Javier Escareno
CSCI 117 Lab 6

Part 1 : Understanding threads

1. Possible sequences
    S1 T1 S2 S3 S3.1 T2 -- displays true
    S1 T1 S2 T2 -- displays an error
    S1 S2 T2 S3 T1 -- Displays error
    S1 S2 T1 T2 -- Displays error
    S1 S2 T2 S3 T1 -- Displays error
    S2 S2 R2 S3 S3.1 T1 -- Displays True
2. Quantum of infinity

```
        ghci> runFullT (Infinity) "declarative threaded" "thread.txt" "thread.out"
Y : Unbound

T2 : Unbound

T1 : Unbound

ghci> runFullT (Finite 1) "declarative threaded" "thread.txt" "thread.out"
Y : 3

T2 : 3

T1 : Unbound
```

Description : The difference between finite and infinity is that in the case of finite 1 when it is executed the values are Y:3 T2:3 and T1:Unbound while infinity has everything become unbound. This happens in the finite 1 case because of it being coveted from sugar to kernel and 4+3 cant be bound to T1 in kernel so it reverts to being unbound. When running the infinite Quantum Y,T1, and T2 are unbounded because as the main thread finishes it run the execution never happens and their are no values to bind to the variables.

3.) Code Update with Skip basic

```
ghci> runFullT (Finite 4) "declarative threaded" "thread.txt" "thread.out"

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
```

```
X : 1

X : 1

Y : 2

Z : 3

Z : 3

Z : 3

X : 1

X : 1

Y : 2
```

Y : 2

Y : 2

Z : 3

Z : 3

X : 1

Y : 2

4. Updated Interpreter

The quantum minimum that causes suspension to happen is 5. When the sugar syntax is converted to kernel syntax additional statements are added which then requires a minimum quantum of 5 to cause a suspension.

5. Fib sequence

X	Result	Fib1	Fib2	Fibthread
8	34	0.20 secs, 27,095,720 bytes	0.11 secs, 4,776,208 bytes	0.19 secs, 2,237,936 bytes
9	55	0.47 secs, 59,418,968 bytes	0.11 secs, 5,031,928 bytes	0.09 secs, 2,237,936 bytes
10	89	0.67 secs, 139,036,840 bytes	0.11 secs, 5,304,920 bytes	0.11 secs, 2,237,976 bytes
11	144	1.38 secs, 338,919,992 bytes	0.11 secs, 5,588,744 bytes	0.10 secs, 2,237,976 bytes
12	233	3.53 secs, 847,857,896 bytes	0.18 secs, 5,884,784 bytes	1.82 secs, 2,240,056 bytes
13	377	9.10 secs, 2,155,943,488 bytes	0.12 secs, 6,195,240 bytes	0.26 secs, 2,237,976 bytes

14	610	23.54 secs, 5,540,138,232 bytes	0.16 secs, 6,519,136 bytes	0.09 secs, 2,237,976 bytes
15	987	58.13 secs, 14,330,888,392 bytes	0.12 secs, 6,855,496 bytes	0.01 secs, 2,175,904 bytes

Fib2 which is an iterative version of the algorithm is the fastest as well as the most memory efficient because of the nature of iteration not increasing the procedures in the call stack. Fib1 comes in second place being the recursive version of fib as it calls alot of instructions to the stack. Fib thread comes in last for me as I could not get it to work properly and it kept giving the same time and bytes despite changing the x value.

Part B: (thread {Fib (In-1)} end + (Thread {Fib (In - 2)} end))

This version was remarkably slower as it call twice as many threads which makes the execution time take longer.

Number	0	1	2	3	4	5	6	7	8
Result	1	1	2	3	5	8	13	21	34

(thread {Fib (In-1)} end + {Fib (In - 2)} end)

This version was faster as it didn't have to make a second call to another thread when running, making it much faster.

Number	0	1	2	3	4	5	6	7	8
Result	1	1	2	3	5	8	13	21	34

A formula that would solve for N = n-1 + n-2 when n >1

Part 2 - Streams

```
local Producer OddFilter Consumer in

  proc {Producer N Limit Out}
    if (N < (Limit + 1)) then T N1 in
      Out = (N|T)
      N1 = (N + 1)
      {Producer N1 Limit T}
    else
```

```
        Out = nil
    end
end

proc {OddFilter P Out}
    case P
    of nil then
        Out = nil
    [] (X|Xs) then Filtered in
        {OddFilter Xs Filtered}

        if ((X mod 2) == 0) then Out = (X|Filtered)
        else Out = Filtered
        end
    end
end

fun {Consumer P} in
    case P
    of nil then 0
    [] (X|Xs) then (X + {Consumer Xs})
    end
end

// Example Testing
local N L P F C in
    N = 0
    L = 100

    // [0 1 2 .. 100]
    thread
        {Producer N L P}
        skip Browse P
    end

    // [0 2 4 .. 100]
    thread
        {OddFilter P F}
        skip Browse F
    end

    thread C = {Consumer F}
        skip Browse C
    end
    end
end
```