# Homework 2: Declarative Computation Model

See Webcourses and the syllabus for due dates.
Note that you can't do quizzes on webcourses after their due date, so be sure to take any webcourses quizzes by the date they are due!
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 programming 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:** For each problem that requires code, turn in (on Webcourses) your code and output of your testing. Please upload code as a plain (text) file with the name given in the problem or testing file and with the suffix `.oz`, and also paste the output from our tests into the answer box on webcourses. For English answers, please paste your answer into the answer box in the problem's "assignment" on Webcourses. For a problem with a mix of code and English, follow both of the above.
Your code should compile with Oz, if it doesn't you probably should keep working on it. Email the staff with your code file if you need help getting it to compile. If you don't have time, at least tell us that you didn't get 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 **`continue`** statements.

Remember that a linguistic abstraction has an alphabetic name.

Remember to paste your answer into the answer box on webcourses. If you are worried about the formatting, use the HTML <pre> before your text, and </pre> afterwards, and check the box marked "Use HTML" under the answer box. Do *not* upload an attachment.

2. (7 points) [Concepts] [MapToLanguages] Read through sections 2.1–2.3 and sections 2.5–2.6 of the textbook [VH04], and then do the quiz on webcourses titled "Declarative Model (Chapter 2) Quiz." (Note that you can't do this after the due date, so be sure to do this on time!)

Read through section 2.4 of the textbook [VH04], and then answer the following questions.

3. [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 Res in
   local Pi in
      thread
         Pi = 3.14
         Res = Pi+Y  % line 5
      end
      thread  % line 7
         {Delay 10000}
         Y = 6.86
         {Browse Res}
      end     % line 11
   end
end
```

Figure 1: Oz program for question 3.

   (a) (2 points) On line 5 of Figure 1, is the occurrence of the variable identifier `Pi` a bound occurrence of that variable identifier, or is it a free occurrence?

   (b) (2 points) When starting to execute line 5 of Figure 1, will the store variable that `Res` denotes be a bound store variable or will it be undetermined?

   (c) (2 points) On line 5 of Figure 1, is the occurrence of the variable identifier `Y` a bound occurrence of that variable identifier, or is it a free occurrence?

   (d) (2 points) When starting to execute line 5 of Figure 1, will the store variable that `Pi` denotes be a bound store variable or will it be undetermined?

   (e) (2 points) On line 5 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) Consider the program after 10 seconds have passed and the second thread (between lines 7 and 11) has executed. When this happens, line 5 in Figure 1 finishes execution. In that situation, after successfully executing line 5 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?

4. [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 (in part B) about anything you had to assume in the translation, especially if that makes your translation less than perfectly general.

   As usual, paste your answer into the answer box on webcourses. If you are worried about the formatting, use the HTML `<pre>` before your text, and `</pre>` afterwards, and check the box marked "Use HTML" under the answer box. Do *not* upload an attachment. Please use the row numbers in the left column, as indicated.

| | 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) Describe any problems you encountered in the translation, which would make your translation less than perfectly general or perfectly equivalent to the Oz code.

   (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 answer box on webcourses.

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 self-test on desugaring in Webcourses.

5. (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 1.2 2012/01/28 18:49:35 leavens Exp leavens $
local Ls=nil Res Together in
   fun {Together Ls1 Ls2}
      {Append Ls1 with|Ls2}
   end
   Res = {Together Ls [a b]}
   % {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. (See Tables 2.1 and 2.2 in our textbook [VH04] for the syntax of the declarative kernel.)

   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.

6. (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` with argument `'Expected exception not raised'!` if the call to `F0` did *not* raise an exception, or
   - calls the procedure `ReportAbout` with argument `'Caught exception '#{Value.toVirtualString X 5 5}#'as expected'` if the call to `F0` raised an exception containing the Value `X`.

   You task is to fill in the missing code in Figure 2 so that it has this behavior. Note the code you write for catching exceptions will have to declare the variable `X` that is used in the call to ReportAbout.

   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 run, 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` (which is in the `hw2-tests.zip` file). 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 in the middle of the test. (These failure messages are necessary to test the testing code.)

Read through sections 2.8.2 and 2.8.3 of the textbook.

7. (2 points) [Concepts] [MapToLangauges] Take the quiz titled "Declarative Model (sections 2.8.2–2.8.3) quiz" on webcourses.

[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 1.2 2012/01/28 19:18:37 leavens Exp leavens $
\insert 'TestingNoStop.oz'
\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: 1.2 $'}

{StartTesting 'exceptions not raised, exactly 2 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.

# Regular Problems

The following problems relate to section 2.4.3 of the textbook [VH04].

8. [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 self-test 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.)

```
Chicken = proc {$ What Combo ?R}
             local Ford in
                local C in
                   C = ' chicken '
                   R = '#'(1: What 2: C 3: Combo)
                end
             end
          end
Curry = proc {$ F ?Res}
           Res = proc {$ X ?R1}
                  R1 = proc {$ Y ?R2}
                          {F X Y R2}
                       end
                end
        end
local Result in
   local Ignored in
      local CC in
         {Curry Chicken CC}
         local Carrot in
            local Soup in
               Carrot = carrot
               Soup = soup
               local CarrotChicken in
                  {CC Carrot CarrotChicken}
                  {CarrotChicken Soup Result}
               end
            end
         end
      end
   end
end
```

Figure 4: Kernel language statement for problem 8.

   (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.

9. [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 variable identifier 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.)

```
Helper = 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 BigLeng in
                             {Number.'+' One Leng BigLeng}
                             {Helper Zs Sum BigLeng Ans}
                          end
                       end
                    end
             else {Int.'div' Count Leng Ans}
             end
          end
Avg = proc {$ Lst2 ?Res}
             local Unused in
                local Zero in
                   Zero = 0
                   {Helper Lst2 Zero Zero Res}
                end
             end
          end
```

Figure 5: Kernel language statement for problem 9.

   (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.

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

```
public class Point2D {
    int xval;
    int yval;

    public Point2D(int x, int y) { xval = x; yval = y; }

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

Figure 6: Code for Problem 10.

Answer the following questions with respect to the program in Figure 6. Note that in Java, local variable declarations declare the name of the local variable, field declarations declare the name of the field, and method declarations are really declarations, and also declare the names of the formal parameters and each method's name. This is unlike Oz, where "fun" statements do not declare the name but merely unify a procedure value with it.

(a) (3 points) Are the occurrences of the identifiers `xval` and `yval` 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?

(d) (3 points) Does `makeOffset` occur in this program as a free or bound identifier?

Read section 2.4.1 of the textbook and answer the following questions.

11. [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?

Read section 2.4.4 of the textbook before answering the following 2 problems.

12. (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.

13. (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`** where $X$ is fresh

    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`** (i.e., for 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 thenPart(1: Mk 2:Mo)}
else local Msg in Msg = 'no match' {Browse elsePart(1: Msg 2:C)} end
end
```

Read section 2.6.1 in the textbook before answering the following problem.

14. (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. See Figure 8 on the next page for the testing code.

# Points

This homework's total points: 166.

# References

[DKS06] Denys Duchier, Leif Kornstaedt, and Christian Schulte. *The Oz Base Environment*. 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.

```
% $Id: OrElseTest.oz,v 1.1 2012/01/28 18:49:35 leavens Exp leavens $
\insert 'OrElse.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'OrElse $Revision: 1.1 $'}
{Test {OrElse
       fun {$} true end
       fun {$} {Assert false} false end
      }
 '==' true}
local X=7 in
   {Test {OrElse
          fun {$} X>0 end
          fun {$} (3 div X) > 5 end
         }
    '==' true}
end
{StartTesting 'false orelse true'}
local Z in
   {Test {OrElse
          fun {$} false end
          fun {$} Z=7 true end
         }
    '==' true}
   {Assert {IsDet Z} andthen Z==7}
end
{StartTesting 'true orelse true'}
local Z Q in
   {Test {OrElse
          fun {$} Z=333 true end
          fun {$} Q=444 true end
         }
    '==' true}
   {Assert {IsDet Z} andthen Z==333}
   {Assert {Not {IsDet Q}}}
end
{StartTesting 'false orelse false'}
local Z in
   {Test {OrElse
          fun {$} Z=22 false end
          fun {$} false end
         }
    '==' false}
   {Assert {IsDet Z} andthen Z==22}
end
{DoneTesting}
```

Figure 8: Testing code for problem Problem 14 on the preceding page.