Programowanie 2009 Programming 2009

Lista zadań nr 7 Problem set no. 7

Na zajęcia 21–22 kwietnia 2009 Due April 22, 2009

Grupy zasadnicze Basic groups

Rozważmy nieskończoną listę wszystkich liczb pierwszych:

Consider the infinite list of all prime numbers:

```
primes :: [Integer]
```

Bez cukru syntaktycznego definicja tej listy jest mało czytelna:

Without a syntactic sugar the definition of this list is hardly readable:

```
primes = 2 : filter (λ n → all (λ p → n `mod` p ≠ 0)
                    (takeWhile (λ p → p*p ≤ n) primes)) (enumFrom 3)
```

Wyrażenia listowe poprawiają czytelność:

List comprehensions improve readability:

```
primes = 2 : [ n | n ← [3..], all (λ p → n `mod` p ≠ 0)
                                  (takeWhile (λ p → p*p ≤ n) primes) ]
```

Skorzystajmy z faktu, że typ `[]` jest monadą i przyjmijmy definicję

Let us use the fact that the type `[]` is a monad and let

```
type Generator = []
```

Dzięki leniwości listę liczb pierwszych możemy traktować jak generator wyliczający na żądanie kolejne liczby pierwsze:

Due to laziness the list of prime numbers may be considered as a generator that computes successive prime numbers on demand:

```
primes :: Generator Integer
primes = return 2 `mplus` do
   n ← enumFrom 3
   guard (all (λ p → n `mod` p ≠ 0) (takeWhile (λ p → p*p ≤ n) primes)
   return n
```

Zapisz we wszystkich trzech przedstawionych wyżej stylach (bez cukru syntaktycznego, z wyrażeniami listowymi oraz z do-notacją) algorytmy opisane w postaci predykatów prologowych w zadaniach 1–4.

Write in all three styles presented above (without any syntactic sugar, using list comprehensions and using the do-notation) the algorithms described as Prolog predicates in Problems 1–4.

Zadanie 1 (1 pkt). Ogony listy:

Problem 1 (1 p). Tails of a list:

```
tails(T,T).
tails([_|T],S) :-
   tails(T,S).
```

Zaprogramuj odpowiadającą haskellową funkcję:

Define a corresponding Haskell function:

```
tails :: [a] → [[a]]
```

Zadanie 2 (1 pkt). Podlisty podanej listy:

Problem 2 (1 p). Sublists of a given list:

```
sublist([],[]).
sublist([H|T],[H|S]) :-
   sublist(T,S).
sublist([_|T],S) :-
   sublist(T,S).
```

Zaprogramuj odpowiadającą haskellową funkcję:

Define a corresponding Haskell function:

```
sublist :: [a] → [[a]]
```

Zadanie 3 (1 pkt). Permutacje przez wybieranie:

Problem 3 (1 p). Permutations by selection:

```
perms([],[]).
perms(S,[H|T]) :-
   select(S,H,R),
   perms(R,T).
```

Zaprogramuj odpowiadającą haskellową funkcję:

Define a corresponding Haskell function:

```
perms :: [a] → [[a]]
```

Zadanie 4 (1 pkt). Permutacje przez wstawianie:

Problem 4 (1 p). Permutations by insertion:

```
permi([],[]).
permi([H|T],S) :-
   permi(T,R),
   select(S,H,R).
```

Zaprogramuj odpowiadającą haskellową funkcję:

Define a corresponding Haskell function:

```
permi :: [a] → [[a]]
```

Zadanie 5 (1 pkt). Czym się różni wyrażenie

Problem 5 (1 p). What is the difference between the expression

```
[ e | p ← e', b ]
```

od wyrażenia

and the expression

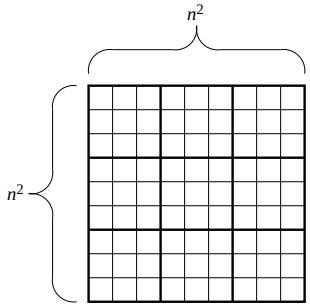
```
do
   p ← e'
   guard b
   return e
```

(Wyrażenia `e`, `e'` i `b` oraz wzorzec `p` są odpowiednich typów i zawierają takie zmienne wolne, jakie powinny.)

(The expressions `e`, `e'` and `b` and the pattern `p` have appropriate types and contain free variables as required.)

Zadanie 6 (1 pkt). Napisz w Haskellu program, który rozwiąże następującą łamigłówkę, pochodzącą z numeru 6/2001 miesięcznika „Wiedza i Życie": niech n będzie ustaloną liczbą dodatnią (w naszym przykładzie n będzie równe 3). Rozważmy kwadrat o boku n^2. Kwadrat ten składa się z n^2 rozłącznych „małych" kwadratów o boku n, jak na rysunku:

Problem 6 (1 p). Write a Haskell program that solves the following puzzle taken from the 6/2001 issue of the „Knowledge and Life" magazine: let n be a fixed positive number (in our example n equals 3). Consider a square whose sides have length n^2. The square consist of n^2 disjoint "small" squares with sides of length n, as shown in the picture below:



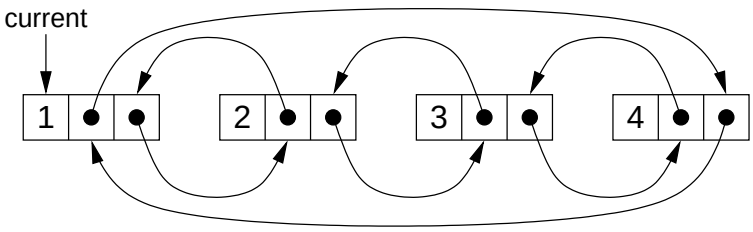
W pola dużego kwadratu należy wpisać liczby z przedziału $\langle 1, n^2 \rangle$ tak, by w każdym wierszu, w każdej kolumnie oraz w każdym małym kwadracie każda liczba występowała dokładnie raz. Dane wejściowe dla programu stanowi liczba n oraz lista zawartości niektórych pól, np.:

The numbers from the range $\langle 1, n^2 \rangle$ should be written into the squares of the grid of the big square in such a way, that in every row, column and small square each number occurs exactly once. Input data for your program consist of the number n and a list of the contents of some squares, *e.g.*:

9					8	1		
	1			7			2	
	4			6			3	
1					5	4		
		8	9					5
	7			2			5	
	6			3			9	
		5	4					7

Zadanie 7 (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).

Problem 7 (1 p). *Two-way cyclic list* is a sequential access data structure in which every element has a pointer to the previous and the next element, and all elements form a (possibly infinite) cycle.



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

A modification of an element of a persistent data structure implies copying all elements reachable from the given element, hence any modification of a two-way cyclic list leads to copying the whole data structure, which is extremely inefficient. Hence we will program selectors and observers only, *i.e.*, operations that do not modify the structure. Thus let:

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

Zdefiniuj funkcje

Define functions

```
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 w przód i w tył, a `label` ujawnia etykietę bieżącego elementu, np.

The function `fromList` creates a cyclic list containing elements of a given list. The current element is the first element of the given list. The functions `forward` and `backward` move the pointer of the current element forward and backward, and the `label` function shows the label of the current element, *e.g.*,

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

Jeśli *xs* jest listą nieskończoną, to

If *xs* is an infinite list then

```
backward . fromlist $ xs  =  ⊥.
```

Zadanie 8 (1 pkt). Zdefiniuj nieskończoną cyklistę

Problem 8 (1 p). Define an infinite list

```
enumInts :: Cyclist Integer
```

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

containing all integers in the natural order with zero being the current element.

Grupy rozszerzone

Extended groups

Zadanie 1 (1 pkt). Niech `Cyclist a` będzie naszym stanem obliczeń. Obliczenie modyfikuje stan i zwraca wynik typu `b`, ma więc typ

Problem 1 (1 p). Let `Cyclist a` be our state of computation. A computation modifies the state and returns a result of type `b`, so it has the type

```
Cyclist a → (b, Cyclist a)
```

Obliczenie z cyklistą jako stanem jest więc wartością typu

A computation with a cyclic list as a state is thus a value of the type

```
newtype Cyc a b = Cyc (Cyclist a → (b, Cyclist a))
```

Uczyń typ `Cyc a` monadą. Zdefiniuj operacje:

Make the type `Cyc a` a monad. Define operations:

```
runCyc :: Cyclist a → Cyc a b → b
fwd :: Cyc a ()
bkw :: Cyc a ()
lbl :: Cyc a a
```

tak by dało się obliczenia zapisywać następująco:

so that we could write the computations as follows:

```
example :: Integer
example = runCyc enumInts (do
   bkw
   bkw
   bkw
   bkw
   x ← lbl
   fwd
   fwd
   y ← lbl
   fwd
   z ← lbl
   return (x+y+z))
```

Zadanie 2 (1 pkt). Oto prosty generator losowy zaprogramowany w C:

Problem 2 (1 p). Here is a simple random generator written in C:

```
int seed;

void init(int newSeed) {
   seed = newSeed;
}

int random() {
   int newSeed = 16807 * (seed % 127773) - 2836 * (seed / 127773);
   return seed = (newSeed > 0 ? newSeed : (newSeed + 2147483647));
}
```

Zaprogramuj abstrakcyjny typ `Random a` reprezentujący obliczenie z wynikiem typu `a` i stanem będącym wartością zarodka losowego. Zainstaluj go w klasie `Monad` i zdefiniuj dla niego operacje

Define an abstract type `Random a` that represents a computation with result of type `a` and with state being a value of the random seed. Install it in the `Monad` class and define for it the operations:

```
init :: Int → Random ()
random :: Random Int
```

Zadanie 3 (1 pkt). Napis i strumień wejściowy są bardzo podobnymi strukturami danych — udostępniają sekwencyjny dostęp do znaków. Np. w Javie napisy (`StringReader`) i pliki (`FileReader`) są instancjami tego samego interfejsu `Reader`. Obliczenie, które działa na strumieniu (napisie) zwraca pewną wartość pewnego typu `a` i zmodyfikowany strumień (modyfikacja strumienia polega na przeczytaniu zeń znaków), jest więc typu

Problem 3 (1 p). The character string and the input stream are very similar data structures — they provide the sequential access to characters. For example in Java the strings (`StringReader`) and files (`FileReader`) are instances of the same interface `Reader`. A computation over a stream (or a string) returns some result value of type `a` and the modified stream (a modification of the stream consists in reading some characters from it), so it has the type

```
String → (a, String).
```

Niech zatem

Hence let

```
newtype SSC a = SSC (String → (a, String))
```

(SSC to skrót od „string stream computation"). Uczyń typ `SSC` monadą. Zdefiniuj dla niego operacje

(SSC stands for "string stream computation"). Make the type `SSC` a monad. Define for it the operations

```
runSSC :: SSC a → String → a
getc :: SSC Char
ungetc :: Char → SSC ()
isEOS :: SSC Bool
```

tak, by dało się zdefiniować np. takie obliczenie:

so that such a computation is definable:

```
countLines :: String → Int
countLines = runSSC $ lines 0 where
   lines :: Int → SSC Int
   lines n = do
      b ← isEOS
      if b
         then return n
         else do
            ch ← getc
            lines (if ch == '\n' then n+1 else n)
```

Zadanie 4 (1 pkt). Zauważ, że definicje operatorów `>>=` i `return` w poprzednich trzech zadaniach są praktycznie identyczne. Zdefiniuj więc monadę realizującą dowolne obliczenie z dowolnym wynikiem w której typem obliczenia jest

Problem 4 (1 p). Note that the definitions of the `>>=` and `return` operators in the previous three problems are practically identical. Define a monad that performs an arbitrary computation with an arbitrary result in which the type of the computation is

```
newtype StateComput s a = SC (s → (a,s))
```

Oto klasa rozszerzająca klasę monad o możliwość czytania znaków i ujawniania bieżącej pozycji w czytanym tekście (tj. liczby dotychczas przeczytanych znaków):

Here is a class that extends the monad class with capabilities of reading characters and showing the current position in the text (*i.e.*, the number of characters read so far):

```
class Monad m => CountInputMonad m where
   getc :: m Char
   eot :: m Bool
   position :: m Int
```

Klasę tę można wykorzystać np. w programie ujawniającym pozycje wystąpień litery `A` w tekście:

We can use this class in a program that shows the positions of occurrences of the letter `A` in the text:

```
lettersA :: CountInputMonad m => m [Int]
lettersA = do
   stop ← eof
   if stop
      then return []
      else do
         ch ← getc
         cnt ← position
         rest ← lettersA
         if ch == 'A'
            then return (cnt : rest)
            else return rest
```

Poniższy typ jest transformatorem dodającym do dowolnej monady `m :: * → *` stan typu `s`:

The following type is a monad transformer that adds the state of type `s` to an arbitrary monad `m :: * → *`:

```
newtype State s m a = State { runState :: s → m (s,a) }
```

Zauważmy, że rodzajem typu `State` jest

Note that the `State` type has kind

```
State :: * → (* → *) → (* → *)
```

Przypomnijmy, że transformatory monad to typy implementujące klasę

Recall that monad transformers are types implementing the class

```
class MonadTransformer (t :: (* → *) → (* → *)) where
   promote :: Monad m => m a → t m a
```

Przypomnijmy na koniec definicję monady identycznościowej (od niej przeważnie zaczyna się składanie transformatorów):

Finally recall the definition of the identity monad (on which we usually start the application of monad transformers):

```
newtype Id a = Id  unId :: a
instance Monad Id where
   return = Id
   (Id a) >>= f = f a
```

Zadanie 5 (1 pkt). Uczyń typ `State s m` monadą, a typ `State s` — transformatorem monad, dla dowolnej monady `m` i dowolnego typu `s`.

Problem 5 (1 p). Make the type `State s m` a monad, and the type `State s` a monad transformer for any monad `m` and an arbitrary type `s`.

Rozważmy klasę rozszerzającą monady o pojedynczy licznik:

Consider a class that extends monads with a single counter:

```
class Monad m => CountMonad m where
   counter :: m Int
   tick :: m ()
```

oraz klasę rozszerzającą monady o możliwość czytania znaków:

and a class extending monads with the ability to read characters:

```
class Monad m => InputMonad m where
   getch :: m Char
   eof :: m Bool
```

Zadanie 6 (1 pkt). a) Wykorzystaj transformator monad z poprzedniego zadania by dodać licznik do dowolnej monady, tj. uczyń typ `State Int m` instancją `CountMonad`-y dla dowolnej monady `m`.

Problem 6 (1 p). a) Use the monad transformer from the previous problem to add a counter to an arbitrary monad, *i.e.*, make the type `State Int m` an instance of the `CountMonad` for any monad `m`.

b) Wykorzystaj transformator monad z poprzedniego zadania by dodać do dowolnej monady możliwość czytania znaków, tj. uczyń typ `State String m` instancją `InputMonad`-y dla dowolnej monady `m`.

b) Use the monad transformer from the previous problem to add to an arbitrary monad the ability to read characters, *i.e.*, make the type `State String m` an instance of the `InputMonad` for any monad `m`.

c) Musimy umieć promować własności monad podczas transformacji. Jeśli `m` jest `InputMonad`-ą, to `State s m` też powinien być `InputMonad`-ą. Podobnie jeśli `m` jest `CountMonad`-ą, to `State s m` też powinien być `CountMonad`-ą. Uczyń więc `State s m` instancją tych dwóch klas (przy odpowiednich założeniach o `m`).

c) We need to promote monad properties during transformations. If `m` is an `InputMonad`, then `State s m` should be an `InputMonad` as well. Similarly if `m` is a `CountMonad`, then so should be `State s m`. Thus make `State s m` an instance of the two classes (with appropriate assumptions about `m`).

d) Jeśli `m` jest instancją zarówno `InputMonad`-y, jak i `CountMonad`-y, to możemy wykorzystać licznik do zliczania wczytywanych znaków i uczynić `m` instancją `CountInputMonad`-y. Zrób to.

Zauważ, że możemy teraz zadać dwie implementacje funkcji `lettersA`:

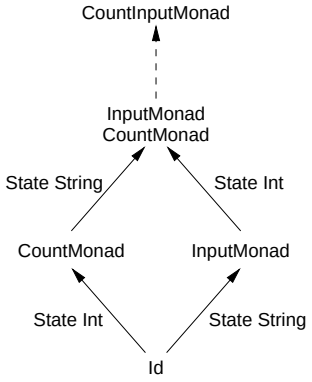
d) If `m` is an instance of both `InputMonad` and `CountMonad`, then we can use the counter to count read characters and make `m` an instance of the `CountInputMonad`. Do this.

Note that we can implement the `lettersA` function in two different ways:

```
lettersAinString1 :: State Int (State String Id) [Int]
lettersAinString1 = lettersA
lettersAinString2 :: State String (State Int Id) [Int]
lettersAinString2 = lettersA
```

ponieważ transformatory możemy składać w dowolnej kolejności:

because we can compose transformers in any order:



e) Wykorzystaj funkcję `lettersAinString` do zdefiniowania funkcji

e) Use the function `lettersAinString` to define a function

```
searchA :: String → [Int]
```

ujawniającej pozycje wystąpień litery `A` w podanym napisie.

that shows the positions of occurrences of the letter `A` in a given string.

Zadanie 7 (1 pkt). Uczyń standardowy typ `IO` instancją klasy `InputMonad` i przetransformuj go do `CountInputMonad`-y za pomocą typów z poprzedniego zadania. Zdefinuj następnie wartość

Problem 7 (1 p). Make the standard `IO` type an instance of the `InputMonad` class and transform it to the `CountInputMonad` using types defined in the previous problem. Then define the value

```
searchA :: IO [Int]
```

ujawniającą pozycje litery `A` w standardowym strumieniu wejściowym.

that shows the positions of occurrences of the letter `A` in the standard input stream.

Zadanie 8 (1 pkt). Parser to monada stanowa w której stanem jest analizowany ciąg tokenów. Niech więc:

Problem 8 (1 p). A parser is a state monad in which the state is the sequence of tokens being analyzed. Hence let:

```
type Parser token m value = Control.Monad.State.StateT [token] m value
```

Zdefiniuj funkcje

Define functions

```
parse :: Monad m => Parser token m value → [token] → m value
isElem :: (Eq token, MonadPlus m) => [token] → Parser token m token
isEmpty :: MonadPlus m => Parser token m ()
many :: MonadPlus m => Parser token m value → Parser token m [value]
many1 :: MonadPlus m => Parser token m value → Parser token m [value]
option :: MonadPlus m => Parser token m value → Parser token m (Maybe value)
number :: MonadPlus m => Parser Char m Integer
```

gdzie `isElem` sprawdza, czy na początku ciągu tokenów występuje jeden spośród wskazanych tokenów, `isEmpty` sprawdza, czy ciąg tokenów jest pusty, `many` parsuje konkatenację zera lub więcej ciągów akceptowanych przez podany parser, `many1` parsuje konkatenację jednego lub więcej ciągów akceptowanych przez podany parser, `option` opcjonalnie parsuje ciąg akceptowany przez podany parser, zaś `number` parsuje niepusty ciąg cyfr dziesiętnych. Użyj powyższych funkcji w implementacji kalkulatora obliczającego wartość wyrażeń opisanych następującą gramatyką:

where `isElem` checks if a sequence of tokens begins with one of given tokens, `isEmpty` checks if the sequence of tokens is empty, `many` parses the concatenation of zero or more sequences accepted by a given parser, `many1` parses the concatenation of one or more sequences accepted by a given parser, `option` parses optionally a sequence of tokens accepted by a given parser, and `number` parses a nonempty sequence of decimal digits. Use the functions presented above in an implementation of a calculator that computes the value of an expression described by the following grammar:

$$
\begin{aligned}
e_0 &\rightarrow e_1\ ((\texttt{+} \mid \texttt{-})\ e_1)^* \\
e_1 &\rightarrow e_2\ ((\texttt{*} \mid \texttt{/} \mid \texttt{\%})\ e_2)^* \\
e_2 &\rightarrow \mathit{number} \mid \texttt{(}\ e_0\ \texttt{)}
\end{aligned}
$$

Grupa zaawansowana

Operacje wejścia/wyjścia w językach czysto deklaratywnych implementowano dawniej przy pomocy dialogów:

Advanced group

Input/output operations in purely declarative languages were formerly implemented by means of the dialogues:

```
module IO(Response(..), Request(..), Dialog) where
   data Request = PutStrLn String | ReadLine
   data Response = OK | OKStr String
   type Dialog = [Response] → [Request]

module Main where
   import IO(Response(..), Request(..), Dialog)
   main :: Dialog
   main resps = ReadLine :
      (case resps of
         OKStr str : resps →
            if str == ""
               then []
               else (PutStrLn . reverse $ str) :
                  (case resps of
                     OK : resps → f resps))
```

Kontynuacje zapewniają łatwiejszą kontrolę skutków ubocznych:

Continuations allow easier control of side effects:

```
module IO(Answer, putStrLn, getLine, done) where
   type Answer
   putStrLn :: String → Answer → Answer
   getLine :: (String → Answer) → Answer
   done :: Answer

module Main where
   import IO(Answer, putStrLn, getLine, done)
   main :: Answer
   main = getLine (λ line →
      if line == ""
         then done
         else putStrLn (reverse line) f)
```

Monady okazały się jednak najlepszym rozwiązaniem:

Monads turned out to be the best solution:

```
module IO(IO, putStrLn, getLine) where
   type IO a
   instance Monad IO
   putStrLn :: String → IO ()
   getLine :: IO String

module Main where
   import IO(IO, putStrLn, getLine)
   main :: IO ()
   main = do
      line ← getLine
      if line == ""
         then return ()
         else do
            putStrLn (reverse line)
            main
```

Zadanie 1 (3 pkt). Zaimplementuj każdy z trzech powyższych mechanizmów w wybranym przez siebie języku imperatywnym używając operacji wejścia/wyjścia, które powodują skutki uboczne.

Problem 1 (3 p). Implement each of the three mechanisms described above in an imperative language of your choice using input/output operations that cause side-effects.

Zadanie 2 (6 pkt). Dla każdego z powyższych trzech mechanizmów przyjmij, że jest on mechanizmem pierwotnym dostępnym w Haskellu i zaimplementuj za jego pomocą pozostałe dwa.

Problem 2 (6 p). For each of the three mechanisms described above assume it is the primitive mechanism available in Haskell and using it implement the other two.