Metody Programowania 2022 — Egzamin

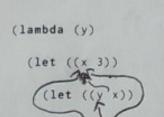


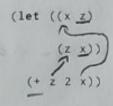
28 czerwca 2022 r.

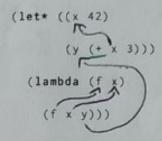
Oceny (progi): 14 pkt - 3.0, 17 pkt - 3.5, 20 pkt - 4.0, 23 pkt - 4.5, 26 pkt - 5.0

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.







Zadanie 2. (3 pkt)

W poniższych trzech definicjach procedur za literami A, ..., E ukryto wszystkie wystąpienia procedur 1 i st, 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 = COMS

Dodaj element na koniec listy

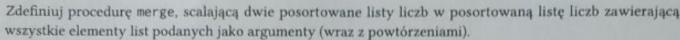
Utwórz listę biorąc na zmianę kolejne elementy z list, np.

```
> (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:



Zadanie 3. (5 pkt)



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

(define (merge xs 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)

x5

(merge (mergesort

(let ([yss (split xs)])

(merge (mergesort (car yss)) (mergesort (cdr yss)))))))
```

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)]))
```

Sformuluj zasadę indukcji dla tego typu danych.

```
Niech W będrie włosnością A List, jeżeli zachodzą
warunki:

i.) Zachodri W (anthy)

ii) Dla dowolnego elementu typu 'a 'jeśli zachodri
W (Atistra) to zachodri W ((a-cons x x s))

iii) Dla dowolnych list xs, ys typu (AList'a) jeśli
zachodri W(xs) r w (ys) to zachodri również
W (a-append xs ys))

U tedy U jest włosnością costo spełnioną dla wszystkóli
AList.
```

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

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))]))
```

Pokaż, że podana implementacja odwracania rozważanych list jest zgodna z odwracaniem standardowych list, tzn. pokaż, że dla dowolnego xs typu (AList 'a) zachodzi (to-list (a-rev xs)) ≡ (reverse (to-list xs)). Twój dowód prawdopodobnie będzie wymagał kilku lematów dotyczących standardowych

funkcji takich jak append albo reverse. Nie musisz ich dowodzić – wystarczy, że je sformulujesz. Niech w bedrie viamosaia Alist e polecenia Niech XS = (a-mull) L= (to-list (a-rev (a-mull))) = (to-list (a-mull)) = empty P = (reverse (to-list (a-mull))) = (reverse emply) = empty ii.) Wie totoing, Dering douding x typu 'a i xs typu (AList'a), reachodri. Poloriem, rie radiodrisk (a-cons x xs)) L = (to-list (a-rev (a-cong x xs))) = (to-list (a-append (a-rev xs)) (a-cons x (a-null)))) = (append (to-list (a-rev xs)) (to-list (a-cons × (a-mull))) = (append (reverse (to-list xs)) (cons × emty))

Dlo dovolnego × i dovolnej listy xs

demotr. (append (reverse xs) (cons × emtypty)) = (reverse (cons × xs)) P = (reverse (to-list (a-cons x x5)) = (reverse (cons x (to-list x5))) Pouvoring, re (to-list xs) zaroeo obliczo się do listy Plaitougi Notem moriemy skorzystać z nesrego demetu 1 i przeksntolcić L L = (reverse (coms x (to-list x5))) = P

cii) Verinny dovolne xs, ys typu (AList'a)
Poloriemy, rie w ((a-appand xs ys)) i roloringie W(xs): W(ys)

L = (to-list (a-rev la-append xs ys))) = (to-list (a-append (a-rev))

(a-rev (s))) = (append (to-list (a-rev (s))) (to-list (a-rev (s)))=

(append (reverse (to-list xx)) (reverse (to-list xx)))

P = (reverse (to-list xx)) = (reverse (append (to-list xx) (to-list yx)))

demot? Dla doublingch list xs, ys reduced i (oppend (reverse xs));

(reverse ()) = (reverse Coppend Ks ys))

Korrystojąc z demotu z przeksrtotcomy L i otorymujemy

L= (reverse (append (to-list xs) (to-list ys)))=P

Zotem viosnosi W rachodri ollo usry stkich AList

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 (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)

```
(sate mode leters (The worker node)

(if Inot (eq?(mode next (mode - mext - mext (mode - mext mode))))

(set - mode - mext) (mode - mext mode) (Mode (mode - mext - mext mode))

(set - mode - mext! (mode - mext mode) mew)

(set - mode - mext - mext! mode mew)))

(letime([mex (make - mode elem (mode - mext - mext mode)))

(if (mot (eq? (mode - mext mode) (mode - mext - mext mode)))

(mode - mext - mext (mode - mext mode))

(mode - mext - mext (mode - mext mode))

(mode - mext - mext (mode - mext mode))

(mode - mext - mext (mode - mext mode))

(mode - mext - mext (mode - mext - mext))
```

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

```
(define/contract (appendimap f xs)

(parametric/c -> [a][b] (-> (a-> a) (Listolb)

(if (null? xs)

null

(append (f (car xs)) (appendimap f (cdr xs)))))
```

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

Kontrokty sa sproudrone u trolcie driolonie programu, gdy lunkée sa uyuolyuone, a typy sa sproudrone u crosie 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?

2. ((lambda (x) (x x)) (lambda (x) (x x))),

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

Zadanie 8. (2 pkt)

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

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

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

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.

Uzupelnij 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] [e1 : Exp] [e2 : Exp])

(lame [x : Symbol] [e : Exp])

(appE [e1 : Exp] [e2 : Exp])

(delugE[e : Exp])

(forent E [e : Exp])
```

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

(del V [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))]
   [(del E e)
    (del v e env)]
   [(for E e)
     Myse-cose Expre lessone
        et (let 4(IV (eval e env)))
                (type-case Value v
                  (Edelverenvi]
                      (evol en enva)
                   (telse v4)))] ))}
```

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

przy swojej implementacji. Możesz przyjąć listową reprezentację środowiska.