**Announcements**

---

**Promises**

Delay Creates a Promise

From the Revised Report on the Algorithmic Language Scheme

\[(\text{delay} \ <\text{expression}>\)]

The delay construct is used together with the procedure force to implement lazy evaluation or call by need. \((\text{delay} \ <\text{expression}>\)) returns an object called a promise which at some point in the future may be asked (by the force procedure) to evaluate \(<\text{expression}>\), and deliver the resulting value...

\[(\text{force} \ <\text{promise}>\)]

Forces the value of promise...

\[(\text{force} \ (\text{delay} \ (+ \ 1 \ 2))) \Rightarrow 3\]

\[(\text{let} \ ((p \ (\text{delay} \ (+ \ 1 \ 2)))) \ (\text{list} \ (\text{force} \ p) \ (\text{force} \ p))) \Rightarrow \ (3 \ 3)\]

---

A Promise Can Be Represented as Function

A delayed expression can be captured along with the current environment using a lambda.

E.g., \((let \ ((p \ (\lambda \ () \ (+ \ 1 \ 2)))) \ (\text{list} \ (\text{force} \ p) \ (\text{force} \ p)))\)

(Demo)

\[(\text{force} \ (\text{delay} \ (+ \ 1 \ 2))) \Rightarrow 3\]

\[(\text{let} \ ((p \ (\text{delay} \ (+ \ 1 \ 2)))) \ (\text{list} \ (\text{force} \ p) \ (\text{force} \ p))) \Rightarrow \ (3 \ 3)\]

---

**Assignment and Caching**

Assignment in Scheme

The built-in set! special form changes the value of an existing variable.

\[\text{scm> (define x 2) \ x} \]

\[\text{scm> (set! x 3) \ okay} \]

\[\text{scm> x} \ 3\]

Local, non-local, and global assignment all use set!

\[(\text{define} \ (\text{sum} \ x \ y)) \ (\text{def} \ \text{sum}(x, y):) \ (\text{total} = 0) \ (\text{total} = \text{total} + x) \ (\text{iterate} = \text{iterate} + 1) \ (\text{return total})\]

\[(\text{force} \ (\text{delay} \ (+ \ 1 \ 2))) \Rightarrow 3\]

\[(\text{let} \ ((p \ (\text{delay} \ (+ \ 1 \ 2)))) \ (\text{list} \ (\text{force} \ p) \ (\text{force} \ p))) \Rightarrow \ (3 \ 3)\]

---

Force Caches the Promise Value

From the Revised Report on the Algorithmic Language Scheme

\[(\text{force} \ <\text{promise}>\)]

Forces the value of promise. If no value has been computed for the promise, then a value is computed and returned. The value of the promise is cached (or "memoized") so that if it is forced a second time, the previously computed value is returned.

\[\text{scm> (define x 2) \ x} \]

\[\text{scm> (let} \ ((p \ (\text{delay} \ (\text{set!} \ x \ (+ \ x \ 1)))))) \ (\text{begin} \ (\text{force} \ p) \ (\text{force} \ p)) \ \text{okay} \]

\[\text{scm> x} \ 3\]

\[\text{scm} \]
Caching Promise

Assignment is required in order to cache the value of a promise (from RS)

```scheme
(define make-promise
  (lambda (proc)
    (let ((result-ready? #f)
          (result #f))
      (lambda ()
        (if result-ready?
            result
            (begin
              (set! result-ready? #t)
              (let ((x (proc)))
                (if result-ready?
                    result
                    (set! result (x)))))�)))))
```

Takes a zero-argument lambda procedure with the delayed expression as its body

Returns a zero-argument lambda procedure that caches the value of proc

Evaluates proc and gives it a local name

Did (proc) get cached while evaluating (proc)?

If not, cache the value

---

Lazy Evaluation

When a procedure is applied:

- **Primitive**: The arguments are evaluated and the primitive procedure is applied to them
- **User-Defined**: All arguments are delayed

When an if expression is evaluated:

- **Predicate**: Must be fully evaluated to determine which sub-expression to evaluate next
- **Consequent/Alternative**: Is evaluated, but call expressions within it are eval’d lazily

(Demo)