6.001 SICP

- Normal (Lazy) Order Evaluation
- Memoization
- Streams

Applicative Order vs. Normal (Lazy) Order

\[
\text{define (foo x)} \\
\text{(write-line "inside foo")} \\
\text{(+ x x))} \\
\text{(foo (begin (write-line "eval arg") 222))} \\
\]

Applicative Order:
- eval arg
- inside foo
  \[
  \Rightarrow 444 \\
  \]

Normal (Lazy) Order:
- inside foo
- eval arg
  \[
  \Rightarrow 444 \\
  \]

We first evaluated argument, then substituted value into the body of the procedure. As if we substituted the unevaluated expression in the body of the procedure.

Exercise – Problem with Applicative Order

Why can't we use define to define any special form?

E.g. write a procedure `safe/` that only divide if \( b \neq 0 \)

\[
\text{(define (unless pred usual exc)} \\
\text{\{if pred exc usual\})} \\
\text{(define (safe/ a b)} \\
\text{\{unless (= b 0) (/ a b) 0\})} \\
\text{(define (safe/ a b)} \\
\text{\{cond \{ (= b 0) 0\}} \\
\text{\{else (/ a b)\})\})}
\]

Normal Order (Lazy) Evaluation

Alternative models for computation:
- Applicative Order:
  - evaluate all arguments, then apply operator
- Normal Order:
  - pass unevaluated expression to function
  - evaluate only when needed for primitive operation

Applicative vs. Normal?

\[
\text{(define (try a b)} \\
\text{(if (= a 0) 1 b))} \\
\text{(try 0 (/ 1 0))} \\
\text{(try 0 (pi-to-this-many-digits 10000000))}
\]

How can we implement lazy evaluation?

\[
\text{(define (l-apply procedure arguments env)} \\
\text{\{changed\}} \\
\text{\{apply-primitive-procedure procedure\}} \\
\text{\{\text{list-of-arg-values arguments env}\}\}) \\
\text{\{(compound-procedure? procedure\}} \\
\text{\{l-eval-sequence procedure-body procedure\}} \\
\text{\{extend-environment procedure-parameters procedure\}} \\
\text{\{list-of-delayed-args arguments env\}\}) \\
\text{\{procedure-environment procedure\})\}) \\
\text{\{else \{error "Unknown proc" procedure\})\})}
\]
Thunks – a delayed argument

- **Abstractly** – a thunk is a “promise” to return a value when later needed (“forced”)
- **Concretely** – our representation:

```
(delay-it exp env) – promise to eval exp later
(force-it exp) – eval exp now, leave no thunks
```

Thunks – delay-it and force-it

```
(define (delay-it exp env) (list 'thunk exp env))
(define (thunk? obj) (tagged-list? obj 'thunk))
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))

(define (force-it obj)
  (cond ((thunk? obj)
         (actual-value (thunk-exp obj)
                       (thunk-env obj))))
               (else obj)))

(define (actual-value exp env) (force-it (l-eval exp env)))
```

Exercise – Applicative versus Normal Order

- Write a function `normal-order?` that returns `#t` if the language is normal order and `#f` if the language is applicative order.

```
(define (normal-order?)
  (let ((x #t))
    ((lambda (x) '())
     (begin (set! x #f) '()))
    x))
```

Memo-izing evaluation

- In lazy evaluation an arg is reevaluate each time it is used
- In applicative order evaluation argument is evaluated once
- Can we keep track of values once we’ve obtained them, and avoid cost of reevaluation?

```
(define (evaluated-thunk? obj) (tagged-list? obj 'evaluated-thunk))
(define (thunk-value evaluated-thunk) (cadr evaluated-thunk))
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (actual-value (thunk-exp obj)
                                      (thunk-env obj))))
          (set-car! obj 'evaluated-thunk)
          (set-car! (cdr obj) result)
          (set-cdr! (cdr obj) '())
          result))
               ((evaluated-thunk? obj) (thunk-value obj))
               (else obj)))))
```

Laziness and Language Design

- We have a dilemma with lazy evaluation
  - Advantage: only do work when value actually needed
  - Disadvantages
    - not sure when expression will be evaluated; can be very big issue in a language with side effects
    - may evaluate same expression more than once
- Memoization doesn’t fully resolve our dilemma
  - Advantage: Evaluate expression at most once
  - Disadvantage: What if we want evaluation on each use?
- Alternative approach: give programmer control!
Variable Declarations: lazy and lazy-memo

- Handle lazy and lazy-memo extensions in an upward-compatible fashion:

  \[
  \text{\texttt{(lambda}} (a \text{ (b lazy)} \ c \text{ (d lazy-memo)}) \ldots \text{\texttt{)}}
  \]

  - "a", "c" are the usual kind of variables (evaluated before procedure application)
  - "b" is lazy, "Normal Order": it gets (re)-evaluated each time its value is actually needed
  - "d" is lazy-memo; it gets evaluated the first time its value is needed, and then that value is returned again any other time it is needed again.

Syntax Extensions – Parameter Declarations

(define (first-variable var-decls) (car var-decls))
(define (rest-variables var-decls) (cdr var-decls))
(define declaration? pair?)
(define (parameter-name var-decl) (if (pair? var-decl) (car var-decl) var-decl))
(define (lazy? var-decl) (and (pair? var-decl) (eq? 'lazy (cadr var-decl))))
(define (memo? var-decl) (and (pair? var-decl) (eq? 'lazy-memo (cadr var-decl))))

Controllably Memo-izing Thunks

- thunk – never gets memoized
- thunk-memo – first eval is remembered
- evaluated-thunk – memoized-thunk that has already been evaluated

When forced

\[
\text{thunk-memo} \quad \text{exp} \quad \text{env}
\]

\[
\text{evaluated-thunk} \quad \text{result}
\]

A new version of delay-it

- Look at the variable declaration to do the right thing...

(define (delay-it decl exp env)
  (cond ((not (declaration? decl))
         (l-eval exp env))
        ((lazy? decl) (list 'thunk exp env))
        ((memo? decl) (list 'thunk-memo exp env))
        (else (error "unknown declaration:" decl)))))

Change to force-it

(define (force-it obj)
  (cond ((thunk? obj) ; eval, but don’t remember it
         (l-eval exp thunk-env obj))
        ((memoized-thunk? obj) ; eval and remember
         (let ((result (actual-value (thunk-exp obj) (thunk-env obj))))
          (set-car! obj 'evaluated-thunk)
          (set-car! (cdr obj) result)
          (set-cdr! (cdr obj) '())
          (result)))
        ((evaluated-thunk? obj) (thunk-value obj))
        (else obj))))

Order Comparison

(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))

Applicative Order:
- eval arg
- inside foo

Normal (lazy) Order:
- eval arg
- inside foo

Memoized Normal (Lazy-memo) Order:
- eval arg
- inside foo

=> 444
=> 444
=> 444
Exercise
Given this definition of l-abs:
\[
\text{define} \ l-abs \ (\text{i lazy}) \ \text{(if} \ (< \text{i 0}) \ (- \text{i}) \ \text{i)}
\]
Trace the following use of l-abs:
\[
\text{(define (down) (begin (set! x (- x 2)) x)}
\text{(define x 3)}
\text{(l-abs (down))}
\]

Stream Object
• A pair-like object, except the cdr part is lazy
  (not evaluated until needed):
  \[
  \text{cons-stream} \\
  \text{a} \\
  \text{stream-car} \\
  \text{a} \\
  \text{value} \\
  \text{thunk-memo} \\
  \text{stream-cdr}
\]
\[
\text{define (cons-stream x (y lazy-memo))}
\text{(cons x y)}
\text{(define stream-car car)}
\text{(define stream-cdr cdr)}
\]

What will these print? (ex1)
\[
\text{(define s (cons 5 (begin (write-line 7) 9)))}
\text{(car s)}
\text{(cdr s)}
\text{(define t (cons-stream 5 (begin (write-line 7) 9)))}
\text{(stream-car t)}
\text{(stream-cdr t)}
\]

Decoupling computation from description
• Can separate order of events in computer from apparent order of events in procedure description
\[
\text{(list-ref}
\text{ (filter (lambda (x) (prime? x))}
\text{(enumerate-interval 1 100000000))}
\text{100)}
\text{(stream-ref}
\text{ (stream-filter (lambda (x) (prime? x))}
\text{(stream-interval 1 100000000))}
\text{100)}
\]

Decoupling computation from description
\[
\text{(define seq (stream-interval 1 10))}
\text{(define y (stream-filter even? seq))}
\text{Now! Lets do some calculation...}
\text{(stream-ref y 3) \rightarrow 8}
\]

Decoupling computation from description
\[
\text{(define (stream-interval a b)}
\text{(if (> a b)}
\text{the-empty-stream)}
\text{(cons-stream a (stream-interval (+ a 1) b))})
\text{(define (filter-stream pred str)}
\text{(if (pred (stream-car str))}
\text{(cons-stream (stream-car str))}
\text{(filter-stream pred (stream-cdr str))})
\text{(filter-stream pred (stream-cdr str))})
\]
Creating streams
There are two basic ways to create streams:
1. Build them up from scratch
2. Modify or combine existing streams

Creating infinite streams
(define ones (cons-streams 1 ones))
(define (add-streams s1 s2)
  (cons-stream (+ (stream-car s1) (stream-car s2))
               (add-streams
                (stream-cdr s1) (stream-cdr s2)))))
(define integers
  (cons-stream 1 (add-streams ones integers)))

Creating fibonacci streams (ex4)
Write a procedure, (fibs), that returns a stream of Fibonacci numbers. The Fibonacci series goes
(1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 ...)
It starts with two 1's, then each successive value is the sum of the previous two.

(define fibs
  (cons-stream
   0
   (cons-stream
    1
    (add-stream (stream-cdr fibs) fibs))))

Add-Streams == Map2
(add-streams integers integers) ==
(map2-stream + integers integers)
(define (map2-stream proc str1 str2)
  (cons-stream
   (proc (stream-car str1) (stream-car str2))
   (map-stream2 proc
                (stream-cdr str1) (stream-cdr str2)))))