CS61A Lecture 22

2011-07-27

Colleen Lewis

No 3 person teams for Project 4

Partner declaration (1 point):
due Friday July 29th at 11:59 pm
https://spreadsheets.google.com/spreadsheet/viewform?formkey=dFdGYWN5b3NWNFZITG1jMnZMaDJtSXc6MQ

Part 1: due Saturday August 6th at 11:59 pm
Part 2: due Monday August 8th at 11:59 pm
(MUCH longer Part 1)

Proj 4 Part II is LONGER than Part I

7	8/1 M		Metacircular evaluator (BH Lect 12)
	8/2 Tu		Analyzing evaluator (BH Lect 13)
	8/3 W	HW 12 due at 11:59pm soln	Lazy evaluator (BH Lect 14)
	8/4 Th		Logic programming (BH Lect 15)
	8/6 Sa	Review Session 1-4pm 306 Soda	
	8/6 Sa	Project 4 - Part 1 due at 11:59pm soln	
8	8/8 M	Project 4 - Part 2 due at 11:59pm soln	Alan Kay: User Interfaces (1) (2)
	8/9 Tu	HW 13 due at 11:59pm soln	Therac
	8/10 W		Review
	8/11 Th	**Final Exam** 7-10pm 155 Dwinelle	Tips for jobs and grad school

`delay` special-form

This creates a promise

It doesn't evaluate its arguments

```
(delay exp)
```

Almost like:

```
(lambda () exp)
```

This won't error!

```
STk> (delay (/ 5 0))
```

`delay` is *almost* like wrapping the expression in a thunk

```
(define promise1 (delay (/ 5 0))

(define promise2 (lambda () (/ 5 0))
```

Although they are different

```
STk> (delay (/ 3 0))
#[promise 7ff1aa18 (not forced)]
STk> (lambda () (/ 3 0))
#[closure arglist=() 7ff1b3d8]
```

`force`

```
(define promise2 (lambda () (+ 2 3))

STk> (force promise2)
5
```

Note: `delay` creates a promise and not a thunk

What definition works?

```
A. (define (force promise) promise)
B. (define (force promise) (promise))
C. (define (force promise) ((promise)))
```

Promises remember if they've ever been `forced`

```
STk> (define p (delay (+ 2 3)))
p
STk> p
#[promise 7ff0e9e8 (not forced)]
STk> (force p)
5
STk> p
#[promise 7ff0e9e8 (forced)]
```

This is how a promise is different than a thunk

Thunks remember their env!

```
(define (apple x)
  (lambda () (+ x x)))
```

Write without synt. sugar & draw the env. diagram

Which one returns 4?

A. `(apple 2)`
B. `((apple 2))`
C. `((apple) 2))`
D. ??
E. None

Promises remember their env!

```
(define (plum x)
  (delay (+ x x)))
```

Write without syntactic sugar

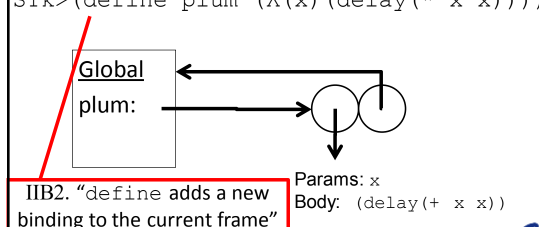
Which one returns 4?

A. `(force plum 2)`
B. `(force (plum 2))`
C. `(force (plum) 2))`
D. `(force ((plum 2)))`
E. None

IIB1. "`lambda` creates a procedure"

- Left bubble points to the formal parameters & body
- Right bubble points to the current environment

```
STk>(define plum (λ(x)(delay(* x x))))
```



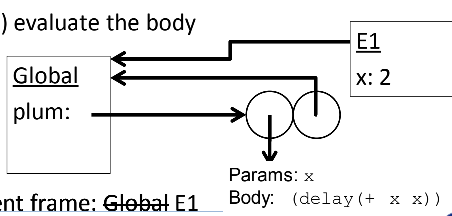
Current frame: Global

Procedure Call

IIA Step1. "evaluate the arguments"

IIA Step2. `STk>(define p (plum 2))`

- (a1) draw a frame
- (a1) bind the formal parameters
- (a2) extend environment the R bubble points to
- (a3) evaluate the body

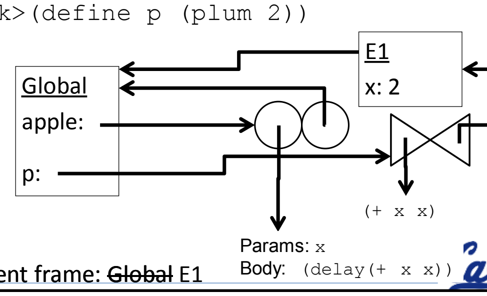


Current frame: ~~Global~~ E1

"`delay` creates a promise"

- Left triangle points to the thing that is delayed
- Right triangle points to the current environment

```
STk>(define p (plum 2))
```



Current frame: ~~Global~~ E1

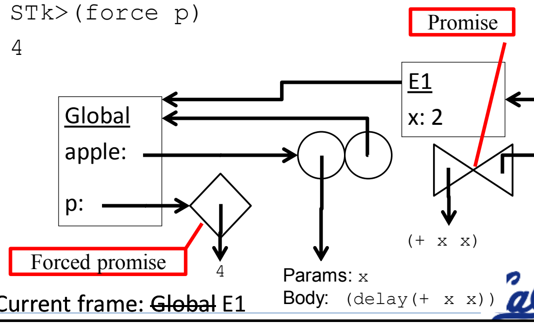
Promises remember if they've ever been `forced`

```
STk> (define p (plum 2))
p
STk> p
#[promise 7ff0e9e8 (not forced)]
STk> (force p)
4
STk> p
#[promise 7ff0e9e8 (forced)]
```

This is how a promise is different than a thunk

"`delay` creates a promise"

```
STk>(define p (plum 2))
STk>(force p)
4
```



Promises SUMMARY

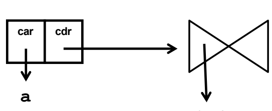
- Are created using `delay`
  - `delay` is a special form
- Are sort-of like delaying execution with a thunk
- Promises remember their environment
- We can force a promise using `force`
- Once we call `force` on a promise we remember the result, it becomes a forced promise.

Streams

`cons-stream`

Is cons-stream a special form?
A) Yes B) No

```
STk> (cons-stream 'a (+ 2 3))
(a . #[promise 7ff23758 (not forced)])

STk> (cons 'a (delay (+ 2 3)))
(a . #[promise 7ff24188 (not forced)])
```



Draw a picture of banana

```
STk> (define banana
        (cons-stream 'a
                (cons-stream 'b 'c)))
banana
```

What type of thing are the `car` and `cdr` of `banana`?

```
A.car & cdr: promises        E. ??
B.car & cdr: words
C.car: word cdr: promise
D.car: word cdr: pair
```

Draw the box-and-pointer & wwsp

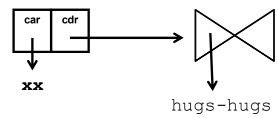
```
STk> (define hugs-kisses (cons 'xo 'xo))
hugs-kisses

STk> hugs-kisses
(xo . xo)

STk> (set-cdr! hugs-kisses hugs-kisses)
Okay

STk> hugs-kisses
_________________________________
```

Are there dots in your answer? A)Yes B)No

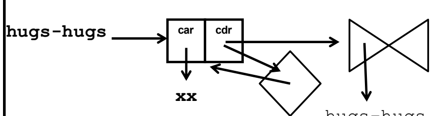
xo

Using cons-stream

```
STk> (define hugs-hugs
        (cons-stream 'xx hugs-hugs))
hugs-hugs

STk> hugs-hugs
(xx . #[promise 7ff67128 (not forced)])
```




```
STk> (cdr hugs-hugs)
#[promise 7ff5eb28 (not forced)]

STk> (force (cdr hugs-hugs))
(xx . #[promise 7ff5eb28 (forced)])

STk> hugs-hugs
(xx . #[promise 7ff5eb28 (forced)])
```



Stream selectors

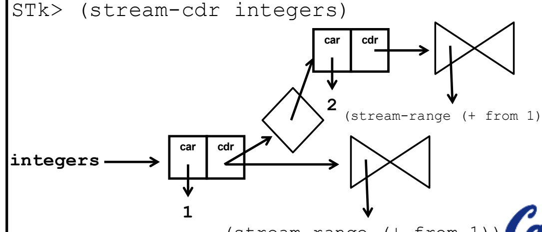
```
(define (stream-car streams)
    (car stream))

(define (stream-cdr stream)
    (force (cdr stream)))
```

Draw a picture

```
(define (stream-range from)
  (cons-stream from
        (stream-range (+ from 1))))

(define integers (stream-range 1))
```

(define (stream-cdr stream) (force (cdr stream)))

```
(define (stream-range from)
  (cons-stream from
         (stream-range (+ from 1))))
STk> (stream-cdr integers)
```



All prime numbers in the world!

Trapped inside your computer!

prime?

```
(define (prime? n)
  (define (prime-iter? factor)
    (cond
     ((= factor n) #t)
     ((= (remainder n factor) 0) #f)
     (else (prime-iter? (+ factor 1)))))
  (trace prime-iter?)
  (prime-iter? 2))
```

Trace (prime? 5)

```
STk> (prime? 5)
.. -> prime-iter? with factor = 2
.... -> prime-iter? with factor = 3
...... -> prime-iter? with factor = 4
........ -> prime-iter? with factor = 5
........ <- prime-iter? returns #t
...... <- prime-iter? returns #t
.... <- prime-iter? returns #t
.. <- prime-iter? returns #t
#t
```

Trace (prime? 9)

```
STk> (prime? 9)
.. -> prime-iter? with factor = 2
.... -> prime-iter? with factor = 3
.... <- prime-iter? returns #f
.. <- prime-iter? returns #f
#f
```

prime? Version 2 (with HOFs)

```
(define (prime? n)
  (null?
   (filter
    (lambda (x)
      (= (remainder n x) 0))
    (range 2 (- n 1)))))
```

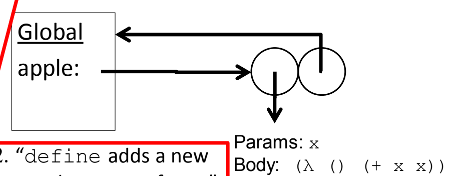
prime? Version 2 (with HOFs)

```
(define (prime? n)
  (stream-null?
   (stream-filter
    (lambda (x)
      (= (remainder n x) 0))
    (stream-range 2))))
```

IIB1. "`lambda` creates a procedure"

- Left bubble points to the formal parameters & body
- Right bubble points to the current environment

```
STk>(define apple(λ(x)(λ()(* x x))))
```



IIB2. "define adds a new binding to the current frame"

Current frame: Global

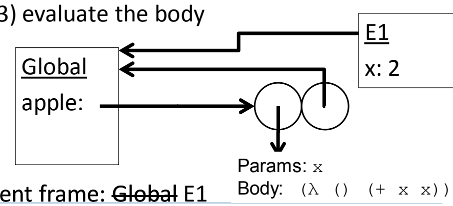
Procedure Call

IIA Step1. "evaluate the arguments"

IIA Step2.

```
STk>(apple 2)
```

- (a1) draw a frame
- (a1) bind the formal parameters
- (a2) extend environment the R bubble points to
- (a3) evaluate the body

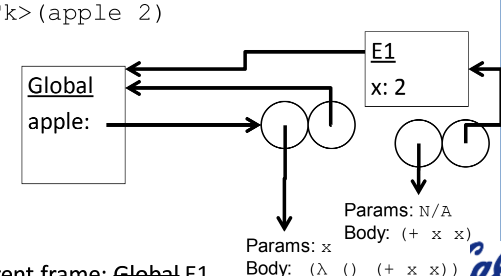


Current frame: ~~Global~~ E1

IIB1. "`lambda` creates a procedure"

- Left bubble points to the formal parameters & body
- Right bubble points to the current environment

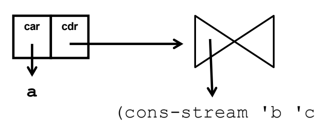
```
STk>(apple 2)
```



Current frame: ~~Global~~ E1

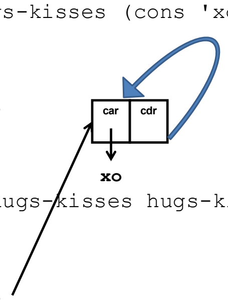
Draw a picture of banana

```
STk> (define banana
       (cons-stream 'a
             (cons-stream 'b 'c)))
banana
```



Draw the box-and-pointer & wwsp

```
STk> (define hugs-kisses (cons 'xo 'xo))
hugs-kisses

STk> hugs-kisses
(xo . xo)

STk> (set-cdr! hugs-kisses hugs-kisses)
Okay

STk> hugs-kisses
(xo xo xo xo xo xo xo xo xo ...)
```



Draw a picture

```
(define (stream-range from)
  (cons-stream from
       (stream-range (+ from 1))))

(define integers (stream-range 1))
```