

## Metody Programowania 2022 – Egzamin

**D**

28 czerwca 2022 r.

Oceny (progi): 14 pkt – 3.0, 17 pkt – 3.5, 20 pkt – 4.0, 23 pkt – 4.5, 26 pkt – 5.0

## Zadanie 1. (2 pkt)

W poniższych wyrażeniach podkreśl wolne wystąpienia zmiennych. Dla każdego związanego wystąpienia zmiennej, narysuj strzałkę od tego wystąpienia do wystąpienia wiążącego je.

`(lambda (y)`

```

(let ((x 3))
  (let ((y x))
    (f x y x)))

```

```

(let ((x z)
      (z x))
  (+ z 2 x))

```

```

(let* ((x 42)
       (y (+ x 3)))
  (lambda (f x)
    (f x y)))

```

## Zadanie 2. (3 pkt)

W poniższych trzech definicjach procedur za literami A, ..., E ukryto wszystkie wystąpienia procedur `list`, `cons` i `append`. Odkryj wywołanie których z tych procedur powinno znajdować się w oznaczonych miejscach tak, by procedury te zachowywały się zgodnie z nazwą i opisem.

- Wybierz  $n$  pierwszych elementów listy. Jeśli lista jest krótsza niż  $n$ , wartością procedury jest cała lista:

```

(define (take n xs)
  (if (or (= n 0) (null? xs))
      null
      (A (car xs) (take (- n 1) (cdr xs)))))

```

A = cons

- Dodaj element na koniec listy

```

(define (snoc xs e)
  (if (null? xs)
      (B e)
      (C (car xs) (snoc (cdr xs) e))))

```

B = list      C = cons

- Utwórz listę biorąc na zmianę kolejne elementy z list, np.

```

> (interleave '(1 2 3 4 5 6) '(a b c d e))
'(1 a 2 b 3 c 4 d 5 e 6)

```

Gdy jedna lista jest pusta, wstaw resztę elementów drugiej listy:

```

(define (interleave xs ys)
  (cond [(null? xs) ys]
        [(null? ys) xs]
        [else (D (E (car xs) (car ys))
                    (interleave (cdr xs) (cdr ys)))]))

```

D = append      E = list

### Zadanie 3. (5 pkt)

Zdefiniuj procedurę merge, scalającą dwie posortowane listy liczb w posortowaną listę liczb zawierającą wszystkie elementy list podanych jako argumenty (wraz z powtórzeniami).

Przykładowo, obliczenie (merge '(1 2 3) '(2 4 6)) powinno dać w wyniku listę '(1 2 2 3 4 6).

(define (merge xs ys)

```
(cond [(null? xs) ys]
      [(null? ys) xs]
      [(if (<= (car xs) (car ys))
            (cons (car xs) (merge (cdr xs) ys))
            (cons (car ys) (merge xs (cdr ys))))])
```

Napisz (jednym słowem) czy Twoja definicja generuje nieużytki: Nie.

Korzystając z procedury merge zdefiniowanej powyżej oraz procedury split, której specyfikację podajemy poniżej, zaimplementuj procedurę mergesort, sortującą listę liczb w kolejności niemalejącej algorytmem sortowania przez scalanie. Zakładamy że procedura split dzieli listę na połowy i zwraca parę list (możemy założyć że w przypadku gdy lista jest nieparzystej długości, druga z otrzymanych list jest dłuższa niż pierwsza): przykładowo obliczenie (split '(5 4 3 2 1)) jest równoważne obliczeniu (cons '(5 4) '(3 2 1)).

**Uwaga:** procedury split nie trzeba definiować, zakładamy że jest dana!

(define (mergesort xs)

```
(if (<= (list-length xs) 1)
    xs
    (merge (mergesort
            (let ([yss (split xs)])
              (merge (mergesort (car yss)) (mergesort (cdr yss))))))
```



#### Zadanie 4. (5 pkt)

5/5

Rozważmy strukturę danych podobną do list, ale która pozwala na konkatenację w czasie stałym: konkatenacja jest dodatkowym konstruktorem tego typu danych. W języku Plait taką strukturę zdefiniujemy przy pomocy następującego typu danych.

```
(define-type (AList 'a)
  (a-null)
  (a-cons [hd : 'a] [tl : (AList 'a)])
  (a-append [l : (AList 'a)] [r : (AList 'a)]))
```

Sformułuj zasadę indukcji dla tego typu danych.

Niech  $W$  będzie własnością  $AList$ , jeżeli zachodzą warunki:

- i.) Zachodzi  $W(\overset{(a-null)}{\text{empty}})$
- ii.) Dla dowolnego elementu  $x$  typu  $'a$  <sup>lub dowolnej listy  $(AList 'a)$</sup>  jeżeli zachodzi  $W(\overset{x}{\text{AList } a})$  to zachodzi  $W((a-cons \ x \ xs))$
- iii.) Dla dowolnych list  $xs, ys$  typu  $(AList 'a)$  jeżeli zachodzi  $W(xs) \wedge W(ys)$  to zachodzi również  $W((a-append \ xs \ ys))$

Wtedy  $W$  jest własnością  $W$  spełnioną dla wszystkich  $AList$ .

Rozważmy następujące dwie funkcje, które odpowiednio odwracają rozważane listy oraz transformują je do zwykłych list.

```
(define (a-rev xs)
  (type-case (AList 'a) xs
    [(a-null) (a-null)]
    [(a-cons x xs) (a-append (a-rev xs) (a-cons x (a-empty)))]
    [(a-append l r) (a-append (a-rev r) (a-rev l))]))
```

```
(define (to-list xs)
  (type-case (AList 'a) xs
    [(a-null) empty]
    [(a-cons x xs) (cons x (to-list xs))]
    [(a-append l r) (append (to-list l) (to-list r))]))
```

Dodatkowo przyjmij poniższą definicję standardowej funkcji reverse, działającą w czasie kwadratowym.

```
(define (reverse xs)
  (type-case (Listof 'a) xs
    [empty empty]
    [(cons x xs) (append (reverse xs) (list x))]))
```







### \* Zadanie 5. (3 pkt)

Cykliczna lista łączona składa się z węzłów, z których każdy wskazuje na swojego następnika (następnikiem ostatniego węzła jest pierwszy węzeł). W tym zadaniu węzły cyklicznej listy łączonej wzbogacimy o wskaźnik na następnika swojego następnika: dla dowolnego węzła  $e$  powinno zachodzić ( $\text{eq? (node-next (node-next e)) (node-next-next e)}$ )).

```
(define-struct node (elem [next #:mutable] [next-next #:mutable]))
```

Zaimplementuj wstawianie nowego węzła zawierającego podany element (elem) do listy za następnikiem podanego węzła (node).

```
(define (insert-after-next node elem)
```

```
(set-node-next let new (make-node (node elem (node-next (node-next node))))
  (if (not (eq? (node-next (node-next-next (node-next node)))
               next!
               (node-next-next (node-next node))))
      (set-node-next! (node-next node) (node-next-next (node-next node)))
      (set-node-next! (node-next node) new))
  (set-node-next-next! node new)))
```

```
(letrec rec ([new (make-node elem (node-next-next node))
  (if (not (eq? (node-next node) (node-next-next node)))
      (node-next-next node (node-next node))
      (node-next-next (make-node e
  (begin
    (node-next-next node)
    (set-node-next-next! new new)))]
```

### Zadanie 6. (2 pkt)

- Uzupełnij kontrakt w poniższej definicji standardowej procedury `append-map`. Kontrakt powinien zostać naruszony dokładnie wtedy, gdy procedurze przekazano niewłaściwe dane, które prowadzą do błędu wykonania.

```
(define/contract (appendmap f xs)
  (parameteric (listof b) k -> [a] [b] (-> (a -> amap) (listof a) (listof b)))
  (if (null? xs)
      null
      (append (f (car xs)) (appendmap f (cdr xs)))))
```

- Napisz (jednym zdaniem) czym różnią się kontrakty od typów.

Kontrakty są sprawdzane w trakcie działania programu, gdy funkcje są wywoływane, a typy są sprawdzane w czasie kompilacji.

### Zadanie 7. (3 pkt)

Czy następujące wyrażenia są dobrze typowane w systemie typów języka Plait i jeśli tak, to jaki jest ich typ główny?

1. `(lambda (x) (lambda (y) (lambda (z) ((x z) (y z))))))`

✓  $((a \rightarrow b) \rightarrow ((a \rightarrow b) \rightarrow (a \rightarrow c)))$

2. `((lambda (x) (x x)) (lambda (x) (x x)))`

✓ Nie jest dobrze typowane

3. `((lambda (x) x) (lambda (x) x), 42)`

✓ Number

### Zadanie 8. (2 pkt)

Liczby Padovana są zdefiniowane przy pomocy równania rekurencyjnego w podobny sposób, jak liczby Fibonacciego:

$$P_0 = 1 \quad P_1 = 1 \quad P_2 = 1 \quad P_{n+3} = P_{n+1} + P_n$$

Zdefiniuj w języku Racket ciąg liczb Padovana w postaci nieskończonego strumienia, używając procedur `stream-cons`, `stream-cdr` i `map2` znanych z wykładu. Eleganckie rozwiązanie nie używa rekurencyjnych procedur pomocniczych; za rozwiązanie używające takich procedur można uzyskać maksymalnie 1 punkt.

`(define padovan-numbers`

2 `(stream-cons 1 (stream-cons 1 (stream-cons 1 (map2 + padovan-numbers (stream-cdr padovan-numbers))))))`



### Zadanie 9. (5 pkt)

W tym zadaniu rozważamy prosty język funkcyjny, dla którego niekompletne definicje składni abstrakcyjnej (typ `Exp`), typu wartości (typ `Value`) oraz ewaluatora (procedura `eval`) pokazane są poniżej. Ewaluator korzysta ze standardowych funkcji pomocniczych dodawania wartości liczbowych (funkcja `add`), aplikowania (domknięcia) funkcji do wartości (funkcja `apply`) oraz obsługi środowiska (funkcje `lookup-env` i `extend-env`), których definicje są takie jak na wykładzie.

Należy rozszerzyć język o dwie jednoargumentowe formy specjalne: `delay` i `force`. Konstrukcja `delay` ma za zadanie odroczyć ewaluację swojego argumentu zwracając wartość reprezentującą odroczenie. Konstrukcja `force` ma za zadanie zwrócić wartość swojego argumentu: jeżeli argumentem jest odroczone obliczenie, to `force` wymusza jego obliczenie, a w przeciwnym razie zachowuje się jak funkcja identycznościowa. **Uwaga:** nie wymagamy by wartość odroczonego obliczenia była spamiętywana. Istotnym natomiast jest, by Twoja implementacja była zgodna ze statycznym wiązaniem zmiennych.

Uzupełnij brakujące fragmenty kodu w poniższych definicjach typów `Exp` i `Value` oraz funkcji `eval`.

```
(define-type Exp
  (numE [n : Number])
  (addE [e1 : Exp] [e2 : Exp])
  (varE [x : Symbol])
  (letE [x : Symbol] [e1 : Exp] [e2 : Exp])
  (lamE [x : Symbol] [e : Exp])
  (appE [e1 : Exp] [e2 : Exp])
```

```
(delayE [e : Exp])
(forceE [e : Exp])
```

```
(define-type Value
  (numV [n : Number])
  (funV [x : Symbol] [e : Exp] [env : Env])
```

```
(delV [e : Exp] [env : Env])
```

```

(define (eval [e : Exp] [env : Env]) : Value
  (type-case Exp e
    [(numE n)
     (numV n)]
    [(addE e1 e2)
     (add (eval e1 env) (eval e2 env))]
    [(varE x)
     (lookup-env x env)]
    [(letE x e1 e2)
     (let ([v1 (eval e1 env)])
       (eval e2 (extend-env env x v1)))]
    [(lamE x b)
     (funV x b env)]
    [(appE e1 e2)
     (apply (eval e1 env) (eval e2 env))])

```

```

[ (delE e)
  (delV e env) ]

```

```

[ (letE e)

```

~~(type-case Exp e)~~

```

  (let ([v1 (eval e env)])

```

(type-case Value v

```

  ([delV e env1]

```

```

    (eval e env1)

```

```

    (else v))) ] ) )

```

Zakładając, że mamy do dyspozycji parser `parse`, a puste środowisko jest reprezentowane przez `mt-env`, podaj wynik wywołania

```

> (eval (parse '{let f {let y 1 {lambda {x} x}}
                {let y 2
                  {f {delay y}}}})
  mt-env)

```

przy swojej implementacji. Możesz przyjąć listową reprezentację środowiska.

```

(apply (funV x (varE x) env) (delV (varE y) env'))

```

W `env` na wierzchu jest `y=1`, a w `env'` `y=2`

```

(delV (varE y) env')

```