Spring, 2009

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 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

<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>15</td>
<td>10</td>
<td>20</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>Score:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
1. (10 points) [UseModels] Write an iterative function

\[ \text{MyLength}: \textbf{fun} \{ \text{List} T\} : \text{Int} > \]

that takes a list \text{Lst} of values of some type \( T \) and returns the number of elements in \text{Lst}, that is, it returns the size of \text{Lst}.

Your solution must have iterative behavior, and must be written using tail recursion. Don’t use the built-in Oz function \text{Length}, don’t use any higher-order functions, and don’t use the Oz \text{for} loop syntax in your solution.

(You are supposed to know what these directions mean.)

The following are examples, that use the \text{Test} procedure from the homework.

\[ \text{\texttt{\{Test\ \{MyLength\ nil\} \textquote{==} 0\}}} \]
\[ \text{\texttt{\{Test\ \{MyLength\ [c]\} \textquote{==} 1\}}} \]
\[ \text{\texttt{\{Test\ \{MyLength\ [b\ a]\} \textquote{==} 2\}}} \]
\[ \text{\texttt{\{Test\ \{MyLength\ [b\ b\ a]\} \textquote{==} 3\}}} \]
\[ \text{\texttt{\{Test\ \{MyLength\ [a\ b\ b\ a]\} \textquote{==} 4\}}} \]
\[ \text{\texttt{\{Test\ \{MyLength\ [b\ d\ a\ c\ a\ b\ a]\} \textquote{==} 8\}}} \]
\[ \text{\texttt{\{Test\ \{MyLength\ [um\ no\ no\ no\ no\ yes\ no\ yes\ no]\} \textquote{==} 9\}}} \]
\[ \text{\texttt{\{Test\ \{MyLength\ [4\ 0\ 7\ 5\ 5\ 1\ 2\ 1\ 2]\} \textquote{==} 10\}}} \]
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: \( \text{<fun} \{\text{$<$List S$>$ \text{<fun} \{\text{$S$}: \text{Bool}\} \text{<fun} \{\text{$S$}: \text{T}\}: \text{List T}\}\} > \)

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

\( \text{\{Test {SelectAndMap nil AlwaysTrue Double} '==' nil}\} \)
\( \text{\{Test {SelectAndMap 4|5|6|7|nil AlwaysTrue Double} '==' 8|10|12|14|nil}\} \)
\( \text{\{Test {SelectAndMap 4|5|6|7|nil Odd Double} '==' 10|14|nil}\} \)
\( \text{\{Test {SelectAndMap 4|5|6|7|nil Even Double} '==' 8|12|nil}\} \)
\( \text{\{Test {SelectAndMap 1|2|3|1|4|5|6|7|nil fun \{\text{$X$} X>3 end} Double\} '==' 8|10|12|14|nil}\} \)
\( \text{\{Test {SelectAndMap 1|2|3|1|4|5|6|7|nil fun \{\text{$X$} X>3 end fun \{\text{$X$} X*10 end\} '==' 40|50|60|70|nil}\} \)
\( \text{\{Test {SelectAndMap 1|2|3|1|4|5|6|7|nil fun \{\text{$X$} X =< 3 end fun \{\text{$X$} X end\} '==' 1|2|3|1|nil}\} \)
\( \text{\{Test {SelectAndMap 1|2|3|1|4|5|6|7|nil fun \{\_\} false end fun \{\text{$X$} X end\} '==' nil\} \)
\( \text{\{Test {SelectAndMap [o o p s l a] AlwaysTrue fun \{\text{$X$} \text{[X]} end\} '==' [[o] [o] [p] [s] [l] [a]]\} \)

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: \texttt{\textbf{fun} \{\texttt{\langle List \langle Pair S T \rangle \rangle S} \rightarrow \texttt{\langle List T \rangle} \}}

that for some types $S$ and $T$, takes an association list, $\text{Pairs}$ (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 $\text{Pairs}$ (by being in the same pair). That is, whenever there is a pair $SVal\#TVal$ in $\text{Pairs}$ 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 $\text{Pairs}$. The following are examples.

\begin{verbatim}
\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]}
\end{verbatim}
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.

```oz
\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.

```oz
\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:

\[
\begin{align*}
    \langle \text{Bexp} \rangle & ::= \\
    & \text{andExp(}\langle \text{Bexp} \rangle \langle \text{Bexp} \rangle) \\
    & \text{orExp(}\langle \text{Bexp} \rangle \langle \text{Bexp} \rangle) \\
    & \text{notExp(}\langle \text{Bexp} \rangle) \\
    & \text{comp(}\langle \text{Comp} \rangle) \\
    \langle \text{Comp} \rangle & ::= \text{equals(}\langle \text{Atom} \rangle \langle \text{Atom} \rangle) \\
    & \text{notequals(}\langle \text{Atom} \rangle \langle \text{Atom} \rangle)
\end{align*}
\]

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

\begin{verbatim}
>\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))\}
\end{verbatim}

Be sure to follow the grammar!
7. (20 points) [UseModels] This problem is about the following grammar for “unit tests” for functions of type
\[ \langle \text{fun} \{ S \} : T \rangle, \] where \( S \) and \( T \) are arbitrary types.

\[
\langle \text{UnitTest} \ S \ T \rangle ::=
\quad \text{testcase(arg: <S> expect: <T>)}
\quad \mid \text{testsuite(<list <UnitTest S T > >)}
\]

Write a function \( \text{AllPass} : \langle \text{fun} \{ S \} : T \rangle \rightarrow \text{Bool} \) that takes a \( \langle \text{UnitTest} \ S \ T \rangle \) record, \( \text{TestRec} \) 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 \( \text{TestRec} \). \( \text{AllPass} \) returns false otherwise. A unit test record of the form
\[
\text{testcase(arg: X expect: E)}
\]
passes for a function \( F \) when \( \{F X\} == E \). A unit test record of the form
\[
\text{testsuite([UT1 ... UTn])}
\]
passes for a function \( F \) when each unit test \( UT1 \ldots UTn \) passes for function \( F \).

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

\[
\begin{align*}
\text{AllPass}\ \text{testcase(arg: 1 expect: 2) fun \{ X \} X+1 end} & \quad '==' \quad \text{true} \\
\text{AllPass}\ \text{testsuite(nil) fun \{ X \} X+1 end} & \quad '==' \quad \text{true} \\
\text{AllPass}\ \text{testsuite(testcase(arg: 1 expect: 3)} \mid \text{nil} \mid \text{fun \{ X \} X+2 end} & \quad '==' \quad \text{true} \\
\text{AllPass}\ \text{testsuite([testcase(arg: 1 expect: 3) testsuite([testcase(arg: 4 expect: 6)]) testsuite(nil) testcase(arg: 9 expect: 11) testsuite([testcase(arg: 0 expect: 2)])]) fun \{ X \} X+2 end} & \quad '==' \quad \text{true} \\
\end{align*}
\]

% Some tests where \( \text{AllPass} \) returns false (below)
\[
\begin{align*}
\text{AllPass}\ \text{testcase(arg: 1 expect: 3) fun \{ X \} X+10 end} & \quad '==' \quad \text{false} \\
\text{AllPass}\ \text{testsuite([testcase(arg: 1 expect: 3) testsuite([testcase(arg: 4 expect: 6)])]} \\
\text{fun \{ X \} X+2 end} & \quad '==' \quad \text{false} \\
\text{AllPass}\ \text{testsuite([testcase(arg: 1 expect: 3) testsuite([testcase(arg: 4 expect: 6)])]} \\
\text{fun \{ X \} X+2 end} & \quad '==' \quad \text{false} \\
\end{align*}
\]

Be sure to follow the grammar!