Funkcionálne programovanie

1mAINp
1mINFb
2mINFb

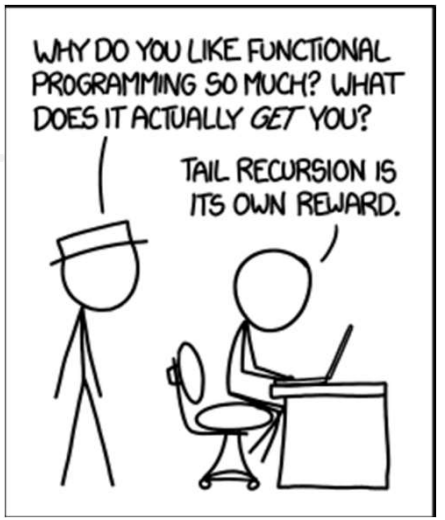
Peter Borovanský

): online :(

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

Prečo funkcionálne programovať ?

(pohľad funkcionálneho programátora)
(populistický pohľad)



- 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 ☺

http://www.forbes.com/sites/quora/2014/08/08/the-best-languages-for-getting-into-functional-programming/

Prečo funkcionálne programovať ?

(pohľad nefunkcionálneho (OOP) programátora)

funkcionálne programovanie

- je súčasťou moderných (aktuálne)... vznikajúcich jazykov,
  - Python, Go, Clojure, Scala, Swift, Haskell, Kotlin, TypeScript, *PureScript*, …
- a vkráda sa do jazykov klasicky procedurálnych/imperatívnych jazykov
  - Java (Java 8), C++ (C++ v.11), *Excel*, *Fortran*… - ale nie vždy máte chuť na takú syntax

- That's awful !!!

Lambda expressions in Fortran

1 Introduction

Many modern programming languages have a feature called *lambda expressions* that allows the programmer to pass simple expressions as *arguments* to some subprogram instead of having to write a subprogram that contains the expression. Here is an example in Java:

```
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

```
function integer_add( x, y ) result(add)
    type(lambda_integer), intent(in), target  :: x
    type(lambda_integer), intent(in), target  :: y
    type(lambda_integer), pointer             :: add

    allocate( add )

    add%operation =  1
    add%first     => x
    add%second    => y
end function integer_add
```

Arjen Markus
arjen.markus@deltares.nl

May 30, 2019

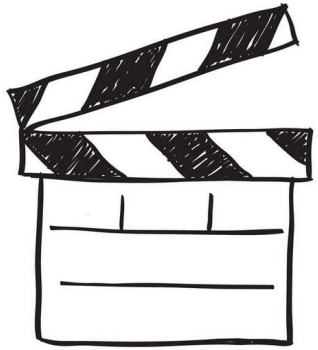
OOP vs. FP

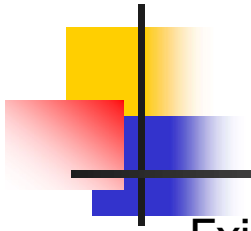
(:- hlavný konkurent OOP je v kríze (čo je na diskusiu :-)

- **Object-Oriented Programming is Bad** https://www.youtube.com/watch?v=QM1iUe6IofM
- provokatívne až konšpiračné 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-paradigmové monštrá,
- v snahe mať v jazyku všetko, strácame jednoduchosť a eleganciu (Java),
- Java vznikla na smetisku jazykov (C/C++, Pascal, VB)
- na začiatky bola jednoduchá a elegantná,
- snaha očistiť (vytiahnúť esenciu) programovania, napr. Haskell,Go,Scala, Kotlin

https://www.youtube.com/watch?v=9ei-rbULWoA&t=44m01s

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





FP vs. OOP

Existuje množstvo diskusií a názorov na tému FP vs. OOP

Ale princípy, ktoré sa v súčasnosti viac a viac objavujú a presadzujú sú:

- **referenčná transparentosť** – čistota – *purity* - funkcia vždy pre rovnaké vstupy vráti rovnaký výsledok
  - apriori to teda zakazuje globálne premenné ako zdroj side-effectu,
  - riešenia: soft (na zodpovednosti programátora), hard (reštrikcia jazyka bez globálnych p.)
  - state-less – čo je presný opak objektov, OOP, ktoré si stav pamätajú
- **nemennosť** (**immutability**) dátových štruktúr – pri pokuse modifikovať dáta musíte vyvoriť ich nezávislú kópiu (príklad List, MutableList)
  - nemusí to byť drahé, závisí od implementácie
  - prvý garbage collector bol práve vo funkcionálnom jazyku LISP
- **inkluzívny** vs. **parametrický polymorfizmus**
  - dedenie a „generics" nájdete súčasne v jazykoch Java, Scala, Kotlin, ...
  - kovariancia a anti-kovariancia
- **rekurzia** vs. **cyklus**
  - bez Tail Recursion Optimisation by to nebolo ono...
  - a TRO je čoraz častejšia výbava programovacích jazykov (Kotlin, JS, …)

Rekuzia vs. cyklus

(čo počíta funkcia)

goo
1. a &b
2. a|b
3. a^b
4. a+b
5. a*b
6. $a*2^b$
7. bit-count $\#_1$ a
8. bit-reverse a
9. bit-concat a,b

```
JAVA:
public static int goo(int a, int b) {
      int sum = 0;
      while (a > 0) {
         if (a % 2 > 0) sum += b;
         a /= 2;
         b *= 2;
      }
      return sum;
}
Haskell:
goo 0 b = 0
goo a b | a `mod` 2 == 0 =     goo (a `div` 2) (2*b)
        | otherwise      = b + goo (a `div` 2) (2*b)
```

$$\text{goo } a\ b = \begin{cases} 0, & \text{ak } a = 0 \\ \text{goo } (a/2)\ (2*b), & \text{ak } 2|a \\ b + \text{goo } (a/2)\ (2*b), & \text{ak a je nepárne} \end{cases}$$

$$\text{goo } a\ b = \begin{cases} 0, & \text{ak } a = 0 \\ \text{goo } (a/2)\ (2*b), & \text{ak } 2 \mid a \\ b + \text{goo } (a/2)\ (2*b), & \text{ak } a \text{ je nepárne} \end{cases}$$

Ruské násobenie

- 17 x 21
- ~~8 x 42~~
- ~~4 x 84~~
- ~~2 x 168~~
- 1 x 336
- ---------
- 357

1
0
0
1 =

$17_{(10)}$

Ruské násobenie

Rekurzia vs. indukcia

$$\text{goo } a\ b = \begin{cases} 0, & \text{ak } a = 0 \\ \text{goo } (a/2)\ (2*b), & \text{ak } 2|a \\ b + \text{goo } (a/2)\ (2*b), & \text{ak } a \text{ je nepárne} \end{cases}$$

Tvrdenie: goo a b = a*b

Indukcia podľa a

- ak a == 0, platí.
- ak a > 0, (induckia nám dovolí spoliehať sa, že to platí pre menšie a):
  - a je párne, tak goo a b = goo (a/2) (2*b) = podľa IP = (a/2) * (2*b) = a*b, platí.
  - a je nepárne, tak si uvedomme, že
    (a/2)*2 = a-1

    goo a b = b+goo (a/2) (2*b) = podľa IP = b + (a/2) * (2*b) = b+(a-1)*b, platí.

Typologia Haskell programátora

$$\text{goo } a\ b = \begin{cases} 0, & \text{ak } a = 0 \\ \text{goo } (a/2)\ (2*b), & \text{ak } 2 \mid a \\ b + \text{goo } (a/2)\ (2*b), & \text{ak } a \text{ je nepárne} \end{cases}$$

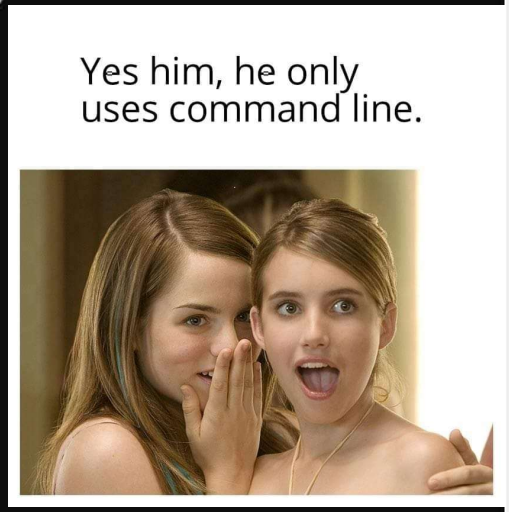
```
module Goo where
import Data.Bits

-- slušák, čo pobral niečo z prednášky
goo :: Int -> Int -> Int
goo 0 b = 0
goo a b | a `mod` 2 == 0 =     goo (a `div` 2) (2*b)
        | otherwise      = b + goo (a `div` 2) (2*b)

-- ľavičiarsky akumulátorčík
goo' :: Int -> Int -> Int
goo' a b = foldl (\acc -> \(a,b) -> acc + if a `mod` 2 > 0 then b else 0) 0 $
             takeWhile ((>0).fst) $
               iterate (\(a,b) -> (a `div` 2, 2*b)) (a,b)

-- bojazlivejší čo pozná map filter z Pythonu
goo'' :: Int -> Int -> Int
goo'' a b = sum $
              map snd $
                filter ((>0).(`mod` 2).fst) $
                  takeWhile ((>0).fst) $
                    iterate (\(a,b) -> (a `div` 2, 2*b)) (a,b)

-- chce provokovať prednášajúceho
goo''' :: Int -> Int -> Int
goo''' a b = sum $
               map snd $
                 filter ((>0).(.&. 1).fst) $
                   takeWhile ((>0).fst) $
                     iterate (\(a,b) -> (shiftR a 1, shiftL b 1)) (a,b)
```

goo.hs

Realita Haskell programátora

$$\text{goo } a\ b = \begin{cases} 0, & \text{ak } a = 0 \\ \text{goo } (a/2)\ (2*b), & \text{ak } 2|a \\ b + \text{goo } (a/2)\ (2*b), & \text{ak } a \text{ je nepárne} \end{cases}$$

```
C:\Program Files\Haskell Platform\8.6.5\bin\ghci.exe
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :l goo.hs
[1 of 1] Compiling Goo              ( goo.hs, interpreted )
Ok, one module loaded.
*Goo> goo 17 21
357
*Goo> goo' 17 21
357
*Goo> goo'' 17 21
357
*Goo> goo''' 17 21
357
*Goo> q.e.d.

<interactive>:6:7: error:
    parse error (possibly incorrect indentation or mismatched brackets)
*Goo> _
```

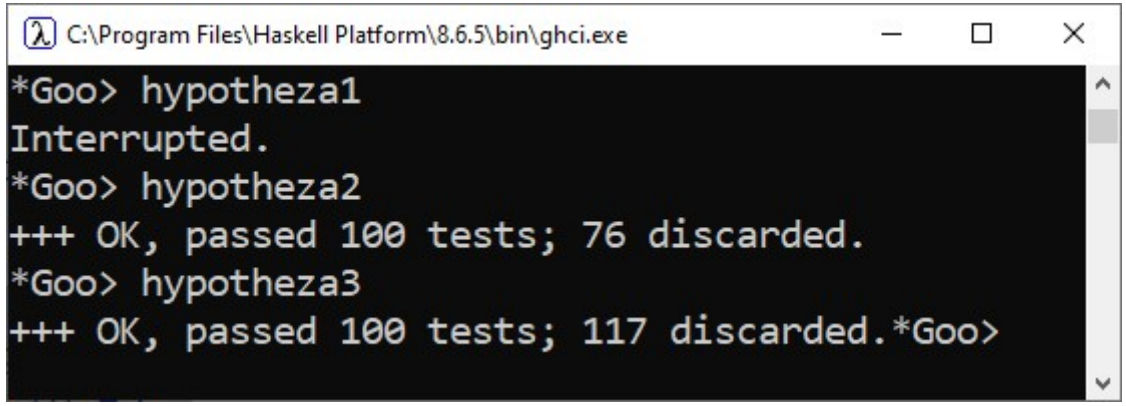


goo.hs

Realita Haskell programátora

$$\text{goo } a\ b = \begin{cases} 0, & \text{ak } a = 0 \\ \text{goo } (a/2)\ (2*b), & \text{ak } 2|a \\ b + \text{goo } (a/2)\ (2*b), & \text{ak } a \text{ je nepárne} \end{cases}$$

```
hypotheza1   = quickCheck(\a -> \b ->
                    goo a b == goo' a b)
hypotheza2   = quickCheck(\a -> \b ->
                    a >= 0 ==> goo a b == goo' a b)
hypotheza3   = quickCheck(\a -> \b ->
                    a >= 0 ==>
      length (nub [goo a b, goo' a b, goo'' a b, goo''' a b]) == 1)
```



goo.hs

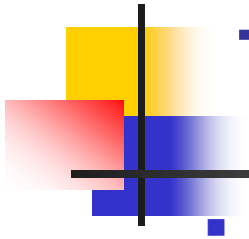
Svet kde neplatí už ani prvé pravidlo

Don't touch it ..

Transformácia/refaktoring funkcionálneho programu

- je hra
- chráni vás type-checker, quick-checker, ale najmä
- referenčná transparentnosť = neexistencia side-effects, alias globálnych premenných
- o FP sa ľahšie uvažuje...

```
JAVA:
public static int goo(int a, int b) {
      int sum = 0;
      while (a > 0) {
         if (a % 2 > 0) sum += b;
         a /= 2;
         b *= 2;
      }
      return sum;
}
```



Trochu histórie

- 1900 David Hilbert – 10./24 Problém:

  Nájsť algoritmus, ktorý zistí, či Diofantická rovnica má celočíselné riešenie

- 1910 Bertrand Russel – Principia Mathematica

  na strane 379 dokázal, že 1+1=2

- 1931 Kurt Gödel: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I

- 1936 Alan Turing: On Computable Numbers with an Application to the Entscheidungsproblem

1940
Turing's Bombe

- 1936 Alonso Church: An Unsolvable Problem of Elementary Number Theory

- 1951 Rózsa Péter: Recursive Functions

- 1958 Curry Haskell: Combinatory Logic

- 1959 John Mc Carthy: Recursive Functions of Symbolic Expressions and Their Computation by Machine – jazyk LISP

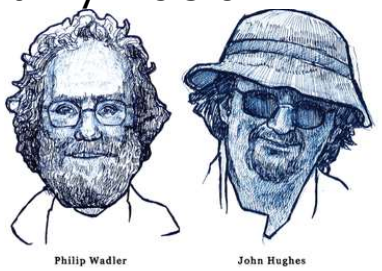
2013
Turing's Pardon

- 1995: ACM Haskell Symposium https://www.futurelearn.com/courses/functional-programming-haskell/0/steps/27218

Jazyky okolo r.1995

- v 80-tkách existovalo množstvo Haskellu podobných ale nie čistých funkcionálnyuch jazykov, ML, Miranda, Gofer, Scheme, ...
- Haskell bol navrhnutý komisiou múdrych hláv 1.apríla ☺ 1990 s hlavným cieľom:

  vhodný na výuku, výskum, tvorbu veľkých aplikácií
- veľmi zaujímavý článok **A history of Haskell**, 2007 s časovým odstupom a nadhľadom popisuje vznik konceptu, ale aj históriu FP
  https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/history.pdf



https://www.futurelearn.com/info/courses/functional-programming-haskell/0/steps/27218

Prečo funkcionálne programovať ?

(pohľad front-end programátora)
(front-end v posledných rokoch reinkarnoval 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 JavaStript 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

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

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


```
> typeof NaN
"number"
> 9999999999999999
10000000000000000
> 0.5+0.1==0.6
true
> 0.1+0.2==0.3
false
> Math.max()
-Infinity
> Math.min()
Infinity
> []+[]
""
> []+{}
"[object Object]"
> {}+[]
0
> true+true+true===3
true
> true-true
0
```

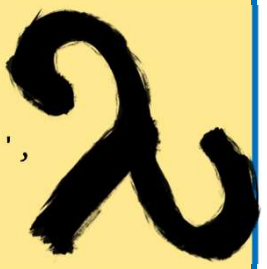
```
> true==1
true
> true===1
false
> (!+[]+[]+![]).length
9
> 9+"1"
"91"
> 91-"1"
90
> []==0
true
```

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) => {..}
```

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

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, umožnuje 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);
            })
    });
}
```

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

Async/Await

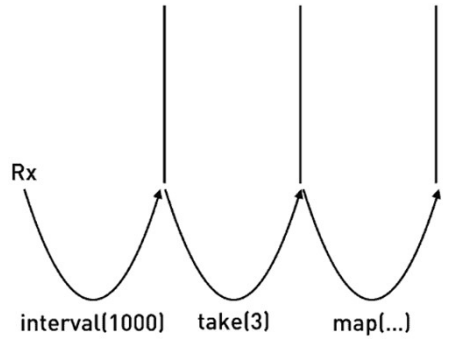
(ES7 syntax extension)

```
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

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

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

https://jsbin.com/dosaviwalu/edit?js,console

https://codecraft.tv/courses/angular/reactive-programming-with-rxjs/observables-and-rxjs/

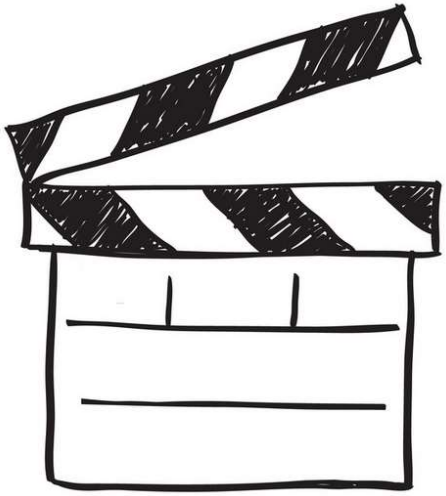
Odporúčané čítanie

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/



Čo sa deje dnes ? 2021

DAY 3

FEBRUARY 18, 2021

LAMBDA DAYS TFP

12:00 - 13:00 | Keynote: Excel meets Lambda
Simon Peyton Jones
Andy Gordon

DAY 4

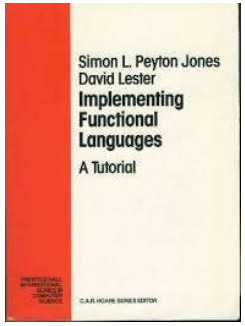
FEBRUARY 19, 2021

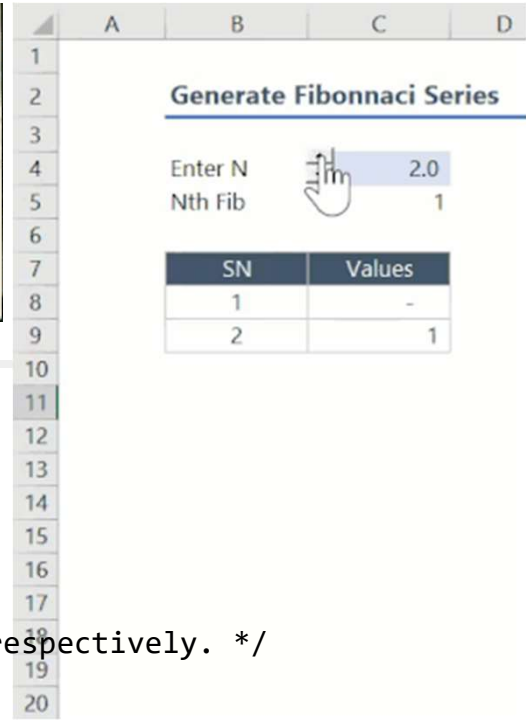
LAMBDA DAYS

12:00 - 13:00 | Keynote: (Programming Languages) in Agda = Programming (Languages in Agda)
Philip Wadler

Excel & Lambdas

Simon Peyton Jones – Microsoft Reseach

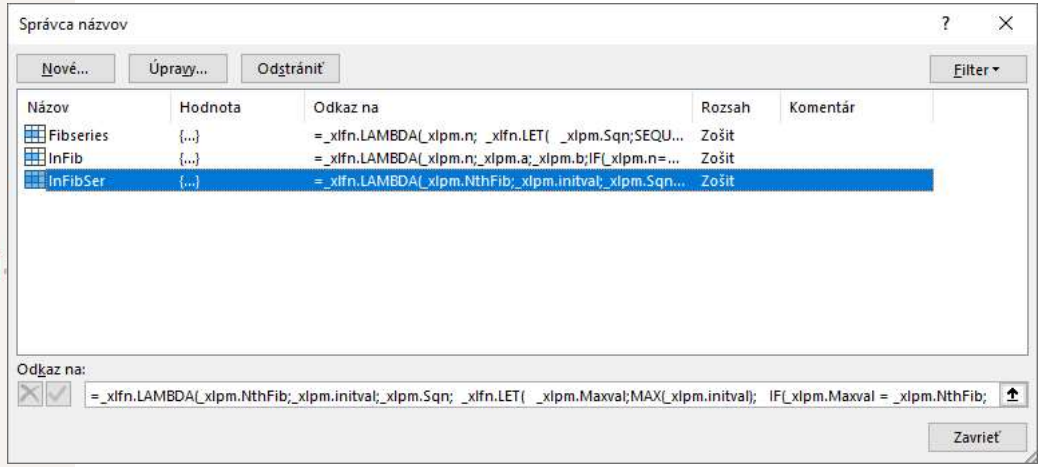





```
Infib=LAMBDA(n,a,b,
        IF(n=1,a,
           IF(n=2,b,Infib(n-1,b,a+b))))
```

```
/* The value for the arguments a and b should be passed as 0 and 1, respectively. */
```

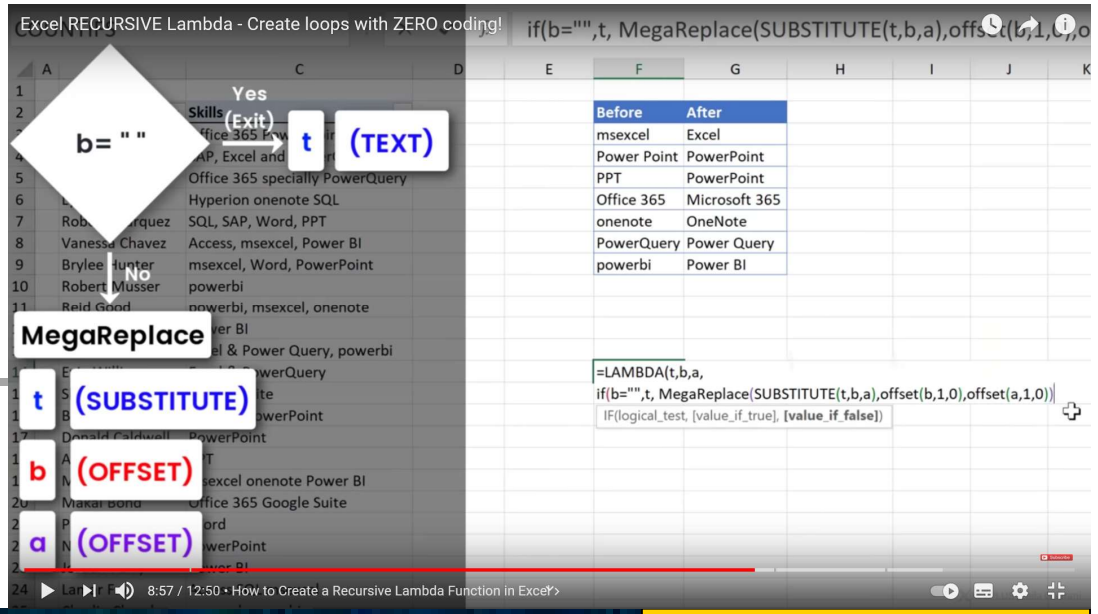
```
=Lambda(NthFib, Initval, Sqn,
    Let(
        Maxval,Max(Initval),
        If(Maxval = NthFib,
        Initval,
        Let(
            NewVal,If(Maxval=1,Initval+1*(Sqn>=4),
                      Let(
                      StPos,Match(Maxval,Initval,0)+1,
                      adval,Large(unique(Initval),2),
                      Initval+adval*(Sqn>=StPos)
                          )
                    ),
            InFibser(NthFib,NewVal,Sqn)
        )
    )
)
 )
```



https://techcommunity.microsoft.com/t5/excel/lambda-example-generate-fibonacci-series/m-p/2034565

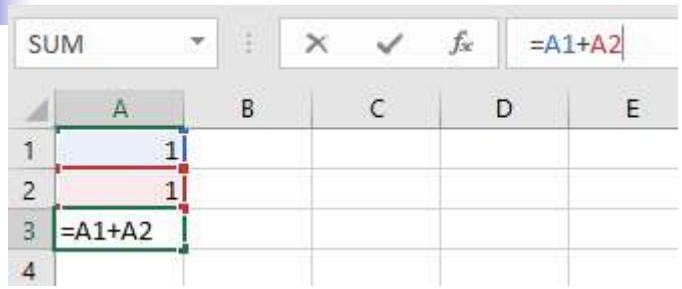
Fibonnaci series.xlsx

Recursive Lambdas



Kde soudruzi z Redmontu udelali chybu ?

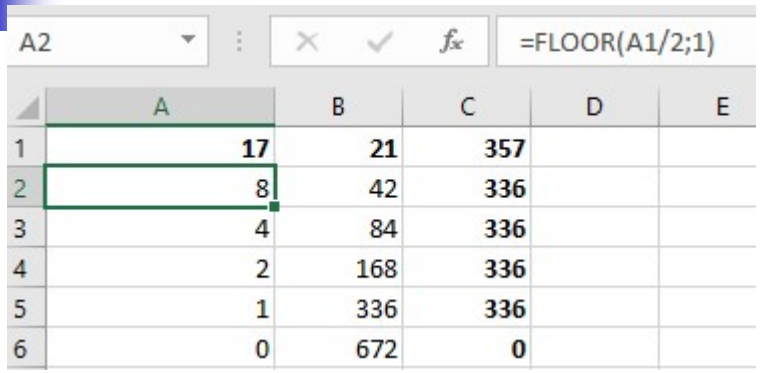
Excel programming



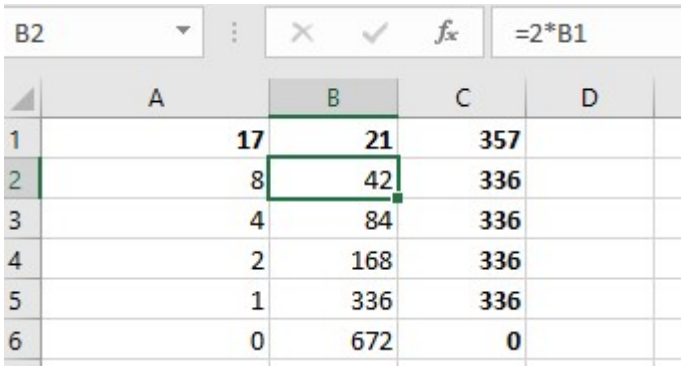
	A
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55
11	89
12	144
13	233
14	377
15	610
16	987
17	1597

ExcelProgramming_Fib_Goo.xlsx

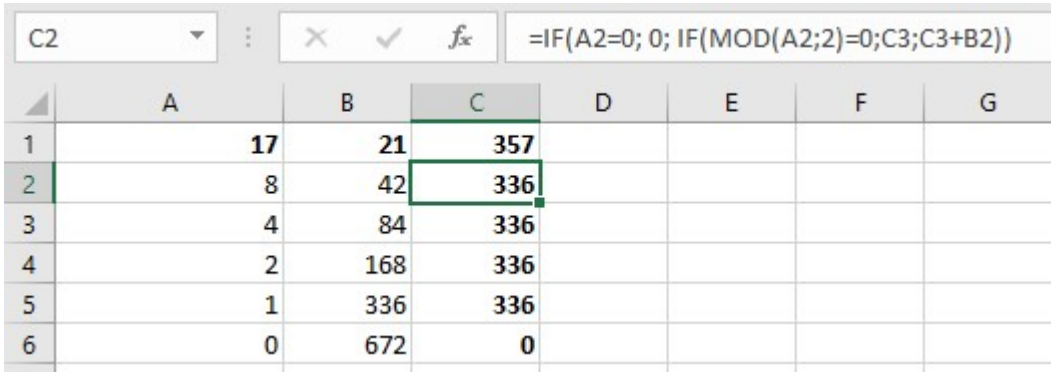
Excel programming



	A	B	C	D	E
1	**17**	**21**	**357**		
2	8	42	**336**		
3	4	84	**336**		
4	2	168	**336**		
5	1	336	**336**		
6	0	672	**0**		



	A	B	C	D
1	**17**	**21**	**357**	
2	8	42	**336**	
3	4	84	**336**	
4	2	168	**336**	
5	1	336	**336**	
6	0	672	**0**	

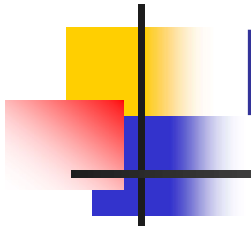


	A	B	C	D	E	F	G
1	**17**	**21**	**357**				
2	8	42	**336**				
3	4	84	**336**				
4	2	168	**336**				
5	1	336	**336**				
6	0	672	**0**				

$$\text{goo } a\ b = \begin{cases} 0, & \text{ak } a = 0 \\ \text{goo } (a/2)\ (2*b), & \text{ak } 2 \mid a \\ b + \text{goo } (a/2)\ (2*b), & \text{ak } a \text{ je nepárne} \end{cases}$$

ExcelProgramming_Fib_Goo.xlsx

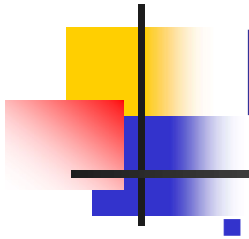
Software is getting slower more rapidly than hardware becomes faster. -- Nicolaus Wirth (Pascal)



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), odporučenia, 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->N), R (R->R), ...
- Funkcií je N->N veľa. Viac ako |N| ?
- Funkcií N->N je toľko ako reálnych čísel...Vieme všetky naprogramovať ?
- Funkcií je R->R je ešte viac. Naozaj ?

- Funkcie skladáme. To je asociatívna operácia... Je komutatívna ?
- x -> 4 je konštantná funkcia napr. typu N->N, či R->R, ...
- {x -> k*x+q} je množstvo (množina) lineárnych funkcií pre rôzne k, q
- Funkcia nie vždy má inverznú funkciu
- Ale skladať ich vieme vždy
  (x -> 2*x+1) . (x -> 3*x+5) je x->2*(3*x+5)+1, teda x->6*x+11
- aj aplikovať v bode z definičného oboru (x->6*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

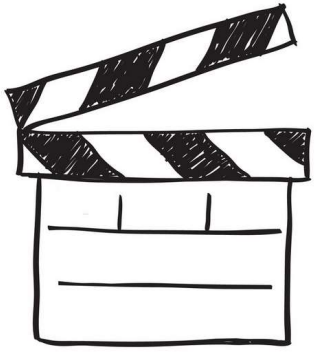
Funkcionálna apokalypsa ☺

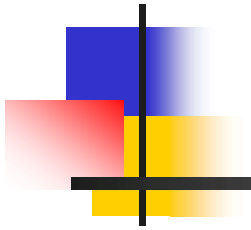
- Predstavme si, že by existovali len funkcie
- a nič iné…
- žiadne čísla, ani prirodzené, ani 0, ani True/False, ani NULL, či nil
- vedeli by sme vybudovať matematiku,

  resp. aspoň aritmetiku ?

  resp. Programovací jazyk ?
- vieme nájsť
  - funkcie, ktoré by zodpovedali číslam 0, 1, 2, ...
  - a operácie zodpovedajúce +, *, …
- tak, aby to fungovalo ako (N, +, *)

- lebo ak áno,
  - podľa Kroneckera sme zachránení
  - podľa Gödela sme stratení
  - v každom formálnom systéme, ktorý obsahuje aspoň **aritmetik prirodzených 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 Haskelli)

```
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)
```

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

Cvičenie na zamyslenie

Reprezentujem dvojicu takto (4 ekvivalentné zápisy v 2 jazykoch):

```
def cons(a, b):
    def pair(f):
        return f(a, b)
    return pair
```

Definujte head, tail, aby platilo

head(cons(a,b)) = a

tail(cons(a,b)) = b

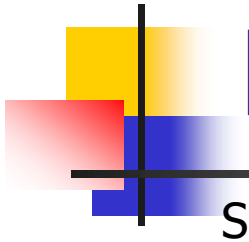
```
def cons(a,b):
    return lambda f: f(a,b)
```

```
dvojica a b = pair
    where pair f = f a b
```

Definujte prvy, druhy, aby platilo

```
prvy (dvojica a b) = a
druhy (dvojica a b) = b
```

```
dvojica a b = \f -> f a b
```



Haskell nemá reálne aplikácie

Spoločnosti používajúce Haskell:

- AT&T, Barclays, Facebook, Google, Microsoft, NYT
- možnosť transparentne uvažovať a transformovať kód (purity)
- striktné a statické typovanie
- konkuretnosť (vďaka ref.transparentnosti)
- efektívnosť kompilovaného kódu

Mýty:

- Haskell je ťažký – učenie, vývoj, kompilovanie/debug
- Haskell Type Checker je ťažký pre začiatočníka
- Haskell je iný – očakávania sú iné, ako pri inom klasickom jazyku
- Haskell je viac abstraktný – knižnice sú plné polymorfických funkcií
- FP is simple, core idead is simple. Abstraction can be hard. It does not mean it's not worth learning – John Huges

https://www.youtube.com/watch?v=mlTO510zO78

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/

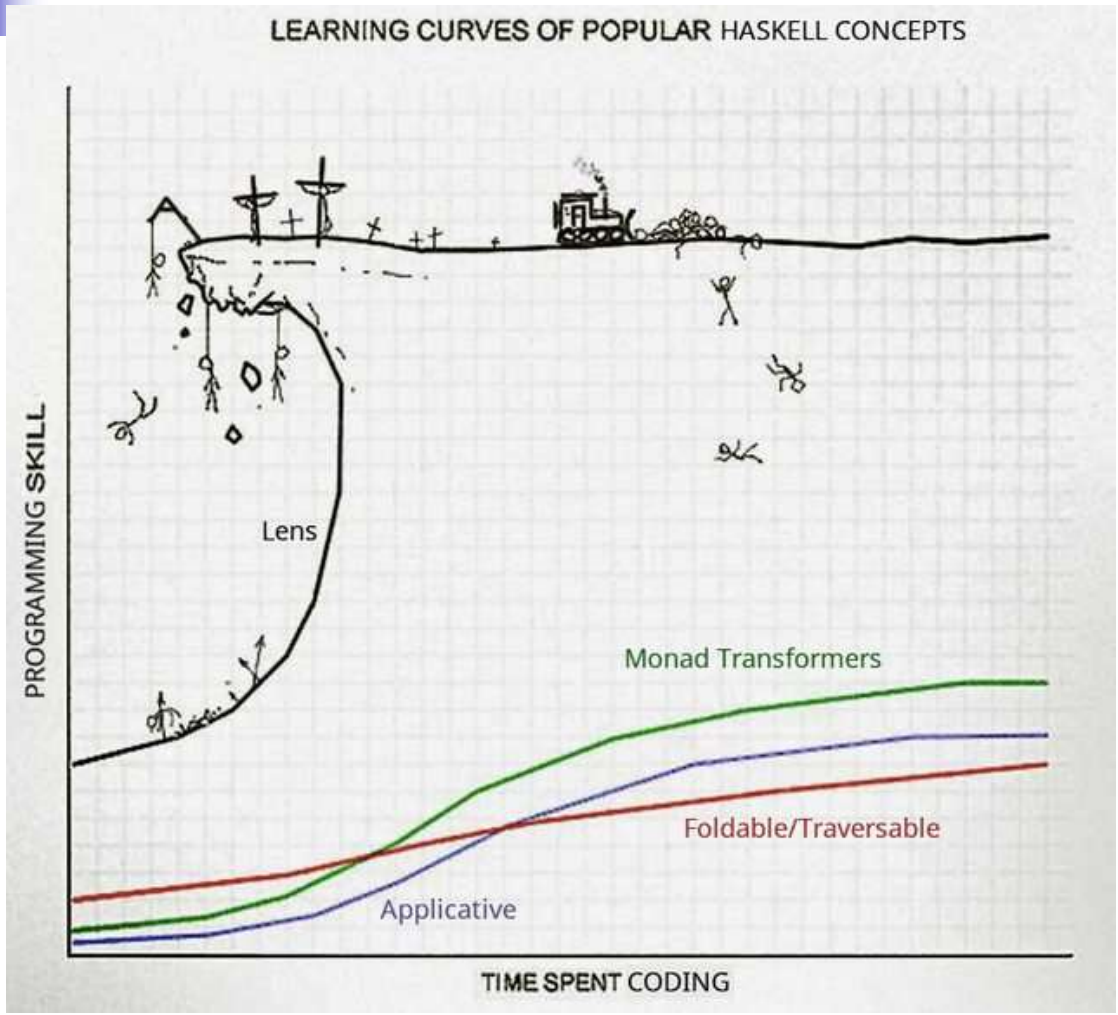
Elixir-Erlang

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

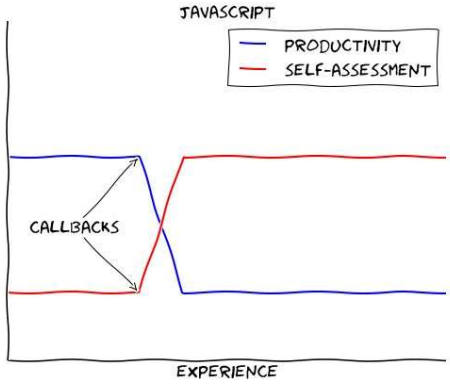
Haskell Learning Curve

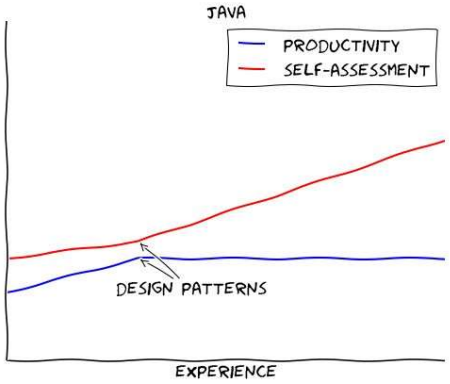


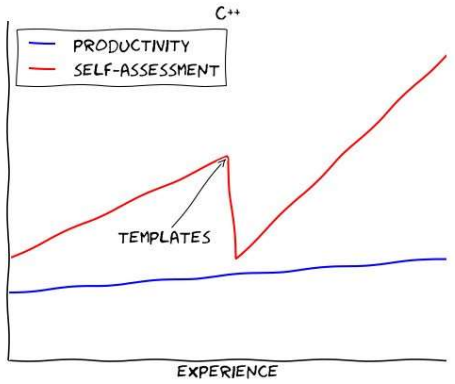
https://twitter.com/hmemcpy/status/889732980163399681

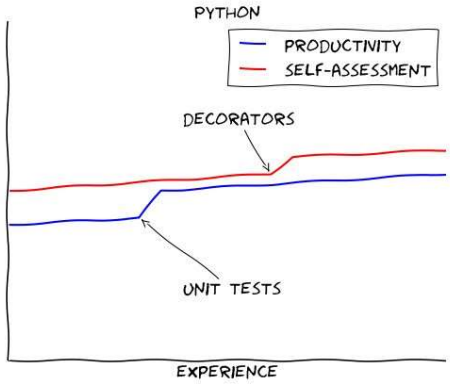
Learning Curves for different programming languages

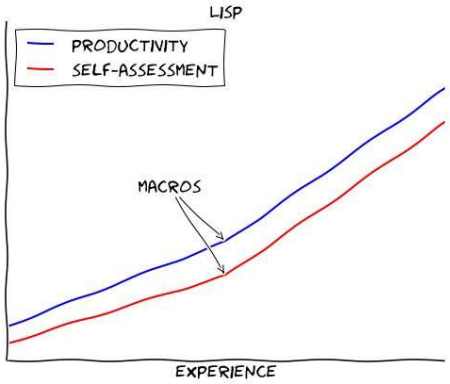
https://github.com/Dobiasd/articles/blob/master/programming_language_learning_curves.md

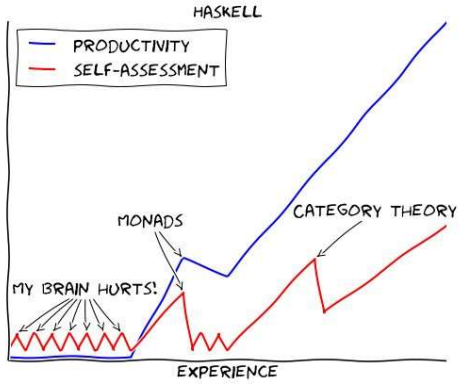


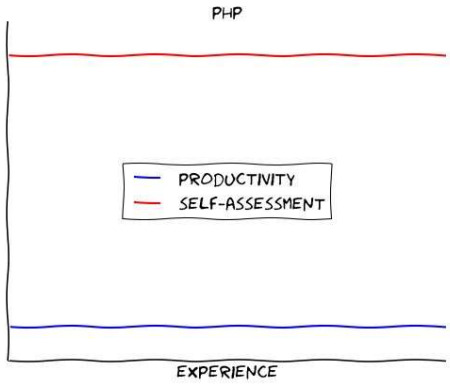






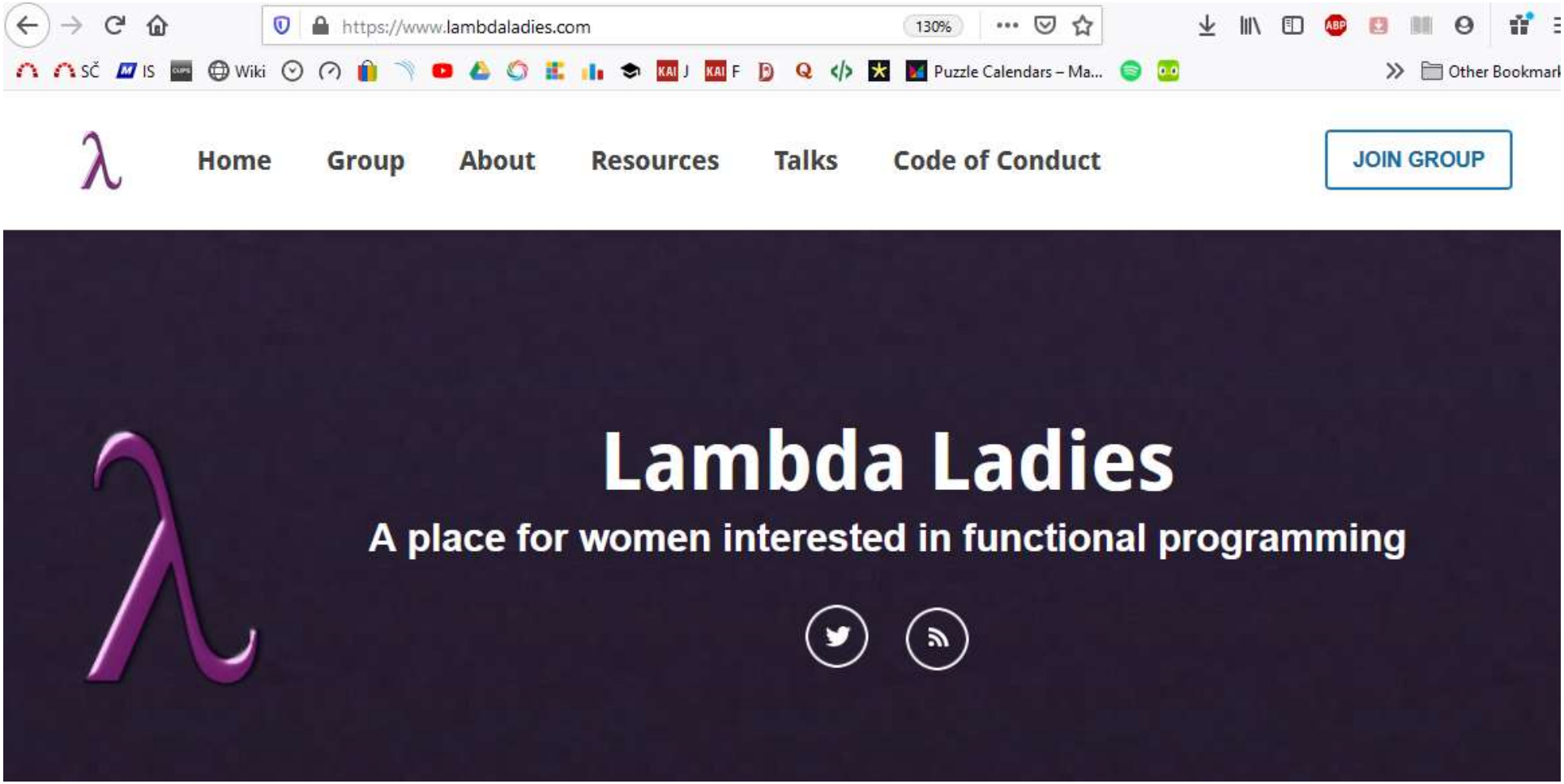




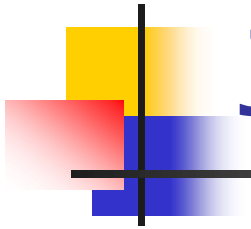


FP je cool

https://www.lambdaladies.com/




Have a break. Have a KitKat
Time for a Break



Jazyky podporujúce FP

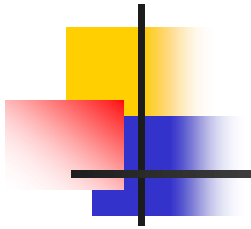
- funkcionálne programovanie je programovanie s funkciami ako hodnotami
- funkcionálne programovanie **nie je** len programovanie funkcií (bolo aj c++)
- funkcionálne programovanie **nie je** jazdenie po zoznamoch (bolo aj pythone)
- funkcionálne programovanie **nie je** v jazyku bez closures

- 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...)

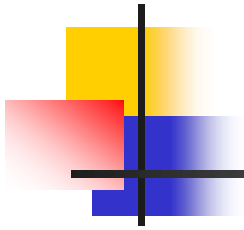
pred FP poznali funkcie ako argumenty, ale *neučili to (na FMFI) na Pascale ani C ...*

```
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.



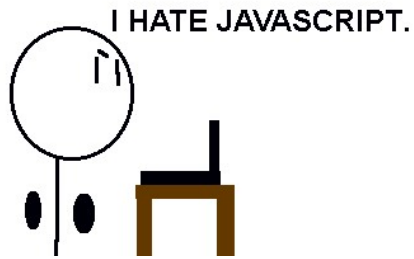
Python Closures

```
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→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 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
```

https://repl.it/

iteruj.py

Javascript Closures




```
function addN(n) {                          -- výsledkom tejto funkcie je funkcia
    return function(x) { return x+n };      -- 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
```

https://repl.it/

iteruj.js

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

https://repl.it/ scoping.js

forEach, map, filter
(Java8 Stream API)

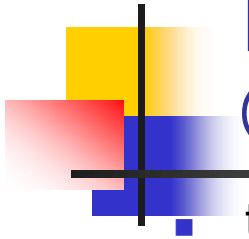
```
List<Karta> vacsieKarty3 = karty
    .stream()
    .map(k->new Karta(k.getHodnota()+1,k.getFarba()))
    .filter(k -> k.getHodnota() > 8)
    .collect(Collectors.toList());
```

[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]

MapFilter.java



Haskell

(typy v kocke)

- typovaný jazyk, teda má typy, napr. Int, Integer, Bool, Char, Float, String,...
- typové konštruktory:
  - n-tica (t_1, …, t_n), zoznam [t], napr. (Int, Int), [Int], [String]
  - funkčné typy t_1->t_2, napr. Int ->Int, Int ->Int ->Int
  - operátor -> je pravo-asociatívny, preto Int ->Int ->Int znamená
    - Int ->(Int ->Int)

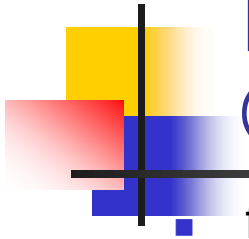
      - funkcia, ktorá pre celé číslo vráti celočíselnú funkciu
    - a nie (Int ->Int) ->Int

      - 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))
```

Iteruj.hs



Haskell

(aplikácia v kocke)

- funkciu f :: A->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 (π ) ) ) ) ) ) )

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
```

Iteruj.hs

Haskell

(definícia v kocke)

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

iteruj n f znamená $f^n = f \circ f \circ \ldots \circ f$, 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
```

Iteruj.hs

Haskell

(pre pokročilejších fajnšmekrov)

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

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

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

iteruj_funkciu              :: Int -> ((t -> t) -> (t -> t) )
iteruj_funkciu n f          = iteruj_f !! n
                              where iteruj_f = id:[f . g | g <- iteruj_f]
```

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

```
cycle :: t -> [t]
cycle 17 = [17,17,17,17,17.17, … ]
cycle f = [f,f,f,f,f,f,f,f,f,f,f,f,f,f,f,f,f,f,f,f,f,
take 5 (cycle f) = [f,f,f,f,f]
```

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

Tu je miesto pre vašu najzvrhlejšiu definíciu iterate

Iteruj.hs

Haskell

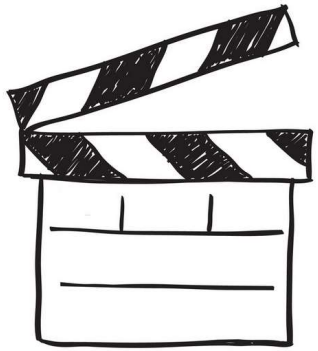
(Hunit testy)

Iteruj.hs

```
module Iteruj where
```

TestIteruj.hs

```
module TestIteruj where
import Iteruj
import Test.Hunit

main = do
    runTestTT $ TestList [
       TestList [
          TestCase $ assertEqual "iteruj 5 (+4) 100" 120 ( iteruj 5 (+4) 100 ),
          TestCase $ assertEqual "iteruj 5 (*4) 100 " 102400 ( iteruj 5 (*4) 100 ),
          TestCase $ assertEqual "iteruj 5 (++ab) c " "cababababab„
                                                    ( iteruj 5 (++"ab") "c" )],
```

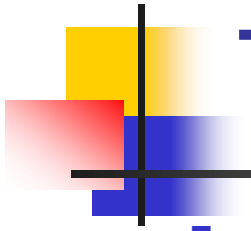


Predchádzalo vzniku FP

- 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 ⊊ 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 rekuzívnej funkcie dá výsledok
- sú rovnako ťažké (nemožné) úlohy

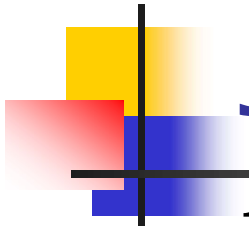
Pôvodne vôbec nešlo o počítanie, ale vypočítateľnosť, či matematiku

- 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**tack-**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



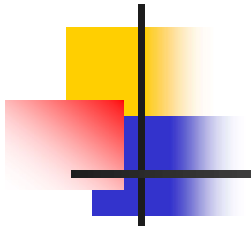
Jazyky FP

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

Can Programming Be Liberated from the von Neumann Style?

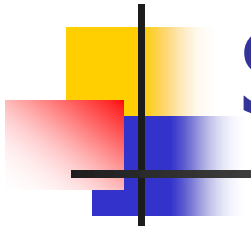
A Functional Style and Its Algebra of Programs

- 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
```

https://repl.it/

scheme.scm

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)))))))
```

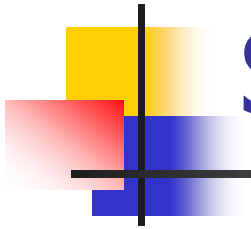
```
(fib 10)
    55
```

```
(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))))
```

https://repl.it/ scheme.scm

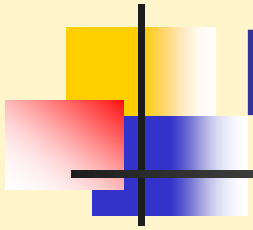


Scheme closures

```
(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)
```

https://repl.it/

scheme.scm



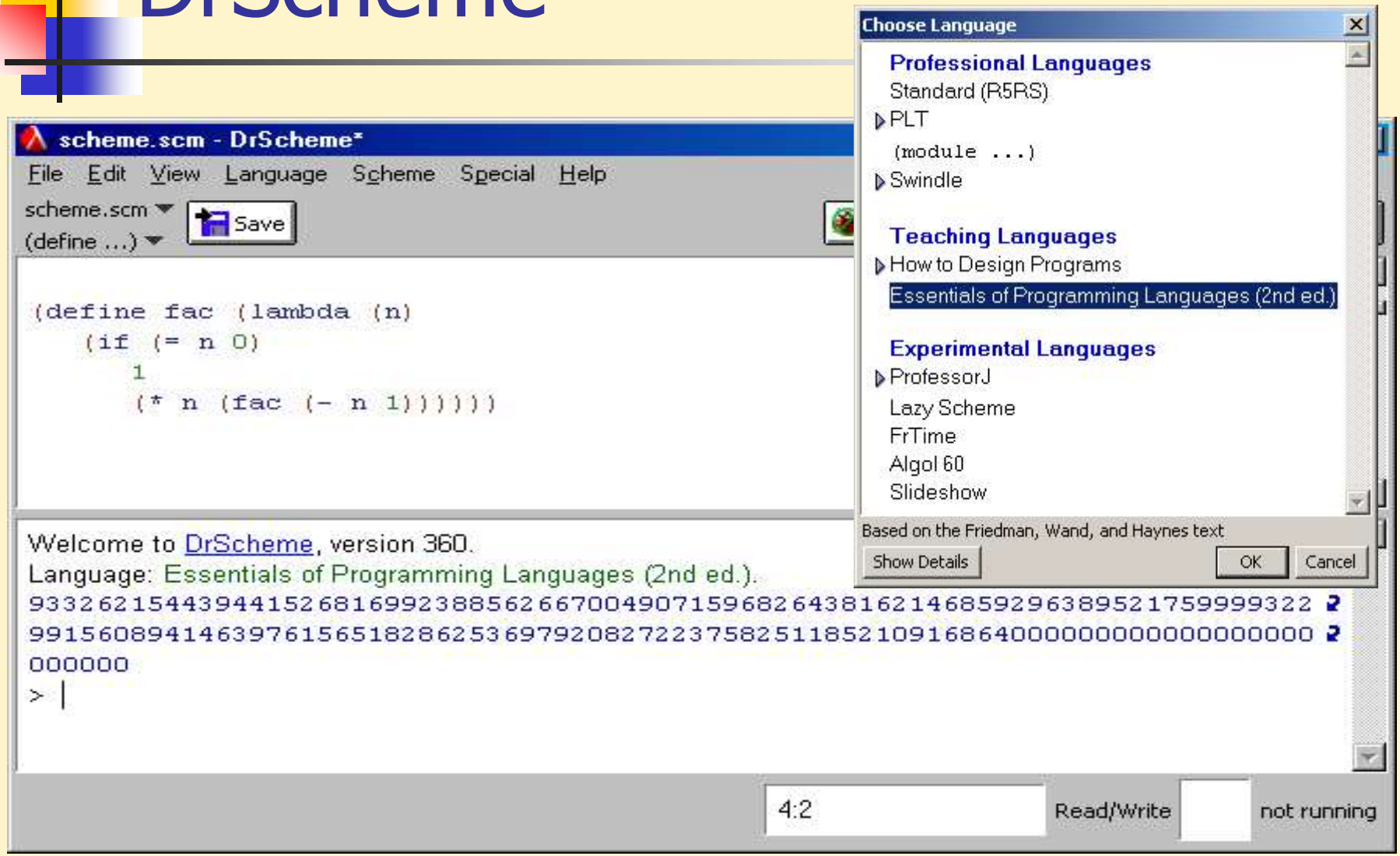
Nepovinné

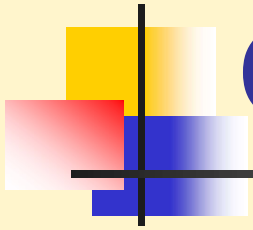
- Takto vyzerá slajd, ktorý je nepovinný len pre záujemcov

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



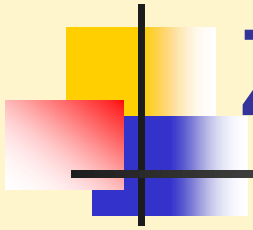


Čísla - help

- complex
- real
- rational
- integer

+,-, *	(* 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

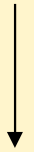
Zoznam je to, čo
má hlavu a chvost



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-Cdrling 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)

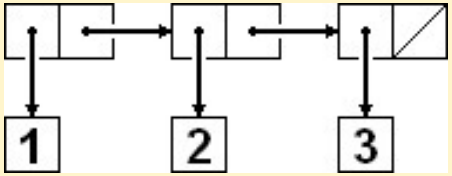
  (cdr '((1 2) 3 4)) vráti (3 4)

- **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
```

zreťazenie zoznamov

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

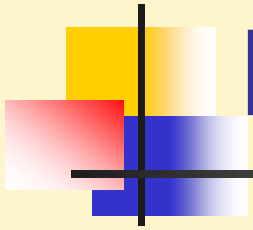
```
(append '(1 2 3) '(a b c))
    (1 2 3 a b c)
```

otočenie zoznamu

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

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

častá chyba



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
```

(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))
()

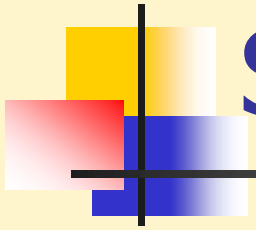
lebo (eq? '(1 2) '(1 2)) je #f

- 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



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)
```

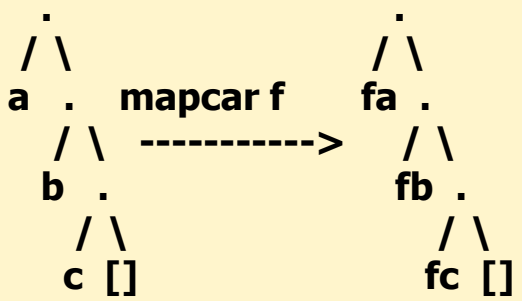
porozmýšľajte, ako odstrániť append z tejto definície

ako napísať flat len s jedným rekurzívnym volaním

Mapping

- mapcar ; aplikuj funkciu na každý prvok zoznamu

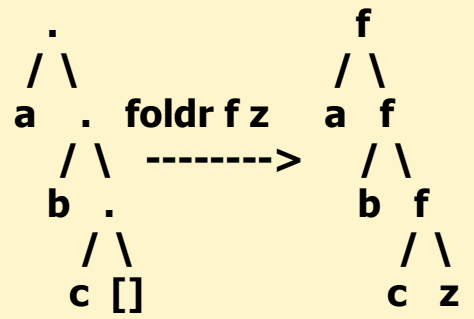
```
(define (mapcar fun lis)
   (cond
      ((null? lis) '())
      (else (cons (fun (car lis))
                  (mapcar fun (cdr lis))))))
```




```
(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 funckia

foldr



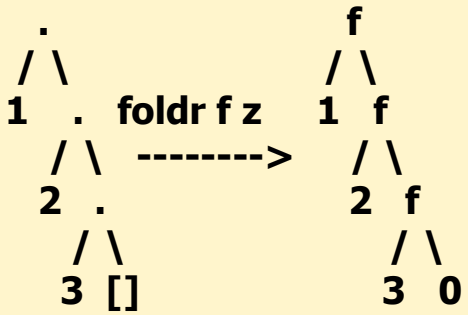

```
(define foldr
  (lambda (func zero lis)
    (if (null? lis)
      zero
      (func (car lis) (foldr func zero (cdr lis))))))
```

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

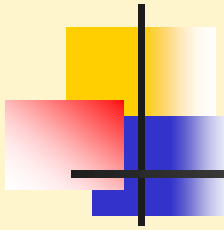
321

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

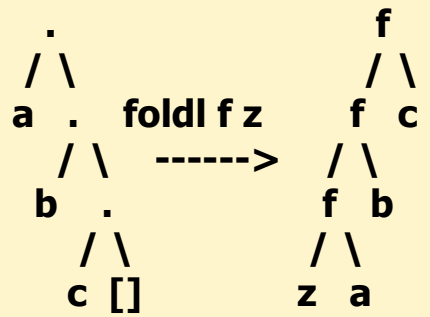
60



porozmýšľajte, ako definovať append pomocou foldr

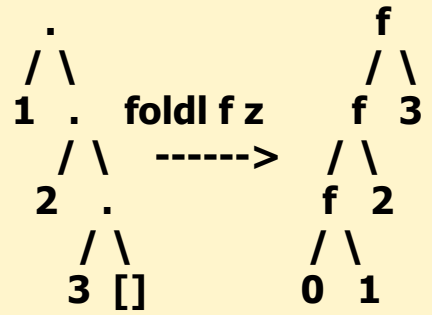


foldl

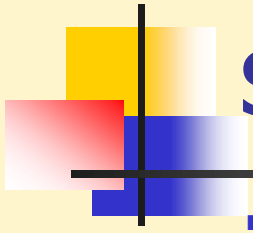



```
(define foldl
  (lambda (func accum lis)
    (if (null? lis)
      accum
      (foldl func (func accum (car lis)) (cdr lis)))))
```

```
(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

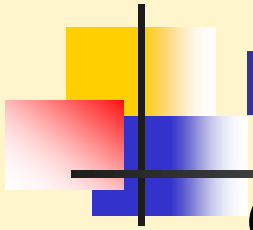


spoj a rozpoj

- (spoj '(1 2 3) '(4 5 6)) = ((1 4) (2 5) (3 6))
- (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
      (list
        (cons (caar pairs) (car (rozpoj (cdr pairs))))
        (cons (cadar pairs) (cadr (rozpoj (cdr pairs))))))))
```

```
(rozpoj '())
        ()
(rozpoj '((1 4)))
        ((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)
```

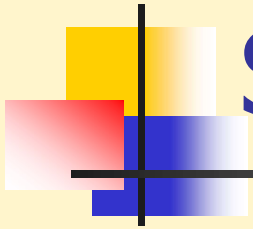
priradí do premenných var_i hodnoty výrazov $expr_i$ a vyhodnotí výraz expr

Queens

doposial dobre položené dámy

```
(define btrackRow (lambda (col row queens)
  (if (> row N)                    ; skús položiť dámu do riadku row v stĺpci col
    #f
    (or
      (and (safe row row row queens)  (btrack (+ col 1) (cons row queens)))
      (btrackRow col (+ row 1) queens))
    )
  )
)

(define btrack (lambda (col queens)          ; skús položiť dámu do stĺpca col
  (if (> col N)
    queens
    (btrackRow col 1 queens)
     )
  )
)
```

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



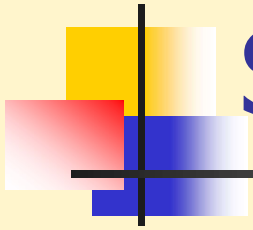
Safe

(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)))))
  ))
```

cykly sú, ale neprezradím



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 ((<var_1> <$expr_1$>) …
  (<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=iteruj 3 f = \x -> x+6,
   f = \x -> x+2
1. f^3=iteruj 3 f = \x -> 27*x
   f = \x -> 3*x
1. f^3=iteruj 3 f = \x -> x+13
   f = \x -> x+13/3
1. f^3=iteruj 3 f = \x -> 11*x
   f = \x -> $\sqrt[3]{11} * x$
1. f^3=iteruj 3 f = \x -> x^8
   f = \x -> x*x
1. f^3=iteruj 3 f = \x -> x^7
   f = ???

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