

Elementy języka **Scheme**

- Historia języka Lisp
- Wyrażenia i ewaluacja wyrażeń
- Identyfikatory i wyrażenie **let**
- Wyrażenia **lambda**
- Definicje globalne
- Wyrażenia warunkowe
- Przypisanie
- Kontynuacje

Historia języka Lisp

- 1958 **John McCarthy** (MIT) rozpoczęcie prac nad obliczeniami symbolicznymi
- 1960 **Lisp 1**
- 1962 **Lisp 1.5**
- 1975 **Scheme**
- 1984 **Common Lisp**
- 2007 **Clojure**

Wyrażenia i ewaluacja wyrażeń

Interakcja z językiem Scheme

```
42 => 42
22/7 => 22/7
3.1415 => 3.1415
+ => #<procedure>
(+ 76 31) => 107
'halo => halo
halo => Unbound variable: halo
'(a b c d) => (a b c d)
(car '(a b c)) => a
(cdr '(a b c)) => (b c)
(cons 'a '(b c)) => (a b c)
```

```
(cons (car '(a b c))
      (cdr '(a b c))) => (a b c)
```

```
(define kwadrat
  (lambda (n)
    (* n n)))
(kwadrat 5) => 25
(kwadrat 1/2) => 1/4
```

Założmy, że w pliku `odwrotna.ss` znajduje się następujący kod:

```
(define odwrotna (lambda (n) (if (= n 0)
  "ojej!" (/ 1 n))))
```

```
(load "odwrotna.ss")
(odwrotna 10) => 1/10
(odwrotna 1/10) => 10
(odwrotna 0) => "ojej!"
```

Wyrażenia proste

```
(+ 1/2 1/2) => 1
(- 1.5 1/2) => 1.0
(* 3 1/2) => 3/2
(/ 1.5 3/4) => 2.0
(+ (+ 2 2) (+ 2 2)) => 8
(- 2 (* 4 1/3)) => 2/3
(quote (1 2 3 4 5)) => (1 2 3 4 5)
(quote (+ 3 4)) => (+ 3 4)
(car '(a b c)) => a
(cdr '(a b c)) => (b c)
(cons 'a '(b c)) => (a b c)
(cons 'a 'b) => (a . b) ← para
(list 'a 'b 'c) => (a b c)
(list 'a) => (a)
(list) => ()
```

Ewaluacja wyrażeń

- Obiekt stały (liczby, napisy) ma wartość identyczną z nim samym.
- (procedura arg1 arg2 ... argN)
 1. obliczana jest wartość **procedura**
 2. obliczana są wartości **arg1, arg2, ..., argN**
 3. stosuje się wartość procedury do wartości argumentów **arg1, arg2, ..., argN**
- (quote obiekt) ma wartość **obiekt** (dokładnie ją „cytuje”)


Przykład:

```
((car (list + - * /)) 2 3) => 5
```

Identyfikatory i wyrażenie **let**

```
(let ((x 2)) (+ x 3)) => 5
```

niech x ma wartość 2 w wyrażeniu (+ x 3)

<pre>(let ((x1 w1)) (let ((x2 w2)) ... (let ((xn wn)) ...))...)</pre>		<pre>(let* ((x1 w1) (x2 w2) ... (xn wn)) ...)</pre>
---	---	---

wartości w1, ..., wn są obliczane i wiązane ze zmiennymi x1, ..., xn jedna po drugiej

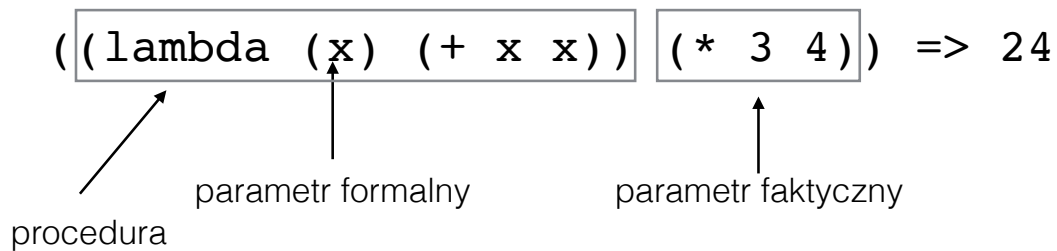
```
(let ((x1 w1)  
      (x2 w2)  
      ...  
      (xn wn))  
  ...)
```

wartości w1, ..., wn są obliczane i wiązane ze zmiennymi x1, ..., xn jednocześnie

Wyrażenia **lambda**

```
(lambda (x) (+ x x)) => #<procedure>
```

```
((lambda (x) (+ x x)) (* 3 4)) => 24
```



procedura parametr formalny parametr faktyczny

```
(let ((podwojone (lambda (x) (+ x x))))  
  (list (podwojone (* 3 4))  
        (podwojone (/ 99 11))  
        (podwojone (- 2 7))))) => (24 18 -10)
```

```
(let ((f (lambda x x)))  
  (f 1 2 3 4)) => (1 2 3 4)
```

```
(let ((g (lambda (x . y) (list x y))))  
  (g 1 2 3 4)) => (1 (2 3 4))
```

```
(let ((h (lambda (x y . z) (list x y z))))  
  (h 1 2 3 4)) => (1 2 (3 4))
```

Definicje globalne

```
(define podwojone-coś (lambda (f x) (f x x)))  
(podwojone-coś + 10) => 20  
(podwojone-coś cons 'a) => (a . a)
```

```
(define kanapka "masło orzechowe z dżemem")  
kanapka => "masło orzechowe z dżemem"
```

```
(define list (lambda x x))  
(define cadr (lambda (x) (car (cdr x))))  
(define cddr (lambda (x) (cdr (cdr x))))  
(cadr '(a b c)) => b   drugi element listy  
(cddr '(a b c)) => (c) ogon ogona listy
```

```
(define podwajacz  
  (lambda (f)  
    (lambda (x)  
      (f x x))))  
(define podwojone (podwajacz +))  
(podwojone 13/2) => 13
```

```
(define podwojone-cons (podwajacz cons))  
(podwojone-cons 'a) => (a . a)
```

```
(define podwojone-coś  
  (lambda (f x) ((podwajacz f) x)))
```

Przykład:

```
(define g (lambda (f x) (f x x)))
```

```
(g g g) = ((lambda (f x) (f x x)) g g)
        = (g g g)
        = ... nieskończone obliczenia
```

Wyrażenia warunkowe

```
(define abs
```

```
  (lambda (n)
```

```
    (if (< n 0)
```

```
        (- 0 n)
```

```
        n)
```

```
  )
```

```
)
```

← forma składniowa
(to nie jest wywołanie procedury)

Wybrane warunki

```
(< -1 0) => #t
(> -1 0) => #f
(not #t) => #f
(not #f) => #t
(not '()) => #t    lista pusta jest fałszem
(not '(a b)) => #f  każda wartość różna od listy
                  pustej i #f jest prawdą
(or) => #f
(or #f) => #f
(of #f #t) => #t
(or #f 'a #f) => a  wartość pierwszego prawdziwego
                  argumentu lub fałsz
(and) => #t
(and #t) => #t
(and #f) => #f
(and #t 'a) => a    wartość ostatniego
                  argumentu lub fałsz
(and #t #f 'a) => #f
```

Przykład:

```
(define odwrotna
  (lambda (n)
    (and (not (= n 0)) (/ 1 n))))
```

```
(odwrotna 1/10) => 10
```

```
(odwrotna 0) => #f
```

```
(null? '()) => #t
(null? 'abc) => #f
(null? '(a b c)) => #f
(equal? 'a 'a) => #t
(equal? 'a 'b) => #f
(equal? '(a b c) '(a b c)) => #t
(pair? '(a . c)) => #t
(pair? 'a) => #f
(pair? '(a b c)) => #t  bo lista jest parą głowy i ogona
                        (a . (b . (c . ())))
```


<code>(cond (test1 wyrażenie1)</code>	testy sprawdzane są po kolei i jeśli któryś
<code>(test2 wyrażenie2)</code>	jest prawdą, to wartością jest wartość
<code>...</code>	odpowiadającego mu wyrażenia,
<code>(else domyślne))</code>	w przeciwnym przypadku wartość
	wyrażenia domyślnego

Przykład:

```
(define podatek
  (lambda (dochód)
    (cond ((<= dochód 3091)
           0)
          ((<= dochód 85528)
           (- (* 0.18 dochód) 556.20))
          (else
           (+ 14839.02
              (* 0.32 (- dochód 85528)))))))
```

```
(define member
  (lambda (x ls)
    (cond ((null? ls) #f)
          ((equal? (car ls) x) ls)
          (else (member x (cdr ls))))))
```

```
(member 'b '(a b b d)) => (b b d)
```

```
(member 'c '(a b b d)) => #f
```

```
(define remove
  (lambda (x ls)
    (cond ((null? ls) '())
          ((equal? (car ls) x)
           (remove x (cdr ls)))
          (else (cons (car ls)
                       (remove x (cdr ls)))))))
```

```
(remove 'b '(a b b d)) => (a d)
```

Przypisanie

```
(define abcde '(a b c d e))  
abcde => (a b c d e)  
(set! abcde (cdr abcde))  
abcde => (b c d e)
```

```
(define licznik1  
  (lambda ()  
    (let ((x 0))  
      (set! x (+ x 1))  
      x)))
```

```
(licznik1) => 1  
(licznik1) => 1
```

...

za każdym razem x jest inicjowane
wartością 0 więc za każdym
razem zwracana jest wartość 1

```
(define akumulator 0)  
(define licznik2  
  (lambda ()  
    (let ((x akumulator))  
      (set! akumulator (+ akumulator 1))  
      x)))
```

```
(licznik2) => 0  
(licznik2) => 1  
(licznik2) => 2
```

...

tylko jeden licznik bo
tylko jeden globalny akumulator

```
(define nowy-licznik
  (lambda ()
    (let ((akumulator 0))
      (lambda ()
        (let ((x akumulator))
          (set! akumulator (+ akumulator 1))
          x))))))
```

```
(define licz1 (nowy-licznik))
(define licz2 (nowy-licznik))
(licz1) => 0
(licz1) => 1          każdy licznik ma swój
(licz2) => 0          lokalny akumulator
(licz1) => 2
...
```

Kontynuacje

- Kontynuacja zawsze jest dla danego podwyrażenia w kontekście szerszego wyrażenia.
- W wyrażeniu `(f (g (h)))` kontynuacją podwyrażenia `(g (h))` jest obliczenie `(f x)`, gdzie `x` jest wartością zwróconą przez `(g (h))`.
- `(call-with-current-continuation procedura)` przekazuje bieżącą kontynuację do jednoargumentowej procedury.
- Będziemy korzystać z następującej definicji skracającej zapis:

```
(define call/cc call-with-current-continuation)
```

Przykłady:

```
(call/cc (lambda (k) (* 5 4))) => 20
```

procedura nie skorzystała z kontynuacji i wyliczyła iloczyn

```
(call/cc (lambda (k) (* 5 (k 4)))) => 4
```

procedura przerwała obliczenie iloczynu i przeniosła obliczenia do kontynuacji oddając wynik 4 (parametr faktyczny kontynuacji)

```
(+ 2 (call/cc (lambda (k) (* 5 (k 4))))) => 6
```

procedura przerwała obliczenie iloczynu i przeniosła obliczenia do kontynuacji oddając wynik 4, który został powiększony o 2

Styl przekazywania kontynuacji (CPS - continuation passing style)

```
(define (f x...) ...) (define (kf x... k) (k (f x...)))
```

```
(define (return x)  
  x)
```

```
(define (k+ a b k)  
  (k (+ a b)))
```

```
(define (k* a b k)  
  (k (* a b)))
```

```
(k+ 1 2 (lambda (x) (k* x 3 return)))
```

Funkcje rekurencyjne w CPS

```
;;; normal factorial
(define (fact n)
  (if (= n 1)
      1
      (* n (fact (- n 1)))))

;;; CPS factorial
(define (kfact n k)
  (if (= n 1)
      (k 1)
      (kfact (- n 1) (lambda (x) (k (* n x))))))

;;; normal
(+ 3 (fact 4))

;;; CPS
(kfact 4 (lambda (x) (k+ x 3 return)))
```

Pętla do

```
(do ((variable init update) ... )
    (test expr)
    body)
```

Połączenie pętli **for** (inicjowanie zmiennych i uaktualnienie ich wartości po każdym przebiegu) z pętlą **while** sprawdzającą warunek **test** i zwracającą wartość wyrażenia **expr** gdy zachodzi. Przy każdym przebiegu pętli wykonywana jest jej treść **body**.

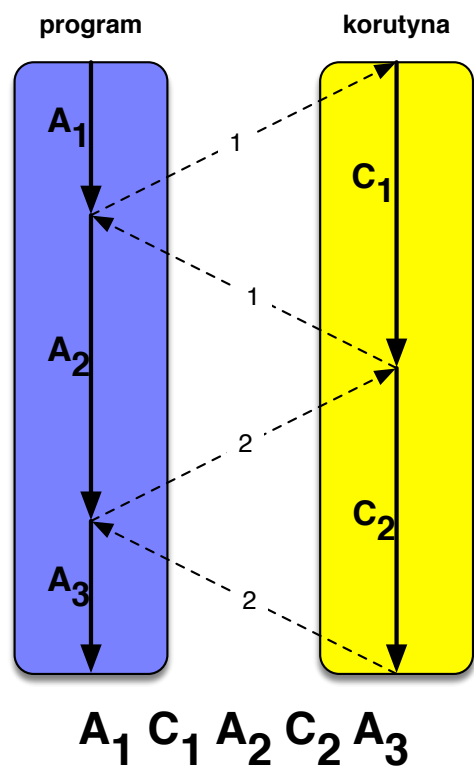
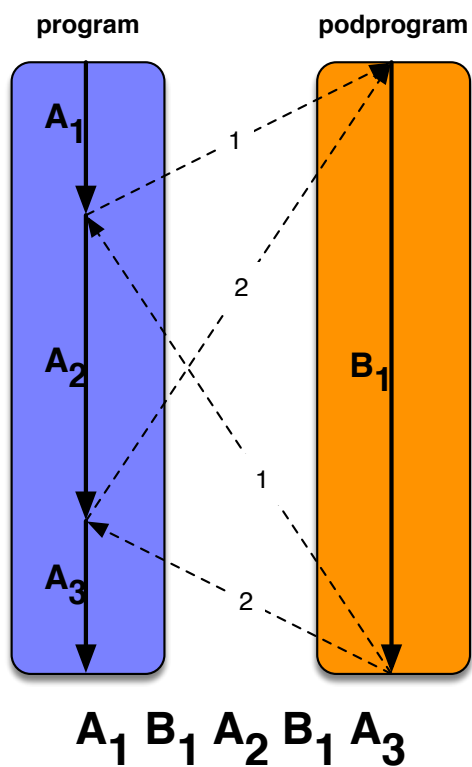
Przykład:

```
(define factorial
  (lambda (n)
    (do ((i n (- i 1)) (a 1 (* a i)))
        ((zero? i) a))))
```

Tworzenie nielokalnego wyjścia z pętli

```
(define member
  (lambda (x ls)
    (call/cc
      (lambda (stop)
        (do ((ls ls (cdr ls)))
          ((null? ls) #f)
          (if (equal? x (car ls))
              (stop ls))))))))
```

Korutyny



```

;;; coroutines.rkt
;;;
;;; Przykład z "Yet Another Scheme Tutorial"
;;; http://www.shido.info/lisp/idx\_scm\_e.html

;;; Zmodyfikowano dla języka Racket.

```

```

(require scheme/mpair)
      mutable pair constructors and selectors

```

```

(mcons A B) (mpair? P)
      (mcar P) (mcd r P)
(set-mcar! P C) (set-mcdr! P C)

```

```

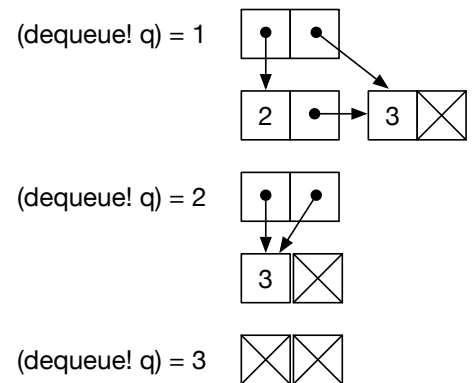
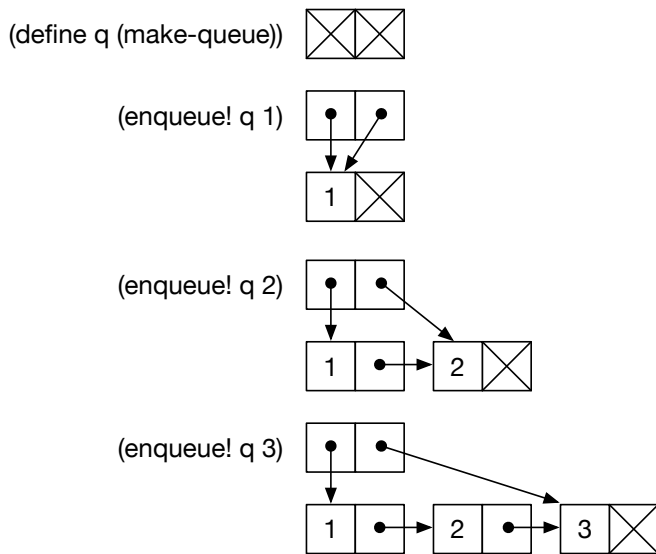
;;; queue

(define (make-queue)
  (mcons '() '()))

(define (enqueue! queue obj)
  (let ((lobj (mcons obj '())))
    (if (null? (mcar queue))
        (begin
          (set-mcar! queue lobj)
          (set-mcdr! queue lobj))
        (begin
          (set-mcdr! (mcd r queue) lobj)
          (set-mcdr! queue lobj)))
    (mcar queue)))

(define (dequeue! queue)
  (let ((obj (mcar (mcar queue))))
    (set-mcar! queue (mcd r (mcar queue)))
    obj))

```



```
;;; coroutine
```

```
(define process-queue (make-queue))
```

```
(define (coroutine thunk)
  (enqueue! process-queue thunk))
```

```
(define (start)
  ((dequeue! process-queue)))
```

```
(define (pause)
  (call/cc
   (lambda (k)
     (coroutine (lambda () (k #f)))
     (start)))))
```



```
;;; example

(coroutine (lambda ()
  (for ([i '(0 1 2 3 4)])
    (display i)
    (pause))))

(coroutine (lambda ()
  (for ([i '(5 6 7 8 9)])
    (display i)
    (pause))))

;;; run example

(start)
```

```
$ racket
Welcome to Racket v6.8.
> (load "coroutines.rkt")
0516273849
> (exit)
$
```