Announcements

• Fourth homework has been posted.

• Roadmap: Defining interpreters, type systems, logic programming (Prolog), parallel programming
**Closure interpreter $ev$ – basic structure**

$ev$ takes as input an AST of an expression $e$ and an environment $env$, and returns the AST of a value.

```scheme
(define ev
  (lambda (e env)
    (cond
      ((eq? (car e) '&const) ;; $e=(&const c)$
        e)
      ((eq? (car e) '&var) ;; $e=(&var v)$
        (lookup env (cadr e)))
      ((eq? (car e) '&lambda) ;; $e=(&lambda parms body)$
        (mk-closure env e))
      ((eq? (car e) '&if)
        (let ((a (cadr e)) ;; $e=(&if a b c)$
               (b (caddr e))
               (c (cadddr e)))
          (ev (if (equal? (ev a env) '(&const #f)) c b) env)))
      ((eq? (car e) '&apply) ;; $e=(&apply f args)$
        (let*(((f (cadr e))
               (args (caddr e))
               (fv (ev f env))
               (av (map (lambda (a) (ev a env)) args)))
               (if (and (pair? fv) (eq? (car fv) '&const))
                   (delta fv av)
                   (apply-cl fv av))))))))
```

198:515, Spring 2016
Call-by-value closure interpreter evaluate

That’s it! We are now ready to put everything together.

(define evaluate
 (lambda (m)
   (unparse (ev (parse m) empty-env))))

Questions

• What parameter passing style does our interpreter use?

• What is the order of evaluation of the actual parameters for a function application?
**Call-by-name closure interpreter**

How do we have to modify our interpreter to implement call–by–name instead of call–by–value?

**Key idea:**
If the argument of an application is not a value, we can postpone its evaluation by wrapping it into a closure that treats the argument expression as a lambda abstraction without parameters. Such a “special” closure is called a **thunk**. If we need the “real” value of an argument expression, we just evaluate the thunk in its environment.

Note that now our environment can contain three types of values, namely *constants, closures*, and *thunks*. 
Dynamically-scoped interpreter

How do we have to modify our interpreter to implement call-by-value with dynamic scoping?

**Key idea:**
The difference between static and dynamic scoping in TINY is with respect to the rules how closures are applied.

- static ⇒ use environment within the closure
- dynamic ⇒ ignore environment within the closure and use current environment at the application.

See example interpreters on **ilab**:

- static:
  
  ~uli/cs515/examples/scheme/DefiningInterpreters/ValueStatic.ss

- dynamic: ???

- static-name: ???
Closure interpreters — Remember the Y

(let (f)
  ((let (x) (f (x x)))
   (let (x) (f (x x)))))

This does not work for call-by-value interpreters!

**Trick:** – Enclose the self application (x x) within a function (let(y) ((x x) y))

(let (f)
  ((let (x) (let (y) ((f (x x)) y)))
   (let (x) (let (y) ((f (x x)) y)))))

Let’s try an example using our call-by-value interpreter:

> (define test-fac3
  '((let (f)
    ((let (x) (let (y) ((f (x x)) y)))
     (let (x) (let (y) ((f (x x)) y)))))
   (let (fac)
     (let (x)
       (if (equal? x 1) 1 (* x (fac (- x 1))))))) 3))

> (evaluate test-fac3)
> 6

**NOTE:** Check out
~uli/cs515/examples/scheme/DefiningInterpreters/YY.ss on the ilab cluster.
Closure interpreters

- Interpreters are important tools to understand and specify the semantics of programming languages.

- Closures are an important concept for any language with functions as first order objects and static scoping. It gives us an idea what “price” we have to pay if we want first–order functions or a particular parameter passing style.

- So far, we only talked about “pure” functional languages without a store and assignment (no side effects). How to extend our interpreters to deal with a store?

  - Introduce a store as a function from addresses to values: \( store = Memory \text{ Locations} \rightarrow Values \)

  - Modify environments to map names to memory locations: \( \rho \in Env = Variables \rightarrow Memory \text{ Locations} \)

  - Modify \( ev \) to take one more argument:
    (define \( ev \) (lambda (e env store) \ldots)). In addition, \( ev \) will return a value and a store.

  - Of course, there is a lot more work to be done here, but I hope you get the idea.