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++, and Java [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. For programming problems for which we provide tests, you can find them all 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.
Don't hesitate to contact the staff if you are stuck at some point.
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, and C++ is a linguistic abstraction of a **`while`** loop. In Java, **`interface`**s are also linguistic abstractions of abstract classes. Give another, different example of a linguistic abstraction in Java, 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) naming the main syntactic construct that it is an abstraction of.

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?
   (c) (3 points) What does a thread in Oz do when a dataflow variable is accessed before its value is determined (i.e., before it is bound).

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

3. (5 points) [Concepts] [MapToLanguages] What kind of typing does C# have? (Note: C# is designed to be very similar to Java.)

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

4. (5 points) [Concepts] What is the main advantage of static (i.e., lexical) scoping? (Give a brief answer.)

5. [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 A in
   local B in
      A = 7
      Res = A*B  % line 4
   end
end
```

Figure 1: Oz program for question 5.

   (a) (2 points) On line 4 of Figure 1, is the occurrence of the variable identifier `B` 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 `B` 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 `A` 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 `A` 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 (i.e., suppose that another thread unified `B` with 10), in that situation, after executing line 4 of Figure 1, would the store variable that `Res` denotes be a bound store variable or would it be undetermined?

   (g) (5 points) Must a bound occurrence of a variable identifier always denote a determined value at runtime?

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." 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?

   (c) (3 points) What kinds of "cleanup" actions should a Java program take to ensure that it does not keep memory allocated that it no longer needs?

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

7. [Concepts] [MapToLanguages]

   (a) (2 points) What is Oz's **`andthen`** operator like in Java or C++?

(b) (3 points) What is the equivalent of the Java or C++ expression `A != B` in Oz?

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

8. (3 points) [Concepts] Give a simple example of the syntax used in Oz to throw an exception.

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

9. [Concepts]
   (a) (2 points) Which language has dynamic type checking: Oz or Java?
   (b) (3 points) Which language has static type checking: Oz or Java?

Regular Problems

We expect you'll do the problems in this section after reading the entire chapter. However, you can probably do some of them after reading only part of the chapter.

Some of the following problems are from the textbook [VH04, section 2.9].

10. [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 2. (Note that there is no **declare** form in the kernel language, so you should not imagine one in the figure.)

```
Composer = proc {$ H G A R}
              local Tmp in
                 {G A Tmp}
                 {H Tmp R}
              end
           end
Adder1 = proc {$ B Result}
           local Seven in
              local Eight in
                  Seven = 7
                  {Adder B Seven Result}
              end
           end
        end
local Ret in
   local Four in
      Four = 3
      {Composer Adder1 Id Four Ret}
   end
end
```

Figure 2: Kernel language statement for problem 10.

   (a) (5 points) Write, in set brackets, the entire set of the variable identifiers that occur free in the statement shown in Figure 2. 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 2. 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 $\{\}$.

11. [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 $\langle$`x`$\rangle$).

    Consider the kernel language statement shown in Figure 3 on the following page. (Note that there is no **declare** form in the kernel language, so you should not imagine one in the figure.)

   (a) (5 points) Write in set brackets, the entire set of the variable identifiers that occur free in the statement shown in Figure 3 on the next page. 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 $\{\}$.

```
AvgIter = proc {$ Ls Total Len ?Answer}
             case Ls of
                '|'(1: X 2: Xs) then
                    local Sum in
                       {Number.'+' X Total Sum}
                       local One in
                          One = 1
                          local NewLen in
                             {Number.'+' One Len NewLen}
                             {AvgIter Xs Sum NewLen Answer}
                          end
                       end
                    end
             else {Int.'div' Total Len Answer}
             end
          end
Average = proc {$ Lst ?Ans}
             local Unused in
                local Zero in
                   Zero = 0
                   {AvgIter Lst Zero Zero Ans}
                end
```

Figure 3: Kernel language statement for problem 11.

(b) (10 points) Write in set brackets, the entire set of the variable identifiers that occur bound in the statement shown in Figure 3. 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 $\{\}$.

12. [Concepts] [MapToLanguages] Consider the Java program in Figure 4.

Answer the following questions with respect to the program in Figure 4.

(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?

```
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 4: Code for Problem 12.

13. [Concepts]

Consider the code in Figure 5.

```
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 5: 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?

14. [Concepts] [MapToLanguages]

    This problem tries to get you to think about how environments are manipulated by calls in Java, and in that sense is similar to the previous problem, but for Java.

    To understand this question, you need to understand how **this** works in Java. First, Java's **this** is an identifier that is implicitly declared by Java's **class** mechanism.

    Second, when Java executes a method call, such as `r.printThis()`, Java looks at the dynamic class of the receiver object, which is the value of the receiver expression (`r`), and uses that to find the code for the method (`printThis`). To execute that code, Java creates up an environment, which maps **this** to the receiver's value, and the formals to the actual parameters' values, and then runs the body of the code it found. Note that the environment created maps the identifier **this** to the current receiver object.

    To see how this works, consider the code in Figure 6 on the next page. This code, when run in Java, produces output like the following.

```
Starting Main
Main@17590db
Honda car 4-door
Main@17590db
Ford truck with 7000 lb payload
```

    We now explain how the code in Figure 6 on the following page generates the above output. After an initial message, the output shows that the value of **this** in the `doPrinting` method is an object of class `Main` at address `17590db`. Then when `c.printThis()` is executing, the value of **this** is a car object. Upon return from that method, the environment inside the method `doPrinting` is unaffected, and again the value of **this** in the `doPrinting` method is an object of class `Main` at address `17590db`. But when `t.printThis()` is executing, the value of **this** is a truck object.

    So, with that in mind, we want to consider why the environment has to be set up in such a way as described above. To do that, consider the Java code in Figure 7 on page 9.

    (a) (5 points) Given the above description of how **this** is declared and used in Java, briefly explain why occurrences of the identifier **this** in lines 3 and 4 of Figure 7 on page 9 should be considered to be bound occurrences?

15. [Concepts] Before starting on this and other problems that ask you to desugar into the kernel language, you may want to do the ungraded quiz on desugaring in Webcourses.

    (a) (10 points) Translate the **proc** statement given in the textbook's chapter 2 problem 1 into the declarative kernel language's syntax. This means to produce a statement that has the same meaning but which only uses the syntax given in Tables 2.1 and 2.2 of the textbook [VH04]. Check carefully that your translation matches that grammar. Since this grammar does not allow the use of infix operators like > and -, in your translation you should use the built-in procedures `Value.'>'` and `Number.'-'` (see the Mozart/Oz system document *The Oz Base Environment* [DKS06], sections 3 and 4 for more about these). For purposes of this problem, we will consider `Value.'>'` and `Number.'-'` to be identifiers (matching the syntax ⟨x⟩).

    Put your translation in a file `Pkernel.oz` and turn that in as your answer for this part of the problem.

    (Hint: to check for some syntax errors, add the line **declare** `P` **in** just before your translation, then and feed the translated code to the Oz system. However, Oz will only check against the full language syntax, so you still might be using parts of the Oz syntax that are not in the kernel syntax [VH04, Tables 2.1 and 2.2]. So you still need to check by hand that your code is in the kernel language. Finally, we allow comments in the kernel syntax.)

    (b) (5 points) Do the textbook's problem 1 (free and bound identifiers).

    (Hint: note that the question refers only to the statement itself; that is, the statement does not include any (implicit) **declare**, since **declare** is not in the kernel language.)

```
public class Main {
    public static void main(String [] argv) {
        System.out.println("Starting Main");
        Main m = new Main();
        m.doPrinting();
    }

    public void doPrinting() {
        System.out.println(this);
        Car c = new Car("Honda", 4);
        Truck t = new Truck("Ford", 7000);
        c.printThis();
        System.out.println(this);
        t.printThis();
    }

}

public abstract class Vehicle {
    protected String name;
    protected Vehicle(String make) { this.name = make; }
    public String toString() { return name; }
    public void printThis() {
        System.out.println(this);
    }
}

public class Car extends Vehicle {
    protected int doors;

    public Car(String make, int num_doors) {
        super(make);
        this.doors = num_doors;
    }

    public String toString() {
        return super.toString() + " car "
            + this.doors + "-door";
    }
}

public class Truck extends Vehicle {
    protected int payload;

    public Truck(String make, int carries) {
        super(make);
        this.payload = carries;
    }

    public String toString() {
        return super.toString() + " truck with "
            + this.payload + " lb payload";
    }
}
```

Figure 6: An example showing how **this** works in Java.

```
public class Adder {
    private int n;
    public Adder(int n) { this.n = n; }
    public int add(int x) { return this.n + x; }
}
```

Figure 7: Code for Problem 14 on page 7.

16. (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.

17. (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 'was true'}
    else {Browse 'was 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 varaibles 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, so if you want to run it, you will have to declare both of these and give them values.)

    ```
    case Y of
       winter(city: C country: K) then {Browse C#K}
    else {Browse 'nope'#C}
    end
    ```

18. (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.

19. (25 points) [Concepts] [UseModels]

    Do the book's problem 9 (tail recursion) parts (a), (b), and (c), but see below for special directions regarding parts (a) and (b).

    For part (a), use *The Oz Base Environment* [DKS06], to find identifiers that you can use in place of the infix operators, so that your expansion into kernel syntax will, for example, use `Value.'=='` instead of the infix operator == and `Number.'-'` instead of -. Put your answer for this part into a text file named `tailrecursion.oz`. Test your code by making at least one call to each procedure.

    For part (b), instead of writing out an answer in detail, just describe how large the stack would become in each of the two cases.

20. (10 points) [Concepts] [UseModels]

    Do problem 10 (expansion into kernel syntax). Again, use *The Oz Base Environment* [DKS06], to find the identifiers that you can use in place of the infix operators. Also, according to *The Oz Notation* [HK06], if a **`case`** statement is missing an **`else`** clause, you should add

    ```
    else raise error(kernel(noElse ...) ...) end
    ```

    as an implicit **`else`** clause (even though this steps outside the declarative model by using exceptions).

Points

This homework's total points: 209.

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.