Part 1: Understanding threads

1. **All possible thread sequences leading up to the unification error**

   S1 S2 T1 S3.1 T2 -- displays true

   Same reason as above

   S1 S2 T1 S3 T2

   Doesn't display anything because T2 changes B to false, making the if B condition in S3 false.

   S1 S2 T1 T2

   Neither T1 nor T2 is inside an if condition block.

   S1 S2 T2 T1

   Same reason as above

   S1 T2 S2 S3 T1

   The order of execution of T2 and T1 doesn't change the result; B is set to true and then false.

   S1 S2 T2 S3 T1

   Same reason as above

   S1 T2 S2 T1

   S1 S2 S3 T1 S3.1 T2 -- displays true

   Displays "true" because T1 sets B to true, and S3.1 is executed due to the if B condition being true.

   S1 T1 S2 S3 T2

   S1 T1 S2 S3 S3.1 T

   Displays "true" because T1 sets B to true, and S3.1 is executed due to the if B condition being true.2 -- displays true

   S1 T1 S2 T2

   T2 changes B to false before the if B condition in S3 is evaluated.

   -SAS Unification error –

2. **-----Quantum of ‘Infinity’**

ghci> runFullT (Infinity) "declarative threaded" "Part1_2.txt" "Part1_2_kernel.txt"
Y : Unbound
T2 : Unbound
T1 : Unbound

Explanation: None of the threads can finish executing the program when the quantum is set to infinite. This is a result of the program attempting to browse Y, T2, and T1's values before any values have been given to them. As a result, these variables have "Unbound" values.

----Quantum of ‘Finite 1’

ghci> runFullT (Finite 1) "declarative threaded" "Part1_2.txt" "Part1_2_kernel.txt"
Y : 3
T2 : 3.
T1 : Unbound

Explanation: A quantum of one allows the program to run in a way that only allows one step per thread.
The first thread to run is Y = X, which sets Y's value to X's value of 3.
The next thread to run is T2 = thread 3, which gives T2 the thread value, 3.
Because it is not run inside the quantum, the thread T1 = thread (4 + 3) is left unbound.

3. **--Running the program without ‘skip basic’ statement with quantum of infinity and quantum of 4**

ghci> runFullT (Infinity) "declarative threaded" "Part1_3.txt" "Part1_3_kernel.txt"
Z : 3
Z : 3
Z : 3
Z : 3
Z : 3
X : 1
X : 1
X : 1
X : 1
X : 1
Y : 2
Y : 2
Y : 2
Y : 2
Y : 2

Explanation: Threads are free to run when the program is executed with a quantum of Infinity. This means that every thread gets the chance to run more than once. The values of Z, X, and Y are repeatedly output by the program.

```
ghci> runFullT (Finite 4) "declarative threaded" "Part1_3.txt" "Part1_3_kernel.txt"
X : 1
X : 1
Y : 2
Y : 2
Z : 3
Z : 3
Z : 3
Z : 3
X : 1
X : 1
X : 1
Y : 2
Y : 2
Y : 2
Z : 3
```

Explanation: The skip Browse statements are repeatedly executed by the program in a loop, resulting in many displays of the variable values. The results are determined by the values of X, Y, and Z as well as the order in which the skip Browse commands are executed.

-----Running the program with 'skip basic' statement with quantum of 4

Part1_32.txt file:

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

Output:

```
ghci> runFullT (Finite 4) "declarative threaded" "Part1_32.txt" "Part1_32_kernel.txt"
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
```

Explanation: Skip Basic statements are included in the provided code to control the execution order.
The values of X, X, Y, Z, Z, Z, X, X, Y, Y, Z, Z, X, Y are now displayed by the program in that particular order.
Because of the presence of skip Basic statements, which require a certain thread to run before others, the program produces this sequence.

4.

ghci> runFullT (Finite 3) "declarative threaded" "Part1_4.txt" "Part1_4txt_kernel.txt"
B : true()
[local ["B"] [thread [B = true()],local ["EXU1"] [EXU1 = B,if EXU1 then [skip/BB] else [skip]]]]

Explanation: When the program is run with a quantum of 3, it evaluates the expression B = thread true end and assigns B the value true(). When the if statement is evaluated, the true() value of B is checked. The skip Browse B statement is carried out because the value is known at this time and doesn't result in suspension.
The result demonstrates that B is true() and that no suspension was place.

ghci> runFullT (Finite 2) "declarative threaded" "Part1_4.txt" "Part1_4_kernel.txt"
B : true()
ghci> runFullT (Finite 3) "declarative threaded" "Part1_4.txt" "Part1_4_kernel.txt"
B : true()
ghci> runFullT (Finite 4) "declarative threaded" "Part1_4.txt" "Part1_4_kernel.txt"
B : true()

ghci> runFullT (Finite 5) "declarative threaded" "Part1_4.txt" "Part1_4_kernel.txt"
thread supsended: [(if EXU1 then [skip/BB] else [skip],[("EXU1",9),("B",8),("IntPlus",1),
("IntMinus",2),("Eq",3),("GT",4),("LT",5),("Mod",6),("IntMultiply",7)])]
B : true()

Explanation: When the program is run with a quantum of 5, it evaluates the expression B = thread true end and assigns B the value true(). When the if statement is evaluated, the true() value of B is checked. The skip Browse B statement is carried out because the value is known at this time and doesn't result in suspension.
After that, though, the program experiences a suspension since the thread that evaluates the if condition is suspended, and the result is "thread suspended."

5.

a.

Values of X	fib1_sugar	fib1_thread	fib2_sugar	Result
10	(0.01 secs, 1,366,500 bytes)	(0.42 secs, 210,865,136 bytes)	(0.02 secs, 4,377,590 bytes)	89
12	(0.10 secs, 1,467,600 bytes)	(1.90 secs, 1,396,400,225 bytes)	(1.30 secs, 1,300,325,240 bytes)	233
13	(2.60 secs, 1,560,700 bytes)	(0.10 secs, 1,500,236,220 bytes)	(1.50 secs, 11, 651,100 bytes)	377
30	(0.01 secs, 1,567,500 bytes)	(230.68 secs, 60,900,132 bytes)	(0.04 secs, 12,050,952 bytes)	1346269

At X=30 program took more than 1 minute for fib1_thread

Explanation: Depending on the value of X and the implementation method, the performance of various implementations varies. The threaded recursive version (fib1_thread) becomes inefficient for higher values of X because of the complexity of managing many threads, whereas the nonthreaded recursive form (fib1_sugar) is the quickest and most memory-efficient for small values of X. Even for larger values of X, the nonthreaded iterative version (fib2_sugar) runs well. This shows how recursion, threading, and iteration can be dealt off in various situations.

b.

n	Original Code	Modified Code
0	0	0
1	0	0
2	2	2
3	4	2
4	8	4
5	16	6
6	32	10
7	64	16
8	128	26

Explanation:

(Original Code)
Since n = 0 and 1 are base cases, no threads are created in these situations, and the function simply returns the value without making any recursive calls. Each recursive call (0 and 1) is carried out in a separate thread, resulting in the creation of two threads for n=2. There are a total of four threads established for n=3. This is because there are two threads in 2 and two more in 3, giving a total of four threads. Moving on, we continue in this manner. For n=4, there are two main threads (2 and 3), each of which generates two additional threads for a total of eight threads.

(Modified Code)

The number of threads created is decreased when the second recursive call in the Fibonacci code is modified to execute on the main thread. With this modification, there is no need to construct a separate thread for the second recursive call because it is computed sequentially in the main thread. Due to this adjustment, fewer threads are formed for each Fibonacci number, lowering the total number of threads created and simplifying the calculation.

Recurrence Relation:

The Fibonacci sequence is represented by the values in the "Original Code" column, where the number of threads doubles with each Fibonacci number. The "Modified Code" column shows how the optimization lowered the number of threads created.

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

T(n-1): The total number of threads created for the (n-1)th Fibonacci number.

T(n-2): The total number of threads created for the (n-2)th Fibonacci number.

C: The constant term representing the two main threads for the (n-1)th Fibonacci numbers.

Part 2: Streams

1.

```
OddFilter = proc {$ P Out}
OddFilterFun=fun{$Xs Res}
        if (P <> nil) then T N in
        if (N mod 2 = 0) then Out = N
        else Out = nil
        end
        N = P
        end
        else Out = nil
        end
 end
```

2.

```
case Ys
of Y|Yr then {Consumer Yr (Y+A)}
[] nil then A end
End
```

Explanation: If Ys has the structure Y|Yr, the stream has a head element named Y and the rest as Yr. In this instance, Yr and an updated accumulator (Y+A) are passed to the Consumer each time it is called.

It returns the value of the accumulator A if Ys is null, which stands for an empty stream.

3.

```
Fun{IsOdd Y}
   Y mod 2\=0
end
fun {Generate X Limit}
   if X<Limit then
      X|{Generate X+1 Limit}
   else nil
   end
fun {Sum Ys A}
  case Ys
   of Y|Yr then {Sum Yr A+Y}
   [] nil then A
   end
  end
local Ys Ps S in
 thread Ys={Generate 0 100}
  end
  thread Ps={Filter Ys IsOdd}
   end
   thread S={Sum Ys 0} end
   {
    Browse S end
    }
  end
```