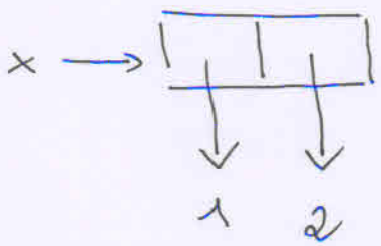
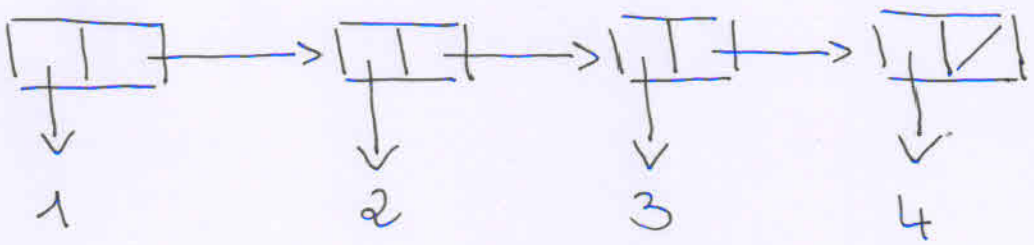
§3 O listach

cons, car, cdr

Podstawą list są pary:

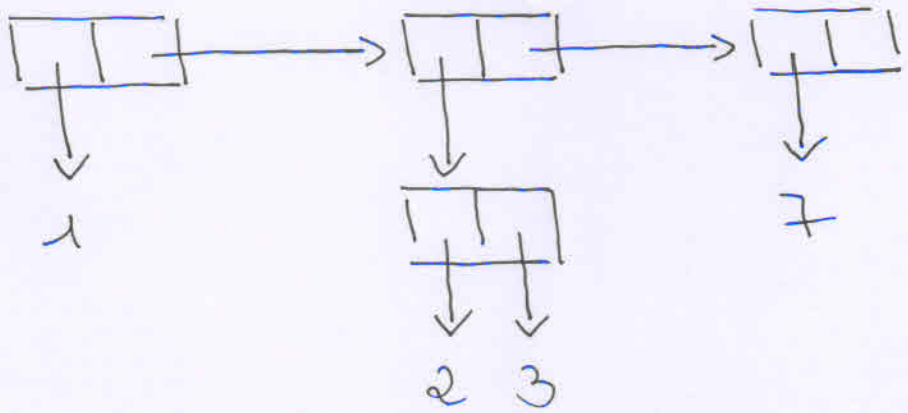
```
> (define x (cons 1 2))
x
> x
(1. 2)
> (car x)
1
> (cdr x)
2
```






```
> (cons 1 (cons 2 (cons 3 '())))
(1 2 3)
> (list 1 2 3)
(1 2 3)
> (cdr (list 1 2 3))
(2 3)
```

```
>(list 1 (cons 2 3) 7)
(1 (2.3) 7)
```




```
(define (length l)
   (if (null? l)
       0
       (+ 1 (length (cdr l)))))

> (length (list 1 2 3 4))
4
> (length (list 1 (cons 2 3) 7))
3
```

a listy z innymymi elementami?

```
> (list a b c)
```

↯ 'a nie ma wartości

quote

```
> (list 'a 'b)
(a b)
> '(a b)
(a b)
> (define a 1)
a
> (define b 2)
b
> (list a b)
(1 2)
> (list a 'b)
(1 b)
```

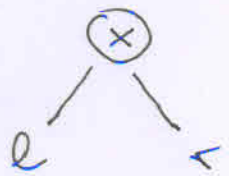
```
(define (member? x l)
  (cond ((null? l) #f)
        ((eq? (car l) x) #t)
        (#t (member? x (cdr l)))))
```

```
> (member? 'a '(1 b a 3))
#t
> (member? 'a '(1 b (2 a) 3))
#f
```

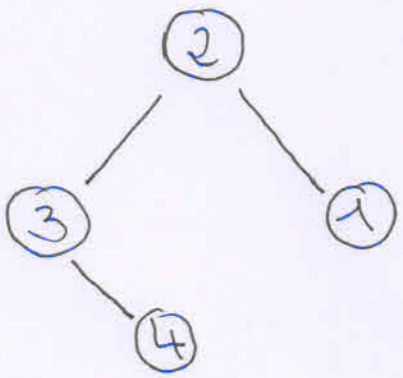
Drzewa binarne

`()` — puste drzewo

`(x l r)`




```
> (define drzewo '(2 (3 () (4 () ())) (1 () ())))
drzewo
```




```
(define (height t)
  (if (eq? t '())
      0
      (+ 1 (max (height (car (cdr t)))
                (height (car (cdr (cdr t))))))))
```

(car (cdr (cdr t))) = (caddr t)

```
> (height drzewo)
2
```

lepiej:

```
(define (is-null? t) (eq? '() t))
(define (left t) (cadr t))
(define (right t) (caddr t))

(define (height t)
   (if (is-null? t)
       0
       (+ 1 (max (left t) (right t)))))

(define (preorder t)
    (if (is-null? t)
        '()
        (append
            (cons (element t)
                  (preorder (lewe t)))
            (preorder (prawe t)))))

(define (element t) (car t))
```

Funkcja "times", która mnoże wszystkie elementy w t przez n ?

```
(define (times n t)
  (if (is-null? t)
      '()
      (make-tree (* (element t) n)
                 (times n (left t))
                 (times n (right t)))))
```

```
(define (make-tree x l r) (cons x (list l r)))
```

Przykład: Symbolic differentiation

$$\frac{dc}{dx} = 0 \quad , c \text{ stała lub } c \equiv y \neq x$$

$$\frac{dx}{dx} = 1$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(uv)}{dx} = u\left(\frac{dv}{dx}\right) + v\left(\frac{du}{dx}\right)$$

```
(define (deriv exp var)
  (cond ((constant? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum
             (deriv (addend exp) var)
             (deriv (augend exp) var)))
        ((product? sum)
         (make-sum
             (make-product .....)
             (make-product .... )))))
```

Reprezentacja

$ax + b \longrightarrow$ `'(+ (* a x) b)`

```
(define (constant? exp) (number? exp))
(define (variable? exp) (symbol? exp))
(define (same-variable? x y)
  (and (variable? x)
       (variable? y)
       (eq? x y)))
```

```
(define (sum? exp)
  (if (not (atom? exp))
      (eq? (car exp) '+)
      (error "....." )))

(define (addend sum) (cadr sum))
(define (augend sum) (caddr sum))
(define (make-sum a b) (list '+ a b))

> (deriv '(* x y) 'x)
(+ (* x 0) (* 1 y))
> (deriv '(+ (* a x) b) 'x)
(+ (+ (* a 1)(* x 0)) 0)
```

List Comprehension

```
(define (sum l)
  (if (null? l)
      0
      (+ (car l) (sum (cdr l)))))

(define (prod l)
  (if (null? l)
      1
      (* (car l) (prod (cdr l)))))
```

↳

```
(define (fold f e l)
  (if (null? l)
      e
      (f (car l) (fold f e (cdr l)))))
```

```
> (fold + 0 '(1 2 3 4))
10
> (fold * 1 '(1 2 3 4))
24
> (fold * 3 '(1 2 3 4))
72
```

czyli

```
(define (sum l) (fold + 0 l))
(define (prod l) (fold * 1 l))
```

```
> (sum '(1 2 3))
=> (fold + 0 '(1 2 3))
*=> (+ 1 (fold + 0 '(2 3)))
*=> (+ 1 (+ 2 (fold + 0 '(3))))
*=> (+ 1 (+ 2 (+ 3 (fold + 0 '()))))
*=> (+ 1 (+ 2 (+ 3 0)))
*=> 6
```

```
(define (fold f e)
  (lambda (l)
    (if (null? l)
        e
        (f (car l) ((fold f e) (cdr l))))))
```

```
> ((fold + 0) '(1 2 3 4))
10
> ((fold * 1) '(1 2 3 4))
24
```