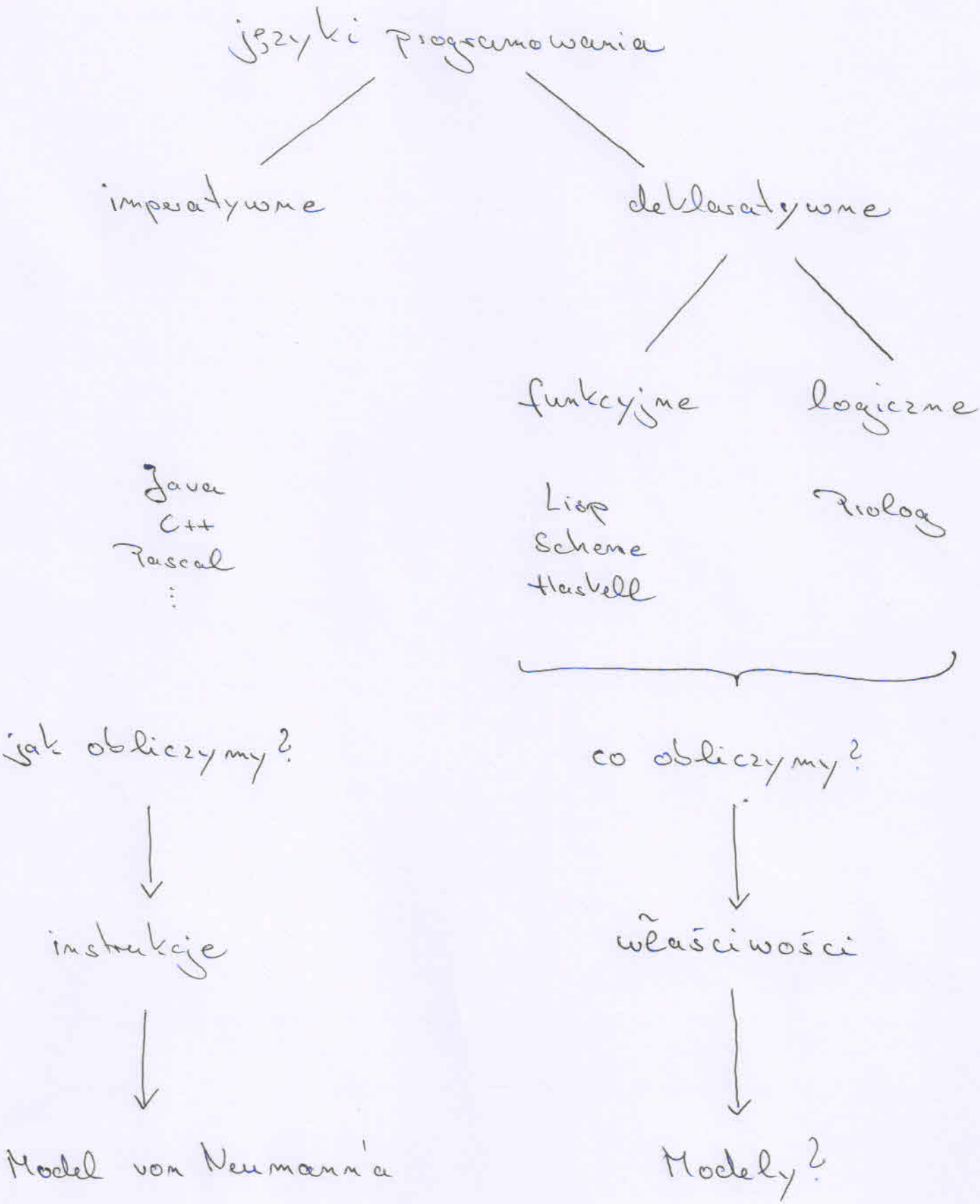
# Programowanie deklaratywne

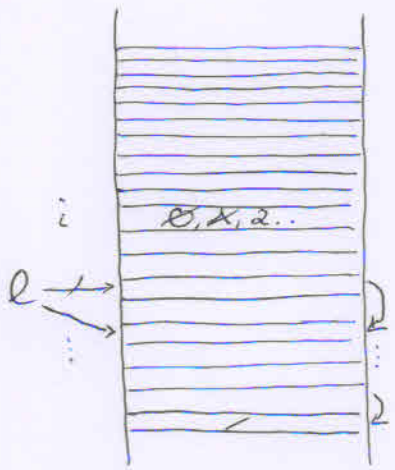


# Przykład: Długość listy

(i) wersja imperatywna

"manipuluje wartości zmiennych w pamięci"

```
int length (lista l) {
    int i := 0;
    while l.next ≠ NULL {
        i := i+1;
        l := l.next;}
    return i;}
```




(ii) wersje deklaratywne

"używają właściwości length"

- długość pustej listy, to 0.
- długość niepustej listy, to jeden plus długość ogona listy.

uwaga: Właściwości te (też) w wersji imperatywnej pokazują poprawność.

wersja funkcyjna (Scheme):

```
(define (length l)
  (if (= l '())
      0
      (+1 (length (cdr l)))))
```

```
(length '(1 2 3))
=> (if (= '(1 2 3) '())
       0
       (+ 1 (length (cdr '(1 2 3)))))
=> (+ 1 (length (cdr '(1 2 3))))
=> (+ 1 (length '(2 3)))
=*> (+ 1 (+ 1 (length '(3))))
=*> (+ 1 (+ 1 (+ 1 (length '()))))
=*> (+ 1 (+ 1 (+ 1 0)))
=*> 3
```

wersja logiczna (Prolog):

predykat length(L, N), który jest spełniony, jeżeli N jest długością listy L.

```
length([], 0).
length([X|L], N) :- length(L, M), N = M+1.
```

```
length([1,3], N)
    | X = 1, L = [3]
length([3], M), N = M+1
    | x' = 3, L' = []
length([], R), M = R+1, N = M+1
    | R = 0
M = 0+1, N = M+1
    | M = 1
N = 1+1          ~> N = 2
```

# Programowanie funkcyjne

(prawie) wszystkie obiekty są funkcjami

↳ nie ma instrukcji
nie ma przepisania
nie ma typów (w Scheme)

jak programować?

↳ definicja funkcji

+

zastosowanie funkcji

- Podstawa: $\lambda$-rachunek

wyrażenia $T$

abstrakcja: $\lambda x.T$ "$f(x) = T$"

aplikacja: $Tx$ "$f(x)$"

Przykład:

$T = x + 5$

$\lambda x.T \equiv \lambda x.(x+5)$ "$f(x) = x+5$"

$(\lambda x.T)3 \equiv (\lambda x.(x+5))3$ "$f(3)$"

$\Rightarrow 3+5$

$\Rightarrow 8$

Każdy wyrażenie ma wartość (też funkcje)

uwaga: wartości (albo argumenty) mogą być funkcją

↳ nie ma różnicy między danymi i funkcjami

Przykład:

$(\lambda f.\ f\ 3)$ "$g(f) = f(3)$"

$(\lambda f.\ f\ 3)(\lambda x.\ x + 5)$

$\Rightarrow (\lambda x.\ x + 5)\ 3$

$\Rightarrow 3 + 5$

$\Rightarrow 8$

## Scheme

"implementacja λ-rachunku" używając listy

```
(+ x 5)
↑      ↖ actualne argumenty
operator/
funkcja
```

```
(lambda (x) (+ x 5))
       ↑      ↖ ciało funkcji
formalne argumenty
funkcji
```

((lambda (x) (+ x 5)) 3)

↳ funkcja anonymowa

(define (square n) (* n n))

≙ (define square (lambda (n) (* n n)))

czyli

(square 3)

≡ ((lambda (n) (* n n)) 3)

⇒ (* 3 3)

⇒ 9

## Wybudowane w Scheme:

- liczby, arytmetyka
- predykaty (#t, #f, =, and, not, ...)
- warunek (if, cond)
- funkcje dla listy:

(cons 1 '(2 3)) ⇒ '(1 2 3)

(car '(1 2 3)) ⇒ 1

(cdr '(1 2 3)) ⇒ '(2 3)

stąd nazwa Lisp — "List processing"

(a Scheme jest "dialektem" Lisp'u)

Literatura:

- Abelson, Sussman; Structure and Interpretation of Computer Programs
- Friedman, Felleisen; The Little Schemer
- Revised (5) Report on the Algorithmic Language Scheme

# §1 Wprowadzenie do Scheme'a

Scheme ewaluuje wyrażenia:

```
> 486
486
> (+ 137 349)
486
> (+ (* 3 5) (- 10 6))
19
> (+ x 3)
```

↯ "x nie ma wartości"

↓

środowisko (environment) ewaluacji

```
> (define x 5)
x
> (+ x 3)
8
> (+ (* 5 x) 3)
28
> (+ x y)
```

↯ 'y nie ma wartości'

→ [ x : 5 ]

# Ewaluacja wyrażeń

(i) Ewaluj wszystkie podwyrażenia ("-" wszystkie elementy listy)

(ii) Stosuj wartość pierwszego podwyrażenia (która musi być funkcją) do wartości pozostałych podwyrażeń.

uwaga:

- wartość liczby, to ta liczba
- wartość operatorów - tzn. funkcji built-in - to kod, który system wykonuje
- wartość innych wyrażeń znajdują się w środowisku
- quote blokuje ewaluację

Przykład:

```
> (+ (* a 2) (* 3 b))
=> (+ (* 3 2) (* 3 b))
=> (+ 6 (* 3 b))
=> (+ 6 (* 3 4))
=> (+ 6 12)
=> 18
```

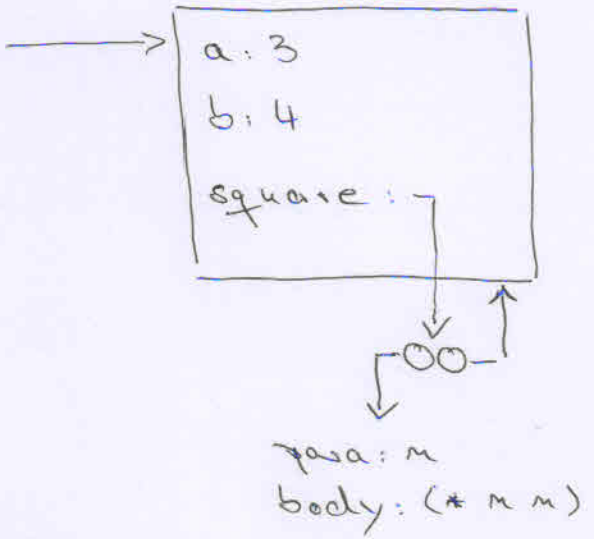
| a : 3 |
| --- |
| b : 4 |

```
> '(+ (* a 2) (* 3 b))
(+ (* a 2) (* 3 b))
```

# Funkcje

```
> (define (square n) (* n n))
square
```

```
> (square 5)
=> (square 4)
=> (* 4 4)
=> 16
```




```
> (square (square 3))
81
```

## Ewaluacja funkcji

Żeby zastosować funkcję z podanymi argumentami, ewaluj ciało funkcji, w którym formalne argumenty zostały zaktualizowany z aktualnymi (podanymi) argumentami.

↳ substitution model (dla aplikacji funkcji)

```
> (define (sum-of-squares x y)
    (+ (square x) (square y)))
sum-of-squares
> (define (f a)
    (sum-of-squares (+ a 1) (* b 2)))
```

```
> (f 5)
=> (sum-of-squares (+ 5 1) (* 5 2))
=> (sum-of-squares 6 (* 5 2))
=> (sum-of-squares 6 (* 4 2))
=> (sum-of-squares 6 8)
=> (+ (square 6) (square 8))
=> (+ (* 6 6) (square 8))
=> (+ 36 (square 8))
=> (+ 36 (* 8 8))
=> (+ 36 64)
=> 100
```

## Predykaty i warunki

```
> (< 2 0)
 #f
> (= 2 (+ 1 1))
 #t
> (and (= 1 1) (not (> 1 3)))
#t
```

```
(define (abs x)
    (cond ((> x 0) x)
          ((= x 0) 0)
          (#t      (- x))))
```

```
(if (< x 0)
    (- x)
    x)
```

Uwaga:

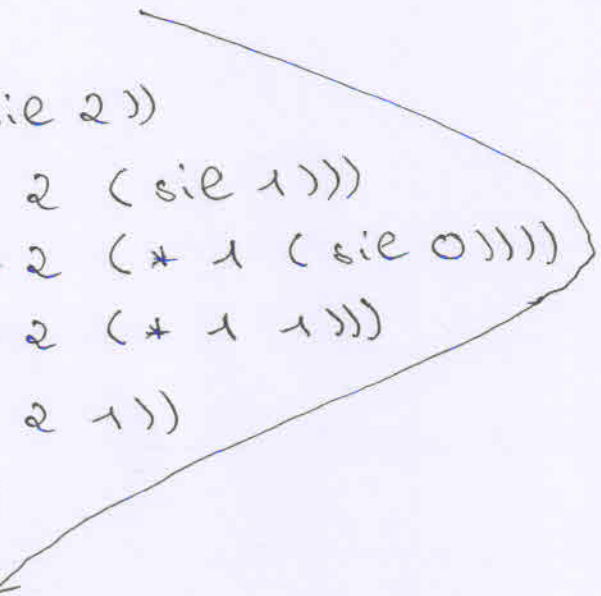
Nie wszystkich podwyrażeń cond'a i if'a zostaje ewaluowane: najpierw tylko pierwszy warunek a jeżeli jego wartość jest #t należące wyrażenie (którego wartością jest wynik ewaluacji). Jeżeli wartość warunku jest #f, kolejny warunek zostaje ewaluowany...

↳ cond i if nie są funkcjami, ale "special forms"

## Rekurencja

```
(define (sil n)
  (if (= n 0)
      1
      (* n (sil (- n 1)))))
```

```
(sil 3)
*=> (* 3 (sil 2))
*=> (* 3 (* 2 (sil 1)))
*=> (* 3 (* 2 (* 1 (sil 0))))
*=> (* 3 (* 2 (* 1 1)))
*=> (* 3 (* 2 1))
=> (* 3 2)
=> 6
```



↳ (linear) proces rekurencyjny

```
(define (sil-help prod licznik max)
  (if (> licznik max)
      prod
      (sil-help (* licznik prod)(+ licznik 1) max )))

(define (sil n)
  (sil-help 1 1 n))
```

```
(sil 3)
=> (sil-help 1 1 3)
=> (sil-help 1 2 3)
=> (sil-help 2 3 3)
=> (sil-help 6 4 3)
=> 6
```

↳ (linear) proces iteracyjne

## Information hiding

```
(define (sil n)
  (define (sil-help prod licznik max)
    (if (> licznik max)
        prod
        (sil-help (* licznik prod) (+ licznik 1) max)))
  (sil-help 1 1 n))
```

Przykład: Pierwiastki metodą Newtona

$$x_{i+1} := (x_i + \frac{x}{x_i}) / 2$$

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))

(define (sqrt x)
  (sqrt-iter 1 x))

(define (improve guess x)
  (average guess (/ x guess)))
(define (average x y)
  (/ (+ x y) 2))
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
```

```
> (sqrt 9)
3.00001
> (sqrt (+ 100 37))
11.7047
```

```
(define (sqrt x)
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess x)
    (average guess (/ x guess)))
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1 x))
```

lexical scoping