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 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

<table>
<thead>
<tr>
<th>Question:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points:</td>
<td>10</td>
<td>10</td>
<td>15</td>
<td>10</td>
<td>10</td>
<td>20</td>
<td>25</td>
<td>100</td>
</tr>
<tr>
<td>Score:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. (10 points) [UseModels] Write an iterative function

\[ \text{DecimalValue: } \langle \text{fun} \ (\text{List Int}) : \text{Int} \rangle \]

that takes a list \( \text{Digits} \) of integers, and returns the base 10 equivalent of the number represented by the list. That is, if \( \text{Digits} \) 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 \( \text{Digits} \) 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.

\[
\begin{align*}
\text{\{Test (DecimalValue nil) \ '==\ 0\}\}} \quad \text{\{Test (DecimalValue [2]) \ '==\ 2\}\}} \quad \text{\{Test (DecimalValue [7 2]) \ '==\ 72\}\}} \quad \text{\{Test (DecimalValue [0 2 0]) \ '==\ 20\}\}} \quad \text{\{Test (DecimalValue [4 0 2 0]) \ '==\ 4020\}\}} \quad \text{\{Test (DecimalValue [6 4 0 2 0]) \ '==\ 64020\}\}} \quad \text{\{Test (DecimalValue [7 8 9 8 2 1]) \ '==\ 7899821\}\}} \\
\text{\{Test (DecimalValue [9 8 7 6 5 4 3 2 1 0 1 2 3 4 5 6 7 8 9]) \ '==\ 9876543210123456789\}\}}
\end{align*}
\]
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 == 1 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

\[\text{CurriedPartition: } \text{<fun \{List T\}: \text{<fun \{\text{fun \{T\}: Bool\}}: \text{Pair \{List T\} \{List T\}}}>>\]

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 \(\text{<fun \{T\}: Bool}\). For a call such as \(\text{CurriedPartition Ls} \text{ Pred}\) 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 \text{ mod } 2 == 1 end
  \fun {Even N} N \text{ mod } 2 == 0 end

in
  \{Test \{\text{CurriedPartition nil} \text{ AlwaysTrue} \} \text{ '==' nil#nil}\}
  \{Test \{\text{CurriedPartition \{4 5 6 7\}} \text{ AlwaysTrue}\} \text{ '==' \{4 5 6 7\}#nil}\}
  \{Test \{\text{CurriedPartition \{4 5 6 7\}} \text{ Odd}\} \text{ '==' [5 7]#\{4 6\}}\}
  \{Test \{\text{CurriedPartition \{4 5 6 7\}} \text{ Even}\} \text{ '==' \{4 6\}#\{5 7\}}\}
  \{Test \{\text{CurriedPartition \{1 2 3 4 5 6 7 8 6 5\}} \text{ fun \{X\} \text{ X}>3 end}\}
     \text{ '==' \{4 5 6 7 8 6 5\}#\{1 2 3\}}\}
  \{Test \{\text{CurriedPartition \{o o s l a\}} \text{ fun \{X\} \text{ X} == o end}\}
     \text{ '==' \{o o\}#\{p s l a\}}\}
  \{Test \{\text{CurriedPartition \{o o s l a\}} \text{ fun \{X\} \text{ X} == l \text{ orelse X} == p end}\}
     \text{ '==' \{p l\}#\{o o s a\}}\}

end

Please write your answer below.

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

\[
\text{WindowLayout} ::= \\
\text{window(name: } \langle \text{Atom} \rangle \text{ width: } \langle \text{Number} \rangle \text{ height: } \langle \text{Number} \rangle \rangle \\
| \text{horizontal}(\langle \text{List \ WindowLayout} \rangle) \\
| \text{vertical}(\langle \text{List \ WindowLayout} \rangle)
\]

Write a function Iconify: \(\text{fun} \ \{ \text{WindowLayout} \} : \langle \text{WindowLayout} \rangle\) that takes a \(\langle \text{WindowLayout} \rangle\), WL, and returns a \(\langle \text{WindowLayout} \rangle\) that is just like WL, except that in each \(\langle \text{window} \rangle\) record, the value of each width and height field is replaced by 1. The following are examples using the Test function from the homework.

\begin{verbatim}
\[
\text{Test \{Iconify window(name: castle width: 1280 height: 740)}
\text{'==' window(name: castle width: 1 height: 1)}
\text{\}}
\text{Test \{Iconify horizontal([window(name: castle width: 1280 height: 740) \text{ window(name: basketball width: 900 height: 900)})]'==' horizontal([window(name: castle width: 1 height: 1) \text{ window(name: basketball width: 1 height: 1)})\)}}
\text{Test \{Iconify vertical([horizontal([window(name: castle width: 1280 height: 740) \text{ window(name: basketball width: 900 height: 900)}) \text{ vertical([window(name: csi width: 1000 height: 500)})]'==' vertical([horizontal([window(name: castle width: 1 height: 1) \text{ window(name: basketball width: 1 height: 1)}) \text{ vertical([window(name: csi width: 1 height: 1)])}\)}}
\end{verbatim}

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.)

\( ⟨\text{Music}⟩ ::= \text{pitch}(⟨\text{Int}⟩) \mid \text{chord}(⟨\text{List Music}⟩) \mid \text{sequence}(⟨\text{List Music}⟩) \)

Write a function

Retrograde: \(<\text{fun} ⟨\langle\text{Music}⟩⟩ : ⟨\text{Music}⟩⟩>

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

\{StartTesting 'Retrograde'}
\{Test {Retrograde pitch(3)} '==' pitch(3)}
\{Test {Retrograde chord([pitch(5) pitch(3) pitch(1)])} '==' chord([pitch(5) pitch(3) pitch(1)])}...

There is room for your answer on the next page
Please put your answer to the Retrograde problem below.