Spring, 2009 Name: _____

COP 4020 — Programming Languages 1

Test on Declarative Concurrency

Special Directions for this Test

This test has 9 questions and pages numbered 1 through 9.
This test is open book and notes.
If you need more space, use the back of a page. Note when you do that on the front.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
Clarity is important; if your programs are sloppy and hard to read, you may lose some points. Correct syntax also makes a difference for programming questions.
When you write Oz code on this test, you may use anything in the declarative concurrent model (as in chapters 2–4 of our textbook). So you must not use not use cells or the library functions `IsDet` and `IsFree`. But please use all linguistic abstractions and syntactic sugars that are helpful.
You are encouraged to define functions or procedures not specifically asked for if they are useful to your programming; however, if they are not in the Oz base environment, then you must write them into your test. (This means you can use functions in the Oz base environment such as `Map`, `FoldR`, `Filter`, `Append`, `IntToFloat`, `List.take`, `List.drop`, etc.)
Recall that the grammar for the type of infinite streams with elements of type T is as follows.

⟨IStream T⟩ ::= `T '|'` ⟨IStream T⟩

Also recall that in Oz, strings are just lists of characters and that characters are just (small) integers.

For Grading

Question:	1	2	3	4	5	6	7	8	9	Total
Points:	10	12	5	15	10	15	15	3	15	100
Score:										

1. (10 points) [Concepts] The Oz demand-driven concurrent model uses lazy execution. Which of the following is a true statement about lazy execution? (Circle the letter of the correct answer.)

   A. In the demand-driven concurrent model, threads are only killed when they need to be killed.

   B. When a call to a lazy function is made, the system immediately starts computing the answer to that call in a new thread.

   C. When the answer to a call to a lazy function is needed, a new thread is started to compute the answer.

   D. Lazy functions are executed so quickly that they cannot have race conditions.

2. [Concepts]

   This question involves expressions executed in an environment where the following declaration and binding for the function `By2` have been made.

```
declare
fun lazy {By2 N} N | {By2 N+2} end
```

   For each part, circle the correct answer's letter.

   (a) (3 points) Suppose we execute the expression `{By2 1}`, what happens?

   A. The expression loops forever, since there is no base case to stop the execution of `By2`, and since the call generates a need for the result.

   B. The expression partially terminates, because it does not generate a need for the result of the call to `By2`.

   C. The expression fails or encounters an error.

   (b) (3 points) If we execute the expression `local Z in Z = {By2 1} Z end` what happens?

   A. The expression loops forever, because unifying the result of a call with an undetermined dataflow variable generates a need.

   B. The expression partially terminates, because unifying the result of a call with an undetermined dataflow variable does not generate a need.

   C. The expression fails or encounters an error.

   (c) (3 points) If we execute the expression `local H1 Ans T Z in Z={By2 1} H1|Ans|T=Z Ans end` what happens?

   A. The expression loops forever (thus encountering an error), because unifying with a pattern generates a need, and when the `By2` function starts running, it has no way to stop.

   B. The expression partially terminates, returning 3, because the statement `H1|Ans|T=4020` generates a need for the first two elements of the result, but the calls to `By2` stop, since it is lazy.

   C. The expression encounters an error due to some reason other than looping forever.

   (d) (3 points) If we execute the expression `{Length {By2 1}}` what happens?

   A. The expression loops forever (or has an error when trying to do so), because the call to `Length` generates needs for each element in the result of `{By2 1}`.

   B. The expression terminates normally, returning 42, because 42 is the answer to all questions.

   C. The expression encounters an error due to some reason other than looping forever.

3. (5 points) [Concepts]

   Why is declarative concurrency useful? (Circle the letter of the correct answer.)

   A. Because a program written in the declarative concurrent model can merge multiple independent user inputs, as might occur in a client-server application, such as eBay or amazon.com.

   B. Declarative concurrency is useful because it helps programs take longer to compute their outputs, thus prompting users to buy faster computers.

   C. Declarative concurrency is useful because it can't be used in a large program, which limits the design choices available to programmers, thus simplifying programming tasks.

   D. Declarative concurrency is useful because it makes reasoning and debugging difficult, due to the use of multiple threads, which provides work for computer scientists.

   E. Declarative concurrency is useful because it simplifies reasoning about and debugging programs, since each part of a program may have only one possible answer.

4. This question is about programming in the declarative concurrent model.

   (a) (10 points) [UseModels] Using the declarative concurrent model, write an incremental lazy function,

   ```
   LSelect : <fun lazy {$ <IStream T> <fun {$ T}: Bool>}: <IStream T>>
   ```

   that takes an infinite stream (i.e., an `<IStream>`), `IStrm`, with elements of some type `T`, and a predicate `Pred`, that takes an element of type `T` and returns a Boolean. A call such as `{LSelect IStrm Pred}` lazily returns an infinite stream that contains each element `E` of `IStrm` for which `{Pred E}` is true, and no other elements. (The ordering of elements in the answer preserves the ordering seen in the input stream `IStrm`.) Be sure that your `LSelect` function does not loop forever when its input is infinite.

   The following are examples, using the `Test` procedure from the homework.

   ```
   \insert 'LSelect.oz'
   \insert 'TestingNoStop.oz'
   declare % the next 3 functions are simply for use in testing
   fun lazy {From N} N|{From N+1} end
   fun {IsOdd N} (N mod 2) == 1 end
   fun {IsEven N} (N mod 2) == 0 end
   {Test {List.take {LSelect {From 1} IsOdd} 6} '==' [1 3 5 7 9 11]}
   {Test {List.take {LSelect {From 1} IsEven} 9} '==' [2 4 6 8 10 12 14 16 18]}
   {Test {List.take {LSelect {From 1} IsEven} 9} '==' [2 4 6 8 10 12 14 16 18]}
   {Test {List.take {LSelect {From 100} IsEven} 5} '==' [100 102 104 106 108]}
   {Test {List.take {LSelect {From 100} fun {$ E} E < 107 orelse E == 999 end} 8}
                      '==' [100 101 102 103 104 105 106 999]}
   ```

   (b) (5 points) Is the function `LSelect` defined by your answer incremental? (Say "yes, it is incremental" or "no, it's not incremental.") Briefly explain.

5. (10 points) [UseModels] Write a lazy function

   `RepeatingListOf : <fun lazy {$ <List T>}: <IStream T> >`

   that, for some type T takes a non-empty finite list `Elements` of elements of type T, and lazily returns an infinite stream (i.e., an `<IStream` T`>`) whose elements repeat the individual elements of `Elements` endlessly. The following are examples, written using the `Test` method from the homework.

```
\insert 'RepeatingListOf.oz'
\insert 'TestingNoStop.oz'
{Test {Nth {RepeatingListOf [1]} 1} '==' 1}
{Test {Nth {RepeatingListOf [7]} 500} '==' 7}
{Test {Nth {RepeatingListOf [2]} 999999} '==' 2}
{Test {List.take {RepeatingListOf [2 3 4]} 7} '==' [2 3 4 2 3 4 2]}
{Test {List.take {RepeatingListOf [a b c d e]} 11} '==' [a b c d e a b c d e a]}
{Test {List.take {RepeatingListOf "homework!"} 11} '==' "homework!ho"}
```

6. (15 points) [UseModels]

   Write a lazy function

   `BlockIStream: <fun lazy {$ <IStream <Char>> <Int>}: <IStream <List <Char> > >`

   that takes an infinite stream of characters, `IStrm`, and an integer, `BlockSize`, and which lazily returns an infinite stream of lists of characters, where each list in the result has exactly `BlockSize` elements, consisting of the first `BlockSize` elements of `IStrm`, followed by a list containing the next `BlockSize` elements of `IStrm`, and so on. That is, a call to `BlockIStream` chunks the characters in `IStrm` into strings of length `BlockSize`. (Hint: in your solution, you may want to use `List.take` and `List.drop`. Note that `{List.drop L N}` returns the list `L` without the first `N` elements.)

```
\insert 'BlockIStream.oz'
\insert 'RepeatingListOf.oz' % from problem above, used here to make the tests
\insert 'TestingNoStop.oz'
declare
fun lazy {From Char} Char|{From Char+1} end % for testing only
{Test {List.take {BlockIStream {From &a} 3} % &a is the character a
       8}
 '==' ["abc" "def" "ghi" "jkl" "mno" "pqr" "stu" "vwx"]}
{Test {List.take {BlockIStream {RepeatingListOf "Now is the time for..."} 1}
       12}
 '==' ["N" "o" "w" " " "i" "s" " " "t" "h" "e" " " "t"]}
{Test {List.take {BlockIStream {RepeatingListOf "Now is the time for..."} 3}
       4}
 '==' ["Now" " is" " th" "e t"]}
{Test {List.take {BlockIStream {RepeatingListOf "Now is the time for..."} 3}
       9}
 '==' ["Now" " is" " th" "e t" "ime" " fo" "r.." ".No" "w i"]}
{Test {List.take {BlockIStream {RepeatingListOf "Now is the time for..."} 6}
       7}
 '==' ["Now is" " the t" "ime fo" "r...No" "w is t" "he tim" "e for."]}
```

7. (15 points) [UseModels]

Write a lazy function

```
EncryptIStream: <fun lazy {$ <IStream <Char>> <Int> <fun {$ <List <Char> >}: <List <Char> >}
                 : <IStream <List <Char> > >
```

that takes 3 arguments: an infinite stream of characters, `IStrm`, an integer, `BlockSize`, and a function, `Encrypt`. The `EncryptIStream` function lazily returns an infinite stream of lists of characters, where each list in the result is the result of applying `Encrypt` to a list containing `BlockSize` elements of `IStrm`. The first list in the result is the result of applying `Encrypt` to the first `BlockSize` elements of `IStrm`, and this is followed by the result of applying `Encrypt` to the next `BlockSize` elements of `IStrm`, and so on. That is, a call to `BlockIStream` first chunks the characters in `IStrm` into strings of length `BlockSize`, and then applies `Encrypt` to each resulting list of strings. The following are some examples, written using various (cryptographically poor) `Encrypt` functions.

```
\insert 'EncryptIStream.oz'
\insert 'RepeatingListOf.oz' % from problem above, used here to make the tests
\insert 'TestingNoStop.oz'
declare
fun {NoEncryption Str} Str end
fun {ReverseNoEncryption Str} {Reverse {NoEncryption Str}} end
fun {CeaserCypher Str} {Map Str fun {$ C}(C+1) mod 512 end} end
fun {ReverseCeaserCypher Str} {Reverse {CeaserCypher Str}} end
{Test {List.take {EncryptIStream {RepeatingListOf "Now is the time for..."} 12 NoEncryption}
       3}
 '==' ["Now is the t" "ime for...No" "w is the tim" ]}
{Test {List.take {EncryptIStream {RepeatingListOf "We're off to see the Wizard, the Wonderful Wizard of Oz"}
                  3 ReverseNoEncryption}
       18}
 '==' ["'eW" " er" "ffo" "ot " "es " "t e" " eh" "ziW" "dra" "t ," " eh"
       "noW" "red" "luf" "iW " "raz" "o d" "O f"]}
{Test {List.take {EncryptIStream {RepeatingListOf "Now is the time for..."} 5 CeaserCypher}
       7}
 '==' ["Opx!j" "t!uif" "!ujnf" "!gps/" "//Opx" "!jt!u" "if!uj"]}
{Test {List.take {EncryptIStream {RepeatingListOf "Now is the time for..."} 5 ReverseCeaserCypher}
       7}
 '==' ["j!xpO" "fiu!t" "fnju!" "/spg!" "xpO//" "u!tj!" "ju!fi"]}
```

Please write your answer below.

```
\insert 'BlockIStream.oz' % So you can use BlockIStream in your answer.
```

8. (3 points) [Concepts] Does your function `EncryptIstream` in question 7 have to be lazy? Answer "yes" or "no" and give a brief explanation.

9. (15 points) [UseModels] Consider the following grammar for binary trees of atoms:

   ⟨BTree⟩ ::= ⟨Atom⟩ | `tree(`⟨BTree⟩ ⟨BTree⟩`)`

   where ⟨Atom⟩ is an Oz atom (or symbol), such as `leaf`, `a`, or `b`.

   Using Oz's demand-driven concurrent model, write a lazy function

   ```
   BTGenerate: <fun lazy {$ <BTree> <BTree> <BTree>} : <IStream <BTree> > >
   ```

   that takes three ⟨BTree⟩s, and lazily returns an infinite stream of ⟨BTree⟩s, as described below. (This might be useful for testing functions over such trees.) The returned infinite stream contains the three arguments as its first three elements, and then contains a tree containing the original second and third argument (as left and right subtrees), then a tree containing the original third argument and the fourth element of the output list, and so on. (In general, the first three elements of the result are the original first three arguments, and for $i > 3$, the i^{th} element of the result is a record of the form `tree(`X Y`)`, where X is the $(i-2)^{th}$ element of the result and Y is the $(i-1)^{th}$ element of the result.) Hint: you can use the three arguments as accumulators; they can keep track of the information your function will need. The following are examples that use the `Test` procedure from the homework.

```
\insert 'BTGenerate.oz'
\insert 'TestingNoStop.oz'
{Test {List.take {BTGenerate leaf tree(a leaf) tree(leaf b)} 6}
 '==' [leaf tree(a leaf) tree(leaf b)
       tree(tree(a leaf) tree(leaf b))
       tree(tree(leaf b) tree(tree(a leaf) tree(leaf b)))
       tree(tree(tree(a leaf) tree(leaf b))
            tree(tree(leaf b) tree(tree(a leaf) tree(leaf b))))]}
{Test {List.take {BTGenerate leaf tree(two tree(leaf leaf))
                             tree(leaf three)} 6}
 '==' [leaf tree(two tree(leaf leaf)) tree(leaf three)
       tree(tree(two tree(leaf leaf)) tree(leaf three))
       tree(tree(leaf three) tree(tree(two tree(leaf leaf)) tree(leaf three)))
       tree(tree(tree(two tree(leaf leaf)) tree(leaf three))
            tree(tree(leaf three)
                 tree(tree(two tree(leaf leaf)) tree(leaf three))))]}
{Test {List.take {BTGenerate tree(a b) tree(tree(c leaf) d) leaf} 7}
 '==' [tree(a b) tree(tree(c leaf) d) leaf
       tree(tree(tree(c leaf) d) leaf)
       tree(leaf tree(tree(tree(c leaf) d) leaf))
       tree(tree(tree(tree(c leaf) d) leaf)
            tree(leaf tree(tree(tree(c leaf) d) leaf)))
       tree(tree(leaf tree(tree(tree(c leaf) d) leaf))
            tree(tree(tree(tree(c leaf) d) leaf)
                 tree(leaf tree(tree(tree(c leaf) d) leaf))))]}
```