Homework 10: Babel

Due Thursday, Apr 30 at 9:55 am
Remember to write how long this took you (2 points)

We've been writing reductions from a language to itself for some time. Now I want you to write reductions from one language to another. One goal is for you to think carefully about exactly what expressions mean in the source language and about how that same feature is presented in other ones. In each case, match the semantics as closely as possible using as little syntax as possible. We have not studied all of these languages in much detail. You may need to use manuals and internet resources to learn more about them...a second goal is for you to use what you've learned in the class to quickly jump to the important part of a language tutorial or reference manual.

Often the "obvious" solution isn't as close as it can be when you consider scope, mutation, expression values, etc. So be careful and think about each one; **they're all puzzles.** The problems are structured to send you down the right path, so think about how the parts of each one are related.

Write a sentence or two if there's something tricky that you're (or I'm) doing. Each solution only needs to be consistent with the other parts of that problem, not with the other problems on the assignment. Italicized words are expression placeholders; copy them verbatim and assume that the expression has been properly translated into the target language for you.

For this assignment, you may work with a partner. You can't talk to anyone except me, your partner, and the TA about the assignment. Partners hand in one solution for the pair. The idea here is for you to have someone to argue the solutions with, since working by yourself you are more likely to be trapped by some of the tricky parts.

You're invited (but not required) to actually type your solutions in and run them for debugging purposes. See the lecture sample code for examples of how to set up a test program in each language. The compilation and execution commands are:

C++: g++ -o *executablename* *file1*.cpp [*file2*.cpp ...]
executablename

Java: javac *file1*.java [*file2*.java ...]
java *file1*

Python: python *file*.py

Scheme: *Use the DrScheme IDE*

The first one is done for you as an example:

Scheme	C++
(define-syntax-rule (ornot a b) (or a (not b)))	`#define ornot(a, b) \` `((a != false) || ! (b != false))`
(ornot *exp1 exp2*)	`ornot(exp1, exp2)`

1. Closures (8 points)

Scheme	Python
(lambda (*id*) *expb*)	**lambda *id* : *expb***
(*expp expa*)	
(let ([*id expa*]) *expb*)	

2. Polymorphism (10 points)

C++	Python
t ? *c* : *a*	*c* if *t* else *a*
throw "Error";	raise Exception("Error")

template<typename T>
T max(T x, T y) {
  return (x > y) ? x : y;
}	

(do not use the built-in max function!)

3. Conditionals (10 points)

Java	Scheme
if (*t*) *c* else *a*	
t ? *c* : *a*	

4. Conditional II (5 points)

C++	Python
if (*t*) *c* else *a*	
a && b	

5. Testing, testing, 1, 2, 3... (15 points)

Write a short program that demonstrates that Python has:

a) Lexical scope
b) Eager evaluation
c) Early-out AND
d) Run-time type checking
e) Dynamically typed variables

Challenge (infinite glory)
Consider the following Scheme program:

```
(define person
  (lambda (name height)
    (let ([methods
           `((set-name   ,(lambda (n) (set! name n)))
             (get-name   ,(lambda ()  name))
             (get-height ,(lambda ()  height))
             (is-tall?   ,(lambda ()  (> height 2.0))))])

      (lambda (method-name . args)
        (apply (second (assoc method-name methods)) args)))))


(define p (person "Morgan" 2.9))

(p 'set-name "Brent")
(p 'get-name)
(p 'is-tall?)
```

It contains two pieces of syntax you haven't seen before: backtick/comma, and dotted arguments. Backtick and comma are a shorthand; backtick is QUASIQUOTE, which is just like QUOTE, except it lets you use comma inside of it. Comma is UNQUOTE. It undoes the quoting so that you can stick a real expression in the middle of a list and have it evaluated. I'm just using it above to avoid writing (list (list 'set-name (lambda (n) ...)

`(1 2 ,(+ 7 8)) => (quasiquote 1 2 (unquote (+ 7 8))) => (list '1 '2 (+ 7 8))

The syntax (lambda (x . y) ...) means that this procedure takes one or more arguments. When the procedure is applied, x is bound to the first actual value and y is bound to *a list of* the remaining actual values. We actually saw this once before, when we defined MAP using APPLY.

I recommend that you paste this into Dr. Scheme and run it before attempting this problem. Here are the actual questions:

a) Translate the above program into Java in the **most elegant way** possible.

b) What did you learn? (Tip: I'm not asking about quasiquote syntax!)