PP Lecture 4
E20

Plan for today

- 9-10
  - Discussion of evaluation order and streams
  - LAML experience
  - Exercise intro
- 10-12
  - Stream exercises
  - The first PP miniproject

Learning Goals

- Evaluation-order
  - Applicative and normal-order evaluation
- Streams in Scheme
  - Applications of delay and force

Evaluation order

```
(f
   (g a b)
   (h c d e)
)
```

- In Scheme:
  - Applicative order
  - Eagerly evaluate all parameters to functions before the function call
- In Haskell:
  - Normal order
  - Lazily postpone the evaluation of parameters until they are needed

Evaluation order does not matter as long as no errors occur and all subexpressions terminate

Delaying an evaluation

- It is easy to delay the evaluation of any expression

```
E
```

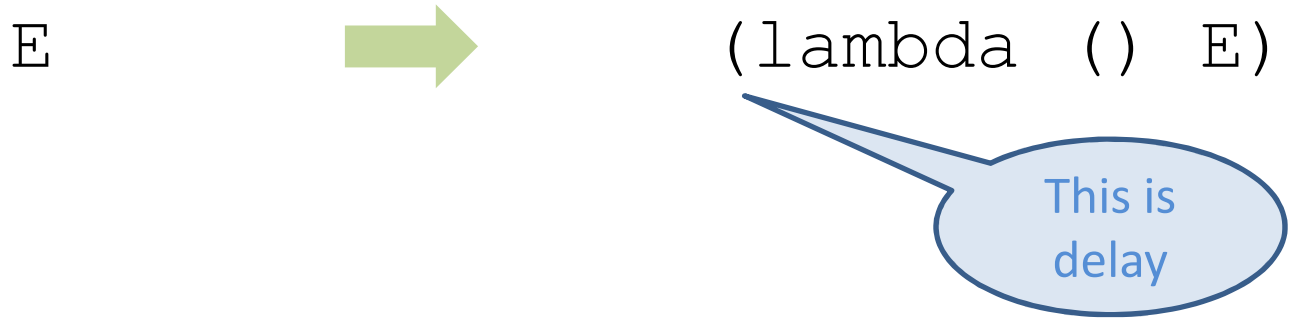


```
(lambda () E)
```

- The primary mechanism in trampolining
- The mechanism behind infinite lists (streams)

Delaying an evaluation

- It is easy to delay the evaluation of any expression



- The primary mechanism in trampolining
- The mechanism behind infinite lists (streams)

Promises

- A promise is the result of delaying the evaluation of an expression

```
(delay x) ~
(lambda () x)
```

```
(define (force promise)
   (promise))
```

`(delay x)` *creates a promise p*

`(force p)` *fulfils the promise*

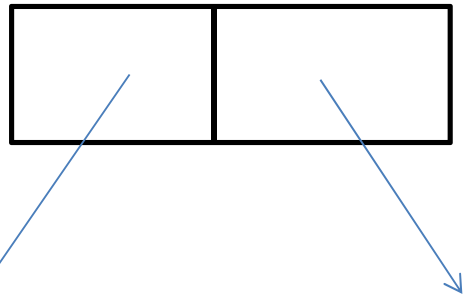
REP session: delaying-basics.scm

Infinite lists (streams)

- A variant of cons that delays the evaluation of the list tail
  - Implemented as a syntactic abstraction
  - A delay in terms of a *promise* (thunk)

```
( element . promise )
```

```
(cons-stream x y) ~
(cons x (delay y))
```



element

(lambda () *rest-of-stream*)

Infinite lists (streams)

- Recursive functions without a base-case

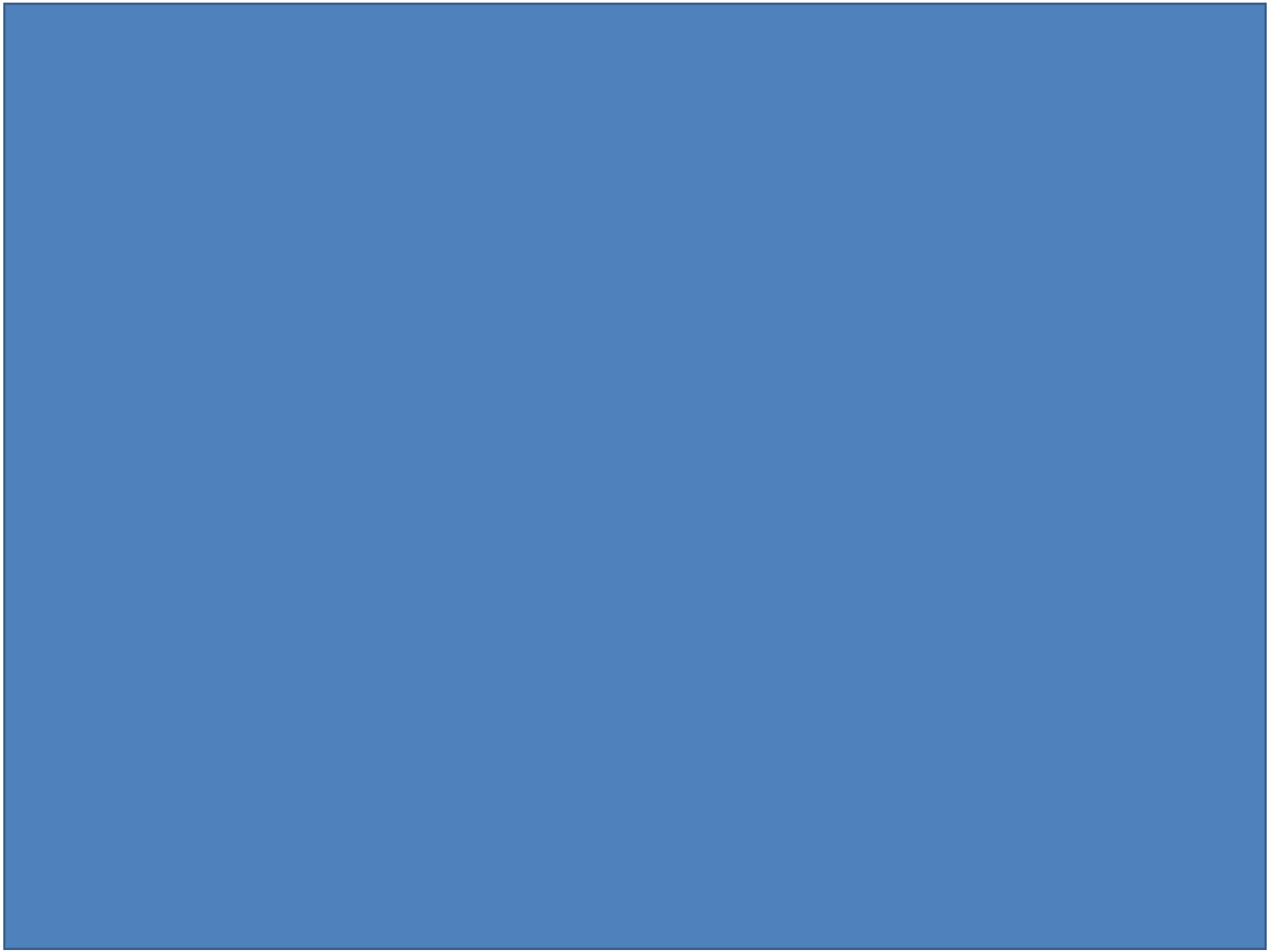
```
(define (integers-starting-from n)
   (cons-stream n (integers-starting-from (+ n 1))))
```

- Recursive definitions of streams

```
(define ones (cons-stream 1 ones))

(define nat-nums
   (cons-stream 1
                (add-streams ones nat-nums)))
```

REP session: stream-basics.scm



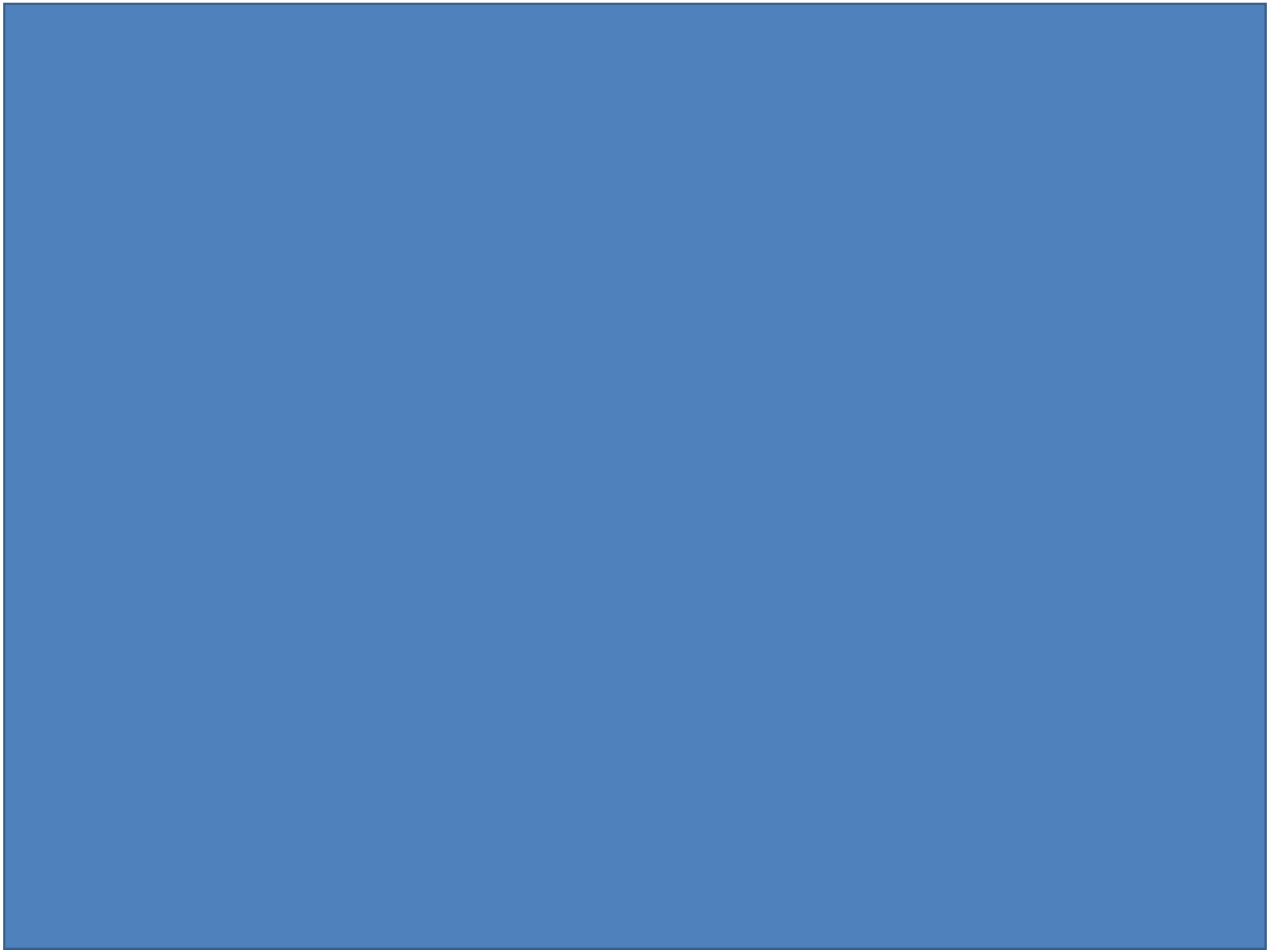
LAML Experience

Markup languages
in Scheme

LAML Overview

- LAML: Lisp Abstracted Markup Language
- For information and inspiration
  - A web page produced by html
- LENO slides
  - ”Working with XML in Scheme via LAML”

c:/Users/Kurt/courses/PP-E20/Auditorium-activities/PP3/LENO/xml-scheme-laml.html



Exercises

- As usual
  - Exercise intro video
  - Exercise outro video

Group exercises from 10:15 - 12

- Exercise 4.2
  - A stream of all factorial numbers
- Exercise 4.4
  - Given a real number *x*
  - A stream of numbers that appromimates (sqrt *x*)
- Exercise 4.3
  - Stream append and stream merge
- **Exercise 1.15**
  - **The PP1 miniproject**

Factorial stream

- Start with the number 1
- The rest should be formed as
  - A stream mulitiplication of
    - The fact stream and *some other stream*

Finding Square Root

- $f(x) = x^2 - a, f'(x) = 2x$
- When $f(x) = 0$ then $x^2 = a$ and $x = \sqrt{a}$
- Plug into $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
- $x_{n+1} = x_n - \frac{x_n^2 - a}{2x_n}$
- $x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$

```
(define (improve-sqrt-guess guess a)
  (/ (+ guess (/ a guess)) 2))
```

```
sqrt(2.0):

double g = 1.0;
while (! done){
   g = improve_sqrt_guess(g, 2.0);
   done = …
}
```

stream-append
stream-merge

- Is it possible to program

```
(stream-append stream-1 stream-2)
```

- Program a stream-merge function

```
(stream-merge stream-1 stream-2)
```

- Use stream-merge to make a stream of all integers (in some order).

```
(define (integers-with-lower-limit low)
  (cons-stream low (integers-with-lower-limit (+ low 1))))

(define (integers-with-upper-limit upper)
  (cons-stream upper (integers-with-upper-limit (- upper 1))))
```

PP1 Miniproject

Miniproject setup

- Submission deadline:
  - Thursday, October 15, 2020 at 23:59.
- PP mini project 1 working day:
  - Tuesday, October 13, 2020 from 10-12
- The miniproject is estimated to 16 hours of work
  - Under the assumtion that you have followed the PP activities until now
- Submission in Moodle
  - Only zip files

Goals and Rules

- Full name, AAU study number, AAU email in top of (all) source file(s)
- 2-3 lines of status in (main) source file
- Info about Scheme system used same place
- Organize your work in a directory named after your AAU email address
- An individual exercise
  - But it is of course OK to discuss the solutions with fellow students

Submission format

- Directory structure:
  - username
    - Program files
- peter17@student.aau.dk
  - Put your program in a directory named peter17, and zip it.

Detailed program goals and requirements

- The program must be written in Scheme.
- The functional program must be well-written, well-structured and well-explained.
- Functions are supposed to be decomposed appropriately, such that 'large functions' that solve 'large problems' are decomposed.
- The program must adhere to the principles of the functional programming paradigm. File IO is allowed, however, if necessary.
- The program must make adequate use of higher-order functions.
- If you structure your data as lists, you are requested to write accessor functions, construction functions, and a recognizing predicate. Widespread use of car/cdr combinations is discouraged.
- Program explanations must be contained in program comments.
- It is not allowed to include program parts (such as functions) from sources such as books, papers, from the internet, or from fellow students. It is allowed to use the Scheme functions from the Scheme system that you use. You are also allowed to use the Scheme functions posted on the Moodle page of the course (named useful Scheme functions, and standard higher-order Scheme functions).

Quality is better than quantity

Concepts

Student

Group

Grouping

Object representation

- Lists
- Structs
- Closures

- Constructors
- Selectors
- Predicate

Group formation

- Random grouping
  - Referential transparency may be challenged
- Grouping by counting
- Balanced grouping

Other functionality

- Get group from grouping
- Number of groups in grouping
- Max/min group size in a grouping

- *Invent others which may help you in various ways*