Kurs języka Haskell

Notatki zamiast wykładu i lista zadań na pracownię nr 5

Do zgłoszenia w SKOS-ie do 10 kwietnia 2020

Ze względu na użycie modułów[1] rozwiązania zadań z bieżącej listy należy umieścić w kliku plikach z rozszerzeniem `.hs`. Każdy plik powinien być udokumentowany w `haddock`-u, w szczególności mieć na początku znacznik `haddock`-a `Copyright:` zawierający imię i nazwisko autora rozwiązania. Każdemu plikowi `*.hs` powinien towarzyszyć plik `*.pdf` z dokumentacją wygenerowany za pomocą `haddock`-a. Funkcje z zadań, w których nie żąda się utworzenia specjalnych modułów należy umieścić w module `Lista_5` (a zatem w pliku `Lista_5.hs`). W SKOS-ie proszę zgłosić pojedyncze archiwum o nazwie *`Imie`*`_`*`Nazwisko`*`_05.tar.bz2`, gdzie *`Imie`* i *`Nazwisko`*, to odpowiednio imię i nazwisko autora rozwiązań zapisane bez znaków diakrytycznych. Archiwum `tar` powinno być skompresowane programem `bzip2` i zawierać pojedynczy katalog o nazwie *`Imie`*`_`*`Nazwisko`*`_05` zawierający pliki `*.hs` i `*.pdf` z rozwiązaniami. Bardzo proszę o pilne przestrzeganie formatu zgłaszania rozwiązań.

Polecenie 1. Przejrzyj dokumentację programu `Haddock` dostępną pod adresem

`https://haskell-haddock.readthedocs.io/`

Jeszcze o językach *non-strict*

Typy danych możemy rozumieć jako *zbiory* wartości. Na przykład na poprzedniej liście przyjęliśmy, że w Haskellu $[\![\texttt{Integer}]\!] = \mathbb{Z} \cup \{\bot\}$. Dla typów zdefiniowanych indukcyjnie do podstawy indukcji dodajemy $\bot$ i przez indukcję strukturalną tworzymy zbiór skończonych i częściowych wartości tego typu. Brakuje w nim wartości, które reprezentują struktury potencjalnie nieskończone. Te wartości są granicami ciągów częściowych aproksymacji (formalnie są kresami górnymi łańcuchów względem relacji porównania wprowadzonej na poprzedniej liście). Rozważmy liczebniki Peano:

```
data Nat = Zero | Succ Nat
```

Mamy zatem:

- $\texttt{Zero} \in [\![\texttt{Nat}]\!]$,
- $\bot :: [\![\texttt{Nat}]\!]$,
- jeśli $n :: [\![\texttt{Nat}]\!]$, to $\texttt{Succ}\, n :: [\![\texttt{Nat}]\!]$,
- jeśli $(n_k)_{k\in\mathbb{N}}$ jest wstępującym ciągiem (łańcuchem) elementów $[\![\texttt{Nat}]\!]$, to $\sup_{k\in\mathbb{N}} n_k \in [\![\texttt{Nat}]\!]$.

W przypadku typu `Nat` mamy tylko jeden ciąg wstępujący: $(\bot, \texttt{Succ}\,\bot, \texttt{Succ}(\texttt{Succ}\,\bot), \ldots)$. Jest to ciąg częściowych aproksymacji wartości wprowadzonej np. definicją[2]

```
infty :: Nat
infty = Succ infty
```

[1] Patrz podrozdział *6.2.1. Modules vs. filenames* dokumentacji kompilatora *Glasgow Haskell Compiler User's Guide*.

[2] Zauważmy, że w matematyce nieskończoność definiuje się właśnie w ten sposób: zbiór jest nieskończony, jeśli powiększony o jeden element nie zmienia swojej mocy. Dla zbiorów przeliczalnych: $\aleph_0 + 1 = \aleph_0$. Definicja ta jest zatem faktycznie definicją nieskończoności w matematycznym sensie. Ma też cechy takiej nieskończoności. W zasięgu definicji

```
Zero <<= _ = True
(Succ n) <<= (Succ m) = n <<= m
```

mamy $\texttt{Succ}^n$ `Zero <<= infty` = `True`, tj. `infty` jest większe niż każda liczba skończona.

Nazwijmy kres górny tego ciągu ∞ (więc $[\![\texttt{infty}]\!] = \infty$). Wtedy

$$[\![\texttt{Nat}]\!] \quad = \quad \{\texttt{Succ}^n\,\texttt{Zero}, \texttt{Succ}^n\,\bot, \infty\}.$$

W języku *strict* mielibyśmy jedynie

$$[\![\texttt{Nat}]\!] \quad = \quad \{\texttt{Succ}^n\,\texttt{Zero}\}.$$

Intuicja jest taka, że w językach *strict* typy danych zawierają jedynie wartości skończone, zaś w językach *non-strict* także nieskończone oraz ich częściowe aproksymacje. Więcej o tym powie Maciek.

Polecenie 2. Zauważmy, że definicja `infty` przypomina definicję

```
ones :: [Integer]
ones = 1 : ones
```

Zastanów się, jakie elementy zawiera zbiór $[\![\texttt{[Integer]}]\!]$. Czy on powinien być przeliczalny? A jeśli tak, jakie kresy łańcuchów wybrać (mamy tu *continuum* łańcuchów)?

W języku *non-strict* traktujemy $\bot$ jako dodatkową podstawę indukcji strukturalnej. Należy jednak pamiętać, że nie jest to *konstruktor* i że nie możemy wykorzystywać go we wzorcach. Rozważmy jeszcze typ

```
data Bool = True | False
```

Mamy

$$[\![\texttt{Bool}]\!] \quad = \quad \{\texttt{True}, \texttt{False}, \bot\}$$

W Haskellu mamy zatem *trzy* wartości logiczne: „prawda”, „fałsz” i „nie wiadomo” (jeszcze nie obliczono, obliczenie się zapętliło, obliczenie zakończyło się błędem itd.).

W językach *strict* zwykle przekrój wartości dwóch różnych typów jest pusty. W Haskellu natomiast $[\![\texttt{Integer}]\!] \cap [\![\texttt{Bool}]\!] = \{\bot\}$. Ogólniej, w Haskellu przekrój wartości *wszystkich* typów zawiera jeden, jedyny element $\bot$. Zauważmy, że jeśli typem wyrażenia jest $\forall a.a$ (czyli wyrażenie to jest każdego typu, a więc jego wartość należy do zbioru wartości każdego typu), to możemy stąd wywnioskować, że jego wartością jest $\bot$. Mamy np.

```
undefined :: a
error :: String -> a
f :: Integer -> a
f n = f (n+1)
```

Zatem $[\![\texttt{undefined}]\!] = \bot$, a wynikiem wywołania funkcyj `error` i `f` na dowolnym argumencie jest $\bot$. Obliczenie `error str` zakończy się zerwaniem programu, zaś `f n` zakończy się zapętleniem. Wspólną cechą tych obliczeń jest to, że *nie zakończą się wyznaczeniem żadnej skończonej wartości.*

W Haskellu dopasowanie do nietrywialnego wzorca, tj., z grubsza, innego niż pojedyncza zmienna i `_` jest *strict* (upraszczamy tu nieco). Aby definiowane funkcje były maksymalnie *non-strict* musimy uważać, gdzie go stosujemy. Funkcję `unzip` możemy zdefiniować naiwnie następująco:

```
unzip' :: [(a,b)] -> ([a],[b])
unzip' [] = []
unzip' ((x,y):ps) = (x:xs,y:ys) where (xs,ys) = unzip' ps
```

Zauważmy, że `unzip` jest *strict* (jak każda funkcja zdefiniowana za pomocą dopasowania do nietrywialnego wzorca). Jej bardziej „rozleniwioną” wersją jest

```
unzip ps = (map fst ps, map snd ps)
```

Teraz `unzip` $\perp = (\perp, \perp) \neq \perp$ i `unzip` jest *non-strict.*

Niekiedy można „rozleniwić” funkcję za pomocą *explicit irrefutable patterns.* Są to nietrywialne wzorce, w których dopasowanie jest odroczone aż do momentu, gdy potrzebujemy obliczyć wartości, które się do nich dopasowały. Na przykład[3]

```
f' (x:y:_) = (x,y)
```

jest *strict*, podczas gdy

```
f ~(x:y:_) = (x,y)
```

jest *non-strict* i jest równoważne deklaracji

```
f z = (case z of { x:y:_ -> x }, case z of { x:y:_ -> y })
```

Mamy zatem `f` $\perp = (\perp, \perp) \neq \perp$. Ponieważ dopasowanie następuje dopiero podczas obliczania ciała klauzuli, to dopasowanie do *irrefutable pattern* (jak sama nazwa wskazuje) zawsze kończy się powodzeniem. Taki wzorzec powinien więc wystąpić tylko w ostatniej (a przeważnie jedynej) klauzuli.

Polecenie 3. Przeczytaj podrozdziały *3.17.2 Informal Semantics of Pattern Matching*, *3.12 Let Expressions* oraz *4.4.3.2 Pattern bindings* definicji języka *Haskell 2010 Language Report.* Zwróć szczególną uwagę na definicję *irrefutable patterns.*

Polecenie 4. Zastanów się, które z poniższych funkcji są *strict*:

- `map`
- `map id`
- `map (const 1)`

Jaką wartość ma wyrażenie:

- `(\ ~(x:_:_) -> x) (1:undefined)`

Wyrażenie jest w *postaci normalnej* (NF), jeśli zostało już całkowicie obliczone (nie można zastosować do niego żadnych reguł redukcji). Wyrażenie jest w *słabej czołowej postaci normalnej* (WHNF), jeśli w jego korzeniu znajduje się abstrakcja funkcyjna lub konstruktor. Obliczenie wyrażenia w Haskellu zwykle zatrzymuje się po osiągnięciu słabej czołowej postaci normalnej. Zauważmy, że w przypadku list obliczenie zatrzyma się po ustaleniu, czy lista jest pusta, czy niepusta. Ani głowa, ani ogon listy nie będą obliczane. Propagacja obliczenia do głowy i ogona może nastąpić dopiero wówczas, gdy wyizolujemy głowę i ogon za pomocą dopasowania wzorca. Jeśli w korzeniu wyrażenia jest abstrakcja funkcyjna (lambda abstrakcja), to ciało funkcji nigdy nie jest obliczane — obliczanie ciała funkcji rozpoczyna się dopiero po związaniu parametru formalnego z faktycznym (tj. po wywołaniu funkcji z podanym argumentem). Wyrażenia, w których korzeniu jest konstruktor można jednak redukować *głębiej.*

W Haskellu operatorem wymuszającym obliczenie do słabej czołowej postaci normalnej jest

```
seq :: a -> b -> b
infixr 0 `seq`
```

Wartością wyrażenia `seq a` jest funkcja identyczności, tak jak w definicji

```
a `seq` b = b
```

a skutkiem ubocznym jej aplikacji do argumentu — wyznaczenie słabej czołowej postaci normalnej wyrażenia `a`.

Polecenie 5. Zastanów się, jakie wartości mają wyrażenia:

- ``(seq undefined) `seq` 1``

[3]Bardziej gorliwe wersje funkcji zwykle w Haskellu zawierają w nazwie apostrof, por. `foldl` i `foldl'`.

- `(undefined:undefined) `seq` 1`
- `undefined `seq` 1`

Preludium Standardowe definiuje też operatory leniwej i gorliwej aplikacji:

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

Pamiętajmy, że gorliwa aplikacja oblicza argument jedynie do słabej czołowej postaci normalnej!

Polecenie 6. Zastanów się, jakie wartości mają poniższe wyrażenia:

- `const 1 $ undefined`
- `const 1 $! undefined`
- `const 1 $! undefined:undefined`

Polecenie 7. Przeczytaj podrozdział *4.2.1 Algebraic Datatype Declarations* definicji języka *Haskell 2010 Language Report*. Zwróć szczególną uwagę na definicję *strictness flags*.

Strictness flags są np. wykorzystane w definicjach typów liczb wymiernych i zespolonych:

```
data Ratio a = !a :% !a
data Complex a = !a :+ !a
```

Polecenie 8. Przeczytaj podrozdział *11.29. Bang patterns and Strict Haskell* dokumentacji kompilatora *Glasgow Haskell Compiler User's Guide*.

Chcielibyśmy mieć funkcje

```
deepseq :: a -> b -> b
($!!) :: (a -> b) -> a -> b
```

podobnie do `seq` i `($!)`, ale powodujące obliczenie wyrażenia nie do słabej czołowej postaci normalnej, tylko do pełnej postaci normalnej.

Polecenie 9. Zastanów się, czemu nie można zdefiniować takich funkcji. Weź pod uwagę np. typy zawierające funkcje.

Wszystko, co możemy zrobić, to postąpić podobnie jak z klasą `Show`, tj. zdefiniować takie funkcje dla możliwie dużej klasy typów.

Zadanie 1 (1 pkt). Niech

```
class NFData a where
   rnf :: a -> ()
```

Nazwa klasy, to skrót od *normal form data*, zaś metody — *reduce to normal form*. Zauważ, że dla typów prostych, takich jak `Integer`, których dane nie mają wewnętrznej struktury, słaba czołowa postać normalna jest tożsama z postacią normalną. Zainstaluj w klasie `NFData`:

- typy proste należące do klasy `Num` (zauważ, że niektóre z nich mają wewnętrzną strukturę, np. `Rational` — czy nie będzie z tym kłopotu?),
- listy elementów typów należących do `NFData`,
- krotki elementów typów należących do `NFData`.

Zaprogramuj też funkcje

```
deepseq :: a -> b -> b
($!!) :: (a -> b) -> a -> b
```

Polecenie 10. Wyjaśnij różnicę pomiędzy wyrażeniami:

```
1 ‘const‘ undefined
undefined ‘seq‘ 1
(undefined:undefined) ‘seq‘ 1
(undefined:undefined) ‘deepseq‘ 1
```

oraz pomiędzy wyrażeniami:

```
xs ‘seq‘ 1
length xs ‘seq‘ 1
xs ‘deepseq‘ 1
length xs ‘deepseq‘ 1
```

gdzie `xs :: [Integer]`.

Polecenie 11. Zapoznaj się z dokumentacją modułu `Control.DeepSeq` z pakietu `deepseq`.

Na koniec rozważmy jeszcze ważny przykład funkcji wykorzystującej `seq`. Poza standardową funkcją

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ c [] = c
foldl (+) c (x:xs) = foldl (+) c’ xs where c’ = c + x
```

mamy też, podobnie jak `foldr’` (a właściwie ich uogólnienia na klasę `Foldable`), zdefiniowaną w module `Data.List` funkcję

```
foldl’ :: (b -> a -> b) -> b -> [a] -> b
foldl’ _ c [] = c
foldl’ (+) c (x:xs) = c’ ‘seq‘ foldl’ (+) c’ xs where c’ = c + x
```

Funkcję `foldl` często wykorzystujemy w obliczeniach monolitycznych, takich jak

```
length = foldl (const . (+1)) 0
```

Jeśli przyjmiemy leniwą strategię redukcji (zawsze zamykamy najbardziej zewnętrzny, skrajnie lewy redeks), to w trakcie rekurencyjnego przechodzenia przez listę dodawanie jedynki nie będzie wykonywane (redeks ten leży bowiem wewnątrz redeksu zadanego przez drugą klauzulę definiującą `foldl`). Najpierw zostanie zatem zbudowane wyrażenie postaci $0+1+1+1+1+\ldots+1$ zawierające tyle jedynek, jak długa jest lista, a dopiero potem, gdy na skutek wykorzystania pierwszej klauzuli definiującej funkcję `foldl` jej nazwa zostanie usunięta z korzenia redukowanego wyrażenia, to zostanie ono policzone. Funkcja `foldl’` wymusza dodawanie jedynki w trakcie wykonywania kroku rekursji. Oczywiście Haskell nie jest *lazy* — jest tylko *non-strict*. Znaczy to, że ma prawo obliczać wyrażenia zgodnie z dowolną, wybraną przez siebie strategią redukcji tak długo, jak długo nie oblicza podwyrażeń, które nie są niezbędne do wyznaczenia wyniku.[4] Optymalizator może wykonać tzw. *strictness analysis* i na jej podstawie podjąć decyzję, że skoro dodawanie i tak na końcu będzie musiało być wykonane, to można je wykonać wcześniej. Używając `foldl’` jedynie pomagamy mu podjąć właściwą decyzję.

Polecenie 12. Przeczytaj podrozdział *Space leaks and strict evaluation* z Rozdziału 4 książki *Real World Haskell*, a następnie wpisz w `ghci` wyrażenia

[4] Uwaga Maćka: *To chyba nie do końca tak jest: kompilator może sobie obliczać te podwyrażenia, które chce, nawet te, które nie biorą potem udziału w wyniku, byle nie zrobić sobie krzywdy przez zapętlenie się, jakiś wyjątek czy próbę policzenia undefined-a.*

```
Prelude> import Data.List
Prelude Data.List> foldl (+) 0 [1..10000000]
50000005000000
Prelude Data.List> foldl' (+) 0 [1..10000000]
50000005000000
```

Czy widzisz różnicę? Zauważ, że w trybie interaktywnym kompilator nie wykonuje optymalizacji programu.

Funkcje w Haskellu dzieli się zwykle na monolityczne i inkrementacyjne oraz ogonowe i nieogonowe. Funkcja jest monolityczna, jeśli obliczenie słabej czołowej postaci normalnej wywołania tej funkcji prowadzi do wykonania całości obliczenia (choć nie oznacza to obliczenia pełnej postaci normalnej), tzn. że funkcji tej nie można wykonywać *inkrementacyjnie*, po kawałku. Ze względu na leniwość funkcje inkrementacyjne są cenione w Haskellu — pozwalają leniwie pomijać wykonanie części pracy, a nawet budować wartości nieskończone. Aby funkcja była inkrementacyjna, to musi produkować fragment wyniku zanim przetworzy cały argument. Przykładem funkcji monolitycznej jest `reverse` (aby wygenerować głowę odwróconej listy, trzeba odwrócić całą listę). Przykładem funkcji inkrementacyjnej jest `(++)` (głowę wyniku możemy wygenerować w czasie stałym, niezależnym od długości spinanych list).

Polecenie 13. Uzasadnij, czemu mamy taki dualizm: funkcja `foldr (*) c` jest inkrementacyjna, jeśli operator `(*)` jest *non-strict* względem prawego argumentu. Funkcja `foldl (+) c` jest monolityczna, jeśli operator `(+)` jest *strict* względem lewego argumentu. Wywnioskuj stąd, że funkcję `foldr` warto wykorzystywać do obliczeń inkrementacyjnych, w których obliczenie `(*)` jest leniwe, zaś `foldl` do obliczeń monolitycznych, gdzie operator `(+)` jest obliczany gorliwie. Jeśli operator `(*)` jest obliczany gorliwie, wówczas funkcja `foldr (*) c` działa nieefektywnie, ponieważ jest nieogonowa. W takich wypadkach lepiej użyć `foldl`, a właściwie `foldl'`.

Abstrakcyjne typy danych i *code reuse*

Konkretny typ danych, to typ, do reprezentacji wartości którego mamy dostęp. W Haskellu konkretnymi typami danych są algebraiczne typy danych, jeśli mamy dostęp do ich konstruktorów.

Abstrakcyjny typ danych, to typ posiadający zbiór pewnych abstrakcyjnych wartości, których struktury nie znamy, wraz z ustalonym zbiorem funkcji działających na jego (abstrakcyjnych) elementach. W Haskellu abstrakcyjne typy danych tworzymy zapakowując algebraiczny typ danych w module i pomijając na liście eksportowej konstruktory tego typu. Na przykład

```
data Stack a = Push a (Stack a) | Empty
```

jest konkretnym typem danych (w tym przypadku stosem elementów typu `a`). Mamy dostęp do reprezentacji wartości tego typu i możemy je przetwarzać za pomocą dopasowania wzorca. Z drugiej strony moduł

```
module Stack (Stack, empty, push, pop) where
   data Stack = Push a (Stack a) | Empty
   empty = Empty
   isEmpty Empty = True
   isEmpty _ = False
   push = Push
   pop (Push a s) = (a,s)
```

jest implementacją abstrakcyjnego typu danych `Stack`. Nie mamy dostępu do konstruktorów tego typu, a na jego wartościach możemy operować wyłącznie za pomocą określonego zbioru operacji — *konstruktorów*:

```
empty :: Stack a
push :: a -> Stack a -> Stack a
```

destruktora:

```
pop :: Stack a -> (a, Stack a)
```

i *obserwatora*:

```
isEmpty :: Stack a -> Bool
```

(możemy definiować także *operatory*). Specyfikacja abstrakcyjnego typu danych zawiera też zwykle abstrakcyjne prawa działania na jego elementach, np. że konstruktor `push` i destruktor `pop` są wzajemnie odwrotne:

$$\texttt{pop . uncurry push} \;=\; \texttt{id} \tag{1}$$

ale nie opisuje szczegółów implementacji tego typu. Abstrahowanie od zbytecznych detali implementacyjnych i tworzenie oprogramowania w postaci nałożonych na siebie warstw, z których każda korzysta wyłącznie z API warstwy niższej (nie mając dostępu do szczegółów jej implementacji) i udostępnia API dla warstwy wyższej (ukrywając szczegóły jego implementacji) jest fundamentalną techniką tworzenia niezawodnego oprogramowania.[5]

Polecenie 14. Przeczytaj Rozdział 5 *Modules* definicji języka *Haskell 2010 Language Report* oraz podrozdział *6.2.1. Modules vs. filenames* dokumentacji kompilatora *Glasgow Haskell Compiler User's Guide.*

Innym sposobem tworzenia abstrakcyjnych typów danych w Haskellu jest wykorzystanie mechanizmu klas:

```
class Stack s where
   empty :: s a
   -- | prop> pop . uncurry push = id
   push :: a -> s a -> s a
   pop :: s a -> (a, s a)
   isEmpty :: s a -> Bool
```

Deklaracji funkcjonalności metod powinny towarzyszyć aksjomaty, takie jak równość (1). Niestety nie można ich wyrazić wprost w Haskellu, ale mamy znacznik `prop>` w Haddocku, jak w powyższym przykładzie, który może być przetwarzany np. przez automatyczne narzędzia do weryfikacji kodu. Każdy typ zainstalowany w klasie `Stack` jest teraz implementacją stosu. Program korzystający ze stosów możemy sparametryzować klasą `Stack` i używać stosów wyłącznie poprzez metody tej klasy.

Klasy typów są praktycznym sposobem abstrahowania i uogólniania w Haskellu. Przypuśćmy, że chcielibyśmy zaprogramować algorytm szybkiego potęgowania w Haskellu. Moglibyśmy napisać

```
(^) :: Integer -> Integer -> Integer
x ^ 0 = 1
x ^ n
   | n ‘mod‘ 2 = 0 = x’ * x’
   | otherwise = x * x’ * x’ where x’ = x ^ (n ‘div‘ 2)
```

Aby udowodnić, że szybkie potęgowanie zgadza się z indukcyjną definicją

```
x ^ 0 = 1
x ^ (n+1) = (x ^ n) * x
```

musimy skorzystać z następujących praw arytmetyki:

- `1` jest elementem neutralnym mnożenia,
- mnożenie jest łączne.

[5] Na przykład twórcy systemu operacyjnego NetBSD twierdzą, że wysoka jakość kodu tego systemu wynika z faktu, że podstawowym założeniem projektowym jest zmaksymalizowanie przenośności (obecne dostępnych jest 57 portów tego systemu). Architektury procesorów różnią się tak znacznie, że wymusiło to tworzenie abstrakcyjnych interfejsów i warstwowej struktury oprogramowania oraz ukrywanie detali implementacyjnych w modułach zależnych od architektury.

Zauważmy, że praktycznie taki sam algorytm możemy mieć dla napisów:

```
(^) :: String -> String -> String
x ^ 0 = ""
x ^ n
   | n `mod` 2 = 0 = x' ++ x'
   | otherwise = x ++ x' ++ x' where x' = x ^ (n `div` 2)
```

czy ogólniej — list. Teraz korzystamy z faktu, że operacja spinania list jest łączna, a jej elementem neutralnym jest lista pusta. Możemy też chcieć szybko potęgować liczby zmiennopozycyjne lub macierze kwadratowe. Za każdym razem binarna operacja powinna być łączna, a wyróżniony element — jej elementem neutralnym. Za pomocą klas typów możemy wyrazić to następująco:

```
class Monoid a where
   (<+>) :: a -> a -> a
   e :: a
   -- | prop> (x <+> y) <+> z = x <+> (y <+> z)
   --   prop> x <+> e = x
   --   prop> e <+> x = x
```

Możemy teraz zdefiniować ogólną operację potęgowania:

```
(^) :: Monoid a => a -> Integer -> a
x ^ 0 = e
x ^ n
   | n `mod` 2 = 0 = x' <+> x'
   | otherwise = x <+> x' <+> x' where x' = x ^ (n `div` 2)
```

Na koniec możemy zainstalować w klasie `Monoid` wiele typów, otrzymując od razu operację potęgowania dla nich wszystkich.

Proces generalizacji polega tu na wykonaniu następujących kroków:

- zidentyfikowanie powtarzającego się schematu w programach;
- wyspecyfikowanie niezbędnych funkcjonalności za pomocą klasy — wraz z zestawem *aksjomatów* (łączność itp.);
- zaprogramowanie ogólnych funkcji unifikujących te powtarzające się schematy;
- zaprogramowanie instancji klasy dla każdej instancji powtarzającego się schematu.

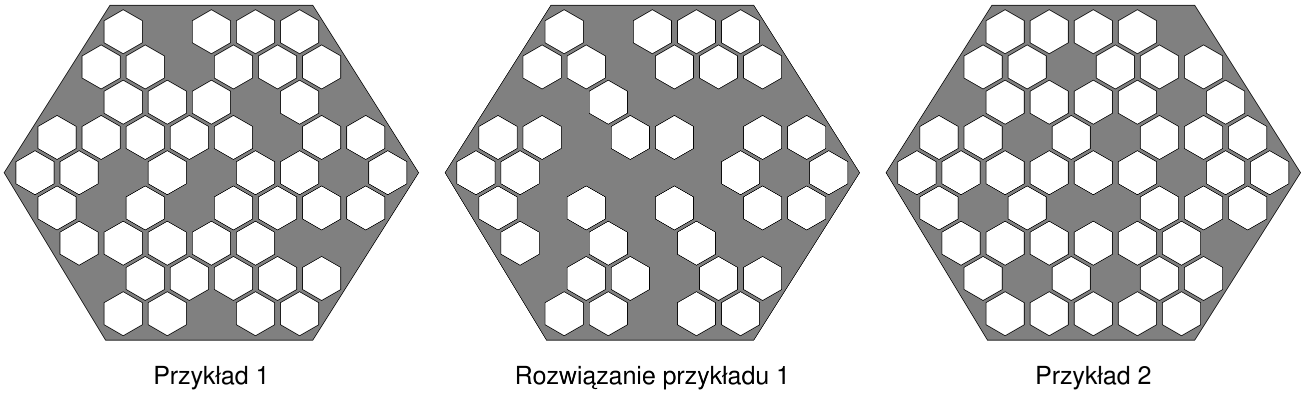
W ten sposób wprowadzono do Haskella szereg klas: `Functor` — typ, dla którego można zdefiniować taką funkcję `map`, jak dla list, `Foldable` — typ, dla którego można zdefiniować taką funkcję `foldr`, jak dla list, `Traversable` — typy, takie jak lista, których elementy można przeglądać od lewej do prawej, `Applicative` (funktory wyposażone w dodatkową operację aplikacji — funktory aplikatywne) i — przede wszystkim — monady.

Monady przypominają nieco monoidy (ich nazwa pochodzi od monoidów). Są uogólnieniem pewnego powtarzającego się wzorca, do którego należą m. in. typy:

- `Maybe` — obliczenie, które może nie dostarczyć wyniku,
- `Either String` — obliczenie, które może się nie powieść (napis zawiera wówczas opis błędu),
- `[]` — obliczenie, które może dostarczyć zero, jeden lub więcej wyników.

Ostatni typ można dalej uogólnić do `MonadPlus` — struktury przypominającej pierścień w algebrze: jest monoidem względem operacji `mplus` z elementem neutralnym `mzero` i monadą względem operacji `>>=`.

Zadanie 2 (1 pkt). W zadaniu 1 z listy 2 implementowaliśmy funkcje generujące dowolne podciągi podanej listy oraz permutacje (przez wstawianie i wybieranie). Uogólnij te funkcje na dowolną monadę z plusem, tj. zaprogramuj funkcje



Rysunek 1: Przykład łamigłówki i rozwiązania

```
subseqM :: MonadPlus m => [a] -> m [a]
ipermM :: MonadPlus m => [a] -> m [a]
spermM :: MonadPlus m => [a] -> m [a]
```

Tam, gdzie to możliwe, użyj `do`-notacji.

Polecenie 15. Przeczytaj artykuł: Philip Wadler, How to Declare an Imperative, *ACM Comput. Surveys*, **29**(3):240–263, September 1997.

Zadanie 3 (5 pkt). Rozważmy następującą łamigłówkę pochodzącą z numeru 3/2000 miesięcznika „Wiedza i Życie”: *Miodowe wyspy:* Usuń [w oryginale *zaczerń*] *niektóre* sześciokąty [zwane w oryginale *sześcianikami*] *tak, aby* pozostałe [w oryginale *żółte*] *utworzyły sześć „wysp”. Każda wyspa powinna składać się z sześciu pól i nie może dotykać do żadnej z pozostałych wysp (jak w zamieszczonym przykładzie z rozwiązaniem).*

Aby móc wygodnie opisywać łamigłówki (sytuacje początkowe na sześciokątnej planszy przypominającej plaster miodu), wprowadzamy układ współrzędnych o środku w środku tego plastra. Środki komórek sąsiadujących poziomo układają się na liniach prostych. W pionie jednak komórki sąsiednich rzędów są względem siebie przesunięte. Aby rozważać wyłącznie współrzędne całkowite przyjmijmy, że jednostką osi odciętych jest $\frac{3}{2}r$, zaś osi rzędnych $\sqrt{3}r$, gdzie r jest promieniem okręgu opisanego na pojedynczej sześciokątnej komórce plastra. Na przykład lista

```
[(-2, 4), (-1, 3), (-6, 2), ( 6, 2), ( 2, 2), ( 3, 1), (-4, 0), ( 0, 0),
 ( 6, 0), (-5,-1), (-1,-1), ( 4,-2), ( 6,-2), (-5,-3), ( 0,-4)]
```

opisuje zaczernione pola z Przykładu 1 na Rysunku 1, zaś lista

```
[( 4, 4), (-1, 3), (-6, 2), ( 4, 2), (-3, 1), ( 1, 1), ( 4, 0), (-5,-1),
 (-1,-1), ( 1,-1), ( 6,-2), ( 1,-3), (-3,-3)]
```

— z Przykładu 2. Para liczb całkowitych (x, y) tworzy współrzędne środka pewnej komórki, jeśli liczby x i y mają taką samą parzystość i nie są zbyt wielkie. W podanych przykładach odcięte przebiegają zbiór $[-4..4]$, jednak rozmiar plastra miodu (górne ograniczenie odciętych) lepiej uczynić parametrem naszego programu (będziemy mogli wówczas uruchamiać nasz program na trywialnych zadaniach małego rozmiaru, a następnie przeprowadzać testy wysiłkowe na zadaniach znacznie większych). Danymi wejściowymi dla programu powinny być zatem: rozmiar zadania, liczba wysp, rozmiar wysp i lista zaczernionych na początku komórek.

Zaprogramuj moduł `MiodoweSolver` eksportujący następujące typy i funkcję

```
type Komorki = [(Int,Int)]
data Miodowe = Miodowe {
   rozmiar :: Int,
   liczba_wysp :: Int,
```

```
   rozmiar_wysp :: Int,
   pola :: Komorki
}
solver :: MonadPlus m => Miodowe -> m Komorki
```

Funkcja `solver` powinna realizować przeszukiwanie z nawrotami. Planszę można potraktować jak graf, w którym komórki są wierzchołkami, zaś krawędzie prowadzą do sąsiadujących komórek. Możemy realizować przeszukiwanie grafu w głąb. Za każdym razem, gdy przechodzimy do jeszcze nie odwiedzonego wierzchołka, to mamy do wyboru dwie możliwości: albo do niego przechodzimy (i staje się on częścią bieżącej wyspy), albo decydujemy, że wierzchołek ten nie należy do grafu (tj. komórka jest zaczerniona). Podczas przeszukiwania można obcinać sporo gałęzi: nie można usunąć wierzchołka, jeśli w grafie pozostanie ich zbyt mało, by utworzyć zadaną liczbę wysp. Nie można pozostawić wierzchołka, jeśli dołączenie go do wyspy spowodowałoby przekroczenie jej rozmiaru.

Zaprogramuj moduł `MiodoweInput` eksportujący funkcję

```
miodoweInput :: String -> Miodowe
```

która wczytuje opis łamigłówki z pliku o podanej nazwie i tworzy daną typu `Miodowe`. Przykład 1 z Rysunku 1 powinien być opisany następującym plikiem:

```
4
6
6
[(-2, 4), (-1, 3), (-6, 2), ( 6, 2), ( 2, 2), ( 3, 1), (-4, 0), ( 0, 0),
 ( 6, 0), (-5,-1), (-1,-1), ( 4,-2), ( 6,-2), (-5,-3), ( 0,-4)]
```

Napisz następnie moduł `Miodowe` zawierający funkcję

```
miodowe :: String -> IO ()
```

która wczytuje opis łamigłówki z pliku i wypisuje znalezione rozwiązania na standardowe wyjście.

Zadanie 4 (1 pkt). Napisz moduł `MiodoweASCII` zawierający funkcję

```
miodoweASCII :: String -> IO ()
```

która wczytuje opis łamigłówki z pliku i wypisuje znalezione rozwiązania na standardowe wyjście w postaci ASCII artu.

Zadanie 5 (bonus 5 pkt). Napisz program, który wykorzystuje moduły `MiodoweSolver` i `Miodowe-Input` i przedstawia rozwiązania w postaci interfejsu graficznego (np. takiego, jak na Rysunku 2).

Zadanie 6 (3 pkt). Rozważmy typ widoku danych jako listy:

```
data List t a = Cons a (t a) | Nil
```

Jak zwykle zwykłe listy możemy zdefiniować jako punkt stały operatora `List`:

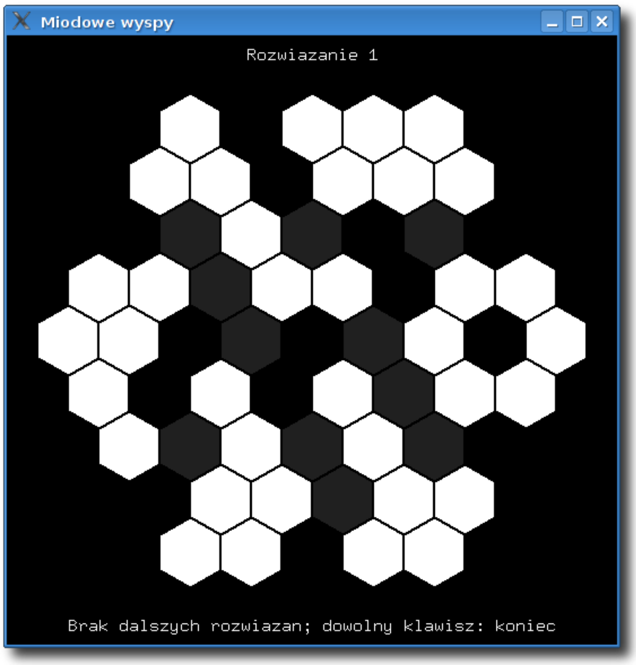
```
newtype SimpleList a = SimpleList { fromSimpleList :: List SimpleList a }
```

Jak zwykle w widokach wykorzystamy metodę klasy

```
class ListView t where
   viewList :: t a -> List t a
   toList :: t a -> [a]
   cons :: a -> t a -> t a
   nil :: t a
```

Dodaj do powyższej definicji klasy domyślną implementację metody `toList`.

Listy konkatenowalne w czasie stałym, to w istocie drzewa binarne o etykietowanych liściach:



Rysunek 2: Graficzny interfejs dla Miodowych Wysp

```
data CList a = CList a :++: CList a | CSingle a | CNil
```

Przyjmujemy, że elementy tak reprezentowanej listy, to etykiety liści w kolejności od lewej do prawej. Operatorem konkatenacji jest konstruktor `:++:`. Działa on w czasie stałym (kosztem liniowego czasu operacji takich, jak `head` i `tail`). Zainstaluj typ `CList` w klasach `ListView`, `Functor`, `Applicative`, `Monad`, `MonadPlus`, `Foldable` i `Traversable`.

Zadanie 7 (3 pkt). *Listy różnicowe*, to (podobnie jak w Prologu) listy, w których wstawianie na koniec odbywa się w czasie stałym. W Haskellu implementujemy je za pomocą funkcji:

```
newtype DList a = DList { fromDList :: [a] -> [a] }
```

Rozważmy listę różnicową zawierającą liczby `1`, `2` i `3`:

```
xs = DList (\ tl -> 1:2:3:tl)
```

Aby dodać na jej koniec liczbę `4` wystarczy napisać

```
DList (fromDList xs . (4:))
```

Metoda `toList` powinna być dla tego typu zdefiniowana osobno, gdyż projekcja na zwykłe listy jest tu wyjątkowo prosta:

```
fromDList xs []
```

Zaprogramuj operacje

```
dappend :: DList a -> DList a -> DList a
```

Zainstaluj typ `DList` w klasach `ListView`, `Functor`, `Applicative`, `Monad`, `MonadPlus`, `Foldable` i `Traversable`.

Zadanie 8 (3 pkt). *Listy o dostępie swobodnym*, to listy pełnych drzew binarnych o etykietowanych wierzchołkach wewnętrznych o rosnących wysokościach. Drzewo o pewnej wysokości występuje na tej liście dokładnie wtedy, gdy odpowiadający mu bit w dwójkowym rozwinięciu liczby elementów listy wynosi 1. Analogia pomiędzy tą strukturą danych i dwójkowymi rozwinięciami liczb jest widoczna w definicji typu:

```
data LTree a = LTree a :/\: LTree a | LLeaf a
data Digit a = Zero | One (LTree a)
newtype RAList a = RAList { fromRAList :: [Digit a] }
```

Przyjmujemy, że elementy tak reprezentowanej listy, to etykiety liści kolejnych drzew w kolejności od lewej do prawej. Dodanie elementu do głowy listy działa dokładnie tak, jak obliczenie następnika liczby. Obliczenie ogona listy działa tak, jak obliczenie poprzednika liczby. Zaprogramuj operacje

```
ralookup :: RAList a -> Int -> a
raupdate :: RAList a -> Int -> a -> RAList a
```

Operacje te powinny działać w czasie logarytmicznym względem liczby elementów listy (stąd pochodzi nazwa tych list). Zainstaluj typ `RAList` w klasach `ListView`, `Functor`, `Applicative`, `Monad`, `MonadPlus`, `Foldable` i `Traversable`.