# Functional Programming
## Lecture 2: Lambda abstraction

Viliam Lisý
Rostislav Horčík
Artificial Intelligence Center
Department of Computer Science
FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz
xhorcik@fel.cvut.cz

# Last lecture

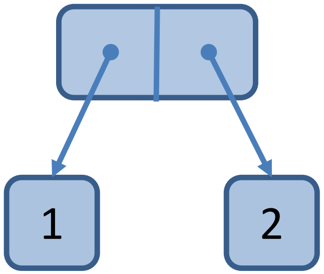
- What is (pure) functional programming
- Why do we care about it?
- Recursion is the main tool
- Scheme

  S-expression, quote,  identifiers, define, if, cond

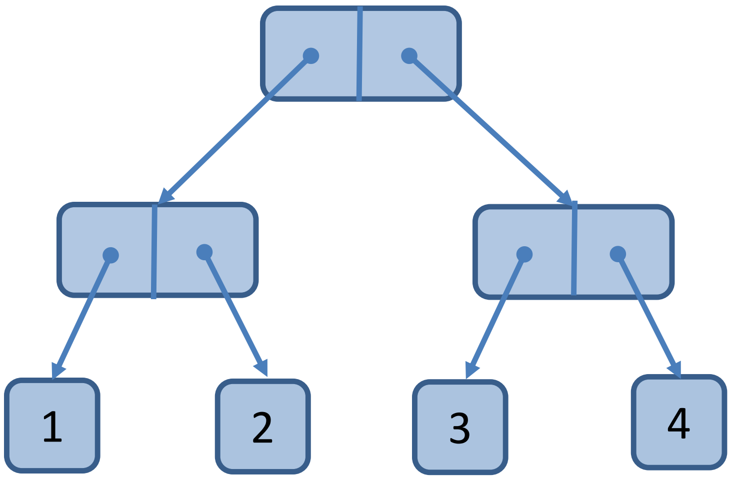
# Pairs

Allow to construct compound data structures

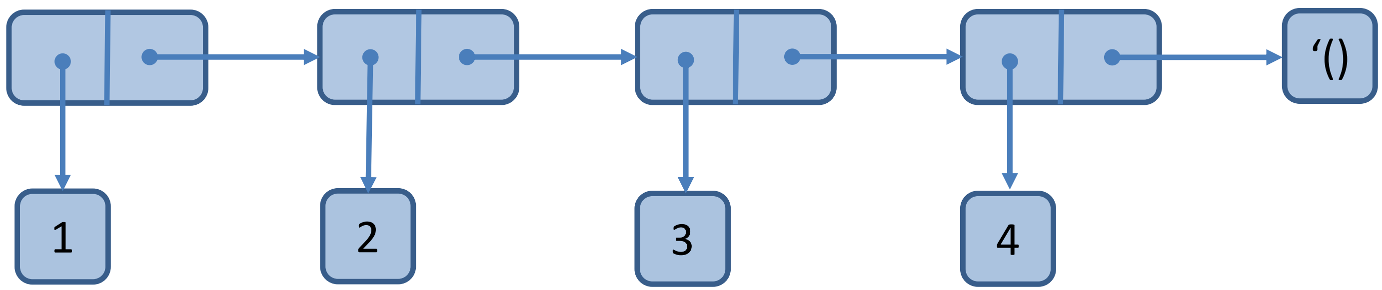
```
(cons 1 2)
    => (1 . 2)
```




```
(cons
    (cons 1 2)
    (cons 3 4))
```



# Lists




Lists are linked lists of pairs with `'()` at the end

S-expressions are just lists

```
'(+ 1 2 3 4 5)
```

Lists can be created by a function `cons` or

```
(list item1 item2 … itemN)
```

# Lists

Pairs forming the lists can be decomposed by

`car` *[car]* first element of the pair

`cdr` *[could-er]* second element of the pair

`(caddr x)` shortcut for `(car (cdr (cdr x )))`

Empty list is a null pointer

`null?` tests whether the argument is the empty list

# Append

```scheme
;;; Append two lists
(define (append2 a b)
  (cond
    ((null? a) b)
    (else (cons (car a)
                (append2 (cdr a) b)))
    )
  )
```

# Equality

Function = is only for numbers

Equivalence of the objects `eqv?`

`(eqv? 1 1), (eqv? 'a 'a) ===> #t`

`(eqv? (list 'a) (list 'a))  ===> #f`

More restrictive version is `eq?`

Typically the same pointer

Recursive version of `eqv?` on lists is

`(equal? (list 'a) (list 'a)) ===>#t`

# Debugging Basics

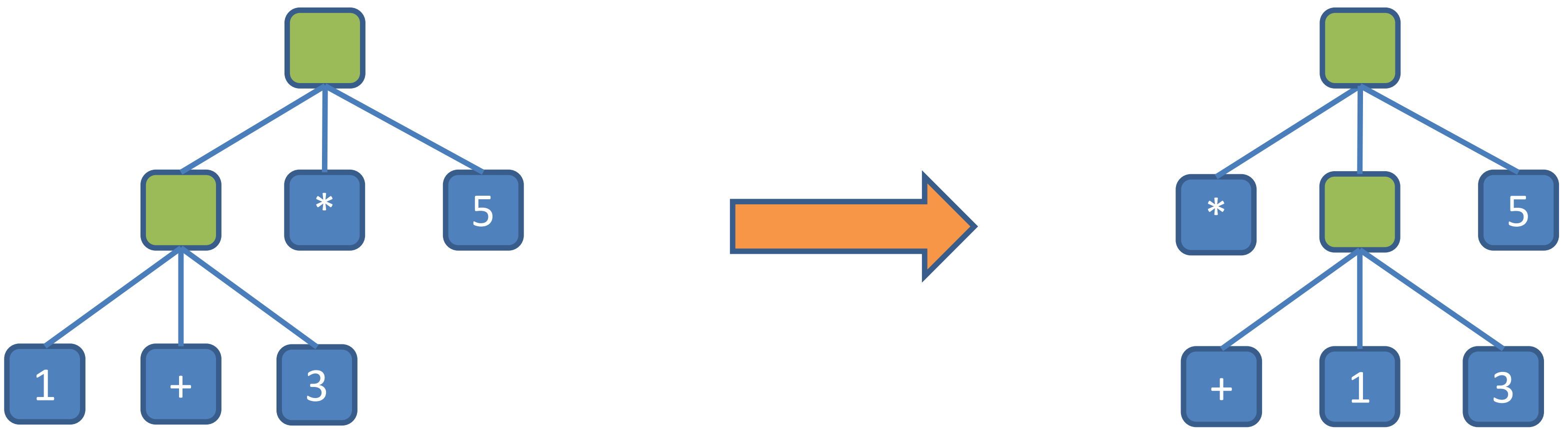
Tracing function calls and returns

```
#lang scheme
(require racket/trace)
(trace append2)
(untrace append2)
```

Helper print-outs

```
(begin (display x)
       (newline)
        <do-work>)
```

# Infix -> prefix




```
((1 + 3) * 5)
```

```
(* (+ 1 3) 5)
```

# Evaluation strategy

- Defines the order of evaluating the expressions
  - influences program termination, not the result
- Evaluation of scheme is eager (or strict)
  - left to right
  - evaluate all arguments before executing a function
- Evaluation of some special forms is lazy
  - if, and, or, **lambda**

# Functions are 1st-class citizens

- They may be named by variables
- They may be passed as arguments to a function
- They may be returned by a function
- They may be included in data structures

# Lambda abstraction

A construction for creating nameless procedures

```
(lambda (arg1 … argN) <expr>)
```

Define for functions is an abbreviation

```
(define (<var> <formals>) <body>)
```

Is the same as

```
(define <var>
    (lambda (<formals>) <body>))
```

# Filter

```
;;; Filter a list by the given predicate
(define (my-filter pred lst)
  (cond
    ((null? lst) '())
    ((pred (car lst))
     (cons (car lst)
      (my-filter pred (cdr lst))))
    (else (my-filter pred (cdr lst)))
    )
  )
```

# Derivative

$$Dg(x) \approx \frac{g(x+dx) - g(x)}{dx}$$

```
(define dx 0.00001)
(define (deriv g)
  (lambda (x)
    (/ (- (g (+ x dx)) (g x)) dx)
    )
  )
```

# Functions as data

```
(define (mult-fn fns x y)
  (cond
    ((null? fns) '())
    (else
      (cons
       ((car fns) x y)
       (mult-fn (cdr fns) x y)))
    )
  )
```

# Let

## Motivation

reuse of computation/result is often required

e.g., `minimum`, `roots` from the labs

```
(let ((<var1> <exp1>)
      (<var2> <exp2>))
                <body-using-var1-var2>)
```

$$f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$$

```scheme
;;; Local variables
(define (f x y)
  (let
      ((a (+ 1 (* x y)))
       (b (- 1 y)))
    (+ (* x a a)
       (* y b)
       (* a b))
    )
  )
```

# Implementing let

```
(let ((x <exp1>)
      (y <exp2>)) <body>)
```

Can be implemented as

```
((lambda (x y) <body>) <exp1> <exp2>)
```

# Let as lambda

```
(define (f2 x y)
  ((lambda (a b)
    (+ (* x a a)
       (* y b)
       (* a b))
    )
   (+ 1 (* x y))
   (- 1 y))
  )
```

# Let*

We might want to use the earlier definitions in the following.

```
(let ((x <exp>))
  (let ((y <exp-with-x>)) <body-x-y>)
```

Equivalent to

```
(let* ((x <exp>)
       (y <exp-with-x>)) <body-x-y>)
```

# Quicksort

```scheme
(define (qsort lst cmp)
  (cond
    ((null? lst) '())
    (else (let*
              ((pivot (car lst))
               (smaller (lambda (x) (cmp x pivot)))
               (greater (lambda (x) (not (cmp x pivot)))))
            (append
              (qsort (filter smaller (cdr lst)) cmp)
              (list pivot)
              (qsort (filter greater (cdr lst)) cmp))
            )
          )
    )
  )
```

# Scheme home assignments

Three connected assignments

- Robot simulation
- Population evaluation
- Code synthesis

Why this assignment?

Work on your own

Submit by midnight of the day after your lecture

- https://cw.felk.cvut.cz/brute/ (in 2 weeks)