

FLP příprava na semestrálku

- Jako v minulém zkouškovém, používejte komentáře - namísto psaní poznámek do textu.

NEMATE NEKDO PDF S REŠENÍM 2013?

Odkazy materiály:

haskell - práce se stromy, soubory, lambda kalkulem: <https://gist.github.com/Pitel/2634678>

prolog - acyklické cesty po šachovnici, nejkratší cesta orientovaným grafem, lambda kalkul, podmnožiny/podřetězce/palindrom

haskell,prolog - vypsiky

https://github.com/petrnohejl/Vypisky/blob/master/haskell_vypisky.pdf

https://github.com/petrnohejl/Vypisky/blob/master/prolog_vypisky.pdf

Haskell - prelude - definice funkcí

<http://undergraduate.csse.uwa.edu.au/units/CITS3211/lectureNotes/tourofprelude.html#take>

<https://www.haskell.org/onlinereport/standard-prelude.html>

foldl :: (a -> b -> a) -> a -> [b] -> a

foldl f z [] = z

foldl f z (x:xs) = foldl f (f z x) xs

foldl1 :: (a -> a -> a) -> [a] -> a

foldl1 f (x:xs) = foldl f x xs

foldl1 _ [] = error "Prelude.foldl1: empty list"

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr f z [] = z

foldr f z (x:xs) = f x (foldr f z xs)

foldr1 :: (a -> a -> a) -> [a] -> a

foldr1 f [x] = x

foldr1 f (x:xs) = f x (foldr1 f xs)

foldr1 _ [] = error "Prelude.foldr1: empty list"

map :: (a -> b) -> [a] -> [b]

map f [] = []

map f (x:xs) = f x : map f xs

```

(++ :: [a] -> [a] -> [a])hmm
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

```

```

concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss1

```

```

concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f(f (prev x) (prev y))

```

Pevný bod: 2015 radny, 1) nadefinovat s nim pow (umocnovani x^y)

Pevný bod, vlastnost pevného bodu, definovat funkci pow

Pomocou: prev, iszero, mult, ternárneho operátoru

Alternatívne riešenie podľa referenčných riešení:

Pro výraz E je pevný bod k: $E_k = k$

Operátor pevného bodu Y: $YE = E(Y E)$

LET pow = Y E

LET E = $\lambda x y. \text{iszero}(y) ? 1 : \text{mult } x \text{ } f \text{ } x \text{ } (\text{prev } y)$

rekl bych ze je treba zavorkovat (podle konvence je aplikace levo asociativni) tedy:

LET E = $\lambda x y. \text{iszero}(y) ? 1 : \text{mult } x \text{ } (f \text{ } x \text{ } (\text{prev } y))$

...

pow 8 1

(Y E) 8 1

E (Y E) 8 1

E pow 8 1

$\lambda x y. \text{iszero } y ? 1 : \text{mult } x \text{ } f \text{ } x \text{ } (\text{prev } y)$

$\text{iszero } 1 ? 1 : \text{mult } 8 \text{ } \text{pow } 8 \text{ } 0$

$\text{mult } 8 \text{ } \text{pow } 8 \text{ } 0$

$\text{mult } 8 \text{ } (Y E) \text{ } 8 \text{ } 0$

$\text{mult } 8 \text{ } E \text{ } (Y E) \text{ } 8 \text{ } 0$

$\text{mult } 8 \text{ } E \text{ } \text{pow } 8 \text{ } 0$

$\text{mult } 8 \text{ } \lambda x y. \text{iszero } y ? 1 : \text{mult } x \text{ } f \text{ } x \text{ } (\text{prev } y) \text{ } \text{pow } 8 \text{ } 0$

$\text{mult } 8 \text{ } \text{iszero } 0 ? 1 : \text{mult } 8 \text{ } \text{pow } 8 \text{ } 0$

¹ oe

mult 8 1

8

nevím, jaké funkce mohli používat, takže předpokládám, že zakázána není žádná použita

LET pow = $\lambda x y . \text{iszero } y ? 1 : \text{powfn } x y x$

LET powfn = Y powf

LET powf = $\lambda f x y r . \text{iszero}(\text{prev } y) ? r : f x (\text{prev } y) (\text{mult } r x)$ // patří tam $\text{iszero}(\text{prev } y)$? - jo, jinak by se to mohlo vynásobit nulou. WTF? když tam bude $\text{iszero}(\text{prev } y)$, tak to nefunguje pro x^1

LET pow = $(\lambda a b . \text{powfn } a b 1)$

LET powfn = Y powf

LET powf = $(\lambda f a b r . (\text{iszero}(b) ? r : f a (\text{prev } b) (\text{mul } r a)))$

Pevný bod: 2014 radny, 1) nadefinovat s ním GE (vetsi nebo rovno) ($x \geq y$)

LET ge = Y ($\lambda f x y . \text{iszero } x ? (\text{iszero } y ? \text{True} : \text{False}) : (\text{iszero } y ? \text{True} : f (\text{prev } x) (\text{prev } y))$))

alternativně řešení: $\lambda f x y . \text{iszero } y ? \text{true} : (\text{iszero } x ? \text{False} : f (\text{prev } x) (\text{prev } y))$

Pevný bod: 2013 radny, 1) nadefinovat s ním minus

nevím, jaké funkce mohli používat, takže předpokládám, že zakázána není žádná použita

LET minus = $\lambda x y . \text{minusfn } x y$

LET minusfn = Y minusf

LET minusf = $\lambda f x y . \text{iszero } x ? 0 : (\text{iszero } y ? x : f (\text{prev } x) (\text{prev } y))$ --- jde to i jednodušeji, viz níže

co znamená to **prev?** $\text{prev } x = x-1$

Pevný bod: 201? radny, 1) nadefinovat s ním NEQ <http://fit.ipoul.cz/flp/#1125>

pomocí iszero a prev a operátoru nadefinovat neq (nerovno)

LET neq = $\lambda x y . \text{neqfn } x y$

LET neqfn = Y neqf

LET neqf = $\lambda f x y . \text{iszero } x ? (\text{iszero } y ? \text{False} : \text{True}) : \text{iszero } y ? \text{True} : f (\text{prev } x) (\text{prev } y)$

Radny 2013

Haskell, lambda kalkul, funkce která vrátí 1 když jsou dva **lambda** výrazy shodné, 0 když jiné proměnné (ne alfa konverze) a -1 když jsou různé Nevideli jste někde podrobnější zadání?

2013 dukaz - totožný na řádném termínu 2016

Haskell - dukaz - funkce $\text{foA } f [] = \text{id}$, $\text{foA } f x:xs = f x . \text{foA } f xs$, dokázat že $\text{foA } f x a = \text{foldr } f a x$

Riadny termin:

Body vacsinou len orientacne, ale myslim ze relatne vzťahy ($<$, $>$, $=$) budu platit 😊. Cas riesenie

2h45min.

Lambda kalkul / Haskell:

1. Definícia vlastnosti operatoru pevneho bodu. Potom pomocou tohoto operatoru, iszero a prev nadefinovat minus, ktore berie 2 cisla a odcita ich $a - b$, pricom vysledok je nezaporny (tj. $=0$ ak je b vacsie ako a).

Mali sme zohladnit, ze iszero, prev a cisla maju neznamu definiciu, ale znamy vyznam. True a False je mozne si nadefinovat podla seba.

[pribl. 6 bodov]

```
let T = \ x y . x
```

```
let F = \ x y . y
```

necht' Y je operátor pevného bodu, E je lambda-výraz a k je pevný bod pro E , potom :

```
Y E = k = E k
```

```
Y E = E (Y E) ~ k = E k
```

```
LET minus = Y (\ f x y. iszero y ? x : (f (prev x) (prev y)))
```

2. Nadefinujte funkciu f , aby platilo $zp\ xs\ ys = zpW\ f\ xs\ ys$ a nasledne dokazte:

1. $zp\ []\ _ = []$
2. $zp\ _\ [] = []$
3. $zp\ (x:xs)\ (y:ys) = (x,y) : zp\ xs\ ys$
4. $zpW\ _\ []\ _ = []$
5. $zpW\ _\ _\ [] = []$
6. $zpW\ f\ (x:xs)\ (y:ys) = f\ x\ y : zpW\ f\ xs\ ys$
7. $f\ x\ y = (x,y)$

$zp\ xs\ ys == zpW\ f\ xs\ ys$

1)

```
xs == [], forall ys!  
L = zp [] ys = []      // 1  
P = zpW f [] ys = []   //4  
L = P
```

2)

```
ys == [], forall xs!  
L = zp xs [] = []      // 2  
P = zpW f xs [] = []   // 5  
L = P
```

3)

```
xs = (a:as), ys = (b:bs)  
I.P. = zp as bs = zpW f as bs    // Indukční Předpoklad
```

```

L = zp (a:as) (b:bs) =           // 3
  = (a,b) : zp as bs =           // 1P
  = (a,b) : zpW f as bs         // 7
  = f a b : zpW f as bs         // 6
  = zpW f (a:as) (b:bs)
  = P

```

Q.E.D.

[8 bodov]

3. Napisat funkciu sort v holom Haskellu, ktora berie zoznam hodnot nad triedou Ord a vracia zordene od najmensieho po najvacsie.

-- len pre ujasnenie typova definicia. V zadani nebola, ale bolo podrobnejsie popisane
 sort :: Ord a => [a] -> [a]

Mohli sme pouzít konstrukcie zoznamu, fold*, map a operacie nad triedov Ord.

```

sort [] = []
sort (x:xs) = foldr ins [x] xs
where
  ins y [] = [y]
  ins y l@(z:zs) = if y > z then z : ins y zs else y : l

```

[pribl. 4 bodov]

4. V Haskellu nadefinovat funkciu pt, ktora berie nazov suboru ako argument. Z tohoto suboru nacita zaznamy v tomto formate Cislo_typu_Integer#String, pripadne prazdny riadok. Zaznam reprezentovat datovym typom DLog. Nasledne vypisat tie zaznamy, ktore su maju prve cislo nasobkom 10 oddelene koncom riadku. Odelene budu tentoraz dvojbodkou (:).

Je potrebne uviest typove definicie pre kazdu pouzitu funkciu.

Poskytnute typove definicie pre pracu s IO (openFile, hGetContents, lines, unlines, print, ...)

[10 bodov]

```

data DLog
  = DVal Integer String
  | DNull
  deriving (Show,Eq)

```

```

notNullV :: DLog -> Bool
notNullV (DVal _ _) = True
notNullV _ = False

```

```

strV :: DLog -> String
strV (DVal i s) = show i ++ ":" ++ s

```

```

pline :: String -> DLog
pline l =
    if null l then DNull else DVal ((read time)::Integer) val
    where
        time = takeWhile (\x -> elem x ['0'..'9']) l
        val = tail$ dropWhile (/='#') l

cvf
isM10 :: DLog -> Bool
isM10 (DVal i _) = (i `mod` 10) == 0

```

```

pt :: String -> IO ()
pt f = do
    h <- openFile f ReadMode
    c <- hGetContents h
    let ml = map pline $ lines c
    let nml = filter notNullV ml
    let d10 = filter isM10 nml
    putStrLn $ unlines $ map strV d10
    hClose h

```

:

1. Vytvorit predikat e, ktorý berie 2 argumenty. Prvým argumentom je ľubovoľne zánorený zoznam zoznamov (aj prázdnych) napr. `[[], [1, 2], [[[]]], [atom, atom]]`
 Druhý argument je výstupný zoznam hodnôt bez zánorenia. [pribl. 6 bodov]

```

e([], []).
e([[] | YS], ZS) :- e(YS, ZS),!.
e([[H | T] | YS], ZS) :- e([H | [T | YS]], ZS),!.
e([X | XS], [X | ZS]) :- e(XS, ZS).

```

2. Napísať predikat add, ktorý berie 2 vstupné argumenty. Prvý argument je zoznam čísl (neomezená dĺžka), druhý argument číslo a výsledkom je súčet opäť reprezentovaný zoznamom čísl napr `add([1, 1], 99, X)`. `X = [1,1,0]`.

Použit je možné operátor rezu, konštrukcie zoznamov, unifikáciu, aritmetické operácie, reverse a to je tuším všetko (viac som ani nepotreboval)
 [pribl. 7 bodov]

```

add(DS,V,RS) :- reverse(DS,RD), sumx(RD,V,[],RS).
sumx([D|DS],C,VS,XS) :-

```

```

    S is C+D, NC is S // 10, V is S - NC*10,
    sumx(DS,NC,[V|VS],XS) .
sumx([],0,VS,VS) :- !.
sumx([],C,VS,XS) :-
    NC is C // 10, V is C - NC*10,
    sumx([],NC,[V|VS],XS) .

// Dle meho nazoru elegantnejši reseni:

add(A1,A2,Result) :- getNum(A1,0,AA1), R is AA1+A2, getList(R,[],Result) .

getNum([],Acc,Acc) .
getNum([H|T],Acc,Res) :- Accn is 10*Acc+H, getNum(T,Accn,Res) .

getList(0,Acc,Acc) .
getList(Number,Acc,Result) :-
    A is Number // 10, B is Number mod 10,
    getList(A,[B|Acc],Result) .

```

3. Napisat predikat lookup. Prvy argument vhodne reprezentovana tabulka symbolov, 2-hy argument kluc, 3-ti argument hodnota. A posledny a vysledny predikat je modifikovana, pripadne vstupna tabulka symbolov.

Predikat pracuje v dvoch rezimoch. Ak je zadana hodnota, tak sa modifikuje pripadne modifikuje zaznam v 901379

969841

tabulke symbolov. Ak nie je zadana hodnota, tak vyhladavame v tabulku hodnotu so zadanyim klucom.

Ak sa nemylim, tak bolo mozne pouzit vsetko zo zakladnej kniznice Prologu. Ja som pouzil var(), nonvar() na zistenie, ci (nie) je zadana hodnota a nemyslim si, ze by to bolo v zadani spomenute. r [pribl. 8 bodov]

```

lookup(T,_,_,_) :- var(T),!,fail.
lookup(T,N,V,NT) :- nonvar(N), nonvar(V), ins(T,N,V,NT), !.
lookup(T,N,V,NT) :- nonvar(N), tst(T,N,V), T=NT.

ins([], N, V, [(N,V)]).
ins([(N,_) | VS], N, V, [(N,V) | VS]) :- !.
ins([X|XS], N, V, [X|VS]) :- ins(XS, N, V, VS).

tst([], _, _) :- !, fail.
tst([(N,V) | _], N, V) :- !.
tst([_ | VS], N, V) :- tst(VS, N, V).

```

4. Napisat predikat `search(Pozice, Cesty)`, ktory najde vsetky cesty od pozicie `a` naspat dlzky 20 az 22 (netrapit sa tym, ci prvý a posledný prvok zarátat).

Definicia pozicie je neznáma, k dispozíci je `nextStep(Pos)` nad neznámym a nekonečným stavovým priestorom. Každý prvok je možné nastaviť len jeden krát vyjma prvého (`=` posledného).

[pribl. 11 bodov]

```
search(P,LL) :-
    retractall (pos(_)),
    bagof(L,track(P,P,0,L),LL).

%track(Zacatek(resp. aktualni pozice), Cil, Pocitadlo,[Cesta (seznam pozic)])
track(P,P,N,[P]) :- N >= 20, N <= 22, !.
track(P,P,N,_ ) :- (N>0,!, fail;N==0,fail).
track(A,P,N,[A|T]) :-
    N < 22,
    assertz(pos(A)),
    nextStep(A,B),
    (not(pos(B)) ;B==P),
    NN is N+1,
    track(B,P,NN,T).

track(A,_,_,_) :-
    pos(A),
    retract(pos(A)),
    !, fail.
```

=====LAMBDA KALKUL=====

1. pevný bod, nadefinovať GE

```
let gef = \ f x y . iszero x ? iszero y : ( iszero y ? True : (f (prev x) (prev y)))
```

```
let GE = Y gef
```

1. Lambda nadefinovať False a EQU funkciu pre porovnanie dvoch hodnôt, a demonštrovať nad EQU False False

```
LET EQU = \ab.ab(NOT b)
```

```
LET NOT = \a.a F T
```

```
LET T = \xy.xy
```



```
LET F = \xy.y
```

```
LET EQU = \ab.a(\t.b)(\b.b F T)
```

```
LET EQU = \ab.a (\t.b) (b (\z.F) T)
```

2. Definovat True a False a pomoci pevneho bodu definovat a^b (predpokladam, ze mult a prev a iszero je k dispozici)

```
LET fpow = \f a b.(iszero b) ? 1 : mult a (f a (prev b))
```

```
LET pow = Y fpow.
```

1)Cisty -\ kalkul, mame iszero a prev a cisla definovana jako ve slajdech a vytvorit neq, ktere bere dve cisla, pokud jsou rozna vraci True, jinak False. True a False nadefinovat dle libosti, zbytek co bude treba dodefinovat.

```
LET T = \x y.x
```

```
LET F = \x y.y
```

```
LET fneq = \f x y. iszero(x) (iszero(y) False True) (iszero(y) True (f (prev x) (prev y)))
```

```
LET neq = Y fneq
```

```
LET fneq = \f x y . (iszero x) ? ( (iszero y) ? False : True ) : ((iszero y) ? True : f (prev x) (prev y) )
```

```
LET neq = Y fneq
```

podle mne staci i takto:

```
LET T = \x y.x
```

```
LET F = \x y.y
```

```
LET neq = \xy. iszero (y prev x) F T //jakoze pokud je rozdil cisel 0 tak vratime F, jinak T
```

```
LET neq = \xy. iszero (y sub x) F T
```

===== **HASKELL** =====

3. parsovane suboru do struktury (na kazdom riadky je meno,id, cas procesu - tvar casu h:m:s, struktura pracuje iba so sekundami)

- openFile
- readFile
- lines

- na parsovani
 - span
 - dropWhile
 - read(x)::Integer/Char...-

```
-- xlogin,1,h:m:s
data DTime =
    DVal String Integer Integer
  | DNull
  deriving (Show, Eq)

toStructure :: String -> DTime
toStructure x =
    if null x then DNull else DVal proc (read(id)::Integer) second
  where
      (proc, l) = span(/=',') x -- ("xlogin", "1,h:m:s")
      (id, t) = span(/=',') $ tail l -- ("1", "h:m:s")
      time = tail t -- h:m:s
      (hours, rest) = span(/=':') time -- ("h", ":m:s")
      (min, s) = span(/=':') $ tail rest -- ("m", ":s")
      ss = tail s -- s
      second = (read(ss)::Integer) + ((read(min)::Integer) * 60) +
        ((read(hours)::Integer)*60*60) -- working only with seconds

parsingFile :: FilePath -> IO()
parsingFile file = do
    f <- openFile file ReadMode
    h <- hGetContents f
    let par = map toStructure $ lines h
    putStrLn $ show par
```

2. Nadefinovat vlastní strukturu pro seznam a udělat fibonacciho posloupnost nad tímto typem

```
data Seznam a
    = Konec
    | Prvek a (Seznam a)
  deriving Show
```

```

fib :: Int -> Seznam Int
fib n =
    if n < 0
    then error "Funkce vyžaduje nezáporné číslo!"

    else fib2 0 1 n where
        fib2 :: Int -> Int -> Int -> Seznam Int
        fib2 x _ 0 = Prvek x Konec
        fib2 x y n = Prvek x (fib2 y (x + y) (n - 1))

```

4. funkci pro zjištění zda je první seznam prefixem druhého nebo naopak

```

jsouPrefixem :: Eq a => [a] -> [a] -> Bool
jsouPrefixem [] _ = True
jsouPrefixem _ [] = True
jsouPrefixem (x:xs) (y:ys) = if x == y then jsouPrefixem xs ys else False

```

5. číst soubor a vypisovat pouze řádky, které nejsou prefixem předcházejícího ani následujícího

```

hlavni :: String -> IO ()
hlavni inputFileName = do
    h <- openFile inputFileName ReadMode
    c <- hGetContents h
    let ls = lines c
    let lsFiltered = filterLines ls
    putStr $ unlines $ lsFiltered
    hClose h

filterLines :: [String] -> [String]
filterLines [] = []
filterLines [x] = [x] -- na vstupu pouze jeden radek, ten nema zadny predchazejici ani nasledny radek
filterLines (x:xx:xs) = if not (isPrefix x xx) then [x] ++ (filterLines' x xx xs) else filterLines' x xx xs -- solo
zpracuju prvni radek
where
    -- predchazejici radek, aktualni radek, nasledujici radky
    filterLines' :: String -> String -> [String] -> [String]
    filterLines' pred act [] = if not (isPrefix pred act) then [act] else [] -- zpracovavam posledni radek

```

```

        filterLines' pred act (nasl:xs) = if not ((isPrefix pred act) || (isPrefix act nasl))
            then act : (filterLines' act nasl xs)    -- není prefixem předcházejícího ani následujícího,
            pridam do seznamu
            else filterLines' act nasl xs            -- je prefixem předcházejícího nebo následujícího,
            preskocim

-- funkce nahore (cast 4.)
-- vraci true, jestliže první seznam je prefixem druhého nebo naopak
-- vstupem jsou seznamy, cíli i String
isPrefix :: (Eq a) => [a] -> [a] -> Bool
isPrefix _ [] = True
isPrefix [] _ = True
isPrefix (x:xs) (y:ys) = if x == y then (isPrefix xs ys) else False

```

1. udělat databázovou projekci tabulky

Neumožňuje projektované schéma se sloupci v jiném uspořádání, než v jakém je má schéma tabulky:

```

project :: Schema -> Table -> Table
project s t = project2 s t ([], []) where
    project2 :: Schema -> Table -> Table -> Table
    project2 [] _ t = t
    project2 l@(x:xs) ((y:ys), (z:zs)) (s, t) =
        if (x == y)
            then project2 xs (ys, zs) ((s ++ [y]), (t ++ [z]))
            else project2 l (ys, zs) (s, t)
    project2 _ ([], _) t = t
    project2 _ (_, []) t = t

```

1. Definujte datovou strukturu pro reprezentaci lambda-kalkulu. (Datový typ proměnné je neznámý.)
Napište funkci, která pro dva lambda-výrazy a proměnnou zjistí, zda je substituce platná. K dispozici máte pouze holý Haskell, fold*, map apod.

```

data LExpr =
    LVar String
    | LAbs String LExpr
    | LApp LExpr LExpr
    deriving Show

-- (\x.\y.xz) (y)
expr = LApp (LAbs "x" (LAbs "y" (LApp (LVar "x") (LVar "z")))) (LVar "y")

```

```

main = do
    print $ sub expr "x" (LAbs "y" (LVar "y"))

freeVars :: LExpr -> [String]
freeVars le = fv le [] where
    fv (LApp e1 e2) acc = (fv e1 acc) ++ (fv e2 acc)
    fv (LAbs var e) acc = (fv e (acc ++ [var]))
    fv (LVar var) acc = if elem var acc then [] else [var]

sub le var les = s le var [] (freeVars les)
    where
        s (LVar v) var bounding free =
            if v /= var then True
            else
                -- v == var
                if (intersect bounding free) /= [] then False
                else True
        s (LAbs a le) var bounding free = s le var (bounding++[a]) free
        s (LApp le1 le2) var bounding free =
            s le1 var bounding free && s le2 var bounding free

```

Asi to lze udělat jednodušeji, tohle mě jenom napadlo během deseti minut, je to alfa-konverze se vším všudy: <http://pastebin.com/PEPGKjCG> (mimochodem, substituce není alfa-konverze, je to mimo otázku).

3. Napište funkci v Haskellu, která otevře tři soubory (2 pro čtení, třetí výstupní). V prvním souboru bude seznam loginů podle zvykostí FITu, v druhém bude nějaký text. Program do třetího souboru zapíše text z druhého souboru tolikrát, kolik je v prvním souboru loginů, zaměňuje každý výskyt řetězce xzzzzz99 za právě zpracovávaný login. Loginy budou ve výstupním souboru ve stejném pořadí jako ve vstupním souboru s loginy.

```

getFile :: FilePath -> IO [String]
getFile file = do
    contents <- readFile file
    return (lines contents)

replaceLine :: String -> String -> String
replaceLine login line = unwords $ map (\ x -> if x == "xzzzzz99" then login else x) (words line)

replaceAll :: [String] -> [String] -> [String]
replaceAll [] _ = []
replaceAll (login:logins) lines = map (replaceLine login) lines ++ replaceAll logins lines

loginText = do

```

```
login <- getFile "login.txt"
text <- getFile "text.txt"
writeFile "result.txt" (unlines $ replaceAll login text)
```

3)Nadefinovat strukturu pro osobu = (jmeno, prijmeni,id typu int), nadefinovat data pro BST (binary search tree) a nacist osoby ze souboru (readData funkce), ve kterem jsou na kazdem radku ve tvaru prijmeni:jmeno:id a spravne je vlozit do BVS (binarni vyhledavaci strom), pricemz osoby se porovnávají prvé dle prijmeni, pak dle jmena a nakonec dle id. Cisty haskell + par IO funkci. Psat typy funkci. (Body: Pr 1,2={8,10}, 3=12)

```
type Surname = String
type Name = String
type Id = Integer

data Bvs a = End
  | Item (String, String, Integer) (Bvs a) (Bvs a)
  deriving (Show)

main = do
  h <- openFile "input" ReadMode
  c <- hGetContents h
  print $ procLines (lines c) End
  hClose h

-- nedoporučuju používat klíčová slova (lines je vestavěná funkce) pro proměnné
procLines :: [String] -> Bvs -> Bvs
procLines [] bvs = bvs
procLines (line:lines) bvs = procLines lines $ insert sur n (read(id)::Integer) bvs
  where
    (sur,r1) = span (/=':') line
    (n,r2)   = span (/=':') $ tail r1
    id      = tail r2

-- možná by slo i nějak kratšejc
insert :: Surname -> Name -> Id -> Bvs -> Bvs
insert sur n id End = Item sur n id End End
insert sur n id (Item sur1 n1 id1 l r)
  | sur < sur1 = Item sur1 n1 id1 (insert sur n id l) r
  | sur > sur1 = Item sur1 n1 id1 l (insert sur n id r)
  | n < n1 = Item sur1 n1 id1 (insert sur n id l) r
  | n > n1 = Item sur1 n1 id1 l (insert sur n id r)
  | id < id1 = Item sur1 n1 id1 (insert sur n id l) r
  | id > id1 = Item sur1 n1 id1 l (insert sur n id r)
```

```
| otherwise = Item sur1 n1 id1 l r
```

3. Nacist soubor a pro kazdy radek zjistit který znak se na nem vyskytuje nejvickrat a nejminkrat a vypsat to na stdout.

```
getFile :: FilePath -> IO [String]
getFile file = do
    contents <- readFile file
    return (lines contents)

checkCount :: String -> String
checkCount line = show ch1 ++ ": " ++ show i1 ++ ", " ++ show ch2 ++ ": " ++ show i2
    where
        counts = [(length $ filter (==x) line, x) | x <- line]
        (i1, ch1) = maximum counts
        (i2, ch2) = minimum counts

checkCountFile = do
    l <- getFile "login.txt"
    putStr $ unlines $ map checkCount l
```

=====PROLOG=====

Cesty koně ze cvik

Vytvořte predikát cesty/7, který zjistí **počet** acyklických cest (N), kterými se může šachový kůň dostat z počáteční pozice (XS, YS) do cílové pozice (XE, YE) na šachovnici o zadaných rozměrech (XR, YR):

cesty(XR, YR, XS, YS, XE, YE, N)

```
test, jestli policko je na sachovnici
testPoz(X,Y) :-
    velikost(XR, YR), % nacteme si velikost sachovnice
    X > 0, Y > 0,
    X =< XR, Y =< YR.

% predikaty pro pohyb kone na sachovnici
skok(X,Y,XN,YN) :- XN is X + 2, YN is Y + 1, testPoz(XN, YN).
skok(X,Y,XN,YN) :- XN is X + 2, YN is Y - 1, testPoz(XN, YN).
```

```

skok(X,Y,XN,YN) :- XN is X - 2, YN is Y + 1, testPoz(XN, YN).
skok(X,Y,XN,YN) :- XN is X - 2, YN is Y - 1, testPoz(XN, YN).
skok(X,Y,XN,YN) :- XN is X + 1, YN is Y + 2, testPoz(XN, YN).
skok(X,Y,XN,YN) :- XN is X + 1, YN is Y - 2, testPoz(XN, YN).
skok(X,Y,XN,YN) :- XN is X - 1, YN is Y + 2, testPoz(XN, YN).
skok(X,Y,XN,YN) :- XN is X - 1, YN is Y - 2, testPoz(XN, YN).

% jedna cesta kone
cesta(X,Y,X,Y,[X:Y]) :- !.
cesta(X,Y,XE,YE,[X:Y|T]) :-
    assertz(pozice(X, Y)),
    skok(X,Y, X2, Y2),
    \+ pozice(X2, Y2), % not
    cesta(X2, Y2, XE, YE, T).

cesta(X, Y, _, _, _) :- retract(pozice(X, Y)), !, fail. % pri zpetnem vynorovani z rekurze
musime po sobe uklidit

% cesty kone - celkova struktura
cesty(XR, YR, XS, YS, XE, YE, N) :-
    XR > 0, YR > 1,
    assertz(velikost(XR, YR)),
    findall(C, cesta(XS, YS, XE, YE, C), B),
    length(B, N), % delka N je pozadovany vysledek
    retractall(poz(_, _)),
    retract(velikost(_, _)). % vymazani velikosti z databaze, jinak by tam byla 2x a
to je problem

```

2. substituce lambda kalkulu v prologu

3. prohledávání stavového prostoru

4. šílenost s vyhodnocováním jednoduchých rovnic

5. test jestli je množina podmnožinou druhé

verze pro množiny (ne seznam):

```

submn([], _).
submn([H|T], YS) :- mem(H,YS), submn(T,YS).
mem(X,[X|T]) :- !.

```



```
mem(X,[H|T]) :- mem(X,T).
```

```
podseznam( _, []).
```

```
podseznam( [X|XS], [X|XSS] ) :- podseznam2( XS, XSS ).
```

```
podseznam( [_|XS], X ) :- podseznam( XS, X ).
```

```
podseznam2( _, []).
```

```
podseznam2( [X|XS], [X|XSS] ) :- podseznam2( XS, XSS ).
```

--už jsem začal “kontrolovat” podseznam, takže už nemůžu v původním seznamu nic přeskočit.

Proto tady chybí ten poslední řádek a tím se to liší od podmnožiny (kde i potom co začnu porovnávat hodnoty můžu něco přeskočit)

zajímavá implementace podseznamu:

```
suffix(Xs, Ys) :- append( _, Ys, Xs).
```

```
prefix(Xs, Ys) :- append(Ys, _, Xs).
```

```
sublist(Xs, Ys) :-
```

```
    suffix(Xs, Zs),
```

```
    prefix(Zs, Ys).
```

1) Najít nejkratší cestu na šachovnici (i obdelníkové) mezi dvěma body s tím, že na pole kde je bariera (predikát) není možno vstoupit. <http://prntscr.com/752e6r>

1. Napište predikát, který vrátí počet všech acyklických cest po šachovnici zadaných rozměrů (i obdelníkové) s maximálně 3 změnami směru...00

2. Napište predikát, který prohledá binární strom, a pokud se v něm vyskytuje zadaný podstrom, hodnoty uzlů ve všech místech jeho výskytu nahradí hodnotou NewValue. Pokud se zadaný podstrom ve stromě nevyskytuje, výsledkem bude původní strom.

3. Napište predikát, který vrátí seznam všech palindromů délky alespoň 3 vyskytujících se jako podřetězec v řetězci zadaném seznamem.

4. Napište jediný predikát, který dělá to, co v Haskellu (`\f l -> map f $ concat l`).

% jediným predikátem to neumím, musel jsem jich napsat víc

```
% dohromady(Funkce, List, Vystup)
```

```
% dohromady(increment, [[1,2],[3,4]], X).
```

```
dohromady(Funkce, List, Vystup) :-
```

```
    concat(List, ListConcat),
```

```
mapMy(Funkce, ListConcat, Vystup).
```

```
concat([], []).
```

```
concat([S|SS], Res) :-
```

```
    concat(SS, SSRes),
```

```
    appendMy(S, SSRes, Res).
```

```
appendMy([], L, L).
```

```
appendMy([H|T], L, [H|Res]) :-
```

```
    appendMy(T, L, Res).
```

```
mapMy(_, [], []).
```

```
mapMy(Funkce, [H|T], [RH|RT]) :-
```

```
    CC =.. [Funkce, H, RH],
```

```
    call(CC),
```

```
    mapMy(Funkce, T, RT).
```

```
increment(X,Y) :- Y is X + 1.
```

1. v prologu najvacsi spolocny delitel (gcd)

<http://stackoverflow.com/questions/22450582/program-for-finding-gcd-in-prolog>

Tohle mi pripadá jednodušší - snad je to 100% správně

```
gcd(A,0,A).
```

```
gcd(A,A,A).
```

```
gcd(A,B,D) :- A>B, (B>0), R is A mod B, gcd(B,R,D).
```

```
gcd(A,B,D) :- A<B, gcd(B,A,D).
```

```
% dalsi reseni
```

```
gcd(X,X,X) :- !.
```

```
gcd(X,Y,Res) :-
```

```
    X > Y, !, XX is X - Y,
```

```
    gcd(XX,Y,Res).
```

```
gcd(X,Y,Res) :-
```

```
    YY is Y - X, !,
```

```
    gcd(X,YY,Res).
```

```
implementace z Prelude.hs
```

```
gcd      :: (Integral a) => a -> a -> a
```

```
gcd 0 0   = error "Prelude.gcd: gcd 0 0 is undefined"
```

```
gcd x y   = gcd' (abs x) (abs y)
```

```
    where gcd' x 0 = x
```

```
          gcd' x y = gcd' y (x `rem` y)
```

```

/*gxd(X, Y, R)*/
gxd(0, Y, R) :- (Y < 0, R is -Y; Y >= 0, R is Y).
gxd(X, 0, R) :- (X < 0, R is -X; X >= 0, R is X).
gxd(X, Y, R) :- X < 0, !, XX is -X, gxd(XX, Y, R).
gxd(X, Y, R) :- Y < 0, !, YY is -Y, gxd(X, YY, R).
gxd(X, X, X) :- !.
gxd(X, Y, R) :- (X < Y, YY is Y-X, gxd(X, YY, R); XX is X-Y, gxd(XX, Y, R)).

```

2. binarna scitacka (vstup su 2 polia roznej dlzky)

```

fullAdder1(X,Y,R) :- fullAdder1(X,Y,0,R) .
fullAdder1([],_,_,[]) :- !.
fullAdder1(_,[],_,[]) :- !.
fullAdder1([H1|T1],[H2|T2],Cin,[S|T]) :-
    S1 is H1 xor H2, S is S1 xor Cin,
    C1 is Cin /\ S1, H3 is H1 /\ H2, Cout is H3 \/ C1,
    fullAdder1(T1,T2,Cout,T) .

```

3. lambda vyraz a freeVars

4. pocet ciest (stejny start/end suradnice, 20-22 skokov)

// nevite nekdo, jak toto funguje? viz odkaz na kody nahore...

hlejdecestu(S, _, _):- used(S), retract(used(S)), !, fail. % ?????

Do databáze se vkládá počátek, zvolí další pozice z něj, ověří se, že není v databázi, a pokud je, nastane backtracking, a pokud všechny následné pozice jsou v databázi, ta druhá klauzule selže a přejde se k téhle třetí. Pokud je zmíněný počátek v databázi, odstraní se z ní a cesta selže.

2)Nadefinova predikaty pro praci s BVS, pokud je zadany klic, vyhledava hodnotu, pokud hodnota, vyhledava klice, pokud hodnota i klic, predikat uspeje jen pokud hodnota s klicem je ve stromu.

```

% Key,val,lchild,rchild
bvs(_,_,_,_).
bvs(end).

initbvs(bvs(end)).
insert(K,V,bvs(end),bvs(K,V,bvs(end),bvs(end))).
insert(K,V,bvs(K1,V1,L1,R1),bvs(K1,V1,L,R1)):- K<K1,insert(K,V,L1,L).
insert(K,V,bvs(K1,V1,L1,R1),bvs(K1,V1,L1,R)):- K>K1,insert(K,V,R1,R).
insert(K,_,bvs(K,V1,L1,R1),bvs(K,V1,L1,R1)).

find(_,_,bvs(end)):-false.

```

```

find(K,V,bvs(K1,_,L1,_)):-atom(K),K<K1,find(K,V,L1).
find(K,V,bvs(K1,_,_,R1)):-atom(K),K>K1,find(K,V,R1).
find(K,V,bvs(K,V,_,_)).
find(K,V,bvs(K1,V1,L1,R1)):-find(K,V,L1);find(K,V,R1).

```

3)Nadefinovat predikat XOR pro dvojici mnozin.

```

setxor(S1,S2,R):- sxor(S1,S2,R1), sxor(S2,S1,R2), append(R1,R2,R).

```

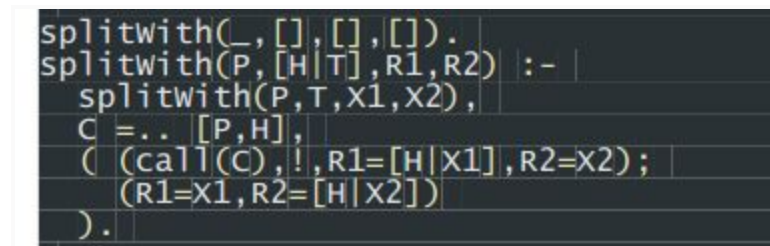
```

% prida vse, co neobsahuje druha mnozina
sxor([],S2,[]).
sxor([S1|S],S2,R):- member(S1,S2),!,sxor(S,S2,R).
sxor([S1|S],S2,[S1|R]):- sxor(S,S2,R).

```

4)Pomoci 2 (pozn: myslim, ze zde mohlo byt spise 3 :D) klauzuli vytvorit jediny predikat, který vezme predikat a seznam a rozdeli vstupni seznam na dva seznamy, přičemž v prvním budou prvky pro které predikat platí a v druhém prvky pro které neplatí - prvky musí být vůči původnímu seznamu zachovány.

(Body: Pr 1=8,2=10, 3,4={4,8})



```

splitwith(_,[],[],[]).
splitwith(P,[H|T],R1,R2):-
    splitwith(P,T,X1,X2),
    C=..[P,H],
    (call(C),!,R1=[H|X1],R2=X2);
    (R1=X1,R2=[H|X2])
).

```

1. Vypsát všechny acyklické cesty po desce z pozice jednoho místa do druhého. Pohybovat se můžu jen rovně nebo zatáčet doprava. Vycházím směr může být libovolný.

2. Určit jestli A je vlastní podmnožinou B, kde obě množiny jsou definovány seznamy. (Mohlo se použít max 2 predikatu a 3 pravidel nebo tak něco)

```

/* predikat uspeje jenom tehdy, pokud je první seznam menší než druhý seznam,
   select vrací seznam LL jako seznam L bez prvku H */
vlastniPodmnozina([],[_|_]).
vlastniPodmnozina([H|T],L):- select(H,L,LL),!,vlastniPodmnozina(T,LL).

```

3. Udelat scitacku dvou binarnich cisel definovanych ve dvou stejne dlouhych seznamech.

```

# samozřejmě to jde i pomocí unifikace a 8+ klauzulí
fullAdder2(X,Y,R):- fullAdder2(X,Y,0,R).
fullAdder2([],[],_,[]).

```

```

fullAdder2([],_,_,_) :- !, fail.
fullAdder2(_,[],_,_) :- !, fail.
fullAdder2([H1|T1],[H2|T2],Cin,[S|T]) :-
    S1 is H1 xor H2, S is S1 xor Cin,
    C1 is Cin /\ S1, H3 is H1 /\ H2, Cout is H3 \/ C1,
    fullAdder2(T1,T2,Cout,T).

```

3. V Prologu nadefinovat predikat foldl(Functor, ValueForEmptyList, ListOfValues, Result), který nad seznamem ListOfValues provede operaci foldl. (8b)

```

foldl(_,V,[],V).
foldl(F,V,[H|T], Ret) :-
    FF =.. [F,V,H,X],
    call(FF),
    foldl(F,X,T,Ret).

```

4. Vytvorit strukturu reprezentující Lambda vyraz a nadefinovat FreeVars, která vrati seznam volných promenných v seznamu

// za správnost neručím :D, ale co jsem testoval, tak to fungovalo...

// inspirováno cvikama z haskellu

```

lambda(v(_)).
lambda(apl(lambda(_), lambda(_))).
lambda(abst(v(_), lambda(_))).

freeVars(Exp, Vars):- fv(Exp, [], Vars).

// var
fv(lambda(v(V)), BL, []):- member(V,BL),!.
fv(lambda(v(V)), _, [V]).
// aplikace
fv(lambda(apl(lambda(E1),lambda(E2))), BL, Vars):-
    fv(lambda(E1),BL,Vars1),
    fv(lambda(E2),BL,Vars2),
    append(Vars1,Vars2,Vars).
// abstrakce
fv(lambda(abst(v(E1),lambda(E2))), BL, Vars):-
    fv(lambda(E2),[E1|BL],Vars).

```

/* TOTO JE Z PDF S RESENIEM.

lvar(NAME).

lapp(E1,E2).

```

labs(NAME,E).
promenna
aplikace
abstrakce
*/
freeVars(lvar(V),[V]) :- !.
freeVars(labs(N,E),R) :-
    !,
    freeVars(E,EF),
    delet(N,EF,R).
freeVars(lapp(E1,E2),R) :-
    !,
    freeVars(E1,F1),
    freeVars(E2,F2),
    union(F1,F2,R).
delet(_,[],[]) :- !.
delet(X,[X|T],R) :-
    !,delet(X,T,R).
delet(X,[H|T],[H|R]) :-
    !,delet(X,T,R).
/* member byl povolen, zde nahrazeno elem */
elem(_,[]) :- !,fail.
elem(X,[X|_]) :- !.
elem(X,[_|T]) :- elem(X,T).
union([],L,L).
union([X|T],L,U) :-
    elem(X,L),!,
    union(T,L,U).
union([X|T],L,[X|U]) :-
    union(T,L,U).
/* 4 */

```

% tohle vypada jako flatten/2

destroy - ze seznamu zanorených seznamů udělá jeden seznam

```

destroy([], []).
destroy([[]|X], Y) :- destroy(X,Y),!.
destroy([[H|T]|X],Y) :- destroy([H|[T|X]],Y),!.
destroy([H|T],[H|TT]) :- destroy(T,TT).

```

% V holom prologu nadefinovat predikat pre symetricky rozdiel dvoch mnozin

% reprezentovanych zoznamom

```
el(_, []) :- !, fail.
```

```
el(X, [X|_]) :- !.
```

```
el(X, [_|T]) :- el(X,T).
```

```
sym([],B,B) :- !.
```

```
sym([H|T],B,[H|X]) :- not(el(H,B)), sym(T,B,X).
```

```
sym([H|T],B,X) :- el(H,B), sym(T,B,X).
```

Pokud to někdo vymyslíte bez použití predikátu el, tak se klidně podělte :-)

2) Asociace pameti asociace(id, key, value), kdyz se zada key tak hleda

value, kdyz se zada value tak hleda klic a kdyz se zada oboje tak pridava do
databaze. Omezení že se nesmí používat seznamy.

```
assocMem(A,K,V) :- var(A), !, fail.  
assocMem(A,K,V) :- var(K), var(V), !, fail.  
assocMem(A,K,V) :- var(V), P =.. [A,K,V], call(P).  
assocMem(A,K,V) :- var(K), P =.. [A,Y,V], setof(Y,P,K).  
assocMem(A,K,V) :- P =.. [A,K,_], call(P), !, retract(P),  
    PP =.. [A,K,V], assert(PP).  
assocMem(A,K,V) :- P =.. [A,K,V], assert(P).
```

===== DŮKAZY =====

2. Důkaz $\text{foldr } (+) \ 0 \ xs = \text{ccat } xs$, kde $\text{ccat } [] = []$ a $\text{ccat } (xs:xss) = xs ++ \text{ccat } xss$

Důkaz $\text{foldr } (+) \ 0 \ xs + \text{foldr } (+) \ 0 \ ys = \text{foldr } (+) \ 0 \ (xs ++ ys)$ a rict, co je méně náročné a proč.

foldl se aplikuje levo asociativním způsobem a nic nevyhodnotí, dokud nedojde na konec seznamu, tedy jej nelze použít pro nekonečné struktury

foldr se naopak vyhodnocuje průběžně a nehrozí přetečení zásobníku

3. Důkaz $\text{len}(xs ++ ys) = \text{len } xs + \text{len } ys$

2. Definujte funkci $\text{add } (x, y)$, která sečte x a y . Uvažujte standardně definované funkce map a foldr a zde definovanou funkci sumIt a dokažte, že $\text{foldr } (+) \ 0 \ (\text{map } \text{add } (x:\text{yxs})) = \text{sumIt } (x:\text{yxs})$ pro všechna x typu (Int, Int) .

(Důkaz z oficiálního řešení nějaké starší zkoušky)

Důkaz1:

$$\text{len}(xs ++ ys) = \text{len } xs + \text{len } ys$$

$$\text{len } [] = 0 \quad (1)$$

$$\text{len } (_ : xs) = 1 + \text{len } xs \quad (2)$$

$$[] ++ ys = ys \quad (3)$$

$$(x:xs) ++ ys = x:(xs ++ ys) \quad (4)$$

1A: Dokaz: $\text{length}(xs ++ ys) = \text{length } xs + \text{length } ys$ ← dale $\text{length} = \text{len}$!!!
předpoklad

$\text{len } [] = 0$ (1) $[] ++ ys = ys$ (3)
 $\text{len}(x:xs) = 1 + \text{len } xs$ (2) $(x:xs) ++ ys = x:(xs ++ ys)$ (4)

xs - indukční proměnná

1) $xs = []$
 $\text{len}([] ++ ys)$
 $\text{len } ys = P$ ✓

$\text{len}[] + \text{len } ys = 0 + \text{len } ys = \text{len } ys$ ← shodím se dostát
 (3) zprava doleva

2) $xs = (a:as)$
 $\text{length}((a:as) ++ ys) =$
 $\text{len}(a:(as ++ ys)) =$
 $1 + \text{len}(as ++ ys) =$
 $1 + \text{len } as + \text{len } ys =$
 $\text{len}(a:as) + \text{len } ys$ ✓

// (4) zleva doprava
 // (2) zleva doprava
 // předpoklad zleva doprava
 // (2) zprava doleva

<https://fituska.eu/download/file.php?id=11612>

Dukaz2:

$\text{map } f (xs ++ ys) = \text{map } f xs ++ \text{map } f ys$

$(++) [] ys = ys$ (1)

$(++) (x:xs) ys = x : (xs ++ ys)$ (2)

$\text{map } f [] = []$ (3)

$\text{map } f (x:xs) = f x : \text{map } f xs$ (4)

Pi. Dokaž - $\text{map } f (xs ++ gs) = \text{map } f xs ++ \text{map } f gs$ - předpoklad

$[] ++ gs = gs$ (1) $\text{map } f [] = []$ (3)

$(x:xs) ++ gs = x:(xs ++ gs)$ (2) $\text{map } f (x:xs) = f x : \text{map } f xs$ (4)

1) $xs = []$

$L = \text{map } f ([] ++ gs) =$

$\text{map } f gs = P$ ✓

$P = \text{map } f [] ++ \text{map } f gs =$ // (1) zleva doprava

$[] ++ \text{map } f gs =$ // 3 zleva doprava

$\text{map } f gs = L$ ✓ // 1 zleva doprava

2) $xs = (a:as)$ $P = \text{map } f (a:as) ++ \text{map } f gs$

$\text{map } f ((a:as) ++ gs) =$ // (2) zleva doprava

$\text{map } f (a:(as ++ gs)) =$ // (4) zleva doprava

$f a : \text{map } f (as ++ gs) =$ // předpoklad

$f a : \text{map } f as ++ \text{map } f gs =$ // (4) zprava doleva

$\text{map } f (a:as) ++ \text{map } f gs = P$ ✓

4

<https://fituska.eu/download/file.php?id=11651>

Dukaz3:

$\text{rev } xs = \text{reverse } xs$

$$\text{rev } [] = [] \quad (1)$$

$$\text{rev } (x:xs) = \text{rev } xs ++ [x] \quad (2)$$

$$\text{reverse } xs = \text{rev}' xs [] \quad (3)$$

$$\text{rev}' [] ys = ys \quad (4)$$

$$\text{rev}' (x:xs) ys = \text{rev}' xs (x:ys) \quad (5)$$

```

1) xs = []
L = rev xs
= rev []           // 1
= []
P = reverse xs
= reverse []       // 3
= rev' [] []       // 4
= []
L == P

```

```

2) xs = (a:as)
IP : rev as = reverse as

```

```

L = rev xs
= rev (a:as)       // 2
= rev as ++ [a]    // IP
= reverse as ++ [a] // 3
= rev' as [] ++ [a] //
... magic          // to jde pouzivat ++ kdyz neni v definicich? navic i kdyby jo, tak se mi tohle nedari z
definice odvodit. ale lepsi reseni nemam.
= rev' as [a]
= rev' as (a:[])
= rev' (a:as) []
= reverse (a:as)
= reverse xs = P

```

Důkaz 4:

```

(IP) foldr (++) [] xs = ccat xs
(1) foldr f zs [] = zs
(2) foldr f zs (xs:xss) = f xs (foldr f zs xss)
(3) ccat [] = []
(4) ccat (xs:xss) = (++) xs (ccat xss)

```

```

1) foldr (++) [] [] = ccat []
   foldr (++) [] [] = [] -- 1 →
   [] = ccat [] -- 3 ←

```

```

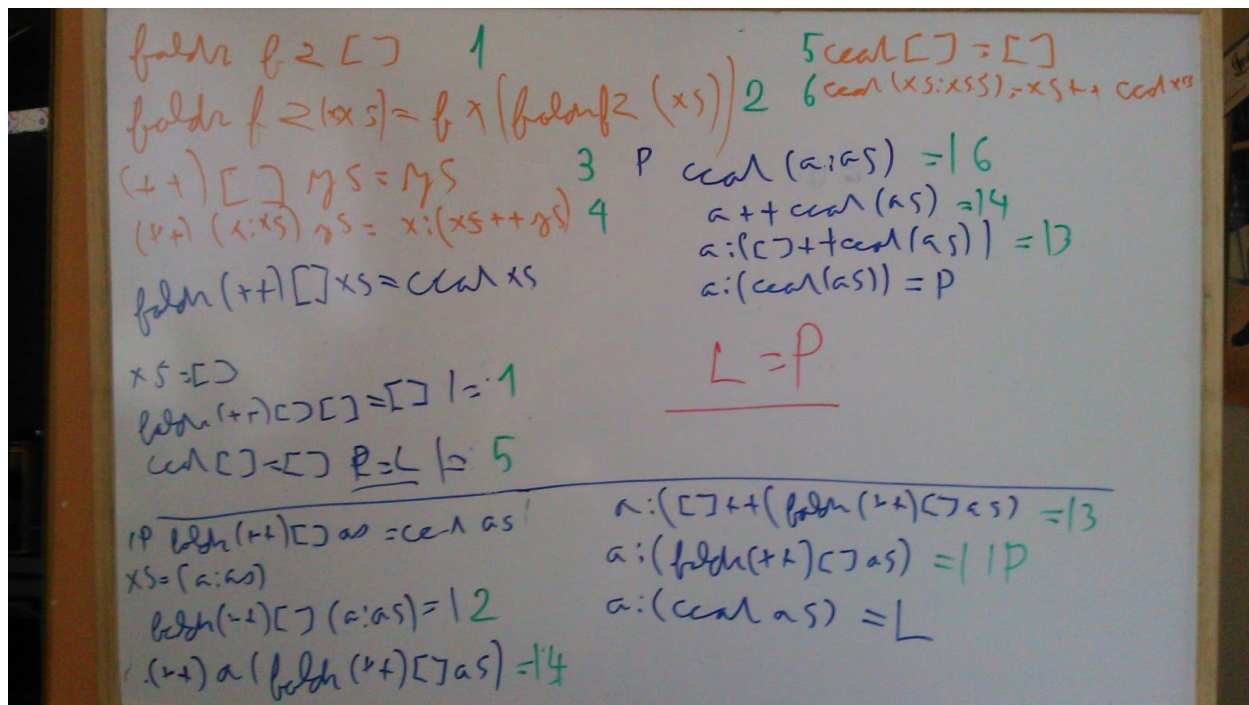
2) foldr (++) [] (zs:zss) = ccat (zs:zss) :
   foldr (++) [] (zs:zss) = (++) zs (foldr (++) [] zss) -- 2 →
   (++) zs (foldr (++) [] zss) = (++) zs (ccat zss) -- IP →

```

$(++) \text{ zs } (\text{ccat zss}) = \text{ccat } (\text{zs:zss}) \text{ -- 4 } \leftarrow$

Q.E.D.

Nejsem si jistej, ale myslíte, že to může být takto: - toto je blbe! díky za info ty hňupe, rozvést bys to nechtěl?



Důkaz 5 - $\text{take } n \text{ xs } ++ \text{drop } n \text{ xs} = \text{xs}$

$\text{take} :: \text{Int} \rightarrow [a] \rightarrow [a]$

$\text{take } n _ \quad | \ n \leq 0 = [] \quad (1.)$

$\text{take } _ [] = [] \quad (2.)$

$\text{take } n (x:xs) = x : \text{take } (n-1) \text{ xs} \quad (3.)$

$\text{drop} :: \text{Int} \rightarrow [a] \rightarrow [a]$

$\text{drop } n \text{ xs} \quad | \ n \leq 0 = \text{xs} \quad (4.)$

$\text{drop } _ [] = [] \quad (5.)$

$\text{drop } n (x:xs) = \text{drop } (n-1) \text{ xs} \quad (6.)$

$(++) :: [a] \rightarrow [a] \rightarrow [a]$

$[] ++ \text{ys} = \text{ys} \quad (7.)$

$(x:xs) ++ ys = x : (xs ++ ys)$ (8.)

Pro $xs = []$:

dokazují: $take\ n\ [] ++ drop\ n\ [] = []$

```
L = take n [] ++ drop n []
  = [] ++ drop n []           // použito 2. zleva doprava
  = [] ++ []                  // použito 5. zleva doprava
  = []                        // použito 7. zleva doprava
L = P
```

Předpoklad:

$take\ n\ as ++ drop\ n\ as = as$

Pro $xs = (a:as)$:

dokazují: $take\ n\ (a:as) ++ drop\ n\ (a:as) = (a:as)$

```
L = take n (a:as) ++ drop n (a:as)
  = a:(take (n-1) as) ++ drop n (a:as) // použito 3. zleva doprava
  = a:(take (n-1) as ++ drop n (a:as)) // použito 8. zleva doprava
  = a:(take (n-1) as ++ drop (n-1) as) // použito 6. zleva doprava
  = (a:as)                            // použít předpoklad
L = P
```

===== **BONUS** =====

5. (Bonusová otázka za +10 bodů.) Co jsou to kontinuity a jaké je jejich využití při definici denotační sémantiky jazyka Haskell. Demonstrujte na příkladu.

Význam „zbytku programu“ od aktuálního místa v programu až po jeho ukončení lze modelovat kontinuitou - funkcí, která na základě aktuálního stavu vrátí výsledek celého programu.

Kontinuity v denotační sémantice - Sémantická funkce obdrží jednu nebo více kontinuit, kterým po vyhodnocení může (a nemusí) předat řízení.

Kontinuity výrazu - Obdrží hodnotu výrazu, stav programu po jeho výpočtu a vrátí výsledek programu

```
type ECont = Int -> State -> Output
```

Kontinuace příkazu - Obdrží stav po provedení příkazu a vrátí výsledek programu

```
type CCont = State -> Output
```

5. Bonusový příklad v Haskellu napsat funkci `isPrv`, která bere jako argument číslo a vrací `True`, ak je to prvočíslo.

2 body, při použití "memoization" za přibližně 7 bodů, při použití "memoization" a zároveň 3 efektivně řádky za 10 bodů

za 2 body:

```
isPrime :: Int -> Bool
```

```
isPrime x
```

```
    | x <= 0 = False
```

```
    | otherwise = ((length [del | del <- [2..(x-1)], (x `mod` del) == 0]) == 0)
```