Metody programowania

Egzamin zasadniczy

17 czerwca 2014

Liczba punktów	Ocena
0 – 14	2.0
15 – 17	3.0
18 – 20	3.5
21 – 23	4.0
24 – 26	4.5
27 – 30	5.0

W każdym pytaniu testowym proszę wyraźnie zaznaczyć dokładnie jedną odpowiedź. Jeśli zostanie zaznaczona więcej niż jedna odpowiedź, to za wybraną zostanie uznana ta, która *nie jest* otoczona kółkiem. W pytaniach otwartych proszę czytelnie wpisać odpowiedź wewnątrz prostokąta. Każde pytanie testowe jest warte 1 punkt, każde pytanie otwarte — 2 punkty. Czas trwania egzaminu: 120 minut.

Pytanie 1. Rozważmy standardowy predykat `repeat/0`:

```
repeat.
repeat :-
   repeat.
```

i cel

```
?- write(a), repeat.
```

☐ a. Cel będzie spełniony na nieskończenie wiele sposobów. Przed każdym sukcesem do standardowego strumienia wyjściowego będzie wypisywana pojedyncza litera a.

■ b. Cel będzie spełniony na nieskończenie wiele sposobów. Przed pierwszym sukcesem do standardowego strumienia wyjściowego zostanie wypisana pojedyncza litera a.

☐ c. Do standardowego strumienia wyjściowego zostanie wypisana pojedyncza litera a, po czym obliczenie celu zapętli się.

☐ d. Cel będzie spełniony na nieskończenie wiele sposobów, a jego obliczenie nie wywoła żadnych skutków ubocznych.

Pytanie 2. Rozważmy predykat

```
from(N,N).
from(N,M) :-
   N1 is N+1,
   from(N1,M).
```

i cel

```
?- from(0,N), !, from(0,M), X is 2^N*3^M.
```

(w którym `^` jest standardowym operatorem potęgowania całkowitoliczbowego). Odcięcie, które występuje w podanym celu

- ■ a. nie ma wpływu na wynik jego obliczenia.
- ☐ b. powoduje, że przy kolejnych nawrotach pod zmienną `X` są podstawiane jedynie kolejne potęgi dwójki.
- ☐ c. powoduje, że obliczenie tego celu zakończy się niepowodzeniem.
- ☐ d. powoduje, że cel jest spełniony tylko na jeden sposób.

Pytanie 3. Obliczenie celu

```
?- X is X+X.
```

- ☐ a. kończy się pojedynczym sukcesem, a pod zmienną `X` jest podstawiona wartość `0`.
- ☐ b. zawodzi.
- ☐ c. zapętla się.
- ■ d. kończy się błędem arytmetycznym.

Pytanie 4. Cel `!` w definicji predykatu może zostać usunięty, gdyż jego wykonanie nie ma żadnego efektu, jeśli

- ☐ a. jest pierwszym celem w ciele pierwszej klauzuli.
- ☐ b. jest ostatnim celem w ciele pierwszej klauzuli.
- ■ c. jest pierwszym celem w ciele ostatniej klauzuli.
- ☐ d. jest ostatnim celem w ciele ostatniej klauzuli.

Pytanie 5. Wynikiem zapytania

```
?- [[],[]] = [[],V|V].
```

jest

- ☐ a. pojedynczy sukces, przy czym `V` = `[[]]`.
- ■ b. pojedynczy sukces, przy czym `V` = `[]`.
- ☐ c. pojedynczy sukces, przy czym `V` jest nieukonkretnioną zmienną.
- ☐ d. niepowodzenie.

Pytanie 6. Obliczenie celu

```
?- \+ member(X, [a]), X=b.
```

■ a. zakończy się niepowodzeniem.

☐ b. zakończy się pojedynczym sukcesem, w którym X = a.

☐ c. zakończy się pojedynczym sukcesem, w którym X = b.

☐ d. zakończy się pojedynczym sukcesem, w którym X pozostanie nieukonkretnioną zmienną.

Pytanie 7. Rozważmy predykat

```
p :- p.
```

Obliczenie celu

```
?- p.
```

☐ a. zawiedzie.

■ b. zapętli się.

☐ c. zakończy się pojedynczym sukcesem.

☐ d. dostarczy nieskończenie wielu sukcesów.

Pytanie 8. Obliczenie celu

```
?- append(X,X,X), !.
```

☐ a. zawiedzie.

☐ b. zapętli się.

■ c. zakończy się pojedynczym sukcesem.

☐ d. dostarczy nieskończenie wielu sukcesów.

Pytanie 9. Zaprogramuj w Prologu taki predykat suffix/2, że cel suffix(l_1, l_2) jest spełniony wówczas, gdy lista l_2 jest sufiksem listy l_1, np.

```
?- suffix([a,b,c],X).
X = [a, b, c] ;
X = [b, c] ;
X = [c] ;
X = [].
```

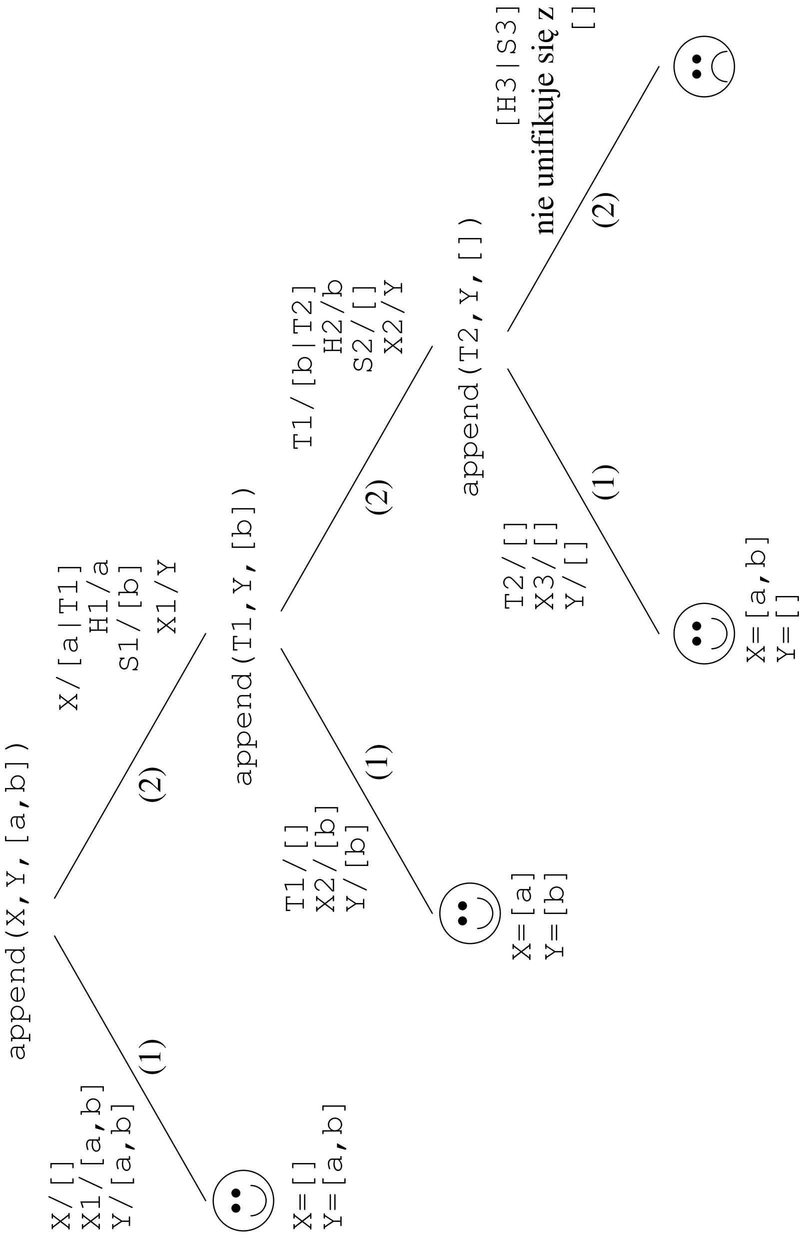
```
suffix(X,X).
suffix([_|T],S) :-
   suffix(T,S).
```

Pytanie 10. Narysuj prologowe drzewo przeszukiwania dla celu

```
?- append(X,Y,[a,b]).
```

gdzie

```
(1)  append([],X,X).
(2)  append([H|T],X,[H|S]) :-
         append(T,X,S).
```



Pytanie 11. Oto predykat dostępny w SWI-Prologu:

```
between(M,N,M) :-
   (N == inf -> true; M =< N).
between(M,N,K) :-
   (N == inf -> true; M < N),
   M1 is M+1,
   between(M1,N,K).
```

Korzystając z tego predykatu napisz w Prologu taki predykat `pairs/2`, że obliczenie celu

```
?- pairs(M,N).
```

przy kolejnych nawrotach podstawia za zmienne `M` i `N` wszystkie kombinacje nieujemnych liczb całkowitych, tj. cel ten jest spełniony na nieskończenie wiele sposobów, a dla każdej pary nieujemnych liczb całkowitych (m, n) istnieje taki sukces, przy którym $\mathtt{M} = m$ i $\mathtt{N} = n$.

```
pairs(M,N) :-
   between(0,inf,S),
   between(0,S,M),
   N is S-M.
```

Pytanie 12. Napisz w Prologu predykat `zip/3` który łączy odpowiadające sobie elementy list w pary, tj. taki, że spełnione są cele postaci

$$\mathtt{zip}([x_1, \ldots, x_m], [y_1, \ldots, y_n], [(x_1, y_1), \ldots, (x_k, y_k)]),$$

gdzie $k = \min(m, n)$. Przyjmij, że będziemy go używać tylko w trybie `(+,+,?)`.

```
zip([],_,[]) :- !.
zip(_,[],[]) :- !.
zip([H|T],[K|S],[(H,K)|R]) :-
   zip(T,S,R).
```

Pytanie 13. Rozważmy standardowe funkcje

```
take n _ | n <= 0 = []
take _ []         = []
take n (x:xs)     = x : take (n-1) xs

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

i niech

```
nieuj x
   | x >= 0 = True
   | otherwise = undefined
```

Wartością wyrażenia

```
take 3 $ map nieuj [1,0..]
```

jest

- ☐ a. $\perp$.
- ☐ b. wartość `True`.
- ☐ c. lista złożona z wartości `True`.
- ■ d. trzyelementowa lista typu `[Bool]`.

Pytanie 14. Rozważmy standardowe funkcje

```
take n _ | n <= 0 = []
take _ []         = []
take n (x:xs)     = x : take (n-1) xs
iterate f x = x : iterate f (f x)
```

Wartością wyrażenia

```
foldr (+) 0 $ take 10 . iterate (*2) $ 1
```

jest

- ☐ a. lista `[1,2,4,8,16,32,64,128,256,512]`.
- ☐ b. nieskończona lista potęg dwójki.
- ■ c. liczba 1023.
- ☐ d. liczba 1024.

Pytanie 15. Niech e :: `T Integer` będzie pewnym wyrażeniem, gdzie `T` :: $* \to *$ jest pewnym typem należącym do klasy `Monad`. Wtedy wyrażenie

$$\texttt{do}\ \{\ x \leftarrow e;\ \texttt{return}\ x\ \}$$

- ☐ a. nie ma typu.
- ☐ b. ma typ `Integer`.
- ☐ c. jest równe x.
- ■ d. jest równe e.

Pytanie 16. Rozważmy standardowe funkcje `foldr` i `unfoldr`, przy czym — dla przypomnienia:

```
foldr (+) c [] = c
foldr (+) c (x:xs) = x + foldr (+) c xs

unfoldr f b =
   case f b of
      Just (a,b') -> a : unfoldr f b'
      Nothing -> []
```

Wyrażenie `unfoldr . foldr`

- ■ a. nie posiada typu.
- ☐ b. ma typ `a -> a`.
- ☐ c. ma typ `[a] -> [a]`.
- ☐ d. jest równe `id`.

Pytanie 17. Wyrażenie `foldr undefined 'a' []`

- ☐ a. nie posiada typu.
- ■ b. ma typ `Char`.
- ☐ c. ma typ `[a] -> Char`.
- ☐ d. ma wartość $\perp$.
- ☐ e. ma wartość `[]`.

Pytanie 18. Niech

```
data Cos a = Cos a
newtype ToSamo a = ToSamo a
```

Wtedy

- ☐ a. `Cos` $\perp = \perp$.
- ■ b. `ToSamo` $\perp = \perp$.
- ☐ c. `Cos` $\perp =$ `ToSamo` $\perp$.
- ☐ d. `ToSamo` $x = x$ dla dowolnego x.

Pytanie 19. Zaprogramuj w Haskellu funkcję

```
levels :: [a] -> [[a]]
```

spełniającą następującą specyfikację:

```
        (>>= id) . levels  =  id
   map length . levels  =  unfoldr f . (,1) . length
```

gdzie

```
f (m,n)
   | m == 0 = Nothing
   | m <= n = Just (m, (0, undefined))
   | otherwise = Just (n, (m-n, 2*n))
```

Możesz używać funkcji z modułu `Prelude`.

```
levels = levelsWith 1 where
   levelsWith _ [] = []
   levelsWith n xs = take n xs : levelsWith (2*n) (drop n xs)
```

(Por. zadanie 6.3.1 z podręcznika, str. 194.)

Pytanie 20. Podaj typ wyrażenia `map map`

```
[a -> b] -> [[a] -> [b]]
```

Pytanie 21. Niech

```
(1)  map f []     = []
(2)  map f (x:xs) = f x : map f xs
(3)  (f . g) x = f (g x)
```

Udowodnij, że dla dowolnych funkcji f i g odpowiednich typów zachodzi równość

$$\mathtt{map}\ (f\ .\ g) \;=\; \mathtt{map}\ f\ .\ \mathtt{map}\ g$$

Na mocy zasady ekstensjonalności wystarczy pokazać, że dla każdej listy $xs :: [a]$ zachodzi

$$\mathtt{map}\ (f\ .\ g)\ xs \;=\; (\mathtt{map}\ f\ .\ \mathtt{map}\ g)\ xs.$$

Dowód ostatniej równości przeprowadzimy przez indukcję strukturalną względem xs.

Przypadki bazowe: Dla $xs = \mathtt{[]}$ mamy:

$$L \equiv \mathtt{map}\ (f\ .\ g)\ \mathtt{[]} \stackrel{(1)}{=} \mathtt{[]} \stackrel{(1)}{=} \mathtt{map}\ f\ \mathtt{[]} \stackrel{(1)}{=} \mathtt{map}\ f\ (\mathtt{map}\ g\ \mathtt{[]}) \stackrel{(3)}{=} (\mathtt{map}\ f\ .\ \mathtt{map}\ g)\ \mathtt{[]} \equiv R.$$

Dla dowolnej funkcji h funkcja $\mathtt{map}\ h$ jest *strict* (bo w (1) i (2) występuje dopasowanie wzorca), tj.

(4) $\mathtt{map}\ h\ \perp = \perp$

Dla $xs = \perp$ mamy zatem:

$$L \equiv \mathtt{map}\ (f\ .\ g)\ \perp \stackrel{(4)}{=} \perp \stackrel{(4)}{=} \mathtt{map}\ f\ \perp \stackrel{(4)}{=} \mathtt{map}\ f\ (\mathtt{map}\ g\ \perp) \stackrel{(3)}{=} (\mathtt{map}\ f\ .\ \mathtt{map}\ g)\ \perp \equiv R.$$

Krok indukcyjny: Przypuśćmy, że:

(Z) $\mathtt{map}\ (f\ .\ g)\ xs = (\mathtt{map}\ f\ .\ \mathtt{map}\ g)\ xs.$

Mamy:

$$\begin{aligned}
L \equiv \mathtt{map}\ (f\ .\ g)\ (x : xs) &\stackrel{(2)}{=} \\
(f\ .\ g)\ x\ :\ \mathtt{map}\ (f\ .\ g)\ xs &\stackrel{(Z)}{=} \\
(f\ .\ g)\ x\ :\ (\mathtt{map}\ f\ .\ \mathtt{map}\ g)\ xs &\stackrel{(3)}{=} \\
(f\ .\ g)\ x\ :\ \mathtt{map}\ f\ (\mathtt{map}\ g\ xs) &\stackrel{(3)}{=} \\
f\ (g\ x)\ :\ \mathtt{map}\ f\ (\mathtt{map}\ g\ xs) &\stackrel{(2)}{=} \\
\mathtt{map}\ f\ (g\ x\ :\ \mathtt{map}\ g\ xs) &\stackrel{(2)}{=} \\
\mathtt{map}\ f\ (\mathtt{map}\ g\ (x : xs)) &\stackrel{(3)}{=} \\
(\mathtt{map}\ f\ .\ \mathtt{map}\ g)\ (x : xs) &\equiv R.
\end{aligned}$$

Pytanie 22. Oto funkcja, która przekształca liczbę całkowitą w napis zawierający jej dziesiętną reprezentację:

```
itoa :: Integer -> String
itoa n = (case n `compare` 0 of
            LT -> '-' : s
            EQ -> "0"
            GT -> s) where
         s = reverse . unfoldr f . abs $ n
         conv x = toEnum . fromEnum $ x
         f ...
```

Dokończ ten program, tj. napisz brakującą definicję funkcji `f`. Użyj w niej funkcji `conv` zdefiniowanej wyżej za pomocą standardowych funkcji

```
toEnum :: Enum a => Int -> a
fromEnum :: Enum a => a -> Int
```

Typy `Integer` i `Char` należą do klasy `Enum`. Instancje funkcji `toEnum` i `fromEnum` dla typu `Char` zamieniają znaki na odpowiadające im kody Unicode (w szczególności ASCII) i odwrotnie, a dla typu `Integer` dokonują konwersji między odpowiadającymi sobie wartościami typu `Int` i `Integer`.

```
f m
   | m == 0 = Nothing
   | otherwise = Just (conv $ m `mod` 10 + conv '0', m `div` 10)
```