Racket la supermarket

- Data publicării: 09.03.2021
- Data ultimei modificări: 26.03.2021 (changelog)
- Tema (o arhivă .zip cu fișierul supermarket.rkt (și queue.rkt pentru etapele 3-4)) se va încărca pe vmchecker [https://vmchecker.cs.pub.ro/ui/#PP]

Descriere generală și organizare

Tema constă într-o aplicație care simulează fluxul clienților pe la casele unui supermarket.

Tema este împărțită în **4 etape**:

- una pe care o veți rezolva după laboratorul 2 (cu deadline în ziua laboratorului 3, la ora 23:59)
- una pe care o veți rezolva după laboratorul 3 (cu deadline în ziua laboratorului 4, la ora 23:59)
- una pe care o veți rezolva după laboratorul 4 (cu deadline în ziua laboratorului 5, la ora 23:59)
- una pe care o veți rezolva după laboratorul 5 (cu deadline în ziua laboratorului 6, la ora 23:59)

Așa cum se poate observa, **ziua deadline-ului variază în funcție de semigrupa în care sunteți repartizați**. **Restanțierii care refac tema și nu refac laboratorul beneficiază de ultimul deadline** (deci vor avea deadline-uri în zilele de 23.03, 30.03, 06.04, 13.04).

Rezolvările tuturor etapelor pot fi trimise până în ziua laboratorului 6, dar orice exercițiu trimis după deadline se punctează cu **jumătate** din punctaj. Nota finală pe temă se calculează conform formulei: **n = (n1 + n2) / 2** (n1 = nota obținută înainte de deadline; n2 = nota obținută după deadline). Când toate submisiile sunt înainte de deadline, nota pe ultima submisie este și nota finală (întrucât n1 = n2).

În fiecare etapă, veți folosi ce ați învățat în săptămâna anterioară pentru a perfecționa simulatorul (ori ca performanță, ori ca număr de situații pe care este capabil să le modeleze).

Pentru fiecare etapă veți primi un **schelet de cod** (dar rezolvarea se bazează în mare măsură pe rezolvările anterioare). Intenția este să puteți rezolva tema utilizând doar indicațiile din schelet (fără a fi necesar să citiți enunțul). Enunțul încearcă să lămurească aspectele care poate nu sunt clare tuturor doar din schelet.

Etapa 1

În această etapă presupunem că supermarket-ul are fix 4 case ("counters" în engleză): C1, C2, C3, C4. Fiecare casă este reprezentată ca o structură:

```
(define-struct counter (index tt queue))
```

- index
  - în cazul nostru, este un număr de la 1 la 4 (1 pentru C1, 2 pentru C2, etc.)
- tt
  - vine de la "total time", și reprezintă timpul total de așteptare la această casă: dacă un client se așază acum la coadă, el va avea de așteptat tt unități de timp până să ajungă în față (pentru convenință vom considera unitatea de timp ca fiind 1 minut, chiar dacă nu este realist)
  - depinde de numărul de produse cumpărate de clienții din coadă (1 produs = 1 minut) și de eventualele întârzieri suferite de casa respectivă
- queue

    - este o listă de perechi (nume . nr_produse), reprezentând persoanele așezate la coadă la această casă (și câte produse au cumpărat ele)
    - primul element din listă trebuie să corespundă primului client care s-a așezat la coadă

Deși nu am studiat structuri la curs sau laborator, utilizarea lor este simplă (și ne ajută să avem un cod mai lizibil). Aveți aici un tutorial foarte scurt cu tot ce vă trebuie pe partea de structuri și pattern matching.

Statutul caselor diferă astfel:

- C2-C4 sunt deschise tuturor clienților
- C1 acceptă doar clienți care au cumpărat maxim ITEMS produse (ITEMS este o constantă setată în scheletul de cod)

În această etapă, simulatorul trebuie să modeleze doar 2 tipuri de situații:

- situația în care un nou client dorește să se așeze la coadă cu coșul său de cumpărături
- situația în care activitatea unei case este întârziată (din diverse motive neprevăzute) cu un număr de minute

Funcțiile principale pe care va trebui să le implementați sunt:

```
(add-to-counter C name n-items)
```

- add-to-counter adaugă în coada casei C persoana name cu n-items produse (ceea ce se adaugă este o pereche care conține ambele informații)
- ex: `(add-to-counter C2 'ana 12)` va determina adăugarea perechii `'(ana . 12)` în câmpul queue al lui C2

```
(min-tt counters)
```

- min-tt determină casa din counters care are tt minim, și întoarce perechea dintre indexul acestei case și valoarea tt-ului ei) (când are de ales între mai multe case, o va alege pe cea cu index minim)
- ex: `(min-tt (list (counter 1 10 '()) (counter 2 12 '((ana . 12))))) ⇒ '(1 . 10)`

```
(serve requests C1 C2 C3 C4)
```

- serve primește o listă de cereri (așezări la coadă, respectiv întârzieri) și le tratează în ordine, în sensul că actualizează C1, C2, C3 și C4 pe măsură ce situația caselor evoluează

Exemplu:

```
(serve '((ana 12) (delay 1 5) (mia 2)) C1 C2 C3 C4)
```

unde presupunem că C1-C4 sunt în prezent lipsite de clienți, iar ITEMS = 5:

- întâi caută să o așeze pe ana la cea mai avantajoasă casă posibilă
    - întrucât ana are 12 produse, ea se poate așeza doar la una dintre C2, C3, C4
    - întrucât toate cele 3 case sunt lipsite de clienți și niciuna nu a suferit întârzieri, vom alege C2 pentru că are index minim
- apoi casa C1 este întârziată cu 5 minute
    - este posibil ca o casă fără clienți să fie întârziată - semnificația este că un client care se așază acum la C1 trebuie să aștepte 5 minute până când cineva îl va lua în primire
- apoi caută să o așeze pe mia la cea mai avantajoasă casă posibilă

- întrucât mia are 2 produse, ea se poate așeza la orice casă
- situația curentă a caselor este: C1 este întârziată cu 5 minute (tt = 5), la C2 stă ana (tt = 12), C3 și C4 nu au clienți și nu sunt întârziate (tt = 0)
- vom alege C3 pentru că, dintre cele cu tt minim, C3 are index minim

Etapa 2

Această etapă își propune exploatarea faptului că funcțiile sunt valori de ordinul întâi. Va trebui să folosiți funcții curry și să abstractizați funcții cu implementări similare. De asemenea, vă încurajăm să valorificați oportunitățile de utilizare a funcțiilor anonime și funcționalelor, deși enunțul nu impune acest lucru.

În această etapă, numărul de case din supermarket nu mai este fixat. Vom avea:

- o listă fast-counters de case care acceptă doar clienți care au cumpărat maxim ITEMS produse
- o listă slow-counters de case deschise tuturor clienților

Pentru ca în etapa următoare să putem determina ordinea ieșirii clienților din supermarket, introducem acum un nou câmp în structura counter:

```
(define-struct counter (index tt et queue))
```

- et
  - vine de la "exit time", și reprezintă timpul rămas până când primul client din coadă va părăsi această casă
  - depinde de numărul de produse cumpărate de acest client (1 produs = 1 minut) și de eventualele întârzieri suferite de casa respectivă

În această etapă, simulatorul trebuie să modeleze atât situațiile de la etapa anterioară, cât și 2 noi situații:

- situația în care cel mai avansat client (din punct de vedere al exit time-ului) părăsește supermarket-ul
- situația în care este necesară deschiderea unor noi case pentru a micșora media timpilor totali de așteptare

În primul rând, va trebui să adaptați o serie de funcții de la etapa 1 astfel încât ele să țină cont de noua reprezentare (adică de numărul variabil de case și de prezența câmpului `'et` în structura de tip casă).

Exceptând aceste adaptări, funcțiile principale pe care va trebui să le implementați sunt:

```
(update f counters index)
```

- update aplică transformarea f casei din counters care are indexul index, și întoarce lista counters actualizată

Exemplu:

```
(update (λ (C) (struct-copy counter C [tt 0]))
        (list (counter 1 2 2 '()) (counter 2 5 5 '()))
        2)
```

⇒ `(list (counter 1 2 2 '()) (counter 2 0 5 '()))`

```
(remove-first-from-counter C)
```

- remove-first-from-counter scoate prima persoană din coada casei C

- tt-ul și et-ul casei C trebuie ajustate în consecință
  - orice întârziere avea casa, ea dispare
  - dispar produsele clientului care pleacă (și minutele asociate acestora)
  - nicio altă casă nu este afectată (este ca și cum ar fi trecut timpul doar pe la casa C; acest lucru se va schimba în etapa 3)

Exemplu:

```
(remove-first-from-counter (counter 1 50 5 '((ana . 3) (leo . 35) (mia . 10))))
```

⇒ `(counter 1 45 35 '((leo . 35) (mia . 10)))`

```
(serve requests fast-counters slow-counters)
```

- serve primește o listă de cereri (așezări la coadă, întârzieri, ieșiri de la casă, ajustări ale numărului de case) și le tratează în ordine, în sensul că actualizează casele din fast-counters și slow-counters pe măsură ce situația lor evoluează

Exemplu:

```
(serve '((ana 8) (mia 2) (mara 14) (ion 7) (remove-first) (ensure 5) (remove-first))
       (list (empty-counter 1) (empty-counter 2))
       (list (empty-counter 3) (empty-counter 4)))
```

pentru ITEMS = 5:

- observăm că avem 2 case fast (pentru simplitate le vom numi C1 și C2) și 2 case slow (le vom numi C3 și C4)
- primele 4 cereri distribuie cei 4 clienți astfel:
  - ana la C3 (prima casă slow cu tt=0) ⇒ `C3 = (counter 3 8 8 '((ana . 8)))`
  - mia la C1 (prima casă fast cu tt=0) ⇒ `C1 = (counter 1 2 2 '((mia . 2)))`
  - mara la C4 (casa slow cu tt minim) ⇒ `C4 = (counter 4 14 14 '((mara . 14)))`
  - ion la C3 (casa slow cu tt minim) ⇒ `C3 = (counter 3 15 8 '((ana . 8) (ion . 7)))`
- remove-first caută cel mai avansat client pentru a-l scoate de la casă
  - cel mai avansat client este mia (et=2)
  - ea este scoasă de la C1 ⇒ `C1 = (counter 1 0 0 '())` (observați tt și et)
- ensure compară media timpilor totali cu 5
  - tt1 + tt2 + tt3 + tt4 = 0 + 0 + 15 + 14 = 29 ⇒ ttmed = 29 / 4 > 5
  - se adaugă o casă slow goală (C5) ⇒ ttmed = 29 / 5 > 5
  - se adaugă o casă slow goală (C6) ⇒ ttmed = 29 / 6 ≤ 5 (deci putem trece la cererea următoare)
- remove-first caută cel mai avansat client pentru a-l scoate de la casă
  - cel mai avansat client este ana (et=8)
  - ea este scoasă de la C3 ⇒ `C3 = (counter 3 7 7 '((ion . 7)))` (observați tt și et)

Etapa 3

Această etapă își propune să sublinieze importanța abstractizării. Va trebui să vă definiți propriul TDA (tip de date abstract) și să oferiți o interfață completă (un set de constructori și operatori) prin care utilizatorul poate manipula valorile tipului, independent de implementarea din spate. Apoi, este esențial ca voi înșivă să folosiți TDA-ul doar prin intermediul interfeței (în etapa 4 veți înțelege și mai bine de ce este esențial).

Veți începe rezolvarea etapei prin a implementa TDA-ul `queue` în fișierul **queue.rkt**.

Acest tip reprezintă o coadă (first-in-first-out) ca pe o structură:

```
(define-struct queue (left right size-l size-r))
```

- left, right
  - sunt stive (last-in-first-out, implementate ca liste Racket)
  - fiecare adăugare în coadă va fi o adăugare în stiva right
  - ex adăugare:
    adaug 1 ⇒ `right = '(1)`,
    adaug 2 ⇒ `right = '(2 1)`
  - fiecare scoatere din coadă va fi o scoatere din stiva left (când left este vidă, va trebui mai întâi să mutăm toate elementele din right în left, apoi să scoatem din left)
  - mutarea este rezultatul unor operații pop (din right) + push (în left) repetate
  - ex mutare:
    mut în `left = '()` din `right = '(2 1)` ⇒
    `left = '(2)`, `right = '(1)` (primul pop din right îl extrage pe 2 și îi face push în left) ⇒
    `left = '(1 2)`, `right = '()` (apoi pop din right îl extrage pe 1 și îi face push în left)
  - ex scoatere:
    scot din coada cu `left = '()`, `right = '(2 1)` ⇒
    `left = '(1 2)`, `right = '()` (după mutarea elementelor) ⇒
    `left = '(2)`, `right = '()` (după scoaterea primului element)
  - observați că primul element adăugat este primul element scos (first-in-first-out)
- size-l, size-r
  - sunt numere naturale, reprezentând numărul de elemente din cele 2 stive

Sarcina voastră este să implementați interfața TDA-ului queue:

```
empty-queue  :              -> queue  (constructor nular pentru o coadă goală)
queue-empty? :        queue -> Bool   (operator care verifică dacă o coadă este goală)
enqueue      : Elem x queue -> queue  (operatorul de adăugare în coadă)
dequeue      :        queue -> queue  (operatorul de scoatere din coadă)
top          :        queue -> Elem   (operatorul de vizualizare a elementului din vârful cozii)
```

Această reprezentare pentru coadă asigură cost amortizat O(1) pentru operațiile de enqueue și dequeue. Vom folosi această reprezentare pentru câmpul queue al structurii counter, întrucât este mai eficientă decât reprezentarea cu liste Racket din etapele 1 și 2.

După ce ați finalizat implementarea TDA-ului, continuați dezvoltarea simulatorului în fișierul **supermarket.rkt**. (Observație: pe vmchecker veți încărca o arhivă .zip cu fișierele queue.rkt și supermarket.rkt.)

În primul rând, va trebui să adaptați o serie de funcții de la etapa 2 astfel încât ele să țină cont de noua reprezentare (în care câmpul queue din structura counter este de tip coadă (queue) - TDA-ul implementat de voi; este o coincidență că numele câmpului coincide cu numele tipului, în niciun caz nu era necesar acest lucru).

În plus față de etapa anterioară, în această etapă simulatorul trebuie să modeleze trecerea timpului. Până acum, simulatorul trata așezările la cozi, întârzierile și deschiderile de noi case ca și cum s-ar produce în ordine, dar la un același moment de timp. Acest lucru nu corespunde realității - între diversele evenimente este firesc să treacă timp, timp în care clienții avansează la case și, la un moment dat, părăsesc supermarketul.

Exceptând adaptările menționate, funcțiile principale pe care va trebui să le implementați sunt:

```
(pass-time-through-counter minutes)
```

- este o funcție curry (aplicată doar pe un număr de minute, va aștepta un al doilea argument de tip counter)
- odată ce și-a primit (pe rând) argumentele, pass-time-through-counter actualizează tt-ul și et-ul casei în tt-ul și et-ul pe care casa ar trebui să le aibă după trecerea numărului dat de minute
- queue-ul casei nu se modifică, pentru că intenția este de a nu folosi niciodată această funcție pentru a avansa cu un număr de minute mai mare decât timpul până la ieșirea primului client din coadă

Exemplu:

```
((pass-time-through-counter 5)
 (counter 1
          12
          7
          (make-queue '() '((ada . 7)) 0 1)))
```

⇒ `(counter 1 7 2 (queue '() '((ada . 7)) 0 1))`

```
(serve requests fast-counters slow-counters)
```

- serve actualizează casele din fast-counters și slow-counters pe măsură ce situația lor evoluează, pe baza listei `requests` în care elementele sunt:
  - cereri (așezări la coadă, întârzieri, ajustări ale numărului de case)
    sau
  - timpi care trec între cereri

Exemplu:

```
(serve '((ana 14) (mia 2) 5 (ion 7) (delay 1 2) 7)
       (list (empty-counter 1))
       (list (empty-counter 2) (empty-counter 3)))
```

pentru ITEMS = 5:

- observăm că avem o casă fast (o vom numi C1) și 2 case slow (le vom numi C2 și C3)
- primele 2 cereri distribuie cei 2 clienți astfel:
  - ana la C2 ⇒ `C2 = (counter 2 14 14 (queue '() '((ana . 14)) 0 1))`
  - mia la C1 ⇒ `C1 = (counter 1 2 2 (queue '() '((mia . 2)) 0 1))`
- apoi trec 5 minute, după care situația trebuie să fie:
  - mia a ieșit de la C1, care a rămas goală ⇒ `C1 = (counter 1 0 0 (queue '() '() 0 0))`
  - la C2 au trecut 5 minute ⇒ `C2 = (counter 2 9 9 (queue '() '((ana . 14)) 0 1))`
  - C3 a rămas cum era: goală și neîntârziată ⇒ `C3 = (counter 3 0 0 (queue '() '() 0 0))`
  - au părăsit supermarketul, în ordine: `'((1 . mia))` (mia de la C1)
- ion se așază la C3 ⇒ `C3 = (counter 3 7 7 (queue '() '((ion . 7)) 0 1))`
- C1 este întârziată cu 2 minute ⇒ `C1 = (counter 1 2 2 (queue '() '() 0 0))`
- apoi trec 7 minute, după care situația trebuie să fie:
  - întârzierea de la C1 s-a consumat ⇒ `C1 = (counter 1 0 0 (queue '() '() 0 0))`
  - la C2 au trecut 7 minute ⇒ `C2 = (counter 2 2 2 (queue '() '((ana . 14)) 0 1))`
  - ion a ieșit de la C3, care a rămas goală ⇒ `C3 = (counter 3 0 0 (queue '() '() 0 0))`
  - au părăsit supermarketul, în ordine:: `'((1 . mia) (3 . ion))`

Etapa 4

În această etapă veți observa un exemplu interesant de utilizare a fluxurilor. Veți reimplementa TDA-ul queue pentru a obține un plus de performanță și, cu condiția să fi respectat bariera de abstractizare la etapa anterioară (folosind valorile de tip coadă doar prin intermediul interfeței), vă veți bucura de faptul că nu trebuie să modificați nimic în fișierul supermarket.rkt pentru a lucra cu această nouă reprezentare. Volumul de lucru este diminuat (relativ la etapele precedente), dar vă încurajăm să folosiți timpul suplimentar pentru a înțelege în profunzime felul în care ne ajută fluxurile.

Veți începe rezolvarea etapei prin a reimplementa TDA-ul queue în fișierul **queue.rkt**.

Din motive de performanță explicate în detaliu în schelet, vom reprezenta câmpul left al structurii queue ca flux (spre deosebire de reprezentarea ca listă din etapa anterioară). Reamintim definiția structurii (care nu s-a modificat):

```
(define-struct queue (left right size-l size-r))
```

- în continuare, fiecare adăugare în coadă va fi o adăugare în stiva right
- în continuare, fiecare scoatere din coadă va fi o scoatere din stiva left
- după fiecare operație enqueue sau dequeue vom avea grijă să menținem invariantul |left| ≥ |right| (dimensiunea stivei left trebuie să fie mai mare sau egală decât dimensiunea stivei right); astfel, stiva left nu va fi niciodată găsită vidă de o operație de dequeue, și nu vom avea nevoie să facem un dequeue costisitor prin mutarea în timp O(n) a tuturor elementelor din right în left
- de fiecare dată când o operație de enqueue sau dequeue ne aduce în situația |left| = |right| - 1, vom aplica o rotație: vom muta în acel moment toate elementele din right în left, însă, left fiind flux, elementele vor fi mutate, în fapt, unul câte unul, pe măsură ce avem nevoie să extragem elemente din left, nu toate deodată (dacă ar fi deodată nu am rezolva problema complexității, ci doar am muta-o asupra altor operații)

Sarcina voastră este să reimplementați interfața TDA-ului queue (aceleași funcții pe care le-ați implementat în etapa 3). Aceasta depinde de implementarea funcției de rotație:

```
(rotate left right Acc)
```

- funcția calculează (cu evaluare întârziată) rezultatul left ++ (reverse right)
- rotația se efectuează doar atunci când |left| = |right| - 1, așadar găsește un număr echilibrat de elemente în cele două stive
- de fiecare dată când extragem un element din left, vom extrage (“pop”) și unul din right și îl vom adăuga (“push”) în acumulatorul Acc
- când left devine goală, automat right conține un singur element (|left| = |right| - 1), iar toate celelalte elemente care au fost în right se găsesc în Acc, deja inversate; acum putem adăuga elementul din right la începutul Acc (în timp O(1)), și acesta este exact conținutul cu care trebuie să reinițializăm stiva left

Exemplu:

```
(rotate (stream-cons 1 (stream-cons 2 (stream-cons 3 empty-stream)))
        '(7 6 5 4)
        empty-stream)
```

⇒ #<stream>
Mai precis, rezultatul este de forma:

```
(stream-cons 1
             (rotate (stream-cons 2 (stream-cons 3 empty-stream))
```

```
                    '(6 5 4)
                    (stream-cons 7 empty-stream)))
```

și, după cum știm din comportamentul constructorului stream-cons, apelul recursiv al funcției rotate este întârziat. Dacă ulterior vom accesa restul acestui flux, vom evalua noul apel al lui rotate și vom obține un rezultat de forma `(stream-cons 2 (rotate ....))`, etc.

După ce ați finalizat implementarea TDA-ului, continuați dezvoltarea simulatorului în fișierul **supermarket.rkt**.

Simulatorul trebuie să modeleze toate situațiile de la etapa anterioară, plus situația în care o casă este închisă.

Exemplu:

```
(serve '((ana 7) (mia 2) 5 (ion 8) (close 2) (delay 1 10) (dan 2) 3 (ensure 5))
       (list (empty-counter 1))
       (list (empty-counter 2) (empty-counter 3)))
```

pentru ITEMS = 5:

- observăm că avem o casă fast (o vom numi C1) și 2 case slow (le vom numi C2 și C3)
- pentru a ilustra starea caselor vom folosi:
  - reprezentarea unui counter ca o colecție de index, tt, et și queue (deși în această etapă aveți libertatea să modificați structura, dacă doriți)
  - reprezentarea elementelor unui flux între acolade (altfel ar trebui să scriem #<stream>, ceea ce nu este tocmai informativ)
- primele 2 cereri distribuie cei 2 clienți astfel:
  - ana la C2 ⇒ `C2 = (counter 2 7 7 (queue {(ana . 7)} '() 1 0))`
  - mia la C1 ⇒ `C1 = (counter 1 2 2 (queue {(mia . 2)} '() 1 0))`
  - obs: în etapa trecută ana și mia apăreau ca elemente în stiva right, dar aici sunt în stiva left pentru că s-a efectuat o rotație, necesară pentru menținerea invariantului |left| ≥ |right|
- apoi trec 5 minute, după care situația caselor trebuie să fie:
  - mia a ieșit de la C1, care a rămas goală ⇒ `C1 = (counter 1 0 0 (queue {} '() 0 0))`
  - la C2 au trecut 5 minute ⇒ `C2 = (counter 2 2 2 (queue {(ana . 7)} '() 1 0))`
  - C3 a rămas cum era: goală și neîntârziată ⇒ `C3 = (counter 3 0 0 (queue {} '() 0 0))`
- ion se așază la C3 ⇒ `C3 = (counter 3 8 8 (queue {(ion . 8)} '() 1 0))`
- C2 este închisă ⇒ nu poate primi clienți noi, dar funcționează normal până la golirea cozii
- C1 este întârziată cu 10 minute ⇒ `C1 = (counter 1 10 10 (queue {} '() 0 0))`
- dan se așază la C3 ⇒ `C3 = (counter 3 10 8 (queue {(ion . 8)} '((dan . 2)) 1 1))`
  - C2 avea tt-ul mai mic, însă C2 este închisă, deci s-a ales între C1 și C3
- apoi trec 3 minute, după care situația caselor trebuie să fie:
  - întârzierea de la C1 s-a consumat parțial ⇒ `C1 = (counter 1 7 7 (queue {} '() 0 0))`
  - ana a ieșit de la C2, care a rămas goală ⇒ `C2 = (counter 2 0 0 (queue {} '() 0 0))`
  - la C3 au trecut 3 minute ⇒ `C3 = (counter 3 7 5 (queue {(ion . 8)} '((dan . 2)) 1 1))`
- ensure compară media timpilor totali ai caselor deschise cu 5
  - tt1 + tt3 = 7 + 7 = 14 ⇒ ttmed = 14 / 2 > 5
  - tt2 nu participă la medie întrucât C2 este închisă
  - se adaugă o casă slow goală (C4) ⇒ ttmed = 14 / 3 ≤ 5 (deci ne oprim aici cu adăugarea)

Tema noastră-i
Gata astfel.
De vă place paradigma,
V-așteptăm la Tema Haskell!

```
(define Tema1 (list etapa1 etapa2 etapa3 etapa4))
(map close Tema1)
```

Precizări

- Veți implementa funcțiile din fișierul **supermarket.rkt**. Pentru testare, veți rula codul din fișierul **checker.rkt**.
- Tema se va încărca pe vmchecker. Acesta va fi disponibil în curând. Testele de vmchecker sunt aceleași cu cele din checker.rkt.
- Tema este în primul rând o temă de programare funcțională - pentru care folosim Racket. Racket este un limbaj multiparadigmă, care conține și elemente "ne-funcționale" (de exemplu proceduri cu efecte laterale), pe care **nu** este permis să le folosiți în rezolvare.
- Pentru fiecare etapă, checker-ul vă oferă un punctaj între 0 și 120 de puncte. Pentru a obține cele 1.33p din nota finală cu care este creditată tema de Racket, este suficient să acumulați 400 de puncte de-a lungul celor 4 etape. Un punctaj între 400 și 480 se transformă într-un bonus proporțional.
- După completarea celor 4 etape, veți prezenta tema asistentului, care poate modifica punctajul dat de checker dacă observă nereguli precum răspunsuri hardcodate, proceduri cu efecte laterale, implementări neconforme cu restricțiile din enunț.

Resurse

- starter kit
- tutorial structuri
- etapa 2
- etapa 3
- etapa 4

Changelog

- 26.03 (ora 00:08) - Adăugat etapa 4 (enunț și arhivă).
- 21.03 (ora 19:57) - Update funcție de afișare în checker (comportamentul rămâne la fel).
- 20.03 (ora 09:02) - Adăugat etapa 3 (enunț și arhivă).
- 20.03 (ora 08:58) - Adăugat formula nota-temă = (n1+n2)/2 (n1/n2 = nota înainte/după deadline).
- 15.03 (ora 22:02) - Precizat că tema se încarcă sub forma unei arhive .zip cu fișierul supermarket.rkt.
- 13.03 (ora 23:12) - Adăugat etapa 2 (enunț și arhivă).
- 10.03 (ora 09:45) - Adăugat tutorialul și în secțiunea de resurse.
- 10.03 (ora 09:35) - Precizat deadline-urile pentru restanțieri.

Referinţe

- Simple and Efficient Purely Functional Queues and Deques [http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.8825&rep=rep1&type=pdf]