تمرینات سری چهارم

بخش اول: توابع فهرست بازگشتی

در این مساله شما سه تابع بازگشتی را در لیست ها تعریف خواهید کرد. برخی از قوانین پایه:

- همه توابع خود را در یک فایل جدید به نام تعریف کنید yourAccountName-ps4-solo-functions.rkt که شما در دکتر راکت می سازید.
- شما باید در تمام تعاریف خود از بازگشت صریح در لیست ها استفاده کنید.
- در تمام تعاریف خود باید از استراتژی تقسیم / فتح / چسب که در کلاسی که در PS3 آموزش داده اید، استفاده کنید.
- شما نباید از هیچ عملیات لیست مرتبه بالاتر در این مساله استفاده کنید. (به عنوان مثال، map, filter, foldr, foldl, iterate یا genlist)
- تنها عملگرهای لیست راکت توکار که می توانید در تعاریف خود از آن ها استفاده کنید عبارتند از: null, null?, cons, list, append, first, second, third, and rest. (همچنین می توانید از هر عملگر ریاضی یا منطقی راکت استفاده کنید، مانند: +, max و ...)
- برای جلوگیری از ارزیابی یک عبارت بیش از یک بار باید از ساختار "let" راکت استفاده کنید.
- در این مساله باید تابع کمکی prob1-map-cons زیر را تعریف و استفاده کنید:

```
(define (prob1-map-cons x yss)
  (if (null? yss)
      Null
      (cons (cons x (first yss)) (prob1-map-cons x (rest yss)))))
```

- (این نام خاص "prob1-map-cons" را دارد تا با تابع "map-cons" که در مساله ۳ تعریف خواهید کرد، در تضاد نباشد.)

- به غیر از "prob1-map-cons" نباید از هیچ تابع کمکی دیگری در این مساله استفاده کنید به جز یک تابع کمکی مشخص شده در 1d "weighted-suffixes".

الف) پیشوند length-n یک لیست، لیستی است که شامل n عنصر اول آن به ترتیب نسبی یکسان است. برای مثال:

- پیشوند length-0 '(5 8 4): ()'
- پیشوند length-0 '(5 8 4): (5)'
- پیشوند length-0 '(5 8 4): (5 8)'
- پیشوند length-0 '(5 8 4): (5 8 4)'

یک تابع "prefixes" تعریف کنید که یک لیست را به عنوان آرگومان تکی خود در نظر می گیرد و لیستی از تمام پیشوندهای مرتب شده از کوتاه ترین تا طولانی ترین را برمی گرداند. برای مثال:

```
> (prefixes '(5 8 4))
 '(() (5) (5 8) (5 8 4))
 > (prefixes '(2 5 8 4))
 '(() (2) (2 5) (2 5 8) (2 5 8 4))
 > (prefixes '(7 2 5 8 4))
 '(() (7) (7 2) (7 2 5) (7 2 5 8) (7 2 5 8 4))
 > (prefixes (range 0 11))
 '(()
   (0)
   (0 1)
   (0 1 2)
   (0 1 2 3)
   (0 1 2 3 4)
   (0 1 2 3 4 5)
   (0 1 2 3 4 5 6)
   (0 1 2 3 4 5 6 7)
   (0 1 2 3 4 5 6 7 8)
   (0 1 2 3 4 5 6 7 8 9)
   (0 1 2 3 4 5 6 7 8 9 10)
```

ب) تعریف یک تابع "sum-max-squaresEvens" که لیستی از اعداد صحیح را با آرگومان تکی خود می گیرد و یک لیست سه تایی (یعنی یک لیست سه عنصری) را برمی

گرداند که سه عنصر آن (۱) مجموع اعداد موجود در لیست است؛ (۲) حداکثر اعداد موجود در لیست و (۳) لیستی از مربعات تمام اعداد زوج در لیست (با حفظ ترتیب نسبی).

```
> (sum-max-squaresEvens '(9 2 8 5 4 7 1 6 3))
 '(45 9.0 (4 64 16 36))
 > (sum-max-squaresEvens '(2 8 5 4 7 1 6 3))
 '(36 8.0 (4 64 16 36))
 > (sum-max-squaresEvens '(8 5 4 7 1 6 3))
 '(34 8.0 (64 16 36))
 > (sum-max-squaresEvens '(5 4 7 1 6 3))
 '(26 7.0 (16 36))
 > (sum-max-squaresEvens '(-9 2 -8 5 4 -7 1 -6 3))
 '(-15 5.0 (4 64 16 36))
 > (sum-max-squaresEvens '(-6 -3 -10 -5 -8))
 '(-32 -3.0 (36 100 64))
 > (sum-max-squaresEvens (append (range 1 101 7) (range 201 0 -2)))
 '(10951 201.0 (64 484 1296 2500 4096 6084 8464))
```

تابع "sum-max-squaresEvens" شما باید یک پاس واحد از لیست ورودی ایجاد کند تا خروجی سه برابر شود. برای محاسبه هر یک از این سه بخش، نیازی به مراجعه جداگانه نیست.

ج) فرض کنید که ما یک مجموعه را به صورت یک لیست بدون تکرار نمایش می دهیم. یک تابع "subsets" تعریف کنید که به عنوان آرگومان تکی خود یک مجموعه را در نظر می گیرد و لیستی از تمام زیرمجموعه های یک مجموعه داده شده را برمی گرداند. زیرمجموعه های موجود در لیست نتایج می توانند به هر ترتیبی باشند، اما ترتیب عناصر در هر مجموعه باید همان ترتیب نسبی در "set" را داشته باشد.

برای مثال در اینجا تعدادی از پاسخ های ممکن برای "(subsets '(3 1 2))" وجود دارد که هر کدام از آن ها صحیح در نظر گرفته می شوند:

```
'(() (1) (2) (3) (1 2) (3 1) (3 2) (3 1 2))
'((3 1 2) (3 2) (3 1) (1 2) (3) (2) (1) ())
'(() (2) (1) (1 2) (3) (3 2) (3 1) (3 1 2))
'((3 1 2) () (3 1) (2) (3) (1 2) (1) (3 2))
```

با این حال، لیست های حاوی زیرمجموعه هایی مانند `(1 2 3)`, `(3 2 1)`, `(1 3)`, `(2 1)` نمی توانند راه حل باشند، زیرا عناصر این زیرمجموعه ها در یک ترتیب نسبی مشابه `(3 1 2)` نیستند.

د) پسوند length-n یک لیست، لیستی است که شامل آخرین n عنصر آن به ترتیب نسبی یکسان است. برای مثال:

- پسوند length-0 '(5 8 4)': ()'
- پسوند length-0 '(5 8 4)': (4)'
- پسوند length-0 '(5 8 4)': (8 4)'
- پسوند length-0 '(5 8 4)': (5 8 4)'

براساس این تعریف، یک پسوند تابع را تصور کنید که یک لیست را به عنوان آرگومان تکی خود در نظر می گیرد و لیستی از همه "suffixes" مرتب شده از طولانی ترین تا کوتاه ترین را برمی گرداند. برای مثال:

```
> (suffixes '(5 8 4))
 '((5 8 4) (8 4) (4) ())
 > (suffixes '(2 5 8 4))
 '((2 5 8 4) (5 8 4) (8 4) (4) ())
 > (suffixes '(7 2 5 8 4))
 '((7 2 5 8 4) (2 5 8 4) (5 8 4) (8 4) (4) ())
 > (suffixes (range 1 11))
 '((1 2 3 4 5 6 7 8 9 10)
   (2 3 4 5 6 7 8 9 10)
   (3 4 5 6 7 8 9 10)
   (4 5 6 7 8 9 10)
   (5 6 7 8 9 10)
   (6 7 8 9 10)
   (7 8 9 10)
   (8 9 10)
   (9 10)
   (10)
   ())
```

در این مساله، از شما خواسته نمی شود که "suffixes" را تعریف کنید، بلکه از شما خواسته می شود که یک تابع مرتبط به نام "weighted-suffixes" را تعریف کنید، که فرض می شود لیستی از اعداد را در بر می گیرد. نتیجه "weighted-suffixes" فهرستی مشابه فهرست "suffixes" است با این تفاوت که هر زیرلیست غیر وزن دار در نتیجه "weighted-suffixes" نتیجه مقیاس گذاری همه اعداد در زیرلیست غیر وزن دار متناظر در نتیجه

"suffixes" توسط عنصر اول آن است. (زیرلیست خالی در "suffixes" زیرلیست خالی را در "weighted-suffixes" به دست می دهد).

برای مثال: `(weighted-suffixes '(7 2 5 8 4))` برمیگرداند

`'((49 14 35 56 28) (4 10 16 8) (25 40 20) (64 32) (16) ())` چون:

- (49 14 35 56 28) نتیجه مقیاس گذاری (4 8 5 2 7) تا 7 است.
- (4 10 16 8) نتیجه مقیاس گذاری (4 8 5 2) تا 2 است.
- (25 40 20) نتیجه مقیاس گذاری (4 8 5) تا 5 است.
- (64 32) نتیجه مقیاس گذاری (4 8) تا 8 است.
- (16) نتیجه مقیاس گذاری (4) تا 4 است.
- () زیرلیست در نتیجه weighted-suffixes است که متناظر با زیرلیست () در نتیجه suffixes است.

در اینجا مثال های بیشتری از "weighted-suffixes" آورده شده است که دو مورد آخر اعداد منفی را نشان می دهند:

```
> (weighted-suffixes (range 3 8))
 '((9 12 15 18 21) (16 20 24 28) (25 30 35) (36 42) (49) ())

 > (weighted-suffixes (range 1 11))
 '((1 2 3 4 5 6 7 8 9 10)
   (4 6 8 10 12 14 16 18 20)
   (9 12 15 18 21 24 27 30)
   (16 20 24 28 32 36 40)
   (25 30 35 40 45 50)
   (36 42 48 54 60)
   (49 56 63 70)
   (64 72 80)
   (81 90)
   (100)
   ())

 > (weighted-suffixes '(-2 6 1 -3 -8 4 7 -5))
 '((4 -12 -2 6 16 -8 -14 10)
   (36 6 -18 -48 24 42 -30)
   (1 -3 -8 4 7 -5)
   (9 24 -12 -21 15)
   (64 -32 -56 40)
   (16 28 -20)
   (49 -35)
   (25)
```

```
    ())

 > (weighted-suffixes (range -3 4))
 '((9 6 3 0 -3 -6 -9) (4 2 0 -2 -4 -6) (1 0 -1 -2 -3) (0 0 0 0) (1 2
3) (4 6) (9) ())
```

به یاد داشته باشید که در اینجا نمی توانید از تابع "map" مرتبه بالاتر استفاده کنید. بنابراین، در این مساله، علاوه بر "weighted-suffixes"، شما باید یک تابع کمکی لیست بازگشتی تعریف کنید که الگوی نگاشت را برای مقیاس بندی همه عناصر در یک لیست از اعداد توسط یک فاکتور مقیاس دهی داده شده پیاده سازی می کند.

بخش دوم: فهرست های Wacky

این مساله نشان می دهد که توابع می توانند برای پیاده سازی ساختارهای داده مانند جفت ها و لیست ها استفاده شوند. جایگزین های زیر را برای کانورتیبل های معمولی cons, car, cdr, null, null? در نظر بگیرید:

```
    (define kons (λ (x y) (λ (s) (s #f x y))))

    (define kar (λ (k) (k (λ (b l r) l))))

    (define kdr (λ (k) (k (λ (b l r) r))))

    (define knil (λ (s) (s #t 0 0)))

    (define knil? (λ (k) (k (λ (b l r) b))))
```

الف) از مدل جانشینی گام کوچک (⇒ using) برای نشان دادن ارزیابی مقدار کراندار به نام p با بیان زیر استفاده کنید:

```
(define p (kons 3 4))
```

هر مرحله را نشان دهید؛ مراحل را از طریق "⇒*" خلاصه نکنید. در هر مرحله، به طور صریح (۱) علامت قرمز (در مهارهای مجذور) و (۲) نام قاعده اعمال شده (در براکت های مربعی) را نشان می دهد.

ب) از مدل جانشینی گام کوچک (⇒ using) برای نشان دادن بیضی بودن هر یک از عبارات زیر استفاده کنید. در این مراحل باید از محیط env استفاده کنید که حاوی عبارت

"p↦λ_p" باشد که در آن "λ_p" مقدار شما از بخش الف است . شما باید فرض کنید که env نیز شامل اتصال هایی برای knil , Kar , kdr , knil و knil? است.

- `(kar p)`
- `(kdr p)`
- `(knil? p)`
- `(knil? knil)`

هر مرحله را نشان دهید؛ مراحل را از طریق "⇒ ❋" خلاصه نکنید. در هر مرحله، به طور صریح (۱) علامت قرمز (در مهارهای مجذور) و (۲) نام قاعده اعمال شده (در براکت های مربعی) را نشان می دهد.

ج) تابع "sum-to" زیر از توابع کمکی "sum" و "down-from" استفاده می کند که بر حسب موجودیت های لیست مانند شامل "kons" و دوستان تعریف شده اند. آیا در واقع مجموع اعداد صحیح از ۱ تا n (به صورت فراگیر) را محاسبه می کند؟ چرا؟

```
(define (sum-to n)
   (sum (down-from n)))


 (define (sum nums)
   (if (knil? nums)
       0
       (+ (kar nums) (sum (kdr nums)))))


 (define (down-from n)
   (if (<= n 0)
       knil
       (kons n (down-from (- n 1)))))
```

د) آیا می توانیم همه موارد `cons` / `car` / `cdr` / `null` / `null?` در راکت را با `kons` / `kar` / `kdr` / `knil` / `knil?` جایگزنی کرد؟ آیا روشی وجود دارد که

kons/kar/kdr/knil/knil? مانند cons/car/cdr/null/null? رفتار نکند. در این زمینه چند نکته را باید در نظر گرفت:

- مقدار "(car null)?" و "(kar knil)'?" چقدر است؟
- آیا "cons" و دوستان می توانند با "kons" و دوستان هم کاری کنند؟

بخش سوم: توابع لیست مرتبه بالاتر

- برای مسئله های ۳ تا ۵، شما باید از دکتر راکت برای ایجاد یک فایل واحد به نام yourAccountName-ps4-functions.rkt استفاده کنید که شامل تمام توابع (از جمله توابع کمکی) است که برای این مشکلات تعریف می کنید.
- در تعاریف شما، مجاز به استفاده از بازگشت در هیچ جایی نیستید. (یک استثنا تابع کمکی "inserts-rec" است که در مسئله 3l داده شده است).
- در تعاریف خود، اگر غیر از این آموزش نداده باشید، نباید هیچ تابع کمکی جدیدی را معرفی کنید، اما می توانید (۱) آزادانه از توابع ناشناس استفاده کنید و (۲) از توابعی که در بخش های قبلی تعریف کرده اید در بخش های بعدی استفاده کنید.

الف) با استفاده از "map" راکت، یک تابع "map-remainder" را تعریف کنید که دو آرگومان (یک "dividor" عدد صحیح و لیست "ints" اعداد صحیح) را می گیرد و یک لیست عدد صحیح با طول مشابه "ints" را برمی گرداند که در آن هر عنصر باقی مانده تقسیم عنصر متناظر "ints" به "divisor" است.

```
> (map-remainder 2 '(16 23 42 57 64 100))
'(0 1 0 1 0 0)
> (map-remainder 3 '(16 23 42 57 64 100))
'(1 2 0 0 1 1)
> (map-remainder 5 '(16 23 42 57 64 100))
'(1 3 2 2 4 0)
> (map-remainder 17 '(16 23 42 57 64 100))
'(16 6 8 6 13 15)
```

ب) با استفاده از "filter" راکت، یک تابع "filter-divisible-by" را تعریف کنید که دو آرگومان (یک "dividor" عدد صحیح و لیست "ints" اعداد صحیح) را می گیرد و یک لیست عدد صحیح جدید شامل تمام عناصر"ints"را برمی گرداند که توسط "دیور" قابل تقسیم هستند.

```
> (filter-divisible-by 2 '(16 23 42 57 64 100))
 '(16 42 64 100)
 > (filter-divisible-by 3 '(16 23 42 57 64 100))
 '(42 57)
 > (filter-divisible-by 4 '(16 23 42 57 64 100))
 '(16 64 100)
 > (filter-divisible-by 5 '(16 23 42 57 64 100))
 '(100)
 > (filter-divisible-by 17 '(16 23 42 57 64 100))
 '()
```

از تابع کمکی زیر استفاده کنید که در این مشکل و برخی از موارد زیر مفید است.

```
(define divisible-by?
  (lambda (num divisor)
    (= (remainder num divisor) 0)))
```

ج) با استفاده از "foldr" راکت، یک تابع " contains-multiple?" تعریف کنید. که یک عدد صحیح m و یک لیست از اعداد صحیح ns می گیرد که "#t" را برمی گرداند اگر m به طور مساوی حداقل یک عنصر از لیست اعداد صحیح ns را تقسیم کند؛ در غیر این صورت "#f" را برمی گرداند. از " divisible-by?" استفاده کنید؟ از بالا برای تعیین تفکیک پذیری.

```
> (contains-multiple? 5 '(8 10 14))
 #t
 > (contains-multiple? 3 '(8 10 14))
 #f
 > (contains-multiple? 5 '())
 #f
```

د) با استفاده از "foldr" راکت، تابع " all-contain-multiple?" را تعریف کنید که یک عدد صحیح n و یک لیست از لیست های اعداد صحیح nss را می گیرد ("enziz" تلفظ می شود) و "#t" را برمی گرداند اگر هر لیست از اعداد صحیح nss شامل حداقل یک عدد صحیح باشد که چند عدد از n باشد؛ در غیر این صورت "#f" را برمی گرداند. استفاده از "contains-multiple?" در تعریف شما از "all-contain-multiple?"

```
 > (all-contain-multiple? 5 '((17 10 2) (25) (3 8 5)))
 #t
```

```
> (all-contain-multiple? 2 '((17 10 2) (25) (3 8 5)))
#f
> (all-contain-multiple? 3 '())
#t ; said to be "vacuously true"; there is no counterexample!
```

ه) با استفاده از تابع "foldr" راکت، یک تابع "snoc" تعریف کنید که مقدار x و یک لیست ys را می گیرد و لیست جدیدی که از اضافه کردن x به انتهای ys حاصل می شود را برمی گرداند.

```
> (snoc 4 '(7 2 5))
'(7 2 5 4)
> (snoc 4 '())
'(4)
```

و) با استفاده از تابع "foldr" راکت، تابعی به نام "my - append" تعریف کنید که دو لیست xs و ys را می گیرد و لیست جدیدی را برمی گرداند که شامل تمام عناصر xs و به دنبال آن تمام عناصر ys است.

```
> (my-append '(7 2 5) '(4 6))
'(7 2 5 4 6)
> (my-append '() '(4 6))
'(4 6)
> (my-append '(7 2 5) '())
'(7 2 5)
> (my-append '() '())
'()
```

نکته: ممکن است در تعریف خود از "append" استفاده نکنید.

ز) با استفاده از تابع "foldr" راکت، تابع "append - all" را تعریف کنید که لیستی از لیست های xss را می گیرد و یک لیست جدید را برمی گرداند که شامل تمام عناصر زیر لیست های xss به ترتیب نسبی آن ها است.

```
> (append-all '((1 2) (3) (4 5 6)))
'(1 2 3 4 5 6)
> (append-all '((1 2) (3)))
'(1 2 3)
> (append-all '((1 2)))
'(1 2)
> (append-all '())
'()
> (append-all '(((1 2) (3 4 5)) ((6)) ((7 8) () (9))))
'((1 2) (3 4 5) (6) (7 8) () (9))
```

نکته: می توانید در تعریف خود از "append" یا "my - append" استفاده کنید.

ح) با استفاده از "map" راکت، تابع "map-cons" را تعریف کنید که هر مقدار x و یک لیست n عنصری ys را می گیرد و یک لیست n عنصری از همه جفت های "(x . y)' " را برمی گرداند که در آن y بر روی عناصر ys قرار می گیرد. این دو "(x . y)' " باید همان موقعیت نسبی را در لیست حاصل داشته باشد که y در ys دارد.

```
> (map-cons 17 '(8 5 42 23))
'((17 . 8) (17 . 5) (17 . 42) (17 . 23))
> (map-cons 3 '((1 6 2) (4 5) () (9 6 8 7)))
'((3 1 6 2) (3 4 5) (3) (3 9 6 8 7))
> (map-cons 42 '())
'()
```

ط) با استفاده از تابع "foldr" راکت، تابعی به نام "my-cartesian-product" تعریف کنید که دو لیست xs و ys را می گیرد و لیستی از همه جفت های "(x . y)' " را برمی گرداند. که در آن x بر روی عناصر xs و y بر روی عناصر ys قرار می گیرد. جفت ها باید ابتدا توسط ورودی x (نسبت به ترتیب در xs)و سپس توسط ورودی y (نسبت به ترتیب در ys)مرتب شوند.

```
> (my-cartesian-product '(1 2) '("a" "b" "c"))
'((1 . "a") (1 . "b") (1 . "c") (2 . "a") (2 . "b") (2 . "c"))
> (my-cartesian-product '(2 1) '("a" "b" "c"))
'((2 . "a") (2 . "b") (2 . "c") (1 . "a") (1 . "b") (1 . "c"))
> (my-cartesian-product '("c" "b" "a") '(2 1))
'(("c" . 2) ("c" . 1) ("b" . 2) ("b" . 1) ("a" . 2) ("a" . 1))
> (my-cartesian-product '("a" "b") '(2 1))
'(("a" . 2) ("a" . 1) ("b" . 2) ("b" . 1))
> (my-cartesian-product '(1) '("a"))
'((1 . "a"))
> (my-cartesian-product '() '("a" "b" "c"))
'()
```

نکته: می توانید در تعریف خود از "map-cons" و "append" یا "my-append" استفاده کنید.

ی) با استفاده از تابع "foldr" راکت، تابع "my-reverse" را تعریف کنید که یک لیست xs می گیرد و یک لیست جدید را برمی گرداند که عناصر آن عناصر xs به ترتیب معکوس هستند. ممکن است از تابع "reverse" داخلی استفاده نکنید.

```
> (my-reverse '(1 2 3 4))
```

```
'(4 3 2 1)
> (my-reverse '(1))
'(1)
> (my-reverse '())
'()
```

نکته:

- از شما می خواهیم که تابع خود را "my-reverse" نامگذاری کنید، زیرا راکت هم اکنون همان تابع را با نام "reverse" ارائه می دهد (که البته نمی توانید از آن استفاده کنید).
- میتوانید در تعریف خود از "cons" و "append" یا "my-append" استفاده کنید.

ک) فرض کنید که عناصر یک لیست با شروع از 0 اندیس گذاری می شوند. با استفاده از با استفاده از تابع "foldr" راکت، تابعی به نام "alts" تعریف کنید که یک لیست xs را می گیرد و یک لیست دو عنصری از لیست ها را برمی گرداند که اولی تمام عناصر نمایه شده زوج (با همان ترتیب نسبی xs) و دومی تمام عناصر نمایه شده فرد (با همان ترتیب نسبی xs) را دارد.

```
> (alts '(7 5 4 6 9 2 8 3))
'((7 4 9 8) (5 6 2 3))
> (alts '(5 4 6 9 2 8 3))
'((5 6 2 3) (4 9 8))
> (alts '(4 6 9 2 8 3))
'((4 9 8) (6 2 3))
> (alts '(3))
'((3) ())
> (alts '())
'(() ())
```

نکته: نیازی به توجه خاص به طول زوج و طول فرد به طور متفاوت وجود ندارد و نیازی به توجه خاص به لیست singleton نیز نیست.

ل) با استفاده از با استفاده از تابع "foldr" راکت، تابعی را تعریف کنید که یک مقدار x و یک لیست n عنصری ys را می گیرد و یک لیست n+1-element از لیست ها را برمی گرداند که تمام راه های درج یک کپی از x در ys را نشان می دهد.

```
> (inserts-foldr 3 '(5 7 1))
 '((3 5 7 1) (5 3 7 1) (5 7 3 1) (5 7 1 3))
 > (inserts-foldr 3 '(7 1))
 '((3 7 1) (7 3 1) ( 7 1 3))
 > (inserts-foldr 3 '(1))
 '((3 1) (1 3))
 > (inserts-foldr 3 '())
 '((3))
 > (inserts-foldr 3 '(5 3 1))
 '((3 5 3 1) (5 3 3 1) (5 3 3 1) (5 3 1 3))
```

نکته:

- تابع "map-cons" از بالا در اینجا بکار می رود.
- در مورد حالت پایه و تابع ترکیبی برای حالت بازگشتی بسیار دقیق فکر کنید.
- تعریف شما باید دقیقا این الگو را داشته باشد:

```
(define (inserts-foldr x ys)
  (foldr ; binary combiner goes here
         ; null value goes here
         ys))
```

م) با استفاده از با استفاده از تابع "foldr" راکت، تابع "my-permutations" را تعریف کنید که به عنوان آرگومان تکی خود یک لیست xs از عناصر متمایز (یعنی بدون تکرار) می گیرد و لیستی از تمام جایگشت های عناصر xs را برمی گرداند. ترتیب جایگشت ها مهم نیست.

```
> (my-permutations '())
'(())
> (my-permutations '(4))
'((4))
> (my-permutations '(3 4))
'((3 4) (4 3)) ; order doesn't matter
> (my-permutations '(2 3 4))
'((2 3 4) (3 2 4) (3 4 2) (2 4 3) (4 2 3) (4 3 2))
> (my-permutations '(1 2 3 4))
'((1 2 3 4) (2 1 3 4) (2 3 1 4) (2 3 4 1)
  (1 3 2 4) (3 1 2 4) (3 2 1 4) (3 2 4 1)
  (1 3 4 2) (3 1 4 2) (3 4 1 2) (3 4 2 1)
  (1 2 4 3) (2 1 4 3) (2 4 1 3) (2 4 3 1)
  (1 4 2 3) (4 1 2 3) (4 2 1 3) (4 2 3 1)
  (1 4 3 2) (4 1 3 2) (4 3 1 2) (4 3 2 1))
```

توجه: استفاده از "append-all"، "map" و "inserts-foldr" در راه حل شما مفید است.

اگر از زیرمسئله قبلی قادر به تعریف "inserts-foldr" نیستید، می توانید از نسخه بازگشتی زیر تابع "inserts" استفاده کنید:

```
(define (inserts-rec x ys)
  (if (null? ys)
      (list (list x))
      (cons (cons x ys)
            (map-cons (first ys)
                      (inserts-rec x (rest ys))))))
```

بخش چهارم: forall?, exists?, find and zip

در زیر برخی از توابع لیست – پردازش که در راکت ساخته نشده اند، اما در بسیاری از موقعیت ها مفید هستند، آورده شده است:

```
(define (forall? pred xs)
  (or (null? xs)
      (and (pred (first xs))
           (forall? pred (rest xs)))))

(define (exists? pred xs)
  (and (not (null? xs))
       (or (pred (first xs))
           (exists? pred (rest xs)))))

(define (find pred not-found xs)
  (if (null? xs)
      not-found
      (if (pred (first xs))
          (first xs)
          (find pred not-found (rest xs)))))

(define (zip xs ys)
  (if (or (null? xs) (null? ys))
      '()
      (cons (cons (first xs) (first ys))
            (zip (rest xs) (rest ys)))))

;; Can also define zip using Racket's mutlilist map
;; (but need to make both list args have same length;
;; otherwise Racket map will generate error if theyr don't)
;;
;; (define (zip xs ys)
;;   (let ((minlen (min (length xs) (length ys))))
;;     (map cons (take xs minlen) (take ys minlen))))
```

"forall?" ، "exists?" و "find" توابع لیست مرتبه بالاتر شامل یک پیش بینی هستند.

- "forall?" اگر پیش بینی در همه عناصر لیست درست باشد "#t" را برمی گرداند، و در غیر این صورت "#f" را برمی گرداند.

```
> (forall? (λ (x) (> x 0)) '(7 2 5 4 6))
#t
> (forall? (λ (x) (> x 0)) '(7 2 -5 4 6))
#f
```

- "esixts?" "#t" را برمی گرداند اگر پیش بینی حداقل روی یکی از عناصر لیست درست باشد، و در غیر این صورت "#f" را برمی گرداند.

```
> (exists? (λ (x) (< x 0)) '(7 2 -5 4 6))
#t
> (exists? (λ (x) (< x 0)) '(7 2 5 4 6))
#f
```

- "find" اولین عنصر لیست را برمی گرداند که پیش بینی برای آن درست است. اگر چنین عنصری وجود نداشته باشد، مقدار عرضه شده به عنوان آرگومان "not-found" را برمی گرداند.

```
> (find (λ (x) (< x 0)) #f '(7 2 -5 4 -6))
-5
> (find (λ (x) (< x 0)) #f '(7 2 5 4 6))
#f
```

تابع "zip" مرتبه بالاتری ندارد، اما دو لیست را با جفت کردن (با استفاده از "cons") عناصر متناظر دو لیست ترکیب می کند. اگر لیست ها طول یکسانی نداشته باشند، "zip" لیستی از جفت هایی را برمی گرداند که طول آن ها طول کوتاه تر دو لیست ورودی است:

```
> (zip '(1 2 3) '("a" "b" "c"))
'((1 . "a") (2 . "b") (3 . "c"))
> (zip '(1 2 3 4 5) '("a" "b" "c"))
'((1 . "a") (2 . "b") (3 . "c"))
> (zip '(1 2 3) '("a" "b" "c" "d" "e"))
'((1 . "a") (2 . "b") (3 . "c"))
```

در این مسئله، شما از توابع "forall?"، "exists?"، "find" و "zip" برای تعریف توابع دیگر. این مشکل را با کپی کردن تعاریف این چهار تابع در بالای فایل yourAccountName-ps4-functions.rkt شروع کنید.

الف) استفاده از "exists?" ، یک تابع "عضو"member?" که تعیین می کند آیا عنصر x در لیست ys ظاهر می شود یا خیر.

```
> (member? 4 '(7 2 5 4 6))
#t
> (member? 3 '(7 2 5 4 6))
#f
> (member? '(7 8) '((1 2) (3 4 5) (6) (7 8) () (9)))
#t
> (member? '() '((1 2) (3 4 5) (6) (7 8) () (9)))
#t
> (member? '(5 6) '((1 2) (3 4 5) (6) (7 8) () (9)))
#f
```

نکته: از "equal?" استفاده کنید برای مقایسه برابری دو مقدار.

ب) استفاده از "forall?" و "exists?" ، یک تابع تعریف کنید "all-contain-multiple-alt?" که یک پیاده سازی جایگزین از "all-contain-multiple?" تابعی از مساله ۳.

```
> (all-contain-multiple-alt? 5 '((17 10 2) (25) (3 8 5)))
#t
> (all-contain-multiple-alt? 2 '((17 10 2) (25) (3 8 5)))
#f
> (all-contain-multiple-alt? 3 '())
#t ; said to be "vacuously true"; there is no counterexample!
```

نکته: ممکن است از تابع "divisible_by" از بالا استفاده کنید، اما نه "contains-multiple?"، و شما نمی توانید هیچ تابع کمکی جدیدی را تعریف کنید.

ج) فهرست پیوندی فهرستی از جفت ها است که یک نگاشت از کلید به مقدار را نشان می دهد. هر جفت کلید و مقدار با یک سلول مخروطی، با کلید در "car" و مقدار در "cdr" نشان داده می شود. به عنوان مثال، فهرست پیوندی:

```
'((2 . 3) (5 . 1) ("mountain" . #t))
```

کلید 2 را به مقدار 3، کلید 5 را به مقدار 1 و کلید "mountain" را به مقدار "#t" رسم کنید.

با استفاده از "find"، تابع "lookup" را تعریف کنید که یک کلید "k" و "as" فهرست پیوندی را به صورت زیر می گیرد و برمی گرداند:

- "#f" اگر هیچ نگاشتی با کلید "k" در لیست یافت نشود؛ و
- یک سلول cons که "car" آن "k" است و "cdr" آن مقدار متناظر برای کوچک ترین نگاشت "k" در لیست پیوندی است.

برای نمونه:

```
> (lookup 5 '((2 . 3) (5 . 1) ("mountain" . #t)))
 '(5 . 1)
 > (lookup 1 '((2 . 3) (5 . 1) ("mountain" . #t)))
 #f
 > (lookup '(6 4) '((2 . 3) (5 . 1) ((6 4) . 8) (5 . 1) (17 23 42)))
 '((6 4) . 8)
 > (lookup 17 '((2 . 3) (5 . 1) ((6 4) . 8) (5 . 1) (17 23 42)))
 '(17 23 42) ; '(17 23 42) has a car of 17 and a cdr of '(23 42)
 > (lookup 23 '((2 . 3) (5 . 1) ((6 4) . 8) (5 . 1) (17 23 42)))
 #f
```

نکته: از "equal?" استفاده کنید تا برابری کلیدها را آزمایش کند. این کار از کلیدهایی جالب تر از مقادیر ساده پشتیبانی خواهد کرد.

د) استفاده از "forall?" و "zip"، یک تابع "sorted?" تعریف کنید؟ که تعیین می کند آیا فهرستی از اعداد "ns" به ترتیب از پایین به بالا مرتب شده است یا خیر.

```
> (sorted? '(7 4 2 5 4 6))
 #f
 > (sorted? '(2 3 3 5 6 7))
 #t
 > (sorted? '(2))
 #t
 > (sorted? '())
 #t
 > (sorted? (range 1000))
 #t
 > (sorted? (append (range 1000) '(1001 1000)))
 #f
 > (sorted? (range 1000 0 -1))
 #f
```

نکته: برای لیست خالی باید مورد خاصی داشته باشید.

ه) می توان نسخه های جایگزین "forall?" و "exists?" را از نظر "foldr" همان طور که در شکل زیر مشاهده می کنید.

```
(define (forall-alt? pred xs)
   (foldr (λ (x subres) (and (pred x) subres))
          #t
          xs))

 (define (exists-alt? pred xs)
   (foldr (λ (x subres) (or (pred x) subres))
          #f
          xs))

 > (forall-alt? (λ (x) (> x 0)) '(7 2 5 4 6))
 #t
 > (forall-alt? (λ (x) (> x 0)) '(7 2 -5 4 6))
 #f
 > (exists-alt? (λ (x) (< x 0)) '(7 2 -5 4 6))
 #t
 > (exists-alt? (λ (x) (< x 0)) '(7 2 5 4 6))
```

با این حال، تنها به این دلیل که می توان یک تابع را بر حسب "foldr" تعریف کرد، به معنای ایده خوب آن نیست. مثالی عینی از وضعیتی که در آن "forall?" بهتر از "forall-alt" است؟

توجه: برای این مساله، مهم است که بدانیم "(and e1 e2)" به "(if e1 e2 #f)"

بخش پنجم: foldr-ternop

گاهی اوقات بیان تجمع لیست بازگشتی بر حسب "foldr" دشوار است زیرا تابع ترکیب کننده دودویی به اطلاعات بیشتری از لیست نسبت به عنصر اول آن نیاز دارد. تابع لیست مرتبه بالاتر "foldr-ternop" زیر این مشکل را با داشتن تابع ترکیب کننده به صورت یک تابع سه تایی (یعنی سه آرگومان) حل می کند که علاوه بر نتیجه پردازش بازگشتی لیست، هم اولین و هم بقیه لیست داده شده را می گیرد:

```
(define (foldr-ternop ternop null-value xs)
  (if (null? xs)
      null-value
      (ternop (first xs)
              (rest xs)
              (foldr-ternop ternop null-value (rest xs)))))
```

در این مساله، شما از "foldr-ternop" برای پیاده سازی دو تابع لیست استفاده می کنید که پیاده سازی آن ها بر حسب "foldr" چالش برانگیز. این مشکل را با کپی کردن تعریف "foldr-terntop" در بالای فایل "yourAccountName-ps4-functions.rkt" آغاز کنید.

الف) با استفاده از "foldr-ternop" تابع "inserts-foldr-ternop" را تعریف کنید که یک مقدار x و یک لیست n عنصری ys را می گیرد و یک لیست n+1 عنصری از لیست ها را برمی گرداند که تمام راه های درج یک کپی از x در ys را نشان می دهد.

```
> (inserts-foldr-ternop 3 '(5 7 1))
'((3 5 7 1) (5 3 7 1) (5 7 3 1) (5 7 1 3))
> (inserts-foldr-ternop 3 '(7 1))
'((3 7 1) (7 3 1) ( 7 1 3))
> (inserts-foldr-ternop 3 '( 1))
'((3 1) (1 3))
> (inserts-foldr-ternop 3 '())
'((3))
> (inserts-foldr-ternop 3 '(5 3 1))
'((3 5 3 1) (5 3 3 1) (5 3 3 1) (5 3 1 3))
```

نکته:

- تعریف شما باید دقیقا این الگو را داشته باشد:

```
(define (inserts-foldr-ternop x ys)
  (foldr-ternop {ternary-combiner} {null-value} ys))
```

- شما می توانید از "map-cons" در تابع ترکیب کننده سه تایی خود استفاده کنید.

ج) با استفاده از "foldr-ternop" تابعی به نام "sorted-alt?" تعریف کنید که پیاده سازی جایگزین "sorted?" تابعی از مسئله ٤ است.

```
> (sorted-alt? '(7 4 2 5 4 6))
#f
> (sorted-alt? '(2 3 3 5 6 7))
#t
> (sorted-alt? '(2))
#t
> (sorted-alt? '())
#t
> (sorted-alt? (range 1000))
#t
> (sorted-alt? (append (range 1000) '(1001 1000)))
#f
> (sorted-alt? (range 1000 0 -1))
```

```
#f
```

نکته:

- تعریف شما باید دقیقا این الگو را داشته باشد:

```
(define (sorted-alt? xs)
   (foldr-ternop ; ternary-combiner goes here
                 ; null-value goes here
                 xs))
```