Spring, 2008 Name: ______________________________

COP 4020 — Programming Languages 1

Test on Declarative Programming Techniques and 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 cells and assignment in your Oz solutions. (Furthermore, note that the declarative concurrent model does not include the primitive `IsDet` or the library function `IsFree`; thus you are also prohibited from using either of these functions in your solutions.) 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`, etc.)

For Grading

Problem	Points	Score
1	10	
2	10	
3	10	
4	10	
5	10	
6	10	
7	20	
8	20	

1. (10 points) [Concepts] Write a function Consider the following Oz code.

```
declare
fun {C F G}
   fun {$ X} {F {G {F X}}} end
end

fun {U F}
   fun {$ X}
      fun {$ Y} {F X Y} end
   end
end

fun {Add A B} A+B end

{Show {{C {{U Add} 2} {{U Add} 3}} 100}}
```

   (a) Does this code terminate normally, partially terminate, or fail (with an exception)?

   (b) If it partially terminates or completes normally, give its output (or write "no output" if there is none); otherwise briefly explain why it fails.

2. (10 points) [Concepts] Consider the following Oz program.

```
declare
fun {F X Y}
   if X==Y then 85 else 11 end
end

thread
   local A B in
      local R = {F A B} in
         {Show first(R A B)}
      end
   end
end

local D E in
   thread E=D+0 end
   thread D=5 end
   local R = {F D E} in
      {Show second(R D E)}
   end
end
```

   (a) Does this code terminate normally, partially terminate, or fail (with an exception)?

   (b) If it partially terminates or completes normally, give one possible output (or write "no output" if there is none); otherwise briefly explain why it fails.

3. (10 points) [Concepts] [EvaluateModels] What is the advantage of having referential transparency in a language with concurrency?

4. (10 points) [UseModels] [Concepts] In Oz, write a function

   ```
    Curry2: <fun {$ <fun {$ T U}: S>}: <fun {$ T}: <fun {$ U}: S>>>
   ```

   which takes a function a two-argument function `F` and returns a curried version of `F`. (You are supposed to know what "currying" a function means.) The following are examples, that use the `Test` method from the homework.

   ```
   declare
   fun {Mult A B} A * B end
   fun {Pair X Y} X#Y end
   {StartTesting 'Curry2'}

   {Test {{{Curry2 Mult} 3} 5} '==' 15}
   {Test {{{Curry2 Mult} 7} 10} '==' 70}
   {Test {{{Curry2 Pair} 7} 10} '==' 7#10}
   {Test {{{Curry2 Pair} 9} 66} '==' 9#66}
   local MapC = {Curry2 Map} in
      local Map123 = {MapC [1 2 3]} in
         {Test {Map123 {{Curry2 Mult} 5}} '==' [5 10 15]}
         {Test {Map123 {{Curry2 Pair} a}} '==' [a#1 a#2 a#3]}
      end
   end
   {Test {{{Curry2 fun {$ X Y} X*Y + 100 end} 2} 11} '==' 122}
   ```

5. (10 points) [UseModels] Using `FoldR` write a function

```
 Positive: <fun {$ <List Int>}: <List Int>>
```

that takes a list of integers `Nums` and produces a list that contains just the strictly positive elements of `Nums`, in their original order. The following are examples, that use the `Test` method from the homework. (Note that `~3` is the Oz way of writing negative numbers, such as -3.)

```
{Test {Positive nil} '==' nil}
{Test {Positive [~1]} '==' nil}
{Test {Positive [3 7 ~1]} '==' [3 7]}
{Test {Positive [~3 3 7 ~1]} '==' [3 7]}
{Test {Positive [~2 0 5 ~3 3 7 ~1]} '==' [5 3 7]}
{Test {Positive [0 5 ~3 3 7 ~1]} '==' [5 3 7]}
```

Write your answer below by filling in the arguments in the call to `FoldR` in the following.

```
fun {Positive Nums}
   {FoldR











     }
end
```

6. (10 points) [UseModels] Using Oz's **for** loop with `collect:` write a function `LazyLanguages`, which takes a list DB of records of the form `lang(name:` N `model:` M `evaluation:` L`)` and returns a list of records of the form `lang(name:` N `model:` M`)` for each record of the form `lang(name:` N `model:` M `evaluation:` L`)` whose `evaluation` field L is equal to `'lazy'`. The answer should also preserve the original ordering of the records. The following are examples.

```
declare
PLDB = [lang(name: 'C' model: procedural evaluation: eager)
        lang(name: 'C++' model: objectOriented evaluation: eager)
        lang(name: 'C#' model: objectOriented evaluation: eager)
        lang(name: 'Java' model: objectOriented evaluation: eager)
        lang(name: 'Algol 60' model: procedural evaluation: eager)
        lang(name: 'Erlang' model: messagePassing evaluation: eager)
        lang(name: 'Prolog' model: relational evaluation: eager)
        lang(name: 'Haskell' model: functional evaluation: 'lazy')
        lang(name: 'ML' model: functional evaluation: eager)
        lang(name: 'Miranda' model: functional evaluation: 'lazy')
        lang(name: 'Oz' model: multiple evaluation: eager) ]
SCRIPTS = [lang(name: 'Bash Shell' model: procedural evaluation: eager)
           lang(name: 'Python' model: objectOriented evaluation: eager)
           lang(name: 'Perl' model: procedural evaluation: eager)
           lang(name: 'Ruby' model: objectOriented evaluation: eager) ]

{Test {LazyLanguages PLDB}
 '==' [lang(name: 'Haskell' model: functional)
       lang(name: 'Miranda' model: functional)]}
{Test {LazyLanguages SCRIPTS}
 '==' nil}
{Test {LazyLanguages {Append SCRIPTS PLDB}}
 '==' [lang(name: 'Haskell' model: functional)
       lang(name: 'Miranda' model: functional)]}
```

Remember that you must use Oz's **for** loop with `collect:` in your solution

7. (20 points) Solve the same problem as problem 6 on the previous page, but instead of using **`for`** in your solution, use the Oz standard functions `Filter` and `Map`.

8. (20 points) [UseModels] Using Oz's built-in `Filter` and `Map` functions, write a function

   ```
   FilterThenMap: <fun {$ <List T> <fun {$ T}: Bool> <fun {$ T} S>}: <List S>>
   ```

   that takes a list `Ls`, a predicate `Pred`, and a function `Fun`, and returns the list that results from first filtering `Ls` using `Pred` and then mapping `Fun` over the result. The following are examples, that use the `Test` method from the homework.

   ```
   fun {IsOdd N} N mod 2 == 1 end
   fun {IsEven N} N mod 2 == 0 end
   fun {Double N} N * 2 end
   {Test {FilterThenMap nil IsOdd Double} '==' nil}
   {Test {FilterThenMap [1 2 3 4 5 4 1 3 7] IsOdd Double} '==' [2 6 10 2 6 14]}
   {Test {FilterThenMap [1 2 3 4 5 4 1 3 7] IsEven Double} '==' [4 8 8]}
   {Test {FilterThenMap [1 2 3 4 5 4 1 3 7] fun {$ X} X>4 end Double}
    '==' [10 14]}
   {Test {FilterThenMap [3 9 17 21 5] fun {$ X} X>4 end fun {$ Y} Y+3 end}
    '==' [12 20 24 8]}
   {Test {FilterThenMap [3 9 9 3 8 4 2] fun {$ _} false end fun {$ Y} Y+3 end}
    '==' nil}
   {Test {FilterThenMap [3 9 9 3 8 4 2] fun {$ _} true end fun {$ Y} Y+3 end}
    '==' [6 12 12 6 11 7 5]}
   {Test {FilterThenMap [3 9 9 3 8 4 2] fun {$ Y} Y<7 end fun {$ Z} a#Z end}
    '==' [a#3 a#3 a#4 a#2]}
   {Test {FilterThenMap [3 9 9 3 8 4 2] IsOdd IsOdd}
    '==' [true true true true]}
   ```