Eric Smrkovsky
Lab 12
CSci 117
11/10/12019

Part 1

1a) Quantum: A specified number of statements that can be executed within a thread before the program suspends that thread and moves on to the next thread to execute the same number of statements. If the thread stops anyway because the thread has completed the quantum is reset for the next thread. When all of the threads in the program have had a chance to run the program resumes execution within the first thread that didn't finish execution and continues until all threads have finished executing and the program is finished.

Ex)

```
local T H A in      //1, 2, 3
skip Browse A       //4    //4
thread              //5    //5
  skip Browse H     //7    //9
  skip Browse H     //8    //10
  skip Browse H     //9    //11
  T = 5             //13   //12
  end
skip Browse T       //6    //6
skip Browse T       //10   //7
skip Browse T       //11   //8
skip Browse T       //12   //13
end
```

The program to the left when ran with a quantum of 3 (plain) will execute all of the browse statements before T has a chance to be bound to 5. But if ran with a quantum of 4 (yellow) the T is bound with 5 while there is still one browse statement left, so T = 5 will be displayed once. I have marked the flow of the program in the commented parts.

1b) Thread: A thread is a basically a small program created within a larger program and used when we want the program we are writing to do different things at their own pace (concurrency). Threads are used in Oz to control the dataflow of the program, if say we are using a quantum of 3 with 5 threads, each thread will execute 3 lines within each thread until they are all complete. A thread is an executing statement with it's own stack but shares the store with the rest of the program, it can suspend and execution can be altered with a schedular such as a quantum.

```
local B in
  thread B=true end
  if B then Y in
  Y = 6
  skip Browse Y end end
```

The program to the left assigns the variable B with a true value in it's own thread. Since the thread shares a store with the rest of the program, the if statement will run.

Eric Smrkovsky
Lab 12
CSci 117
11/10/12019

1c) Kernel syntax translations: It is important to realize that statements such as **case, if, local** and others have more lines of code then their syntactic sugar representations would have you believe. When using a quantum, the statements executed are measured by the expanded kernel syntax versions. So, in order to correctly predict execution of a program, we will want to know what statements are expanded and which aren't.

```
local T1 T2 in
 thread T2 = (7+9) end
 thread T1 = (4+3) end
 skip Browse T2
 skip Browse T1
 skip Browse T2
 skip Browse T1
 skip Browse T2
 skip Browse T1
end
```

In this program, the last browse statements for T2 and T1 are the only ones that actually show the values that are created within the two threads. This is because when the syntax here is expanded, a quantum of one will take a while to catch up within the two threads.

1d) Interleaving's: When threads take turns executing a little, they are considered to be interleaved. Threads could take turns altering the store, if two threads try to alter the same variable in the store an error could occur. Interleaved execution could also be very handy if a system is implemented with multiple processors. The program below is said to be interleaved because the threads take turns depending on the form of scheduling chosen.

```
local Z A B in
  Z = 12
  thread local X in
    X = 34
    skip Browse X
    skip Browse X
    skip Browse X
    end  end
  thread local Y in
    Y = 56
    skip Browse Y
    skip Browse Y
    skip Browse Y
    skip Browse Y
    end  end
  skip Basic
  skip Basic
  skip Browse Z
  skip Browse Z
```

Eric Smrkovsky
Lab 12
CSci 117
11/10/12019

2a) The program below has a stream producer function and a transducer function. The producer starts at 0 and goes to 25, the transducer takes the first element (if zero) and makes it 333555 and the last element 333555 as well, it also multiplies all even numbers by 3 and the odd ones by 5.

```
local Producer Out Stream TimesFiveThree in
  Producer = fun {$ N Limit}
    if (N<Limit) then
    (N|{Producer (N + 1) Limit})
    else nil
    end
  end

  TimesFiveThree = fun {$ Stream}
    case Stream of nil then
      333555          //this is last element
    [] '|'(1:H 2:T) then H2 H3 M in
      {Mod H 2 M}          //check if even
      if (M==0) then         //if even then
        H2 = (H * 5)          //multiply by 5
      else                 //otherwise multiply by 3
        H2 = (H * 3)
      end
      if (H2==0) then        //if head is zero
        H3 = 333555          // make same as last element
      else H3 = H2 end
      (H3|{TimesFiveThree T})
    end
  end

{Producer 0 25 Stream}
{TimesFiveThree Stream Out}
skip Browse Out
end
```

Out : [333555 3 10 9 20 15 30 21 40 27 50 33 60 39 70 45 80 51 90 57 100 63 110 69 120 333555]

Eric Smrkovsky
Lab 12
CSci 117
11/10/12019

2b)

```
local SquareTrdcr Gen Gen2 Gen3 G1 G2 G3 ST in

 Gen = fun{$ N} //generate ones
  fun {$} (N#{Gen (N+1)}) end
 end

 Gen2 = fun{$ N} //generate twos
  fun {$} (N#{Gen2 (N+2)}) end
 end

 Gen3 = fun{$ N} // generate threes
  fun {$} (N#{Gen3 (N+3)}) end
 end

 SquareTrdcr = fun{$ G1 G2 G3}
  fun {$}
    (H1#F1) = {G1}
    (H2#F2) = {G2}
    (H3#F3) = {G3}
    H in
    H = (H1+H2)
    ((H*H3)#{SquareTrdcr F1 F2 F3})
  end end

G1 = {Gen 0}
G2 = {Gen2 0}
G3 = {Gen3 0}
ST = {SquareTrdcr G1 G2 G3}

 local
   (X#F) = {ST}
   (X1#F1) = {F}
   (X2#F2) = {F1}
   (X3#F3) = {F2}
   (X4#F4) = {F3}
   (X5#F5) = {F4}
   List in
 List = [X X1 X2 X3 X4 X5]
 skip Browse List
 end end
```

The program produces every third square in an “interesting” way. The input is zero for the generators and the output is shown below. I made a list of the first 5 values of the output stream to show.

Output:

```
List : [ 0  9  36  81  144  225 ]
```

Eric Smrkovsky
Lab 12
CSci 117
11/10/12019

3a)

```
b_sort(Scramble,Out) :- switch(Scramble,STail), !,
                        b_sort(STail,Out).
b_sort(Out,Out).

switch([H,H2|T],[H2,H|T]) :- H2<H.
switch([H|T],[H|T2])      :- switch(T,T2).
```

Sample output

```
b_sort([8,5,2,4,8,9,6,5,2,0,1,4,5,6,3,2,0,1,4,2,2,0],X).
X = [0, 0, 0, 1, 1, 2, 2, 2, 2, 2, 3, 4, 4, 4, 5, 5, 5, 6, 6, 8, 8, 9]

b_sort([8,5,2,4,8,9,6,5,0,1,4,2,2,0],X).
X = [0, 0, 1, 2, 2, 2, 4, 4, 5, 5, 6, 8, 8, 9]

b_sort([8,5,2,4,1414,2,2,0],X).
X = [0, 2, 2, 2, 4, 5, 8, 1414]
```

3b1)

```
:- use_module(library(clpfd)).

n_kings(N, Ks) :-
        length(Ks, N),
        Ks ins 1..N,
        safe_kings(Ks).

safe_kings([]).
safe_kings([K|Ks]) :-
        safe_kings(Ks, K, 1),
        safe_kings(Ks).

safe_kings([], _, _).
safe_kings([K|_], K0, D0) :-
        K0 #\= K,
        abs(K0 - K) #\= D0.
```

3b2)

Couldn't figure this out. I tried and tried and tried. Trying to keep my prefect lab score perfect so please have mercy! =)