Meta-Evaluation
MIT 6.001 Recitation

Basics

The meta evaluator is just an implementation of the substitution and environment models.

Notion of syntax vs. semantic.

Dynamic vs. lexical scoping

What do you need to change to update the syntax?

Implement a new version of if: `(si <predicate> alors <expression1> sinon <expression2>)`

Implement a new version of `begin` with an explicit `end: (begin <exp1> <exp2>… <expn> end)`

What do you need to change to update the semantics?

Change the semantics of what `define` returns. You can decide to return the value or the symbol.

Make the evaluation of subexpressions truly random.

How do you add a special form?

Add `is-defined?` and that tells you if a variable is bound.

Note that the binding of variables is stored differently from our usual association lists. Mostly, it makes it easier to manipulate parameter and argument lists in a lambda. What do we need to change to use association lists?

Add `unbind` that removes a binding. This could have been useful for the object-oriented system.

An example

Trace what the Meta-Circular Evaluator does on the following code, draw the corresponding environment diagram as well as the box-and-pointer diagram corresponding to this environment. =

```
(m-eval '(define z (+ 1 3)) the-global-environment)
```

```
(m-eval '(define mult (lambda(x y) (* x y))) the-global-environment)
```

```
(m-eval '(mult z 3) the-global-environment)
```

Scoping

Describe the behavior of the following code for the lexical vs. dynamic scoping implementations of the evaluator.

```
(define PRECISION 0.1)
(define close-enuf? (lambda (guess x)
    (< (abs (- (square guess) x)) PRECISION)))
(define improve (lambda (guess x) (/ (+ guess (/ x guess)) 2)))
(define sqrt-loop (lambda G X)
  (if (close-enuf? G X) G
      (sqrt-loop (improve G X) X )
(define sqrt (lambda (x) (sqrt-loop 1.0 x)))

(define (f x)
    (define PRECISION 0.001)
    (sqrt x))
```

How could we achieve the same behavior using lexical scoping?

Let*

Recall that let* makes the bindings sequential and allows one to use an earlier binding in the later ones.

```
(let* ((x 3)
       (y (+ x 2))
       (z (+ x y)))
    (* x z))
```

Implement `let*` as a syntactic translation.

Letrec

Recall that letrec allows you to define recursive procedures in a let:

```
(letrec ((fact (lambda(n)
    (if (< n 2) 1
                (* n (fact (- n 1)))))))
  (fact 3))
```

By the way, why wouldn't let work in this case?

Implement `letrec` as a syntactic translation.