Fall, 2006 Name: ____________________

Com S 541 — Programming Languages 1

# Test on the Declarative Model

## Special Directions for this Test

This test has 12 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 we have seen in chapters 2–3 of our textbook. But unless specifically directed, you should not use imperative features (such as cells).

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.

## For Grading

| Problem | Points | Score |
|---|---|---|
| 1 | 5 | |
| 2 | 10 | |
| 3 | 10 | |
| 4 | 5 | |
| 5 | 5 | |
| 6 | 5 | |
| 7 | 5 | |
| 8 | 10 | |
| 9 | 5 | |
| 10 | 10 | |
| 11 | 10 | |
| 12 | 15 | |

1. (5 points) Briefly, what makes Oz useful for studying programming languages, compared to Java, C, or C++?

2. (10 points) Using only features in the declarative model, is it possible to make cyclic data structures? That is, can one make data structures that contain references to themselves (or parts of themselves)? If so, then give an example, otherwise explain why this is not possible.

3. (10 points) In Oz one can write the body of a procedure as a declaration followed by **`in`**, followed by a statement. This form, $D$ **`in`** $S$, is what the language reference materials call an ⟨in statement⟩. Give a brief, but precise and general semantics for an ⟨in statement⟩.

4. (5 points) Briefly, why does a programming language need a built-in mechanism for throwing and catching exceptions?

5. (5 points) Briefly, describe one way that dataflow variables are useful in writing sequential, declarative Oz programs.

6. (5 points) Briefly, what is the relationship between modules and records in Oz?

7. (5 points) Explain how closures are used in the operational semantics of Oz to enforce static scoping.

8. (10 points) Desugar the following Oz expression into the kernel language of the declarative model.

```
local
   Answer
   G = fun {$ X} X * X end
in
   Answer = {G 3}
end
```

9. (5 points) Write a function `ScalarMultiply`: ⟨**fun** {$ ⟨List ⟨Int⟩⟩ ⟨Int⟩}: ⟨List ⟨Int⟩⟩}⟩ that takes a list of Ints $L$ and an Int $N$ and returns a list with the same length as $L$ but whose $i^{th}$ element is the product of the $i^{th}$ element of $L$ and $N$. The following examples are written using the `Test` procedure from homework 4.

```
{Test {ScalarMultiply nil 3} '=' nil}
{Test {ScalarMultiply 7|nil 3} '=' 21|nil}
{Test {ScalarMultiply [1 2 3 4 2 1] 10} '=' [10 20 30 40 20 10]}
{Test {ScalarMultiply [2 3 4 2 1] 10} '=' [20 30 40 20 10]}
```

10. (10 points) Consider the following grammar for simplified type expressions.

```
⟨TExp⟩ ::= baseT( ⟨Atom⟩ )
    | productT( left:⟨TExp⟩ right:⟨TExp⟩ )
    | listT( ⟨TExp⟩ )
```

where ⟨Atom⟩ is an Oz symbol, such as `int` or `bool`.

Write a function `Depth`: ⟨**fun** {$ ⟨TExp⟩}: ⟨Int⟩⟩ that takes a ⟨TExp⟩ `T` and an ⟨Atom⟩ `A` and returns an ⟨Int⟩ that tells the maximum depth of nesting of a `baseT` record within `T`. The following examples are written using the `Test` procedure from homework 4.

```
% This is file DepthTest.oz
{Test {Depth baseT(int)} '==>' 0}
{Test {Depth baseT(float)} '==>' 0}
{Test {Depth productT(left: baseT(int) right: baseT(float))} '==>' 1}
{Test {Depth productT(left: listT(baseT(int)) right: baseT(float))}
      '==>' 2}
{Test {Depth productT(left: baseT(int)
                      right: productT(left: baseT(float)
                                      right: baseT(bool)))}
      '==>' 2}
{Test {Depth listT(productT(left: baseT(int) right: baseT(float)))}
      '==>' 2}
{Test {Depth listT(productT(left: listT(baseT(int))
                            right: baseT(float)))}
      '==>' 3}
{Test {Depth listT(productT(left: listT(baseT(int))
                            right: listT(listT(baseT(float)))))}
      '==>' 4}
```

11. (10 points) Consider again the grammar for simplified type expressions from the previous problem. Write a function `Subst`: ⟨**fun** {$ ⟨TExp⟩ ⟨Atom⟩ ⟨Atom⟩}: ⟨TExp⟩⟩ that takes a ⟨TExp⟩ `T`, an ⟨Atom⟩ `Old`, and an atom ⟨Atom⟩ `New`, and returns a ⟨TExp⟩ that is just like `T` except that it has all occurrences of `Old` replaced by `New`. The following examples are written using the `Test` procedure from homework 4.

```
% This is file SubstTest.oz
{Test {Subst baseT(int) int bool} '=' baseT(bool)}
{Test {Subst baseT(int) foo bar} '=' baseT(int)}
{Test {Subst baseT(float) float double} '=' baseT(double)}
{Test {Subst productT(left: baseT(int) right: baseT(float)) int bool}
         '=' productT(left: baseT(bool) right: baseT(float))}
{Test {Subst listT(productT(left: baseT(int) right: baseT(float)))
             int bool}
         '=' listT(productT(left: baseT(bool) right: baseT(float)))}
{Test
 {Subst
     productT(left: listT(productT(left: baseT(int) right: baseT(float)))
           right: productT(left: baseT(int) right: listT(baseT(int))))
     int bool}
 '=' productT(left: listT(productT(left: baseT(bool) right: baseT(float)))
           right: productT(left: baseT(bool) right: listT(baseT(bool))))}
```

12. (15 points) Consider again the previous two problems. Write a function `FoldTExp`, which can be called with a ⟨TExp⟩, and three functions as arguments, as in `{FoldTExp TE BF PF LF}`. The function `FoldTExp` should generalize the previous two problems in the sense that the following are examples.

```
declare
fun {Depth TE}
   {FoldTExp
    TE
    fun {$ _} 0 end
    fun {$ LD RD} 1 + {Max LD RD} end
    fun {$ D} 1 + D end}
end
\insert 'DepthTest.oz' % test cases for the Depth problem above

fun {Subst TE Old New}
   {FoldTExp
    TE
    fun {$ A} baseT(if A == Old then New else A end) end
    fun {$ TL RT} productT(left: TL right: RT) end
    fun {$ LT} listT(LT) end}
end
\insert 'SubstTest.oz' % test cases for the Subst problem above

fun {Count TE A}
   {FoldTExp
    TE
    fun {$ B} if B == A then 1 else 0 end end
    fun {$ LC RC} LC + RC end
    fun {$ C} C end}
end
{Test {Count baseT(int) int} '==>' 1}
{Test {Count baseT(float) int} '==>' 0}
{Test {Count listT(baseT(int)) int} '==>' 1}
{Test {Count productT(left: baseT(bool) right: baseT(bool)) bool}
      '==>' 2}
{Test {Count productT(left: productT(left: baseT(bool)
                                     right: listT(baseT(bool)))
                      right: listT(baseT(bool))) bool}
      '==>' 3}
```