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

Code for programming problems should be written in Oz's 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 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 the course resources web page or 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.

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 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.

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 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] Give an 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] [MapToLanguages]
   (a) (4 points) What happens in C or C++ if a program uses a variable before its value is determined?
   (b) (3 points) What happens in Java if a program uses a field before its value is determined?
   (c) (3 points) What does Java do with a method that tries to access a local variable before it has been assigned a value?

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

3. (5 points) [Concepts] [MapToLanguages] What kind of typing does Java have?

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

4. [Concepts]
   (a) (5 points) What is the main purpose of static scoping? (Give a brief answer.)
   (b) (5 points) What is the difference between a "bound variable identifier occurrence" and a "store variable that is bound"?

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

5. [Concepts] [MapToLanguages]
   (a) (5 points) Do C, C++, and Java compilers (in your experience) use the last call optimization?
   (b) (2 points) Does Java have garbage collection?
   (c) (3 points) Can a Java program have dangling reference errors?

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

6. [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++ operator `!=` in Oz?

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

7. (3 points) [Concepts] In Oz, what types of values can be used in Oz for throwing exceptions?

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

8. [Concepts]
   (a) (5 points) Would it make sense to have a unification operator in Java or C++?
   (b) (2 points) What kind of typing (i.e., type checking) is in Oz?
   (c) (3 points) What kind of typing (i.e., type checking) is in 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.

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

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

```
Copy = proc {$ LstArg ?Result}
          case LstArg of
             '|'(1:Hd 2:Tl) then
                local NewTail in {Copy Tl NewTail}  Result='|'(1:Hd 2:NewTail) end
          else Result=nil
          end
```

Figure 1: Kernel language statement for problem 9.

   (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. 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 $\{\}$.

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. 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 2.

```
Bump = proc {$ J I ?R}
       local JpI in
          {Number.'+' J M JpM}
          local Evn in
             Evn={IsEven J}
             if Evn
             then local Tmp in {Int.'div' JpM J Tmp} {Bump Tmp M R}  end
             else R = JpM
             end
          end
       end
    end
```

Figure 2: Kernel language statement for problem 10.

(a) (10 points) Write in set brackets, the entire set of the variable identifiers that occur free in the statement shown in Figure 2 on the preceding 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 $\{\}$.

(b) (10 points) Write in set brackets, the entire set of the variable identifiers that occur bound in the statement shown in Figure 2 on the previous 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 $\{\}$.

11. (6 points) [Concepts] [MapToLanguages]

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

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

Answer the following questions with respect to this entire program.

(a) (3 points) Are the occurrences of the identifier `i` in the second line free or bound?

(b) (3 points) Is the occurrence of `fact` on the second line free or bound?

12. [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 [VH04]. When a call, such as the call `{MulByN A B}` considered in the problem, is executed the semantics checks 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 $\{\texttt{N} \mapsto 3, \texttt{X} \mapsto \texttt{10}, \texttt{Y} \mapsto x_1\}$, 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) (5 points) 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) (3 points) Can the semantics avoid using the closure's environment to execute the call because some binding for `N` will always be present in the environment that is current where the call is made?

(c) (5 points) Write an example that demonstrates your answer to the parts above.

(d) (2 points) Won't the environment that is current where the call is made always map `N` to 3?

(e) (5 points) Write another example that demonstrates your answer to this last question.

Hint: In a statically scoped language like Oz, each properly nested region (e.g., from **`local`** to the matching **end** or from **proc** to the matching **end**) is associated with a set of declarations that are effective in that region. The "current environment" available at runtime is determined by these declarations and the values associated with the locations that these declarations denote, as given by the program's execution of various statements.

13. [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 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 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 4 on page 7.

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

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

14. [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 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.

```
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 13 on page 5.

(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.)

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 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, 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 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:

$\{P\ E\}$
$\Rightarrow$
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.)

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

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.

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.

18. (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.

19. (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).

20. (10 points) [Concepts]

    Do problem 13 (unification).

Extra Credit Problems

Extra credit problems are entirely optional. See the course grading policy for details.

21. (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.

Points

This homework's total points: 240. Total extra credit points: 20.

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.