Homework 2: Declarative Computation Model

Due: problems 1–4 on Friday, September 14, 2007; problems 5–10 on Monday, September 17, 2007; extra credit problems on Monday, September 24, 2007.

In this homework you will learn about the declarative computational model [Concepts], including the concepts of linguistic abstractions, syntactic sugars, and exception handling. You'll see how the declarative computational model relates to C, C++, and Java [MapToLanguages].

For all programing tasks, you must run your code using the Mozart/Oz system. For these you must also provide evidence that your program is correct (for example, test cases). Hand in a printout of your code and the output of your testing, for all questions that require code.

You must also provide evidence that your program is correct (for example, output from test cases). For each problem that requires code, turn in, using your WebCT account for this course, your code and your evidence that your code is correct (e.g., your testing code and its output).

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]. You may also want to refer to reference and tutorial material on the Mozart/Oz web site `http://www.mozart-oz.org/`.

Textbook Problems

The following problems are from the textbook [RH04, section 2.9].

1. (3 points) [Concepts]

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

2. (6 points) [Concepts]

   Consider the following C program.

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

   Answer the following questions. (a) Are the occurrences of `n` in the second line free or bound? (b) Is the occurrence of `fact` on the second line free or bound?

3. (20 points) [Concepts] [MapToLanguages]

   Do the textbook's problem 2 (contextual environment).

4. (15 points) [Concepts] [MapToLanguages]

   This problem tries to get you to think about how environments are manipulated 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 envrionment, 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 1 on page 3. 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
```

Which after an initial message, 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 enviroment 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 following Java code.

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

(a) Given the above description of how **`this`** is declared and used in Java, Should its occurrences in lines 3 and 4 of the above code be considered free or bound?

(b) 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) 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` method that associates **`this`** with the wrong object for accessing the field `n`. (Hint: look at the code in Figure 1 on the next page.)

5. (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.) What we mean by a translation (or desugaring) rule is shown by the following example that desugars a 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 ==. Also, write the 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 kernel syntax (for the relevant parts), as we can desugar an **`if`** or **`case`** statement that uses more than kernel syntax into one that only uses kernel syntax.

6. (15 points) [Concepts] [UseModels]

Do problem 6 (the case statement again).

7. (10 points) [Concepts]

Do problem 8 (control abstraction).

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

8. (25 points) [Concepts] [UseModels]

   Do the book's problem 9 (tail recursion) parts (a) and (c). 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.

9. (10 points) [Concepts] [UseModels]

   Do problem 10 (expansion into kernel syntax).

10. (10 points) [Concepts]

    Do problem 13 (unification).

11. (20 points; extra credit) [Concepts]

    Using the operational semantics presented in class, trace the execution of the code in the textbook's problem 7.

Other Problems

12. (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.)

13. (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.

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