Spring, 2010 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 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 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 such as `Append`, `Not`, `Reverse`, `Length`, `Pow`, `Filter`, `Map`, and `FoldR`.

For Grading

Question:	1	2	3	4	5	6	7	Total
Points:	10	10	15	10	10	20	25	100
Score:								

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

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

   that takes a list `Digits` of integers, and returns the base 10 equivalent of the number represented by the list. That is, if `Digits` is [$N_{m-1} \ldots N_1\ N_0$], then the result is $N_{m-1} \times 10^{m-1} + \cdots N_1 \times 10^1 + N_0$. (You may assume that each element of `Digits` is an integer that is strictly less than 10.)

   Your solution must have iterative behavior, and must be written using tail recursion. 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 'DecimalValue.oz'
{Test {DecimalValue nil} '==' 0}
{Test {DecimalValue [2]} '==' 2}
{Test {DecimalValue [7 2]} '==' 72}
{Test {DecimalValue [0 2 0]} '==' 20}
{Test {DecimalValue [4 0 2 0]} '==' 4020}
{Test {DecimalValue [6 4 0 2 0]} '==' 64020}
{Test {DecimalValue [7 8 9 9 8 2 1]} '==' 7899821}
{Test {DecimalValue [9 8 7 6 5 4 3 2 1 0 1 2 3 4 5 6 7 8 9]}
 '==' 98765432101234567 89}
```

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

   `PrefixAll: <fun {$ <List <List T>> <List T>}: <List T>`

   that takes a list of lists of some type T, `LoL`, and a list of type T, `Prefix`, and returns a list that is just like `LoL` except that each element of `LoL` has `Prefix` appended to its front. (Hint, you can use Oz's built in `Append` function.)

   The following are examples.

```
\insert 'PrefixAll.oz'
{Test {PrefixAll nil [foo bar]} '==' nil}
{Test {PrefixAll [[bar baz] [belch] [bomb]] [foo bar]}
       '==' [[foo bar bar baz] [foo bar belch] [foo bar bomb]]}
{Test {PrefixAll [[7] [8] [9 10 25]] [0 0]}
      '==' [[0 0 7] [0 0 8] [0 0 9 10 25]]}
{Test {PrefixAll ["are lists of characters" "are lists"] "strings "}
    '==' ["strings are lists of characters" "strings are lists"]}
```

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

   `Partition: <fun {$ <List T> <fun {$ T}: Bool>}: <Pair <List T> <List T> >`

   that, for some type `T`, takes two arguments: `Ls`, which is a list of values of type `T`, and `Pred`, which is a Boolean-valued function of type `<fun {$ T}: Bool>`. The function you are to write, `Partition`, returns a pair of lists of the form `L1#L2`, in which the elements of `L1` are the elements of `Ls` which satisfy `Pred`, and the elements of `L2` are the elements of `Ls` that do not satisfy `Pred`. The relative order of the elements in `Ls` is preserved within each of the two lists in the pair, that is, within `L1` and `L2`. The following are examples.

```
\insert 'Partition.oz'
local % for testing
   fun {AlwaysTrue _} true end
   fun {Odd N} N mod 2 == 1 end
   fun {Even N} N mod 2 == 0 end
in
   {Test {Partition nil AlwaysTrue} '==' nil#nil}
   {Test {Partition [4 5 6 7] AlwaysTrue} '==' [4 5 6 7]#nil}
   {Test {Partition [4 5 6 7] Odd} '==' [5 7]#[4 6]}
   {Test {Partition [4 5 6 7] Even} '==' [4 6]#[5 7]}
   {Test {Partition [1 2 3 4 5 6 7 8 6 5] fun {$ X} X>3 end}
    '==' [4 5 6 7 8 6 5]#[1 2 3]}
   {Test {Partition [o o p s l a] fun {$ X} X == o end}
    '==' [o o]#[p s l a]}
   {Test {Partition [o o p s l a] fun {$ X} X == l orelse X == p end}
    '==' [p l]#[o o s a]}
end
```

   Your solution must use `Filter` (but you can also write additional helping functions if you wish)! Hint: you may want to use Oz's built in function `Not`.

4. (10 points) [Concepts] [UseModels] Write a curried version of the function `Partition`, from question 3 on the previous page. The function you are to write should be called `CurriedPartition`. That is, write

   ```
   CurriedPartition: <fun {$ <List T>} : <fun {$ <fun {$ T}: Bool>}}: <Pair <List T> <List T>>>>
   ```

   that, for some type `T`, takes an argument, `Ls`, which is a list of values of type `T`, and returns a function that takes as an argument, `Pred`, which is a Boolean-valued predicate of type `<fun {$ T}: Bool>`. For a call such as `{{CurriedPartition Ls} Pred}` the result is a pair of lists of the form `L1#L2`, in which the elements of `L1` are the elements of `Ls` which satisfy `Pred`, and the elements of `L2` are the elements of `Ls` that do not satisfy `Pred`. The relative order of the elements in `Ls` is preserved within each of the two lists in the pair, that is, within `L1` and `L2`. The following are examples. To save time, you can call `Partition` in your answer, instead of writing out the body of `Partition` again.

   The following are examples.

   ```
   \insert 'CurriedPartition.oz'
   local % for testing
      fun {AlwaysTrue _} true end
      fun {Odd N} N mod 2 == 1 end
      fun {Even N} N mod 2 == 0 end
   in
      {Test {{CurriedPartition nil} AlwaysTrue} '==' nil#nil}
      {Test {{CurriedPartition [4 5 6 7]} AlwaysTrue} '==' [4 5 6 7]#nil}
      {Test {{CurriedPartition [4 5 6 7]} Odd} '==' [5 7]#[4 6]}
      {Test {{CurriedPartition [4 5 6 7]} Even} '==' [4 6]#[5 7]}
      {Test {{CurriedPartition [1 2 3 4 5 6 7 8 6 5]} fun {$ X} X>3 end}
       '==' [4 5 6 7 8 6 5]#[1 2 3]}
      {Test {{CurriedPartition [o o p s l a]} fun {$ X} X == o end}
       '==' [o o]#[p s l a]}
      {Test {{CurriedPartition [o o p s l a]} fun {$ X} X == l orelse X == p end}
       '==' [p l]#[o o s a]}
   end
   ```

   Please write your answer below.

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

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

   ```
   AppendAll: <fun {$ <List <List T>>}: <List T>>
   ```

   that, for some type `T`, takes a list of lists of `T` elements, `LL`, and returns a list which has all the elements of `LL`, which are lists, appended together. The following are examples.

   ```
   \insert 'AppendAll.oz'
   {StartTesting 'AppendAll'}
   {Test {AppendAll nil} '==' nil}
   {Test {AppendAll [nil [3 4 5] nil [6 7 3] nil]} '==' [3 4 5 6 7 3]}
   {Test {AppendAll [["a good" "time"] ["was" "had"] ["by" "all"]]}
    '==' ["a good" "time" "was" "had" "by" "all"]}
   {Test {AppendAll [["far" "away"] nil ["and" "over"] ["the" "sea"]]}
          '==' ["far" "away" "and" "over" "the" "sea"]}
   ```

   Write your answer by filling in the blanks in the following solution.

   ```
   fun {AppendAll LL}
      {FoldR

         _____________________________________________

         _____________________________________________

         _____________________________________________

      }
   end
   ```

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

⟨WindowLayout⟩ ::=
`window(name:` ⟨Atom⟩ `width:` ⟨Number⟩ `height:` ⟨Number⟩`)`
| `horizontal(`⟨List WindowLayout⟩`)`
| `vertical(`⟨List WindowLayout⟩`)`

Write a function `Iconify: <fun {$ <WindowLayout>}: <WindowLayout> >` that takes a ⟨WindowLayout⟩, `WL`, and returns a ⟨WindowLayout⟩ that is just like `WL`, except that in each ⟨window⟩ record, the value of each `width` and `height` field is replaced by 1. The following are examples using the `Test` function from the homework.

```
\insert 'Iconify.oz'
{StartTesting 'Iconify'}
{Test {Iconify window(name: castle width: 1280 height: 740)}
 '==' window(name: castle width: 1 height: 1)}
{Test {Iconify horizontal([window(name: castle width: 1280 height: 740)
                           window(name: basketball width: 900 height: 900)])}
 '==' horizontal([window(name: castle width: 1 height: 1)
                  window(name: basketball width: 1 height: 1)])}
{Test {Iconify vertical([horizontal([window(name: castle width: 1280 height: 740)
                                     window(name: basketball width: 900 height: 900)])
                         vertical([window(name: csi width: 1000 height: 500)])])}
 '==' vertical([horizontal([window(name: castle width: 1 height: 1)
                            window(name: basketball width: 1 height: 1)])
                vertical([window(name: csi width: 1 height: 1)])])}
```

Be sure to follow the grammar!

7. (25 points) [UseModels] This problem works with the type "Music," as defined by the following grammar. (Note that all the ⟨Int⟩s that occur in a ⟨Music⟩ are guaranteed to be non-negative.)

   ⟨Music⟩ ::= `pitch(`⟨Int⟩`)` | `chord(`⟨List Music⟩`)` | `sequence(`⟨List Music⟩`)`

   Write a function

   `Retrograde: <fun {$ <Music>} : <Music> >`

   that takes a ⟨Music⟩, `Song`, and returns a ⟨Music⟩ that is like `Song` except that every `sequence` record occuring anywhere within `Song` has the list it contains reversed. (Hint: you can use `Reverse` in your code.) See the following examples.

```
\insert 'Retrograde.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'Retrograde'}
{Test {Retrograde pitch(3)} '==' pitch(3)}
{Test {Retrograde chord([pitch(5) pitch(3) pitch(1)])}
 '==' chord([pitch(5) pitch(3) pitch(1)])}
{Test {Retrograde chord([sequence([pitch(3) pitch(5)])
                         sequence([pitch(7) pitch(1)])])}
 '==' chord([sequence([pitch(5) pitch(3)])
             sequence([pitch(1) pitch(7)])])}
{Test {Retrograde sequence([pitch(2) pitch(8) pitch(3) pitch(9)])}
 '==' sequence([pitch(9) pitch(3) pitch(8) pitch(2)])}
{Test {Retrograde sequence([sequence([pitch(3) pitch(5)])
                            pitch(3)
                            chord([pitch(1) pitch(3) pitch(5)])
                            sequence([pitch(7) pitch(1)])])}
 '==' sequence([sequence([pitch(1) pitch(7)])
                chord([pitch(1) pitch(3) pitch(5)])
                pitch(3)
                sequence([pitch(5) pitch(3)])])}
{Test {Retrograde sequence([sequence([pitch(1) pitch(2) pitch(3)])
                            sequence([chord([pitch(5) pitch(9)])
                                      chord([pitch(3) pitch(5)])])])}
 '==' sequence([sequence([chord([pitch(3) pitch(5)])
                          chord([pitch(5) pitch(9)])])
                sequence([pitch(3) pitch(2) pitch(1)])])}
{Test {Retrograde sequence([sequence([sequence([pitch(3) pitch(5)])
                                      pitch(3)
                                      chord([pitch(1) pitch(3) pitch(5)])
                                      sequence([pitch(7) pitch(1)])])
                            sequence([sequence([pitch(1) pitch(2) pitch(3)])
                                      sequence([chord([pitch(5) pitch(9)])
                                                chord([pitch(3) pitch(5)])])])
                            pitch(5)])}
 '==' sequence([pitch(5)
                sequence([sequence([chord([pitch(3) pitch(5)])
                                    chord([pitch(5) pitch(9)])])
                          sequence([pitch(3) pitch(2) pitch(1)])])
                sequence([sequence([pitch(1) pitch(7)])
                          chord([pitch(1) pitch(3) pitch(5)])
                          pitch(3)
                          sequence([pitch(5) pitch(3)])])])}
```

   There is room for your answer on the next page

Please put your answer to the `Retrograde` problem below.