PP Lecture 3
2020

Learning Goals

- Simulation of OOP with functions
- Handling of imperative language mechanisms in relation to functional programming
- Understanding and use of continuations
  - Including continuation passing style
- Delayed evaluation with trampolining

The plan for this live stream

- Something about OOP simulation
- Working with objects in Scheme
- Flexible parameters
- Something about continuations
- Programming as language development
- Exercise intro

Questions and answers – mediated by Jonas Hansen

OOP simulation

```
(define (send message obj . par)
  (let ((method (obj message)))
    (apply method par)))

(define (point x y)
  (letrec ((getx    (lambda () x))
           (gety    (lambda () y))
           (add     (lambda (p)
                      (point
                       (+ x (send 'getx p))
                       (+ y (send 'gety p)))))
           (type-of (lambda () 'point))
          )
    (lambda (message)
      (cond ((eq? message 'getx) getx)
            ((eq? message 'gety) gety)
            ((eq? message 'add)  add)
            ((eq? message 'type-of) type-of)
            (else (error "Message not understood"))))))
```

Enhance our understanding of *closures*

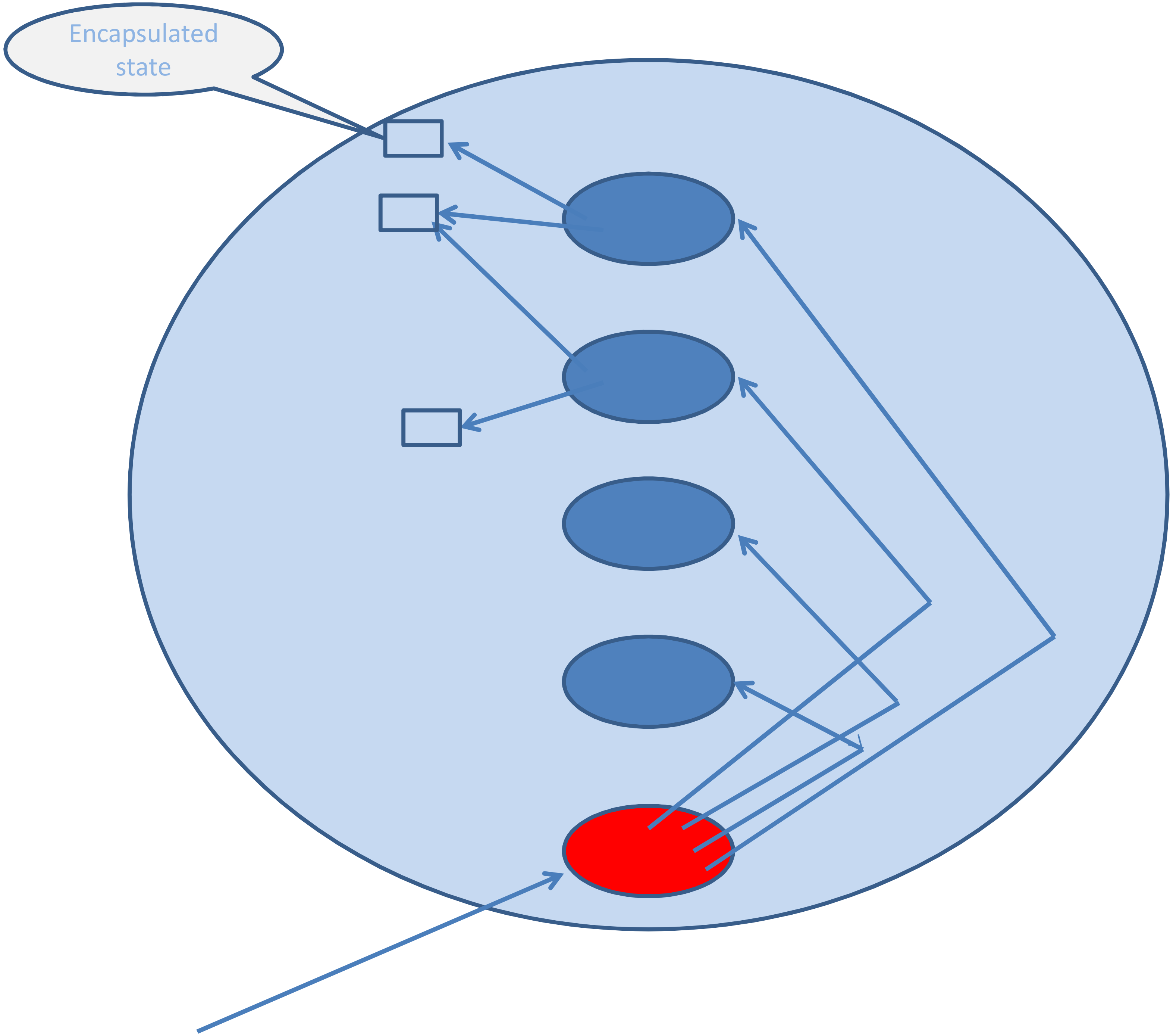
Define `send` before `point` in Racket

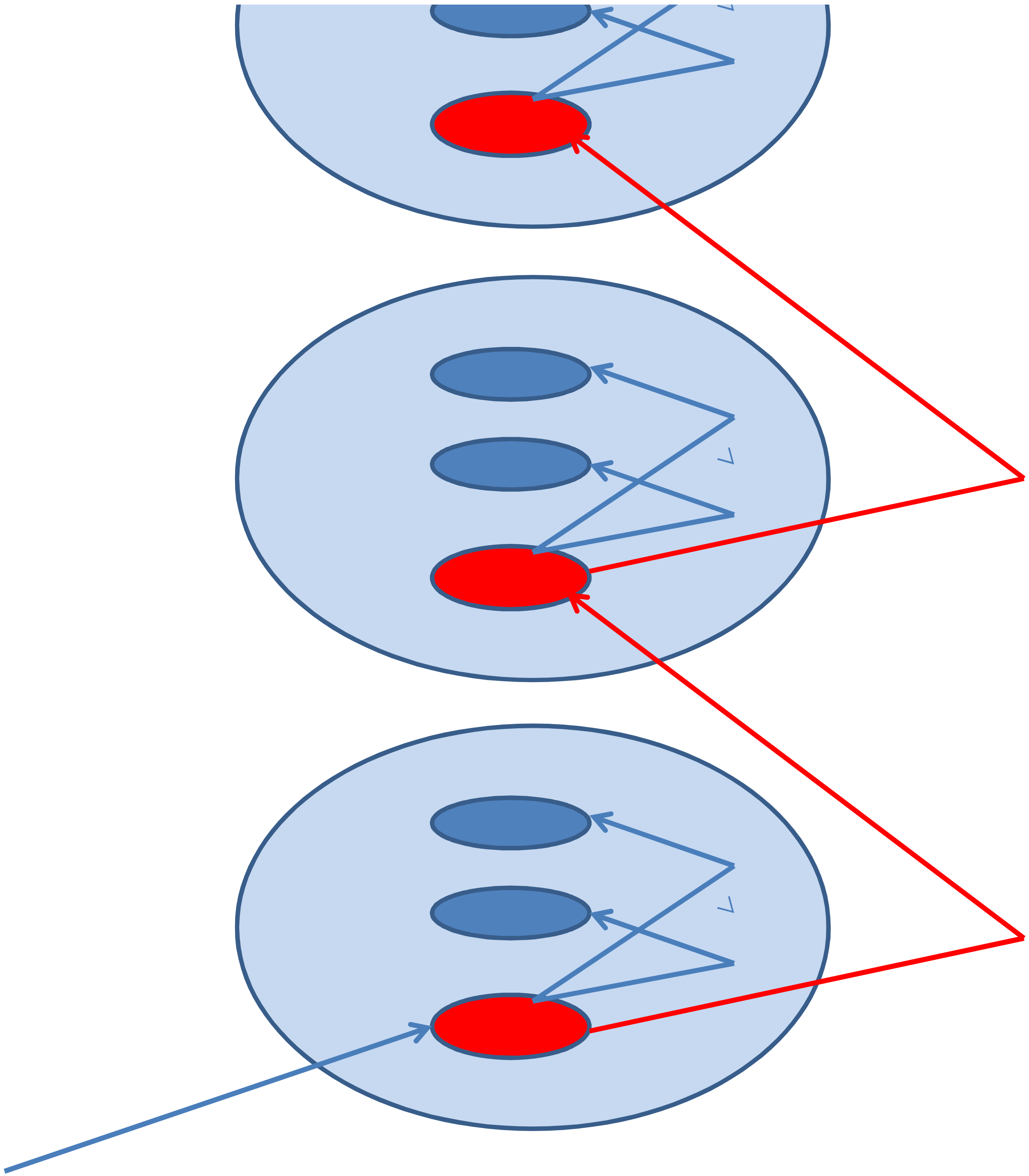
```
(define (point x y)
  (letrec ((getx    (lambda () x))
           (gety    (lambda () y))
           (add     (lambda (p)
                      (point
                       (+ x (send 'getx p))
                       (+ y (send 'gety p)))))
           (type-of (lambda () 'point))
          )
    (lambda (message)
      (cond ((eq? message 'getx) getx)
            ((eq? message 'gety) gety)
            ((eq? message 'add)  add)
            ((eq? message 'type-of) type-of)
            (else (error "Message not understood"))))))
```

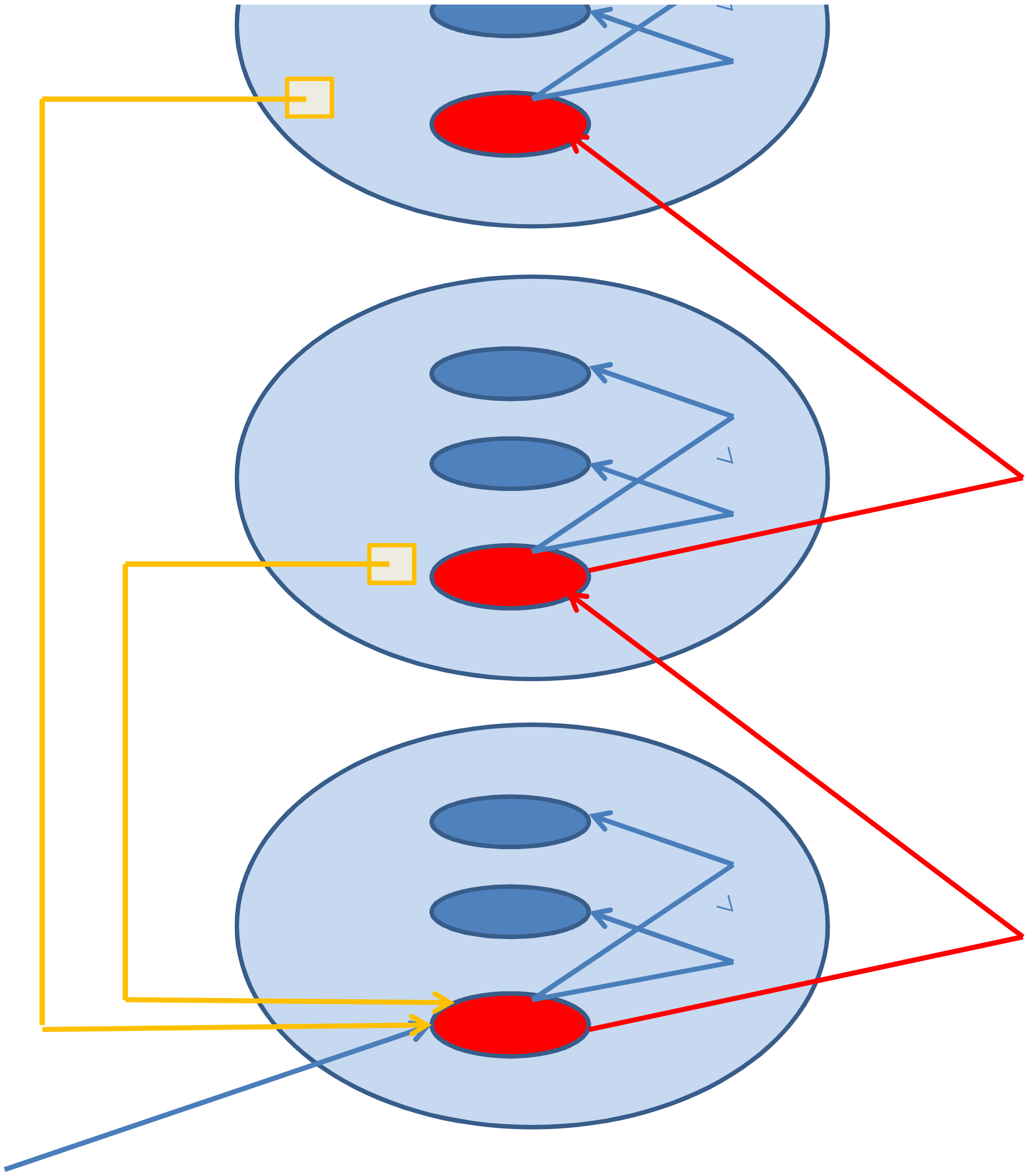
```
(define (point x y)
 (let ((x x)
       (y y))

   (define (getx) x)

   (define (gety) y)

   (define (add p)
    (point
     (+ x (send 'getx p))
     (+ y (send 'gety p))))

   (define (type-of) 'point)

   (define (self message)
     (cond ((eqv? message 'getx) getx)
           ((eqv? message 'gety) gety)
           ((eqv? message 'add)  add)
           ((eqv? message 'type-of) type-of)
          (else (error "Message not understood" message))))

   self))
```

```scheme
(define (new-instance class . parameters)
  (apply class parameters))

(define (send message object . args)
  (let ((method (method-lookup object message)))
    (cond ((procedure? method) (apply method args))
          (else (error "Error in method lookup " method)))))

(define (method-lookup object selector)
  (cond ((procedure? object) (object selector))
        (else
          (error "Inappropriate object in method-lookup: "
                 object))))
```

REPL SESSIONS:
naked-point.scm and point.scm


Encapsulated state





Working with *objects* in Scheme

Objects in a broad sense

Objects as lists

```
(point 5 -10)
```

- Just lists, not alists
  - First element is a tag
- OOP inspiration:
  - Constructor, predicate, selectors

List-based representations are flexible

It is easy to add additional elements, for the purpose of future program modifications.

REPL SESSION: list-based-objects.scm

Objects as association lists

- Association lists
  - A flexible approach
  - Updates by shadowing
  - Constructor functions
  - Predicate
  - Selector functions

```
(
 (type . point)
 (x . 5)
 (y . -10)
)
```

```
(
 (x . 17)
 (type . point)
 (x . 5)
 (y . -10)
)
```

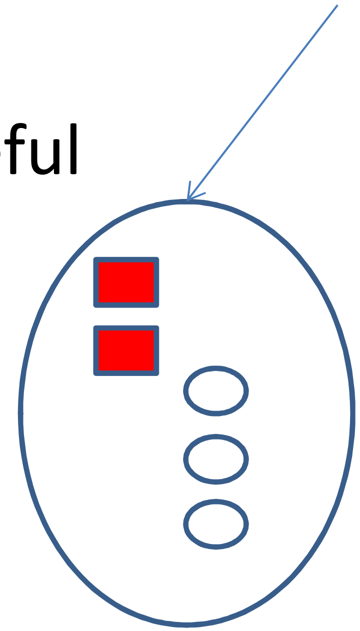
A kind of dictionary – mapping keys to values

Avoid car/cdr programming

- With constructors, predicates and selector functions:
  - Only these functions know about the list representation.
  - Only these functions use car/cdr
- Without:
  - Hard to change your mind wrt. Representation
  - All parts of your program will be polluted with car/cdr.

Objects as closures

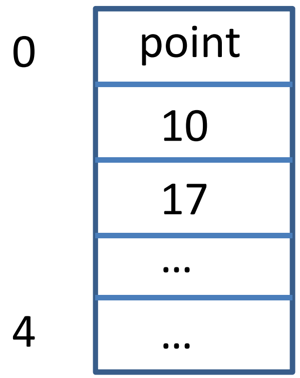
- Representation of objects as closures
  - Local variables in let-bindings inside a closure
- Interesting in principle
  - But it can be used for real-life purposes
  - Troublesome in several respects
  - Better language support would be useful
    - Syntactically
    - Implementation-wise



Objects as structs

- Structs are not part of R5RS
  - But they exist in R6RS and Racket
  - SRFI 9 defines a portable record type
    - May rely on R5RS vectors
    - Vectors are heterogeneous in Scheme
      - Of fixed lengths – cannot easily be enlarged
      - Mapping of field names to indexes

type → 0
x → 1
y → 2



Objects in the PP1 Mini Project

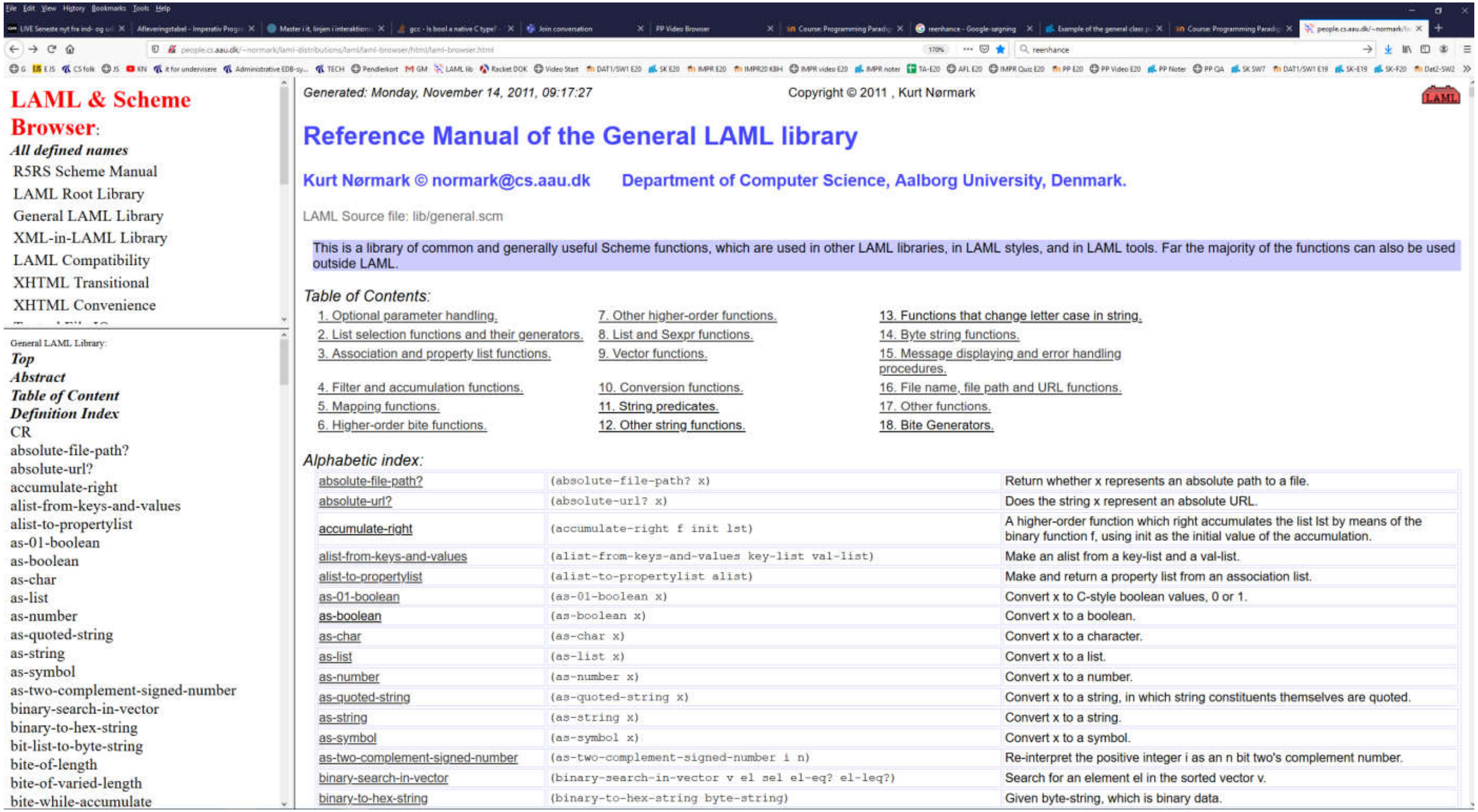
- Representation of student, group and grouping
- Be sure to encapsulate your representation decisions
- List-based representations are natural in a list-based language…

You may start on this part of the mini project already today

Flexible parameters
- the use of rest parameters

REPL session: flexible-parameters

The LAML libraries



LAML & Scheme Browser:
All defined names
R5RS Scheme Manual
LAML Root Library
General LAML Library
XML-in-LAML Library
LAML Compatibility
XHTML Transitional
XHTML Convenience

General LAML Library:
Top
Abstract
Table of Content
Definition Index
CR
absolute-file-path?
absolute-url?
accumulate-right
alist-from-keys-and-values
alist-to-propertylist
as-01-boolean
as-boolean
as-char
as-list
as-number
as-quoted-string
as-string
as-symbol
as-two-complement-signed-number
binary-search-in-vector
binary-to-hex-string
bit-list-to-byte-string
bite-of-length
bite-of-varied-length
bite-while-accumulate

Generated: Monday, November 14, 2011, 09:17:27 Copyright © 2011 , Kurt Nørmark

LAML

Reference Manual of the General LAML library

Kurt Nørmark © normark@cs.aau.dk Department of Computer Science, Aalborg University, Denmark.

LAML Source file: lib/general.scm

This is a library of common and generally useful Scheme functions, which are used in other LAML libraries, in LAML styles, and in LAML tools. Far the majority of the functions can also be used outside LAML.

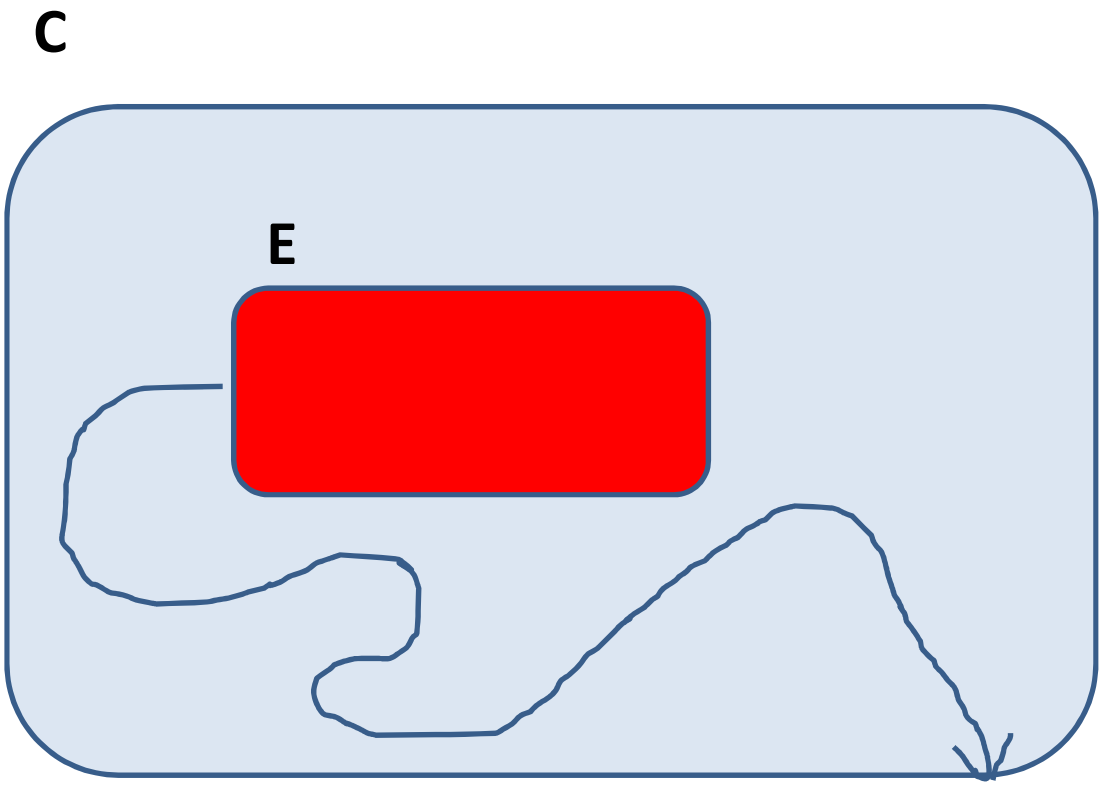
Table of Contents:

1. Optional parameter handling.
2. List selection functions and their generators.
3. Association and property list functions.
4. Filter and accumulation functions.
5. Mapping functions.
6. Higher-order bite functions.
7. Other higher-order functions.
8. List and Sexpr functions.
9. Vector functions.
10. Conversion functions.
11. String predicates.
12. Other string functions.
13. Functions that change letter case in string.
14. Byte string functions.
15. Message displaying and error handling procedures.
16. File name, file path and URL functions.
17. Other functions.
18. Bite Generators.

Alphabetic index:

absolute-file-path?	`(absolute-file-path? x)`	Return whether x represents an absolute path to a file.
absolute-url?	`(absolute-url? x)`	Does the string x represent an absolute URL.
accumulate-right	`(accumulate-right f init lst)`	A higher-order function which right accumulates the list lst by means of the binary function f, using init as the initial value of the accumulation.
alist-from-keys-and-values	`(alist-from-keys-and-values key-list val-list)`	Make an alist from a key-list and a val-list.
alist-to-propertylist	`(alist-to-propertylist alist)`	Make and return a property list from an association list.
as-01-boolean	`(as-01-boolean x)`	Convert x to C-style boolean values, 0 or 1.
as-boolean	`(as-boolean x)`	Convert x to a boolean.
as-char	`(as-char x)`	Convert x to a character.
as-list	`(as-list x)`	Convert x to a list.
as-number	`(as-number x)`	Convert x to a number.
as-quoted-string	`(as-quoted-string x)`	Convert x to a string, in which string constituents themselves are quoted.
as-string	`(as-string x)`	Convert x to a string.
as-symbol	`(as-symbol x)`	Convert x to a symbol.
as-two-complement-signed-number	`(as-two-complement-signed-number i n)`	Re-interpret the positive integer i as an n bit two's complement number.
binary-search-in-vector	`(binary-search-in-vector v el sel el-eq? el-leq?)`	Search for an element el in the sorted vector v.
binary-to-hex-string	`(binary-to-hex-string byte-string)`	Given byte-string, which is binary data.

Continuations



A continuation of an expression E in a contextual expression C is the future/rest of the computation C, which waits for (and depends on) the value of E

(lambda (E))

```
(let ((x 5)
      (y 4))
  (if (even? x)
      (+ (* x x) (* y y))
      (+ (* x x) (call/cc
                  (lambda (e)
                    (+ (* y y) (e y) x))))))
```

The value is 29.

Why?

REPL: continuations.scm

```
(define (list-length l)
  (call-with-current-continuation
   (lambda (do-exit)
     (letrec ((list-length1
                (lambda (l)
                   (cond ((null? l) 0)
                         ((pair? l)
                            (+ 1 (list-length1 (cdr l))))
                         (else (do-exit 'improper-list))))))
       (list-length1 l)))  ))
```

Continuation Passing Style

```
(define (f a b c d k)
  (k (+ a b c d)))

(define (g a k)
  (k (* a a)))

(define (h a k)
  (k (* a a a)))

(define (w a b k0)
  (g a (lambda(v1)
         (h b (lambda (v2)
                (g b (lambda (v3)
                       (h a (lambda (v4)
                              (f v1 v2 v3 v4 k0))))))))))
```

Direct Style

```
(define (f a b c d)
  (+ a b c d))

(define (g a)
  (* a a))

(define (h a)
  (* a a a))

(define (w a b)
 (f (g a) (h b) (g b) (h a)))
```

Continuation Passing Style Interesting Observations

- Functions written in CPS are always tail recursive
- Tail recursive in CPS functions do not need to contruct new continuations functions
- Functions written in CPS do not need 'the magic primitive' call-with-current-continuation
- Functions written in CPS are specific about the evaluation order of sub-expressions
- Functions in CPS are typically automatically translated from functions in direct style
- Functions written in CPS are subject to trampolining

The use of continuations in Scheme Programming

- Trickery one-liners
- Exit from deep expressions
  - list-length of improper lists
  - find-in-tree programmed with for-each
- Advanced control flow
  - Producer consumer funcitons
  - Simultaneous traversal of two binary trees
  - Coroutines

Programming as language development

- Your own functions are similar to the native functions in the language
- Your own syntactic extensions are similar to the special forms
- As part of your solution, you may implement an interpreter for a little language
- Ultimately, you may play with a meta circular interpreter
  - Scheme in Scheme

About the exercises

- *There are intros and outros to all exercises apart from the last one*

- Two exercises about simulated OOP with closures
- One about call-with-current-continuation
- One about continuation passing style
- One about trampolining

Exercise 3.1

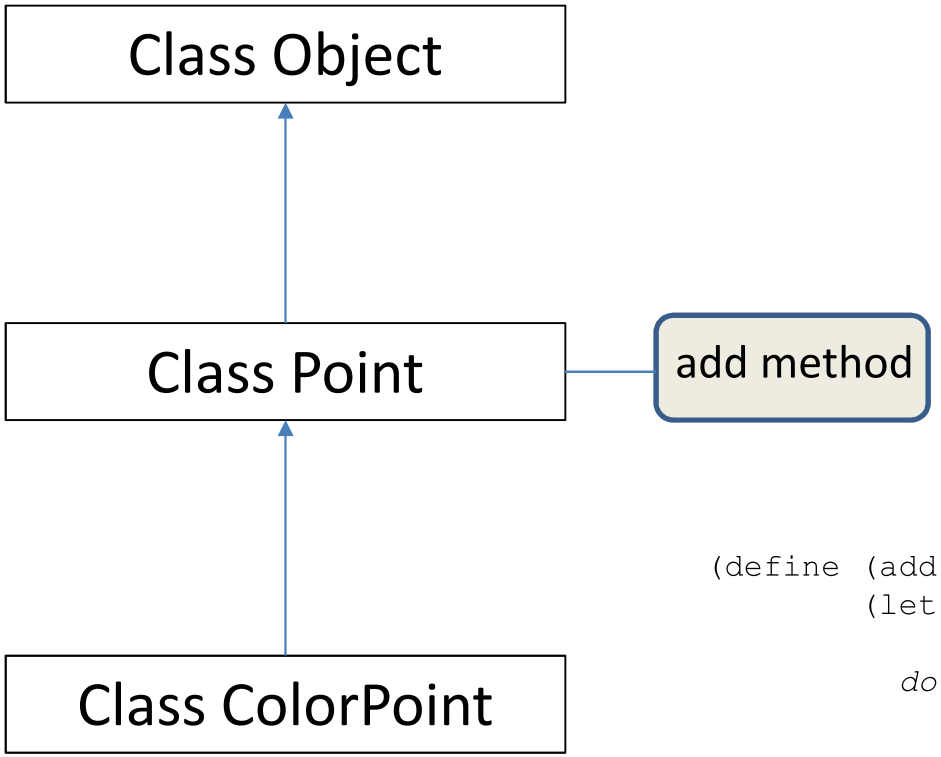
Class Point and class Rectangle

- A move method in class point
- A new class rectangle
  - Represented by points
  - move and area method

Exercise 3.2

Class Point and subclass ColorPoint.
The add method should get the class of the point
... and instantiate a new point of that class

- Almost trivial methods class-of in point and color-point
- Modifying the add method, such that it instantiates the most class
  - With a single problem, perhaps




```
(define (add p)
        (let ((class-to-instantiate
                 class of receiver))
          do the instantiation of the point sum)))
```

```
(send 'add cp (new-instance color-point 1 2 'green))
```

Exercise 3.7

- An exercises about
  - call-with-current-continuation
- Understanding an expression that
  - captures a continuation
  - uses the continuation

Exercise 3.4

- An expression that trains you to rewrite simple expressions to *continuation passing style*

Exercise 3.6

- Not all of you get to this exercise
- About a refined sessaw trampolining function

Start to think about your work on the PP1 miniproject

- Representation of students, groups and groupings
- Constructors
  - Students
  - Groups
  - Groupings
- Selectors and predicates

PP from 10:15 – 12:00

- Uninterrupted exercises in groups
- Enter your group channel in the Programming Paradigms team
  - **Start a meeting** in the channel
  - We will drop into the meeting and discuss PP topics
- Ask for help in the general channel:
  - Group @<group-id> needs help about ...