Spring, 2009 Name: ______________________________

COP 4020 — Programming Languages 1

Test on Declarative Programming Techniques

Special Directions for this Test

This test has 7 questions and pages numbered 1 through 8.
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 model (as in chapters 2–3 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. You can use the built-in functions in the Oz base environment like `Append`, `Member`, `Max`, `Filter`, `Map`, and `FoldR`.

For Grading

Question:	1	2	3	4	5	6	7	Total
Points:	10	10	15	15	10	20	20	100
Score:								

1. (10 points) [UseModels] Write an iterative function

   ```
   MyLength: <fun {$ <List T>}: <Int> >
   ```

   that takes a list `Lst` of values of some type T and returns the number of elements in `Lst`, that is, it returns the size of `Lst`.

   Your solution must have iterative behavior, and must be written using tail recursion. Don't use the built-in Oz function `Length`, don't use any higher-order functions, and don't use the Oz **for** loop syntax in your solution. (You are supposed to know what these directions mean.)

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

   ```
   \insert 'MyLength.oz'
   {Test {MyLength nil} '==' 0}
   {Test {MyLength [c]} '==' 1}
   {Test {MyLength [b a]} '==' 2}
   {Test {MyLength [b b a]} '==' 3}
   {Test {MyLength [a b b a]} '==' 4}
   {Test {MyLength [b d a c a b b a]} '==' 8}
   {Test {MyLength [um no no no no yes no yes no]} '==' 9}
   {Test {MyLength [4 0 7 5 5 5 1 2 1 2]} '==' 10}
   ```

2. (10 points) [UseModels] Write a function

   ```
   SquareAll: <fun {$ <List <Number> >}: <List <Number> >
   ```

   that takes a list of numbers, `Nums`, and produces a list containing the squares of each number in `Nums`, in the original order. The following are examples.

   ```
   \insert 'SquareAll.oz'
   {Test {SquareAll nil} '==' nil}
   {Test {SquareAll [9 8 7 6 5 4 3 2 1 0 10]}
         '==' [81 64 49 36 25 16 9 4 1 0 100]}
   {Test {SquareAll [10 3 10]} '==' [100 9 100]}
   {Test {SquareAll [11 22 2009]} '==' [121 484 4036081]}
   ```

3. (15 points) [UseModels] Using `Filter` and `Map`, write the function

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

that for some types `S` and `T` takes three arguments: `Ls`, which is a list of values of type `S`, `Pred`, which is a boolean-valued function of type `<fun {$ S}: Bool>`, and `F`, which is a function of type `<fun {$ S}: T>`. The function you are to write, `SelectAndMap`, selects elements `E` of `Ls` for which `{Pred E}` returns true, and returns a list of the result of applying `F` to those selected elements (preserving the original order). The following are examples.

```
\insert 'SelectAndMap.oz'
local
   fun {AlwaysTrue _} true end
   fun {Odd N} N mod 2 == 1 end
   fun {Even N} N mod 2 == 0 end
   fun {Double X} X*2 end
in
   {Test {SelectAndMap nil AlwaysTrue Double} '==' nil}
   {Test {SelectAndMap 4|5|6|7|nil AlwaysTrue Double} '==' 8|10|12|14|nil}
   {Test {SelectAndMap 4|5|6|7|nil Odd Double} '==' 10|14|nil}
   {Test {SelectAndMap 4|5|6|7|nil Even Double} '==' 8|12|nil}
   {Test {SelectAndMap 1|2|3|1|4|5|6|7|nil fun {$ X} X>3 end Double}
    '=='8|10|12|14|nil}
   {Test {SelectAndMap 1|2|3|1|4|5|6|7|nil fun {$ X} X>3 end fun {$ X} X*10 end}
    '==' 40|50|60|70|nil}
   {Test {SelectAndMap 1|2|3|1|4|5|6|7|nil fun {$ X} X =< 3 end fun {$ X} X end}
    '==' 1|2|3|1|nil}
   {Test {SelectAndMap 1|2|3|1|4|5|6|7|nil fun {$ _} false end fun {$ X} X end}
    '==' nil}
   {Test {SelectAndMap [o o p s l a] AlwaysTrue fun {$ X} [X] end}
                  '==' [[o] [o] [p] [s] [l] [a]]}
end
```

Your solution must use both `Filter` and `Map` (but you can also write additional helping functions if you wish).

4. (15 points) [UseModels] Write a function

```
Project: <fun {$ <List <Pair S T> > S} : <List T> >
```

that for some types `S` and `T`, takes an association list, `Pairs` (that is, a list of #-tuples of `S` and `T` elements), and an element of type `S`, `E`, and returns a list of all the elements of type T to which `E` is associated in `Pairs` (by being in the same pair). That is, whenever there is a pair *SVal#TVal* in `Pairs` and `E` == *SVal*, then *TVal* is in the result. The relative order of the values of type `T` that are in the result should be the same as their relative order in `Pairs`. The following are examples.

```
\insert 'Project.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'Project'}
{Test {Project [1#2 2#3 3#4 2#5 1#6] 2} '==' [3 5]}
{Test {Project [a#1 a#0 b#2 b#~1 c#3 c#4 d#4 d#5] a} '==' [1 0]}
{Test {Project [a#1 a#0 b#2 b#~1 c#3 c#4 d#4 d#5] b} '==' [2 ~1]}
{Test {Project [a#1 a#0 b#2 b#~1 c#3 c#4 d#4 d#5] b} '==' [2 ~1]}
{Test {Project [a#1 a#0 b#2 b#~1 c#3 c#4 d#4 d#5] c} '==' [3 4]}
{Test {Project [a#1 a#0 b#2 b#~1 c#3 c#4 d#4 d#5] z} '==' nil}
{Test {Project nil q} '==' nil}
{Test {Project [q#ok] q} '==' [ok]}
```

5. (10 points) [Concepts] [UseModels] Write a curried version of the function `Project`, from question 4 on the previous page. The function you are to write should be called `CurriedProject`. That is, write

```
CurriedProject: <fun {$ <List <Pair S T> >}: <fun {$ S}: <List T> > >
```

that for some types `S` and `T`, takes an association list, named `Pairs`, of type `<List <Pair S T> >`, and returns a function that takes an element, `E`, of type `S`, and returns a list of all the elements of type T to which `E` is associated in `Pairs` (by being in the same pair). To save time, just call `Project` in your answer, instead of writing out the code for `Project` again.

The following are examples.

```
\insert 'CurriedProject.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'CurriedProject'}
{Test {{CurriedProject [1#2 2#3 3#4 2#5 1#6]} 2} '==' [3 5]}
{Test {{CurriedProject [a#1 a#0 b#2 b#~1 c#3 c#4 d#4 d#5]} a} '==' [1 0]}
{Test {{CurriedProject [a#1 a#0 b#2 b#~1 c#3 c#4 d#4 d#5]} b} '==' [2 ~1]}
{Test {{CurriedProject [a#1 a#0 b#2 b#~1 c#3 c#4 d#4 d#5]} b} '==' [2 ~1]}
{Test {{CurriedProject [a#1 a#0 b#2 b#~1 c#3 c#4 d#4 d#5]} c} '==' [3 4]}
{Test {{CurriedProject [a#1 a#0 b#2 b#~1 c#3 c#4 d#4 d#5]} z} '==' nil}
{Test {{CurriedProject nil} q} '==' nil}
{Test {{CurriedProject [q#ok]} q} '==' [ok]}
```

Please write your answer below.

```
\insert 'Project.oz'  % so you can use Project from the previous problem
```

6. (20 points) [UseModels] This problem is about the following grammar for boolean expressions:

⟨Bexp⟩ ::=
    andExp(⟨Bexp⟩ ⟨Bexp⟩)
  | orExp(⟨Bexp⟩ ⟨Bexp⟩)
  | notExp(⟨Bexp⟩)
  | comp(⟨Comp⟩)
⟨Comp⟩ ::= equals(⟨Atom⟩ ⟨Atom⟩)
  | notequals(⟨Atom⟩ ⟨Atom⟩)

Write a function `Subst: <fun {$ <BExp> <Atom> <Atom>}: <BExp> >` that takes a ⟨BExp⟩ record, `E`, and two atoms `Old` and `New`, and returns a ⟨BExp⟩ that is just like `E`, except that in each ⟨Comp⟩ record, each ⟨Atom⟩ that is equal to `Old` is replaced by `New`. The following are examples using the `Test` function from the homework.

```
\insert 'Subst.oz'
{StartTesting 'Subst'}
{Test {Subst andExp(comp(notequals(o n))
                    comp(equals(a o)))
             o sym} '==' andExp(comp(notequals(sym n))
                                comp(equals(a sym)))}
{Test {Subst orExp(comp(equals(q r))
                   andExp(comp(notequals(o n)) comp(equals(r q))))
       r newvar}
       '==' orExp(comp(equals(q newvar))
                  andExp(comp(notequals(o n)) comp(equals(newvar q))))}
{Test {Subst notExp(comp(equals(x y))) x a} '==' notExp(comp(equals(a y)))}
{Test {Subst notExp(andExp(comp(equals(q oldr))
                           andExp(comp(notequals(o n)) comp(equals(oldr q)))))
       oldr newvar}
       '==' notExp(andExp(comp(equals(q newvar))
                          andExp(comp(notequals(o n)) comp(equals(newvar q)))))}
{Test {Subst comp(equals(x y)) y z} '==' comp(equals(x z))}
{Test {Subst comp(equals(x y)) x a} '==' comp(equals(a y))}
{Test {Subst comp(notequals(o n)) o sym} '==' comp(notequals(sym n))}
```

Be sure to follow the grammar!

7. (20 points) [UseModels] This problem is about the following grammar for "unit tests" for functions of type `<fun {$ S}: T>`, where `S` and `T` are arbibrary types.

```
⟨UnitTest S T⟩ ::=
     testcase(arg: <S> expect: <T>)
   | testsuite(<list <UnitTest S T> >)
```

Write a function `AllPass: <fun {$ <UnitTest S T> <fun {$ S}: T>}: Bool>` that takes a ⟨UnitTest S T⟩ record, `TestRec` and a function `F` (which takes arguments of type `S` and returns results of type `T`), and returns true when `F` passes each test in `TestRec`. `AllPass` returns false otherwise. A unit test record of the form `testcase(arg: X expect: E)` passes for a function `F` when `{F X} == E`. A unit test record of the form `testsuite([UT1 ... UTn])` passes for a function `F` when each unit test `UT1` . . . `UTn` passes for function `F`.

To simplify this problem, we will make the somewhat unrealistic assumption that the function given to `AllPass` never throws an exception and always terminates normally when called. The following are examples using the `Test` function from the homework.

```
\insert 'AllPass.oz'
{Test {AllPass testcase(arg: 1 expect: 2) fun {$ X} X+1 end} '==' true}
{Test {AllPass testsuite(nil) fun {$ X} X+1 end} '==' true}
{Test {AllPass testsuite(testcase(arg: 1 expect: 3)|nil) fun {$ X} X+2 end}
 '==' true}
{Test {AllPass testsuite([testcase(arg: 1 expect: 3) testsuite([testcase(arg: 4 expect: 6)])
                          testsuite(nil) testcase(arg: 10 expect:12)
                          testsuite([testcase(arg: 9 expect: 11) testsuite([testcase(arg: 0 expect: 2)])])])
                fun {$ X} X+2 end}
 '==' true}
% Some tests where AllPass returns false (below)
{Test {AllPass testcase(arg: 1 expect: 3) fun {$ X} X+10 end} '==' false}
{Test {AllPass testsuite([testcase(arg: 1 expect: 3)
                          testsuite([testcase(arg: 4 expect: 6)])])
                fun {$ _} 3 end}
 '==' false}
{Test {AllPass testsuite([testcase(arg: 1 expect: 3) testsuite([testcase(arg: 4 expect: 6)])
                          testsuite(nil) testcase(arg: 10 expect:12)
                          testsuite([testcase(arg: 9 expect: 11) testsuite([testcase(arg: 0 expect: 2)])])])
                fun {$ _} 3 end}
 '==' false}
```

Be sure to follow the grammar!