



Funkcionálne programovanie

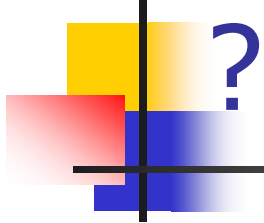
1-AIN-512/12

Peter Borovanský

I-18

<http://dai.fmph.uniba.sk/courses/FPRO/>

Prečo funkcionálne programovať?



- Because of their relative concision and simplicity, functional programs tend to be easier to reason about than imperative ones.
- Functional programming idioms are elegant and will help you become a better programmer in all languages.
- “The smartest programmers I know are functional programmers.” – one of my undergrad professors 😊



Funkcie v matematike

- Funkcia je typ zobrazenia, relácie, ...
- Funkcie sú rastúce, klesajúce, ..., derivujeme a integrujeme ich, ...
- Funkcie sú nad N ($N \rightarrow N$), R ($R \rightarrow R$), ...
- Funkcií je $N \rightarrow N$ veľa. Viac ako $|N|$?
- Funkcií $N \rightarrow N$ je toľko ako reálnych čísel...Vieme všetky naprogramovať ?
- Funkcií je $R \rightarrow R$ ešte viac. Naozaj ?
- Funkcie skladáme. To je asociatívna operácia... Je komutatívna ?
- $x \rightarrow 4$ je konštantná funkcia napr. typu $N \rightarrow N$, či $R \rightarrow R$, ...
- $\{x \rightarrow k \cdot x + q\}$ je množstvo (množina) lineárnych funkcií pre rôzne k, q
- Funkcia nie vždy má inverznú funkciu
- Ale skladať ich vieme vždy
 $(x \rightarrow 2 \cdot x + 1) \cdot (x \rightarrow 3 \cdot x + 5)$ je $x \rightarrow 2 \cdot (3 \cdot x + 5) + 1$, teda $x \rightarrow 6 \cdot x + 11$
- aj aplikovať v bode z definičného oboru $(x \rightarrow 6 \cdot x + 11) 2 = 23$
a lineárne funkcie sú uzavreté na skladanie

Prirodzené čísla nám dal sám dobrotivý
pán Boh, všetko ostatné je dielom človeka.
(L. Kronecker)

Funkcionálna apokalypsa ☺

- Predstavme si, že by existovali len funkcie
- a nič iné...
- Žiadne čísla, ani prirodzené, ani 0, ani True/False, ani NULL, či nil
- Vedeli by sme vybudovať matematiku, resp. aspoň aritmetiku ?
- vieme nájsť
 - funkcie, ktoré by zodpovedali číslam 0, 1, 2, ...
 - a operácie zodpovedajúce +, *, ...
- tak, aby to fungovalo ako $(\mathbb{N}, +, *)$





Jazyky podporujúce FP

- Funkcionálne programovanie je programovanie s funkciami ako hodnotami
- Kým sa objavilo funkcionálne programovanie, hodnoty boli: celé, reálne, komplexné číslo, znak, pole, zoznam, ...
- Funkcionálne programovanie rôzne jazyky rôzne podporujú:
 - Haskell
 - Scheme
 - Go, Groovy, Ruby, Scala, Python,
 - F#, Clojure, Erlang, ...
 - Pascal, C, Java, ...

Ktorý ako...



Funkcia ako argument

(Pascal/C)

Aj pred FP poznali posielanie funkcie ako argument

program example;

```
function first(function f(x: real): real): real;  
begin  
    first := f(1.0) + 2.0;  
end;
```

```
function second(x: real): real;  
begin  
    second := x/2.0;  
end;
```

```
begin  
    writeln(first(second));  
end.
```

To isté v jazyku C:

```
float first(float (*f)(float)) {  
    return (*f)(1.0)+2.0;  
    return f(1.0)+2.0; // alebo  
}
```

```
float second(float x) {  
    return (x/2.0);  
}
```

```
printf("%f\n",first(&second));
```

Podstatné: použiteľných funkcií je konečne veľa = len tie, ktoré sme v kóde definovali. Nijako nemôžeme dynamicky vyrobiť funkciu, s ktorou by sme pracovali ako s hodnotou.



Funkcia ako hodnota

(požičané z Go-ovského cvičenia, Programovacie paradigmy)

```
type realnaFunckia /*=*/ func(float64) float64

func kompozicia(f, g realnaFunckia) realnaFunckia {
    return (func(x float64) float64 { // kompozicia(f,g) = f.g
        return f(g(x)) // tu vzniká v run-time nová funkcia,
                        // ktorá nebola v čase kompilácie
    })
} // iteracia(n,f)=f^n

func iteracia(n int, f realnaFunckia) realnaFunckia {
    if n == 0 {
        return (func(x float64) float64 { return x }) //id
    } else {
        return kompozicia(f, iteracia(n-1, f))
    } // f . iter(n-1,f)
}
```



Python Closures

```
def addN(n):                # výsledkom addN je funkcia,  
    return (lambda x:n+x)  # ktorá k argumentu pripočína N
```

```
add5 = addN(5)              # toto je jedna funkcia x 5+x  
add1 = addN(1)              # toto je iná funkcia y 1+y  
                                # ... môžem ich vyrobiť neobmedzene veľa  
print(add5(10))             # 15  
print(add1(10))             # 11
```

```
def iteruj(n,f):             # výsledkom je funkcia fn  
    if n == 0:  
        return (lambda x:x) # identita  
    else:  
        return(lambda x:f(iteruj(n-1,f)(x))) # f(fn-1) = fn
```

```
add5SevenTimes = iteruj(7,add5) # +5(+5(+5(+5(+5(+5(+5(100)))))))  
print(add5SevenTimes(100))      # 135
```




Javascript Closures

```
function addN(n) {  
    return function(x) { return x+n};  
};  
add5 = addN(5);  
add1 = addN(1);  
writeln(add5(10));  
writeln(add1(10));  
  
function iteruj(n, f) {  
    if (n == 0)  
        return function(x) {return x;};  
    else  
        return function(x) { return iteruj(n-1,f)(f(x)); };  
}  
writeln(iteruj(7,add5)(100));
```

<http://math.chapman.edu/~jipsen/js/>



JDK8 - funkcionálny interface

```
interface FunkcionalnyInterface { // koncept funkcie v J8
    public void doit(String s);    // jediná "procedúra"
}

// „procedúra“ ako argument
public static void foo(FunkcionalnyInterface fi) {
    fi.doit("hello");
}

// „procedúra“ ako hodnota, výsledok
public static FunkcionalnyInterface goo() {
    return (String s) -> System.out.println(s + s);
}
```

```
foo(goo())
"hellohello"
```



JDK8 - funkcionálny interface

```
public interface FunkcionalnyInterface { //String->String
    public String doit(String s); // jediná "funkcia"
}

// "funkcia" ako argument
public static String foo(FunkcionalnyInterface fi) {
    return fi.doit("hello");
}

// "funkcia" ako hodnota
public static FunkcionalnyInterface goo() {
    return
        (String s) -> (s+s);
}

System.out.println(foo(goo()));
"hellohello"
```



JDK8 - funkcionálny interface

```
public interface RealnaFunkcia {
    public double doit(double s);    // funkcia R->R
}

public static RealnaFunkcia iterate(int n, RealnaFunkcia
    f) {
    if (n == 0)
        return (double d)->d;    // identita
    else {
        RealnaFunkcia rf = iterate(n-1, f);    // f^(n-1)
        return (double d)->f.doit(rf.doit(d));
    }
}

RealnaFunkcia rf = iterate(5, (double d)->d*2);
System.out.println(rf.doit(1));
```



Java 8

```
String[] pole = { "GULA", "cerven", "zelen", "ZALUD" };  
Comparator<String> comp =  
    (fst, snd) -> Integer.compare(fst.length(), snd.length());
```

```
Arrays.sort(pole, comp);  
for (String e : pole) System.out.println(e);
```

```
Arrays.sort(pole,  
    (String fst, String snd) ->  
        fst.toUpperCase().compareTo(snd.toUpperCase()));
```

```
for (String e : pole) System.out.println(e);
```



forEach, map, filter v Java8

```
class Karta {
    int hodnota;
    String farba;
    public Karta(int hodnota, String farba) { ... }
    public void setFarba(String farba) { ... }
    public int getHodnota() { ... }
    public void setHodnota(int hodnota) { ... }
    public String getFarba() { ... }
    public String toString() { ... }
}

List<Karta> karty = new ArrayList<Karta>();
karty.add(new Karta(7, "Gula"));
karty.add(new Karta(8, "Zalud"));
karty.add(new Karta(9, "Cerven"));
karty.add(new Karta(10, "Zelen"));
```



forEach, map, filter v Java8

[Gula/7, Zalud/8, Cerven/9, Zelen/10]

```
karty.forEach(k -> k.setFarba("Cerven"));
```

[Cerven/7, Cerven/8, Cerven/9, Cerven/10]

```
Stream<Karta> vacsieKartyStream =
```

```
    karty.stream().filter(k -> k.getHodnota() > 8);
```

```
List<Karta> vacsieKarty =
```

```
    vacsieKartyStream.collect(Collectors.toList());
```

[Cerven/9, Cerven/10]

```
List<Karta> vacsieKarty2 = karty
```

```
    .stream()
```

```
    .filter(k -> k.getHodnota() > 8)
```

```
    .collect(Collectors.toList());
```

[Cerven/9, Cerven/10]



forEach, map, filter v Java8

```
List<Karta> vacsieKarty3 = karty
```

```
.stream()  
.map(k->new Karta(k.getHodnota()+1,k.getFarba()))  
.filter(k -> k.getHodnota() > 8)  
.collect(Collectors.toList());
```

[Cerven/9, Cerven/10, Cerven/11]

```
List<Karta> vacsieKarty4 = karty
```

```
.stream()  
.parallel()  
.filter(k -> k.getHodnota() > 8)  
.sequential()  
.collect(Collectors.toList());
```

[Cerven/9, Cerven/10]



Trochu z histórie FP

- 1930, Alonso Church, lambda calculus
 - teoretický základ FP
 - kalkul funkcií: abstrakcia, aplikácia, kompozícia
 - Princeton: A.Church, A.Turing, J. von Neumann, K.Gödel - skúmajú formálne modely výpočtov
 - éra: WWII, prvý von Neumanovský počítač: Mark I (IBM), balistické tabuľky
- 1958, Haskell B.Curry, logika kombinátorov
 - alternatívny pohľad na funkcie, menej známy a populárny
 - „premenné vôbec nepotrebujeme“
- 1958, LISP, John McCarthy
 - implementácia lambda kalkulu na „von Neumanovskom HW“

Niektoré jazyky FP:

- 1.frakcia: Lisp, [Common Lisp](#), ..., [Scheme](#) ([MIT](#), [DrScheme](#), [Racket](#))
- 2.frakcia: [Miranda](#), Gofer, [Erlang](#), [Clean](#), [Haskell Platform](#)([Hugs](#)),



1960 LISP

- LISP je rekurzívny jazyk
- LISP je vhodný na list-processing
- LISP používa dynamickú alokáciu pamäte, GC
- LISP je skoro beztypový jazyk
- LISP používal dynamic scoping
- LISP má globálne premenné, priradenie, cykly a pod.
 - ale nič z toho vám neukážem ☺
- LISP je vhodný na prototypovanie a je *všelikde*
- Scheme je LISP dneška, má viacero implementácií, napr.



Scheme - syntax

`<Expr> ::= <Const> |
 <Ident> |
 (<Expr0> <Expr1> ... <Exprn>) |
 (lambda (<Ident1>...<Identn>) <Expr>) |
 (define <Ident> <Expr>)`

definícia funkcie:

```
(define gcd  
  (lambda (a b)  
    (if (= a b)  
        a  
        (if (> a b)  
            (gcd (- a b) b)  
            (gcd a (- b a))))))
```

volanie funkcie:

```
(gcd 12 18)  
6
```



Rekurzia na číslach

```
(define fac (lambda (n)
  (if (= n 0)
      1
      (* n (fac (- n 1))))))
```

(fac 100)
933262....000

```
(define fib (lambda (n)
  (if (= n 0)
      0
      (if (= n 1)
          1
          (+ (fib (- n 1)) (fib (- n 2)))))))
```

(fib 10)
55

```
(define ack (lambda (m n)
  (if (= m 0)
      (+ n 1)
      (if (= n 0)
          (ack (- m 1) 1)
          (ack (- m 1) (ack m (- n 1)))))))
```

(ack 3 3)
61

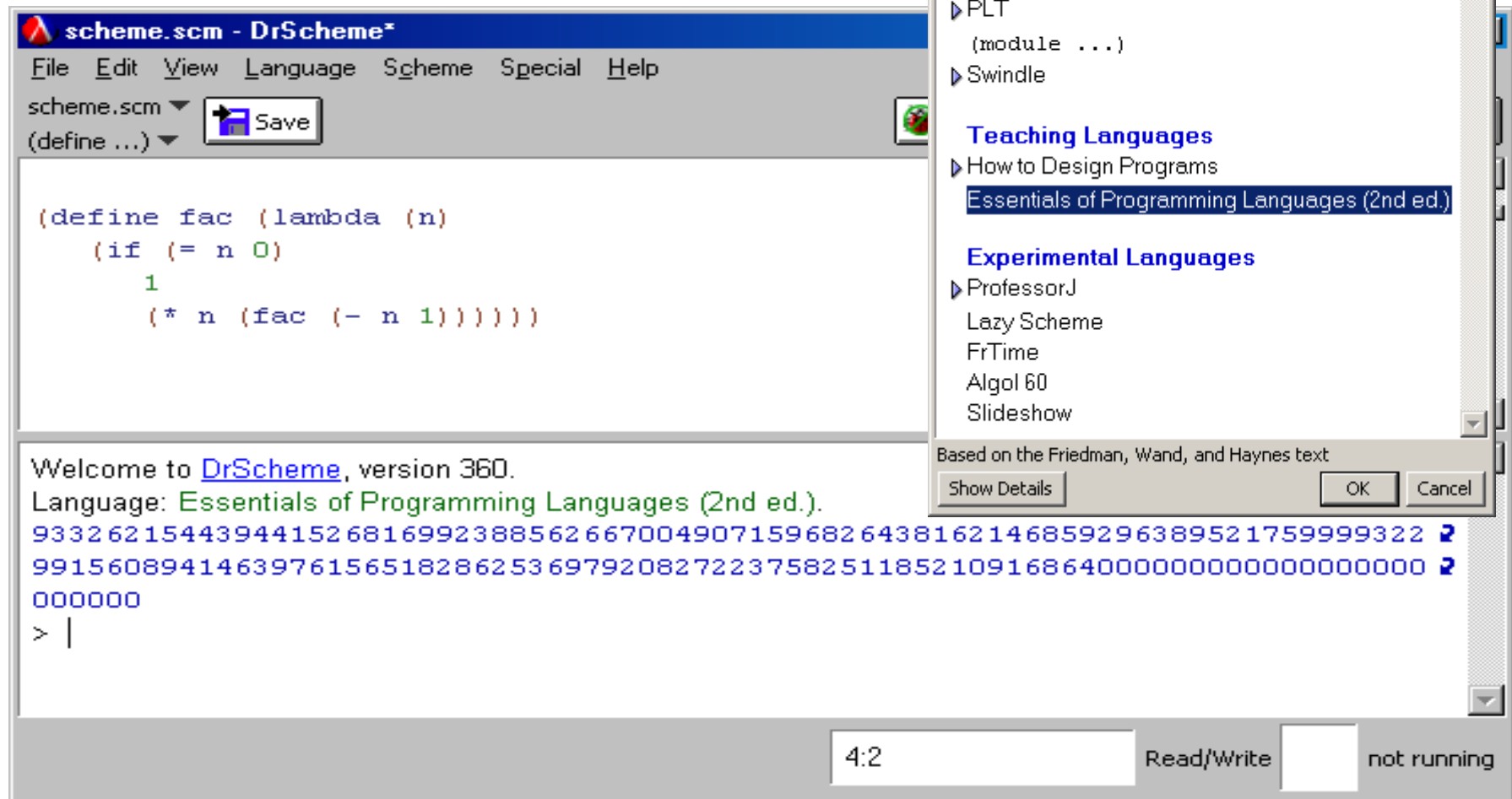
```
(define prime (lambda (n k)
  (if (> (* k k) n)
      #t
      (if (= (remainder n k) 0)
          #f
          (prime n (+ k 1))))))
```

```
(define isPrime?(lambda (n)
  (and (> n 1) (prime n 2))))
```

DrScheme

po inštalácii DrScheme treba
zvoliť si správnu úroveň jazyka
(t.j. advanced user, resp.
Essentials of PL)

<http://www.plt-scheme.org/software/drscheme/tour/tour-Z-H-4.html>

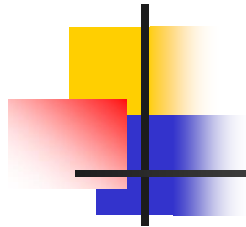




Čísla - help

- complex
- real
- rational
- integer

- $+$, $-$, $*$
`(* 3 (+ 5 7) 2) |--> 72`
- quotient, remainder, modulo
`(quotient 7 3) |--> 2`
`(remainder -11 3) |--> -2, (modulo -11 3) |--> 1`
- max, min, abs
`(max 1 2 3 4 5) |--> 5`
- gcd, lcm
`(gcd 18 12) |--> 6, (lcm 18 12) |--> 36`
- floor, ceiling
`(floor (/ 5 3)) |--> 1, (ceiling (/ 5 3)) |--> 2`
`(floor -4.3) |--> -5.0, (ceiling -4.3) |--> -4.0`
- truncate, round
`(truncate -4.3) |--> -4.0, (round -4.3) |--> -4.0`
- expt
`(expt 2 5) |--> 32`
- eq?, =, <, >, <=, >=
- zero?, positive?, negative?
- odd?, even?
`(odd? 2) |--> #f, (even? 2) |--> #t`



Scheme closures

```
(define (iterate n f)
  (if (= n 0)
      (lambda (x) x)
      (lambda (x) (f ((iterate (- n 1) f) x) ))
  )
)
```

```
(define addN5 (lambda (n) (+ n 5)))
(define addN1 (lambda (n) (+ n 1)))
```

```
((iterate 7 addN5 ) 100)
```

Zoznam je to, čo
má hlavu a chvost



Zoznam a nič iné

- má dva konštruktory
 - Scheme: `()`, `(cons h t)`
 - Haskell: `[]`, `h:t`
- vieme zapísať konštanty *typu* zoznam
 - Haskell: `1 : 2 : 3 : [] = [1,2,3]`
 - Scheme: `(cons 1 (cons 2 (cons 3 ())))`
- poznáme konvencie
 - Scheme: `(list 1 2 3)`, `'(1 2 3)`, `(QUOTE (1 2 3))`
- môžu byť heterogénne
 - Scheme: `'(1 (2 3) 4)`, `(list 1 ' (2 3) 4)`, `(list 1 (2 3) 4)`
 - Haskell: nie, vždy sú typu `List<t> = [t]`

procedure application:
expected procedure,
given: 2; arguments were: 3



K zoznamu vždy
pristupujeme cez hlavu

Car-Cdring v Lispe/Scheme

- **car**

(car '(1 2 3)) vráti 1

(car '((1 2) 3 4)) vráti (1 2)

- **cdr**

(cdr '(1 2 3)) vráti (2 3)

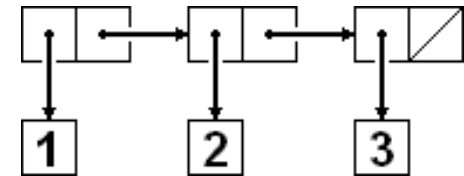
(cdr '((1 2) 3 4)) vráti (3 4)

- **cons**

(cons 1 '(2 3)) vráti (1 2 3)

- **quote**

'(1 2) je ekvivalentné (**quote** (1 2))



(cddr '(1 2 3))

(3)

(cddddr '(1 2 3))

()

(cadr '(1 2 3))

2

(caddr '(1 2 3))

3

(if vyraz ak-nie-je-nil ak-je-nil)

null? je #t ak argument je ()

Zoznamová rekurzia 1

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

(length (1 2 3))
3

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

(sum (1 2 3))
6

```
(define append
  (lambda (x y)
    (if (null? x)
        y
        (cons (car x)
                (append (cdr x) y)))))
```

zreťazenie zoznamov

(append (1 2 3) (a b c))
(1 2 3 a b c)

```
(define reverse
  (lambda (l)
    (if (null? l)
        '()
        (append (reverse (cdr l))
                  (list (car l))))))
```

otočenie zoznamu

(reverse '(1 2 3 4))
(4 3 2 1)

častá chyba

(cond (v1 r1) (v2 r2) (else r))

list? je #t ak argument je

zoznam

Zoznamová rekurzia 2

■ member ; nachádza sa v zozname
(define member
 (lambda (elem lis)
 (cond
 ((null? lis) '()) ; #f
 ((eq? elem (car lis)) #t)
 (else (member elem (cdr lis))))))

(member 3 '(1 2 3))

#t

(member '(1 2) '(1 2 (1 2) 3))

()

lebo (eq? '(1 2) '(1 2)) je #f

■ equal ; rovnosť dvoch zoznamov
(define equal

(lambda (lis1 lis2)

(cond

((not (list? lis1))(eq? lis1 lis2))

((not (list? lis2)) '()) ; #f

((null? lis1) (null? lis2))

((null? lis2) '())

((equal (car lis1) (car lis2))

(equal (cdr lis1) (cdr lis2)))

(else '()))))

(equal '(1 2) '(1 2))

#t

ak sa rovnajú hlavy,
porovnávame chvosty



Splošenie zoznamu - flat

```
(define flat (lambda (lis)
  (cond
    ((null? lis) lis) ; prázdny zoznam
    ((list? lis)      ; zoznam
     (append (flat (car lis)) (flat (cdr lis))))
    (else (list lis)))) ; atóm
```

```
(flat '(1 2 (3 (4 5) ()) (6 (7))))
(1 2 3 4 5 6 7)
```

porozmýšľajte, ako odstrániť append z tejto definície

ako napísať flat len s jedným rekurzívnym volaním



Mapping

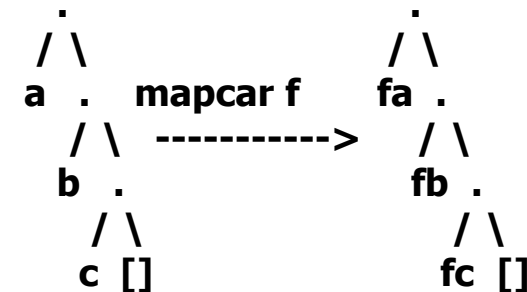
„ mapcar ; aplikuj funkciu na každý prvok zoznamu

```
(define (mapcar fun lis)
  (cond
    ((null? lis) '())
    (else (cons (fun (car lis))
                  (mapcar fun (cdr lis))))))
```

```
(mapcar fib '(1 2 3 4 5 6))
(1 1 2 3 5 8)
```

```
(mapcar (lambda (x) (* x x)) '(1 2 3 4 5 6))
(1 4 9 16 25 36)
```

konštanta typu funkcia





Syntax - help

- volanie fcie

(*<operator>* *<operand1>* ...) (max 2 3 4)

- funkcia

(lambda *<formals>* *<body>*) (lambda (x) (* x x))

- definícia funkcie

(define *<fname>* (lambda *<formals>* *<body>*))

- if, case, do, ...

(if <i><test></i> <i><then></i>)	(if (even? n) (quotient n 2))
(if <i><test></i> <i><then></i> <i><else></i>)	(if (even? n) (quotient n 2) (+ 1 (*3 x)))
(cond (<i><test1></i> <i><expr1></i>)	(cond ((= n 1) 1)
(<i><test2></i> <i><expr2></i>)	((even? n) (quotient n 2))
(<i><else></i> <i><exprn></i>))	(else (+ 1 (*3 x))))

- let

(let ((<i><var₁></i> <i><expr₁></i>) ...	
(<i><var_n></i> <i><expr_n></i>))	
<i><expr></i>)	(let ((x (+ n 1))) (* x x))