

Funkcionálne programovanie

1mAINp

1mINFb

2mINFb

Peter Borovanský

I-18

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



Prečo funkcionálne programovať ?

(pohľad funkcionálneho programátora)

- 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 😊

Prečo funkcionálne programovať ?

(pohľad nefunkcionálneho (OOP) programátora)

funkcionálne programovanie

- je súčasťou moderných (aktuálne)... vznikajúcich jazykov,
 - Python, Go, Clojure, Scala, Swift, Haskell, Kotlin, TypeScript, ...
- a vkráda sa do jazykov klasicky procedurálnych/imperatívnych jazykov
 - Java (Java 8), C++ (c++ v.11)

hlavný konkurent OOP je v kríze (na diskusiu :-)

- **Object-Oriented Programming is Bad** – provokatívne video, ale stojí za to !
- základné princípy OOP (enkapsulácia, inheritance) sú zlé/nebezpečné,
- objekt (stav) je skrytý vstupno-výstupný argument metódy (funkcie),
- z pôvodne elegantných jazykov C, Java sú jazykové multi-paradigmatické monštrá,
- v snahe mať v jazyku všetko, strácame jednoduchosť a eleganciu (Java),
- Java vznikla na smetisku jazykov (C/C++, Pascal, VB) jednoduchá a elegantná,
- snaha očistiť (vytiahnuť esenciu) programovania, napr. Haskell, Go, Scala, ...

Prečo funkcionálne programovať ?

(pohľad front-end programátora)
(front-end v posledných rokoch reinkaroval FP)

- JavaScript je assembler internetu, a historicky Brendan Eich sa inšpiroval jazykom funkcionálnym jazykom Scheme, čo je klon jazyka Lisp
- ale JavaScript dostal C-like syntax, lebo Lisp/Scheme boli nečitateľné,
- jadro JavaScript(fy.Netscape), vzniklo za 10 dní v ére Java, Sun Microsystems
- pre lepšie pochopenie JavaScript treba prečítať Douglas Crockford [json]: **JavaScript: The Good Parts**, absolvovať kurz JS/ES6 (EmacsScript 2015)
- mnohé front-endové nástroje zakrývajú biedu samotného JavaScriptu (TS)
- JavaScript splnil úlohu *trojského koňa* funkcionálneho programovania do F-E
- front-end je v jadre asynchrónne programovanie, ale JS je single thread
- **callback** (*callback hell*), ES6 priniesol **promises**, ES7 **async/await**
- Reactive Functional Programming (RxJS) prinieslo **observables**
- za tým všetkým sú (skryté)

funkcie a
funkcionálne programovanie

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



https://en.wikipedia.org/wiki/ECMAScript#6th_Edition_-_ECMAScript_2015

Callback Hell

```
func1(param, function(err, res) {  
  func2(param, function(err, res) {  
    func3(param, function(err, res) {  
      func4(param, function(err, res) {  
        func5(param, function(err, res) {  
          func6(param, function(err, res) {  
            func7(param, function(err, res) {  
              func8(param, function(err, res) {  
                func9(param, function(err, res) {  
                  // Do something...  
                };  
              };  
            };  
          };  
        };  
      };  
    };  
  };  
});
```

```
const request = require('request');  
request('http://dai.fmph.uniba.sk/courses/FPRO/',  
  function (error, response, body) {  
    if (error) {  
      console.log('Error');  
    } else {  
      console.log('Success');  
    }  
  });
```

```
const request = require('request');  
request('http://dai.fmph.uniba.sk/courses/FPRO/',  
  function (firstError, firstResponse, firstBody) {  
    if (firstError){  
      console.log('first error');  
    } else {  
      console.log('first success');  
      request('http://dai.fmph.uniba.sk/courses/FPRO/{firstBody.path}',  
        function (secondError, secondResponse, secondBody) {  
          if(secondError){  
            console.log('second error');  
          } else {  
            console.log('second success');  
          }  
        });  
    }  
  });
```

(error, response, body) => {..}

Promises

(ES6 - chaining instead of nesting)

```
const axios = require('axios');
axios.get('http://dai.fmph.uniba.sk/courses/FPRO/')
  .then(function (response) {
    console.log('first success');
    return axios.get('http://dai.fmph.uniba.sk/courses/FPRO/index.html');
  })
  .then(function (response) {
    console.log('second success');
  })
  .catch(function (error) {
    console.log('fail');
  });
```

- promise má **.then()** a **.catch()**, majú za argument funkciu, čo robiť, ak nastal **úspech/error**
- funkcia môže vrátiť hodnotu alebo nový promise, umožňuje ich reťaziť nie vnárať
- promise musí mať **.catch()**

```
function getAsyncData(someValue){
  return new Promise(function(resolve, reject){
    getData(someValue,
      function(error, result){
        if (error) reject(error);
        else      resolve(result);
      })
  });
}
```



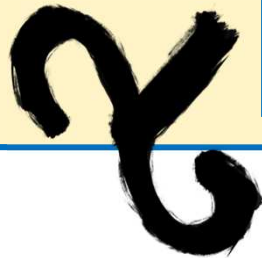


Async/Await

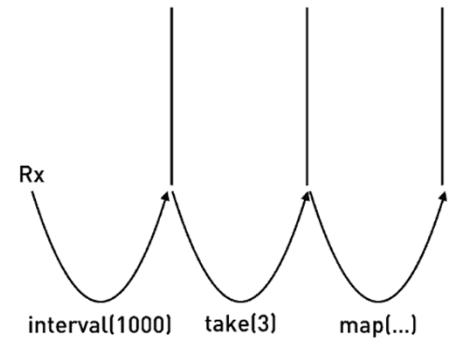
(ES7 syntax extension)

```
async function fetchTheFirstData(value) {  
    return await get("someUrl", value);  
}  
async function fetchTheSecondData(value){  
    return await getFromDatabase(value);  
}  
async function getSomeData(value){  
    try {  
        const firstResult = await fetchTheFirstData(value);  
        const result = await fetchTheSecondData(firstResult.someValue);  
        return result;  
    }  
    catch (error) {  
  
    }  
}
```

- **async/await** rozšírenie JS zakrylo funkcie
- kód vyzerá viac synchrónnejšie
- ale ľahko ho späť odmaskujete do promises
- funkcie sa nemajú zakrývať, treba im rozumieť



RxJS Observables



```
let obs = Rx.Observable
  .interval(1000)
  .take(100)
  .map((v) => v*v)
  .filter((w) => w%5 == 0)
```

```
obs.subscribe(value => console.log("Stvorce delitelne 5: " + value));
```

- "Stvorce delitelne 5: 0"
- "Stvorce delitelne 5: 25"
- "Stvorce delitelne 5: 100"
- "Stvorce delitelne 5: 225"
- "Stvorce delitelne 5: 400"
- "Stvorce delitelne 5: 625"
- "Stvorce delitelne 5: 900"
- "Stvorce delitelne 5: 1225"
- "Stvorce delitelne 5: 1600"
- "Stvorce delitelne 5: 2025"
- "Stvorce delitelne 5: 2500"



<https://codepen.io/mmiszy/pen/jGwzdY>



Odporúčané čítanie



Cristi Salcescu: These are the features in ES6 that you should know

<https://medium.freecodecamp.org/these-are-the-features-in-es6-that-you-should-know-1411194c71cb>

Jecelyn Yeen: JavaScript Promises for Dummies

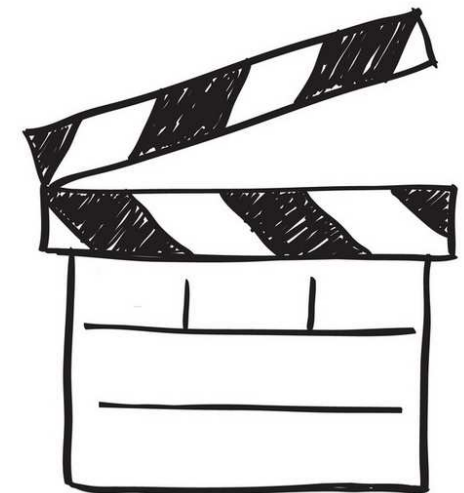
<https://scotch.io/tutorials/javascript-promises-for-dummies>

Sebastian Lindström: Getting to know asynchronous JavaScript: Callbacks, Promises and Async/Await

<https://medium.com/codebuddies/getting-to-know-asynchronous-javascript-callbacks-promises-and-async-await-17e0673281ee>

JS Compatibility Table:

<https://kangax.github.io/compat-table/es6/>



Software is getting slower more rapidly than hardware becomes faster. -- Nicolaus Wirth (Pascal)



Matematika – zdroj elegancie

- matematici zväčša nestratili zmysel po kráse (dôkazov), elegancii (definícií),
- kým informatici či programátori vylepšujú (efektívnosť) svoje algoritmy nie vždy s cieľom, aby boli čitateľnejšie, elegantnejšie, a často ani nie sú ...
- pri týchto transformáciach je často veľa matematiky (2., 3. prednáška)
- programátori sa primárne sústredia na korektnosť (*inak tam máš bug*),
- eleganciu (ako nevyhnutnosť) riešia, až keď sa to už nedá čítať/udržiavať,
- a to prichádza skoro...
 - nepriamo úmerne veľkosti tímu, kódu, ...
 - priamo úmerne zručnostiam, technikám, dodržiavaným pravidlám
- majú na to metodológie, metodiky(clean code), odporúčenia, code-checkery

The speed of software halves every 18 months.

-- Bill Gates (povedal a vypustil Windows10)



Funkcie v matematike

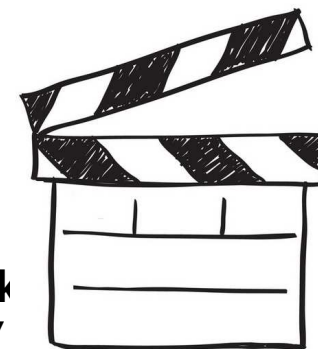
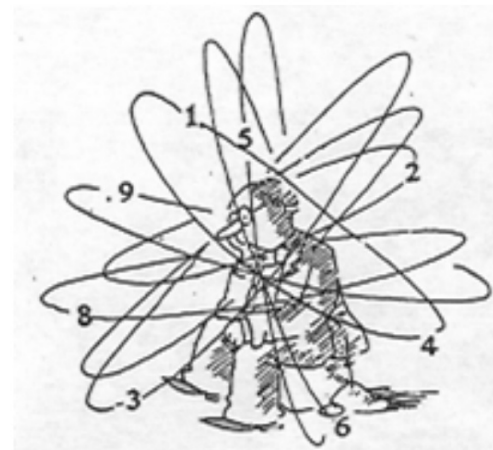
- Funkcia je typ zobrazenia, relácie, ...
- Funkcie sú rastúce, klesajúce, ..., derivujeme a integrujeme ich, ...
- Funkcie sú nad N ($N \rightarrow N$), R ($R \rightarrow R$), ...
- Funkcií je $N \rightarrow N$ veľa. Viac ako $|N|$?
- Funkcií $N \rightarrow N$ je toľko ako reálnych čísel... Vieme všetky 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

Prírodné 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 prírodné, ani 0, ani True/False, ani NULL, či nil
- vedeli by sme vybudovať matematiku,
resp. aspoň aritmetiku ?
resp. Programovací jazyk ?
- vieme nájsť
 - funkcie, ktoré by zodpovedali číslam 0, 1, 2, ...
 - a operácie zodpovedajúce +, *, ...
- tak, aby to fungovalo ako $(\mathbb{N}, +, *)$
- lebo ak áno,
 - podľa Kroneckera sme zachránení
 - podľa Gödela sme stratení
 - v každom formálnom systéme, ktorý obsahuje aspoň **aritmetik**
prírodných čísiel, existujú výroky, ktoré sa nedajú odvodiť





Prečo na FP záleží



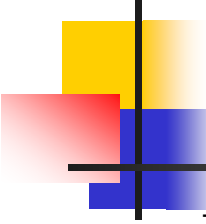
Ako úvodnú – motivačnú prednášku reprodukujem
prvú časť prednášky od John Hughes:
Why Functional Programming Matters
z λ -days, Krakow 2017

original (0-20min):

<https://www.youtube.com/watch?v=XrNdvWqxBvA>

originálny papier, 1984:

www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf



Tento kód je obsahom talku

(Churchove čísla v Haskellu)

```
true  x y = x
false x y = y

ifte  c t e = c t e

two   f x = f (f x)
one   f x = f x
zero  f x = x

incr n f x = f (n f x)

add    m n f x = m f (n f x)
mul    m n f x = m (n f) x

isZero n =  n (\_ -> false) true

decr n = n (\m f x -> f (m incr zero))
        zero
        (\x -> x)
        zero

fact :: (forall a. (a->a)->a->a) -> (a->a) -> a -> a
fact n =
    ifte (isZero n)
        one
        (mul n (fact (decr n)))

main =
    -- print $ (decr (add (mul two two) one)) (+1) 0
    -- print $ (fact (add (mul two two) one)) (+1) 0
    print $ (fact (add two
                    (add (mul two two) (mul two two))))
            (+1) 0

-- 3628800
-- (4.75 secs, 2,598,673,208 bytes)
```

<https://github.com/Funkcionalne/Prednasky/blob/master/01/src/Church.hs>

Odporúčané čítanie



- Hughes (video): <https://www.youtube.com/watch?v=XrNdVWqxBvA>
- Hughes (papier): www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf

z videa:

- Ladin: <https://www.cs.cmu.edu/~crary/819-f09/Landin66.pdf>
- Backus: https://www.thocp.net/biographies/papers/backus_turingaward_lecture.pdf
- Hudak: haskell.cs.yale.edu/wp-content/.../03/HaskellVsAda-NSWC.pdf

Best history FP overview:

- Hudak: <https://ccrma.stanford.edu/~jos/pdf/FunctionalProgramming-p359-hudak.pdf>



Haskell in FB spam filtering

Fighting spam with Haskell



Simon Marlow

Prečo Haskell vyhral:

1. Čistý funkcionálny, striktne typovaný
2. Konkurencia kdekoľvek sa len dá (Haxl)
3. Deployment pravidiel ľahko a rýchlo
4. Rýchlosť exekúcie
5. Interaktívny vývoj

One of our weapons in the fight against spam, malware, and other abuse on Facebook is a system called Sigma. Its job is to proactively identify malicious actions on Facebook, such as spam, phishing attacks, posting links to malware, etc. Bad content detected by Sigma is removed automatically so that it doesn't show up in your News Feed.

We recently completed a two-year-long major redesign of Sigma, which involved replacing the **in-house FXL language** previously used to program Sigma with **Haskell**. The Haskell-powered Sigma now runs in production, serving more than one million requests per second.

<https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/>

Elixir-Erlang

Inside Erlang, The Rare Programming Language Behind WhatsApp's Success

Facebook's \$19 billion acquisition is winning the messaging wars thanks to an unusual programming language.



<https://www.fastcompany.com/3026758/inside-erlang-the-rare-programming-language-behind-whatsapps-success>



Jazyky podporujúce FP

- funkcionálne programovanie je programovanie s funkciami ako hodnotami
- funkcionálne programovanie **nie je** len programovanie funkcií (bolo aj c++)
- funkcionálne programovanie **nie je** jazdenie po zoznamoch (bolo aj pythone)
- funkcionálne programovanie **nie je** v jazyku bez closures

- kým sa objavilo funkcionálne programovanie, hodnoty boli: celé, reálne, komplexné číslo, znak, pole, zoznam, ...
- funkcionálne programovanie rôzne jazyky rôzne podporujú:
 - Haskell, Scheme, Go, Scala, Python, ...
 - Groovy, Ruby, F#, Clojure, Erlang, ...

 - Pascal, C, Java, ...

Ktorý ako...



Funkcionálne jazyky

- Lisp (McCarthy, 1960)
- Iswim (Landin, 1966)
- Scheme (Steele and Sussman, 1975)
- ML (Milner, Gordon, Wadsworth, 1979)
- Haskell (Hudak, Hughes, Peyton Jones, and Wadler, 1987)
- O'Caml (Leroy, 1996)
- Erlang (Armstrong, Virding, Williams, 1996)
- Scala (Odersky, 2004)
- F# (Syme, 2006)
- Clojure (Hickey, 2007)
- Elm (Czaplicki, 2012)



Funkcia ako argument

(Pascal/C - Ktorý ako...)

pred FP poznali funkcie ako argumenty, ale *neučili to (na FMFI) na Pascale ani C ...*

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);
}

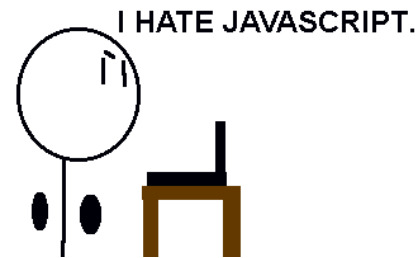
printf("%f\n",first(&second));
```

Podstatné: Pascal ani C nemajú closures - 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.



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  $x \rightarrow 5+x$   
add1 = addN(1)              # toto je iná funkcia  $y \rightarrow 1+y$   
                                # ... môžem ich vyrobiť neobmedzene veľa  
  
print(add5(10))             # 15  
print(add1(10))             # 11  
  
def iteruj(n,f):            # výsledkom je funkcia  $f^n$   
    if n == 0:  
        return (lambda x:x) # identita  
    else:  
        return(lambda x:f(iteruj(n-1,f)(x))) #  $f(f^{n-1}) = f^n$   
  
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 };  
}
```

-- výsledkom tejto funkcie je funkcia
-- a toto je closure, fcia
-- jej komponenty odkazujú na objekty mimo argumentov funkcie

```
function iteruj(n, f) {  
    if (n === 0)  
        return function(x) {return x;};  
    else  
        return function(x) { return iteruj(n-1,f)(f(x)); };  
}
```

```
add5 = addN(5);  
add1 = addN(1);
```

```
Native Browser JavaScript  
=> add5(100)  
105  
=> add1(10)  
11  
=> iteruj(7,add5)  
[Function]  
=> iteruj(7,add5)(100)  
135  
=> iteruj(7,iteruj(4,add1))(100)  
128
```



Clojure

Closure a Scope

(príklad je v javascript)

```
function f() {  
  y = 1  
  return function (x) { return x + y } -- closure - funkcia, ktorá viaže nelok.premennú y  
}  
function g() {  
  y = 2  
  h = f() -- výsledkom je funkcia (closure), ktorú aplikujeme na 10  
  console.log(h(10)) -- otázka s akou hodnotou y sa vykoná sčítanie, y=1 alebo y=2  
}  
g()
```

Odpovede:

- 11 – lexical / static scoping vlastný „normálnym“ moderným jazykom, Java, C++, ...
- 12 – dynamic scoping Basic, Lisp, CommonLisp, ... ale nie Scheme



forEach, map, filter

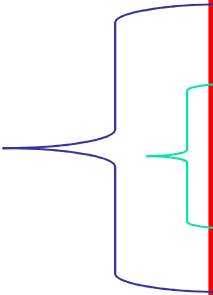
(Java8 Stream API)

```
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());
```

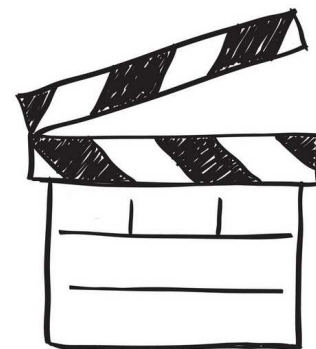
[Cerven/9, Cerven/10]

Break

(reklamná prestávka – Scala publicity)



If I were to pick a language to use today other than Java,
it would be Scala
-- James Gosling





Haskell

(typy v kocke)

- typovaný jazyk, teda má typy, napr. Int, Integer, Bool, Char, Float, String,...
- typové konštruktory:
 - n-tica (t_1, \dots, t_n) , zoznam $[t]$, napr. (Int, Int), [Int], [String]
 - funkčné typy $t_1 \rightarrow t_2$, napr. Int \rightarrow Int, Int \rightarrow Int \rightarrow Int
 - operátor \rightarrow je pravo-asociatívny, preto Int \rightarrow Int \rightarrow Int znamená
 - Int \rightarrow (Int \rightarrow Int)
 - funkcia, ktorá pre celé číslo vráti celočíselnú funkciu
 - a nie (Int \rightarrow Int) \rightarrow Int
 - funkcia, ktorá pre celočíselnú funkciu vráti číslo.
 - zátvorkovanie ruší defaultnú pravú asociativitu

Príklad:

```
iteruj    :: Int -> ((Float -> Float) -> (Float -> Float) )
(.)       :: (Float -> Float) -> ((Float -> Float) -> (Float -> Float)) – skladanie
iteruj    :: Int -> ( (t -> t) -> (t -> t) ) – pre každé t – polymorfizmus
(.)       :: (v -> w) -> ((u -> v) -> (u -> w))
```



Haskell

(aplikácia v kocke)

- funkciu $f :: A \rightarrow B$ môžeme aplikovať na argument typu A , výsledkom je B
- $f\ a$, alebo $(f\ a)$, nie ako v iných jazykoch $f(a)$

Napr.

`sin :: Float -> Float`

`(+5) :: Int -> Int`

`iteruj 7 :: (Float -> Float) -> (Float -> Float)`

`(iteruj 7) sin :: Float -> Float`

`((iteruj 7) sin) pi :: Float` `-- sin (sin (sin (sin (sin (sin (sin (pi))))))))`

`iteruj 4 :: (Int -> Int) -> (Int -> Int)`

`iteruj 4 (+5) :: (Int -> Int)`

`(iteruj 3) (iteruj 4 (+5)) :: (Int -> Int)`

`((iteruj 3) (iteruj 4 (+5))) 10 :: Int`



Haskell

(definícia v kocke)

```
iteruj    :: Int -> ((Float -> Float) -> (Float -> Float) )
```

```
iteruj    :: Int -> ((t -> t) -> (t -> t) ) – pre každé t – polymorfizmus
```

iteruj n f znamená $f^n = f \circ f \circ \dots \circ f$, teda iteruj n f x znamená $f^n(x)$

```
iteruj 0 f = (\x -> x)                -- identita
```

```
iteruj n f = (\x -> f (iteruj (n-1) f x))
```

alebo

```
iteruj n f = (\x -> iteruj (n-1) f (f x))
```

alebo

```
iteruj 0 f x = x
```

```
iteruj n f x = f (iteruj (n-1) f x)
```

alebo

```
iteruj n f = f . (iteruj (n-1) f)
```

alebo

```
iteruj n f = (iteruj (n-1) f) . f
```

(pre pokročilejších fajňšmekrov)

```
iteruj_foldr      :: Int -> ((t -> t) -> (t -> t) )
iteruj_foldr n f  = foldr (.) id (replicate n f)
```

```
replicate :: Int -> t -> [t]
replicate 5 13 = [13,13,13,13,13]
replicate 5 f  = [f,f,f,f,f]
```

```
iteruj_foldl      :: Int -> ((t -> t) -> (t -> t) )
iteruj_foldl n f  = foldl (.) id (take n (cycle [f]))
```

[illegible]

```
iteruj_using_iterate      :: Int -> ((t -> t) -> (t -> t))
iteruj_using_iterate n f x = iterate f x !! n
```

```
iterate :: (t->t) -> t -> [t]
iterate f x = [x, fx, ffx, fffx, fffffx, ...]
[x, fx, ffx, fffx, fffffx, ...]!!2 = ffx
```

```
iteruj_funkciu      :: Int -> ((t -> t) -> (t -> t))
iteruj_funkciu n f  = iteruj_f !! n
```

```
where iteruj_f = id:[f . g | g <- iteruj_f]
```

Tu je miesto pre vašu najzvrhlejšiu definíciu iterate

Haskell

(Hunit testy)

Iteruj.hs

module Iteruj where

TestIteruj.hs

module TestIteruj where

import Iteruj

import Test.Hunit

main = do

runTestTT \$ TestList [

TestList [

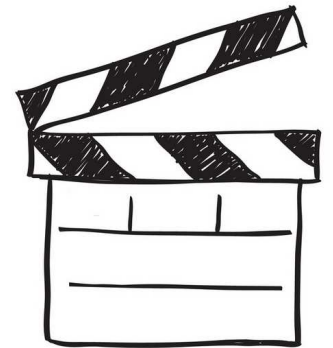
Testcase \$ assertEquals "iteruj 5 (+4) 100" 120 (iteruj 5 (+4) 100),

Testcase \$ assertEquals "iteruj 5 (*4) 100 " 102400 (iteruj 5 (*4) 100),

Testcase \$ assertEquals "iteruj 5 (++ab) c " "cababababab,,

(iteruj 5 (++"ab") "c")],

Iteruj.hs





Predchádzalo vzniku FP

- 19.storočie – DeMorgan, Boole – výrokový a predikátový počet
- 19.storočie – Peano – teória čísel, indukcia
- 20.storočie – Russel, Gödel, Hilbert – Princípy matematiky
- ~1930 ... ~1950 – vypočítateľnosť
 - Turingove stroje – krok, stav, abeceda, výpočet
 - Rekurzívne funkcie (Kleene) – štruktúra funkcií podľa ich konštrukcie, napr. primitívne rekurzívne funkcie \subsetneq vypočítateľné (Ackermannova fcia),
 - λ -kalkul (Church) – abstrakcia a aplikácia,
 - Markovove algoritmy - symbolické úpravy reťazcov, prepisovací systém
- všetky modely (i mnohé ďalšie) sú ekvivalentné,
 - zastavenie Turingovho stroja,
 - zastavenie programu v Pascale-Jave-Pythone...
 - výpočet v λ -kalkul,
 - výpočet rekurzívnej funkcie dá výsledok
- sú rovnako ťažké (nemožné) úlohy

Pôvodne vôbec nešlo o počítanie, ale vypočítateľnosť, či matematiku

- ak počítali, tak to vyzeralo takto [The Bombe @ Bletchley.mov](#)



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“
- 1960, SECD (**S**ta**C**k-**E**nvironment-**C**ontrol-**D**ump) Machine, Landin
 - predchodca p-code, rôznych stack-orientovaných bajt-kódov, virtuálnych mašín.
 - SECD použili pri implementácii
 - Algol 60, PL/1 – predchodcu Pascalu
 - LISP – prvého funkcionálneho jazyka založenom na λ -kalkule



Jazyky FP

1977, John Backus, IBM – odpálil boom rôznych FP jazykov

Can Programming Be Liberated from the von Neumann Style?

A Functional Style and Its Algebra of Programs

- 1.frakcia:

- Lisp, Common Lisp, ..., Scheme (MIT, DrScheme, Racket),
- ML, Standard ML, CAML, oCAML, ...

- 2.frakcia:

- Miranda, Gofer, Hope, Erlang, Clean, Hugs, Haskell Platform



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 (= (mod n k) 0)
          #f
          (prime n (+ k 1))))))
```

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



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)
((iterate 7 (iterate 4 addN1)) 100)
```



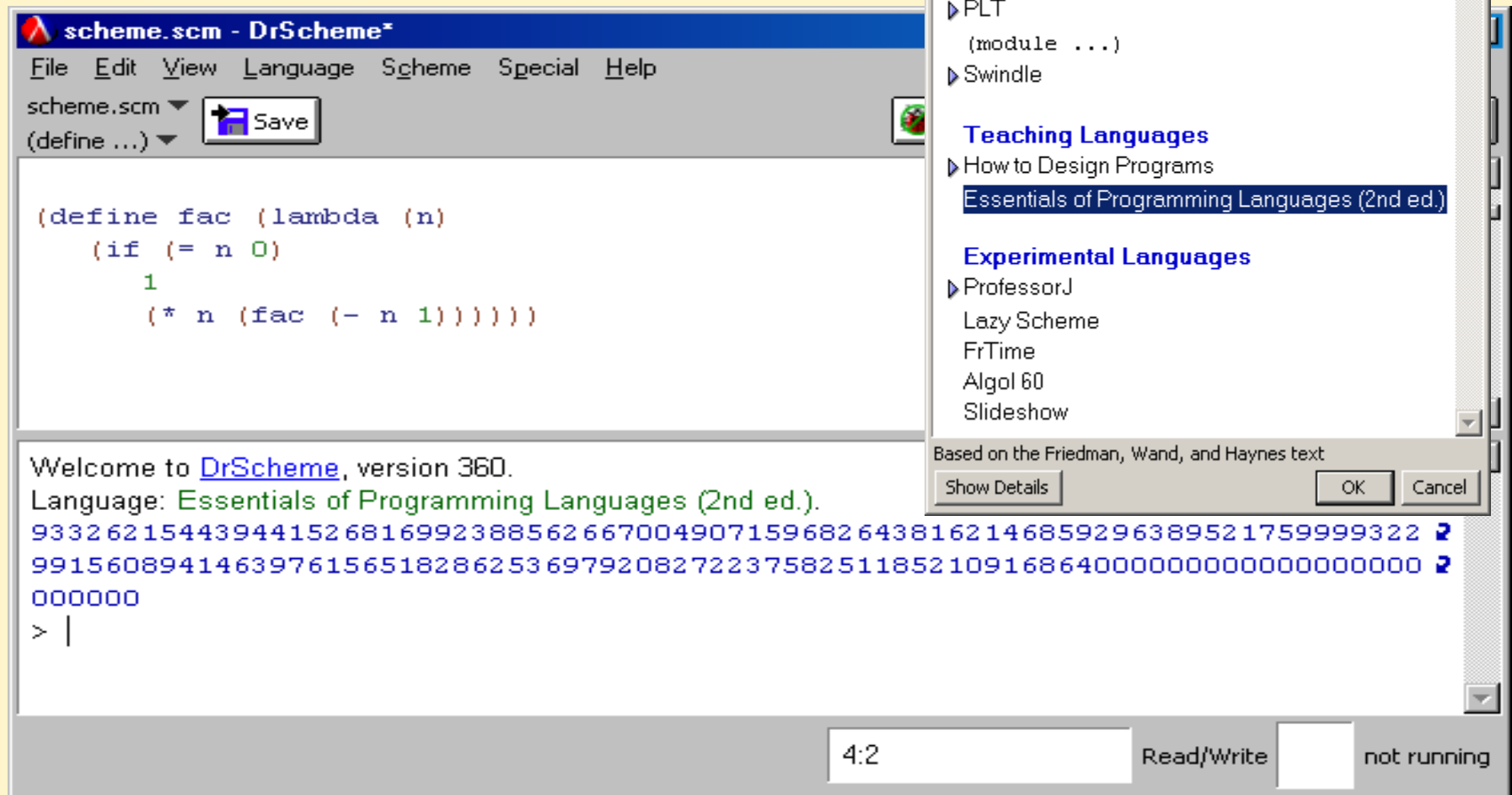
Nepovinné

- Takto vyzerá slajd, ktorý je nepovinný len pre záujemcov

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

- +, -, *
- quotient, remainder, modulo

(`* 3 (+ 5 7) 2`) |--> 72

(`quotient 7 3`) |--> 2

(`remainder -11 3`) |--> -2, (`modulo -11 3`) |--> 1

- max, min, abs
- gcd, lcm
- floor, ceiling

(`max 1 2 3 4 5`) |--> 5

(`gcd 18 12`) |--> 6, (`lcm 18 12`) |--> 36

(`floor (/ 5 3)`) |--> 1, (`ceiling (/ 5 3)`) |--> 2

(`floor -4.3`) |--> -5.0, (`ceiling -4.3`) |--> -4.0

- truncate, round
- expt

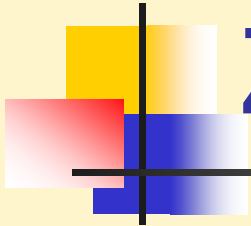
(`truncate -4.3`) |--> -4.0, (`round -4.3`) |--> -4.0

(`expt 2 5`) |--> 32

- eq?, =, <, >, <=, >=
- zero?, positive?, negative?
- odd?, even?

(`odd? 2`) |--> #f, (`even? 2`) |--> #t

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)

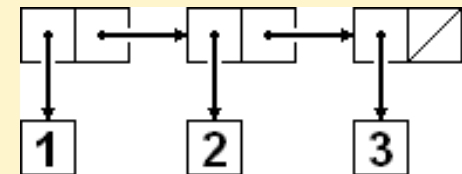
(cdr '((1 2) 3 4)) vráti (3 4)

■ cons

(cons 1 '(2 3)) vráti (1 2 3)

■ quote

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



(cddr '(1 2 3))

(3)

(cdddr '(1 2 3))

()

(cadr '(1 2 3))

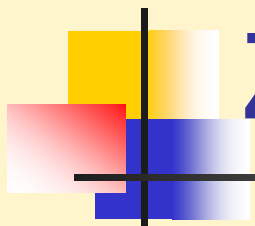
2

(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

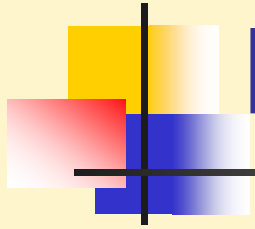
```
(append '(1 2 3) '(a b c))
(1 2 3 a b c)
```

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

otočenie zoznamu

častá chyba

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



Príklad - zásobník

```
(define empty_stack
  ( lambda ( stack ) ( if ( null? stack ) #t #f )))
```

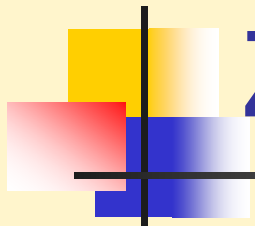
```
(define push
  ( lambda ( element stack ) ( cons element stack ) ))
```

```
(define pop
  ( lambda ( stack ) ( cdr stack )))
```

```
(define top
  ( lambda ( stack ) ( car stack )))
```

```
(define st (push 3 (push 2 (push 1 '()))))
st
```

```
(top (pop (pop st)))
1
```



Zoznamová rekurzia 2

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

list? je #t ak argument je

zoznam

■ 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))
()

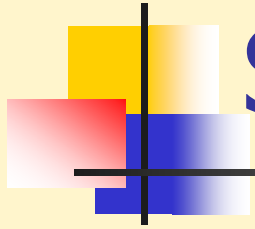
lebo (eq? '(1 2) '(1 2)) je #f

■ 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



Spošenie 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

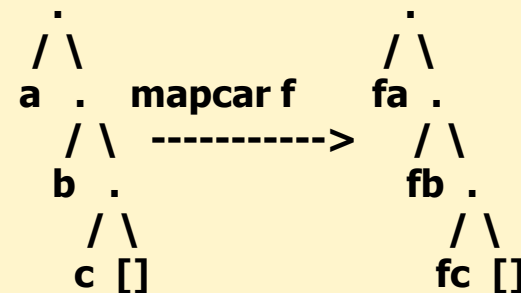
- `mapcar` ; aplikuj funkciu na každý prvok zoznamu

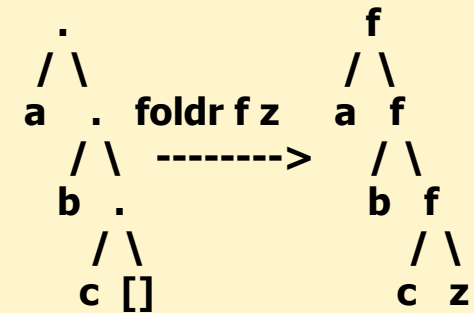
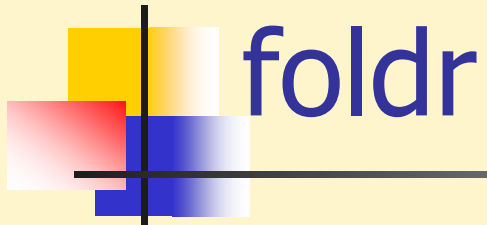
```
(define (mapcar fun lis)
  (cond
    ((null? lis) '())
    (else (cons (fun (car lis))
                  (mapcar fun (cdr lis))))))
```

```
(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

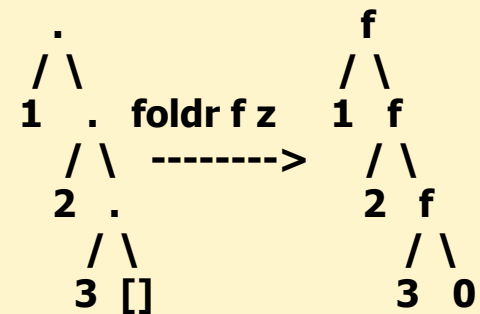




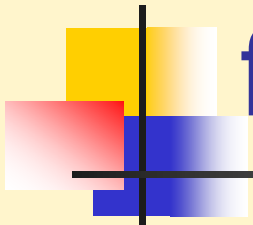
```
(define foldr
  (lambda (func zero lis)
    (if (null? lis)
        zero
        (func (car lis) (foldr func zero (cdr lis))))))
```

```
(foldr (lambda (x y) (+ (* 10 y) x)) 0 '(1 2 3))
321
```

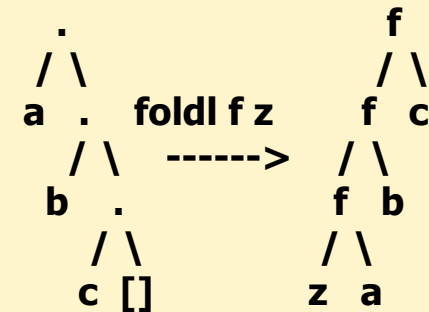
```
(foldr (lambda (x y) (+ (* 10 x) y)) 0 '(1 2 3))
60
```



porozmýšľajte, ako definovať append pomocou foldr



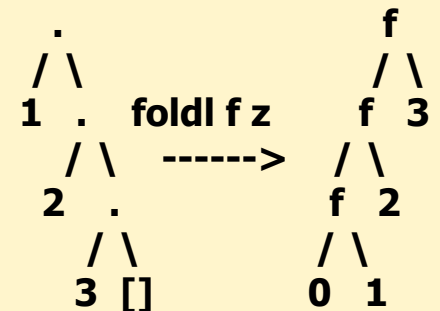
foldl



```
(define foldl
  (lambda (func accum lis)
    (if (null? lis)
        accum
        (foldl func (func accum (car lis)) (cdr lis))))))
```

```
(foldl (lambda (x y) (+ (* 10 x) y)) 0 '(1 2 3))
123
```

```
(foldl (lambda (x y) (+ (* 10 y) x)) 0 '(1 2 3))
100
```



porozmýšľajte, ako definovať reverse pomocou foldl



spoj a rozpoj

- $(\text{spoj } '(1\ 2\ 3) \ '(4\ 5\ 6)) = ((1\ 4)\ (2\ 5)\ (3\ 6))$
- $(\text{rozpoj } '((1\ 4)\ (2\ 5)\ (3\ 6))) = ((1\ 2\ 3)\ (4\ 5\ 6))$

```
(define rozpoj (lambda (pairs)
```

```
  (cond
```

```
    ((null? pairs) '())
```

; prázdny zoznam

```
    ((null? (cdr pairs))
```

; jedno-prvkový zoznam

```
      (list (list (caar pairs)) (list (cadar pairs))))
```

```
    (else
```

; dvoj-a-viac prvkový

```
(rozpoj '())      (list
```

```
  ()      (cons (caar pairs) (car (rozpoj (cdr pairs))))
```

```
(rozpoj '((1 4))) (cons (cadar pairs) (cadr (rozpoj (cdr pairs))))))
```

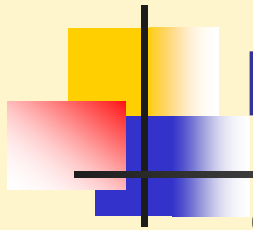
```
((1) (4))
```

```
(rozpoj '((1 4) (2 3)))
```

```
((1 2) (4 3))
```

```
(rozpoj '((1 11) (2 12) (3 13) (4 14)))
```

```
((1 2 3 4) (11 12 13 14))
```

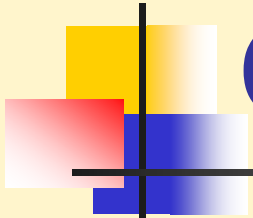


rozpoj pomocou let

```
(define rozpoy (lambda (pairs)
  (cond
    ((null? pairs) '())
    ((null? (cdr pairs)) (list (list (caar pairs)) (list (cadar pairs))))
    (else
     (let ((pom (rozpoy (cdr pairs))))
       (list
        (cons (caar pairs) (car pom))
        (cons (cadar pairs) (cadr pom))))))))
```

```
(let (
  (var1 expr1) ...
  (varn exprn)
  expr)
```

priradí do premenných var_i hodnoty výrazov expr_i a vyhodnotí výraz expr



Queens

doposiaľ dobre položené dámy

```
(define btrackRow (lambda (col row queens)  
  (if (> row N)  
      #f  
      (or  
        (and (safe row row row queens) (btrack (+ col 1) (cons row queens)))  
        (btrackRow col (+ row 1) queens))  
      )  
  )  
)
```

; skús položiť dámu do riadku row v stĺpci col

```
(define btrack (lambda (col queens)  
  (if (> col N)  
      queens  
      (btrackRow col 1 queens)  
  )  
)
```

; skús položiť dámu do stĺpca col

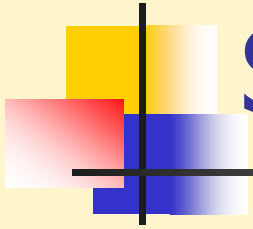
```
> (btrack 1 ())  
(4 2 7 3 6 8 5 1)
```



(define **N** 8)

```
(define safe (lambda (row diag1 diag2 queens)
  (if (null? queens)
      #t
      (if (or (eq? row (car queens)) ; kolizia v riadku
              (eq? (+ diag1 1) (car queens)) ; kolizia na 1.uhlopriecke
              (eq? (- diag2 1) (car queens))) ; kolizia na 2.uhlopriecke
          #f
          (safe row (+ diag1 1) (- diag2 1) (cdr queens))))
  )
)

(define safe (lambda (row diag1 diag2 queens)
  (let ((diag11 (+ diag1 1)) (diag21 (- diag2 1)))
    (or (null? queens)
        (and (not (or (eq? row (car queens)) ; kolizia v riadku
                      (eq? diag11 (car queens)) ; kolizia na 1.uhlopriecke
                      (eq? diag21 (car queens)))) ; kolizia na 2.uhlopriecke
            (safe row diag11 diag21 (cdr queens))))))
  ))
```



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 *<test>* *<then>*) (if (even? n) (quotient n 2))
(if *<test>* *<then>* *<else>*) (if (even? n) (quotient n 2) (+ 1 (*3 x)))
(cond (*<test1>* *<expr1>*) (cond ((= n 1) 1)
(*<test2>* *<expr2>*) ((even? n) (quotient n 2))
(else *<exprn>*)) (else (+ 1 (*3 x))))
- let
(let ((*<var₁>* *<expr₁>*) ...
(*<var_n>* *<expr_n>*))
 <expr>) (let ((x (+ n 1))) (* x x))

Venujem neznámemu Viktorovi :
napriek menu menu svojmu,
padol za predstavu svoju,
o spojitých funkciach...

Kvíz funkcionára

Každý slušný funkcionár hľadá (vo voľbách) nejakú dobrú funkciu

Tréning: viete nájsť funkciu f a zapísať je v matematike/Haskelli, ktorá

1. $f^3 = \text{iteruj } 3 \text{ } f = \lambda x \rightarrow x+6,$

$$f = \lambda x \rightarrow x+2$$

1. $f^3 = \text{iteruj } 3 \text{ } f = \lambda x \rightarrow 27 * x$

$$f = \lambda x \rightarrow 3 * x$$

1. $f^3 = \text{iteruj } 3 \text{ } f = \lambda x \rightarrow x+13$

$$f = \lambda x \rightarrow x+13/3$$

1. $f^3 = \text{iteruj } 3 \text{ } f = \lambda x \rightarrow 11 * x$

$$f = \lambda x \rightarrow \sqrt[3]{11} * x$$

1. $f^3 = \text{iteruj } 3 \text{ } f = \lambda x \rightarrow x^8$

$$f = \lambda x \rightarrow x * x$$

1. $f^3 = \text{iteruj } 3 \text{ } f = \lambda x \rightarrow x^7$

$$f = ???$$

Neexistuje ? Alebo len nemá meno (v matematike, či Haskellu) ?

