr Fs }}
  [] X|Xr then
    {Fs.1.1 X {NFold Xr Fs }}
  end
end
```

Q3 Scan

The function ScanL L Z F takes a list L, Initial value Z, and a binary function F. This will return a list with successive left folds. With L = [X1 X2 X3 X4 ...] we will get the list
[Z, F Z X1, F (F Z X1) X2, ...] where the last element of the output is exactly the FoldL of L Z F.

```
fun {ScanL L Z F}
  case L of nil then Z|nil
  [] X|Xr then Z|{ScanL L {F Z X} F}
  end
end
```

P2 – Secure Dictionary
Secure Dictionary
Implement the list-based declarative dictionary as an ADT, as in Figure 3.27 on p. 199, but in a secure way, using wrap and unwrap, as outlined in Section 3.7.6 (Page 210). Each dictionary will come with two extra features, a binary function F on integers and an integer Z. Your dictionary will have integers as keys (aka features) and pairs of integer lists and atoms as values. The key for each list-atom pair will be calculated from the list by performing a left-associative fold on the list using F and Z. As a result, the Put function will not take a Key as argument but calculate it from the Value. Make sure the code for Put is updated appropriately.

After creating your dictionary, run several Put, CondGet, and Domain examples, displaying the inputs and outputs in a comment. Answer the following questions:
a) What happens when two distinct lists have the same Key value after the folding operation, based on the definition of Put from the book? Give an example.
b) Describe the NewWrapper function on page 207. How does the wrapper/unwrapper created by this function secure the dictionary?
c) Are the F and Z values associated with the dictionary secure? If not, how could you make these secure as well?

```
declare
proc{NewWrapper ?Wrap ?Unwrap}
   Key = {NewName}
in
   fun{Wrap X}
      fun{$ K} if K==Key then X end
      end
   end
   fun{Unwrap W}
      {W Key}
   end
end
fun{Fold Ls F Z}
   case Ls
   of nil then Z
   [] L|Ls then {Fold Ls F {F Z L}}
   end
end
fun {Map Ls F}
   case Ls
   of nil then nil
   [] H|T then {F H} | {Map T F}
   end
end
local Wrap Unwrap Dictest Dictest1
   {NewWrapper Wrap Unwrap}
   fun{NewDictionary F Z}
      nil#F#Z
   end
```

```
   fun{Push Ds#F#Z Value#Atom}
      local
      Key = {Fold Value F Z}
      NewDs = {PushHelper Ds Key Value#Atom} in
          NewDs#F#Z end
   end
   fun{PushHelper Ds Key Value}
      case Ds
      of nil then [Key#Value]
      [](K#V)|Dr andthen Key==K then
         (Key#Value)|Dr
      [](K#V)|Dr andthen K>Key then
         (Key#Value)|(K#V)|Dr
      [](K#V)|Dr andthen K<Key then
         (K#V)|{PushHelper Dr Key Value}
      end
   end
   fun{CondPop Ds#F#Z Value Default}
      local
      Key = {Fold Value F Z}
      in {CondPopHelper Ds Key Default} end
   end
   fun{CondPopHelper Ds Key Default}
      case Ds
      of nil then Default
      [](K#V)|Dr andthen Key==K then
         V
      [](K#V)|Dr andthen K>Key then
         Default
      [](K#V)|Dr andthen K<Key then
         {CondPopHelper Dr Key Default}
      end
   end
   fun{Domain Ds}
      {Map Ds.1 fun{$ K#_} K end}
   end
in
   fun{NewDictionary2 F Z}
      {Wrap {NewDictionary F Z}}
   end
   fun{Push2 Ds Value}
      {Wrap {Push {Unwrap Ds} Value}}
   end
   fun{CondPop2 Ds K Default}
      {Wrap {CondPop {Unwrap Ds} K Default}}
   end
   fun{Domain2 Ds}
      {Domain {Unwrap Ds}}
   end
Dictest = {NewDictionary2 fun {$ X Y} X+Y end 0}
Dictest1 = {Push2 Dictest [1 2 3]#a}
   {Browse {Domain2 Dictest1}}
   {Browse {Unwrap Dictest1}}
end
```

Part 3: Declarative Concurrency

Given the following program code:

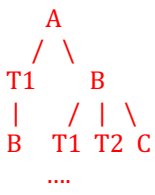
```
local A B C in
  thread    %%%%%%%%%%%%%%%%%%%%%%%%% A
    A = 5   %%%%%%%%%%%%%%%%%%%%%%%%% T1
  end
  thread    %%%%%%%%%%%%%%%%%%%%%%%%% B
    B = 7   %%%%%%%%%%%%%%%%%%%%%%%%% T2
  end
  thread    %%%%%%%%%%%%%%%%%%%%%%%%% C
    C = 3   %%%%%%%%%%%%%%%%%%%%%%%%% T3
  end
  if C > A then  %%%%%%%%%%%%%%%%%%%% S1
    {Browse "C is greater than A"} %% S2
  else
    if B > A then  %%%%%%%%%%%%%%%%%% S3
      {Browse "B is greater than A"}% S4
    end
  end
end
```

What are all the possible interleavings of the statements A, B, C, T1, T2, T3, S1..S4? How about when A = 2?

["A","B","C","T1","T2","T3","S1","S3","S4"],
["A","T1","B","C","T2","T3","S1","S3","S4"],
["A","B","T1","C","T2","T3","S1","S3","S4"],
["A","B","T2","C","T1","T3","S1","S3","S4"],
["A","B","C","T2","T1","T3","S1","S3","S4"],
["A","T1","B","T2","C","T3","S1","S3","S4"],
["A","B","T2","T1","C","T3","S1","S3","S4"],
["A","B","T1","T2","C","T3","S1","S3","S4"],
["A","B","C","T3","T1","T2","S1","S3","S4"],
["A","B","C","T1","T3","T2","S1","S3","S4"],
["A","T1","B","C","T3","T2","S1","S3","S4"],
["A","B","T1","C","T3","T2","S1","S3","S4"],
["A","B","T2","C","T3","T1","S1","S3","S4"],
["A","B","C","T3","T2","T1","S1","S3","S4"],
["A","B","C","T2","T3","T1","S1","S3","S4"],
["A","B","C","T3","T1","S1","T2","S3","S4"],
["A","B","C","T1","T3","S1","T2","S3","S4"],
["A","B","T1","C","T3","S1","T2","S3","S4"],
["A","T1","B","C","T3","S1","T2","S3","S4"]

One was to approach this problem is by using a tree, where branching is equivalent to making a choice between multiple threads. Once the tree has been completed the paths starting from the root an ending at a leaf will be one execution sequence.

e.g.



Lab5
Part 1 (Invariants and Proofs)
For this section, you will state the invariant, and write an inductive proof (base case and inductive case similar to the Fact example from class)

(a)

```
local
  fun {IterLength I Ys}
    case Ys
    of nil then I
    [] _|Yr then {IterLength I+1 Yr} end
    end
  in
    fun {Length Xs}
      {IterLength 0 Xs}
    end
  end
```

Invariant → for all lists of length n, Solution = I + Length(Ys)
Base Case: Ys = nil: Solution = 0 + Length(nil) = 0
Inductive Case: assume the invariant holds for recursive calls, and take Ys to be your list at some arbitrary length in the execution
Then, the invariant holds for Solution = I+1 + Length(Yr). (Recursive call)
Since Yr is the tail of Ys, Length(Ys) = Length(Yr)+1, so by substitution we have
Solution = I + Length(Ys)

Now, Length(Xs) = IterLength 0 Xs,, = 0 + Length(Xs) = Length(Xs)

% (b)

```
local
  fun {IterReverse Rs Ys}
    case Ys
    of nil then Rs
    [] Y|Yr then {IterReverse Y|Rs Yr} end
    end
  in
    fun {Reverse Xs}
      {IterReverse nil Xs}
    end
  end
```

Invariant → for all lists of length n, Solution = Reverse(Ys) ++ Rs
Base Case: Ys = nil: Solution = Reverse(nil) ++ nil = nil
Inductive Case: assume the invariant holds for recursive calls, and take Ys to be your list at some arbitrary length in the execution.
Then, the invariant holds for Solution = Reverse(Yr) ++ Y|Rs (Recursive call)
Since Yr is the tail of Ys, Reverse(Yr) ++ Y = , Reverse(Ys)
We have Solution = Reverse(Yr) ++ Y|Rs = Reverse(Yr) ++ Y ++ Rs = Reverse(Ys) ++ Rs by substitution.

Now Reverse(Xs) =IterReverse nil Xs = Reverse(Xs) ++ nil = Reverse(Xs)

(c) - write an iterative version of SumList, then find the invariant and create an inductive proof
******* Non-iterative version of SumList ***********

```
fun {SumList L}
  case L
  of nil then 0
  [] X|L1 then X+{SumList L1}
  end
end
```

******* Non-iterative version of SumList ***********

```
fun {Sum L}
      fun {SumList L S}
        case L
        of nil then S
        [] X|L1 then {SumList L1 X+S}
        end
      end
      in
      {SumList L 0}
end
```

Invariant → for all lists of length n, Solution = S + Sum(L)
Base Case: L = nil: Solution = 0 + Sum(nil) = 0
Inductive Case: assume the invariant holds for recursive calls, and take L to be your list at some arbitrary length in the execution.
Then, the invariant holds for Solution = S+X + Sum(L1) (Recursive call)
Since X|L1 is exactly the list L, Sum(L) = Sum(L1)+X, so by substitution we have
Solution = S + Sum(L)

Now, Sum(L) = SumList 0 L = 0 + Sum(L)= Sum(L)

(d)

```
fun {Merge Xs Ys}
  case Xs # Ys
  of nil # Ys then Ys
  [] Xs # nil then Xs
  [] (X|Xr) # (Y|Yr) then
     if X<Y then X|{Merge Xr Ys}
     else Y|{Merge Xs Yr}
     end
  end
end
```

Invariant → For all natural numbers n, if length(Xs) + length(Ys) = n and both Xs and Ys are sorted, then {Merge Xs Ys} is a sorted list of length n.

Base Case: Ys = nil: Out = Xs, orederd by assumption and size of Xs
Base Case Xs = nil: Out = Ys, ordered by assumption and size of Ys
Inductive Case: assume the invariant holds for recursive calls, and take Ys and Xs to be your list at some arbitrary point in the execution. With length (Ys) = n length(Xs) = m
Then, the invariant holds for
Case1: {Merge Xr Ys} is the sorted list of length m
+n-1. X is less than Y, and therefore X < every element in Yr since Ys is a sorted list, X < every element in Xr because X is a sorted list, therefore X|{Merge Xr Ys} is a sorted list of length n+m
case 2: similar to case 1

```
fun {MergeSort Xs}
  fun {MergeSortAcc L1 N}
```

```
    if N==0 then
      nil # L1
    elseif N==1 then
      [L1.1] # L1.2
    elseif N>1 then
      NL=N div 2
      NR=N-NL
      Ys # L2 = {MergeSortAcc L1 NL}
      Zs # L3 = {MergeSortAcc L2 NR}
    in
      {Merge Ys Zs} # L3
    end
  end
in
  {MergeSortAcc Xs {Length Xs}}.1
end
```

Invariant → For all natural numbers n, if {MergeSortAcc L n} == L1 # L2, then L1 is the sorted list of n elements, and L2 is the remaining unsorted elements.

Base Case: n = 1, then [L1.1]#L1.2 on a list of size 1 will return the sorted list

Inductive Case: assume the invariant is true for all values smaller than n (strong induction). Then {MergeSortAcc L1 NL) returns Ys sorted for value NL with L2 being the rest of L1, and {MergeSortAcc L2 NR} returns Zs sorted, which is the first NR elements of L2, with L3 unsorted. So we have NL elements (Ys) sorted and NR elements (Zs) sorted, corresponding to the first NL + NR = N elements of L1 sorted.

Part 2

1. Section 3.4.2 defines the Append function by doing recursion on the first argument.
What happens if we try to do recursion on the second argument? Here is a possible solution:

```
fun {Append Ls Ms}
  case Ms
  of nil then Ls
  [] X|Mr then {Append {Append Ls [X]} Mr}
  end
end
```

Is this program correct? Does it terminate? Why or why not?
This program does not terminate because you will reach the case

{Append _ [X]}
which will make the recursive call {Append {Append _ [X]} nil} hence causing an infinite loop.

2. This exercise explores the expressive power of dataflow variables.
In the declarative model, the following definition of append is iterative:

```
fun {Append Xs Ys}
  case Xs
  of nil then Ys
  [] X|Xr then X|{Append Xr Ys}
  end
end
```

We can see this by looking at the expansion:

```
proc {Append Xs Ys ?Zs}
  case Xs
  of nil then Zs=Ys
  [] X|Xr then Zr in
    Zs=X|Zr
    {Append Xr Ys Zr}
  end
end
```

This can do a last call optimization because the unbound variable Zr can be put in the list Zs and bound later.
Now let us restrict the computation model to calculate with values only. How can we write an iterative append? One approach is to define two functions: (1) an iterative list reversal and (2) an iterative function that appends the reverse of a list to another. Write an iterative append using this approach.
Note - by values only, we mean that every identifier must be bound to a value upon declaration

```
fun {Append Xs Ys}
  fun {AppendH Xs Ys}
    case Xs of nil then Ys
    [] X|Xt then {Append Xt X|Ys}
    end
  end
in
  {AppendH {Reverse Xs} Ys}
end
```

3. Calculate the number of operations needed by the two versions of the Flatten function given in Section 3.4.4. With n elements and maximal nesting depth k, what is the worst-case complexity of each version?
Note - Assume IsList uses constant time to check if an input is a list and the append in the first function works in O(n) time

First, run the functions on the example [[1 2 3] [1 2] [1 2 [2 3 4]] 3 4] and give the exact number of operations for execution.

```
fun {Flatten Xs}
  case Xs
  of nil then nil
  [] X|Xr andthen {IsList X} then
    {Append {Flatten X} {Flatten Xr}}
  [] X|Xr then
    X|{Flatten Xr}
  end
end
```

```
fun {Flatten Xs}
  proc {FlattenD Xs ?Ds}
    case Xs
    of nil then Y in Ds=Y#Y
    [] X|Xr andthen {IsList X} then Y1 Y2 Y4 in
      Ds=Y1#Y4
      {FlattenD X Y1#Y2}
      {FlattenD Xr Y2#Y4}
    [] X|Xr then Y1 Y2 in
      Ds=(X|Y1)#Y2 {FlattenD Xr Y1#y2}
    end
  end Ys
```

```
  in {FlattenD Xs Ys#nil} Ys
end
```

In the first version we have the list creation in the third case X|..., and also in the second case on the call to Append, which has list creations equal to the size of X. The number of operations for the example will be 25 when looking at these two cases, with the former case giving 1 for each element in the list. The only thing to be careful of is the order of appends. For the second case, we only have one list creation at X|Y1 in the third case, so we get one list creation for each element in the list, which is 12. Worst case complexity of the first would occur in a case with single elements at the nesting depth k, e.g. [[[[1]]] [[[2]]] [[[3]]]] for n = 3 k = 3; where we have k operations for each individual nested list, then n appends, to get n + n*k.

4. The following is a possible algorithm for sorting lists. Its inventor, C.A.R. Hoare, called it quicksort, because it was the fastest known general-purpose sorting algorithm at the time it was invented. It uses a divide and conquer strategy to give an average time complexity of O(n log n). Here is an informal description of the algorithm for the declarative model. Given an input list L.
Then do the following operations:

(a) Pick L's first element, X, to use as a pivot.
(b) Partition L into two lists, L1 and L2, such that all elements in L1 are less than X and all elements in L2 are greater or equal than X.
(c) Use quicksort to sort L1 giving S1 and to sort L2 giving S2.
(d) Append the lists S1 and S2 to get the answer.

Write this program with difference lists to avoid the linear cost of append.

```
declare
fun {Pivot N L L1 L2}
   case L of nil then L1#L2
    [] H|T then
    if H < N then {Pivot N T H|L1 L2}
    else {Pivot N T L1 H|L2}
    end
  end
end

fun {QuickSort L}
  case L of nil the nil
  [ ]  [X] then [X]
  [ ]  H|T then local L1 L2 L1S L2S in
     L1#L2 = {Pivot H T nil nil}
     L1S = {QuickSort L1}
    L2S = {QuickSort L2}
    {Append L1S {Append H L2S}}
    end
  end
end

fun {QuickSort L}
fun {QuickSortD L Dif}
  case L of nil then Dif#Dif
  []  [X] then  (X|Dif)#Dif
  []  H|T then local L1 L2 L1S L2S D1 D2 in
                  L1#L2 = {Pivot H T nil nil}
                  if L1 == nil then L2S#_ = {QuickSortD L2 Dif} (H|L2S)#Dif
                  else L1S#_ = {QuickSortD L1 D1}
                     L2S#_ = {QuickSortD L2 Dif}
                     D1 = H|L2S
                     L1S#Dif
                  end
```

```
                end
  end
end
Ans
Dif
in
   Ans#_= {QuickSortD L nil}
   Ans
end
```

Lab4

1. Section 2.4 explains how a procedure call is executed. Consider the following procedure MulByN:

```
declare MulByN N in
N=3
proc {MulByN X ?Y}
    Y=N*X
end
```

together with the call {MulByN A B}. Assume that the environment at the call contains {A → 10, B → x1}. When the procedure body is executed, the mapping N → 3 is added to the environment. Why is this a necessary step? In particular, would not N → 3 already exist somewhere in the environment at the call? Would not this be enough to ensure that the identifier N already maps to 3? Give an example where N does not exist in the environment at the call. Then give a second example where N does exist there, but is bound to a different value than 3.

Example where N has left the environment

```
local MulByN in
  local N in
    N = 3
    proc {MulByN X ?Y}
      Y=N*X
    end
  end
  {MulByN A B}
end
```

Example where N is pointing to a different variable in the environment.

```
local MulByN  N in
  N = 3
  proc {MulByN X ?Y}
    Y=N*
  end
  local N in
    N = 44
    {MulByN A B}
  end
end
```

In both of these cases, the value of N when the procedure MulByN is called is 3 because N was captured in the Contextual Environment of MulByN upon its declaration. So it does not matter what happens to N in the environment after the creation of MulByN.

2. This exercise examines the importance of tail recursion, in the light of the semantics given in the chapter. Consider the following two functions:

```
fun {Sum1 N}
  if N==0 then 0 else N+{Sum1 N-1} end
end

fun {Sum2 N S}
  if N==0 then S else {Sum2 N-1 N+S} end
end
```

(a) Expand the two definitions into kernel syntax. It should be clear that Sum2 is tail recursive and Sum1 is not.

```
declare Sum1 three out  One Z
Sum1 = proc {$ N O}                  %1
```

```
  local C in                          %2
    {Equiv N Z C}                     %3
    if C then                         %4
      O = 0                           %5
    else
      local N1 in                     %6
        local O1 in                   %7
          {Sub N One N1}              %8
          {Sum1 N1 O1}                %9
          {Add O1 N O}                %10
          end
        end
      end
    end
  end
end

three = 3                             %a
One = 1                               %b
Z = 0                                 %c
{Sum1 Three Out}                      %d


declare Sum2 S Out One Z
Sum2 = proc {$ N S O}
  local C in
    {Equiv N Z C}
    if C then O = S
    else
      local N1 in
        local O1 in
          {Sub N One N1}
          {Ass N S O1}
          {Sum2 N1 O1 O}
          end
        end
      end
    end
  end
end
One = 1
Zero = 0
Three = 3
{Sum2 Three S Out}
```

(b) Execute the two calls {Sum1 3} and {Sum2 3 0} by hand, using the semantics of this chapter to follow what happens to the stack and the store. Specifically, for the first iteration through the procedure definition, show the affect of each statement on the stack, environment, and store similar to Dr. Wilson's Piazza post @85. Iteration two and three will be similar so only show the environment, store, and stack right before the recursive call. Then, for iteration 4 (Base Case)go through each statement, and finish popping statements off of the stack from the previous procedure calls.

σ = {{sum1},{three},{out},{one},{z}} and E = {Sum1->sum1,Three->three, Out->out, One->one, Z->z} after the declare statement before any code is executed.

After unfolding the statement sequencing, we will have a stack of [(1,E),(a,E),(b,E),(c,E),(d,E)] *where all other statements are nested inside the declaration of Sum1, to be added to the stack when Sum1 is called in statement d*

1 Value creation of a procedure, with closure for One and Z We update the store with a new variable sum1' pointing to the value of the procedure, then we bind {sum1} with {sum1'} to get {sum1=sum1'=proc{$ N O} 2 end. {One->one, Z->z}}, with (1,E) popped from the stack.

a,,b,c Value creations like 1. Update the store with {{three = three' = 3}, {one = one' = 1},{z=z'=0}, popping (a,E) from the stack

d Procedure call where E(Sum1) is a procedure value expecting two arguments. We create E', which is the closure of sum1 added with the mappings of the input variables E(Three) and E(Out)
E' = {N->three, O->out} + {One->one, Z->z}
Then (2,E') replaces (d,E) on the stack

2 is a local so we update the Environment E' with a mapping and add the variable to its own equivalence set in the store
E'' = E' ++ {C->c}, add {c} to the current store
(2,E') is popped from the stack and we fill it with the (de-sequenced) statements (3,E''),(4,E'')

3 this is a procedure call, and we look up E''(N)=3, E''(Z)=0, E''(C)={c} and assign the value false to E''(C) {c=false}

4 this is an if statement, so we look up E''(C)=false, so we put statement (6,E'') onto the stack.

6,7 are local statements, and we will end up with two new Environment mappings and Two new equivalence sets in the store
E''' = E'' ++ {N1 -> n1, O1 ->o1}
Pop 6,7 off the stack, and de-sequence their interior statement to get (8,E'''),(9,E'''),(10,E''')

8 this is a procedure call, and we look up E'''(N)=3 E'''(One)=1 and bind E'''(N1) with the value 2, {n1=2}
Pop 8 from the stack.

9 Procedure call where E'''(Sum1) is a procedure value expecting two arguments. We create E'''', which is the closure of sum1 added with the mappings of the input variables E'''(N1) and E'''(O1)
E'''' = {N->n1, O->O1} ++ {One->one, Z->z},
Then (2,E'''') replaces 9 on the stack

E'''' = { N->n1, O->o1, One->one, Z->z }
E''' = { N->three, O->out, C->c , N1 -> n1, O1 ->o1, One->one, Z->z}
σ = {{ sum1=sum1'=proc{$ N O} 2 end. {One->one, Z->z}},{three=three'=3},{one=one'=1},{z=z'=0}, {c=false},{n1=2},{o1},{out}}
Stack = {(2,E''''),(10,E''')}

The execution from 2 through two recursive calls will be the same as above, with new variable namings.
After the first recursive call:

E6= { N->n1', O->o1', One->one, Z->z }
E5 = {{ N->n1, O->o1, One->one, Z->z , N1->n1', C->c', O1->o1'}
σ' = σ ++ {{ o1'},{c' = false}, {n1' = 1}}
Stack = {(2,E6),(10,E5),(10,E''')}

After the second:
E8= { N->n1'', O->o1'', One->one, Z->z }

E7 = {{ N->n1', O->o1', One->one, Z->z , N1->n1'', C->c'', O1->o1''}
σ'' = σ' ++ {{ o1''},{c'' = false}, {n1'' = 0}}
Stack = {(2,E8),(10,E7),(10,E5),(10,E''')}

2 is a local so we update the Environment E8 with a mapping and add the variable to its own equivalence set in the store
E9 = E8 ++ {C->c'''}, add {c'''} to the current store
2 is popped from the stack and we fill it with the (de-sequenced) statements (3,E9),(4,E9)

3 this is a procedure call, and we look up E9(N)=0, E9(Z)=0, E9(C)={c'''} and assign the value false to E9(C) {c'''=true}

4 this is an if statement, so we look up E9(C)=true, so we put statement (5,E9) onto the stack.

5 assignment statement, bind E(O) with 0, so {o1'' = 0} in the store.

Stack is now {(10,E7),(10,E5),(10,E''')}

10 Procedure call, Look up E7(O1)=o1''=0 E7(N) = n1' = 1 and assign to E7(O)=o1'=1

10 Procedure call, Look up E5(O1)=o1'=1 E5(N) = n1 = 2 and assign to E5(O)=o1=3

10 Procedure call, Look up E'''(O1)=o1=3 E'''(N) = three = 3 and assign to E'''(O)=out=6

Sum2 is handled similarly

Q3 Include the Records divide(X Y), list(L) which returns the list L, and append(H T) which takes an integer and appends it to a list such that the function Eval will return either an integer, a list, or an error. Change the catch into a pattern matching catch (Page 96) with the following exceptions

illFormedExpr(E) -- same as the already existsing error

illFormedList(E) -- if list(L) is evaluated and L is not a list (using a helper function IsList that you define)
IsList checks if the head of the input is an integer, then recursively checks the rest of the list. Base case is nil which returns true.

illFormedAppend(E) -- if append(H T) is passed to Eval and H is not an integer (using the IsNumber function)

Include another exception for dividing by 0, such that the exception will then execute the division, by changing the denominator to 1, and output the result to the browser. This exception will not be in the pattern matching catch described above, but will be on the outside (You will need a nested try, catch statement to achieve this)

```
fun {IsList L}
  case L of nil then true
  [] H|T then if {IsNumber H} then {IsList T} else raise illFormedList(L) end
end
  end
end

fun {Eval E}
  if {IsNumber E} then E
    else
    case E
    of plus(X Y) then {Eval X}+{Eval Y}
    [] times(X Y) then {Eval X}*{Eval Y}
    [] divide(X Y) then if {Eval Y} == 0 then raise divByZero(E) end
                        else {Eval X}/{EvalY}
```

```
                        end
    [] append(H L) then if {IsNumber H} then H|{Eval L} else raise
illFormedAppend(E) end
    [] list(L) then if {IsList L} then L end
    else raise illFormedExpr(E) end
    end
  end
end
try
  try
    {Browse {Eval plus(plus(5 5) 10)}}
    {Browse {Eval times(6 11)}}
    {Browse {Eval minus(7 10)}}
  catch illFormedExpr(E) then
    {Browse ´*** Illegal expression ´#E#´ ***´}
    [] illFormedList(E) then
    {Browse ´*** Illegal List ´#E#´ ***´}
    [] illFormedAppend(E) then
    {Browse ´*** Illegal Append ´#E#´ ***´}
    [] X then raise X end
  end
catch divByZero(divide(X _))
  {Browse X}
end
```

Describe the process, in terms of the stack, from the moment a division by 0 exception is raised, to the moment the division division is executed with a new denominator. (Ignore Environment and Store)

When an exception is raised, statements are popped from the stack until a catch is reached. In the case of a division by zero error, statements will be popped from the stack until the inner catch is executed. The division by zero will match the else case of the catch, thus re-raising the exception, (raising the exact exception to a outer catch statement). Once again the stack will have statements popped until the outer catch is reached, this just so happened to be the very next statement sitting on the stack after the second exception has been raised.

4. Based on the unification algorithm on page 103, describe the unification process for the following example Describe the Stack, Environment, and Store as each statement is executed, similar to Q2(b), and show the output store
Remark: Describe each step in the unification when it occurs, using the syntax unify(X,Y), bind(ESx,ESy), etc. as shown on page 103

```
declare X Y A B C D E F G H I J K L M N
L = D
M = D
N = F
A = birthday(day:3 month:C year:1986)
B = birthday(day:D month:D year:F)
I = J
J = 19
K = D
X = person(age:I name:"Stan" birthday:A)
Y = person(age:G name:H birthday:B)
X=Y
```

Environment Mapping {X->x, Y->y … }
Store {{x}, {y}, {a}, …}
Before the final statement X = Y, the Store will be

{{x = x' = person(age:i name:s birthday :a)}, {y = y' = person(age:g name:h birthday:b)},{a = a' = birthday(day:t month:c year:nine)},{b = b' = birthday(day:d month:d year:f)},{c},{d,l,m,k},{e},{f,n},{g},{h},{i=j=19},{t = 3},{nine = 1986},{s = "stan"}}

Now for Unify(x,y)

- Bind(i,Es(g)) → {i=j=g=19}
- Bind(s,Es(h)) → {h = s = "stan"}
- Unify(a,b)
    - Bind(t,Es(d)) → {d=l=m=k=3}
    - Bind(Es(c),d) → {c=d=l=m=k=3}
    - Bind(nine,Es(f)) → {f=n=nine=1986}

Lab3

1. If a function body has an 'if' statement with a missing 'else' clause, then an exception raised if the 'if' condition is false. Explain why this behavior is correct. This situation does not occur for procedures. Explain why not.

A function must return a single value, whereas a procedure does not return any value, unless using the $. If a function has a missing else clause, then there is a chance, if the else is entered, that the function does not return any value, which breaks the rule that functions return values. This behavior is okay for procedures, because they have no return.

2. Using the following:
(1) - if X then S1 else S2 end
(2) - case X of Lab(F1: X1 ... Fn: Xn) then S1 else S2 end
(a) Define (1) in terms of the 'case' statement.

```
case X of true then S1 else S2 end
```

(b) Define (2) in terms of the 'if' statement, using the operations Label, Arity, and '.' (feature selection). Note - Don't forget to make assignment before S1. You should use ... when ranging from F1 to Fn.

```
if {Label X} == Lab then
  if {Arity X} == [F1 F2 .. Fn] then
    local X1 in
      local X2 in …
        X1 = X.F1
        X2 = X.F2 …
        S1
      end
    end
  else S2
  end
 else S2
 end
```

(c) Rewrite the following 'case' statement using 'if' statements

```
declare L
L = lab(f1:5 f2:7 f3:'jim')

case L of lab(f1:X f2:Y f3:Z) then
   case L.f1 of 5 then
      {Browse Y}
   else
      {Browse a}
   end
else
   {Browse b}
end
```

```
if {Label L} == lab then
  if {Arity L} == [f1 f3 f3] then
    local X Y Z in
      X = L.f1 Y = L.f2 Z = L.f3
      if L.f1 == 5 then {Browse Y} else {Browse a} end
    end
  else {Browse b} end
```

```
else {Browse b} end
```

3. Given the following procedure:

```
declare
proc {Test X}
  case X
  of a|Z then {Browse  'case (1)'}
  [] f(a) then {Browse  'case (2)'}
  [] Y|Z andthen Y==Z then {Browse  'case (3)'}
  [] Y|Z then {Browse  'case (4)'}
  [] f(Y) then {Browse  'case (5)'}
  else {Browse  'case (6)'} end
end
```

Without executing any code, predict what will happen when you feed
{Test [b c a]}, {Test f(b(3))}, {Test f(a)}, {Test f(a(3))}, {Test f(d)}, {Test [a b c]},
{Test [c a b]}, {Test a|a}, and {Test ´|´(a b c)}
Run the code to verify your predictions.

4. Given the following procedure:

```
declare
proc {Test X}
  case X of f(a Y c) then {Browse 'case (1)'}
  else {Browse  'case (2)'} end
end
```

(a) Without executing any code, predict what will happen when you feed
declare X Y {Test f(X b Y)}
declare X Y {Test f(a Y d)}
declare X Y {Test f(X Y d)}
Run the code to verify your predictions.

(b) Run the following example:

```
declare X Y
if f(X Y d)==f(a Y c) then {Browse 'case (1)'}
else {Browse 'case (2)'} end
```

Is the result different from the previous example? Explain.
Run the code to verify your predictions.

5. Given the following code:

```
declare Max3 Max5
  proc {SpecialMax Value ?SMax}
    fun {SMax X}
      if X>Value then X else Value end
  end
end
{SpecialMax 3 Max3}
{SpecialMax 5 Max5}
```

Without executing any code, predict what will happen when you feed
{Browse [{Max3 4} {Max5 4}]}

It is important to know that in question 4, the statements will suspend until X and Y are bound to values. In the case that we are matching f(X b Y) with f(a Y c), this will suspend because the case statement must know if X==a and Y==c to determine which statement to execute next.

In question 5, the argument Value in SpecialMax is begin captured in the contextual environment of SMax, when SMax is defined. This is why different calls to SpecialMax will cause different numbers (Value) to be captured in SMax.

6. Expand the following function SMerge into the kernel syntax.
Note - X#Y is a tuple of two arguments that can be written '#'(X Y).
The resulting procedure should be tail recursive if the rules from section 2.5.2 are followed correctly.

```
declare
fun {SMerge Xs Ys}
  case Xs#Ys
  of nil#Ys then Ys
  [] Xs#nil then Xs
  [] (X|Xr)#(Y|Yr) then
    if X=<Y then
      X|{SMerge Xr Ys}
    else
      Y|{SMerge Xs Yr}
    end
  end
end
```

```
local Smerge in
Smerge = proc {$ Xs Ys Zs}
  case Xs of nil then
    Zs = Ys
  else
    case Ys of nil then
      Zs = Xs
    else
      case Xs of XH|XT then
        case Ys of YH|YT then
          local ZH in
            local ZT in
              localZT C in
                Zs = ZH|ZT
                C = XH=<YH
                if C then
                  ZH = XH
                  {Smerge Xr YT ZT}
                else
                  ZH = YH
                  {Smerge XY Yr ZT}
                end
              end
            end
          end
        else
          skip
        end
      else
        skip
      end
```

```
    end
  end
end
end
```

Lab2

1. Write a more effecient version of the function Comb from section 1.3

(a) use the definition n choose r = n x (n-1) x ... x (n-r+1) / r x (r-1) x ... x 1 calculate the numerator and denominator separately, then divide the results. note: the solution is 1 when r = 0

```
fun {CombH X Y}
  if Y == 0 then X
  else X * {CombH X-1 Y-1}
  end
end

fun {Comb N R}
  if N == 0 then 1
  else {CombH N (N-R-1)} / {CombH R R}
  end
end
```

b) use the identity n choose r = n choose (n-r) to further increase efficiency So, if r > n/2 then do the calculation with n-r instead of r.

```
fun {NewComb N R}
  if R > N div 2 then {Comb N-R R} else {Comb N R} end
end
```

2. Based on the example of a correctness proof from section 1.6, write a correctness % proof for the function Pascal from section 1.5.

Lemma, Correctness of AddList by induction on the length of the List L1:
Base Case, (L1 = nil) then return nil
Inductive Hypothesis, {AddList L1' L2} is correct for L1' one element shorter than L1
Inductive Case, {AddList L1 L2} where L1 = H1|T1 and L2 = H2|T2
By the IH, {AddList T1 T2} is the correct sum of the lists T1 T2 because the length of the list T1 is one less than the length of L1, then adding (H1+H2) to the head of {AddList T1 T2} will produce the sum of the list L1 L2.

Proof of correctness for Pascal N by induction on N:
Base case (N = 1), {Pascal 1} returns [1] which is the first row of Pascal's triangle
Inductive Hypothesis: {Pascal K-1} is correct
Inductive case (K): in {Pascal K} the 'if' instruction takes the 'else' case, and executes {AddList {ShiftLeft {Pascal N-1}} {ShiftRight {Pascal N-1}}}. By the IH, {Pascal N-1} correctly returns the N-1th row of Pascals triangle, and by the Lemma above, AddList returns exactly the pairwise sum of the lists {ShiftLeft {Pascal N-1}} and {ShiftRight {Pascal N-1}}, which is exactly the definition of the Nth row of Pascal's Triangle.

3. Write a lazy function (section 1.8) that generates the list
N | N-1 | N-2 | ... | 1 | 2 | 3 | ... where N is a positive number

```
fun lazy {ListGen N B}
  if B == 0 then
    if N == 1 then N|{ListGen N+1 1}
    else N|{ListGen N-1 0}
    end
  else N+1|{ListGen N+1 1}
  end
end
```

4. Write a procedure (proc) that displays ({Browse}) the first N elements of a List and run this procedure on the list created in Q3

```
proc {Brow L N}
  if N == 0 skip
  else {Browse L.1} {Brow L.2 N-1}
  end
end
```

Browses one element at a time, otherwise you can save the list up to N elements, and browse that list all at once

```
{Brow {ListGen 5 0} 10}
```

6.

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

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

What does Browse display in each fragment? Explain.

In Fragment 1, 23 is displayed. The X declared by the first 'local' is shadowed before the assignment 'X=44' by a new identifier X. That inner X exists in a different environment than the outer X. and when the first 'end' is passed, the program moves to the outer environment, where X is bound to the value 23. In Fragment 2, 44 is displayed. There is only one environment, and one identifier X. The value in the cell X points to is changed from 23 to 44, hence 44 is displayed by the browse.

7. Define functions {Accumulate N} and {Unaccumulate N} such that the output of {Browse {Accumulate 5}} {Browse {Accumulate 100}} {Browse {Unaccumulate 45}} is 5, 105, and 60. This will be implemeted using memory cells (section 1.12).

```
declare C Accumulate Unaccumulate
C={NewCell 0}
proc {Accumulate N}
   C:=@C+N
   {Browse @C}
end
proc {Unaccumulate N}
   C:=@C-N
   {Browse @C}
end
{Accumulate 5}

{Accumulate 100}

{Unaccumulate 45}
```