Spring, 2009 Name: _____

COP 4020 — Programming Languages 1

# Test on Declarative Concurrency

## Special Directions for this Test

This test has 8 questions and pages numbered 1 through 7.
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`, etc.)

### For Grading

| Question: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Total |
|---|---|---|---|---|---|---|---|---|---|
| Points: | 8 | 12 | 10 | 15 | 10 | 10 | 10 | 25 | 100 |
| Score: | | | | | | | | | |

1. (8 points) [Concepts] In a language like Java, multithreaded programs have race conditions and are thus observably observably nondeterministic. Which of the reasons below describes why the declarative concurrent model in Oz is observably deterministic? (Circle the letter of the correct answer.)

   A. The declarative concurrent model in Oz is too lazy to have race conditions.

   B. The declarative concurrent model in Oz has no mutable state.

   C. The declarative concurrent model in Oz does not allow multiple threads to run at the same time.

   D. The declarative concurrent model in Oz is too slow to have race conditions.

2. [Concepts]

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

```
declare
fun lazy {HundredsPlus N M}  N*100 + M  end
```

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

   (a) (3 points) Suppose we execute the expression `{HundredsPlus 40 20}`, what happens?

   A. The expression terminates normally, returning 4020, because the call to `HundredsPlus` itself generates a need.

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

   C. The expression fails (throwing an exception).

   (b) (3 points) If we execute the expression `local Z in Z = {HundredsPlus 40 20} Z end` what happens?

   A. The expression terminates normally, returning 4020, 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 the call to `HundredsPlus` does not generate a need for the call's result.

   C. The expression fails (throwing an exception).

   (c) (3 points) If we execute the expression `local Z in Z={HundredsPlus 40 20} Z=4020 Z end` what happens?

   A. The expression terminates normally, returning 4020, because the statement `Z=4020` generates a need.

   B. The expression partially terminates, because the call to `HundredsPlus` suspends and so execution goes no further.

   C. The expression fails (throwing an exception).

   (d) (3 points) If we execute the expression `{HundredsPlus 40 20} + 0` what happens?

   A. The expression terminates normally, returning 4020, because the '+' operation generates a need.

   B. The expression partially terminates, because the call to `HundredsPlus` suspends and so execution goes no further.

   C. The expression fails (throwing an exception).

3. [Concepts] [MapToLanguages]

The homework compared Java (or C#) iterators to lazy streams in Oz. This question extends that analogy.

(a) (5 points) Can an iterator in Java act like a lazily generated stream in Oz, in the sense that it only computes the next value in the iteration when it is needed (if ever)? Briefly explain.

(b) (5 points) We discussed two kinds of flow control for stream-based systems in Oz: data-driven and demand-driven. Which of these two kinds of flow control best describes computations built on iterators in Java (or C#)? Briefly explain.

4. [Concepts] Consider the following code (from the textbook).

```
% From page 295 of CTM
declare
fun lazy {LMap Xs F}
   case Xs
   of nil then nil
   [] X|Xr then {F X}|{LMap Xr F}
   end
end
```

(a) (5 points) Does this code work on (i) infinite streams, (ii) only on finite lists, or (iii) on both finite and infinite streams? Briefly explain.

(b) (10 points) Is the function `LMap` defined by this code incremental? (Say "yes, it is incremental" or "no, it's not incremental.") Briefly explain.

5. (10 points) [UseModels]

   Write a lazy function

   ```
   CAvgIter: <fun lazy {$ <IStream <Float>> <Float> <Float>}: <IStream Float>>
   ```

   that takes an infinite stream of Floating point numbers, $X_1|X_2|X_3|\ldots$, a Floating point number `Sum`, and a Floating point number `Count`, and lazily produces an infinite stream that contains

   $$((\texttt{Sum}+X_1)/(\texttt{Count}+1.0))|((\texttt{Sum}+X_1+X_2)/(\texttt{Count}+1.0+1.0))|((\texttt{Sum}+X_1+X_2+X_3)/(\texttt{Count}+1.0+1.0+1.0))|\ldots$$

   That is, the result is the infinite stream whose $i^{th}$ item, counting from 1 is $(\texttt{Sum}+\sum_{j=1,i} X_j)/(\texttt{Count}+i\times 1.0)$. The following are examples, that use the `WithinTest` method from the homework's `FloatTesting` file. (Recall that `WithinTest` checks that the actual answers are within a standard tolerance of $0.001$, compared to the expected answers.)

   ```
   \insert 'CAvgIter.oz'
   \insert 'FloatTesting.oz'
   local ETC in
      {WithinTest {List.take {CAvgIter 3.0|6.0|9.0|12.0|15.0|ETC 0.0 0.0} 5} '~=~' [3.0 4.5 6.0 7.5 9.0]}
   end
   local ETC in
      {WithinTest {List.take {CAvgIter 1.0|1.0|8.0|2.0|3.4|5.0|7.5|3.2|ETC 0.0 0.0} 7}
       '~=~' [1.0 1.0 3.3333 3.0 3.08 3.4 3.9857]}
   end
   local Ones = 1.0|Ones in
      {WithinTest {List.take {CAvgIter Ones 0.0 0.0} 10}
       '~=~' [1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0]}
      {WithinTest {List.take {CAvgIter 2.0|2.0|2.0|Ones 0.0 0.0} 10}
       '~=~' [2.0 2.0 2.0 1.75 1.6 1.5 1.4286 1.375 1.3333 1.3]}
      {WithinTest {List.take {CAvgIter Ones 6.0 3.0} 10}
       '~=~' [1.75 1.6 1.5 1.4286 1.375 1.3333 1.3 1.2727 1.25 1.2308]}
   end
   ```

   Your solution should avoid any repeated code. In Oz `A/B` divides `A` by `B`, assuming both are Floating point numbers.

6.

(a) (5 points) Does the function `CAvgIter` in the question 5 have to be lazy? Answer "yes" or "no" and give a brief explanation.

(b) (5 points) Consider the following function that computes the cumulative average of an infinite stream, using the code from question 5.

```
\insert 'CAvgIter.oz'  % insert the code from question 4
declare
fun {CumulativeAverage IStrm}
   {CAvgIter IStrm 0.0 0.0}
end
```

Should this function, `CumulativeAverage`, have been declared to be lazy? Answer "yes, it should be lazy" or "no, it need not be lazy" and give a brief explanation.

7. (10 points) [UseModels]

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

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

   that takes an infinite stream, `IStrm`, whose elements are of some type `T`, and a strictly positive integer, `N`, and produces an infinite stream of lists of type `T`, such that the $i^{th}$ element of the result is an `N`-element list consisting of the `N` elements numbered $i, i+1, ..., i+\texttt{N}-1$ from `IStrm`. The following are examples that use the `Test` procedure from the homework.

```
\insert 'SizeNSublists.oz'
\insert 'TestingNoStop.oz'
declare
fun lazy {From M} M|{From M+1} end
{Test {List.take {SizeNSublists {From 1} 2} 10}
 '==' [[1 2] [2 3] [3 4] [4 5] [5 6] [6 7] [7 8] [8 9] [9 10] [10 11]]}
{Test {List.take {SizeNSublists {From 1} 3} 7}
 '==' [[1 2 3] [2 3 4] [3 4 5] [4 5 6] [5 6 7] [6 7 8] [7 8 9]]}
{Test {List.take {SizeNSublists {From 2} 5} 6}
 '==' [[2 3 4 5 6] [3 4 5 6 7] [4 5 6 7 8] [5 6 7 8 9] [6 7 8 9 10] [7 8 9 10 11]]}
{Test {List.take {SizeNSublists {From 22} 1} 9}
 '==' [[22] [23] [24] [25] [26] [27] [28] [29] [30]]}
{Test {List.take {SizeNSublists {From 100} 7} 2}
 '==' [[100 101 102 103 104 105 106] [101 102 103 104 105 106 107]]}
local ETC in
   {Test {List.take {SizeNSublists a|b|c|d|e|f|g|a|b|c|ETC 4} 5}
    '==' [[a b c d] [b c d e] [c d e f] [d e f g] [e f g a]]}
end
```

8. (25 points) [UseModels]

Using Oz's declarative concurrent model, write a function

```
MovingAverage: <fun lazy {$ <IStream <Float>> <Int>>} : <IStream <Float>> >
```

that takes an infinite stream of Floating point numbers, `IStrm`, and a positive integer `N`, and returns an infinite stream of Floating point numbers representing the moving average of `IStrm`. The moving average is computed by taking sublists of size `N`, and averaging each such sublist. Thus the $i^{th}$ element of the result is $A_i$, where $A_i$ is the average of elements numbered $i + \texttt{N} - 1, i + \texttt{N} - 2, \ldots, i$, which is the sum of those elements divided by `N`. For example, if `N` is 2, then $A_1$ is the average of elements 2 and 1 in `IStrm`, and $A_2$ is the average of elements 3 and 2. The following are examples, that use the `WithinTest` method from the homework's `FloatTesting` file.

```
\insert 'MovingAverage.oz'
\insert 'FloatTesting.oz'
declare
{StartTesting 'MovingAverage'}
local Ones = 1.0|Ones  Down = 10.0|9.0|8.0|7.0|6.0|5.0|4.0|3.0|2.0|Ones in
   {WithinTest {List.take {MovingAverage Ones 20} 10} '~=~' [1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0]}
   {WithinTest {List.take {MovingAverage Down 2} 11} '~=~' [9.5 8.5 7.5 6.5 5.5 4.5 3.5 2.5 1.5 1.0 1.0]}
   {WithinTest {List.take {MovingAverage Down 4} 11} '~=~' [8.5 7.5 6.5 5.5 4.5 3.5 2.5 1.75 1.25 1.0 1.0]}
   {WithinTest {List.take {MovingAverage Down 6} 11} '~=~' [7.5 6.5 5.5 4.5 3.5 2.6667 2.0 1.5 1.1667 1.0 1.0]}
end
fun lazy {FromByF X Delta} X|{FromByF X+Delta Delta} end
{WithinTest {List.take {MovingAverage {FromByF 1.0 1.0} 2} 7} '~=~' [1.5 2.5 3.5 4.5 5.5 6.5 7.5]}
{WithinTest {List.take {MovingAverage {FromByF 3.0 2.0} 3} 5} '~=~' [5.0 7.0 9.0 11.0 13.0]}
{WithinTest {List.take {MovingAverage {FromByF 100.0 ~10.0} 2} 5} '~=~' [95.0 85.0 75.0 65.0 55.0]}
{WithinTest {List.take {MovingAverage {FromByF 100.0 ~10.0} 3} 5} '~=~' [90.0 80.0 70.0 60.0 50.0]}
```

Hint, you can use `SizeNSublists` from the previous problem (even if you haven't written that yet). Also, since in Oz, the division operator `/` requires two Floating point numbers as arguments, you will need to use `IntToFloat`.