Poznámky z přednášek
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

# Neprocedurální programování
# BONUS

Peter Černo, 2010
petercerno@gmail.com

**Garant**: RNDr. Rudolf Kryl
**E-mail**: Rudolf.Kryl@mff.cuni.cz
**Domácí stránka**: `http://ksvi.mff.cuni.cz/~kryl/`

**Anotace**: Přednáška je věnována neprocedurálnímu programování. Většina semestru je věnována programování v jazyku Prolog, ve kterém studenti i ladí zápočtové programy. Informativně se studenti seznámí i s jazykem LISP a neprocedurálními částmi programovacích systémů.

**Sylabus**:

1. Tvar programu v Prologu a jeho interpretace, unifikace, backtracking. Deklarativní a operační sémantika programu.

2. Práce se seznamy, různé variace základních predikátů pro práci s nimi, efektivní obracení seznamu.

3. Aritmetika v Prologu, vyčíslení kontra unifikace.

4. Základní standardní predikáty (atom, atomic, number, name, func, arg, ...) predikáty pro práci s databází (assert, retract), predikáty grupování termů (bagof, setof) a jejich použití v programech). predikáty.

5. Vstup a výstup, definice operátorů.

6. Řez a jeho vliv na semantiku programů a jejich efektivitu. Definice negace a její vlastnosti.

7. Efektivita programů v prologu, neúplně definované datové struktury (např. rozdílové seznamy).

8. Logické programování a Prolog.

9. Funkcionální programování, základy jazyka LISP a programování v něm.

10. Haskell: zakladní syntax programů, definice typů a datových struktur, polymorfismus, porovnávání se vzorem (pattern matching), rekurze, líné vyhodnocování, funkce vyšších řádů, typové třídy a instance

**Literatura**:

1. Bratko I.: PROLOG Programming for Artificial Intelligence Addison-Wesley, Reading, Massachussets, 1986 ISBN 0-201-14224-4

2. Harold Abelson, Gerald Jay Sussman, Julie Sussman : Structure and Interpretation of Computer Programs Mc Graw-Hill Book Company 1985 ISBN 0-07-000-422-6

3. Paul Hudak, Joseph H. Fasel: A Gentle Introduction to Haskell, `http://www.haskell.org/`

This page is intentionally left blank.

19.01.2007 Fr
PETER ČERNO

# PRG005 NEPROC. PROG. - SKÚŠKA

## CYKLY V PERM.

```
permtocycle(Perm, Cycle) :-
   extend(Perm, ExPerm),
   extocc(ExPerm, Cycle).
extend(Perm, ExPerm) :-
   extend@(1, Perm, ExPerm).
extend(_, [], []).
extend(N, [P|Ps], [(N,P)|Qs]) :-
   N1 is N+1, extend(N1, Ps, Qs).
extocc([], []).
extocc([(_,B)|Qs], [C|Cs]) :-
   excycle(B, Qs, NewQs, C),
   extocc(NewQs, Cs).
excycle(B, Qs, NewQs, [B|Cs]) :-
   select((B,C), Qs, Qs2), !,
   excycle(C, Qs2, NewQs, Cs).
excycle(B, Qs, Qs, [B]).
```

```
data BST a = Nil
           | Branch (BST a) a (BST a)

build :: (Ord a) => Int -> [a] -> (BST a, [a])
build 0 xs = (Nil, xs)
build n xs = (Branch left x right, ys)
             where
               n1 = div (n-1) 2
               n2 = n-1-n1
               (left, ys1) = build n1 xs
               x = head ys1
               (right, ys) = build n2 (tail ys1)
```

```
sort xs = sortacc xs []
sortacc [] = id
sortacc (x:xs) = sortacc [y | y <- xs, y < x ] .
                 (x:) .
                 sortacc [y | y <- xs, y >= x ]

transpose (m, n, s) =
  (n, m, sort (map (\(i,j,a) -> (j,i,a)) s))
```

$$\begin{pmatrix} 4 \\ 2 & 3 \\ 2 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 3 & \\ & 2 & 5 \end{pmatrix} \begin{pmatrix} 10 & 9 \\ 14 & 6 \end{pmatrix}$$

```
multiply (m1, n1, s1) (m2, n2, s2)
  | n1 == m2 = add (sort [(i,j, a*b) |
                          (i,k,a) <- s1, (l,j,b) <- s2, k == l ])
  | otherwise = error "Incompatible dimensions"
  where
    add [] = []
    add [(i0, j0, a0)] = [(i0, j0, a0)]
    add ((i1, j1, a1) : (i2, j2, a2) : xs)
      | i1 == i2 && j1 == j2 = add ((i1, j1, a1+a2) : xs)
      | otherwise = (i1, j1, a1) : add ((i2, j2, a2) : xs)
```

19.01.2007 Fr
PETER ČERNO

# PRG005 NEPROC. PROG. - SKÚŠKA

## TRIEDENIE ZOZNAMOV PRIAMYM ZLIEVANÍM

```
tried (Xs, SortXs) :-
    nekles (Xs, Ys, Zs),
    slej (Ys, Zs, SortXs).

nekles ([], [], []).
nekles ([X], [X], []).
nekles ([X1, X2 | Xs], [X1 | Ys], Zs) :-
    X1 =< X2, nekles ([X2 | Xs], Ys, Zs).
nekles ([X1, X2 | Xs], [X1], [X2 | Xs]) :- X1 > X2.
slej (Xs, [], Xs) :-!.
slej (Xs, Ys, Zs) :-
    nekles (Y0s, Y1s, Y2s),
    merge (Xs, Y1s, X2s),
    slej (X2s, Y2s, Zs).
```

marad(kto, Co)  vlastni(kto, Co)

a) ZOZNAM TERMOV mnv (kto, veci ktore MaRad A Vlastni Ich)

```
makea(A) :-
  setof(
    (Kto, Mnv),
    Co^( (marad(Kto, Co); vlastni(Kto, Co)),
         findall(Y, (marad(Kto, Y), vlastni(Kto, Y)), Mniv) ),
    A ).
```

b) ZOZNAM TERMOV dar (Vec, ZoznamPrevodov)
ZoznamPrevodov JE TVARU d(Odkoho, Komu)

```
makeb(B) :-
  setof(
    (Co, ZoznamPrevodu),
    Kto^( (marad(Kto, Co); vlastni(Kto, Co)),
          setof( d(Odkoho, Komu),
                 (vlastni(Odkoho, Co), not(marad(Odkoho, Co)),
                  marad(Komu, Co), not(vlastni(Komu, Co))),
                 ZoznamPrevodu) ),
    B).
```

c) ZOZNAM TÍCH, ČO MAJÚ IBA TIE VECI, ČO MAJÚ RADI

```
makec(C) :-
  setof(
    Kto,
    Co^( (marad(Kto, Co); vlastni(Kto, Co)),
         findall(Vec,
                 ((vlastni(Kto, Vec), not(marad(kto, Vec)));
                  (marad(Kto, Vec), not(vlastni(Kto, Vec)))),
                 ZoznamVeci),
         ZoznamVeci = [] ),
    C ).
```

20.01.2007 Sa
PETER ČERNO

# PRG005 NEPROC. PROG. - SKÚŠKA

## DELENIE RIEDKYCH POLYNÓMOV

```
poldiv [] _ = ([], [])
poldiv p@((x,e):xs) q@((y,f):ys)
  | e < f = ([], p)
  | otherwise = let
      item = (x/y, e-f)
      poly = poldel (polmul q item)
      newp = poldel (polsub p poly)
      (frc, rem) = poldiv newp q
    in (item:frc, rem)
polmul [] _ = []
polmul ((x,e):xs) (y,f) = (x * y, eff): polmul xs (y,f)
polsub p [] = p
polsub [] ((y,f):ys) = (-y,f): polsub [] ys
polsub lhs@((x,e):xs) rhs@((y,f):ys)
  | e > f = (x,e): polsub xs rhs
  | e == f = (x-y,e): polsub xs ys
  | e < f = (-y,f): polsub lhs ys
poldel [] = []
poldel ((x,e):xs) | x == 0 = poldel xs
                  | otherwise = (x,e):poldel xs
```

## VŠETKY CESTY Z LISTOV DO KOREŇA

```
data NTree a = Nil
             | Node a [NTree a]

paths :: (NTree a) -> [[a]]
paths Nil = []
paths (Node x []) = [[x]]
paths (Node x subtrees) =
  [x:xs | subtree <- subtrees, xs <- paths subtree ]
```

## NÁJSŤ PODMNOŽINU XS TAKÚ, ŽE JEJ SÚČET JE NAJVÄČŠÍ MOŽNÝ ≤ n

```
add :: (Num a) => [(a,[b])]->(a,[b])->[(a,[b])]
add [] _ = []
add ((x,ys):ps) p@(x0,y0) =
  (x+x0, y0:ys) : add ps p
union :: (Ord a) => [(a,[b])]->[(a,[b])]->[(a,[b])]
union ps _ = ps
union _ qs = qs
union lhs@((x,ys):ps) rhs@((u,vs):qs)
  | x < u = (x,ys) : union ps rhs
  | x > u = (u,vs) : union lhs qs
  | x == u = (x,ys) : union ps qs
subset :: (Ord a, Num a) => a -> [a] -> [(a,[a])]
subset n [] = []
subset n (x:xs) = let
    sub = subset n xs
  in filter (\x -> (fst x <= n))
            (union sub ((x,[x]) : add sub (x,x)))
```

20.01.2007 Sa
PETER ČERNO

# PRG005 NEPROC. PROG. – SKÚŠKA

## TOPOLOGICKÉ USPORIADANIE GRAFU

```
type Graph a = [(a,[a])]

topology :: (Eq a) => (Graph a) -> [a]
topology graph = dfsvisititer graph (vertices graph) [] []

dfs :: (Eq a) => (Graph a) -> [a] -> [a] -> [a]
dfs graph [] black = black
dfs graph (x:xs) black
   | member x black = dfs graph xs black
   | otherwise =
       dfs graph xs (x: dfsvisit graph x [] black)

dfsvisit :: (Eq a) => (Graph a) -> a -> [a] -> [a] -> [a]
dfsvisit graph x grey black =
   dfsvisititer graph (edges graph x) (x:grey) black

dfsvisititer :: (Eq a) => (Graph a) -> [a] -> [a] -> [a] -> [a]
dfsvisititer graph [] grey black = black
dfsvisititer graph (x:xs) grey black
   | member x grey  = error "cycle"
   | member x black = dfsvisititer graph xs grey black
   | otherwise = dfsvisititer graph xs grey
       (x: dfsvisit graph x grey black )

member :: (Eq a) => a -> [a] -> Bool
member x [] = False
member x (y:ys) | x == y = True
                | otherwise = member x ys

vertices :: (Graph a) -> [a]
vertices = map fst

edges :: (Eq a) => (Graph a) -> a -> [a]
edges [] _ = []
edges ((x:xs):gs) y | x == y = xs
                    | otherwise = edges gs y
```

# PREHĽADÁVANIE DO ŠÍRKY

```
type Graph a = [(a, [a])]

bfs :: (Eq a) => (Graph a) -> [a] -> [a] -> [a]
bfs graph [] black = []
bfs graph (x:xs) black
  | member x black = bfs graph xs black
  | otherwise = x:bfs graph
      (xs ++ edges graph x) (x:black)
-- member, vertices, edges AKO DOMA
```

20.01.2007
PETER ČERNO

# PRG005 NEPROC. PROG. - SKUSKA

## TOPOLOGICKÉ USPORIADANIE

```
topology(Graph, Topology) :-
    vertices(Graph, Vertices),
    dfsiter(Graph, Vertices, [], [], Topology).
dfsvisit(Graph, V, Grey, Black, NewBlack) :-
    edges(Graph, V, Vertices),
    dfsiter(Graph, Vertices, [V|Grey], Black, NewBlack).
dfsiter(_, [], _, Black, Black).
dfsiter(_, [V|_], Grey, _, _) :-
    member(V, Grey), !, fail. % Cycle
dfsiter(Graph, [V|Vs], Grey, Black, NewBlack) :-
    member(V, Black), !
    dfsiter(Graph, Vs, Grey, Black, NewBlack).
dfsiter(Graph, [V|Vs], Grey, Black, NewBlack) :-
    dfsvisit(Graph, V, Grey, Black, Black2), !
    dfsiter(Graph, Vs, Grey [V|Black2], NewBlack).

vertices([], []).
vertices([(X,_)|Gs], [X|Ys]) :-
    vertices(Gs, Ys).

edges([], _, []).
edges([(X, Xs)|_], X, Xs) :- !.
edges([_|Gs], Y, Ys) :-
    edges(Gs, Y, Ys).
```

PREHĽADÁVANIE DO ŠÍRKY

```
bfs(Graph, Vertex, BfsList) :-
  bfs(Graph, [Vertex], [], BfsList).
bfs(_, [], _, []).
bfs(Graph, [V|Vs], Black, BfsList) :-
  member(V, Black), !,
  bfs(Graph, Vs, Black, BfsList).
bfs(Graph, [V|Vs], Black, [V|BfsList]) :-
  edges(Graph, V, Vertices),
  append(Vs, Vertices, NewVs),
  bfs(Graph, NewVs, [V|Black], BfsList).
% edges AKO MINULE
```

21.01.2007 Su
PETER ČERNO

# PRG005 NEPROC. PROG. - SKÚŠKA

## N-ÁRNY <-> BINÁRNY STROM

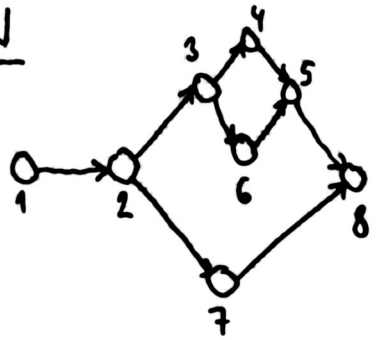
```
ntobin(nil, nil).
ntobin(NTree, BinTree) :- ntobiniter([NTree], BinTree).
ntobiniter([], nil).
ntobiniter([n(Head, Childs)|NTree], b(Left, Head, Right)) :-
    ntobiniter(Childs, Left),
    ntobiniter(NTree, Right).
```

```
data NTree a = NNil | Node a [NTree a]
data BTree a = BNil | Branch (BTree a) a (BTree a)

ntobin :: (NTree a) -> (BTree a)
ntobin NNil = BNil
ntobin ntree = ntobiniter [ntree]

ntobiniter :: [NTree a] -> (BTree a)
ntobiniter [] = BNil
ntobiniter (Node x childs : xs) = let
      left = ntobiniter childs
      right = ntobiniter xs
   in Branch left x right
```

JOIN




```
join(Graph, VerticesToJoin, NewGraph) :-
   vertices(Graph, Vertices),
   setsub(Vertices, VerticesToJoin, GoodVertices),
   [Pivot|_] = VerticesToJoin,
   makeedges(Graph, GoodVertices, Pivot, VerticesToJoin, TmpGraph),
   alledges(Graph, VerticesToJoin, OutEdges),
   NewGraph = [(Pivot, OutEdges)|TmpGraph].
vertices(Graph, Vertices) :- findall(V, member((V,_), Graph), Vertices).
edges(Graph, V, OutEdges) :- member((V, OutEdges), Graph), !.
edges(_,_,[]).
alledges(Graph, VTJ, OutEdges) :-
   alledges(Graph, VTJ, VTJ, [], OutEdges).
alledges(_,_,[],OutEdges,OutEdges).
alledges(Graph, VTJ, [X|Xs], Buffer, OutEdges) :-
   edges(Graph, X, XEdges),
   setsub(Edges, VTJ, XOutEdges),
   append(XOutEdges, Buffer, NewBuffer),
   alledges(Graph, VTJ, Xs, NewBuffer, OutEdges).
setsub([],_,[]).
setsub([X|Xs], Ys, Zs) :- member(X,Ys), !, setsub(Xs, Ys, Zs).
setsub([X|Xs], Ys, [X|Zs]) :- setsub(Xs, Ys, Zs).
makeedges(_,[],_,_,[]).
makeedges(Graph, [V|Vs], Pivot, VerticesToJoin, [(V,Edges)|Gs]) :-
   edges(Graph, V, OutEdges),
   selectedges(Graph, OutEdges, no, Pivot, VerticesToJoin, Edges),
   makeedges(Graph, Vs, Pivot, VerticesToJoin, Gs).
selectedges(_,[],no,_,_,[]).
selectedges(_,[],yes,Pivot,_,[Pivot]).
...
```

2003

21.01.2007 SU
PETER ČERNO

# PRG005 NEPROG. PROG. - SKÚŠKA

## 1-2 STRAN

```
data T12 a = Nil | N1 a (T12 a) |
             N2 a (T12 a) (T12 a)

fold :: b -> (a -> b -> b) -> (a -> b -> b -> b) -> (T12 a) -> b
fold p _ _ Nil = p
fold p f g (N1 x subt) = f x (fold p f g subt)
fold p f g (N2 x subt1 subt2) =
    g x (fold p f g subt1) (fold p f g subt2)

f0 :: a -> [a] -> [a]
f0 _ = id

g0 :: a -> [a] -> [a] -> [a]
g0 x l r = x : (l ++ r)
```

# KARTÉZSKÝ SOČIN GRAFOV

```
cart([], _, []).
cart([X|Xs], Ys, XYs) :-
    cartitem(X, Ys, XYs_a),
    cart(Xs, Ys, XYs_b),
    append(XYs_a, XYs_b, XYs).
cartitem(_, [], []).
cartitem(X, [Y|Ys], [(X,Y)|XYs]) :-
    cartitem(X, Ys, XYs).

cartgraph(nil, _, nil) :- !.
cartgraph(_, nil, nil) :- !.
cartgraph([], _, []).
cartgraph([(X,XChilds)|G1], G2, G1G2) :-
    cartgraphitem((X,XChilds), G2, G1G2_a),
    cartgraph(G1, G2, G1G2_b),
    append(G1G2_a, G1G2_b, G1G2).
cartgraphitem(_, [], []).
cartgraphitem((X,Childs), [(Y,YChilds)|G],
    [((X,Y), XYChilds)|XYG]) :-
    cart(XChilds, YChilds, XYChilds),
    cartgraphitem((X,XChilds), G, XYG).
```

21.01.2007 Su
PETER ČERNO

# PRG005 NEPROC. PROG. - SKÚŠKA

```
find :: (Num a, Ord a) => Int -> [a] -> [a]
find n set = take n (finditer 1 (sort set))
  where
    finditer k [] = k:finditer (k+1) []
    finditer k (x:xs) | k == x = finditer (k+1) xs
                      | otherwise = k:finditer (k+1) (x:xs)
```

18.01.2007 TT
PETER ČERNO

# PRG005 NEPROC. PROG.

## PROLOG

ATOM - SLOVO ZAČÍNAJÚCE MALÝM PÍSMENOM
KLAUZULA - FAKTY, PRAVIDLÁ (UKONČENÉ .)
PREDIKÁT < NULÁRNY ATOM, UNÁRNY VLASTNOSŤ, BINÁRNY VZŤAH,...
PRAVIDLO < HLAVA, TELO
PREMENNÁ - SLOVO ZAČÍNAJÚCE VEĽKÝM PÍSMENOM
ANONYMNÁ PREMENNÁ _

TERM < JEDNODUCHÝ TERM < KONŠTANTA < ATOM (radic, nam_pivok, a263 ..., , ; <===> ..., 'Adam' 'Horné Dolné') / ČÍSLO
JEDNODUCHÝ TERM < PREMENNÁ
TERM < ŠTRUKTÚRA - functor ( term1, ..., termN ) %arita N

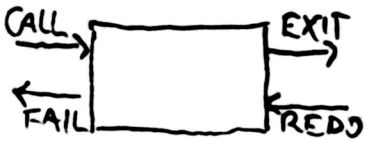
PROCEDÚRA = POSTUPNOSŤ KLAUZULÍ KT. HLAVY MAJÚ ROVNAKÝ FUNKTOR VRÁTANE ARITY (ZACHOVANÉ PORADIE)

KOMENTÁRE % /* ... */

KEDY SI DVA TERMY T1 A T2 ODPOVEDAJÚ, AKO PREBIEHA UNIFIKÁCIA?

PREMENNÁ NEDZNAČUJE MIESTO V PAMÄTI POČÍTAČA
ALE IBA MIESTO (V TERME), KAM MOŽNO SUBSTITUOVAŤ NEJAKÝ TERM.
NEEXISTUJÚ GLOBÁLNE PREMENNÉ

KRABIČKOVÝ MODEL VÝPOČTU




DEKLARATÍVNY VÝZNAM PROGRAMU = FORMULA VÝROKOVÉHO POČTU, KTORÁ ZODPOVEDÁ PROGRAMU
= KONJUNKCIA FORMULÍ, KT. UDÁVAJÚ VÝZNAM JEDN. KLAUZULÍ

FAKT p ... ≡ TVRDENIE p
PRAVIDLO p :- r1, ..., rn ≡ r1 ∧ ... ∧ rn ⇒ p

DEKLARATÍVNY VÝZNAM NEZÁVISÍ ANI NA PORADÍ KLAUZULÍ V PROGRAME, ANI NA PORADÍ PREDIKÁTOV V TELE JEDN. PRAVIDIEL

PLATÍ: KEĎ JE PROGRAM DEKLARATÍVNE SPRÁVNY, NIKDY NEDOSTANEME ZLÝ VÝSLEDOK

ARITMETIKA

=:= =\= < > =< >=
is

VSTUP / VÝSTUP

see, seen, seeing, tell, told, telling
read. write
getO, get, put, tab, nl

OPERÁTORY

op (?Priorita, ?Asociativita, ?Meno)
xfx xfy yfx
fx fy
xf yf

consult
reconsult
assert, assertz, asserta, retract, retractall

halt

atom, atomic, integer, var, nonvar
==, \==

Term =.. L

name (?Atom, ?List)

bagof
setof
findall

16.01.2007 Tu
PETER ČERNO

# PRG005 NEPROC. PROG. - SKÚŠKA

## HASKELL

HODNOTA (VALUE) (JE FIRST-CLASS!) MALÉ PÍSMENKÁ (inc)
KAŽDÝ VÝRAZ (EXPRESSION) MÁ TYP. (TYP NIE JE FIRST-CLASS)
TYPOVÝ VÝRAZ (TYPE EXPRESSION) ↑ VEĽKÉ PÍSMENO (Char)

TYPING : 5 :: Integer
'a' :: Char
↑
HAS TYPE

DECLARATION inc n = n+1
TYPE SIGNATURE DECLARATION inc :: Integer -> Integer
HASKELL MÁ STATICKÝ TYPOVÝ SYSTÉM, HASKELL JE TYPE SAFE
POLYMORFICKÉ TYPY [a] → TYPE VARIABLE (MALÉ PÍSMENKÁ)

PATTERN MATCHING

PRINCIPAL TYPE = NAJVŠEOBECNEJŠÍ VŠEOBECNÝ TYP, KT. OBSAHUJE VŠETKY INŠTANCIE VÝRAZU
EXISTENCIA JEDINEČNÉHO PRINCIPAL TYPU JE VLASTNOSŤ HINDLEY-MILNER TYPOVÉHO SYSTÉMU

data Bool = False | True
Bool JE NULÁRNY TYPE CONSTR. = SA NAZÝVA (disjoint) union, sum type
False, True SÚ NULÁRNE DATA CONSTR.

data Point a = Pt a a
SA NAZÝVA tuple type = KARTÉZSKY SÚČIN INÝCH TYPOV

[] JE TYPOVÝ KONŠTRUKTOR, KTORÝ Z TYPU t SPRAVÍ [t]
-> JE —II— , KTORÝ Z TYPOV t A u
SPRAVÍ TYP t->u : TYP FUNKCIE (MAPUJÚCEJ) TYP t NA TYP u

TYP Pt JE Pt :: a -> a -> Point a
Pt 2.0 2.0 :: Point Float

DATA KONŠTRUKTOR VYTVÁRA HODNOTU (VALUE)
TYPE KONŠTRUKTOR VYTVÁRA TYP (COMPILE TIME)

TYPE SYNONYMS
type Person = ([Char], Integer)

LIST COMPREHENSION  [f x | x <- xs]
GENERATOR

```
quicksort [] = []
quicksort (x:xs) = quicksort [y | y <- xs, y<x]
                   ++ [x]
                   ++ quicksort [y | y<-xs, y>=x]
```
GUARD

CURRIED FUNCTION
PARTIAL APPLICATION

FUNKCIE SÚ FIRST-CLASS ... HIGH-ORDER FUNCTIONS

INFIX OP. POZOSTÁVAJÚ IBA ZO SYMBOLOV
HASKELL NEMÁ PREFIXOVÉ OP. (- JE PREF. AJ INF.)

```
\x -> \y -> x+y   ≡   \x y -> x+y
```

PARTIAL APPLICATION OF AN INFIX OP. = SECTION
`(+y) ≡ \x -> x+y`

FUNKCIE SÚ NON-STRICT

FUNKCIA JE STRICT ⇔ KEĎ JE APLIKOVANÁ NA NEKONEČNÝ VÝRAZ, POTOM NESKONČÍ

DEFINITION ≠ ASSIGNMENT

EXISTUJE 1 HODNOTA, KT. ZDIEDAJÚ VŠETCI TYPY : ⊥

```
error :: String -> a        (HRACH ⊥)
```

PATTERN MATCHING
PATTERNS NIE SÚ FIRST-CLASS
JE IBA KONEČNE VEĽA RÔZNYCH PATTERNOV

DATA CONSTRUCTOR PATTERNS

LINEARITY = NIE POVOLENÝ VIAC AKO JEDEN VÝSKYT TOHO ISTÉHO FORMÁLNEHO PARAMETRU V PATTERNE

IRREFUTABLE = NEVER FAIL TO MATCH

```
FORMAL PAR.   : f x = x^2 - 1
AS-PATTERN    : f s@(x:xs) = x:s
WILD-CARDS    : head (x:_) = x
LAZY PATTERNS
```

10.01.2007
PETER OERNO

# PRG005 NEPROC. PROG. - SKÚŠKA

## HASKELL

PATTERN MATCHING ← FAIL / SUCCEED / DIVERGE (KEĎ HODNOTA POUŽITÁ V PATTERNOCH OBSAHUJE „error" ⊥)

MATCH : TOP-DOWN, LEFT-RIGHT

KEĎ VŠETKY SÚ FAILUJÚ ⇒ VÝSLEDOK JE ⊥

BOOLEAN GUARDS

CASE EXPRESSIONS

LAZY PATTERNS

PATTERN BINDINGS

LET EXPRESSION

```
let  y = a*b
     f x = (x+y)/y
in   f c + f d
```

WHERE CLAUSES

```
f x y | y>z = ...
      | otherwise = ...
  where z = ***
```

TYPE CLASSES & OVERLOADING

PARAMETRIC POLYMORPHISM

AD HOC POLYMORPHISM = OVERLOADING

```
class Eq a where
  (==) :: a -> a -> Bool
```

Eq a NIE JE TYPE EXPRESSION. SKÔR IDE O OHRANIČUJÚCU PODM. PRE TYP a. NAZÝVA SA KONTEXT (CONTEXT)

```
elem :: (Eq a) => a -> [a] -> Bool
```

INSTANCE DECLARATION

```
instance Eq Integer where
  x == y = x `integerEq` y
instance Eq Float where
  x == y = x `floatEq` y
```

```
instance (Eq a) => Eq (Tree a) where
  Leaf a == Leaf b = a == b
  (Branch l1 r1) == (Branch l2 r2) = (l1==l2) && (r1==r2)
  _ == _ = False
```

CLASS EXTENSION

```
class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a->a->Bool
  max, min :: a->a->a
```

```
class Functor f where
  fmap :: (a->b) -> fa -> fb

instance Functor Tree where
  fmap f (Leaf x) = Leaf (f x)
  fmap (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)
```

TYPE EXPRESSIONS JSI KLASIFICIRANE DO RAZNIH DRUHOV (KINDS)

```
Tree JE   * -> *
Tree Int JE *
```

NEWTYPE DECLARATION

```
newtype Natural = MakeNatural Integer

toNatural :: Integer -> Natural
toNatural x | x < 0 = error "Can't create negative naturals!"
            | otherwise = MakeNatural x

fromNatural :: Natural -> Integer
fromNatural (MakeNatural i) = i

instance Num Natural where
  fromInteger = toNatural
  x + y = toNatural (fromNatural x + fromNatural y)
  ...
```

2/3

17.01.2007 lvk
PETER ČERNO

# PRG005 NEPROC. PROG. - SKLÍŠKA

## FIELD LABELS

```
data Point = Pt {pointx, pointy :: Float}
abs Point :: Point -> Float
abs Point p = sqrt (pointx p * pointx p +
                    pointy p * pointy p)
abs Point (Pt {pointx = x, pointy = y}) = sqrt (x*x + y*y)
```

KEĎ JE p Point, POTOM p {pointx = 2} JE ...

```
data T = C1 {f :: Int, g :: Float}
       | C2 {f :: Int, h :: Bool}
```

AK JE x TYPU T, POTOM x {f = 5} BUDE FUNGOVAŤ...

STRICTNESS FLAG ! (IBA V DATA DEKLARÁCIACH)

```
data RealFloat a => Complex a = !a :+ !a
```

```
type ShowS = String -> String
showsTree :: (Show a) => Tree a -> ShowS
showsTree (Leaf x) = shows x
showsTree (Branch l r) = ('<':).showsTree l.('|':).showsTree r.('>':)
```

```
type ReadS a = String -> [(a, String)]
readsTree :: (Read a) => ReadS (Tree a)
readsTree ('<':s) = [(Branch l r, u) |
                     (l, '|':t) <- readsTree s,
                     (r, '>':u) <- readsTree t ]
readsTree s = [(Leaf x t) | (x,t) <- reads s]
```

Int, Integer, Float, Double sú základné typy (primitíva)

```
data (RealFloat a) => Complex a = !a :+ !a deriving (Eq, Text)

conjugate :: (RealFloat a) => Complex a -> Complex a
conjugate (x:+y) = x:+(-y)
```

POLIA (ARRAYS)

DVA PRÍSTUPY < INCREMENTAL
MONOLITHIC

```
import Array
```

Ix LIBRARY

```
class (Ord a) => Ix a where
   range   :: (a,a) -> [a]
   index   :: (a,a) a -> Int
   inRange :: (a,a) -> a -> Bool
```

a MOŽE BYŤ Int, Integer, Char, Bool ALEBO k-TICA (k≤5) NÁSOBKOV

```
range (0,4) => [0,1,2,3,4]
index ((0,0),(1,2)) (1,1) => 4
```

MONOLITHIC ARRAY CREATION

```
array :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
squares = array (1,100) [(i, i*i) | i <- [1..100]]

squares!7 => 49
bounds squares => (1,100)

mkArray :: (Ix a) => (a -> b) -> (a,a) -> Array a b
mkArray f bnds = array bnds [(x, f x) | x <- range bnds]

fibs :: Int -> Array Int Int
fibs n = a where a = array (0,n) ([(0,1),(1,1)] ++
                  [(i, a!(i-2) + a!(i-1)) | i <- [2..n]])
```

17.01.2007 MK
PETER ČERNO

# PRG005 NEPROC. PROG. - SKOŠKA

```
wavefront :: Int -> Array (Int, Int) Int
wavefront n = a where
    a = array ((1,1), (n,n))
        ( [((1,j), 1) | j <- [1..n]] ++
          [((i,1), 1) | i <- [1..n]] ++
          [((i,j), a!(i,j-1) + a!(i-1,j-1) + a!(i-1,j)) |
              i <- [2..n], j <- [2..n]] )
```

ACCUMULATION

```
accumArray :: (Ix a) -> (b -> c -> b) -> b -> (a,a) -> [Assoc a c]
                                                    -> Array a b
```

INCREMENTAL UPDATES

```
(//) :: (Ix a) => Array a b -> [(a,b)] -> Array a b
```

```
swapRows :: (Ix a, Ix b, Enum b) => a -> a ->
    Array (a,b) c -> Array (a,b) c
swapRows i1 i2 a = a //
    ( [((i1, j), a!(i2,j)) | j <- [jLo..jHi]] ++
      [((i2, j), a!(i1,j)) | j <- [jLo..jHi]])
    where ((iLo,jLo),(iHi, jHi)) = bounds a
```

```
swapRows i1 i2 a = a //
    [assoc | j <- [jLo..jHi],
             assoc <- [((i1, j), a!(i2,j)),
                       ((i2,j), a!(i1,j))] ]
    where ((iLo, jLo), (iHi, jHi)) = bounds a
```