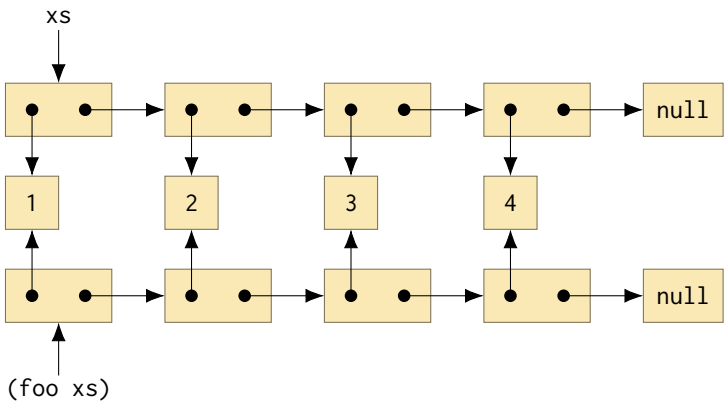
Lista zadań nr 5

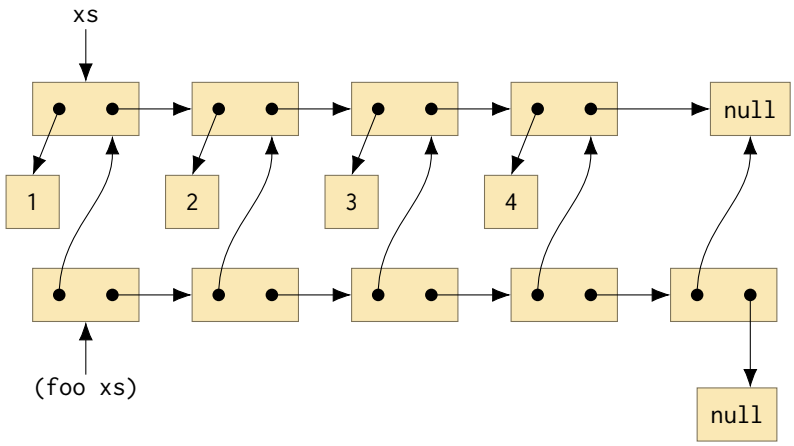
Ćwiczenie 1.

W każdym podpunkcie rozpoznaj tajemniczą procedurę `foo`, która tworzy w pamięci dane struktury. Jeśli jest to procedura omawiana już na wykładzie lub ćwiczeniach, nazwij ją. Jeśli jeszcze się nie pojawiła, napisz jej definicję.

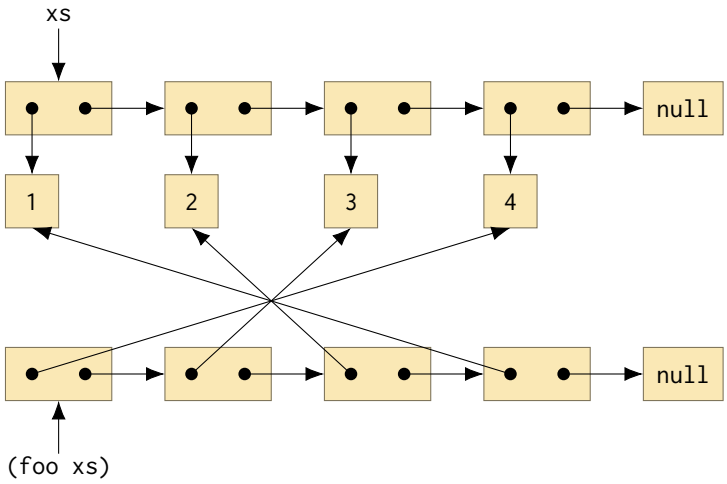
(1)



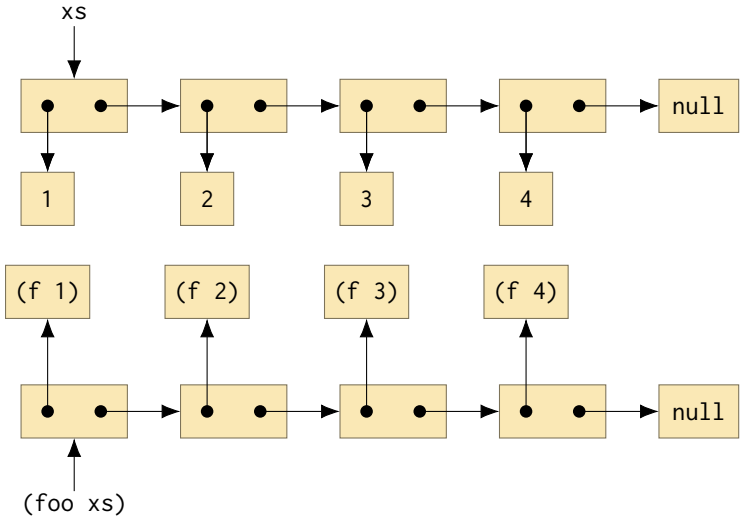
(2)

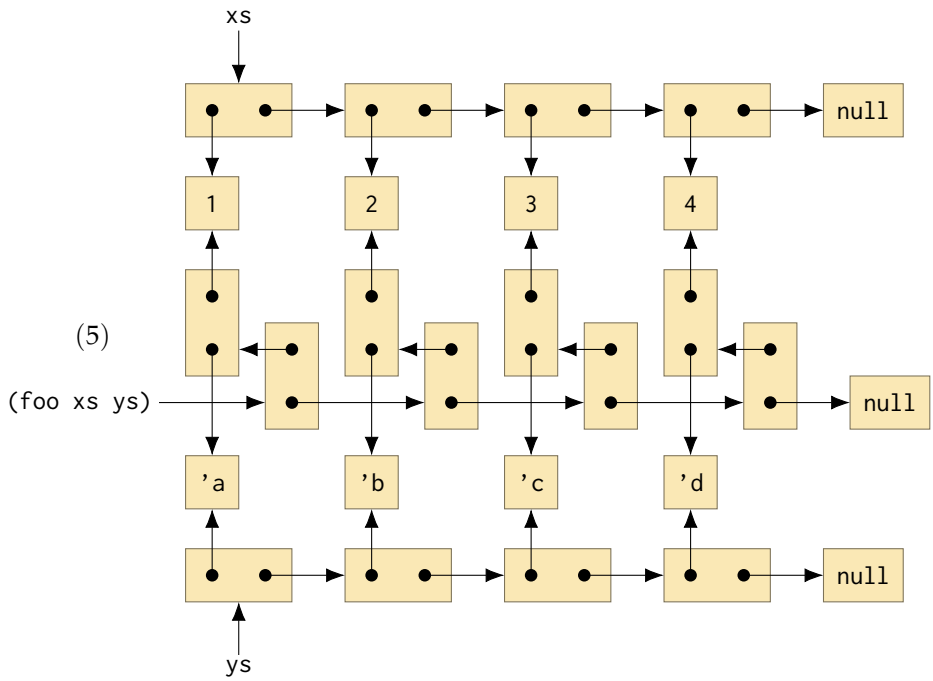


(3)



(4)



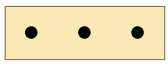


Ćwiczenie 2.

Rozważ drzewa binarne z wykładu. Narysuj, jak w pamięci reprezentowane jest drzewo `t`:

```
(define t
  (node 5 (node 2 leaf leaf)
          (node 8 (node 6 leaf leaf)
                  (node 9 leaf leaf))))
```

Dla ułatwienia rysuj pojedynczego node-a jako trójkę



Pokaż, jak będzie wyglądał stan pamięci po wykonaniu wstawienia BST wartości `7`. Które fragmenty drzewa `t` są współdzielone między drzewem `t` i `(innsert-bst 7 t)`?

Ćwiczenie 3.

Rozważ następującą procedurę `flip`:

```
(define (flip t)
  (if (leaf? t)
      leaf
      (node (node-val t)
            (flip (node-right t))
            (flip (node-left  t)))))
```

Udowodnij, że dla każdego drzewa `t` zachodzi `(flip (flip t))` $\equiv$ `t`.

Ćwiczenie 4.

Tak jak najprostszy schemat rekursji na listach można wyabstrahować do procedury `foldr`, tak najprostszy schemat rekursji na drzewach z wykładu można wyabstrahować do procedury `fold-tree`. Zdefiniuj tę procedurę i wyraź przy jej użyciu procedury[1] `tree-sum`, `flip`, `height` (wysokość drzewa), `tree-span` (para wartości skrajnie prawego i skrajnie lewego węzła w drzewie) i `tree-max` (maksymalna wartość w drzewie).

Ćwiczenie 5.

Zdefiniuj procedurę `sum-paths`, która dla drzewa `t` etykietowanego liczbami produkuje drzewo o takim samym kształcie, w którym etykietą danego węzła jest suma wartości węzłów znajdujących się na ścieżce prowadzącej od korzenia do odpowiadającego wierzchołka w drzewie `t`. Np.

```
> (define tt
 (node 1 (node 2  (node 3  leaf leaf)
                  (node 30 leaf leaf))
         (node 9  (node 2  leaf leaf)
                  (node 3  leaf leaf))))
> (sum-paths tt)
'(node 1 (node 3  (node 6  leaf leaf)
                  (node 33 leaf leaf))
         (node 10 (node 12 leaf leaf)
                  (node 13 leaf leaf)))
```

Wersja zaawansowana: Wyraź procedurę `sum-paths` używając `fold-tree`, a nie używając otwarcie rekursji.

Wskazówka dla wersji zaawansowanej: Użyj procedur jako wyniku `fold-tree`.

[1] Dla uproszczenia możemy założyć, że wartości w węzłach to liczby.

Ćwiczenie 6.

Udowodnij, że procedury `flatten-1.0` i `flatten-2.0` z wykładu są równoważne.

Ćwiczenie 7.

Zdefiniuj typ danych `rose-tree`, który jest drzewem z wartościami w liściach i węzłach, gdzie każdy węzeł może mieć dowolnie wiele poddrzew (tak więc liść jest tożsamy z węzłem z zerową liczbą poddrzew). Sformułuj twierdzenie o indukcji dla tego typu danych.

Ćwiczenie 8.

Kolejka FIFO (*first in, first out*) to struktura danych, do której można dodać element „na koniec", a także podejrzeć i wyjąć element „na początku". Prosta implementacja kolejki przy pomocy listy może wyglądać tak:

```
(define empty-queue       ; pusta kolejka
  null)

(define (empty? q)        ; czy kolejka jest pusta?
  (null? q))

(define (push-back x q)   ; dodaj element na koniec kolejki
  (append q (list x)))

(define (front q)         ; podejrzyj element na poczatku kolejki
  (car q))

(define (pop q)           ; zdejmij element z przodu kolejki
  (cdr q))
```

Ta implementacja kolejki jest bardzo czytelna, ale ma wielką wadę związaną z użyciem procedury `append` w implementacji procedury `push-back`. Lepszą reprezentajcą kolejki jest **para list**: pierwsza to prefiks kolejki, a druga to sufiks w odwróconej kolejności. W ten sposób mamy łatwy dostęp do przodu kolejki (bo reprezentowany jest przez przód listy) i do tyłu kolejki (bo też reprezentowany jest przez przód listy). Dokładniej: pierwszy element kolejki to pierwszy element pierwszej listy, a nowe elementy możemy dokładać poprzez dokładanie ich na przód drugiej listy. Dopiero gdy skończą się elementy na pierwszej liście, zastępjemy ją odwróconą drugą listą.

Sformalizuj ten nieformalny opis poprzez lepszą implementację podanego wyżej interfejsu. By zawsze mieć dostęp do pierwszego elementu kolejki, zachowaj następujący niezmiennik: pierwsza lista jest pusta tylko wtedy, gdy druga lista jest pusta.