

Haskell

alebo čo sa stane, keď to dizajnéri jazykov preháňajú s hubami

- 1. Čo to je
- 2. Ako to vyzerá
- 3. Čím sa to líši
- 4. Ako to funguje
- 5. Aké to má výhody
- 6. Aké to má nevýhody

1. Čo to je

1. Čo to je

- *programovací jazyk*
- high-level
- general-purpose, domain-specific
- kompilovaný, interpretovaný
- funkcionálny
- lazy (non-strict)
- staticky a silno typovaný

2. Ako to vyzerá

2. Ako to vyzerá

obligátny hello world:

```
main = putStrLn „Hello world!“
```

2. Ako to vyzerá

- obvykle sa však nesústredujeme na celé programy
- zaujímajú nás **funkcie**
- funkcia vykonáva jednu úlohu
 - parsuje zadaný text
 - počíta farbu pixelu
 - filtruje zákazníkov

2. Ako to vyzerá

- filtruje zákazníkov

```
data Customer = Customer {  
    name :: String, payment :: Double  
}
```

```
badCustomers :: [Customer] -> [Customer]  
badCustomers cs = filter ((avg >) . payment) cs  
  where  
    avg = (sum $ map payment cs) / length cs
```

2. Ako to vyzerá

- parsuje zadaný text

```
openTag :: Parser Tag
```

```
openTag = do
```

```
    name <- try (char '<' >> ident)
```

```
    attrs <- many attrPair
```

```
    slash <- optionMaybe (char '/')
```

```
    char '>'
```

```
    case slash of
```

```
        Nothing -> return $ OpenTag name attrs
```

```
        Just _   -> return $ ShortTag name attrs
```

2. Ako to vyzerá

- počíta riadky v súbore

```
main = do
  [file] <- getArgs
  count <- length . lines <$> readFile file
  putStrLn $ „Počet riadkov: “ ++ show count
```

2. Ako to vyzerá

- počíta fibonacciho čísla

```
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

-- (Nápoveda)

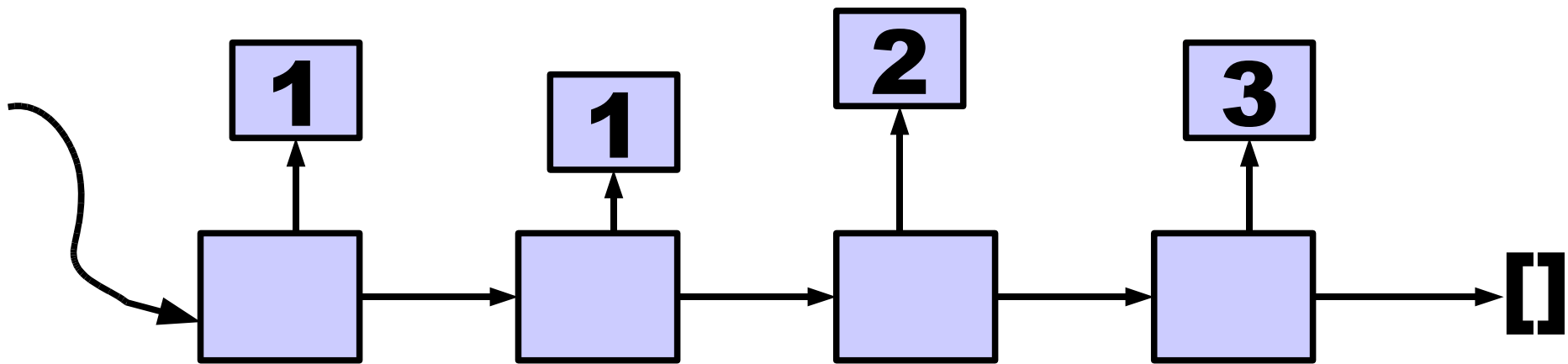
```
zipWith (+) [1, 1, 2] [2, 8, 6] = [1+1, 1+8, 2+6]
```

```
tail [1, 2, 3, 4, 5] = [2, 3, 4, 5]
```

```
1 : [2, 3, 4, 5] = [1, 2, 3, 4, 5]
```

Zoznamy

- Zoznam typu a sa značí $[a]$
- Zoznam Int-ov sa teda značí $[Int]$
- **data** $[a] = [] \mid a : [a]$
- $1 : 1 : 2 : 3 : []$ skrátene $[1, 1, 2, 3]$
- $1 : (1 : (2 : (3 : [])))$



Syntaktický cukor

- pri aplikácii funkcie sa parametre oddeľujú medzerami:

```
asciicode = map ord „password“
```

- prefix fcia sa môže stať infixovou

```
asciicode = ord `map` „password“  
result    = 255 `div` 13
```

- infix op sa môže stať prefixovým

```
sum      = (+) 3 5  
vecSum  = zipWith (+) [1,2,3] [4,5,6]
```

3. Čím sa to líši

3. Čím sa to líši

- dôraz na **hodnotu**, nie dej/činnosť
- veci sa dejú inak, než sú napísané

```
int fib(int n)
{
    int a = 1, b = 1;
    for (int i = 0; i < n; ++i)
    {
        int tmp = b;
        b = a + b;
        a = tmp;
    }
    return b;
}
```

```
-- n-té Fibonacciho číslo
fib n = snd (fibPair n)
```

```
-- n-tá Fibonacciho dvojica
fibPair 0 = (1,1)
fibPair n = (b,a+b)
    where
        (a,b) = fibPair (n-1)
```

3. Čím sa to líši

- **premenné sú nemenné**
 - orientácia na hodnoty
 - premenná je názov pre hodnotu
 - matematický význam slova
- funkcie svoje výsledky nemôžu nikam zapísať
- všetko sa deje vracaním hodnôt

3. Čím sa to líši

- funkcie sú tiež hodnotami
 - môžu sa odovzdávať v parametroch funkcií

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

```
map f [] = []
```

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

```
map (+1) [1, 2, 3, 4] = [2, 3, 4, 5]
```

```
map (\x -> x + 2^x) [3, 4, 5] = [11, 20, 37]
```

3. Čím sa to líši

- Haskell je **lazy**

```
ones = 1 : ones
```

```
-- ones = [1, 1, 1, 1, 1, 1, 1, 1, 1, ... ]
```

```
head (x:xs) = x
```

```
tail (x:xs) = xs
```

```
head ones = ?
```

3. Čím sa to líši

- Haskell je **lazy**

```
ones = 1 : ones
```

```
-- ones = [1, 1, 1, 1, 1, 1, 1, 1, 1, ... ]
```

```
head (x:xs) = x
```

```
tail (x:xs) = xs
```

```
head ones = 1
```

3. Čím sa to líši

- algebraické dátové typy
(čosi ako union struct-ov z C)

```
type Name      = String
type AttrPair  = (String, String)
data Tag =
    OpeningTag Name [AttrPair] -- <tag>
  | ShortTag    Name [AttrPair] -- <tag />
  | ClosingTag Name         -- </tag>

imageTag :: Tag
imageTag = ShortTag „img“ [(„src“, „file.png“)]
```

3. Čím sa to líši

- pattern matching

```
type Name      = String
type AttrPair  = (String, String)
data Tag =
    OpeningTag Name [AttrPair]
  | ShortTag   Name [AttrPair]
  | ClosingTag Name
```

```
getName (OpeningTag name _) = name
getName (ShortTag   name _) = name
getName (ClosingTag name)   = name
```

3. Čím sa to líši

- pattern matching II.

```
data Expr =  
    Plus Expr Expr  
| Minus Expr Expr  
| Mul Expr Expr  
| Div Expr Expr  
| Num Integer
```

```
eval :: Expr -> Integer  
eval (Plus p q) = eval p + eval q  
eval (Minus p q) = eval p - eval q  
eval (Mul p q) = eval p * eval q  
eval (Div p q) = eval p `div` eval q
```

```
> eval (Plus (Mul (Num 3) (Num 5)) (Num 27)) -- bude 42
```

3. Čím sa to líši

- pattern matching II.

```
data Expr =  
    Expr :+: Expr  
| Expr :- Expr  
| Expr :* Expr  
| Expr :/ Expr  
| Num Integer  
  
eval :: Expr -> Integer  
eval (p :+: q) = eval p + eval q  
eval (p :- q) = eval p - eval q  
eval (p :* q) = eval p * eval q  
eval (p :/ q) = eval p `div` eval q
```

```
> eval ((Num 3 :* Num 5) :+: Num 27) -- bude 42
```

3. Čím sa to líši

- typové triedy
 - **nie sú to triedy v OOP zmysle**
 - nerobí sa ad-hoc preťažovanie
 - operácie patria triedam
 - sú to skôr **interfaces** à la Java
 - (OOP sa v Haskellu nerobí)

3. Čím sa to líši

- typové triedy

```
class Group a where
  ( $\otimes$ ) :: a -> a -> a
  inv  :: a -> a
  neutr :: a
```

```
instance Group Integer where
  a  $\otimes$  b = a + b
  inv a = -a
  neutr = 0
```

3. Čím sa to líši

- typové triedy

```
data Node = Nil | Internal Int Node Node
```

```
class Ord a where  
  (<) :: a -> a -> Bool
```

```
instance Ord Node where  
  (Internal left _ _) < (Internal right _ _)  
    = left < right  
  Nil < (Internal _ _ _) = True  
  (Internal _ _ _) < Nil = False  
  Nil < Nil = False
```

4. Ako to funguje

4. Ako to funguje

- v Haskellu sa zapisujú vzťahy, nie postupy
- je na jazyku, aby vzťahy vyriešil, aj keď mu nepoviem, ako má postupovať
- dokáže vyriešiť občas zaujímavé závislosti
- „čo je napísané, nie je pravda“

4. Ako to funguje

```
isPrefixOf :: Eq a => [a] -> [a] -> Bool
[]      `isPrefixOf` _      = True
(x:xs)  `isPrefixOf` []    = False
(x:xs)  `isPrefixOf` (y:ys) =
    (x == y)
    &&
    (xs `isPrefixOf` ys)
-- „santa“ `isPrefixOf` „santa claus“ = True
-- „claus“ `isPrefixOf` „santa claus“ = False
```

```
any :: (a -> Bool) -> [a] -> Bool
any = foldr ((||) . f) False
-- any (> 3) [1, 3, 2] = False
-- any (> 3) [1, 7, 2] = True
```

4. Ako to funguje

`contains :: Eq a => [a] -> [a] -> Bool`

```
haystack `contains` needle =  
  (s `isPrefixOf`) `any` (tails haystack)
```

```
-- f a b = a `f` b
```

```
-- „santa“ `isPrefixOf` „santa claus“ = True
```

```
-- „claus“ `isPrefixOf` „santa claus“ = False
```

```
-- any (> 3) [1, 3, 2] = False
```

```
-- any (> 3) [1, 7, 2] = True
```

```
-- tails [1,2,3] = [[1,2,3], [2,3], [3], []]
```

4. Ako to funguje

```
contains :: Eq a => [a] -> [a] -> Bool
```

```
haystack `contains` needle =  
    (s `isPrefixOf`) `any` (tails haystack)
```

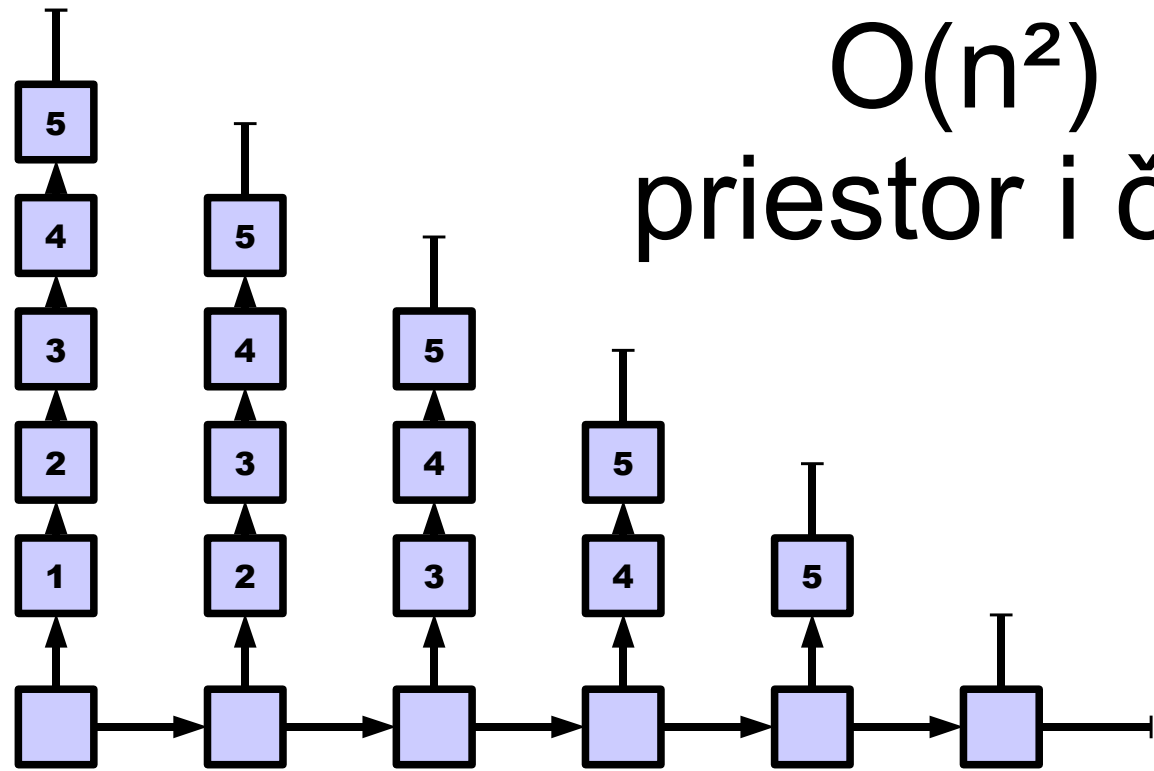
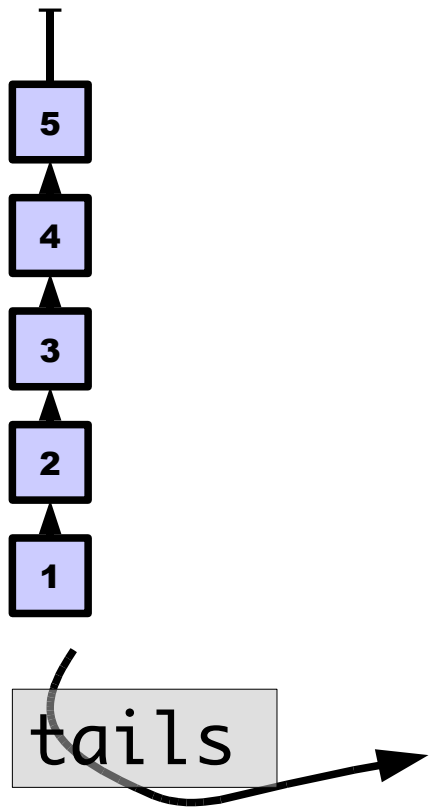
- ako dlho pobeží tento kód?

```
haystack = replicate 1000000 'a' -- milión a-čok  
needle   = „aaab“
```

```
main = if haystack `contains` needle  
      then putStrLn „Obsahuje.“  
      else putStrLn „Neobsahuje.“
```

4. Ako to funguje

```
tails :: [a] -> [[a]]  
tails []    = []  
tails (x:xs) = (x:xs) : tails xs
```

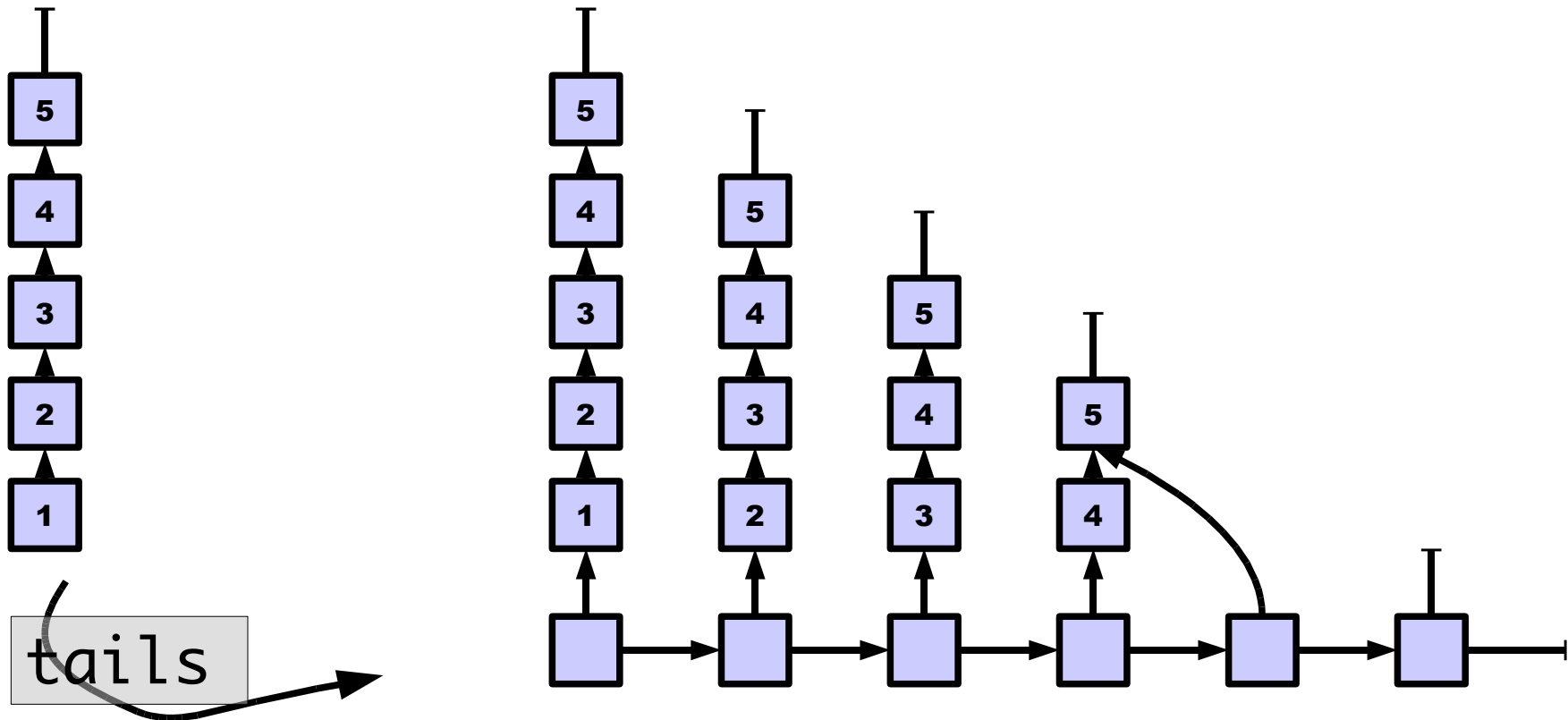


4. Ako to funguje

`tails :: [a] -> [[a]]`

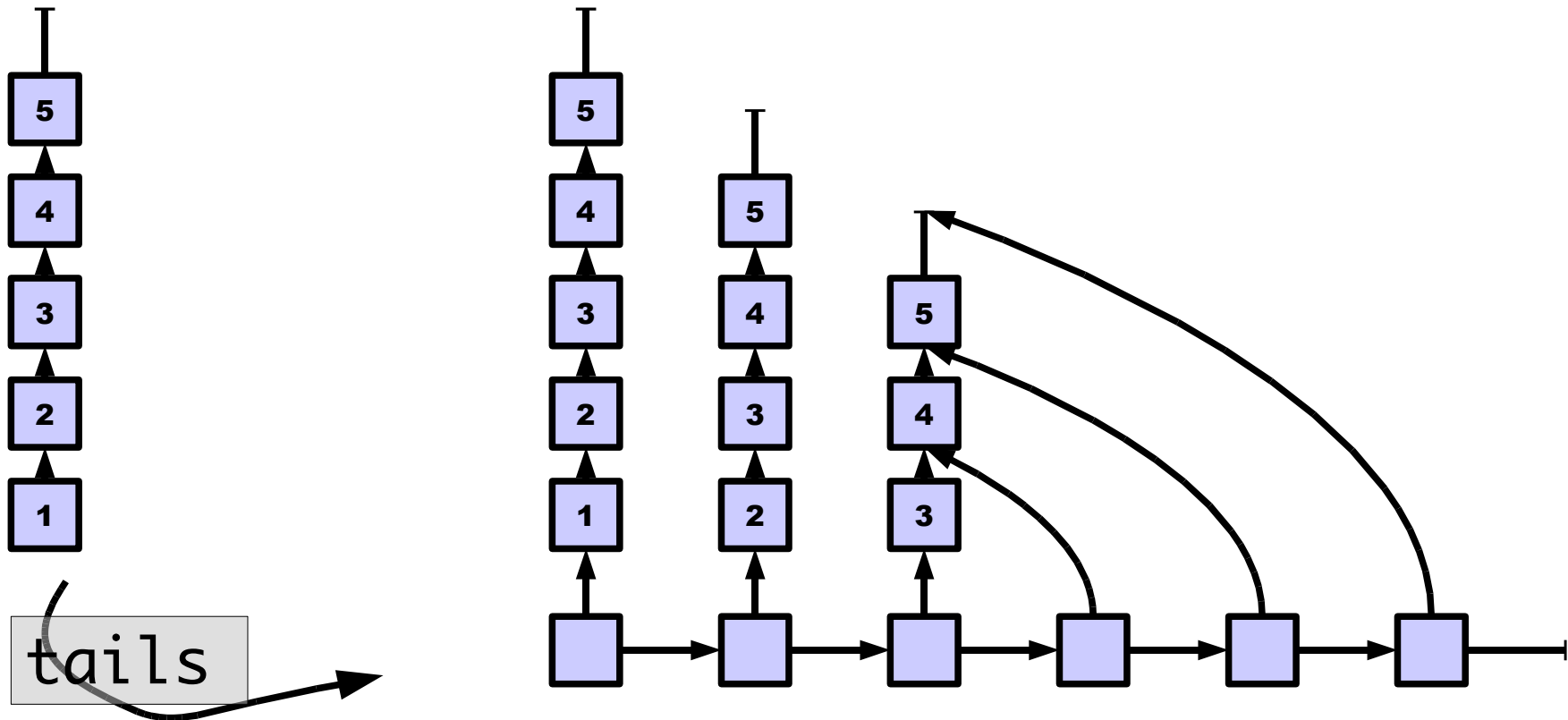
`tails [] = [[]]`

`tails (x:xs) = (x:xs) : tails xs`



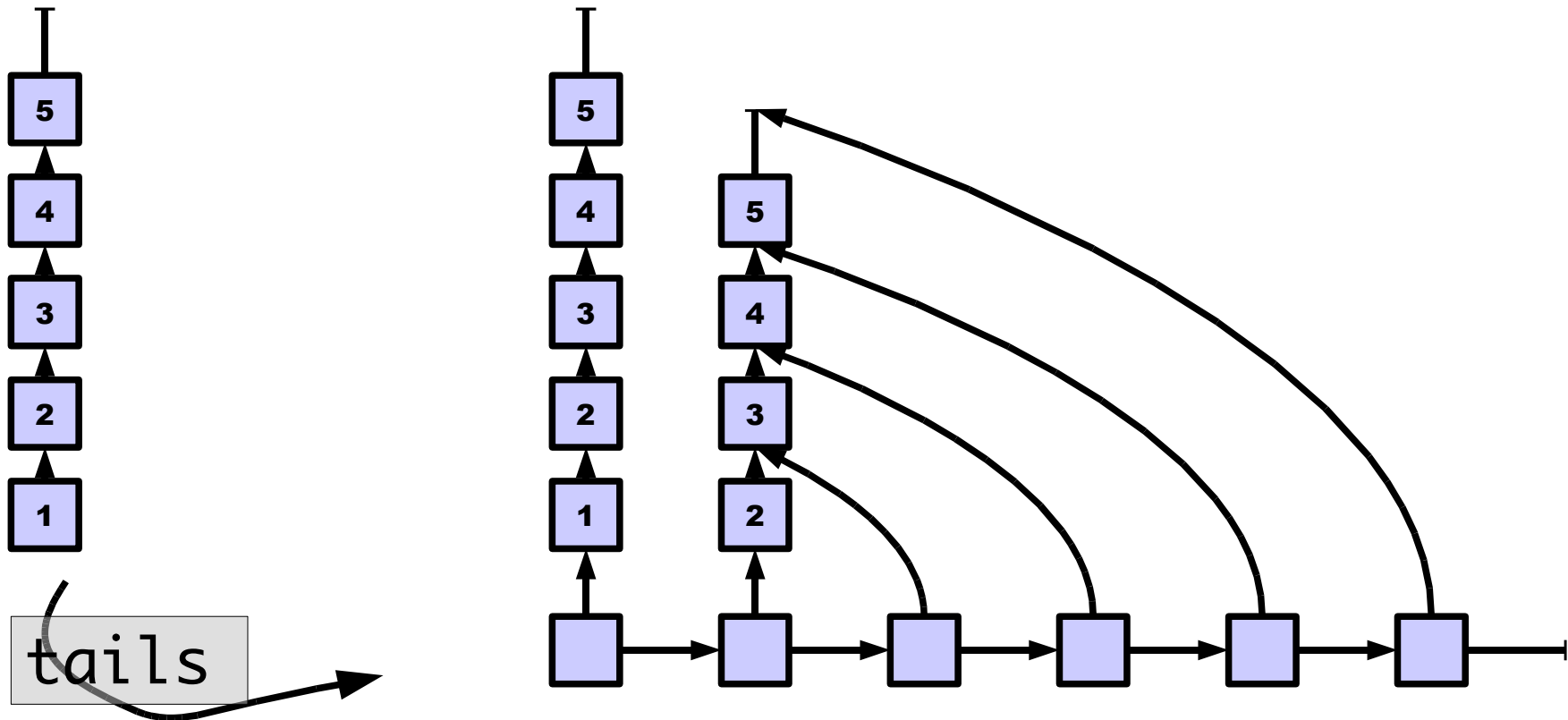
4. Ako to funguje

```
tails :: [a] -> [[a]]  
tails []    = []  
tails (x:xs) = (x:xs) : tails xs
```



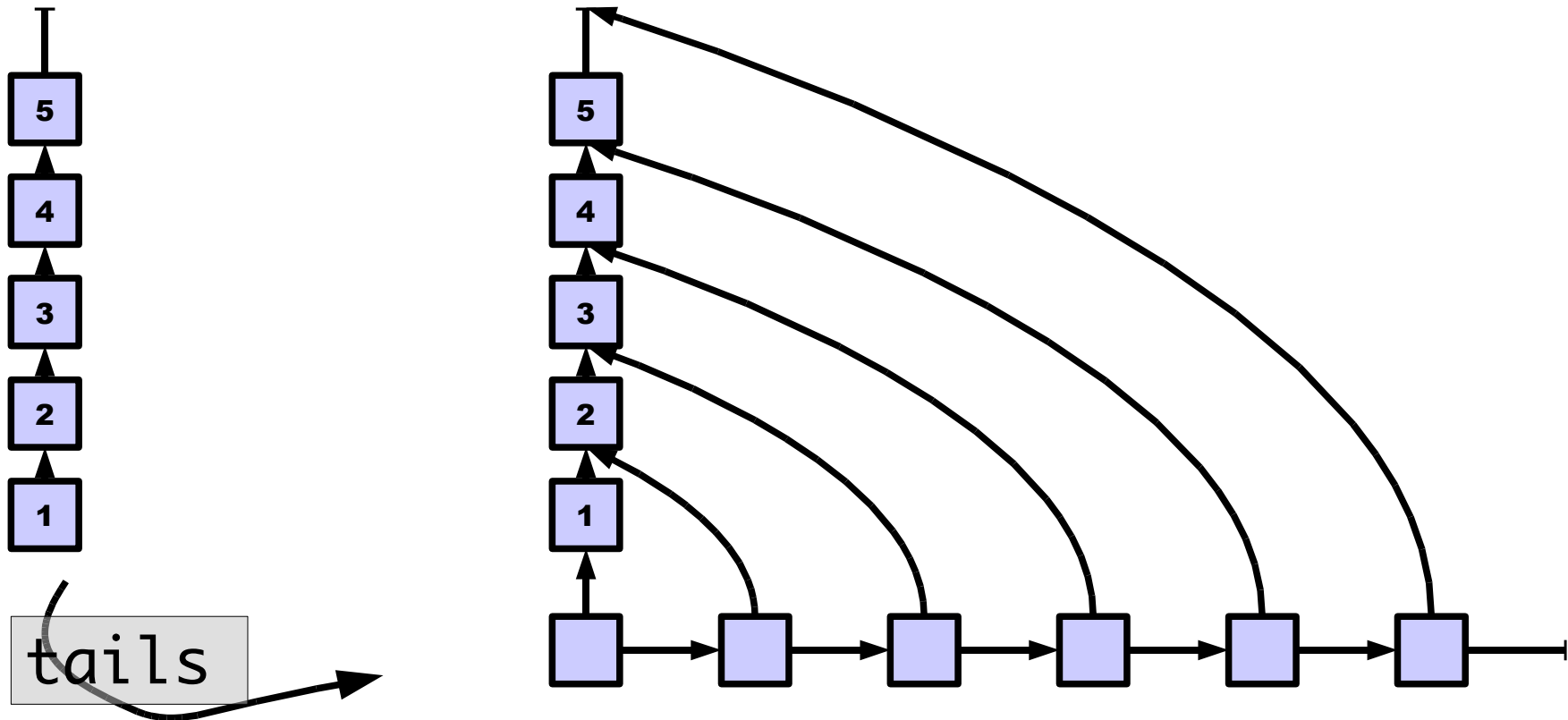
4. Ako to funguje

```
tails :: [a] -> [[a]]  
tails []    = []  
tails (x:xs) = (x:xs) : tails xs
```



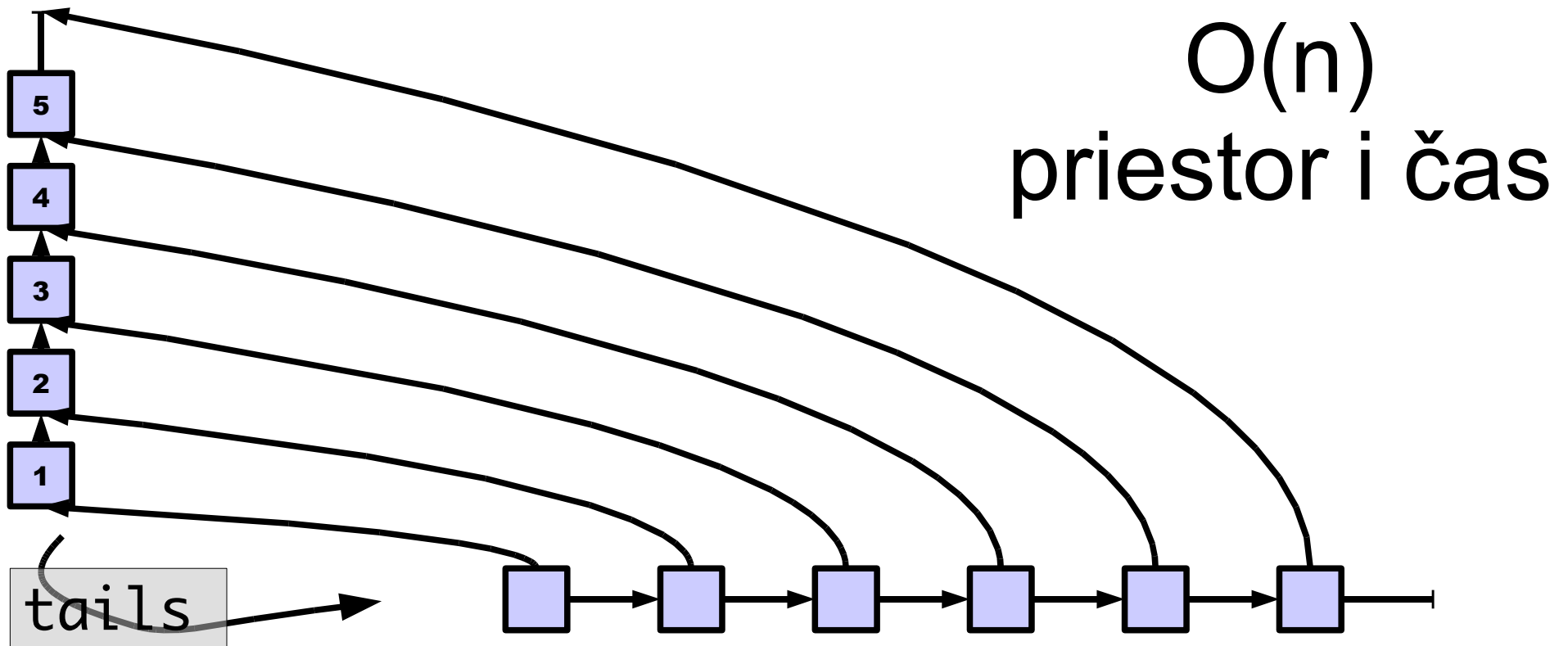
4. Ako to funguje

```
tails :: [a] -> [[a]]  
tails []    = [[]]  
tails (x:xs) = (x:xs) : tails xs
```



4. Ako to funguje

```
tails :: [a] -> [[a]]  
tails []    = [[]]  
tails (x:xs) = (x:xs) : tails xs
```



4. Ako to funguje

Dostávame teda kód ekvivalentný klasickému `strstr()` z C:

```
haystack `contains` needle = (s `isPrefixOf`) `any` (tails haystack)
```

```
bool strstr(char *haystack, char *needle)
{
    len = strlen(needle);
    while (*haystack)
        if (memcmp(haystack++, needle, len) == 0)
            return TRUE;
    return FALSE;
}
```

4. Ako to funguje

- Haskell zvláda nekonečné štruktúry
 - ako do počítača zadám niečo nekonečné?

4. Ako to funguje

- Haskell zvláda nekonečné štruktúry

```
leets :: [Integer]
leets = 1337 : leets    -- [1337, 1337, 1337, ... ]
```

```
naturals :: [Integer]
naturals = [1..]       -- [1, 2, 3, 4, 5, ... ]
evens    = [2,4..]     -- [2, 4, 6, 8, 10, ... ]
```

```
nats :: [Integer]
nats = 1:map (+1) nats  -- [1, 2, 3, 4, 5, ... ]
```

```
fibs :: [Integer]
fibs = 1:1:zipWith (+) fibs (tail fibs)
```

4. Ako to funguje

```
nats :: [Integer]
```

```
nats = 1:map (+1) nats -- [1, 2, 3, 4, 5, ... ]
```

```
nats = [1, 2, 3, 4, 5, 6, ... ]
```

```
nats = 1 : (2 : (3 : (5 : (6 : .....))))
```

4. Ako to funguje

```
nats :: [Integer]
```

```
nats = 1:map (+1) nats -- [1, 2, 3, 4, 5, ... ]
```

```
nats = [1, 2, 3, 4, 5, 6, ... ]
```

```
nats = 1 : (2 : (3 : (5 : (6 : .....))))
```

```
1 : [2, 3, 4, 5, 6, 7, ... ]
```

```
1 : map (+1) [1, 2, 3, 4, 5, 6, ...]
```

```
1 : map (+1) nats
```

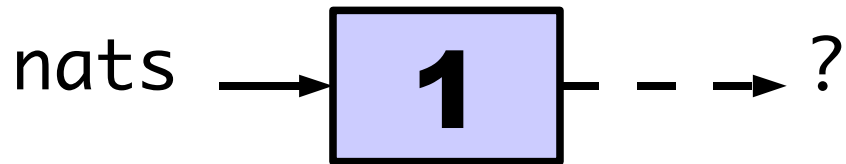
4. Ako to funguje

```
nats :: [Integer]
nats = 1:map (+1) nats -- [1, 2, 3, 4, 5, ... ]
```

nats - - →

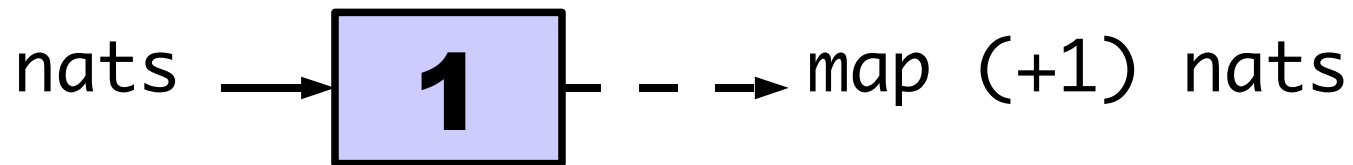
4. Ako to funguje

```
nats :: [Integer]
nats = 1:map (+1) nats -- [1, 2, 3, 4, 5, ... ]
```



4. Ako to funguje

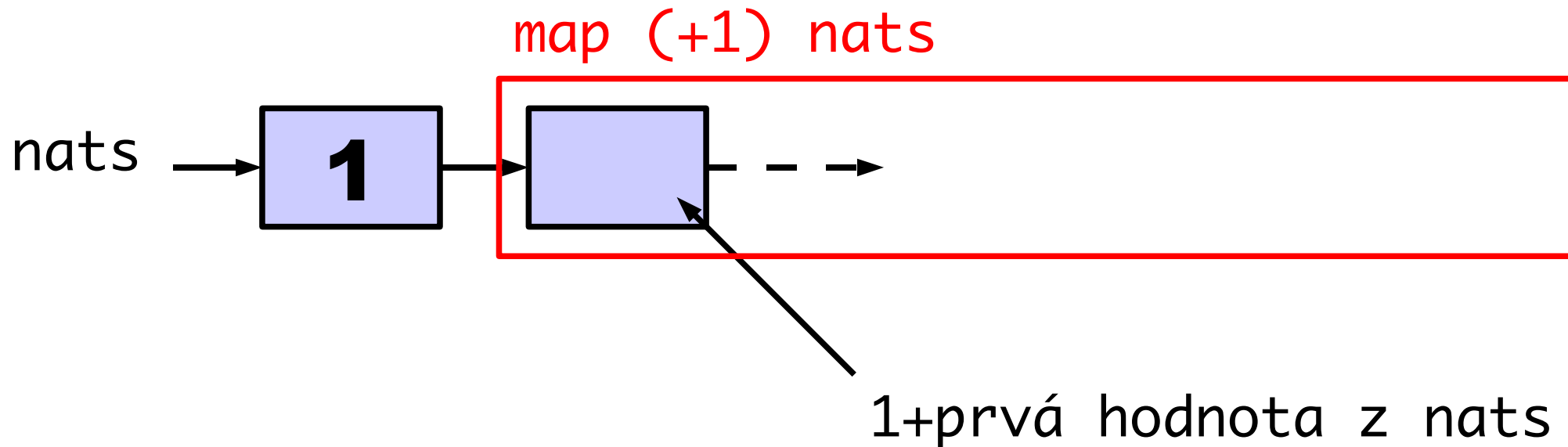
```
nats :: [Integer]
nats = 1:map (+1) nats -- [1, 2, 3, 4, 5, ... ]
```



4. Ako to funguje

```
nats :: [Integer]
```

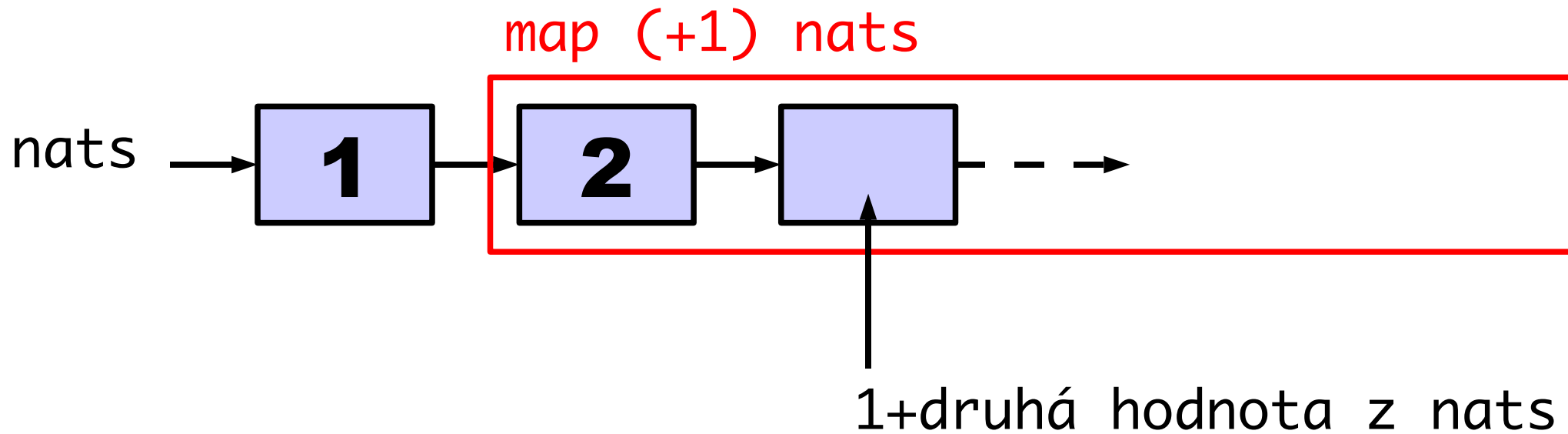
```
nats = 1:map (+1) nats -- [1, 2, 3, 4, 5, ... ]
```



4. Ako to funguje

```
nats :: [Integer]
```

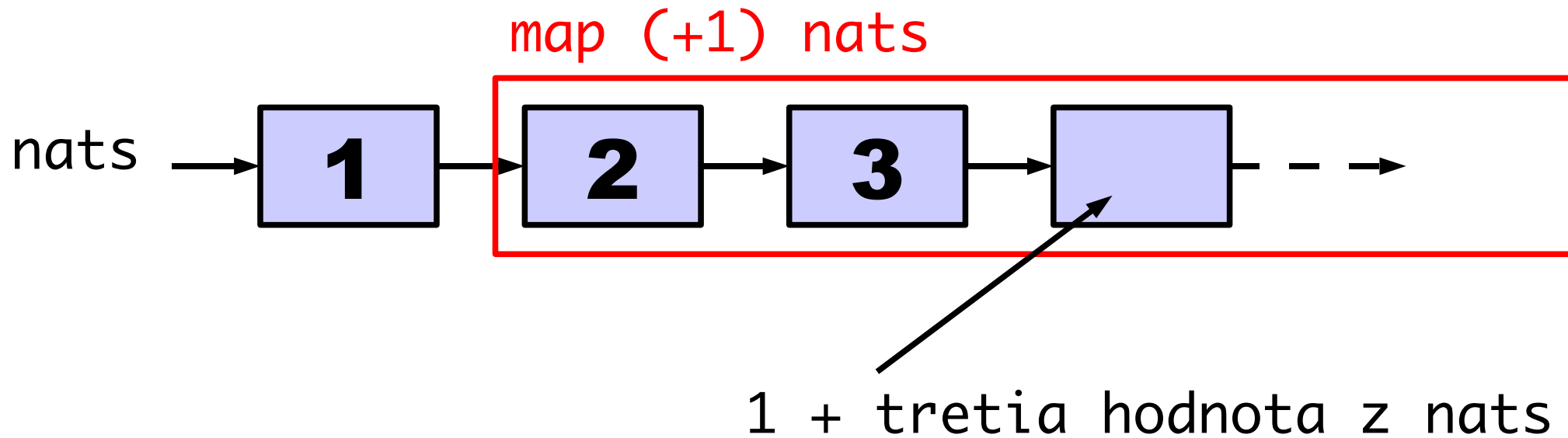
```
nats = 1:map (+1) nats -- [1, 2, 3, 4, 5, ... ]
```



4. Ako to funguje

```
nats :: [Integer]
```

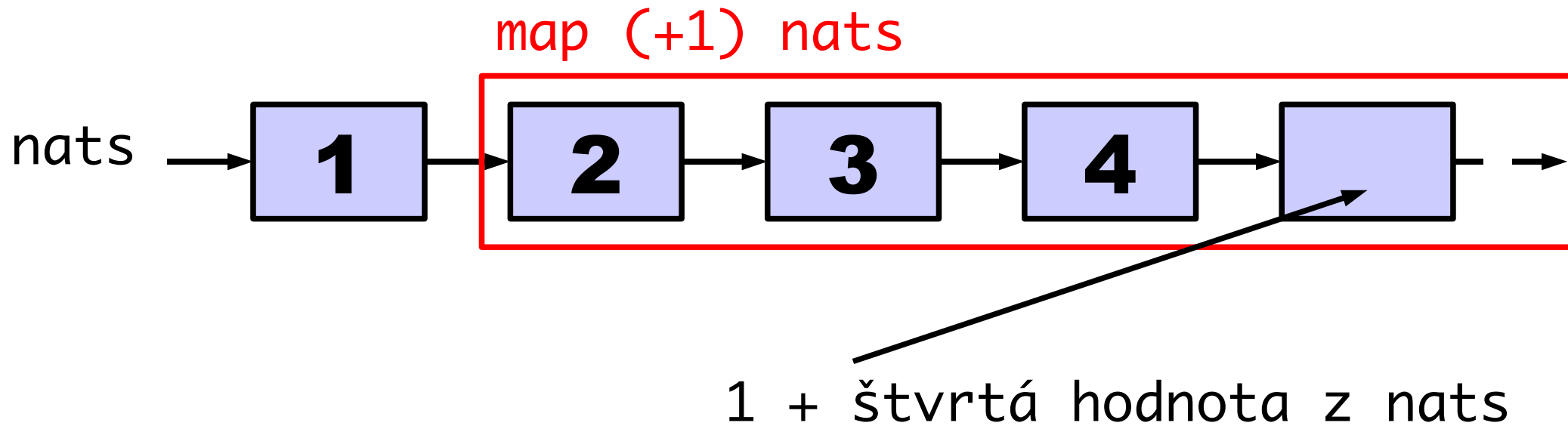
```
nats = 1:map (+1) nats -- [1, 2, 3, 4, 5, ... ]
```



4. Ako to funguje

```
nats :: [Integer]
```

```
nats = 1:map (+1) nats -- [1, 2, 3, 4, 5, ... ]
```



4. Ako to funguje

- Postupne sa počíta celý zoznam
- Vo väčšine prípadov sú algoritmy sekvenčné
- Garbage collector vtedy užiera zo začiatku zoznamu nepotrebné prvky
- Spotreba pamäte nekonečného zoznamu: $O(1)$ -- obvykle pár uzlov

4. Ako to funguje

```
fibs :: [Integer]
fibs = 1:1:zipWith (+) fibs (tail fibs)
```

`1:1:zipWith (+)`



fibs

1	1	2	3	5	8	13
----------	----------	----------	----------	----------	----------	-----------

tail fibs

1	2	3	5	8	13	21
----------	----------	----------	----------	----------	-----------	-----------

1	1	2	3	5	8	13	21	34
----------	----------	----------	----------	----------	----------	-----------	-----------	-----------

4. Ako to funguje

```
fibs :: [Integer]  
fibs = 1:1:zipWith (+) fibs (tail fibs)
```

1:1:zipWith (+)



fibs

1	1							
----------	----------	--	--	--	--	--	--	--

tail fibs

1								
----------	--	--	--	--	--	--	--	--

1	1							
----------	----------	--	--	--	--	--	--	--

4. Ako to funguje

```
fibs :: [Integer]  
fibs = 1:1:zipWith (+) fibs (tail fibs)
```

1:1:zipWith (+)



fibs

1	1	2					
1	2						

tail fibs

1	1	2						
----------	----------	----------	--	--	--	--	--	--

4. Ako to funguje

```
fibs :: [Integer]  
fibs = 1:1:zipWith (+) fibs (tail fibs)
```

1:1:zipWith (+)



fibs

1	1	2	3					
----------	----------	----------	----------	--	--	--	--	--

tail fibs

1	2	3						
----------	----------	----------	--	--	--	--	--	--

1	1	2	3					
----------	----------	----------	----------	--	--	--	--	--

4. Ako to funguje

```
fibs :: [Integer]  
fibs = 1:1:zipWith (+) fibs (tail fibs)
```

1:1:zipWith (+)



fibs

1	1	2	3	5				
----------	----------	----------	----------	----------	--	--	--	--

tail fibs

1	2	3	5					
----------	----------	----------	----------	--	--	--	--	--

1	1	2	3	5				
----------	----------	----------	----------	----------	--	--	--	--

4. Ako to funguje

```
fibs :: [Integer]  
fibs = 1:1:zipWith (+) fibs (tail fibs)
```

1:1:zipWith (+)



fibs

1	1	2	3	5	8			
----------	----------	----------	----------	----------	----------	--	--	--

tail fibs

1	2	3	5	8				
----------	----------	----------	----------	----------	--	--	--	--

1	1	2	3	5	8			
----------	----------	----------	----------	----------	----------	--	--	--

4. Ako to funguje

```
fibs :: [Integer]  
fibs = 1:1:zipWith (+) fibs (tail fibs)
```

1:1:zipWith (+)



fibs

1	1	2	3	5	8	13
----------	----------	----------	----------	----------	----------	-----------

tail fibs

1	2	3	5	8	13	
----------	----------	----------	----------	----------	-----------	--

1	1	2	3	5	8	13		
----------	----------	----------	----------	----------	----------	-----------	--	--

4. Ako to funguje

```
fibs :: [Integer]
fibs = 1:1:zipWith (+) fibs (tail fibs)
```

`1:1:zipWith (+)`



fibs

1	1	2	3	5	8	13
----------	----------	----------	----------	----------	----------	-----------

tail fibs

1	2	3	5	8	13	21
----------	----------	----------	----------	----------	-----------	-----------

1	1	2	3	5	8	13	21	
----------	----------	----------	----------	----------	----------	-----------	-----------	--

4. Ako to funguje

```
fibs :: [Integer]
fibs = 1:1:zipWith (+) fibs (tail fibs)
```

`1:1:zipWith (+)`



fibs

1	1	2	3	5	8	13
----------	----------	----------	----------	----------	----------	-----------

tail fibs

1	2	3	5	8	13	21
----------	----------	----------	----------	----------	-----------	-----------

1	1	2	3	5	8	13	21	34
----------	----------	----------	----------	----------	----------	-----------	-----------	-----------

4. Ako to funguje

```
haystack `contains` needle =  
  (s `isPrefixOf`) `any` (tails haystack)
```

```
haystack = [1..]  
needle   = [1337, 1338, 1339]
```

```
haystack `contains` needle = True
```

5. Aké to má výhody

5. Aké to má výhody

- Abstrakcia
 - Programátor sa nezaťažuje
 - Memory managementom
 - Konkrétnymi postupmi
 - Kód je názornejší, vecnejší
 - Miesto deklarácií premenných a manipulovania s pamäťou vypisuje vzťahy

5. Aké to má výhody

- Stručnosť
 - Rovnaký program v Haskell
býva kratší než ekvivalenty v
iných jazykoch
- Expresívnosť
 - Mocné, univerzálne konštrukcie
(napr. funkcie ako parametre)

5. Aké to má výhody

- Bezpečnosť
 - Behová
 - žiaden buffer overflow
 - žiaden segfault
 - Statická
 - silné typovanie
 - odhaľuje chyby už pri kompilácii

5. Aké to má výhody

- Modularita
 - Namiesto veľkých blokov kódu maličké funkcie
 - Žiaden globálny stav
- Domain-specific Languages
 - napr. Parsec

5. Aké to má výhody

- Paralelizmus
 - Funkcionálne programy sa prirodzene paralelizujú
 - V Haskellu to ide bez prekopávania kódu, hintmi
- Konkurentné programy
 - Lightweight threads
 - STM

5. Aké to má výhody

`fib 0 = 1`

`fib 1 = 1`

`fib n = fib (n-1) + fib (n-2)`

5. Aké to má výhody

```
fib 0 = 1
fib 1 = 1
fib n = p `par` q `pseq` (p + q)
  where
    p = fib (n-1)
    q = fib (n-2)
```

5. Aké to má výhody

```
fib 0 = 1
fib 1 = 1
fib n
  | n < 20    = p + q
  | otherwise = p `par` q `pseq` p + q
where
  p = fib (n-1)
  q = fib (n-2)
```

```
$ ghc --make -O2 -threaded fibs.hs -o fibs
$ ./fibs +RTS -N2
```

5. Aké to má výhody

```
$ ghc --make -O2 fibs.hs -o fibs  
$ time ./fibs
```

```
real 0m12.862s  
user 0m14.005s
```

```
$ ghc --make -O2 -threaded fibs.hs -o fibs  
$ time ./fibs +RTS -N2
```

```
real 0m7.983s  
user 0m14.141s
```

6. Aké to má nevýhody

6. Aké to má nevýhody

- Človek sa učí odznova programovať
- Nepredvídateľnosť spotreby zdrojov
- GC - vyššia spotreba pamäte
- Na quick'n'dirty utility je stále lepší

Perl

- O čosi pomalší než C
- Stringy/ByteStringy

Ďakujem za pozornosť Diskusia Otázky

Centrála:

<http://haskell.org>

Real World Haskell (kniha):

<http://book.realworldhaskell.org/>

A Taste of Haskell (video):

<http://lambda-the-ultimate.org/node/2427>

Learn you a Haskell (tutoriál):

<http://learnyouahaskell.com>