Homework 2: Declarative Computation Model

See Webcourses and the syllabus for due dates.
In this homework you will learn about the declarative computation model [Concepts], including the concepts of free and bound identifier occurrences, linguistic abstractions, syntactic sugars, and also about the extension of the declarative model to exception handling. You'll also see how the declarative computation model relates to C, C++, Java, or your favorite programming language [MapToLanguages].
Answers to English questions should be in your own words; don't just quote text from the textbook.
Code for programming problems should be written in Oz's declarative model, so do not use either cells or cell assignment in your Oz solutions. (Furthermore, note that the declarative model does *not* include the primitive `IsDet` or the library function `IsFree`; thus you are also prohibited from using either of these functions in your solutions.)
You should use helping functions whenever you find that useful. Unless we specifically say how you are to solve a problem, feel free to use any functions that are compatible with the declarative model from the Oz library (base environment), especially functions like `Map` and `FoldR`.
For all Oz programing exercises, you must run your code using the Mozart/Oz system. You can find all the tests we provide in a zip file, which you can download from problem 1's assignment on Webcourses.
If the tests don't pass, please try to say why they don't pass, as this enhances communication and makes commenting on the code easier and more specific to your problem.
What to Turn In: Turn in (on Webcourses) your code and output of your testing for each problem that requires code. Please upload code as a plain (text) file with the name given in the problem or testing file and with the suffix `.oz`. Please upload test output and English answers by pasting them into the answer box in the problem's "assignment" on Webcourses. If you have a mix of code and English, use a text file with a `.oz` file suffix, and put English answers in the answer box. (In any case, don't put spaces or tabs in your file names!)
Your code should compile with Oz, if it doesn't you probably should keep working on it. If you don't have time, at least tell us that you didn't get it to compile.
Email the staff with your code file if you need help getting it to compile.
For background, you should read Chapter 2 of the textbook [VH04]. But you may also want to refer to the reference and tutorial material on the Mozart/Oz web site. See also the course resources page.

Reading Problems

The problems in this section are intended to get you to read the textbook, ideally in advance of class meetings.

Read chapter 2, through section 2.1 of the textbook [VH04] and answer the following questions.

1. [Concepts] [MapToLanguages] A **`for`** loop in Java, C, C++, and C# is a linguistic abstraction of a **`while`** loop. In Java, **`interface`**s are also linguistic abstractions of abstract classes, and **`switch`** statements, are linguistic abstractions of **`if`** statements. Give another, different example of a linguistic abstraction in Java, C, C++, or C# by:
   (a) (2 points) saying which of these languages you are describing,
   (b) (3 points) naming a linguistic abstraction in that language, and
   (c) (5 points) explain briefly how to desugar that in the programming language by giving a schematic example. For example, a Java **`for`** loop of the form **`for`** `(`D`;` B`;` E`)` S is desugared into a **`while`** loop of the form `{` D`;` **`while`** `(`B`)` `{`S E`}` `}`, where D is an aribtrary declaration, B is an arbitrary Boolean-valued expression, E is an arbitrary expression, and S is an arbitrary statement (and we interpret these meta-variables to mean the same thing on both sides of the translation), provided that S does not contain any **`break`** or **`continue`** statements.

   Remember to paste your answer into the answerbox on webcourses.

Read through section 2.2 of the textbook and answer the following questions.

2. [Concepts]
   (a) (4 points) What is a partial value?
   (b) (3 points) What happens in Oz when a program executes a statement such as `X = Z` but both `X` and `Z` are undetermined (i.e., unbound) dataflow variables?

Read through section 2.3 of the textbook and answer the following questions.

3. (13 points) [MapToLanguages] Consider the following code in Oz

   ```
   40 \= X andthen 20 =< Y orelse {Pred Z}
   ```

   Assume that `X`, `Y`, `Pred`, and `Z` are have determined values and that `Pred` is a function that returns a Bool.

   How would you translate this code into C, C++, C#, or Java? (Note all these languages essentially have the same syntax for expressions.)

Read through section 2.4 of the textbook and answer the following questions.

4. [Concepts] This question is about the subtle but important difference between the confusingly similar terms "bound variable identifier occurrence" and "bound store variable."

   Consider the Oz program in Figure 1.

```
local X in
   local Y in
      X = 99
      Res = X+Y  % line 4
   end
end
```

Figure 1: Oz program for question 4.

   (a) (2 points) On line 4 of Figure 1, is the occurrence of the variable identifier `Y` a bound occurrence of that variable identifier, or is it a free occurrence?
   (b) (2 points) When starting to execute line 4 of Figure 1, will the store variable that `Y` denotes be a bound store variable or will it be undetermined?
   (c) (2 points) On line 4 of Figure 1, is the occurrence of the variable identifier `X` a bound occurrence of that variable identifier, or is it a free occurrence?
   (d) (2 points) When starting to execute line 4 of Figure 1, will the store variable that `X` denotes be a bound store variable or will it be undetermined?
   (e) (2 points) On line 4 of Figure 1, is the occurrence of the variable identifier `Res` a bound occurrence of that variable identifier, or is it a free occurrence?
   (f) (2 points) Suppose that line 4 in Figure 1 were to finish execution (e.g., suppose that another thread unified `Y` with 86). In that situation, after successfully executing line 4 of Figure 1, would the store variable that `Res` denotes be determined or undetermined?
   (g) (5 points) Must a bound occurrence of a variable identifier always denote a store variable that is determined (has a determined value) at runtime?

5. [Concepts] [MapToLanguages]

   (a) (12 points) Fill in the following table with the equivalent features in your favorite programming language, which go in the third column. The equivalent should have syntax in your favorite language, and execute in a way that is equivalent to the Oz statement in the same row. In the table, `S`, `S1`, and `S2` represent arbitrary statements (in both languages), thus the Oz code `S1 S2` in the second row means a sequence of two statements. Similarly, `X`, `Y1`, `Y2`, and `Y3` are arbitrary variable identifiers in both languages. Finally, `V` is an arbitrary value expression. You should use these same conventions in your answer; any other notation needed for your answer should be clearly explained.

   But sure to put the name of your language in the top row! Make notes in English about anything you had to assume in the translation, especially if that makes your translation less than perfectly general.

	Oz Feature	Equivalent in ____
1	**`skip`**	
2	`S1 S2`	
3	**`local`** `X` **`in`** `S1` **`end`**	
4	`X = V`	
5	**`if`** `X` **`then`** `S1` **`else`** `S2` **`end`**	
6	`{X Y1 Y2 Y3}`	

   (b) (3 points) Are there any problems translating uses of Oz's dataflow variables (as in line 3 of the table above) into your favorite language? If so describe these problems; if not, then briefly describe how to do the translation in general.

   (c) (3 points) Are there any problems translating Oz's **`case`** statement into your favorite language? If so briefly describe them; if not, then briefly describe how to do the translation.

   Remember to paste your answer into the answerbox on webcourses.

Read through section 2.5 of the textbook and answer the following questions.

6. [Concepts] [MapToLanguages]

   (a) (5 points) Suppose you are programming in a language (like C, C++, or Java) in which the compiler does not implement the "last call optimization" that is described in the text. In such a language should you use recursion to write code that may execute an unbounded number of times? Briefly explain.

   (b) (2 points) Does Oz have garbage collection like Java and C#?

Read through section 2.6 of the textbook and answer the following questions.

Before starting on this and other problems that ask you to desugar into the kernel language, you should try the ungraded quiz on desugaring in Webcourses.

7. (15 points) [Concepts] Desugar the following into the declarative kernel language. That is, write a program in the declarative kernel language (see tables 2.1 and 2.2 of the textbook [VH04]) that does the same thing as the following code. (Note, you don't have to desugar the comments, and we allow comments in the kernel syntax.)

```
% $Id: ToTranslate.oz,v 1.1 2011/09/12 00:21:45 leavens Exp leavens $
local Res in
   Res = [40 20]
   % {Browse Res}
end
```

   Hint: Your desugared code can be typed into a `.oz` file and you can use Oz to check that it is syntactically legal; it should run and produce the same value in `Res` as the code above. But Oz will not check that the code conforms to the declarative kernel's syntax; you have to do that yourself.

   For this problem, as with all Oz coding problems, upload a `.oz` file containing your code for the desugaring.

Read through section 2.7 of the textbook and answer the following questions.

8. (10 points) [Concepts] [UseModels] Figure 2 gives some of the code for a procedure

   `ExpectException: <proc {$ <fun {$ }: <Value>>}>`

   that takes a zero-argument function, `F0`, calls it with no arguments, discards the result of that call, and either

   - calls the procedure `NotifyAbout` if the call to `F0` did *not* raise an exception, or
   - calls the procedure `ReportAbout` if the call to `F0` raised an exception.

   You task is to fill in the missing code in Figure 2 so that it has this behavior.

   To explain what `ExpectException` does further, consider a call such as `{ExpectException G}`. Such a call would be used in testing, in conjunction with an environment (like our TestingNoStop.oz file) that defines the procedures `NotifyAbout` and `ReportAbout`.[1] When `ExpectException G` is called, it must call `G` with zero arguments (as the first code fragment in Figure 2 does). If that call does *not* raise an exception, then `NotifyAbout` must be called (and `ReportAbout` must not be called). However, if the call to `G` does raise an exception, then `ReportAbout` must be called (and `NotifyAbout` must not be called).

```
% $Id: ExpectException.oz,v 1.1 2011/09/12 17:34:53 leavens Exp $
\insert 'TestingNoStop.oz' % defines NotifyAbout and ReportAbout
declare
% ExpectException: <proc {$ <fun {$ }: <Value>>}>
proc {ExpectException F0}


      local _ = {F0} % ignore the result of calling function F0 with no args
      in
         {NotifyAbout 'Expected exception not raised!'}
      end


      {ReportAbout 'Caught exception '#{Value.toVirtualString X 5 5}#' as expected'}


end
```

Figure 2: Template for code for the procedure `ExpectException`, found in the file `ExpectException.oz`. You are to fill in the missing code (where there is empty space).

There is testing code in Figure 3 on the following page for testing the procedure `ExpectException`. Note, however, that the testing in that file contains 2 expected failures (which are necessary to test the testing code).

[1] The idea is that `NotifyAbout` is used to tell the user about test failures, and `ReportAbout` is used to give messages to the user about tests that do not fail. See the code in our `TestingNoStop.oz` file for details.

```
% $Id: ExpectExceptionTest.oz,v 1.1 2011/09/12 00:21:45 leavens Exp leavens $
\insert 'ExpectException.oz'
declare
fun {ThrowsSilly N}
   raise sillyException(N) end
end
fun {ThrowX ExceptionObject}
   raise ExceptionObject end
end
fun {Add2 N}
   N+2
end
{StartTesting 'ExpectExceptionTest $Revision: 1.1 $'}

{StartTesting 'exceptions not raised, failure messages in this part are needed'}
{ExpectException fun {$ } {Add2 3} end} % fails to raise exception
{ExpectException fun {$ } relax end} % fails to raise exception
if {GetFailures} \= 2 % these were expected
then
   raise 'testing missed notifying about 2 failures' end
else
   {ReportAbout 'expect to see 2 failures in the \'Finished with...\' message below...'}
end
{DoneTesting}  % should show 2, also resets the counter

{StartTesting 'testing catching of exceptions raised as expected'}
{ExpectException fun {$ } {ThrowsSilly 3} end}
{ExpectException fun {$ } {ThrowX oops} end}
{ExpectException fun {$ } raise expectedEx end end}
{DoneTesting}
```

Figure 3: Code for testing the procedure `ExpectException`. Note that the output from the middle section will show 2 failures, but those are expected, as shown in the messages.

Read through sections 2.8.2 and 2.8.3 of the textbook and answer the following questions.

9. [Concepts]

   (a) (4 points) In Oz, assuming that `M` and `C` are undetermined, does

   ```
   shoe(make: nike model: 'Free Run 2+' color: gold)
   ```

   unify with the following?

   ```
   shoe(make: M color: C)
   ```

   Answer (i) "yes" or "no" and (ii) either give a brief reason or describe an experiment with Oz that confirms your answer.

   (b) (2 points) Which language has dynamic type checking: Oz or Java?

Regular Problems

10. (20 points) [MapToLanguages] Write code in your favorite programming language (other than Oz, for example, in C, C++, C#, or Java) that implements immutable lists of items of an arbitrary type. Your answer should (a) say what language your code is written in, (b) and have operations to do all of the following:

    - Create an empty list (like `nil` in Oz). This might be a function or a constructor, depending on your language.
    - Create a new list from an item and an existing list, where the item is added to front of the new list (like `Item|Lst` in Oz). This might be a function or a constructor, depending on your language.
    - A function that tests whether a list given to it as an argument is empty, returning a Boolean result. This would be like the following in Oz:

    ```
    fun {IsEmpty Ls}
       case Ls of
          _|_ then false
       else true
       end
    end
    ```

    - A function that returns the head of a non-empty list. This would be like the following function in Oz:

    ```
    fun {Head Ls}  % assume Ls is not empty
       case Ls of
         H|_ then H
       end
    end
    ```

    - A function that returns the tail of a non-empty list. This would be like the following function in Oz:

    ```
    fun {Tail Ls}  % assume Ls is not empty
       case Ls of
         _|T then T
       end
    end
    ```

Your code must not limit the length of a list.

It is okay to just describe a built-in type in your language that has all of these operations, as long as you describe the names of those operations.

Test your code, including writing and testing a function that computes the length of a list.

Hand in the code in your favorite language, and its testing, along with the output of your testing, as separate attachments in webcourses. In the answer box, put a short description giving (a) the name of the language and an overview of the files you have uploaded for part (b) (if any).

The following problems relate to the textbook [VH04, section 2.9].

11. [Concepts] This is a problem about free and bound identifier occurrences. See the end of section 2.4.3 of the textbook for a definition of free and bound identifier occurrences.

    You may also want to do the ungraded quiz on free and bound identifiers in Webcourses before starting this.

    Consider the kernel language statement shown in Figure 4. (Note that there is no **declare** form in the kernel language, so you should not imagine one in the figure.)

```
FordRoad = proc {$ Kind Name R}
              local Ford in
                 Ford = ' ford '
                 R = '#'(1: Name 2: Ford 3: Kind)
              end
           end
Curry = proc {$ F R}
           R = proc {$ X R1}
                  R1 = proc {$ Y R2}
                          {F X Y R2}
                       end
               end
        end
local Res in
   local Ignored in
      local CFR in
         {Curry FordRoad CFR}
         local CFRoad in
            local Road in
               Road = road
               {CFR Road CFRoad}
               local Henry in
                  Henry = henry
                  {CFRoad Henry Res}
               end
            end
         end
      end
   end
```

Figure 4: Kernel language statement for problem 11.

(a) (5 points) Write, in set brackets, the entire set of the variable identifiers that occur free in the statement shown in Figure 4. For example, write $\{V, W\}$ if the variable identifiers that occur free are V and W. If there are no variable identifiers that occur free, write $\{\}$.

(b) (10 points) Write, in set brackets, the entire set of the variable identifiers that occur bound in the statement shown in Figure 4. For example, write $\{V, W\}$ if the variable identifiers that occur bound are V and W. If there are no variable identifiers that occur bound, write $\{\}$.

Remember to paste your answer into the answer box and clearly identify each part of the answer.

12. [Concepts]

    This is a problem about free and bound identifier occurrences. In this problem, we will consider `Number.'+'` and `Int.'div'` to each be single identifiers (that is, each matches the syntax ⟨x⟩).

    Consider the kernel language statement shown in Figure 5. (Note that there is no **declare** form in the kernel language, so you should not imagine one in the figure.)

```
AvgHelper = proc {$ Lst Count Leng ?Ans}
             case Lst of
                '|'(1: Z 2: Zs) then
                    local Sum in
                       {Number.'+' Z Count Sum}
                       local One in
                          One = 1
                          local NewLeng in
                             {Number.'+' One Leng NewLeng}
                             {AvgHelper Zs Sum NewLeng Ans}
                          end
                       end
                    end
             else {Int.'div' Count Leng Ans}
             end
          end
Avg = proc {$ Lst2 ?Res}
             local Unused in
                local Zero in
                   Zero = 0
                   {AvgHelper Lst2 Zero Zero Res}
                end
```

Figure 5: Kernel language statement for problem 12.

(a) (5 points) Write in set brackets, the entire set of the variable identifiers that occur free in the statement shown in Figure 5. For example, write $\{V, W\}$ if the variable identifiers that occur free are V and W. If there are no variable identifiers that occur free, write $\{\}$.

(b) (10 points) Write in set brackets, the entire set of the variable identifiers that occur bound in the statement shown in Figure 5. For example, write $\{V, W\}$ if the variable identifiers that occur bound are V and W. If there are no variable identifiers that occur bound, write $\{\}$.

Remember to paste your answer into the answer box and clearly identify each part of the answer.

13. [Concepts] [MapToLanguages] Consider the Java program in Figure 6.

```
public class Point2D {
    int x;
    int y;

    public Point2D(int a, int b) { x = a; y = b; }

    public Point2D makeOffset(int dx, int dy) {
        int newx = x+dx;
        int newy;
        return new Point2D(newx, y+dy);
    }
}
```

Figure 6: Code for Problem 13.

Answer the following questions with respect to the program in Figure 6.

(a) (3 points) Are the occurrences of the identifiers x and y within the constructor free or bound?

(b) (3 points) Does dx occur in this program as a free or bound identifier?

(c) (3 points) Does newy occur in this program as a free or bound identifier?

14. [Concepts]

Consider the code in Figure 7.

```
local Z in
   local MulByN in
      local N in
         N = 7
         MulByN = proc {$ X ?Y}
                     {Number.'*' N X Y}
                  end
      end
      local N in
         N = true      % line 10
         {MulByN 6 Z}
      end
   end
   {Browse Z}
end
```

Figure 7: Code that calls `MulByN`.

Answer the following questions.

(a) (5 points) When you run this code in Oz, what, if anything, is shown in the browser?

(b) (5 points) In what way does the binding of `N` to **true** on line 10 affect the output of the program?

(c) (5 points) If this code were executed with dynamic scoping, what would happen when the program was run?

15. (0 points) [Concepts] [UseModels] For practice (note that this is optional, you will not turn this in), do problems 5 (the case statement) and 6 (the case statement again) in the textbook. These problems allow you to check your understanding of the **case** statement using the Oz implementation.

16. (20 points) [Concepts]

    Do the textbook's problem 4 (**if** and **case** statements). For your answers, give a both a rule for the translation and translate our challenge examples using your translation rule. (That is, don't just show us your translation of our example, but give both the rule and your translation.) Check your translated examples, which should be Oz code, by executing them in the Oz system. For each example, both the original code and its translation should run and give the same results.

    What we mean by a translation (or desugaring) rule is shown by the following example rule. The example rule below desugars an arbitrary but fixed call to a procedure P with an expression E as an argument:

    $\{P\ E\}$
    $\Rightarrow$
    local X **in** X=E $\{P\ X\}$ **end**

    In the part of the solution that translates a **case** statement into a statement that uses **if** statements, you can use the built-in functions `IsRecord`, `Label`, and `Arity`, as well as the operators `.` and `==` (see the Mozart/Oz system document *The Oz Base Environment* [DKS06]). (You can use `.` and `==` infix, as you don't have to translate all the way to the kernel language.)

    Finally, for this problem it seems most sensible to only consider inputs that are in kernel syntax. This is sensible because we can use other rules to desugar an **if** or **case** statement that uses more than kernel syntax into one that only uses kernel syntax. This assumption will also simplify what you have to do.

    As a challenge example for translating **if** to **case** (part (a)), you are to translate the following example. (Note that in this example, `X` is a free variable identifier, so if you want to run it, you will have to declare `X` and give it a value.)

```
if X
then {Browse 'is true'}
else {Browse 'is false'}
end
```

    For part (b), describe your translation for the **case** statement for an arbitrary, but fixed, pattern of the form $L(F_1 : P_1 \cdots F_n : P_n)$. That is, your translation rule for **case** should start out with:

    case X **of** $L(F_1 : P_1 \cdots F_n : P_n)$ **then** S_1 **else** S_2 **end**
    $\Rightarrow$
    ... **if** ...

    where X is a variable identifier, L is a literal, $n \geq 0$, $F_1, \ldots, F_n$ are field names in sorted order, $P_1, \ldots, P_n$ are variable identifiers (that we assume, without loss of generality, are distinct from the names of built-in functions), and S_1 and S_2 are statements. Note that S_1 and S_2 can have (free) occurrences of the variables declared in P_1 to P_n.

    As a challenge example for translating **case** to **if**, you are to translate the following example. (Note that in this example, `Y` and `C` are free variable identifiers that have no built-in value in Oz; so if you want to run the code, you will have to declare both of these and give them values.)

```
case Y of
   shoe(make: Mk model: Mo) then {Browse Mk#Mo}
else {Browse 'no match'#C}
end
```

17. (10 points) [Concepts]

    Do problem 8 (control abstraction) in the textbook.

    For this problem, please put your code for part (b) in a file `OrElse.oz` and (after doing your own testing) use our test cases (in `OrElseTest.oz`) to test your code.

Points

This homework's total points: 207.

References

[DKS06] Denys Duchier, Leif Kornstaedt, and Christian Schulte. *The Oz Base Environment*. mozart-oz.org, June 2006. Version 1.3.2.

[HK06] Martin Henz and Leif Kornstaedt. *The Oz Notation*. mozart-oz.org, June 2006. Version 1.3.2.

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, Mass., 2004.