تمرینات سری سوم

۱.قوانین کاهشی Small-Step در عبارات شرطی (if) بصورت زیر هستند :

- `(if V_test E_then E_else)` ⇒ `E_then` , if `V_test` is a value that is not `#f` [if nonfalse]
- `(if #f E_then E_else)` ⇒ `E_else` [if false]

لوییس فکر می کند که باید قوانین بصورت زیر تغییر پیدا کنند :

- `(if V_test V_then V_else)` ⇒ `V_then` , if `V_test` is a value that is not `#f` [if nonfalse Lois]
- `(if #f V_then V_else)` ⇒ `V_else` [if false Lois]

توضیح دهید که چرا روش پیشنهادی لوییس ، روش جایگزین خوبی نیست ؟

```
(define sum-between
  (lambda (lo hi)
    (if (> lo hi)
        0
        (+ lo (sum-between (+ lo 1) hi)))))
```

۲.این تمرین دارای توابع بازگشتی رکت بصورت زیر است :

```
(define g
  (lambda (n)
    (if (<= n 2)
        n
        (+ n
           (g (thirdish n))
           (g (half n))))))
```

thirdish و half بصورت زیر تعریف شده اند :

```
(define thirdish
  (λ (int) (* (remainder int 3) (quotient int 3))))

(define half
  (λ (int) (quotient int 2)))
```

half خارج قسمت هر عدد را بر ۲ بصورت صحیح (integer) برمیگرداند(۵ = ۱۰/۲ و ۵ = ۱۱/۲) و thirdish حاصل ضرب صحیح خارج قسمت و باقی مانده بر ۳ را برمیگرداند.

```
› (map (λ (n) (cons n (thirdish n))) (range 12))
'((0 . 0) (1 . 0) (2 . 0) (3 . 0) (4 . 1) (5 . 2) (6 . 0) (7 . 2) (8 . 4) (9 . 0) (10 . 3) (11 . 6))
```

الف)برای ارزیابی g(11) از روش Small-Step استفاده کنید.

ب)تابع g چند بار در روند ارزیابی g(11) فراخوانی شده است ؟

ج) حداکثر عمق پشته (اندازه گیری بر حسب حداکثر تعداد تو در تو + عملیات) در ارزیابی (g 11) چقدر است؟

د) یک تابع رکت بازگشتی به نام num-g-calls تعریف کنید که یک آرگومان صحیح n را می گیرد و تعداد دفعاتی را که تابع g در ارزیابی g(n) فراخوانی می شود، برمی گرداند.

* (num-g-calls n) باید تماس (g (n را به عنوان یکی از تماس ها حساب کند.

* تابع num-g-calls را در فایل جدیدی به نام yourAccountName-ps3-solo-functions.rkt که در Dr.Racket ایجاد می کنید، تعریف کنید. این فایل همچنین باید شامل تعاریف half و thirdish از بالا باشد.

* سعی نکنید تعریف g را طوری تغییر دهید که تعداد دفعاتی که g با تغییر محتوای یک متغیر سراسری فراخوانی می شود را بشمارد.

* یک تابع بازگشتی جدید num-g-calls بنویسید که بسیار شبیه g است، اما به جای برگرداندن عدد محاسبه‌شده توسط g ، تعداد فراخوانی‌های انجام‌شده با g را در آن محاسبه و برمی‌گرداند. تعریف شما باید از توابع کمکی half و thirdish استفاده کند.

*(num-g-calls 11) باید پاسخ شما را از قسمت ب برگرداند.

* نتیجه فراخوانی‌های num-g را در ورودی‌های دیگر بررسی کنید تا مطمئن شوید که صحیح هستند.

* تابع خود را با استفاده از این عبارت تست کنید و نتیجه آن را در نوشته های خود بگنجانید:

```
(map (λ (n) (cons n (num-g-calls n))) (range 100))
```

ه) یک تابع رکت بازگشتی به نام max-depth-g تعریف کنید که یک آرگومان صحیح n را می گیرد و حداکثر عمق پشته (که بر حسب حداکثر تعداد عملیات تودرتو اندازه گیری می شود) را در ارزیابی g(n) برمی گرداند. نکته:

*تابع max-depth-g را به فایل yourAccountName-ps3-solo-functions.rkt اضافه کنید.

*(max-depth-g 11) باید پاسخ شما را از قسمت ج برگرداند.

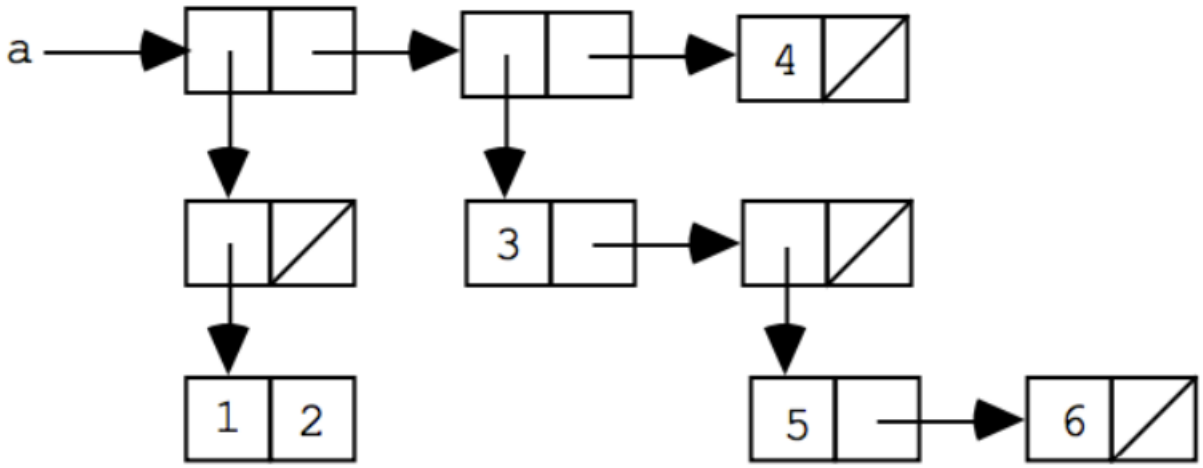
*نتیجه max-depth-gs را در سایر ورودی‌ها بررسی کنید تا مطمئن شوید که معقول هستند.

❊ تابع خود را با استفاده از این عبارت تست کنید و نتیجه آن را در نوشته خود بگنجانید:

```
(map (λ (n) (cons n (max-depth-g n))) (range 100))
```

۳.دیاگرام های Box-and-Pointer :

دیاگرام زیر را بعنوان ساختار آرایه a در نظر بگیرید :



الف) برای هر یک از اعداد ۱ تا ٦، عبارت رکت بنویسید که از car و cdr برای استخراج آن عدد از a استفاده می کند.

ب) نمایش چاپ شده a را بنویسید (یعنی چه چیزی توسط مترجم رکت برای ارزیابی a برگردانده می شود؟).

ج) یک تعریف رکت از فرم بنویسید (تعریف a expr)، که در آن expr عبارتی است با استفاده از cons ، list، و اعداد ۱ تا ٦ برای ایجاد ساختار نشان داده شده در نمودار تعریف می شود.

٤.توابع بازگشتی رکت لیست (بخش اول) :

برای هر یک از مشخصات تابع رکت زیر، یک تابع بازگشتی را بنویسید و آزمایش کنید که آن مشخصات را برآورده کند. در تمام تعاریف خود، باید از استراتژی حل مسئله بازگشتی زیر استفاده کنید:

برای کدام آرگومان(های) تابع آنقدر ساده است که بتوان بلافاصله پاسخ را برگرداند؟ این مورد پایه است.

برای مورد(های) دیگر (معروف به حالت(های) کلی یا حالت(های) بازگشتی)، از divide/conquer/glue استفاده کنید:

Divide : ایجاد یک یا چند زیرمسئله که نمونه های کوچکتری از مسئله داده شده است.

Conquer : فرض کنید تابع بازگشتی که تعریف می کنید به سادگی کار می کند و پاسخ صحیح را برای همه مسائل کوچکتر برمی گرداند.

Glue : نتیجه(های) فراخوانی(های) تابع بازگشتی را با اطلاعات موجود در مسئله اصلی ترکیب کنید تا نتیجه صحیح را برای کل مشکل ایجاد کنید.

الف) یک تابع map-remainder را تعریف کنید که دو آرگومان (یک مقسوم علیه عدد صحیح و یک لیست int اعداد صحیح) را می گیرد و یک لیست عدد صحیح به همان طول int برمی گرداند که در آن هر عنصر باقیمانده تقسیم عنصر متناظر intها بر مقسوم علیه است.

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

ب) یک تابع filter-divisible-by را تعریف کنید که دو آرگومان (یک مقسوم علیه عدد صحیح و یک لیست int اعداد صحیح) می گیرد و یک لیست عدد صحیح جدید حاوی تمام عناصر int که بر مقسوم علیه قابل تقسیم هستند را برمی گرداند.

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

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

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

ج) یک تابع چندگانه را تعریف کنید. که یک عدد صحیح m و یک لیست از اعداد صحیح ns را می گیرد که اگر m حداقل یک عنصر از لیست عدد صحیح ns را به طور مساوی تقسیم کند، #t را برمی گرداند. در غیر این صورت #f را برمی گرداند.

```
> (contains-multiple? 5 (list 8 10 14))
#t
> (contains-multiple? 3 (list 8 10 14))
#f
> (contains-multiple? 5 null)
#f
```

د) یک تابع all-contain-multiple بنویسید. که یک عدد صحیح n و لیستی از لیست های اعداد صحیح nss (تلفظ "enziz") را می گیرد و اگر هر لیست از اعداد صحیح در nss حداقل یک عدد صحیح مضرب n باشد، #t را برمی گرداند. در غیر این صورت #f را برمی گرداند.

```
> (all-contain-multiple? 5 (list (list 17 10 2) (list 25) (list 3 8 5)))
#t
> (all-contain-multiple? 2 (list (list 17 10 2) (list 25) (list 3 8 5)))
#f
> (all-contain-multiple? 3 null)
#t ; said to be "vacuously true"; there is no counterexample!
```

ه) یک 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 در محدوده عناصر ys قرار دارد. جفت '(x . y) باید همان موقعیت نسبی را در لیست حاصل داشته باشد که y در ys دارد.

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

۵. توابع بازگشتی رکت لیست (بخش دوم) :

برای هر یک از این توابع باید مراحل زیر را از استراتژی حل مسئله divide/conquer/glue (DCG)به صراحت نشان دهید:

برای مثال لیست ورودی L مشخص شده در هر مسأله:

۱.نمایش نتیجه فراخوانی تابع درL ؛

۲.نمایش نتیجه فراخوانی تابع در (rest L) ؛

۳.عبارتی بنویسید که مقدار (first L)را با نتیجه (۲) ترکیب کند تا نتیجه(۱) به دست آید.

٤.عبارت موجود در مورد ۳ را به یک عبارت برای حالت کلی تعریف تابع بازگشتی تعمیم دهید.

توضیح دهید که وقتی در لیست خالی فراخوانی می شود، تابع بازگشتی باید چه چیزی را برگرداند. اگر نمی دانید، حالت فراخوانی

تابع را در یک لیست تکی در نظر بگیرید و در حالت کلی نتیجه لیست خالی باید چه باشد. یک عبارت کلی برای حالت پایه بیان کنید.

نتایج بخش های ۱ و ۲ (تمرین توابع بازگشتی رکت لیست) را با هم ترکیب کنید تا تعریف تابع بازگشتی نهایی خود را ایجاد کنید.

الف) یک تابع my-cartesian-product را تعریف کنید که دو لیست xs و ys را می گیرد و فهرستی از همه جفت ها را برمی گرداند. جفت ها باید ابتدا با ورودی x (به ترتیب در xs) و سپس با ورودی y (به ترتیب در ys) مرتب شوند.

```
> (my-cartesian-product (list 1 2) (list "a" "b" "c")) ; yes, Racket has string values
'((1 . "a") (1 . "b") (1 . "c") (2 . "a") (2 . "b") (2 . "c"))
> (my-cartesian-product (list 2 1) (list "a" "b" "c"))
'((2 . "a") (2 . "b") (2 . "c") (1 . "a") (1 . "b") (1 . "c"))
> (my-cartesian-product (list "c" "b" "a") (list 2 1))
'(("c" . 2) ("c" . 1) ("b" . 2) ("b" . 1) ("a" . 2) ("a" . 1))
> (my-cartesian-product (list "a" "b") (list 2 1))
'(("a" . 2) ("a" . 1) ("b" . 2) ("b" . 1))
> (my-cartesian-product (list 1) (list "a"))
'((1 . "a"))
> (my-cartesian-product null (list "a" "b" "c"))
'()
```

ب) فرض کنید که عناصر یک لیست با شروع ۰ ایندکس می شوند. تابع alts را تعریف کنید که یک لیست xs را می گیرد و یک لیست دو عنصری از لیست ها را برمی گرداند، که اولی دارای تمام عناصر زوج اندیس شده است (به همان ترتیب نسبی مانند در xs) و دومی دارای تمام عناصر با نمایه فرد (به همان ترتیب نسبی در xs) است.

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

ج) تابعی را تعریف کنید که یک مقدار x و یک لیست n عنصر ys را می گیرد و یک لیست n+1 عنصری از لیست ها را برمی گرداند که تمام راه های درج یک نسخه از x را در ys نشان می دهد.

```
> (inserts 3 (list 5 7 1))
'((3 5 7 1) (5 3 7 1) (5 7 3 1) (5 7 1 3))
> (inserts 3 (list 7 1))
'((3 7 1) (7 3 1) ( 7 1 3))
> (inserts 3 (list 1))
'((3 1) (1 3))
> (inserts 3 null)
'((3))
> (inserts 3 (list 5 3 1))
'((3 5 3 1) (5 3 3 1) (5 3 3 1) (5 3 1 3))
```

د) یک تابع my-permutations را تعریف کنید که به عنوان آرگومان واحد خود یک لیست xs از عناصر متمایز (یعنی بدون تکرار) را در نظر می گیرد و لیستی از همه جایگشت های عناصر xs را برمی گرداند. ترتیب جایگشت ها مهم نیست.

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