Functional Programming
Lecture 3

Rostislav Horčík

Czech Technical University in Prague
Faculty of Electrical Engineering
xhorcik@fel.cvut.cz

Higher-order functions


WHAT IF I TOLD YOU
YOU CAN PASS A
FUNCTION TO FUNCTION
ELLORA
SOCIAL

Higher-order functions

Definition

A function taking other functions as arguments or returning a function as the result or both is called a higher-order function.

Higher-order functions

Definition

A function taking other functions as arguments or returning a function as the result or both is called a higher-order function.

- Allow capturing and reusing common programming patterns

Higher-order functions

Definition

A function taking other functions as arguments or returning a function as the result or both is called a higher-order function.

- Allow capturing and reusing common programming patterns
- Provide higher level of abstraction

Higher-order functions

Definition

A function taking other functions as arguments or returning a function as the result or both is called a higher-order function.

- Allow capturing and reusing common programming patterns
- Provide higher level of abstraction
- The reason why functional programs are compact

Apply

Sometimes we need to apply a function to arguments which are stored in a list.

Apply

Sometimes we need to apply a function to arguments which are stored in a list.

`(+ '(1 2 3))` does not work

Apply

Sometimes we need to apply a function to arguments which are stored in a list.

`(+ '(1 2 3))` does not work

`(apply fn arg1 ... argN lst)`

Apply

Sometimes we need to apply a function to arguments which are stored in a list.

`(+ '(1 2 3))` does not work

```
(apply fn arg1 ... argN lst)
```

```
(apply + '(1 2 3)) => 6
(apply + -3 2 '(1 2 3)) => 5
```

Apply

Sometimes we need to apply a function to arguments which are stored in a list.

`(+ '(1 2 3))` does not work

```
(apply fn arg1 ... argN lst)
```

```
(apply + '(1 2 3)) => 6
(apply + -3 2 '(1 2 3)) => 5
```

```
(apply append '((1 2 3) (a b))) => (1 2 3 a b)
```

Higher-order functions for lists

Higher-order functions for lists

- **filter** — filter elements of a list w.r.t. a predicate

Higher-order functions for lists

- **filter** — filter elements of a list w.r.t. a predicate
- **map** — apply a function element-wise to a list/lists

Higher-order functions for lists

- **filter** — filter elements of a list w.r.t. a predicate
- **map** — apply a function element-wise to a list/lists
- **foldr** — aggregate elements in a list using a binary operation from right to left

Higher-order functions for lists

- **filter** — filter elements of a list w.r.t. a predicate
- **map** — apply a function element-wise to a list/lists
- **foldr** — aggregate elements in a list using a binary operation from right to left
- **foldl** — aggregate elements in a list using a binary operation from left to right

Map

```
(map fn '(a1 a2 ...)
        '(b1 b2 ...)
        '(c1 c2 ...))
     => ((fn a1 b1 c1)  (fn a2 b2 c2) ...)
```

Map

```
(map fn '(a1 a2 ...)
        '(b1 b2 ...)
        '(c1 c2 ...))
     => ((fn a1 b1 c1)  (fn a2 b2 c2) ...)

(map (lambda (x) (* 2 x)) '(1 2 3)) => (2 4 6)
```

Map

```
(map fn '(a1 a2 ...)
        '(b1 b2 ...)
        '(c1 c2 ...))
     => ((fn a1 b1 c1)  (fn a2 b2 c2) ...)

(map (lambda (x) (* 2 x)) '(1 2 3)) => (2 4 6)

(map list-ref '((1 2 3)
                (4 5 6)
                (7 8 9)) (range 0 3)) => (1 5 9)
```

Folding

Let $f : A \times B \to B$ be a binary function and $b_0 \in B$.

Folding

Let $f : A \times B \to B$ be a binary function and $b_0 \in B$. Given a list $\vec{a} = (a_1, a_2, a_3, a_4)$ of values from A, define

Folding

Let $f : A \times B \to B$ be a binary function and $b_0 \in B$. Given a list $\vec{a} = (a_1, a_2, a_3, a_4)$ of values from A, define

$$foldl(f, b_0, \vec{a}) = f(a_4, f(a_3, f(a_2, f(a_1, b_0))))$$

Folding

Let $f : A \times B \to B$ be a binary function and $b_0 \in B$. Given a list $\vec{a} = (a_1, a_2, a_3, a_4)$ of values from A, define

$$foldl(f, b_0, \vec{a}) = f(a_4, f(a_3, f(a_2, f(a_1, b_0))))$$
$$foldr(f, b_0, \vec{a}) = f(a_1, f(a_2, f(a_3, f(a_4, b_0))))$$

Folding

Let $f : A \times B \to B$ be a binary function and $b_0 \in B$. Given a list $\vec{a} = (a_1, a_2, a_3, a_4)$ of values from A, define

$$foldl(f, b_0, \vec{a}) = f(a_4, f(a_3, f(a_2, f(a_1, b_0))))$$

$$foldr(f, b_0, \vec{a}) = f(a_1, f(a_2, f(a_3, f(a_4, b_0))))$$

$$foldl(+, 0, (1, 2, 3)) = 3 + (2 + (1 + 0)) = 6$$

$$foldr(+, 0, (1, 2, 3)) = 1 + (2 + (3 + 0)) = 6$$

Folding

Let $f : A \times B \to B$ be a binary function and $b_0 \in B$. Given a list $\vec{a} = (a_1, a_2, a_3, a_4)$ of values from A, define

$$foldl(f, b_0, \vec{a}) = f(a_4, f(a_3, f(a_2, f(a_1, b_0))))$$

$$foldr(f, b_0, \vec{a}) = f(a_1, f(a_2, f(a_3, f(a_4, b_0))))$$

$$foldl(+, 0, (1, 2, 3)) = 3 + (2 + (1 + 0)) = 6$$

$$foldr(+, 0, (1, 2, 3)) = 1 + (2 + (3 + 0)) = 6$$

$$foldl \quad b_0 \xrightarrow{f(a_1, b_0)} b_1 \xrightarrow{f(a_2, b_1)} b_2 \xrightarrow{f(a_3, b_2)} b_3 \xrightarrow{f(a_4, b_3)} b_4$$

Folding

Let $f : A \times B \to B$ be a binary function and $b_0 \in B$. Given a list $\vec{a} = (a_1, a_2, a_3, a_4)$ of values from A, define

$$foldl(f, b_0, \vec{a}) = f(a_4, f(a_3, f(a_2, f(a_1, b_0))))$$

$$foldr(f, b_0, \vec{a}) = f(a_1, f(a_2, f(a_3, f(a_4, b_0))))$$

$$foldl(+, 0, (1, 2, 3)) = 3 + (2 + (1 + 0)) = 6$$

$$foldr(+, 0, (1, 2, 3)) = 1 + (2 + (3 + 0)) = 6$$

$$foldl \quad b_0 \xrightarrow{f(a_1, b_0)} b_1 \xrightarrow{f(a_2, b_1)} b_2 \xrightarrow{f(a_3, b_2)} b_3 \xrightarrow{f(a_4, b_3)} b_4$$

$$foldr \quad b_0 \xrightarrow{f(a_4, b_0)} b_1 \xrightarrow{f(a_3, b_1)} b_2 \xrightarrow{f(a_2, b_2)} b_3 \xrightarrow{f(a_1, b_3)} b_4$$

Folding

```
(define (trace mat)
  (foldl + 0 (map list-ref
                  mat
                  (range 0 (length mat)))))

(trace '((1 2 3) (4 5 6) (7 8 9))) => 15
```

Folding

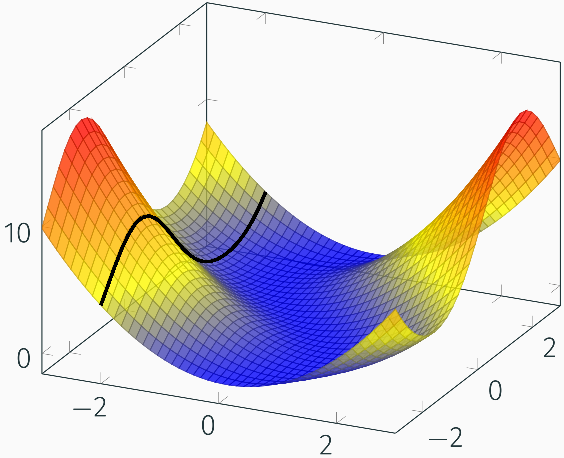
```
(define (trace mat)
  (foldl + 0 (map list-ref
                  mat
                  (range 0 (length mat)))))

(trace '((1 2 3) (4 5 6) (7 8 9))) => 15

(foldl cons '() '(a b c)) => (c b a)
(foldr cons '() '(a b c)) => (a b c)
(foldl min +inf.0 '(0 -3 2)) => -3.0
```

Currying and compositions

Partial evaluation — Currying



Partial evaluation — Currying

$$\begin{aligned} f : A \times B \to C &\quad\Longrightarrow\quad \hat{f} : A \to (B \to C) \\ f : A \times B \times C \to D &\quad\Longrightarrow\quad \hat{f} : A \to (B \to (C \to D)) \end{aligned}$$

Partial evaluation — Currying

$$
\begin{aligned}
f : A \times B \to C \quad &\Longrightarrow \quad \hat{f} : A \to (B \to C) \\
f : A \times B \times C \to D \quad &\Longrightarrow \quad \hat{f} : A \to (B \to (C \to D))
\end{aligned}
$$

```
(list-ref '(a b c) 2) => c
```

Partial evaluation — Currying

$$\begin{aligned} f: A \times B \to C \quad &\Longrightarrow \quad \hat{f}: A \to (B \to C) \\ f: A \times B \times C \to D \quad &\Longrightarrow \quad \hat{f}: A \to (B \to (C \to D)) \end{aligned}$$

```
(list-ref '(a b c) 2) => c
```

```
(define (curried-list-ref lst)
  (lambda (i) (list-ref lst i)))
```

Partial evaluation — Currying

$$\begin{aligned} f : A \times B \to C \quad &\Longrightarrow \quad \hat{f} : A \to (B \to C) \\ f : A \times B \times C \to D \quad &\Longrightarrow \quad \hat{f} : A \to (B \to (C \to D)) \end{aligned}$$

```
(list-ref '(a b c) 2) => c
```

```
(define (curried-list-ref lst)
  (lambda (i) (list-ref lst i)))
```

```
((curried-list-ref '(a b c)) 2) => c
```

Partial evaluation — Currying

$$f : A \times B \to C \quad \Longrightarrow \quad \hat{f} : A \to (B \to C)$$

$$f : A \times B \times C \to D \quad \Longrightarrow \quad \hat{f} : A \to (B \to (C \to D))$$

```
(list-ref '(a b c) 2) => c
```

```
(define (curried-list-ref lst)
  (lambda (i) (list-ref lst i)))
```

```
((curried-list-ref '(a b c)) 2) => c
```

```
(((curry list-ref) '(a b c)) 2) => c
```

Partial evaluation — Currying

$$f : A \times B \to C \quad \Longrightarrow \quad \hat{f} : A \to (B \to C)$$
$$f : A \times B \times C \to D \quad \Longrightarrow \quad \hat{f} : A \to (B \to (C \to D))$$

```
(list-ref '(a b c) 2) => c
```

```
(define (curried-list-ref lst)
  (lambda (i) (list-ref lst i)))
```

```
((curried-list-ref '(a b c)) 2) => c
```

```
(((curry list-ref) '(a b c)) 2) => c
```

Simplified syntax:

```
((curry list-ref '(a b c)) 2) => c
(map (curry * 2) '(1 2 3)) => (2 4 6)
```

Function composition

Let $f\colon A \to B$ and $g\colon B \to C$ be functions.

The function composition $(g \circ f)\colon A \to C$ is defined as

$$(g \circ f)(x) = g(f(x))$$

Function composition

Let $f: A \to B$ and $g: B \to C$ be functions.

The function composition $(g \circ f): A \to C$ is defined as

$$(g \circ f)(x) = g(f(x))$$

$$\begin{tikzcd}
A \arrow[r, "f"'] \arrow[rr, bend left, "g \circ f"] & B \arrow[r, "g"'] & C
\end{tikzcd}$$

Function composition

Let $f\colon A \to B$ and $g\colon B \to C$ be functions.

The function composition $(g \circ f)\colon A \to C$ is defined as

$$(g \circ f)(x) = g(f(x))$$

$$\begin{tikzcd}
A \arrow[r, "f"'] \arrow[rr, bend left, "g \circ f"] \arrow[rrr, bend right, "h \circ g \circ f"'] & B \arrow[r, "g"'] & C \arrow[r, "h"] & D
\end{tikzcd}$$

Compose function

Traditional style:

```
(define (str->alpha str)
  (list->string (filter char-alphabetic?
                        (string->list str))))
```

Compose function

Traditional style:

```
(define (str->alpha str)
  (list->string (filter char-alphabetic?
                        (string->list str))))
```

Point-free style:

```
(define str->alpha2
  (compose list->string
           (curry filter char-alphabetic?)
           string->list))
```

Example — morphic sequences

Let $\phi\colon \{0,1\} \to \{0,1\}^*$ such that $\phi(0) = 011$, $\phi(1) = 0$.

Example — morphic sequences

Let $\phi\colon \{0,1\} \to \{0,1\}^*$ such that $\phi(0) = 011$, $\phi(1) = 0$.

$\phi(10110) = 001100011$

Example — morphic sequences

Let $\phi\colon \{0,1\} \to \{0,1\}^*$ such that $\phi(0) = 011$, $\phi(1) = 0$.

$\phi(10110) = 001100011$

Define sequence:

$$\begin{aligned}
\phi^0(0) &= 0 \\
\phi^1(0) &= h(0) = 011 \\
\phi^2(0) &= h(011) = 01100 \\
\phi^3(0) &= h(01100) = 01100011011 \\
&\vdots
\end{aligned}$$

Closures

Free variables

Lambda expressions define functions whose body might contain free variables.

Free variables

Lambda expressions define functions whose body might contain free variables.

```
(define u 1)
(define fn (lambda (x) (+ x u)))
; u is a free variable
```

Free variables

Lambda expressions define functions whose body might contain free variables.

```
(define u 1)
(define fn (lambda (x) (+ x u)))
; u is a free variable
```

```
(fn 5) ; => 6 = 5 + 1
```

Free variables

Lambda expressions define functions whose body might contain free variables.

```
(define u 1)
(define fn (lambda (x) (+ x u)))
; u is a free variable
```

```
(fn 5) ; => 6 = 5 + 1
```

```
(let ([u -1])
  (fn 5))
```

Free variables

Lambda expressions define functions whose body might contain free variables.

```
(define u 1)
(define fn (lambda (x) (+ x u)))
; u is a free variable
```

```
(fn 5) ; => 6 = 5 + 1
```

```
(let ([u -1])
  (fn 5))
```

```
=> 6
```

Binding scopes

Binding scope is a portion of the source code where a value is bound to a given name.

Binding scopes

Binding scope is a portion of the source code where a value is bound to a given name.

- Lexical scope — functions use bindings available where **defined**

Binding scopes

Binding scope is a portion of the source code where a value is bound to a given name.

- Lexical scope — functions use bindings available where **defined**
- Dynamic scope — functions use bindings available where **executed**

Binding scopes

Binding scope is a portion of the source code where a value is bound to a given name.

- Lexical scope — functions use bindings available where **defined**
- Dynamic scope — functions use bindings available where **executed**

Scheme/Racket uses the lexical scope as most of the modern programming languages.

Function closures

```
(define (make-adder x) (lambda (y) (+ x y)))

(define adder1 (make-adder 10))
(define adder2 (make-adder 3))
(define adder3 (make-adder -7))
```

Function closures

```
(define (make-adder x) (lambda (y) (+ x y)))

(define adder1 (make-adder 10))
(define adder2 (make-adder 3))
(define adder3 (make-adder -7))
```

```
adder1 -> (lambda (y) (+ 10 y))
adder2 -> (lambda (y) (+ 3 y))
adder3 -> (lambda (y) (+ -7 y))
```

Function closures

```
(define (make-adder x) (lambda (y) (+ x y)))

(define adder1 (make-adder 10))
(define adder2 (make-adder 3))
(define adder3 (make-adder -7))
```

~~`adder1 -> (lambda (y) (+ 10 y))`~~
~~`adder2 -> (lambda (y) (+ 3 y))`~~
~~`adder3 -> (lambda (y) (+ -7 y))`~~

```
adder1 -> (x 10) (lambda (y) (+ x y))
adder2 -> (x 3) (lambda (y) (+ x y))
adder3 -> (x -7) (lambda (y) (+ x y))
```

Closures

A **lambda** expression defining a function evaluates to a lexical/function closure value.

Closures

A **lambda** expression defining a function evaluates to a lexical/function closure value.

A lexical/function closure is a pair of pointers to:

- code of the function
- environment where the function was defined

Closures

A **lambda** expression defining a function evaluates to a lexical/function closure value.

A lexical/function closure is a pair of pointers to:

- code of the function
- environment where the function was defined

A **function call** expression:

- Evaluates the code of a function closure
- In the environment of the function closure extended by with bindings for the arguments

Applications of closures

Closures allow to “store” data in functions.

Applications of closures

Closures allow to “store” data in functions.

```
(define (point x y)
  (lambda (m) (m x y)))
```

Applications of closures

Closures allow to “store” data in functions.

```
(define (point x y)
  (lambda (m) (m x y)))

(define (get-x p)
  (p (lambda (x y) x)))
```

Applications of closures

Closures allow to “store” data in functions.

```
(define (point x y)
  (lambda (m) (m x y)))

(define (get-x p)
  (p (lambda (x y) x)))

(define (get-y p)
  (p (lambda (x y) y)))
```

Applications of closures

Closures allow to “store” data in functions.

```
(define (point x y)
  (lambda (m) (m x y)))

(define (get-x p)
  (p (lambda (x y) x)))

(define (get-y p)
  (p (lambda (x y) y)))

(define p (point 3 10))
(get-x p) => 3
(get-y p) => 10
```

Structures

Racket allows to define new datatypes so-called structures.

Structures

Racket allows to define new datatypes so-called structures.

```
(struct person (first-name surname age)
  #:transparent) ; defines type

(define pers (person "John"
                     "Down"
                     33)) ; defines instance
```

Structures

Racket allows to define new datatypes so-called structures.

```
(struct person (first-name surname age)
  #:transparent) ; defines type

(define pers (person "John"
                     "Down"
                     33)) ; defines instance
```

Accessor functions to all fields are automatically defined.

```
(person-first-name pers) => "John"
(person-surname pers) => "Down"
(person-age pers) => 33
(person? pers) => #t
(person? "John") => #f
```

Homework assignment

Homework assignment 1 — ASCII art generator

Aim: Practice applications of higher-order functions

```
;;;::::,,,,...
o;;;;::::,,,....
ooo;;;;:::,,,,....
xoooo;;;::::,,,....
xxoooo;;;;:::,,,,...
xxxxoooo;;;::::,,,,.
%%xxxxooo;;;;::::,,,
%%%%xxxoooo;;;;:::,,
##%%%xxxxoooo;;;::::
###%%%xxxoooo;;;;::
@####%%%xxxxoooo;;;:
@@@###%%%%xxxxooo;;;
```

Homework assignment 1 — ASCII art generator

Aim: Practice applications of higher-order functions

`ascii-art`

```
;;;::::,,,,...
o;;;;::::,,,....
ooo;;;;:::,,,,....
xoooo;;;::::,,,....
xxoooo;;;;:::,,,,...
xxxxoooo;;;::::,,,,.
%%xxxxooo;;;;::::,,,
%%%%xxxoooo;;;;:::,,
##%%%xxxxoooo;;;::::
###%%%xxxoooo;;;;::
@####%%%xxxxoooo;;;:
@@@###%%%%xxxxooo;;;
```

Homework assignment 1 — ASCII art generator

Aim: Practice applications of higher-order functions

```
;;;::::,,,,...
o;;;;::::,,,....
ooo;;;;:::,,,,....
xoooo;;;::::,,,....
xxoooo;;;;:::,,,,...
xxxxoooo;;;::::,,,,.
%%xxxxooo;;;;::::,,,
%%%%xxxoooo;;;;:::,,
##%%%xxxxoooo;;;::::
###%%%%xxxoooo;;;;::
@####%%%xxxxoooo;;;:
@@@###%%%%xxxxooo;;;
```

	.	,	:	;	o	x	\%	\#	@

Homework assignment 1 — ASCII art generator

Aim: Practice applications of higher-order functions

```
;;;::::,,,,...
o;;;;::::,,,....
ooo;;;;:::,,,,....
xoooo;;;::::,,,....
xxoooo;;;;:::,,,,...
xxxxoooo;;;::::,,,,.
%%xxxxooo;;;;::::,,,
%%%%xxxoooo;;;;:::,,
##%%%xxxxoooo;;;::::
###%%%%xxxoooo;;;;::
@####%%%xxxxoooo;;;:
@@@###%%%%xxxxooo;;;
```

	.	,	:	;	o	x	\%	\#	@

Points: 10

Homework assignment 1 — ASCII art generator

Aim: Practice applications of higher-order functions

```
;;;::::,,,,...
o;;;;::::,,,....
ooo;;;;:::,,,,....
xoooo;;;::::,,,....
xxoooo;;;;:::,,,,...
xxxxoooo;;;::::,,,,.
%%xxxxooo;;;;::::,,,
%%%%xxxoooo;;;;:::,,
##%%%xxxxoooo;;;::::
###%%%%xxxoooo;;;;::
@####%%%xxxxoooo;;;:
@@@###%%%%xxxxooo;;;
```

	.	,	:	;	o	x	\%	\#	@

Points: 10

Deadline: in 3 weeks (March 30)

Homework assignment 1 — ASCII art generator

Aim: Practice applications of higher-order functions

```
;;;::::,,,,...
o;;;;::::,,,....
ooo;;;;:::,,,,....
xoooo;;;::::,,,....
xxoooo;;;;:::,,,,...
xxxxoooo;;;::::,,,,.
%%xxxxooo;;;;::::,,,
%%%%xxxoooo;;;;:::,,
##%%%xxxxoooo;;;::::
###%%%%xxxoooo;;;;::
@####%%%xxxxoooo;;;:
@@@###%%%%xxxxooo;;;
```

	.	,	:	;	o	x	\%	\#	@

Points: 10

Deadline: in 3 weeks (March 30)

Penalty: after deadline -1 points every day (at most -9)

Homework assignment 1 — ASCII art generator

Aim: Practice applications of higher-order functions

```
;;;::::,,,,...
o;;;;::::,,,....
ooo;;;;:::,,,,....
xoooo;;;::::,,,....
xxoooo;;;;:::,,,,...
xxxxoooo;;;::::,,,,.
%%xxxxooo;;;;::::,,,
%%%%xxxoooo;;;;:::,,
##%%%xxxxoooo;;;::::
###%%%%xxxoooo;;;;::
@####%%%xxxxoooo;;;:
@@@###%%%%xxxxooo;;;
```

	.	,	:	;	o	x	\%	\#	@

Points: 10

Deadline: in 3 weeks (March 30)

Penalty: after deadline -1 points every day (at most -9)

Description: all details can be found in CW

What have we learned?

- Higher-order functions have functional arguments or returns a function or both.

What have we learned?

- Higher-order functions have functional arguments or returns a function or both.
- `apply` allows to apply a function to arguments enclosed in a list.

What have we learned?

- Higher-order functions have functional arguments or returns a function or both.
- `apply` allows to apply a function to arguments enclosed in a list.
- `map` applies a function to lists elementwise.

What have we learned?

- Higher-order functions have functional arguments or returns a function or both.
- `apply` allows to apply a function to arguments enclosed in a list.
- `map` applies a function to lists elementwise.
- `foldl`, `foldr` aggregate values in a list by a given function.

What have we learned?

- Higher-order functions have functional arguments or returns a function or both.
- `apply` allows to apply a function to arguments enclosed in a list.
- `map` applies a function to lists elementwise.
- `foldl`, `foldr` aggregate values in a list by a given function.
- `curry` transforms a function into its currified form.

What have we learned?

- Higher-order functions have functional arguments or returns a function or both.
- `apply` allows to apply a function to arguments enclosed in a list.
- `map` applies a function to lists elementwise.
- `foldl`, `foldr` aggregate values in a list by a given function.
- `curry` transforms a function into its currified form.
- `compose` performs functional composition.

What have we learned?

- Higher-order functions have functional arguments or returns a function or both.
- `apply` allows to apply a function to arguments enclosed in a list.
- `map` applies a function to lists elementwise.
- `foldl`, `foldr` aggregate values in a list by a given function.
- `curry` transforms a function into its currified form.
- `compose` performs functional composition.
- Scheme/Racket uses lexical scope.

What have we learned?

- Higher-order functions have functional arguments or returns a function or both.
- `apply` allows to apply a function to arguments enclosed in a list.
- `map` applies a function to lists elementwise.
- `foldl`, `foldr` aggregate values in a list by a given function.
- `curry` transforms a function into its currified form.
- `compose` performs functional composition.
- Scheme/Racket uses lexical scope.
- Function closures are pairs storing the code of a function together with the current environment.