Kurs języka Haskell

Notatki zamiast wykładu i lista zadań na pracownię nr 4

Do zgłoszenia w SKOS-ie do 3 kwietnia 2020

Trwałe struktury danych i współdzielenie

Trwałe struktury danych (*persistent data structures*), to takie struktury, które posiadają wiele *wersji*. Modyfikacje trwałych struktur prowadzą do powstania nowych wersji. Oryginalne wersje pozostają niezmienione i nadal są dostępne. Jedynymi strukturami danych w językach czysto deklaratywnych (do których zalicza się Haskell) są struktury trwałe. Języki, które nie są czysto deklaratywne, a jedynie wspierają metodologię deklaratywną (OCaml, Python, także Java) udostępniają również struktury *ulotne* (*ephemeral*), które są modyfikowane w miejscu. Modyfikacja struktury powoduje zniszczenie poprzedniej wersji. Po modyfikacji różne fragmenty programu, które korzystały z oryginalnej wersji nagle „widzą" strukturę zmienioną. Rozważmy fragment programu:

```
xs = [1,2,3]
ys = [4,5]
p = (xs,ys)
zs = xs ++ ys
```

W języku imperatywnym, takim jak C, można by przypisać wskaźnikowi `zs` wartość wskazywaną przez `xs` i zmodyfikować ostatni element listy `[1,2,3]` tak, by zamiast wartości `NULL` wskazywał na to samo, na co wskazuje `ys`. Jeśli mielibyśmy dostęp do ostatniego elementu listy `xs` (w językach imperatywnych reprezentowanie listy jednokierunkowej w postaci pary wskaźników na pierwszy i ostatni element jest popularne), to połączenie list moglibyśmy wykonać w czasie stałym. Jednak wartość wskazywana przez `p` w wierszu drugim uległaby zmianie — zamiast na parę `([1,2,3],[4,5])` zmienna `p` po wykonaniu wiersza czwartego wskazywałaby na parę `([1,2,3,4,5],[4,5])`.

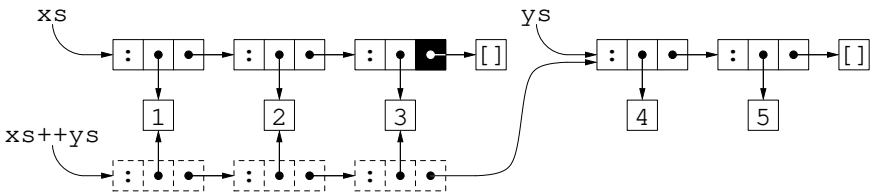
W języku czysto deklaratywnym, takim jak Haskell, jest to niemożliwe. Aby połączyć listy `xs` i `ys` musimy wykonać *kopię* listy `xs`. Listę `ys` możemy *współdzielić*. Zatem obliczenie wiersza czwartego polega na utworzeniu w pamięci kopii listy `[1,2,3]` i dołączeniu na jej koniec listy `[4,5]`.

Konstruktory typów danych możemy porównać do konstruktorów domyślnych obiektów w językach obiektowych. Nie trzeba ich osobno programować, a wszystko co robią, to alokują pamięć dla nowego obiektu i inicjalizują jego pola podanymi wartościami. Tę analogię możemy rozszerzyć na *smart konstruktory* z poprzedniej listy, które odpowiadają specjalnie zaprogramowanym konstruktorom obiektów, wykonującym dodatkowe czynności oraz na *widoki*, które „udają" wykorzystanie tych smart konstruktorów we wzorcach.

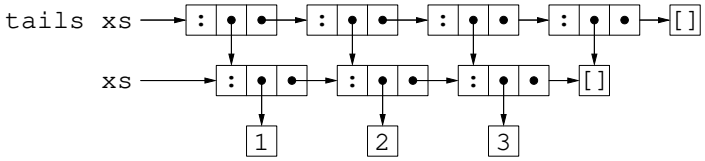
Zatem każde wystąpienie konstruktora w wyrażeniu haskellowym (mimo iż nie poprzedzone, jak w językach obiektowych, słowem kluczowym `new`) jest związane z alokacją pamięci. Mówimy tu o wyrażeniach znajdujących się w ciałach klauzul, a nie o wzorcach. Rozważmy operację łączenia list:

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

Indukcja biegnie tu względem pierwszego argumentu. Liczba wywołań rekurencyjnych, a zatem i liczba wywołań konstruktora `(:)` w ciele ostatniej klauzuli jest równa liczbie elementów listy, będącej lewym argumentem `(++)`. Lista ta podczas obliczenia funkcji `(++)` ulega zatem skopiowaniu. Zauważmy, że kopiujemy jedynie *kręgosłup* listy — jej elementy są *współdzielone*. Współdzielony jest też prawy argument funkcji `(++)`. Zatem `(++)` kopiuje cały swój lewy argument, a współdzieli prawy. Jej koszt (zarówno czasowy, jak i pamięciowy) jest proporcjonalny do długości lewego argumentu.



Rysunek 1: Wynik połączenia list `xs = [1,2,3]` i `ys = [4,5]`. Modyfikowane pole oznaczono kolorem czarnym, a elementy skopiowane — linią przerywaną



Rysunek 2: Wynik obliczenia funkcji `tails` dla listy trzyelementowej

Kluczem do projektowania efektywnych trwałych struktur danych i algorytmów na nich działających jest minimalizacja kopiowania i maksymalizacja współdzielenia. Trwałe struktury danych gwaratują, że wersja, którą posiadamy, nie ulegnie zmianie. Ten niezmiennik pozwala na znaczące zwiększenie współdzielenia. Jeśli potraktujemy struktury danych jako grafy, w których wierzchołki reprezentują węzły struktury przechowywane w pamięci, a krawędzie odpowiadają wskaźnikom, to aby zmodyfikować węzeł trwałej struktury musimy skopiować całą składową grafu, która wskazuje na modyfikowany (a więc kopiowany) węzeł (aby wcześniejsze wersje nie uległy zmianie). Aby zatem obliczyć `xs ++ ys`, co wymaga modyfikacji końca listy `xs`, musimy skopiować całą listę `xs` (bo z wszystkich elementów listy ten koniec jest osiągalny). Natomiast całą listę `ys` możemy współdzielić, gdyż mamy gwarancję, że nie ulegnie ona zmianie. Dokładnie tę strategię realizuje przedstawiony wyżej kod funkcji `(++)`. Wynik obliczenia `xs ++ ys` jest przedstawiony na Rysunku 1. Uproszczono na nim problem przechowywania w pamięci wartości typów prostych, które mieszczą się w pojedynczym słowie maszynowym. Kompilator może optymalizować zużycie pamięci wykonując *unboxing*, tj. kopiując wskazywaną wartość w miejsce wskaźnika, który na nią wskazywał.[1] Rysunek ten jednak dobrze pokazuje, że kopiowaniu ulega jedynie kręgosłup listy. Z powodu unboksingu nie warto jednak w praktyce starać się współdzielić wartości, które mieszczą się w pojedynczym słowie maszynowym, np. listy pustej `[]`. Zatem pisanie np.

```
f xs@[] = (xs,xs)
```

zamiast

```
f [] = ([],[])
```

jest zbytecznym zaciemnianiem programu. Zauważmy, że funkcja

```
tails :: [a] -> [[a]]
tails [] = [[]]
tails xs@(_:xs') = xs : tails xs'
```

dla listy n-elementowej alokuje jedynie $n + 1$ elementów listy (patrz Rysunek 2), podczas gdy funkcja

```
inits :: [a] -> [[a]]
inits [] = [[]]
inits xs@(x:xs') = xs :: map (x:) $ inits xs'
```

potrzebuje ich aż $\frac{1}{2} \cdot n \cdot (n + 1)$. Wynika to wprost z trwałości struktury danych (modyfikując element musimy skopiować wszystkie elementy, z których jest osiągalny) i nic na to nie poradzimy.

Często jednak nadmiar kopiowania w stosunku do współdzielenia bierze się z niewłaściwej konstrukcji algorytmu. Wówczas niepotrzebnie kopiowane elementy stają się też często *nieużytkami*, tj. komórkami pamięci nieosiągalnymi z żadnej *żywej* struktury danych. Nieużytki muszą być następnie usunięte

[1] W Haskellu mamy też specjalne unboksowane typy danych, takie jak `Int#`.

przez *garbage collector*. Rozważmy dla przykładu nieumiejętną implementację funkcji odwracającej listę:

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

Powyższy kod jest czasem podawany jako *specyfikacja* funkcji odwracającej listę, gdyż jest prosty i zrozumiały (dzięki m. in. użyciu indukcji strukturalnej względem argumentu). Jak ustaliliśmy jednak wcześniej, funkcja `(++)` kopiuje cały swój lewy argument i współdzieli prawy. Wynik zakończonego wywołania rekurencyjnego `reverse xs` jest zatem kopiowany, a sam staje się nieużytkiem (nie ma bowiem żadnego żywego wskaźnika, który by nań wskazywał).

Polecenie 1. Udowodnij, że dla powyższej funkcji `reverse` i listy n-elementowej zarówno czas działania, jak i ilość alokowanej pamięci wynoszą $\frac{1}{2} \cdot n \cdot (n+1)$, z czego jedynie n komórek pozostaje żywych.

Uniwersalną metodą unikania takiego zbytecznego kopiowania jest zaprogramowanie ogólniejszej funkcji `rev`, zależnej od dodatkowego argumentu i powiązanej z funkcją `reverse` następującą zależnością:

$$\texttt{rev}\ a\ xs \quad = \quad \texttt{reverse}\ xs\ \texttt{++}\ a$$

Aby wyprowadzić definicję funkcji `rev` przez indukcję strukturalną, wstawiamy najpierw w miejsce xs listę pustą i otrzymujemy

$$\texttt{rev}\ a\ \texttt{[]} = \texttt{reverse}\ \texttt{[]}\ \texttt{++}\ a = \texttt{[]}\ \texttt{++}\ a = a$$

Kładąc zaś $(x : xs)$ mamy

$$\begin{gathered}\texttt{rev}\ a\ (x : xs) = \texttt{reverse}\ (x : xs)\ \texttt{++}\ a = (\texttt{reverse}\ xs\ \texttt{++}\ \texttt{[}x\texttt{]})\ \texttt{++}\ a = \\ \texttt{reverse}\ xs\ \texttt{++}\ (\texttt{[}x\texttt{]}\ \texttt{++}\ a) = \texttt{reverse}\ xs\ \texttt{++}\ (x : a) = \texttt{rev}\ (x : a)\ xs\end{gathered}$$

Dzięki dodatkowemu parametrowi a mogliśmy skorzystać z łączności funkcji `(++)` i w efekcie ją wyeliminować. Mamy zatem:

```
reverse = rev [] where
   rev a [] = a
   rev a (x:xs) = rev (x:a) xs
```

Teraz funkcja `reverse` działa w czasie liniowym i nie tworzy nieużytków — cała zaalokowana pamięć zostaje wbudowana w zwracany wynik. Parametr a jest często nazywany *akumulatorem*, gromadzi bowiem wyniki częściowych obliczeń.

Niefrasobliwe użycie funkcji `(++)` bardzo często prowadzi do powstawania zbytecznych nieużytków, jak np. w funkcjach:

```
data BTree a = BNode (BTree a) a (BTree a) | BLeaf

flatten :: BTree a -> [a]
flatten Leaf = []
flatten (Node l x r) = flatten l ++ [x] ++ flatten r

qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ [x] ++ qsort [y | y <- xs, y >= x]
```

W obu funkcjach wynik lewego wywołania rekurencyjnego jest kopiowany i staje się nieużytkiem.

Zadanie 1 (1 pkt). Zaprogramuj dla powyższych funkcji wersje z funkcjami pomocniczymi wykorzystującymi akumulator tak, by nie było konieczności wywoływania silnie kopiującej funkcji `(++)`. Funkcje pomocnicze powinny spełniać specyfikacje:

$$\begin{aligned} \texttt{flatten\_aux}\ a\ t &= \texttt{flatten}\ t\ \texttt{++}\ a \\ \texttt{qsort\_aux}\ a\ xs &= \texttt{qsort}\ xs\ \texttt{++}\ a \end{aligned}$$

Zauważmy, że tak usprawniona funkcja `flatten` nie generuje żadnych nieużytków, podczas gdy `qsort` nadal generuje nieużytki (choć o około połowę mniej, niż wcześniejsza wersja) — są to listy będące argumentami wywołań rekurencyjnych. Prawdopodobnie nie da się zaprogramować funkcji sortującej, która działałaby w czasie $n \log n$ i generowała mniej niż $n \log n$ nieużytków.

W niektórych przypadkach kompilator potrafi samodzielnie przebudować program tak, żeby zmniejszyć ilość tworzonych nieużytków. Oczywiście nie zawsze to jest możliwe. Zbalansowane drzewo poszukiwań

```
toTree -> Node (Node Leaf 1 Leaf) 2 (Node Leaf 3 Leaf)
```

możemy uzyskać jako wynik wstawienia etykiety `3` do drzewa

```
toTree -> Node (Node Leaf 1 Leaf) 2 Leaf
```

oraz jako wynik wstawienia etykiety `1` do drzewa

```
toTree -> Node Leaf 2 (Node Leaf 3 Leaf)
```

Problem znalezienia dwóch równych wyników obliczeń jest statycznie nierozstrzygalny, a dynamicznie bardzo kosztowny. W Haskellu sprowadza się do sprawdzania izomorfizmu grafów, a nawet dla drzew wymagałby ogromnych obliczeń. Tzw. agresywne współdzielenie, tj. wyszukiwanie i sklejanie równych poddrzew, jest używane w systemach dowodzenia twierdzeń do redukowania rozmiaru zbiorów klauzul. W języku programowania takie podejście byłoby niepraktyczne. Jednak kompilator może czasami statycznie zauważyć, że dane ulegają duplikacji i zwiększyć współdzielenie. Więcej o tym powie później Maciek.

W odróżnieniu od funkcji `reverse`, której naiwna wersja działała w czasie kwadratowym, generowanie nieużytków przez funkcje `flatten` i `qsort` nie zmienia klasy złożoności czasowej i pamięciowej. Można się zatem zastanawiać, czy warto komplikować i zaciemniać te programy przez dodanie akumulatora, zwłaszcza że kompilator może niekiedy sam je zoptymalizować.

Zmniejszenie współdzielenia ma również miejsce w sytuacji, gdy w ciele klauzuli „odbudowujemy" rozebrany na części przez wzorzec argument funkcji, jak w przypadku funkcji scalającej posortowane listy:

```
(<+>) :: Ord a => [a] -> [a] -> [a]
[] <+> ys = ys
xs <+> [] = xs
(x:xs) <+> (y:ys)
   | x <= y    = x : (xs <+> (y:ys))
   | otherwise = y : ((x:xs) <+> ys)
```

Dobrze optymalizujący kompilator może sam przerobić powyższy kod do następującego:

```
[] <+> ys = ys
xs <+> [] = xs
xs@(x:xs') <+> ys@(y:ys')
   | x <= y    = x : (xs' <+> ys)
   | otherwise = y : (xs <+> ys')
```

ale w tym przypadku warto wykonać tę optymalizację samodzielnie, gdyż zwiększa ona czytelność kodu.

Projektując trwałe struktury danych należy zadbać, by liczba węzłów struktury, z których można osiągnąć modyfikowany element, zawsze była mała (bo węzły te *musimy* skopiować). Listy nadają się do

wykorzystania jako kolekcje elementów tylko wtedy, gdy przetwarzamy je sekwencyjnie, element po elemencie. Użycie takich kosztownych operacji, jak `(++)` i `(!!)` zwykle degraduje efektywność programu. Jeśli potrzebujemy modyfikować dowolne elementy kolekcji, to drzewa zbalansowane (tj. takie, w których liczba wierzchołków jest wykładnicza względem wysokości) są znacznie lepszym rozwiązaniem. Dla dowolnego wierzchołka drzewa zbalansowanego wszystkie wierzchołki, z których dany wierzchołek jest osiągalny, leżą na ścieżce od korzenia do tego wierzchołka. Aby zachować trwałość, modyfikacja dowolnego wierzchołka wymaga zatem skopiowania całej ścieżki od korzenia do tego wierzchołka (*path copying*), ale jest to koszt jedynie logarytmiczny względem rozmiaru drzewa. Wersja zmodyfikowana i oryginalna współdzielą przeważającą część drzewa.

W językach imperatywnych zwykle mamy dostęp jedynie do najnowszej wersji struktury danych. Projektujemy więc nasze algorytmy tak, żeby wcześniejsze wersje nie były potrzebne. Często rzutujemy nasze doświadczenia z programowania imperatywnego na algorytmy implementowane w języku czysto deklaratywnym — tworzymy algorytmy, które działają *jednowątkowo*, tj. nie korzystają z poprzednich wersji struktury. Jeśli poprzednie wersje nie są potrzebne, to wystarczy o nich *zapomnieć*, tzn. wystarczy zapewnić, by nie było żadnego żywego wskaźnika na te wersje. W językach niskiego poziomu takie zjawisko nazywamy *wyciekami pamięci*, jednak w Haskellu cała zapomniana pamięć, czyli *nieużytki*, jest odzyskiwana przez *garbage collector*. Oczywiście w miarę możności należy unikać tworzenia nieużytków, ale są one naturalnym efektem przetwarzania trwałych struktur danych. W całkiem realistycznym modelu obliczenia z *garbage collectorem* można pokazać, że odzyskiwanie pamięci nie zmienia klasy złożoności czasowej algorytmu: jeśli obliczenie bez *garbage collectora* działa w czasie t, a w jego trakcie maksymalna żywa liczba komórek pamięci wynosi m, to istnieją takie stałe $\epsilon, \delta > 0$, że obliczenie to można przeprowadzić z *garbage collectorem* w czasie $(1 + \epsilon)t$ na maszynie zawierającej $(1 + \delta)m$ komórek pamięci. Na przykład listę n-elementową można posortować w czasie $O(n \log n)$ w pamięci zawierającej $3n$ komórek pamięci (mimo iż wykonamy przy tym $\Theta(n \log n)$ alokacji — większość zostanie odzyskana przez *garbage collector*).

Jeśli projektując algorytm imperatywny popełnimy błąd i spróbujemy skorzystać z (nieistniejącej już) poprzedniej wersji struktury danych, to błąd ten będzie zwykle krytyczny — *null pointer assignment*, *segmentation fault* itp. Jeśli ten sam algorytm wykonamy w środowisku czysto deklaratywnym, to jedynym efektem naszego błędu będzie to, że nie wszystkie wcześniejsze wersje struktury staną się nieużytkami i nie zostaną odzyskane przez *garbage collector*. Krytyczny błąd programu zamienia się więc w środowisku czysto deklaratywnym w zwiększenie zużycia pamięci. Nie dochodzi jednak do awarii programu (chyba, że zużycie to jest bardzo duże). Zapewnienie jednowątkowości w dostępie do danych jest bardzo trudne, a popełniane w tym zakresie błędy są częstą przyczyną krytycznych awarii programów. Język czysto deklaratywny gwarantuje, że do takich awarii nie dojdzie.

Skoro jednak z definicji wszystkie struktury danych w Haskellu są trwałe, to warto z tej trwałości korzystać.

Zadanie 2 (1 pkt). Zaprogramuj w Haskellu algorytm przeszukiwania wyczerpującego, który znajduje wszystkie rozstawienia n hetmanów na szachownicy $n \times n$:

```
queens :: Int -> [[Int]]
```

Lista liczb $[r_k, \ldots, r_n]$ reprezentuje rozstawienie, w którym hetmany stoją na polach $(k, r_k), \ldots, (n, r_n)$. Ustawienia hetmana w kolumnie $k-1$ i wierszu r_{k-1} odpowiada dodaniu głowy r_{k-1} do listy. Dla argumentu n funkcja powinna zwracać listę list n-elementowych, reprezentujących poprawne rozstawienia hetmanów.

Porównaj swoje rozwiązanie z rozwiązaniem imperatywnym, w którym rozstawienie hetmanów przechowujemy w n-elementowej ulotnej tablicy. Zauważ, że w Haskellu nie musimy się zajmować odtwarzaniem konfiguracji podczas próbowania kolejnych ustawień — nieudane ustawienia po prostu zapominamy.

Polecenie 2. Przeczytaj uważnie rozdziały 1 i 2 z książki: Chris Okasaki, *Purely Functional Data Structures*, CUP 1998 (wycinek książki jest dostępny na stronie zajęć).

Rozważmy drzewa binarne bez etykiet:

```
data BinTree = BinTree :/\: BinTree | BinTreeLeaf
```

Kompilator Haskella agresywnie optymalizuje przekład programu. Być może potrafi także zoptymalizować funkcję

```
complete :: Int -> BinTree
complete d
   | d < 0 = undefined
   | d = 0 = BinTreeLeaf
   | d > 0 = (complete $ d-1) :/\: (complete $ d-1)
```

która w naiwnej interpretacji działałaby w czasie i pamięci wykładniczej do

```
complete d
   | d < 0 = undefined
   | d = 0 = BinTreeLeaf
   | d > 0 = t :/\: t where t = complete (d-1)
```

która działa w czasie i pamięci liniowej i tworzy w pamięci nie pełne drzewo wysokości d, tylko d-wierzchołkowy graf acykliczny. Na potrzeby następnego zadania (jest to zadanie 2.5b z książki Chrisa Okasakiego) przyjmijmy jednak taką naiwną interpretację. Naszym zadaniem będzie jawne zadbanie o zmaksymalizowanie współdzielenia.

Zadanie 3 (1 pkt). Zaprogramuj funkcję

```
binTree :: Int -> BinTree
```

która wywołana z parametrem n:

- działa w czasie $O(\log n)$ (a zatem też alokuje $O(\log n)$ komórek pamięci),
- tworzy drzewo o n wierzchołkach wewnętrznych,
- drzewo to jest niemal zbalansowane: w każdym wierzchołku różnica rozmiarów lewego i prawego poddrzewa tego wierzchołka wynosi co najwyżej 1.

Napisz funkcję pomocniczą, która dla podanego n buduje parę drzew, które mają, odpowiednio, n i $n+1$ wierzchołków. Zauważ, że dla n parzystego mamy:

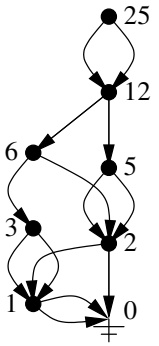
$$\begin{aligned} n &= \left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + \left(\left\lfloor \frac{n-1}{2} \right\rfloor + 1\right) + 1 \\ n+1 &= \left(\left\lfloor \frac{n-1}{2} \right\rfloor + 1\right) + \left(\left\lfloor \frac{n-1}{2} \right\rfloor + 1\right) + 1 \end{aligned}$$

zaś dla n nieparzystego mamy:

$$\begin{aligned} n &= \left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + \left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + 1 \\ n+1 &= \left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + \left(\left\lfloor \frac{n-1}{2} \right\rfloor + 1\right) + 1 \end{aligned}$$

zatem aby zbudować drzewa o n i $n+1$ wierzchołkach trzeba zbudować drzewa o $\lfloor \frac{n-1}{2} \rfloor$ oraz $\lfloor \frac{n-1}{2} \rfloor + 1$ wierzchołkach. I dalej przez indukcję...

Zauważ, że jeśli n jest nieparzyste i nie potrzebujemy drzewa o $n+1$ wierzchołkach, to nie potrzebujemy też drzewa o $\lfloor \frac{n-1}{2} \rfloor + 1$ wierchołkach (które, gdybyśmy je zbudowali, stałoby się nieużytkiem). Zoptymalizuj odpowiednio swoją funkcję tak, żeby nie budowała takich niepotrzebnych węzłów drzewa. Przykładowe drzewo o 25 wierzchołkach jest przedstawione na Rysunku 3.



$$
\begin{aligned}
t_{25} &= t_{12} \texttt{ :/\: } t_{12} \\
t_{12} &= t_6 \texttt{ :/\: } t_5 \\
t_6 &= t_3 \texttt{ :/\: } t_2 \\
t_5 &= t_2 \texttt{ :/\: } t_2 \\
t_3 &= t_1 \texttt{ :/\: } t_1 \\
t_2 &= t_1 \texttt{ :/\: } t_0 \\
t_1 &= t_0 \texttt{ :/\: } t_0 \\
t_0 &= \texttt{BinTreeLeaf}
\end{aligned}
$$

Rysunek 3: Prawie zbalansowane drzewo o 25 wierzchołkach reprezentowane w postaci 7-wierzchołkowego *dag*-a

Autorem kolejnego zadania (a raczej rozwiązania tego zadania) jest Maciek, któremu źle wytłumaczyłem zadanie Okasakiego.

Zadanie 4 (1 pkt). Zaprogramuj funkcję

```
binTreeLeaves :: Int -> BinTree
```

która wywołana z parametrem n:

- działa w czasie $O(\log n)$ (a zatem też alokuje $O(\log n)$ komórek pamięci),
- tworzy drzewo o n liściach.

Zauważ, że sumowanie liczby liści jest o tyle prostsze, że nie musimy pamiętać o dodaniu 1 za każdym razem, gdy tworzymy wierzchołek wewnętrzny (drzewo t_1 `:/\:` t_2 ma $n_1 + n_2$ liści, jeśli t_i ma n_i liści, dla $i = 1, 2$). Twoja funkcja powinna zbudować drzewo

$$t_{n_k} \texttt{ :/\: } (t_{n_{k-1}} \texttt{ :/\: } \ldots \texttt{ :/\: } t_{n_1})$$

gdzie $n_1, \ldots, n_k$ są pozycjami jedynek w binarnym rozwinięciu liczby n (tzn. $n = \sum_{i=1}^{k} 2^{n_i}$), zaś t_h są pełnymi drzewami binarnymi wysokości h (mają zatem 2^h liści). Oczywiście drzewa te powinny być w pamięci grafami acyklicznymi i współdzielić wierzchołki — jeśli $j < i$ to drzewo t_j jest poddrzewem drzewa t_i.

Kolejki to abstrakcyjne struktury danych posiadające funkcjonalność opisaną klasą `Queue`:

```
class Queue q where
   emptyQ :: q a
   isEmptyQ :: q a -> Bool
   put :: a -> q a -> q a
   get :: q a -> (a, q a)
   get q = (top q, pop q)
   top :: q a -> a
   top = fst . get
   pop :: q a -> q a
   pop = snd . get
```

Ponieważ kolejki realizują strategię LIFO, to musimy modyfikować tę strukturę z obu końców. Dlatego naiwna implementacja jest wyjątkowo nieefektywna:

```
instance Queue [] where
   emptyQ = []
   isEmptyQ = null
   put = (:)
   top = last
   pop = init
```

Zauważ, że koszt operacji `pop`, `top` i `get` jest proporcjonalny do liczby elementów kolejki, a nie stały, tak jak w implementacjach ulotnych. Cena za trwałość struktury danych jest tu zatem zbyt wysoka. Nieco lepszą implementację otrzymamy „przełamując" listę elementów kolejki w połowie.

Zadanie 5 (1 pkt). Zainstaluj typ

```
data SimpleQueue a = SimpleQueue { front :: [a], rear :: [a] }
```

w klasie `Queue` w taki sposób, aby lista

```
front q ++ reverse (rear q)
```

zawierała elementy kolejki od pierwszego do ostatniego. Dzięki temu operacje `put` i `get` będą mogły być wykonywane w czasie stałym, no chyba że `front q` będzie pusty... Przyjmij zatem niezmiennik: jeśli `rear q` jest niepusty, to `front q` też jest niepusty. Zauważ, że kolejki `SimpleQueue [] rs` oraz `SimpleQueue (reverse rs) []` zawierają te same elementy w tej samej kolejności. Gdy zatem dojdzie do naruszenia niezmiennika, to przywróć go korzystając z przytoczonej równości.

Dla kolejek z powyższego zadania zwykle dowodzi się, że operacje `pop`, `top` i `get` działają w zamortyzowanym czasie stałym: operacja odwracania `rear q` działa w czasie proporcjonalnym do $n =$ `length (rear q)`. Była jednak poprzedzona n operacjami `put`. Obciążając każdą z tych operacji (działających w czasie stałym) dodatkowym kosztem jednostkowym amortyzujemy koszt odwracania. Przytoczony dowód zakłada *implicite*, że obliczenie jest jednowątkowe. Jeśli istotnie korzystamy z trwałości tej struktury danych, to schemat amortyzacji się załamuje. Możemy bowiem wykonać $n + 1$ operacji `put` zaczynając od `emptyQ`, a następnie n krotnie wykonać na otrzymanej wersji struktury operację `pop`. Każda z nich będzie kosztować n. Zatem wykonaliśmy $2n + 1$ operacji, które łącznie kosztowały $n^2 + n + 1$. Nie możemy zatem zamortyzować ich do czasu stałego.

Leniwe wartościowanie i struktury nieskończone

W teorii języków programowania znacznie programu zapisuje się zwykle ujmując go w podwójne nawiasy kwadratowe. Jeżeli funkcja w programie ma typ $f :: \sigma \to \tau$, to jej znaczenie $[\![f]\!]$ jest funkcją (w matematycznym sensie) ze zbioru $[\![\sigma]\!]$ w zbiór $[\![\tau]\!]$. Na przykład możemy przyjąć, że $[\![\texttt{Integer}]\!] = \mathbb{Z}$, a wtedy w zasięgu definicji

```
f :: Integer -> Integer
f n = n+1
```

mamy $[\![\texttt{f}]\!] : \mathbb{Z} \to \mathbb{Z}$ oraz $[\![\texttt{f}]\!](n) = n + 1$, dla $n \in \mathbb{Z}$.

Problem pojawia się wówczas, gdy funkcja w programie nie zwraca wartości dla każdego możliwego argumentu. Rozważmy np. funkcję

```
silnia :: Integer -> Integer
silnia 0 = 1
silnia n = n * silnia (n-1)
```

Funkcja (w matematycznym sensie) $[\![\texttt{silnia}]\!]$ nie jest określona np. dla argumentu -1 (gdyż dla takiego argumentu obliczenie się zapętla). Moglibyśmy rozważać funkcje częściowe, ale z wielu powodów jest to niewygodne. Wygodniej dodać do interpretacji każdego typu dodatkową wartość $\bot$, która oznacza „brak wartości". Niech zatem $[\![\texttt{Integer}]\!] = \mathbb{Z} \cup \{\bot\}$. Możemy teraz napisać

$$[\![\texttt{silnia}]\!](n) \quad = \quad \begin{cases} n!, & \text{gdy } n \geqslant 0, \\ \bot, & \text{w p.p.} \end{cases}$$

dla $n \in \mathbb{Z}$. Oczywiście teraz $\bot$ należy też do dziedziny funkcji $[\![\texttt{silnia}]\!]$, musimy ją zatem zdefiniować dla tej wartości. Pytanie, czym jest $[\![\texttt{silnia}]\!](\bot)$, to pytanie, jaki będzie wynik wywołania funkcji `silnia` dla argumentu, którego obliczenie nie dostarcza wartości (zapętla się, kończy się zerwaniem programu na skutek błędu itp.). W tym przypadku $[\![\texttt{silnia}]\!](\bot) = \bot$, ale nie zawsze tak musi być.

W większości języków programowania przed wywołaniem funkcji oblicza się wartość argumentu faktycznego, a następnie tę wartość wiąże się z argumentem formalnym. W pamięci programu istnieją tylko obliczone wartości. Taki sposób przekazywania parametrów nazywa się *przekazywaniem przez wartość* (*call by value*). W rachunku lambda taką strategię nazywa się *gorliwym wartościowaniem* (*eager evaluation*). Można też przyjąć inną strategię: nie obliczamy parametru faktycznego, tylko nieobliczone *wyrażenie* wiążemy z argumentem formalnym. Wymaga to możliwości przechowywania w pamięci nie tylko wartości, ale też nieobliczonych wyrażeń. Parametr zostaje obliczony podczas wywałania funkcji dopiero wtedy, gdy będzie potrzebna jego wartość. Taki sposób przekazywania parametrów nazywa się *przekazywaniem przez nazwę* (*call by name*). W rachunku lambda taką strategię nazywa się *leniwym wartościowaniem* (*lazy evaluation*). Wiąże się z nim spora nieefektywność. Jeśli wartość parametru będzie podczas obliczania funkcji potrzebna wielokrotnie, to ten argument będzie wielokrotnie obliczany. W językach takich jak Haskell, w których z definicji nie ma skutków ubocznych a wszystkie struktury danych są trwałe, każde obliczenie parametru faktycznego dostarczy zawsze tę samą wartość. Możemy zatem obliczyć wyrażenie jednokrotnie i następnie zapamiętać obliczoną wartość. Taki mechanizm nazywa się *spamiętywaniem* (*memoization*), a przekazywanie przez nazwę wraz ze spamiętywaniem nazywa się *call by need*. Taki sposób przekazywania parametrów przyjęto w Haskellu. Zauważmy, że spamiętywanie, to modyfikowanie istniejących w pamięci struktur danych (co ogólnie zostało zabronione ze względu na postulat trwałości struktur), ale w taki sposób, że zamieniamy nieobliczone wyrażenie na obliczone wyrażenie o tej samej wartości. Struktury danych zmieniają się zatem w pamięci, ale nie jest to obserwowalne na poziomie języka.

Skoro w Haskellu nie wolno obliczać argumentów, jeśli ich wartości nie są niezbędne do wyznaczenia wartości funkcji, to istnieją takie funkcje f, dla których $[\![f]\!](\bot) \neq \bot$, np.

```
const1 :: Integer -> Integer
const1 _ = 1
```

Mamy $[\![\texttt{const1}]\!](n) = 1$ dla $n \in \mathbb{Z} \cup \{\bot\}$, w szczególności $[\![\texttt{const1}]\!](\bot) = 1$. Takie funkcje f, że $[\![f]\!](\bot) \neq \bot$ nazywamy funkcjami *non-strict*. W językach używających strategii *call by value* nie ma takich funkcji. Te języki nazywają się *strict*. Zauważmy, że również matematyka jest *strict*: jeśli parametr x nie należy do dziedziny funkcji częściowej g, to nie należy też do złożenia fg, nawet jeśli funkcja f jest stała. Dlatego potrzebowaliśmy dodać $\bot$ do zbioru wartości funkcji, by móc używać matematyki do opisu funkcji *non-strict*.

W Haskellu są jednak funkcje, które są *strict*. Na przykład operacje arytmetyczne, takie jak `(+)` nie mogą dostarczyć wyniku zanim ich argumenty nie zostaną obliczone do wartości numerycznych. Dopasowanie wzorca wymaga obliczenia wartości przynajmniej na tyle, by argument faktyczny dało się z tym wzorcem porównać — ale ani kroku więcej. Dlatego np. funkcja

```
f :: [Integer] -> Integer
f (x:y:_) = x+y
```

żąda obliczenia pierwszych dwóch elementów listy będącej jej argumentem (trzeci element może nawet nie istnieć). Dlatego obliczenie `f [1..]` zakończy się powodzeniem i dostarczy wartość 3, podobnie jak obliczenie `f (1:2:undefined)`. Zmienna `undefined` jest związana z wyrażeniem, które w razie próby obliczenia zrywa działanie programu. Wartością wyrażenia `undefined` jest $\bot$.

Ponieważ wartości możemy reprezentować za pomocą wyrażeń haskellowych, to często opuszczamy nawiasy $[\![$ i $]\!]$ i piszemy np. $f\,\bot = \bot$ zamiast $[\![f]\!]\bot = \bot$. Mieszanie wyrażeń z języka z ich denotacjami zwykle nie prowadzi do nieporozumień.

Zauważmy, że funkcja `f` z powyższego przykładu zależy jedynie od pierwszych dwóch elementów listy będącej jej argumentem i np. `f (1:2:`*cokolwiek*`) = 3`. Często używamy wartości $\bot$ na oznaczenie wyrażenia, którego wartości nie znamy i nie zamierzamy jej obliczać. Dlatego piszemy `f (1:2:`$\bot$`) = 3`. Lista *częściowa* `1:2:`$\bot$ jest *aproksymacją* dowolnej listy `1:2:`*cokolwiek*. Formalnie możemy na listach wprowadzić porządek częściowy, w którym $\bot$ jest elementem najmniejszym (tj. $\bot \leqslant x$ dla dowolnego x) i jeśli $xs \leqslant ys$, to $x\texttt{:}xs \leqslant x\texttt{:}ys$ dla dowolnego x. Na przykład `1:`$\bot \leqslant$ `1:2:`$\bot$. Wtedy `1:2:`$\bot$ jest elementem minimalnym w zbiorze wszystkich list, na których obliczenie funkcji `f` przebiega tak samo.

Aproksymacje za pomocą list częściowych przydają się do opisu wartości wyrażeń, które mogą się obliczać w nieskończoność. Na przykład listy $\bot$, `1:`$\bot$, `1:2:`$\bot$, `1:2:3:`$\bot$ są skończonymi aproksymacjami

nieskończonej listy `[1..]`. Lista ta jest granicą powyższego ciągu aproksymacji (formalnie jest kresem górnym tego łańcucha). Dzięki temu możemy listy nieskończone przybliżać za pomocą list częściowych. Dla dowolnego skończonego obliczenia na liście `[1..]` zawsze znajdziemy dostatecznie długą skończoną aproksymację $1{:}2{:}3{:}\ldots{:}n{:}\bot$ na której to obliczenie będzie działać tak samo, jak na liście `[1..]`.

Leniwe wartościowanie pozwala definiować wartości *potencjalnie* nieskończone, jak np. w funkcji

```
enumFrom :: Integer -> Integer
enumFrom n = n : enumFrom (n+1)
```

(definicja haskellowa jest ogólniejsza — obejmuje całą klasę `Enum`). Dla tej funkcji mamy też wygodny cukier syntaktyczny — notację $[n..]$.

Dopóki wykorzystujemy wynik funkcji `enumFrom` w wyrażeniu, które wymaga obliczenia jedynie skończonej liczby elementów listy przez nią produkowanej, to nieskończoność tej listy pozostaje potencjalna (pracujemy jedynie na aproksymacji $n{:}(n+1){:}\ldots m{:}\bot$). Jeśli obliczenie wymaga wyznaczenia nieskończonej liczby elementów tej listy, wówczas nieskończoność z potencjalnej zamienia się w *faktyczną* i obliczenie się zapętla.[2] Przykładem takiego wyrażenia jest `length [1..]`.

Ponieważ nigdy bez potrzeby nie obliczamy żadnego wyrażenia, to możemy definiować zależności rekurencyjne także dla wartości, które nie są funkcjami, np.

```
xs :: [Integer]
xs = 1 : xs
```

W języku *strict* (gorliwym) próbowano by najpierw wyznaczyć wartość wyrażenia `1:xs`, a następnie związać wyliczoną wartość z nazwą `xs`. Zakończyłoby się to błędem, gdyż w chwili obliczenia wyrażenia `1:xs` wartość `xs` nie jest jeszcze znana. W języku *non-strict* związujemy *wyrażenie* `1:xs` z nazwą `xs`. Dopiero w chwili próby obliczenia pierwszego elementu tej listy jest wywoływany konstruktor `(:)`, który alokuje pamięć dla swoich argumentów, ale czyni to *przed* ich wyliczeniem (bo konstruktory też są obliczane leniwie). Dopiero potem, w miarę potrzeby obliczamy jego argumenty `1` oraz `xs`. Ale wówczas nazwa `xs` jest już związana z policzoną i spamiętaną wartością `1:xs`. Dostajemy zatem w pamięci strukturę cykliczną.

Trzeba pamiętać, żeby taka rekursja była dobrze ufundowana. Np. deklaracja

```
n :: Integer
n = n + 1
```

jest poprawna, ale niezbyt użyteczna, gdyż próba obliczenia wartości `n` zakończy się zapętleniem obliczeń (zatem $\mathtt{n} = \bot$).

Struktury potencjalnie nieskończone pozwalają bardzo zwięźle i elegancko zapisywać wiele algorytmów. Jeśli np. chcielibyśmy wyrównać pewien napis `str` do `n` znaków dodając na jego początku `n - length str` spacji, to możemy napisać np.

```
pad_left :: Int -> String -> String
pad_left n str = take (max 0 $ n - length str) (cycle " ") ++ str
```

Funkcja

```
cycle :: [a] -> [a]
cycle xs = xs ++ cycle xs
```

tworzy nieskończoną listę powtarzających się ciągów elementów. Inne rozwiązanie, obcinające napis `str` gdy `length str` $>$ `n` jest następujące:

```
pad_left n = reverse . take n . (++ cycle " ") . reverse
```

Sito Eratostenesa polega na utworzeniu nieskończonego ciągu `[2..]`, a następnie na powtarzaniu następującej operacji: z otrzymanego ciągu liczb wybieramy pierwszą liczbę x. Jest ona kolejną liczbą pierwszą. Następnie usuwamy z tego ciągu wszystkie wielokrotności x. Czynności te powtarzamy w nieskończoność. Funkcja, która dla podanego ciągu `x:xs` wykonuje opisaną operację wyraża się następująco:

[2] Por. pojęcia nieskończoności potencjalnej i faktycznej (*potential and actual infinity*) w metamatematyce.

```
\ (x:xs) -> [ y | y <- xs, y `mod` x /= 0 ]
```

Funkcję tę powinniśmy iterować w nieskończoność poczynając od `[2..]`. W Preludium Standardowym mamy funkcję

```
iterate :: (a -> a) -> a -> [a]
iterate f c = c : iterate f c
```

W skrócie `iterate f c = [c, f c, f(f c), f(f(f c)),...]`. Zatem

```
primes :: [Integer]
primes = map head $ iterate (\ (x:xs) -> [ y | y <- xs, y `mod` x /= 0 ]) [2..]
```

jest listą wszystkich liczb pierwszych utworzonych przez algorytm sita Eratostenesa.

Zadanie 6 (1 pkt). Listę wszystkich liczb pierwszych też możemy zdefiniować zapomocą zależności rekurencyjnej podobnej do poniższej:

```
primes :: [Integer]
primes = [ p | p <- [2..], and [ p `mod` q /= 0 | q <- primes, q /= p ]]
```

Definicja ta mówi, że liczby pierwsze, to te spośród liczb `[2..]` które nie dzielą się przez żadną liczbę pierwszą różną od nich samych. Oczywiście powyższa rekursja nie jest dobrze ufundowana i próba obliczenia choćby głowy tej listy zakończy się zapętleniem. Zauważ, że musimy mieć choćby jedną liczbę pierwszą podaną *explicite*, inaczej mamy problem jajka i kury. Zauważ też, że nie musimy sprawdzać wszystkich liczb pierwszych `q <- primes`, wystarczy tylko te, które są nie większe niż $\sqrt{\texttt{p}}$, tj. takie, że `q*q <= p`. Popraw powyższą definicję.

Zadanie 7 (1 pkt). Jeśli `fib = [`$f_1, f_2, f_3, \ldots$`] :: [Integer]` jest listą wszystkich liczb Fibonacciego, to $f_{i+2} = f_{i+1} + f_i$ dla dowolnego i. Zatem jeśli utworzymy listę liczb poprzez zsumowanie odpowiadających sobie elementów list `fib` oraz `tail fib`, to otrzymamy listę `fib` bez dwóch pierwszych elementów. Napisz rekurencyjną definicję listy `fib` wykorzystującą tę zależność. Do sumowania elementów list możesz użyć funkcji `zipWith`. Zadbaj o to, by rekursja była dobrze ufundowana.

Zadanie 8 (1 pkt). Zaprogramuj funkcję, która scala dwie różnowartościowe listy posortowane w jedną różnowartościową listę posortowaną (usuwa zatem duplikaty):

```
(<+>) :: Ord a => [a] -> [a] -> [a]
```

i wykorzystaj ją do rozwiązania tzw. problemu 2-3-5 Dijkstry: zdefiniuj nieskończony rosnący ciąg liczb całkowitych

```
d235 :: [Integer]
```

zawierający liczbę 1 i taki, że jeśli n jest elementem tego ciągu, to są nimi też $2n$, $3n$ i $5n$.

Zadanie 9 (1 pkt). Niech

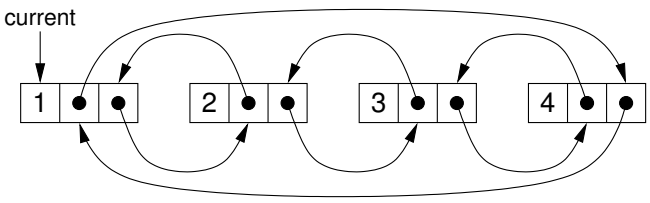
```
data BTree a = BNode (BTree a) a (BTree a) | BLeaf
```

Na wzór listy `[1..]` zbuduj nieskończone drzewo binarne typu `BTree Int`, które zawiera wszystkie liczby naturalne dodatnie. Jeśli etykietą wierzchołka jest $n = (1b_{k-1}\ldots b_0)_2$, to ciąg cyfr binarnych $(b_{k-1}, \ldots, b_0)$ jest ścieżką od tego wierzchołka do korzenia. W szczególności korzeń ma etykietę 1.

Zadanie 10 (1 pkt). Niech

```
data RoseTree a = RNode a [RoseTree a]
```

będzie typem drzew o zmiennej arności wierzchołków. Zauważ, że nie mamy tu drzewa pustego, a skoro w Haskellu listy mogą być nieskończone, to drzewo takie może być nieskończone zarówno wszerz, jak i w głąb (tj. każdy wierzchołek może mieć nieskończenie wiele synów, a ścieżki w tym drzewie mogą być nieskończonej długości). Na wzór listy cyklicznej



Rysunek 4: Czteroelementowa cyklista `fromList [1..4]`

```
xs :: [Int]
xs = 1 : xs
```

zaprogramuj drzewa typów `BTree Int` i `RoseTree Int`. Pierwsze powinno mieć we wszystkich wierzchołach etykietę `1`, a wszystkie ścieżki w tym drzewie powinny być nieskończonej długości. W pamięci powinien znajdować się pojedyczy konstruktor `BNode`. Drugie powinno być nieskończone zarówno wszerz, jak i w głąb. W pamięci powinien znajdować się jeden konstruktor `RNode` i jeden konstruktor `(:)`.

Zadanie 11 (1 pkt). W tym zadaniu rozważamy funkcje, które tworzą napisowe reprezentacje skończonych aproksymacji potencjalnie nieskończonych struktur danych. Napisz funkcję

```
showFragList :: Show a => Int -> [a] -> String
```

wypisującą listę podobnie do standardowej metody `show`, z tą jednak różnicą, że dla podanego parametru n funkcja powinna zastąpić n-ty i wszystkie kolejne elementy listy pojedynczym znakiem Unicode „…” (`U+2026`). Znak „…” odgrywa tu rolę wartości $\bot$ — zastępuje potencjalnie nieskończony ogon listy informacją, że „jest tu coś, ale nie wiadomo co.” Na przykład

```
showFragList 3 [1,2,3,4,5,6] = "[1,2,3,…]"
let xs = 1:xs in showFragList 2 xs = "[1,1,…]"
```

Dla drzew należacych do klasy `BT` z poprzedniej listy oraz dla drzew typu `RoseTree` napisz funkcje

```
showFragTree :: (BT t, Show a) => Int -> t a -> String
showFragRose :: Show a => Int -> RoseTree a -> String
```

które powinny działać podobnie do standardowej (derywowanej) metody `show` dla tych drzew. Dla podanego parametru n pierwsza z nich powinna wypisywać podrzewo, którego ścieżka do korzenia ma długość n, w postaci pojedynczego znaku Unicode „…”. Druga powinna w ten sam sposób zamieniać n-tego i każdego następnego potomka (użyj funkcji `showFragList`).

Zadanie 12 (1 pkt). *Dwukierunkowa lista cykliczna*, to struktura danych o dostępie sekwencyjnym, w której każdy element posiada wskaźnik do elementu poprzedniego oraz następnego i elementy te tworzą cykl (być może nieskończony). Przykładowa cyklista jest przedstawiona na Rysunku 4. Uwaga: utworzenie struktury cyklicznej w pamięci wymaga użycia rekursji na danych! Budowanie takiej struktury za pomocą *funkcji rekurencyjnych* prowadzi do utworzenia w pamięci *dag*-a, tj. obliczenie `backward . forward` nie prowadzi na powrót do wierzchołka, z którego wyszliśmy, tylko do nowo utworzonego wierzchołka posiadającego tę samą etykietę, co oryginalny. Chcemy uniknąć takiej alokacji pamięci! Ponieważ w trwałej strukturze danych modyfikacja elementu pociąga za sobą konieczność skopiowania wszystkich elementów, z których dany element jest osiągalny, więc modyfikacja dwukierunkowej listy cyklicznej zawsze prowadzi do konieczności skopiowania całej struktury, co jest bardzo nieefektywne. Ograniczymy się więc do zaprogramowania selektorów i obserwatorów, tj. operacji, które nie modyfikują struktury. Niech zatem

```
data Cyclist a = Elem (Cyclist a) a (Cyclist a)
```

Zdefiniuj funkcje

```
fromList :: [a] -> Cyclist a
forward, backward :: Cyclist a -> Cyclist a
label :: Cyclist a -> a
```

Funkcja `fromList` tworzy cyklistę zawierającą elementy podanej listy. Bieżącym elementem jest pierwszy element listy. Funkcje `forward` i `backward` przemieszczają wskaźnik elementu bieżącego, odpowiednio, w przód i w tył, a `label` ujawnia etykietę bieżącego elementu, np.

```
label . forward . forward . forward . backward . forward . forward $ fromList [1,2,3]
```

ma wartość 2. Jeśli xs jest listą nieskończoną, to

$$\texttt{backward . fromlist \$}\ xs \;=\; \bot.$$

Zadanie 13 (1 pkt). Zdefiniuj nieskończoną cyklistę

```
enumInts :: Cyclist Integer
```

która zawiera wszystkie liczby całkowite w naturalnym porządku i której bieżącym elementem jest zero.

Zadanie 14 (2 pkt). Zaprogramuj funkcję

```
knight :: (Int,Int) -> (Int,Int) -> [[Int]]
```

wyszukującą drogę konika szachowego. Wywołanie `knight (n,m) (i,j)` powinno zwrócić listę tras konika startującego z pola `(i,j)` na planszy o rozmiarach `n` × `m`, gdzie $1 \leqslant$ `i` $\leqslant$ `n` i $1 \leqslant$ `j` $\leqslant$ `m`. Aby przyspieszyć wyszukiwanie trasy przygotuj najpierw w pamięci graf (cykliczny!) o wierzchołkach będących polami planszy i krawędziach odpowiadających legalnym ruchom konika. Wykorzystaj przy tym typ

```
data Field = Field Int [Field]
```

reprezentujący pola. `Field k xs` oznacza pole o numerze `k = m * (i-1) + (j-1)`, a `xs` jest listą pól, na które konik może przejść w jednym kroku (lista ta ma od 2 do 8 elementów). Jedyna trudność w przygotowaniu takiej planszy polega na właściwym zacykleniu wskaźników umieszczonych na listach. Ta faza preprocesingu nie musi być wykonana efektywnie. Do szybkiego sprawdzania, które wierzchołki zostały już odwiedzone, przechowuj numery odwiedzonych pól w strukturze `Data.IntSet`.