1.1

```
local B in
  thread            // S1
    B=true          // T1
  end

  thread            // S2
    B=false         // T2
  end

  if B then         // S3
    skip Browse B   // S3.1
  end

end

/*
S1 T1 S2 T2 S3 S3.1
  unification error as B is bound twice by T1 and T2 before the
    skip Browse statement

S1 T1 S2 S3 T2 S3.1
  unification error as B is bound twice by T1 and T2 before the
    skip Browse statement

S1 T1 S2 S3 S3.1 T2
  displays true as S3.1 occurs after T1 occurs and binds B, but before
    T2 runs and causes an error rebinding B

S1 S2 T1 T2 S3 S3.1
  unification error as B is bound twice by T1 and T2 before the
    skip Browse statement

S1 S2 T2 T1 S3 S3.1
  unification error as B is bound twice by T1 and T2 before the
    skip Browse statement

S1 S2 S3 T1 S3.1 T2
  displays true as S3.1 occurs after T1 occurs and binds B, but before
    T2 runs and causes an error rebinding B

S1 S2 S3 T2 S3.1 T1
  does not display, B is false as S3.1 occurs after T1 occurs and binds B,
    but before T1 runs and causes an error rebinding B

S1 S2 S3 S3.1 T1 T2
  S3 hangs until T1 is executed, then a unficiation error occurs caused
    by T2

S1 S2 S3 S3.1 T2 T1
  S3 hangs until T2 is executed, then a unficiation error occurs caused
    by T1

S1 S2 T1 S3 S3.1 T2
  displays true as S3.1 occurs after T1 occurs and binds B, but before
    T2 runs and causes an error rebinding B

*/
```

1.2

```
local X Y T in
  thread Y = X end
  X = 3
  skip Browse Y
end

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

/*
Infinity
*Hoz> runFullT (Infinity) "declarative threaded" "lab6-threads/part1-2.txt" "lab6-threads/part1-2.out"
Y : Unbound

T2 : Unbound

T1 : Unbound

Having the program run with a quantum of inifinity causes threads to complete before they move on to another thread.
This is why every variable is unbound as the thread gets removed from the stack on completion removing
all variables in its scope from the stack leaving them unbound.

Finite 1
*Hoz> runFullT (Finite 1) "declarative threaded" "lab6-threads/part1-2.txt" "lab6-threads/part1-2.out"
Y : 3

T2 : 3

T1 : Unbound
*/
```

Having the program run with a quantum of 1 causes the program so 'skip' around a bit while executing threads. This does not guarantee that a thread will be before moving onto other parts of the program allowing values in the thread to be accessed by other parts of the program. This can be see in the output of Y which is bound to 3 meaning the thread in the first part of the program did not complete before skip Browse Y was called. The same can be said for the thread found after T2 as the skip Browse was called before the thread completed leading to a bound output of 3. The thread for T1 did complete before skip Browse T1 was called which can be seen by its unbound output.

1.3

```
local Z in
  Z = 3
  thread local X in
    X = 1
    skip Browse X
    skip Browse X
    skip Basic
    skip Browse X
    skip Browse X
    skip Basic
    skip Browse X
    end
  end
  thread local Y in
    Y = 2
    skip Browse Y
    skip Basic
    skip Browse Y
    skip Browse Y
    skip Browse Y
    skip Basic
    skip Browse Y
    end
  end
  skip Browse Z
  skip Browse Z
  skip Browse Z
  skip Basic
  skip Browse Z
  skip Browse Z
 end
```

1.4

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

/*
The minimum quantum that will not cause the kernel to suspend is 5.
This is because the thread is only allowed to work on 4 statements
at once with a quantum of 4. There are 4 statements preceeding the
if statement when looking at the elborated syntax in the .out file.
Once the quantum is set to 5 the thread can work on 5 statements at
once and the fifth statement happens to be the if statement. This
causes the if statement to have to wait for B to be bound to
resolve the program suspension.
*/
```

1.5

		time	in	seconds						
file	n	8	9	10	11	12	13	14	15	16
f1sugar		0.06	0.13	0.029	0.069	1.169	4.23	10.77	28.73	72.25
f2sugar		0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.02
f1thread		0.15	0.34	0.86	2.23	5.55	14.34	36.56	96.99	

a) The results in the table occur because calculating the fib sequence recursively takes 2^n time. In fib2_sugar the fib sequence is found iteratively which takes n time and therefore does not ramp up in execution time nearly as fast as the other two methods.

b) By writing out the first few iterations we can quickly see a pattern for the number of threads created and the corresponding fib number

```
f(0) = 0 threads, 1
f(1) = 0 threads, 1
f(2) = 2 threads, 2
f(3) = 4 threads, 3
f(4) = 8 threads, 5
f(5) = 14 threads, 8
f(6) = 24 threads, 13
```

While a recurrence relation could be used to solve this problem, the table above makes the function quite obvious and removes the need for one. If we let the number of threads be a function T of n then we can calculate n number of threads to be:

```
T(n) = 2 * f(n) - 2
```

2.1

```
local A B C D Producer OddFilter in     //define A B C and D for start, end, stream output and filter output
                                        //define Produce and OddFilter for procedures
    Producer = proc {$ N Limit Out}
        if (N<Limit) then T N1 in
            Out = (N|T)
            N1 = (N + 1)
            {Producer N1 Limit T}
        else Out = nil
        end
    end

    OddFilter = proc {$ P Out}
        case P of nil then  //if P is nil then set Out as nil (base case)
            Out = nil
        [] '|'(1:X 2:Y) then T in   //if P has values remaining (recursive case)
            if ((X mod 2) == 0) then    //if X is even
                Out = (X|T)                 //set Out to tuple containing X and T (T is place holder for next value)
                {OddFilter Y T}             //recursively call OddFilter on remaining values (Y) given next value (T)
            else                        //else
                {OddFilter Y Out}           //recursively call OddFilter on reamining values (Y) and next value (Out)
            end
        end
    end

    A = 0               //set start value of stream to 0
    B = 100             //set limit to stream (stream will end at limit - 1)
    {Producer A B C}    //call producer and store stream in C
    {OddFilter C D}     //call filter and store result in D
    skip Browse D       //print out result
end
```

2.2

```
local InputStream Result Consumer in
    InputStream = [1 32 7 12 97 4 74]

    Consumer = fun {$ P} in
        case P of nil then
            0
        [] '|'(1:X 2:Xs) then R in
            (X + {Consumer Xs})
        end
    end

    Result = {Consumer InputStream}
    skip Browse Result
end
```

2.3

```
local Producer Consumer Summation A B C in
    Producer = thread
        proc {$ N Limit Out}
            if (N<Limit) then T N1 in
                Out = (N|T)
                N1 = (N + 1)
                {Producer N1 Limit T}
            else Out = nil
            end
        end
    end

    Consumer = thread
        proc {$ P Out}
            case P of nil then
                Out = nil
            [] '|'(1:X 2:Xs) then T in
                if ((X mod 2) == 0) then
                    Out = (X|T)
                    {Consumer Xs T}
                else
                    {Consumer Xs Out}
                end
            end
        end
    end

    Summation = thread
        fun {$ L}
            case L of nil then
                0
            [] '|'(1:X 2:Xs) then S in
                (X + {Summation Xs})
            end
        end
    end

    {Producer 0 100 A}
    {Consumer A B}
    C = {Summation B}
    skip Browse C
end
```