# *Wykład 5*

## *Ewaluacja gorliwa i leniwa*

Strategie ewaluacji

Ewaluacja leniwa w języku Haskell

Ewaluacja gorliwa w języku Haskell: funkcje seq i deepseq

Ewaluacja gorliwa w języku Haskell: gorliwe konstruktory

Ewaluacja leniwa w językach gorliwych

Listy leniwe - OCaml

Moduł Lazy, wyrażenia i wzorce leniwe – OCaml

Przekazywanie argumentów do funkcji

Argumenty pozycyjne, nazwane i domyślne

Przykład: problem ośmiu hetmanów

# *Ewaluacja gorliwa i leniwa*

Najważniejsze strategie ewaluacji (wartościowania, obliczania), stosowane w językach programowania to ewaluacja gorliwa i leniwa.

- *Ewaluacja gorliwa* lub ewaluacja sterowana danymi (ang. eager evaluation, strict evaluation, data-driven evaluation, supply-driven evaluation).
  Jest stosowana najczęściej.

- *Ewaluacja leniwa* lub ewaluacja sterowana popytem (ang. lazy evaluation, non-strict evaluation, demand-driven evaluation).

Stosowana strategia ewaluacji stanowi jedną z najważniejszych charakterystyk języka programowania. Zwykle obok głównej strategii, języki programowania udostępniają mechanizmy, umożliwiające lokalne stosowanie drugiej strategii.

OCaml i Scheme wykorzystują ewaluację gorliwą, natomiast Haskell – leniwą.

Poniżej idea obu strategii zostanie wyjaśniona na przykładzie definiowania zmiennej. Precyzyjna definicja strategii wartościowania wymagałaby użycia pewnych formalizmów, np. rachunku lambda.

# *Definicja zmiennej*

Definicja zmiennej składa się z trzech faz:

(1) deklaracja zmiennej
(2) zdefiniowanie wyrażenia, z wartością którego zmienna ma być związana
(3) ewaluacja wyrażenia i związanie zmiennej z obliczoną wartością

Powyższe fazy mogą być wykonywane jednocześnie lub kolejno.
We wszystkich językach funkcyjnych (1) i (2) **muszą** być wykonane jednocześnie.

- W językach funkcyjnych z ewaluacją gorliwą (OCaml, SML, Scheme, …) wykonywana jest od razu faza (3) i zmienna jest wiązana z wartością.
- W językach funkcyjnych z ewaluacją leniwą (Haskell, …) faza (3) jest wykonywana dopiero wtedy, kiedy wartość zmiennej jest potrzebna.

Analogicznie:

- W językach funkcyjnych z ewaluacją gorliwą argument funkcji jest obliczany zawsze, a do funkcji jest przekazywana jego wartość.
- W językach funkcyjnych z ewaluacją leniwą argument funkcji jest przekazywany do funkcji jako wyrażenie, którego wartość jest obliczana dopiero wtedy, kiedy jest potrzebna.

# *Leniwa ewaluacja koniunkcji i alternatywy*

Większość współczesnych języków programowania, m.in. Scala, OCaml, Java, C++, ewaluuje koniunkcję i alternatywę leniwie.

---------- OCaml

```
# true || failwith "error";;
- : bool = true
# false && failwith "error";;
- : bool = false
```

---------- Haskell

```
Prelude> True || error "error"
True
Prelude> False && error "error"
False
```

# *Struktury nieskończone w języku Haskell*

Haskell jest językiem funkcyjnym z wartościowaniem leniwym. Struktury nieskończone otrzymujemy w nim „za darmo”. Nieskończoną listę, składającą się z samych jedynek definiujemy następująco:

```
ones :: [Int]
ones = 1:ones

*Main> take 5 ones
[1,1,1,1,1]
```

Ciąg rosnących liczb całkowitych zaczynający się od *k* to po prostu [*k*..], np.

```
*Main> take 5 [30..]
[30,31,32,33,34]
```

Dostępny jest też bardzo czytelny sposób konstrukcji list (ang. list comprehensions), np.

```
*Main> [x^2 | x <- [1..5]]
[1,4,9,16,25]
```

Idea i nazwa pochodzą z teorii mnogości, gdzie analogiczny zbiór można zdefiniować tak: $\{x^2 \mid x \in \{1..5\}\}$ (patrz wykład 3).

# *Sito Eratostenesa w języku Haskell*

Nieskończoną listę liczb pierwszych, otrzymaną za pomocą sita Eratostenesa, definiujemy następująco:

```
primes :: [Int]
primes = sieve [2..]
  where
     sieve  :: [Int] -> [Int]
     sieve (p:xs) = p:sieve [x| x<-xs, x `mod` p /= 0]

*Main> take 6 primes
[2,3,5,7,11,13]
```

Wykorzystywanie list nieskończonych wymaga ostrożności. Na przykład w wyniku ewaluacji wyrażenia filter (<= 5) [1..] wyprodukowany zostanie początek listy wynikowej [1,2,3,4,5

po czym proces zapętla się w poszukiwaniu liczb spełniających predykat w ogonie nieskończonej listy wejściowej. Biblioteka standardowa Haskella zawiera funkcję takeWhile :: (a -> Bool) -> [a] -> [a] , której ewaluacja kończy się po sfalsyfikowaniu predykatu.

```
Prelude> takeWhile (<= 5) [1..]
[1,2,3,4,5]
```

# *Ciąg liczb Fibonacciego w języku Haskell*

Poniższe wyrażenie generuje liczby Fibonacciego w sposób, który może zaskoczyć programistów, nieprzywykłych do ewaluacji leniwej, ale jest naturalny w Haskellu.

Dwie początkowe liczby Fibonacciego zostały umieszczone jawnie w nieskończonej liście. Reszta jest konstruowana następująco. Tworzona jest lista par, zbudowana za pomocą zip (patrz wykład 2, zip i zipWith są w standardowym Preludium) z dwóch list liczb Fibonacciego, przesuniętych o jedną pozycję. Pierwsza para to (0,1). Funkcjonał map bierze taki nieskończony strumień par i zwraca nieskończony strumień sum elementów tych par, co jest zgodne z definicją kolejnych liczb Fibonacciego.

Typ Integer dla liczb całkowitych dowolnej precyzji umożliwia wyliczenie dowolnie dużych liczb Fibonacciego. Biblioteki wszystkich współczesnych języków programowania udostępniają takie typy, np. OCaml – moduł Big_int, Java – klasa BigInteger, Scala – klasa BigInt.

```
fibs :: [Integer]
fibs = 0 : 1 : map (\(a,b) -> a+b) (zip fibs (tail fibs))
-- lub zwięźlej
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
*Main> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

# *Ewaluacja gorliwa w języku Haskell: funkcje* seq *i* deepseq

Haskell domyślnie wykorzystuje leniwą ewaluację, ale widzieliśmy na wykładzie 2 (patrz też niżej) przykład, kiedy potrzebna była ewaluacja gorliwa.

Haskell w standardowym preludium ma zdefiniowaną funkcję seq:

```
seq :: a -> b -> b
```

Wymusza ona ewaluację pierwszego argumentu, a następnie zwraca drugi argument.

Ewaluacja argumentu nie musi być kompletna. Kompletna ewaluacja wykonywana jest dla typów bazowych, np. dla Int czy Bool. Dla wartości strukturalnych jest inaczej. Jeśli argument jest parą, np. (Int, Bool), to ewaluacja jest kontynuowana do otrzymania pary wyrażeń, które nie muszą jeszcze być wartościami. Ogólnie, ewaluacja jest kontynuowana do osiągnięcia **konstruktora wartości**.

```
Prelude> (1, undefined) `seq` "OK"
"OK "
```

W module Control.DeepSeq jest zdefiniowana funkcja deepseq, przeprowadzająca ewaluację pierwszego wyrażenia do postaci normalnej i zwracająca drugi argument.

```
deepseq :: NFData a => a -> b -> b                -- NFData == Normal Form Data
```

```
Prelude> ((1, undefined)::(Int,Int)) `Control.DeepSeq.deepseq` "OK"
*** Exception: Prelude.undefined
```

# *Ewaluacja gorliwa w języku Haskell: funkcje* seq *i* deepseq

Funkcja seq jest wykorzystana w definicji operatora gorliwej aplikacji (ang. strict application) z łącznością prawostronną (jest również w standardowym preludium) :

```
infixr 0 $!
($!)  :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

W analogiczny sposób w module Control.DeepSeq za pomocą deepseq jest zdefiniowany operator:

```
($!!)  :: NFData a => (a -> b) -> a -> b
```

W przypadku funkcji w postaci rozwiniętej operatora gorliwej aplikacji można użyć do wymuszenia gorliwej ewaluacji dowolnych argumentów.

Jeśli f jest funkcją w postaci rozwiniętej od dwóch argumentów, to aplikacja f x y może być zmodyfikowana na trzy sposoby:

(f $! x) y  wymusza ewaluację x
(f x) $! y  wymusza ewaluację y
(f $! x) $! y  wymusza ewaluację x i y

# *Ewaluacja gorliwa w języku Haskell: funkcje* seq *i* deepseq

Funkcje seq, deepseq, ($!) i ($!!) powinny być wykorzystywane z umiarem ze względu na narzut czasowy. Zwykle standardowa ewaluacja leniwa jest najlepsza. Raczej rzadko potrzebna jest głęboka ewaluacja do postaci normalnej. Ograniczenia funkcji seq można obejść za pomocą standardowych technik programistycznych. Na przykład:

```
strictPair (a,b) = a `seq` b `seq` (a,b)

strictList (x:xs) = x `seq` x:strictList xs
strictList []     = []
```

Użyteczna może być też funkcja force z modułu Control.DeepSeq:

```
force  :: NFData a =>  a -> a
force x = x `deepseq` x
```

# *Ewaluacja gorliwa w języku Haskell: gorliwe konstruktory*

Konstruktory wartości algebraicznych typów danych w Haskellu domyślnie są leniwe.

```
Prelude> data T = K Int deriving (Eq,Ord,Show)
Prelude> let f (K _) = "OK"
Prelude> f (K (error "arg"))
"OK"
```

Istnieje jednak możliwość podania w definicji typu informacji o gorliwej ewaluacji wybranych argumentów konstruktorów. Służy do tego symbol ! , umieszczony przed typem wybranych argumentów konstruktora wartości.

Na przykład:

```
Prelude>  data Tstrict = Kstrict  !Int   deriving (Eq,Ord,Show)
Prelude> let fstrict (Kstrict _) = "OK"
Prelude> fstrict (Kstrict (error "arg"))
*** Exception: arg
```

# *Rekursja ogonowa w języku Haskell (1)*

Patrz wykład 2.
Zwykła rekursja:

```
suc :: Int -> Int
suc n = if n == 0 then 1 else 1 + suc (n-1)

suc 100000000
*** Exception: stack overflow
```

Rekursja ogonowa:

```
sucTail'   :: Int -> Int
sucTail' n = sucAux 1 n
             where
               sucAux accum 0 = accum
               sucAux accum n = sucAux (accum+1) (n-1)

sucTail' 100000000
*** Exception: stack overflow
```

# *Rekursja ogonowa w języku Haskell (2)*

W Haskellu ewaluacja jest leniwa, więc proces obliczania sucAux przebiega tak:

sucAux 1 3
= sucAux (1+1) 2
= sucAux ((1+1)+1) 1
= sucAux (((1+1)+1)+1) 0
= ((1+1)+1)+1 = (2+1)+1 = 3+1 = 4

Całe wyrażenie dla sumy w akumulatorze jest konstruowane, zanim zostaną wykonane dodawania. Pamięć zaoszczędzona na stosie została wykorzystana na zapamiętanie długiego wyrażenia. **W tym przypadku wartość akumulatora powinna być obliczana gorliwie**.

Haskell w standardowym preludium ma zdefiniowany operator gorliwej aplikacji (ang. strict application) z łącznością prawostronną:

```
infixr 0 $!
($!)  :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

Operator `$` zachowuje leniwe wartościowanie, zmienia tylko łączność na prawostronną.

# *Rekursja ogonowa w języku Haskell (3)*

Funkcja sucTail, wykorzystująca operator $! wygląda tak:

```
sucTail   :: Int -> Int
sucTail n = sucAux 1 n
             where
               sucAux accum 0  = accum
               sucAux accum n  = (sucAux $! (accum+1)) (n-1)

sucTail 100000000
100000001
```

Obliczenie tego wyniku trwa dużo dłużej niż w OCamlu. Szacowanie złożoności obliczeniowej programów z ewaluacja leniwą jest trudniejsze, niż w przypadku ewaluacji gorliwej.

Żadna ze strategii ewaluacji nie jest uniwersalnym panaceum. W OCamlu bywa przydatna ewaluacja leniwa, a w Haskellu – gorliwa. Trzeba rozumieć obie.

# *Ewaluacja gorliwa w języku Haskell: przykłady*

Powyższy przykład pokazywał dość typowy przypadek, kiedy trzeba wymuszać ewaluację akumulatora dla rekursji ogonowej. W ogólnym przypadku gorliwa aplikacja jest stosowana w celu poprawienia wykorzystania pamięci.

Kolejny przykład ilustruje to na przykładzie list. Potrzebujemy funkcji, która tworzy listę kolejnych liczb całkowitych od 1 do m. Poprawna definicja powinna oczywiście wyglądać tak:

```
nats :: Int -> [Int]
nats m = [1 .. m]
```

W celu ilustracji problemu użyjemy jednak rekursji.

# *Ewaluacja gorliwa w języku Haskell: przykłady*

```
nats' :: Int -> [Int]
nats' m = aux 1 m
    where
        aux n m = if m > 0 then n : aux (n+1) (m-1) else []
```

Wywołanie: length (nats' 100000000) na komputerze 32-bitowym z pamięcią 2GB powoduje przepełnienie pamięci: <interactive>: out of memory
Komputer 64-bitowy z pamięcią 4GB oblicza poprawny wynik.

Po wymuszeniu ewaluacji argumentu w obu przypadkach został wyliczony poprawny wynik (chociaż obliczenia trwały dużo dłużej, niż dla funkcji nats).

```
nats'' :: Int -> [Int]
nats'' m = aux 1 m
    where
       aux n m = if m > 0 then n : (aux $!  n+1) (m-1) else []
```

Ten sam efekt daje wywołanie: length (strictList (nats' 100000000))

# *Ewaluacja leniwa w językach gorliwych*

Najprostszym sposobem zachowania kontroli nad momentem ewaluacji wyrażenia w językach gorliwych jest utworzenie z niego funkcji. Funkcja (dokładniej: jej domknięcie) jest wartością. Wyrażenie, stanowiące treść funkcji, będzie obliczane dopiero po zaaplikowaniu funkcji do argumentu.

Jaki powinien być argument takiej funkcji? We współczesnych językach funkcyjnych z typizacją statyczną służy do tego typ unit, którego jedyną wartością jest ().

```
function () -> expr        (* abstrakcja funkcyjna, wartość *)

(function () -> expr)()    (* ewaluacja expr *)
```

# *Domknięcie funkcji jest wartością*

---------- OCaml

```
# let f = function () -> 2+5;;
val f : unit -> int = <fun>
# f();;
- : int = 7
```

---------- Haskell

```
Prelude> let f = \() -> 2+5
f :: Num a => () -> a
Prelude> f()
7
```

# *Listy leniwe (1) - OCaml*

Łącząc abstrakcję funkcyjną z abstrakcją danych i wykorzystując fakt, że funkcja (dokładniej – jej domknięcie) jest wartością, możemy zdefiniować nieskończone listy czy drzewa w językach z ewaluacją gorliwą.

Typ `llist` reprezentuje listy leniwe (skończone i nieskończone).

```
# type 'a llist = LNil | LCons of 'a * (unit -> 'a llist);;
type 'a llist = LNil | LCons of 'a * (unit -> 'a llist)

# let lhd = function
    LNil -> failwith "lhd"
  | LCons (x, _) -> x
;;
val lhd : 'a llist -> 'a = <fun>

# let ltl = function
    LNil -> failwith "ltl"
  | LCons (_, xf) -> xf()
;;
val ltl : 'a llist -> 'a llist = <fun>
```

# *Listy leniwe (2) - OCaml*

Funkcja `lfrom` generuje ciąg rosnących liczb całkowitych zaczynający się od *k*.

```
# let rec lfrom k = LCons (k, function () -> lfrom (k+1));;
val lfrom : int -> int llist = <fun>
```

Wywołanie `ltake(n,xl)` zwraca pierwszych *n* elementów listy leniwej *xl* w postaci zwykłej listy.

```
# let rec ltake = function
    (0, _)            -> []
  | (_, LNil)         -> []
  | (n, LCons(x,xf)) -> x::ltake(n-1, xf())
;;
val ltake : int * 'a llist -> 'a list = <fun>
```

Np.

```
# ltake (5,lfrom 30);;
- : int list = [30; 31; 32; 33; 34]
```

# *Listy leniwe (3) - OCaml*

Funkcja `toLazyList: 'a list -> 'a llist` ze zwykłej listy tworzy listę leniwą.

```
let rec toLazyList = function
    [] -> LNil
  | x::xs -> LCons(x, function () -> toLazyList xs);;
```

Powyższe funkcje są bardzo pomocne w testowaniu funkcji działających na listach leniwych.

# *Listy leniwe (OCaml) - wartościowanie*

Obliczenie `ltake(2,lfrom 30)` przebiega następująco:

```
ltake(2,lfrom 30)
=> ltake(2,LCons(30, function () -> lfrom (30+1)))
=> 30::ltake(1,lfrom (30+1))
=> 30::ltake(1,LCons(31, function () -> lfrom (31+1)))
=> 30::31::ltake(0,lfrom (31+1))
=> 30::31::ltake(0,LCons(32, function () -> lfrom (32+1)))
=> 30::31::[]
≡ [30;31]
```

Należy pamiętać, że **literał funkcyjny** (`function` x → e) **jest wartością** (patrz wykład 1, reguła *Fun*), więc wyrażenie e nie jest wartościowane do momentu aplikacji funkcji do argumentu.

# *Funkcjonały dla list leniwych: konkatenacja - OCaml*

Funkcja (@$) konkatenuje dwie listy leniwe (odpowiednik funkcji @ dla zwykłych list). Jeśli *ll1* jest nieskończona to wynikiem *ll1*@$ *ll2* ma być *ll1*.

```
# let rec (@$) ll1 ll2 =
  match ll1 with
      LNil -> ll2
    | LCons(x, xf) -> LCons(x, function () -> (xf()) @$ ll2);;
val ( @$ ) : 'a llist -> 'a llist -> 'a llist = <fun>

# let ll1 = LCons(2,function ()->LCons(1,function ()->LNil));;
val ll1 : int llist = LCons (2, <fun>)
# let ll2 = lfrom 3;;
val ll2 : int llist = LCons (3, <fun>)
# ltake (10, ll1 @$ ll2);;
- : int list = [2; 1; 3; 4; 5; 6; 7; 8; 9; 10]
```

# *Funkcjonały dla list leniwych: lmap - OCaml*

Dla list leniwych można zdefiniować funkcjonały podobne do tych, jakie zdefiniowaliśmy dla zwykłych list, np. `lmap`.

```
# let rec lmap f = function
    LNil -> LNil
  | LCons(x,xf) -> LCons(f x, function () -> lmap f (xf()) )
;;
val lmap : ('a -> 'b) -> 'a llist -> 'b llist = <fun>

# let sqr_llist = lmap (function x -> x*x);;
val sqr_llist : int llist -> int llist = <fun>

# ltake (6, sqr_llist (lfrom 3));;
- : int list = [9; 16; 25; 36; 49; 64]
```

# *Funkcjonały dla list leniwych: lfilter, liter* - OCaml

Funkcja `lfilter` wywoływana jest rekurencyjnie (rekursja ogonowa) tak długo, dopóki nie zostanie znaleziony element, spełniający podany predykat. Jeśli taki element nie istnieje, wartościowanie funkcji się nie skończy (dla nieskończonych list leniwych).

```
# let rec lfilter pred = function
    LNil -> LNil
  | LCons(x,xf) -> if pred x
                   then LCons(x, function () -> lfilter pred (xf()) )
                   else lfilter pred (xf())
;;
val lfilter : ('a -> bool) -> 'a llist -> 'a llist = <fun>
```

Funkcja `lfrom` jest szczególnym przypadkiem funkcji `liter`, która generuje ciąg:
`[x, f(x), f(f(x)), … f`$^k$`(x), …]`.

```
# let rec liter f x = LCons(x, function () -> liter f (f x));;
val liter : ('a -> 'a) -> 'a -> 'a llist = <fun>
```

# *Generowanie liczb pierwszych metodą sita Eratostenesa*

Utwórz ciąg nieskończony [2,3,4,5,6, ...]. Dwa jest liczbą pierwszą. Usuń z ciągu wszystkie wielokrotności dwójki, ponieważ nie mogą być liczbami pierwszymi. Pozostaje ciąg [3,5,7,9,11, ...]. Trzy jest liczbą pierwszą. Usuń z ciągu wszystkie wielokrotności trójki, ponieważ nie mogą być liczbami pierwszymi. Pozostaje ciąg [5,7,11,13,17, ...]. Pięć jest liczbą pierwszą itd.

Na każdym etapie ciąg zawiera liczby, które nie są podzielne przez żadną z wygenerowanych do tej pory liczb pierwszych, więc pierwsza liczba tego ciągu jest liczbą pierwszą. Opisane kroki można powtarzać w nieskończoność.

```
#let primes =
   let rec sieve = function
       LCons(p,nf) -> LCons( p,
                             function () -> sieve
                                (lfilter (function n -> n mod p <> 0)
                                         (nf())
                                )
                           )
     | LNil -> failwith "Impossible! Internal error."
    in sieve (lfrom 2)
;;
val primes : int llist = LCons (2, <fun>)
```

Np.

```
# ltake (6,primes);;
- : int list = [2; 3; 5; 7; 11; 13]
```

# *Generowanie liczb pierwszych „na zamówienie”*

Znalezienie kolejnej liczby pierwszej wymagałoby jednak powtórzenia wszystkich obliczeń od początku. Można temu zapobiec, modyfikując funkcję `ltake` w taki sposób, żeby oprócz skończonej listy początkowych elementów listy leniwej zwracała ogon listy leniwej.

```
# let rec ltakeWithTail = function
    (0, xq) -> ([],xq)
  | (_, LNil) -> ([],LNil)
  | (n, LCons(x,xf)) ->
      let (l,tail)=ltakeWithTail(n-1, xf())
      in (x::l,tail);;
val ltakeWithTail : int * 'a llist -> 'a list * 'a llist = <fun>
```

Teraz kolejne liczby pierwsze można otrzymywać w miarę potrzeby, wykorzystując zapamiętany ogon listy leniwej.

```
# let (p1,t) = ltakeWithTail (3, primes);;
val p1 : int list = [2; 3; 5]
val t : int llist = LCons (7, <fun>)
# let (p2,t) = ltakeWithTail (3, t);;
val p2 : int list = [7; 11; 13]
val t : int llist = LCons (17, <fun>)
# let (p3,t) = ltakeWithTail (5, t);;
val p3 : int list = [17; 19; 23; 29; 31]
val t : int llist = LCons (37, <fun>)
# let (p4,t) = ltakeWithTail (4, t);;
val p4 : int list = [37; 41; 43; 47]
val t : int llist = LCons (53, <fun>)
```

# *Moduł Lazy (1) – OCaml*

W języku OCaml istnieje moduł `Lazy`, który umożliwia kontrolę nad momentem wartościowania wyrażenia. Wykorzystując słowo kluczowe `lazy` możemy utworzyć wartość typu `'a lazy_t`, zawierającą „zamrożone” wartościowanie, które możemy „odmrozić” (tj. wymusić ewaluację) za pomocą funkcji `Lazy.force : 'a Lazy.t -> 'a.`
Typ `'a t` jest zdefiniowany w module `Lazy` jako: `type 'a t = 'a lazy_t`. Np.

```
# let x = lazy (true||false, 3*4);;
val x : (bool * int) lazy_t = <lazy>
# Lazy.force x;;
-: bool * int = (true, 12)
```

„Zamrożone” wartościowanie zostało wykonane i obliczona wartość jest zapamiętana, dzięki czemu ponowna ewaluacja nie będzie potrzebna. Są tu wykorzystane imperatywne cechy języka, o których będziemy mówili później.

```
# x;;
- : (bool * int) lazy_t = lazy (true, 12)
```

# *Moduł Lazy (2) – OCaml*

Oto inny przykład.

```
# let f x = "OK";;
val f : 'a -> string = <fun>
# f(lazy(1/0));;
- : string = "OK"
```

Argument jest wartościowany leniwie, a ponieważ nie jest wykorzystywany w funkcji oznacza to, że nigdy nie jest wartościowany.

```
# f(1/0);;
Exception: Division_by_zero.
```

Tutaj argument jest wartościowany gorliwie.

# *Leniwe wzorce – OCaml*

Do wzorca

lazy *pat*

dopasowuje się wartość *v* typu Lazy.t, jeśli wynik wymuszenia Lazy. force *v* dopasowuje się do wzorca *pat*. Słowo lazy może wystąpić we wzorcu wszędzie tam, gdzie można umieścić zwykły konstruktor wartości.

Wykorzystując ten mechanizm można zaimplementować funkcję biblioteczną Lazy.force:

```
# let force (lazy v) = v;;
val force : 'a lazy_t -> 'a = <fun>
```

Wykorzystując moduł Lazy i leniwe wzorce można definiować struktury nieskończone bez jawnego użycia funkcji dla opóźnienia ewaluacji wyrażenia. Poniżej zostanie to zilustrowane na przykładzie list leniwych.

# *Wzorce leniwe i listy leniwe (1) - OCaml*

```
# type 'a llist = LNil | LCons of 'a * 'a llist Lazy.t;;
type 'a llist = LNil | LCons of 'a * 'a llist Lazy.t

# let lhd = function
     LNil -> failwith "lhd"
   | LCons (x, _) -> x;;
val lhd : 'a llist -> 'a = <fun>
# let ltl = function
    LNil -> failwith "ltl"
  | LCons (_, lazy t) -> t;;
val ltl : 'a llist -> 'a llist = <fun>

# let rec lfrom k = LCons (k, lazy (lfrom (k+1)));;
val lfrom : int -> int llist = <fun>
# let rec ltake = function
    (0, _) -> []
  | (_, LNil) -> []
  | (n, LCons(x,lazy xs)) -> x::ltake(n-1,xs);;
val ltake : int * 'a llist -> 'a list = <fun>
```

# *Wzorce leniwe i listy leniwe (2) - OCaml*

```
# let rec lsquares = function
    LNil -> LNil
  | LCons(x,lazy xs) -> LCons(x*x, lazy(lsquares xs));;
val lsquares : int llist -> int llist = <fun>

# ltake (6, lsquares(lfrom 3));;
- : int list = [9; 16; 25; 36; 49; 64]

# let rec lmap f = function
    LNil -> LNil
  | LCons(x,lazy xs) -> LCons(f x, lazy(lmap f xs));;
val lmap : ('a -> 'b) -> 'a llist -> 'b llist = <fun>

# let sqr_llist = lmap (function x -> x*x);;
val sqr_llist : int llist -> int llist = <fun>

# ltake (6, sqr_llist (lfrom 3));;
- : int list = [9; 16; 25; 36; 49; 64]
```

# *Parametry a argumeny funkcji*

W wywołaniu funkcji (aplikacji funkcji do argumentu) przekazywane wyrażenia są *argumentami* funkcji.

Argumenty są wykorzystywane do inicjowania *parametrów* funkcji, czyli:

parametry = argumenty formalne (Stroustrup) lub

argumenty = parametry aktualne.

To rozróżnienie jest szczególnie ważne we współczesnym C++, ponieważ parametry są l-wartościami (ang. l-values), ale argumenty, którymi są inicjowane mogą być l-wartościami lub r-wartościami.

# *Przekazywanie argumentów do funkcji*

Mechanizmy gorliwe

- przez wartość (ang. call by value)
- przez referencję (ang. call by reference)
- przez przenoszenie w C++ (ang. move semantics)

Mechanizmy leniwe

- przez nazwę (ang. call by name)
- przez potrzebę (ang. call by need)

W językach z ewaluacją gorliwą najczęściej stosowany jest mechanizm przekazywania argumentów do funkcji przez wartość.

# *Przekazywanie argumentów do funkcji*

**Wywołanie przez wartość** (ang. call by value)
Argument aktualny jest ewaluowany zawsze. Do funkcji jest przekazywana jego wartość lub odwołanie do wartości, ale funkcja nie może zmieniać tej wartości w środowisku wywołującym.
Mechanizm wykorzystywany w językach OCaml, Java, C; domyślnie w językach Scala, C++, C#, Pascal.

**Wywołanie przez referencję** (ang. call by reference)
Argument aktualny jest ewaluowany zawsze. Do funkcji jest przekazywane jest odwołanie (referencja) do obliczonej wartości. Zmiana wartości argumentu wewnątrz funkcji powoduje jej zmianę w środowisku wywołującym.
Mechanizm opcjonalny w językach C++ (argument przekazywany przez referencję jest poprzedzany symbolem `&`), Pascal (argument przekazywany przez referencję jest poprzedzany słowem kluczowym `var`), C# (słowo kluczowe `ref`).

**Wywołanie przez nazwę** (ang. call by name)
Argument aktualny nie jest ewaluowany. Do funkcji jest przekazywane jest jego domknięcie (ang. thunk). Obliczanie wartości domknięcia jest *wymuszane* (ang. forced) tylko wtedy, kiedy wartość argumentu jest potrzebna (za każdym razem).
Mechanizm opcjonalny w języku Scala.

**Wywołanie przez potrzebę** (ang. call by need)
Jest to odmiana wywołania przez nazwę. Po pierwszym wymuszeniu obliczania wartości domknięcia argumentu aktualnego, wartość ta jest zapamiętywana i wykorzystywana w dalszych obliczeniach. Mechanizm wykorzystywany w języku Haskell.

# *Argumenty pozycyjne, nazwane i domyślne*

Poniższa klasyfikacja przekazywania argumentów do funkcji jest bardziej związana z udogodnieniami składniowymi, niż z mechanizmami semantycznymi.

**Argumenty pozycyjne**

Jest to najczęściej stosowany sposób. Polega on na podawaniu wartości argumentów w kolejności ich występowania na liście deklaracji parametrów formalnych w definicji funkcji.

**Argumenty nazwane (etykietowane)**

W niektórych językach programowania argumenty można identyfikować za pomocą nazwy parametru formalnego (np. Scala) lub dodatkowej etykiety (np. OCaml). Nie trzeba wtedy przestrzegać kolejności argumentów, ustalonej w definicji funkcji. Szczegóły zależą od języka programowania.

**Argumenty domyślne (parametry opcjonalne)**

Deklarując parametry funkcji, można podać ich domyślne wartości (zwykle po znaku równości).

# *Argumenty nazwane w języku OCaml*

W OCamlu parametry etykietowane są deklarowane za pomocą składni *~label: pattern*. Składnia argumentów etykietowanych jest podobna *~label: expression*.

```
# let minus ~x:m ~y:n = m - n;;
val minus : x:int -> y:int -> int = <fun>
# minus ~y:2 ~x:5;;
- : int = 3
# minus ~y:2;;
- : x:int -> int = <fun>
```

Jeśli etykiety są takie same jak nazwy parametrów, to można użyć skrótu *~label*.

```
# let minus ~m ~n = m - n;;
val minus : m:int -> n:int -> int = <fun>
# let m = 5;;
val m : int = 5
# minus ~n:(2+1) ~m;;
- : int = 2
# minus 5 2;;
- : int = 3
```

# *Argumenty nazwane i domyślne w języku OCaml*

**Argumenty nazwane (etykietowane)**

```
# minus ~m:5 2;;
Characters 11-12:
  minus ~m:5 2;;
                ^
Error: The function applied to this argument has type n:int -> int
This argument cannot be applied without label
```

**Argumenty domyślne**

Parametry opcjonalne mają składnię ?(*label* = *expression*). Parametr opcjonalny nie może być ostatnim parametrem.

```
# let minus ?(m=5) n = m - n;;
val minus : ?m:int -> int -> int = <fun>
# minus 2;;
-    : int = 3
-    # minus ~m:15 2;;
-    - : int = 13
```

# *Funkcje wariadyczne*

Wiele języków programowania pozwala definiować *funkcje wariadyczne* (ang. variadic functions), które przyjmują zmienną liczbę argumentów.

OCaml ani Haskell nie udostępniają specjalnych mechanizmów dla funkcji wariadycznych, ale taka notacja istnieje w języku Scheme (co zobaczymy później).

# *Idea algorytmów z powrotami (1)*

Załóżmy, że dana jest pewna przestrzeń *stanów*, oraz sposób przechodzenia od jednego stanu do drugiego ( czyli możliwe “ruchy”, stanowiące elementy rozwiązania). Zadany jest stan początkowy oraz pożądany stan docelowy (lub zbiór stanów docelowych). Szukanym rozwiązaniem jest wektor ruchów, prowadzących od stanu początkowego do dowolnego stanu docelowego.

Idea algorytmu z powrotami polega na rozszerzaniu rozwiązania częściowego przez  dodawanie do wektora rozwiązania kolejnego ruchu, dopuszczalnego w danym stanie, dopóki nie zostanie osiągnięty stan docelowy. Jeśli takie rozszerzenie nie jest możliwe, należy usunąć z rozwiązania częściowego ostatni element (*powrót*) i próbować wykonać inny ruch, dopuszczalny w otrzymanym stanie. Wymaga to zorganizowania systematycznego przeszukiwania przestrzeni stanów.

# *Idea algorytmów z powrotami (2)*

Proces przeszukiwania przestrzeni stanów wygodnie jest przedstawiać jako obejście w głąb pewnego drzewa. Węzły tego drzewa reprezentują stany. Korzeniowi odpowiada stan początkowy, a liściom stany docelowe lub stany, w których nie istnieje dopuszczalny ruch (wymagające powrotu). Gałęzie wychodzące z węzła reprezentują ruchy, które można wykonać w odpowiadającym mu stanie. Rozwiązanie jest drogą od korzenia do liścia, odpowiadającego stanowi docelowemu. Wyzwaniem jest znalezienie takiego sposobu poruszania się po drzewie, żeby skonstruować rozwiązanie, wizytując jak najmniej wierzchołków. Znane są różne strategie obejścia przestrzeni stanów:

- Przeszukiwanie *w głąb* jest mało kosztowne, ale pozwala na głębokie wejście w niewłaściwą ścieżkę w drzewie przed ewentualnym powrotem, co może być nie do przyjęcia dla dużych drzew (przestrzeni stanów), np.dla drzew nieskończonych.
- Przeszukiwanie *wszerz* (wymaga wykorzystania kolejki,często bardzo dużej) gwarantuje znalezienie wszystkich rozwiązań.
- Można też stosować np. *metodę podziałów i ograniczeń*, bądź znaleźć heurystykę, użyteczną przy poszukiwaniu rozwiązania konkretnego problemu.

Przedstawiając rozwiązania w postaci listy leniwej możemy łatwo oddzielić strategię przeszukiwania przestrzeni stanów od „konsumenta” rozwiązań. Drzewo stanów będziemy reprezentowali za pomocą funkcji `next: 'a -> 'a list`. `next w` generuje listę poddrzew węzła `w`.

# *Strategia obejścia przestrzeni stanów w głąb*

Strategię przeszukiwania w głąb można efektywnie zaimplementować z wykorzystaniem stosu do przechowywania węzłów, które mają być sprawdzane w następnej kolejności. W każdym kroku przeszukiwania element `h` ze szczytu stosu jest usuwany, a w jego miejsce są wkładane następniki, generowane przez funkcję `next h`. Rozwiązania są identyfikowane za pomocą funkcjonału `lfilter` z odpowiednim predykatem.

```
# let depthFirst next x =
  let rec dfs = function
      [] -> LNil
    | (h::t) -> LCons (h, function () -> dfs (next h @ t))
  in dfs [x];;
val depthFirst : ('a -> 'a list) -> 'a -> 'a llist = <fun>
```

# *Przykład: problem ośmiu hetmanów (1)*

Należy ustawić osiem hetmanów na szachownicy w taki sposób, żeby się wzajemnie nie atakowały.

Lista [*q1*, ...,*qk*] reprezentuje szachownicę z hetmanami stojącymi w wierszu *qi* *i*tej kolumny dla *i*=1,...,*k*. Funkcja `isQueenSafe` sprawdza, czy hetman może być postawiony w wierszu *newq*, co utworzy szachownicę [*newq*,*q1*, ...,*qk*] . Nowy hetman nie może się znajdować w tym samym wierszu ani na tej samej przekątnej, co wcześniej ustawione hetmany. Zauważ, że | *newq*-*qi*|=*i* zachodzi wtedy i tylko wtedy, kiedy *newq* i *qi* znajdują się na tej samej przekątnej.

```
let isQueenSafe oldqs newq =
  let rec nodiag = function
      (i, []) -> true
    | (i, q::qt) -> abs(newq-q)<>i && nodiag(i+1,qt)
  in not(List.mem newq oldqs) && nodiag(1,oldqs);;
 val isQueenSafe : int list -> int -> bool = <fun>
```

# *Przykład: problem ośmiu hetmanów (2)*

Funkcja `nextQueen` bierze szachownicę *qs* i generuje listę szachownic z poprawnie ustawionym kolejnym hetmanem. Parametr *n* pozwala rozwiązywać problem *n* hetmanów.

```
# let rec fromTo  a b =
      if a>b then []
      else a::(fromTo (a+1) b);;
val fromTo : int -> int -> int list = <fun>
# let nextQueen n qs =
           List.map (function h -> h::qs)
                    (List.filter (isQueenSafe qs) (fromTo 1 n));;
val nextQueen : int -> int list -> int list list = <fun>
# let isSolution n qs = List.length qs = n;;
val isSolution : int -> 'a list -> bool = <fun>
# let depthQueen n = lfilter (isSolution n)
                             (depthFirst (nextQueen n) []);;
val depthQueen : int -> int list llist = <fun>
```

# *Przykład: problem ośmiu hetmanów (3)*

Funkcja `depthQueen` znajduje wszystkie rozwiązania problemu *n* hetmanów (dla *n*=8 jest ich 92). Funkcjonał `ltakeWithTail` pozwala otrzymywać kolejne rozwiązania na żądanie. W proceduralnych językach programowania wymagałoby to prawdopodobnie wykorzystania współbieżności i scenariusza „producent-konsument”, w językach obiektowych można to osiągnąć przedstawiając „producenta” jako obiekt.

```
# let (dq1,t) = ltakeWithTail (1, depthQueen 8);;
val dq1 : int list list = [[4; 2; 7; 3; 6; 8; 5; 1]]
val t : int list llist = LCons ([5; 2; 4; 7; 3; 8; 6; 1], <fun>)
# let (dq2,t) = ltakeWithTail (3, t);;
val dq2 : int list list =
  [[5; 2; 4; 7; 3; 8; 6; 1]; [3; 5; 2; 8; 6; 4; 7; 1];
   [3; 6; 4; 2; 8; 5; 7; 1]]
val t : int list llist = LCons ([5; 7; 1; 3; 8; 6; 4; 2], <fun>)
```

# *Zadania kontrolne*

1. Zdefiniuj funkcję, która dla danej nieujemnej liczby całkowitej *k* i listy leniwej [$x_0$, $x_1$, $x_2$, ... ] zwraca listę leniwą, w której każdy element jest powtórzony *k* razy, np. dla *k*=3:
   [$x_0$, $x_0$, $x_0$, $x_1$, $x_1$, $x_1$, $x_2$, $x_2$, $x_2$, ... ].

2. Zdefiniuj leniwą listę liczb Fibonacciego *lfib* : `int llist`,

3. Polimorficzne leniwe drzewa binarne można zdefiniować następująco:

```
type 'a lBT = LEmpty
            | LNode of 'a * (unit ->'a lBT) * (unit -> 'a lBT);;
```

a) Napisz funkcję *itr*, która dla zadanej liczby całkowitej *n* konstruuje nieskończone leniwe drzewo binarne z korzeniem o etykiecie *n* i z dwoma poddrzewami itr(2**n*) i itr(2**n*+1). To drzewo jest przydatne do testowania funkcji z następnego podpunktu.

b) Napisz funkcję, tworzącą leniwą listę, zawierającą wszystkie etykiety leniwego drzewa binarnego.
   *Wskazówka*: zastosuj obejście drzewa wszerz, reprezentując kolejkę jako zwykłą listę.

4. Napisz funkcję `breadthFirst : ('a -> 'a list) -> 'a -> 'a llist`, która przeszukuje przestrzeń stanów wszerz i wykorzystaj ją do rozwiązania problemu ośmiu hetmanów.

5. Napisz w języku Haskell funkcje, rozwiązujące problem ośmiu hetmanów.