CSCI-GA-2110 – Problem Set 4 – Written Part

This document describes the written exercises for problem set 4. Each exercise is designated as a "Task" in this document. Please write or type your solutions neatly to these tasks, produce a legible PDF clearly indicating where each question is answered, and upload the results to gradescope.

1 Passing Self with Macros

Recall that in lecture 7, we explored a way to encode objects using a `lambda` that referred to boxes storing the fields of the object. In that encoding, the `lambda` for the object took an argument `m` representing which method to call, and did a `case` on `m`, returning a function for the corresponding method. To support self-reference in objects, each of our methods took as their first argument the object itself. To simplify invoking a method on an object and handling the task of passing the object as the first argument, we wrote a helper function called `msg/self` defined as:

```
(define (msg/self o m . a)
  (apply (o m) o a))
```

With this definition, the following example will return 7, as intended:

```
(define o-self
  (lambda (m)
    (case m
      [(first) (lambda (self x) (msg/self self 'second (+ x 1)))]
      [(second) (lambda (self x) (+ x 1))])))

(msg/self o-self 'first 5)
```

Imagine that instead of implementing `msg/self` as a function in the above, one instead tried to use the following macro definition:

```
(define-syntax (msg/self stx)
  (syntax-case stx ()
    [(msg/self o m a ...)
     #'((o m) o a ...)]))
```

With this macro definition, running `(msg/self o-self 'first 5)` with the definition of `o-self` as in the previous example still correctly returns 7. However, the macro implementation is wrong and leads to unintended behavior.

Task 1.1 (4 pts). Explain what is wrong/unexpected with the macro implementation of `msg/self`. Concretely, give an example of an object and method invocation where replacing the original definition of `msg/self` with the macro version leads to different behaviors.

2 Continuations

In class, we informally used some notation to describe the continuation captured by `let/cc`. For example, in a snippet of code like

```
(+ 1 (+ 2 (+ 3 (+ (let/cc k 4) 5))))
```

we said that the continuation bound to `k` looked like

```
(+ 1 (+ 2 (+ 3 (+ [...] 5))))
```

where if we were to apply `k`, as in `(k v)` the `[...]` represents a "hole" into which the value `v` will be "plugged-in".

Task 2.1 (6 pts). Using the notation in the above example, write down the continuations bound to `k` in `let/cc` for each of the following:

1.
```
(+ (+ 3 5) (+ (let/cc k 4) (+ 6 11)))
```

2.
```
(begin (begin (set-box! b 1) 1)
           (let/cc k (unbox b))
           (set-box! b 2))
```

3.
```
(map (begin
         (begin (set-box! b 1) #f)
         (lambda (x) (+ (unbox b) x)))
       (list (let/cc k 1) 2 3 (+ 2 2)))
```