Homework 2: Declarative Computation Model

Due: problems 1–3 on Wednesday, September 10, 2008; problems 4–11 on Monday, September 22, 2008.

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 exception handling. You'll see how the declarative computation model relates to C, C++, and Java [MapToLanguages].

Your code should be written in the declarative model, so do not use cells and 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.)

For all programing tasks, you must run your code using the Mozart/Oz system. Turn in output for whatever testing you do for these problems. 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.

Turn in (on Webcourses) your code and output of your testing for all exercises that require code. Please upload code as text files with the name given in the problem or testing file and with the suffix `.oz`. Please use the name of the main function as the name of the file. Please upload test output and English answers as plain text files with suffix `.txt` or paste that output into the answer box in the assignment on Webcourses. If you have a mix of code and English, use a text file with a `.oz` file suffix, and put comments in the file for the English parts. (In any case, don't put any spaces 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.

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`.

Don't hesitate to contact the staff if you are stuck at some point.

For background, you should have already read Chapter 2 of the textbook [RH04]. 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.

Normal Problems

The following problems are based on those in the textbook [RH04, section 2.9].

1. [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.

   Consider the kernel language statement shown in Figure 1. (Note that there is no **`declare`** form in the kernel language, so you should not imagine one in the figure.)

```
Copy = proc {$ Ls ?R}
          case Ls of
             '|'(1:H 2:T) then local NT in {Copy T NT}  R='|'(1:H 2:NT) end
          else R=nil
          end
       end
```

Figure 1: Kernel language statement for problem 1.

   (a) (10 points) Write, in set brackets, the entire set of the variable identifiers that occur free in the statement shown in Figure 1. 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 1 on the preceding page. 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 $\{\}$.

2. [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. In this problem, we will consider `Number.'+'` and `Int.'div'` to each be single identifiers (that is, each matches the syntax $\langle \mathsf{x} \rangle$).

Consider the kernel language statement shown in Figure 2.

```
F = proc {$ X Q ?R}
       local XA in
          {Number.'+' X A XA}
          local B in
             B={IsEven X}
             if B
             then local T in  {Int.'div' XA X T} {F T A R}  end
             else R = XA
             end
          end
       end
    end
```

Figure 2: Kernel language statement for problem 2.

(a) (10 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 $\{\}$.

3. (6 points) [Concepts] [MapToLanguages]

Consider the following C (or C++) program.

```
long fact(long n) {
  if (n == 0) { return 1; } else { return n*fact(n-1); }
}
```

Answer the following questions with respect to this entire program. (a) Are the occurrences of the identifier `n` in the second line free or bound? (b) Is the occurrence of `fact` on the second line free or bound?

4. (20 points) [Concepts]

Do the textbook's problem 2 (contextual environment). (The problem asks for answers to three questions and also for two examples.)

To explain this a bit, recall the semantics of procedure calls from 2.4.4 of the text [RH04]. When a call, such as the call `{MulByN A B}` considered in the problem, is executed the semantics checks that that the procedure's identifier, `MulByN` in this problem, has a determined value that is a closure with the right number of arguments (2 in this case). In this problem, we're assuming that the closure denoted by `MulByN` is the following:

(**proc** `{$X ?Y} Y=N*X` **end**, $\{\texttt{N} \mapsto 3\}$).

Execution then proceeds by executing the closure's body, in this case the statement `Y=N*X`, in an environment formed from the closure's environment (in this case $\{\texttt{N} \mapsto 3\}$) and bindings mapping each formal to the actual's value, which in this case is $\{\texttt{N} \mapsto 3, \texttt{X} \mapsto 10, \texttt{Y} \mapsto x_1\}$, assuming that the actual parameter `A` denotes 10 at the call site, and that the actual parameter `B` denotes the location x_1 (whose value is undetermined).

Thus the book is asking:

(a) Why must the execution of a procedure call run the body in an environment that includes the closure's environment (e.g., $\{\texttt{N} \mapsto 3\}$ in this case), when executing such a call?

(b) Can the semantics avoid using the closure's environment to execute the call because some binding for `N` will always be present in the envrionment active at the point of the call? The first example should demonstrate your answer to this.

(c) Won't the environment at the point of the call always map `N` to 3? The second example should demonstrate your answer to this part.

5. [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 `c.printThis()`, Java looks at the dynamic class of the receiver object, which is the value of the expression `c`, and uses that to find the code for the method `printThis`. It then sets up an environment, which maps **`this`** to the receiver, and the formals to the actual parameters, 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 3 on the following 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 3 on the next 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 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. To do that, consider the Java code in Figure 4 on page 5.

(a) (5 points) Given the above description of how **`this`** is declared and used in Java, Should the occurrences of this identifier in lines 3 and 4 of Figure 4 on page 5 be considered free or bound?

(b) (5 points) When a call `(`**`new`**` Multiplier(3)).multiply(8)` is executed, how does the code in the body of `multiply` access the field `n`? That is, in general terms, how can the generated code access the location corresponding to the field `n` of the correct object, in order to obtain 3?

(c) (5 points) Give an example, in Java, of a call to the method `multiply` that shows why the environment must bind **`this`** to the receiver when running the body. That is, give some Java code that, when run, would have an environment at the point of the (only) call to the `multiply`

```
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 3: An example showing how **this** works in Java.

```
public class Multiplier {
    private int n;
    public Multiplier(int n) { this.n = n; }
    public int multiply(int x) { return this.n * x; }
}
```

Figure 4: Code for Problem 5 on page 3.

method that associates **this** with the wrong object for accessing the field `n`. (Hint: look at the code in Figure 3 on the previous page.)

Your answer for this part of the problem should be in a `.java` file, with comments explaining how it answers this question.

6. [Concepts]

   (a) (10 points) Translate the **proc** statement given in the textbook's 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 [RH04]. 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 use parts of the Oz syntax that are not in the kernel syntax [RH04, Tables 2.1 and 2.2]. So you still need to check 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.)

7. (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 an illustrative example that follows your translation rule. (That is, don't just show us an example.) Check your translation rule examples, which should be Oz code, executing them in the Oz system. Both the original and the translation should run and give the same results.

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

   $\{P\ E\}$
   ⇒
   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.)

   Describe your translation for an arbitrary, but fixed, pattern of the form $L\,(F_1 : P_1 \cdots F_n : P_n)$.

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.

8. (10 points) [Concepts]

   Do problem 8 (control abstraction).

   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.

9. (25 points) [Concepts] [UseModels]

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

10. (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).

11. (10 points) [Concepts]

    Do problem 13 (unification).

Suggested Practice and Extra Credit Problems

12. (suggested practice) [Concepts] [UseModels]

    Do problems 5 (the case statement) and 6 (the case statement again) in the textbook.

13. (20 points; extra credit) [Concepts]

    Using the operational semantics presented in the book (or the notes), trace the execution of the code in the textbook's problem 7.

14. (40 points; extra credit) [Concepts] [UseModels]

    Write code in Oz that translates code written in the extended language of chapter 2 into the kernel language. You can use parsing or other tools that come with Oz. The input should be text and the output should also be text. (Hint: you may want to use GUMP.)

    For example, given the input

    ```
    local Th = 3 in {Browse Th*Th} end
    ```

    it would produce the output

    ```
    local Th in Th=3 local X in X=Th*Th {Browse X} end end
    ```

    (Such outputs may be easier to read if indented, but that is not necessary for this problem.)

15. (60 points; extra credit) [Concepts] [UseModels]

    Write code in Oz that prints out a series of steps that the operational semantics of Oz takes when executing a kernel program. The input to this program should be a text string that is in the kernel language.

    For example, given the input

    ```
    local Th in Th=3 local X in X=Th*Th {Browse X} end end
    ```

    it would produce output similar to the following.

    ```
    ([(local Th in Th=3 local X in X=Th*Th {Browse X} end end], {}), {})
    -->
    ([(Th=3 local X in X=Th*Th {Browse X} end, {Th-->x1})], {x1})
    -->
    ([(Th=3, {Th-->x1}) (local X in X=Th*Th {Browse X} end, {Th-->x1})], {x1})
    -->
    ([(local X in X=Th*Th {Browse X} end, {Th-->x1})], {x1=3})
    -->
    ([(X=Th*Th, {X-->x2, Th-->x1}) ({Browse X}, {X-->x2, Th-->x1})], {x1=3,x2})
    -->
    ([({Browse X}, {X-->x2, Th-->x1})], {x1=3, x2=9})
    ```

    At the end, the machine gets "stuck" when trying to execute `Browse`, since it is not found in the environment.

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.

[RH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, Mass., 2004.