Lazy Evaluation

Announcements

Promises

Delay Creates a Promise

Delay Creates a Promise

From the **Revised^5 Report on the Algorithmic Language Scheme**

Delay Creates a Promise

From the **Revised^5 Report on the Algorithmic Language Scheme**

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

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...

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>*)

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...

Delay Creates a Promise

From the **Revised5 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

```
(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

```
(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)))`

```
(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

Assignment in Scheme

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

Assignment in Scheme

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

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

Assignment in Scheme

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

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

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

Assignment in Scheme

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

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

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

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

Assignment in Scheme

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

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

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

```
(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

Force Caches the Promise Value

From the **Revised^5 Report on the Algorithmic Language Scheme**

Force Caches the Promise Value

From the **Revised^5 Report on the Algorithmic Language Scheme**

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

Force Caches the Promise Value

From the **Revised^5 Report on the Algorithmic Language Scheme**

```
(force <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.

Force Caches the Promise Value

From the **Revised^5 Report on the Algorithmic Language Scheme**

(**force** *<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.

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

Caching Promise

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

Caching Promise

Assignment is required in order to cache the value of a promise (from R^{5}RS)

```
        (define make-promise
```

Caching Promise

Assignment is required in order to cache the value of a promise (from R^{5}RS)

```
(define make-promise
  (lambda (proc)
```

Caching Promise

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

```
(define make-promise
  (lambda (proc)
```

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

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)
```

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

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))
```

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

Caching Promise

Assignment is required in order to cache the value of a promise (from R^5RS)

```
(define make-promise
  (lambda (proc)
    (let ((result-ready? #f)
          (result #f))
         (lambda ()
```

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

Caching Promise

Assignment is required in order to cache the value of a promise (from R^{5}RS)

```
(define make-promise
  (lambda (proc)
    (let ((result-ready? #f)
          (result #f))
         (lambda ()
```

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

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?
```

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

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
```

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

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)))
```

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

Caching Promise

Assignment is required in order to cache the value of a promise (from R^{5}RS)

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

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

Caching Promise

Assignment is required in order to cache the value of a promise (from R^{5}RS)

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

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

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?
```

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)?

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
```

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)?

Caching Promise

Assignment is required in order to cache the value of a promise (from R^{5}RS)

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

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)?

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)
```

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)?

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)
```

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)?

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)
```

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

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

A Scheme Evaluator in Scheme

Lots of different programming languages can be expressed using nested combinations

A Scheme Evaluator in Scheme

Lots of different programming languages can be expressed using nested combinations

- Scheme

A Scheme Evaluator in Scheme

Lots of different programming languages can be expressed using nested combinations

- Scheme
- Scheme-syntax calculator

A Scheme Evaluator in Scheme

Lots of different programming languages can be expressed using nested combinations

- Scheme
- Scheme-syntax calculator
- Logic language (next week)

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)

A Scheme Evaluator in Scheme

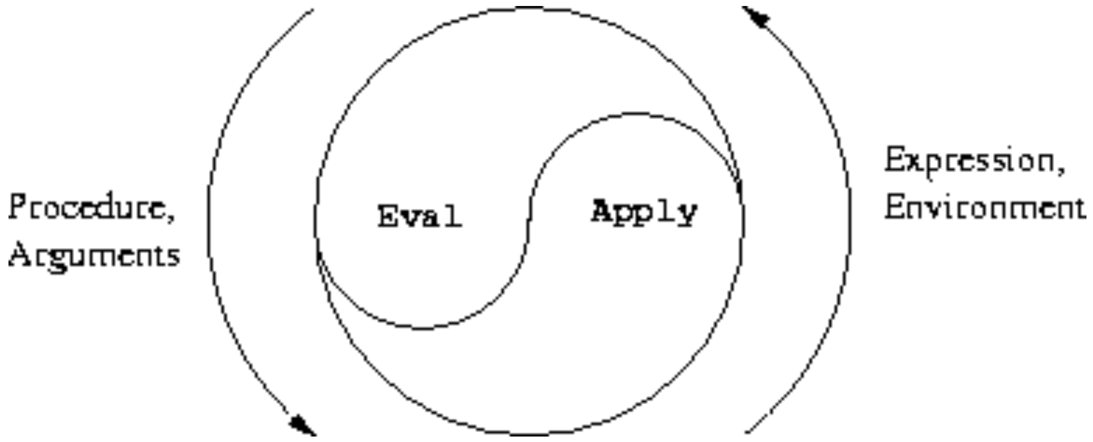
Lots of different programming languages can be expressed using nested combinations

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

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

Lazy Evaluation

```
When a procedure is applied:
```

Lazy Evaluation

When a procedure is applied:

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

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

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

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:

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

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

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)