Announcements
Promises
Delay Creates a Promise

From the Revised^5 Report on the Algorithmic Language Scheme

```
(delay <expression>)
```

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

```
(force <promise>)
```

Forces the value of promise...

```
(force (delay (+ 1 2))) ⇒ 3
(let ((p (delay (+ 1 2)))) (list (force p) (force p))) ⇒ (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)))) (list (p) (p)))`

(Demo)

`((force (delay (+ 1 2))) ⇒ 3)
(let ((p (delay (+ 1 2)))) (list (force p) (force p))) ⇒ (3 3)"
Assignment and Caching
Assignment in Scheme

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

```scheme
(scm> (define x 2)
 x
(scm> (set! x 3)
 okay
(scm> x
 3

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

```scheme
(define (sum a b)
 (let ((total 0))
   (define (iter x)
     (if (< x b)
       (begin
         (set! total (+ total x))
       (iter (+ x 1))))))
   (iter a)
   total))
```

```python
def sum(a, b):
    total = 0
    def iter(x):
        nonlocal total
        if x < b:
            total = total + x
            iter(x + 1)
    iter(a)
    return total
```
**Force Caches the Promise Value**

From the *Revised* Report on the Algorithmic Language Scheme

`forcem = promisem`

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.

```scm
define x 2
x
(let ((p (delay (set! x (+ x 1))))) (begin (force p) (force p)))
ok
x
3
```

```
Caching Promise

Assignment is required in order to cache the value of a promise (from R5RS)

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

- 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
Meta-Circular Evaluator
A Scheme Evaluator in Scheme

Lots of different programming languages can be expressed using nested combinations

• Scheme
• Scheme-syntax calculator
• Logic language (next week)
• The syntactic structure of an English sentence (demo)
• Variations of Scheme
Lazy Evaluator
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)