Programación 1 - Práctica 6

1 Números Naturales

1.1 Introducción

DrRacket provee muchas funciones que consumen listas y algunas pocas que producen listas. Entre estas últimas se encuentra la función `make-list` que toma un valor `n` y otro `v` para devolver una lista de tamaño `n` que contiene a `n` como único valor.

Aquí podemos ver algunos ejemplos:

```
> (make-list 2 "hello")
(cons "hello" (cons "hello" '()))
> (make-list 3 #true)
(cons #true (cons #true (cons #true '())))
> (make-list 0 17)
'()
```

En resumen, aunque esta función consume sólo datos atómicos, permite construir datos arbitrariamente grandes. ¿Cómo es esto posible?

La respuesta a esa pregunta es que el primer argumento de `make-list` no es *cualquier* número sino que es un tipo especial de número. En ciencias de la computación los llamamos 'números naturales', son aquellos que nos permiten contar.

Los números naturales tienen su propia definición de tipo:

```
; Un Natural es:
; - 0
; - (add1 Natural)
; interpretación: Natural representa los números naturales
```

Como puede ver, esta definición es una definición autoreferenciada similar a la que usamos cuando definimos listas.

Miremos de cerca la definición de números naturales:

La primera cláusula nos dice que `0` es un número natural; en este caso indica que no hay objetos que puedan ser contados. La segunda cláusula dice que si `n` es un número natural, entonces `(add1 n)` también lo es. Ya que `add1` es una función que suma `1` a cualquier número, podríamos reescribir esta segunda cláusula como `(+ n 1)`, pero usamos `add1` para indicar que esta suma es especial.

Lo que tiene de especial el uso de `add1` es que acota como un constructor. Por esta razón, *DrRacket* provee también la función `sub1`, que es el "selector" correspondiente a `add1`. Dado un número natural `n` distinto de `0`, podemos usar `sub1` para averiguar el número que se usó para la construcción de `n`. En otras palabras `sub1` devuelve el predecesor de un número natural positivo

Los predicados que se usan para identificar el `0` y el resto de los números naturales son `zero?` y `positive?`, respectivamente.

A partir de ahora, podemos construir funciones que tomen como argumentos números naturales. Definamos por nuestra cuenta la función `copiar` que se comporta como `make-list`:

```
; (copiar Natural String -> Listof String
; El propósito de la función copiar es crear una lista de
; n copias de una cadena s
```

Notar que la aplicación sucesiva de `add1` sobre `0` nos permite contruir todos los naturales: `(add1 0)`=1, `(add1 (add1 0))`=2, etc. Es por eso que para simplificar la escritura vamos a utilizar los valores `1, 2, 3, 4, ...` para representar los números naturales

```
(check-expect (copiar 2 "hola") (list "hola" "hola"))
(check-expect (copiar 0 "hola") '())
(check-expect (copiar 4 "abc") (list "abc" "abc" "abc" "abc"))
```

Una vez que tenemos clara la signatura, el propósito de la función y algunos ejemplos concretos que muestran el comportamiento deseado, debemos definir la función. La función `copiar` deberá analizar si el número natural que se pasa como entrada es el `0` o si es un número positivo.

Así, el cuerpo de la función tendrá una expresión *cond* con dos cláusulas:

```
(define (copiar n s) (cond [(zero? n) '()]
                           [(positive? n) (cons s (copiar (sub1 n) s))]))
```

Si la entrada es `0` eso significa que se debe producir una lista con `0` elementos. Por otro lado, según la declaración de propósito de `copiar`, lo que produce `(copiar (sub1 n) s)` es un lista con `n-1` copias de `s`, por lo tanto, para calcular `(copiar n s)` faltaría únicamente agregar una copia de `s` a dicho resultado. Esto se logra haciendo `(cons s (copiar (sub1 n) s))`.

Resumimos entonces los **constructores**, **selectores** y **predicados** para los números naturales:

Operador	Tipo de Operador	Función
`0`	Constructor	Constante usada para representar el primer número natural
`add1`	Constructor	Calcula el sucesor de un número natural
`sub1`	Selector	Devuelve el predecesor de un número natural positivo
`zero?`	Predicado	Reconoce al natural 0
`positive?`	Predicado	Reconoce naturales construidos con add1

Ahora que tenemos en claro la definición de números naturales, vamos a practicar!

1.2 Ejercitación

Ejercicio 1. Diseñe la función `sumanat` que toma dos números naturales y sin usar `+` devuelve un natural que es la suma de ambos. Use el evaluador paso a paso para evaluar `(sumanat 3 2)`.

Ejercicio 2. Diseñe la función `multnat`. Esta función debe tomar como entrada dos números naturales y debe multiplicarlos sin usar `*` ni `+`. Use el evaluador paso a paso para evaluar `(multnat 3 2)`. Puede utilizar el ejercicio anterior.

Ejercicio 3. Diseñe la función `powernat` que toma dos números naturales y devuelve el resultado de elevar el primero a la potencia del segundo, usando la función `multnat` definida en el ejercicio anterior. Use el evaluador paso a paso para evaluar `(powernat 4 2)`.

Ejercicio 4. Diseñe la función `factnat` que toma un número natural y devuelve su factorial. El factorial de un número natural `n`, denotado `n!`, se define matemáticamente de la siguiente forma:

```
factnat (1) = 1

factnat (n+1) = (n+1) * factnat (n)
```

Use el evaluador paso a paso para evaluar `(factnat 4)`.

Atención: El factorial de un número crece muy rápidamente y *DrRacket* agotará rápidamente su límite de memoria. Le aconsejamos que al probar la función, aumente el valor de `n` gradualmente. Este mismo comentario aplica para muchas de las funciones que siguen.

Ejercicio 5. La sucesión de Fibonacci es una sucesión infinita de números naturales, cuyos primeros elementos son:

$$1, 1, 2, 3, 5, 8, 13, 21, ...$$

Esta sucesión comienza con los numeros naturales `1` y `1`, y partir de ellos, cada elemento es la suma de los dos anteriores. Es decir, se define matemáticamente de la siguiente forma:

```
fibnat (0) = 1

fibnat (1) = 1

fibnat (n+2) = fibnat (n) + fibnat (n+1)
```

Diseñe la función `fibnat` que, dado un número natural `n`, devuelve el elemento de la sucesión de Fibonacci que se ubica en la posición `n`. Use el evaluador paso a paso para calcular `(fibnat 5)`.

Ejercicio 6. Diseñe una función `sigma: Natural (Natural -> Number) -> Number`, que dados un número natural `n` y una función `f`, devuelve la sumatoria de `f` para los valores de `0` hasta `n`. Es decir, calcular:

$$\sum_{i=0}^{n} f(i)$$

Por ejemplo,

```
(check-expect (sigma 4 sqr) 30)
(check-expect (sigma 10 identity) 55)
```

Atención. Dado que la función `f` devuelve un `Number`, es necesario usar el operador `+` en lugar de la función `sumanat`.

Ejercicio 7. A partir del ejercicio anterior, es posible calcular sumatorias que encontramos frecuentemente en los cursos de matemática. Simplemente tenemos que definir la función dentro de la sumatoria y luego combinarla con la función `sigma`.

Siguiendo esta idea, diseñe funciones `R`, `S` y `T` definidas como sigue:

$R(n) = \sum_{i=0}^{2n} \frac{1}{2^i} \quad S(n) = \sum_{i=0}^{n} \frac{i}{i+1} \quad T(n) = \sum_{i=0}^{n} \frac{1}{i+1}$

Observación: Como no conocemos el propósito de estas funciones, sólo pedimos que para las mismas de sus signaturas, casos de prueba y definición.

Atención: Para los casos de test, quizás le convenga explorar la función `check-within`, que es útil para realizar testing con valores no exactos.

Ejercicio 8. Diseñe una función `component: (Number -> Number) Natural Number -> Number` que, dados una función `f`, un natural `n` y un número `x`, devuelva el resultado de aplicar `n` veces la función `f` a `x`.

Por ejemplo:

```
(check-expect (component sqr 2 5) 625)
(check-expect (component add1 5 13) 18)
```

Presente al menos dos ejemplos más en su diseño.

Ejercicio 9. Diseñe la función `intervalo`, que dado un número natural `n`, devuelve la lista `(list n n+1 ... n)`.

Ejercicio 10. Diseñe una función `multiplos` que tome dos naturales `n` y `m`, y devuelva una lista con los primeros `n` múltiplos positivos de `m`, en orden inverso: `m` + `n`, `m` + `(n-1)`, ..., `m` + `1`, `m`.

Por ejemplo:

```
(check-expect (multiplos 4 7) (list 28 21 14 7))
(check-expect (multiplos 0 11) empty)
```

Ejercicio 11. Diseñe una función `list-fibonacci` que, dado un número `n`, devuelve una lista con los primeros `n+1` valores de la sucesión de fibonacci, ordenados de mayor a menor.

Por ejemplo:

```
(check-expect (list-fibonacci 4)
              (list 5 3 2 1 1))
(check-expect (list-fibonacci 0)
              (list 1))
```

Provea dos implementaciones diferentes para esta función en base a los siguientes puntos.

- La primera, de nombre `list-fibonacci-map`, debe definirse a partir de las funciones `map`, `fibnat` e `intervalo`.
- La segunda, de nombre `list-fibonacci-rec`, debe definirse recursivamente. Es decir, para `n` debe devolver `(list 1)` y para `1` debe devolver `(list 1 1)`. Para `n` > `1`, primero debe llamarse recursivamente en `(sub1 n)` para obtener una lista `l` con los primeros `n` valores de la sucesión de fibonacci, y luego debe devolver la lista que resulta de agregar a `l` la suma de sus dos primeros elementos.

¿Qué implementación le parece que se ejecutará más rápido ¿Por qué?

Observación. Racket dispone de la función `time` para medir el tiempo de ejecución de una función. Por ejemplo, puede evaluar las expresiones `(time (list-fibonacci-map 30))` y `(time (list-fibonacci-rec 30))`. De los tres tiempos reportados por `time` mire el de 'cpu time', que reporta el tiempo de CPU medido en milisegundos.

Ejercicio 12. Una persona solicita un préstamo a una entidad bancaria y se compromete a devolver el importe `total` del préstamo más intereses en `n` cuotas mensuales crecientes, con una tasa de interés del 1% anual. La cuota `i`-ésima, con `i` que va de `1` hasta `n`, se calcula sumando dos valores:

- la parte correspondiente a la devolución del préstamo: `total/n`;
- la parte correspondiente a los intereses de la cuota: `(total/n) * (i/100*12)) + 1`.

Diseñe una función `cuotas` que dado un importe `total` de un préstamo, un valor `n` correspondiente al número de cuotas, una tasa `i` de interés, devuelva una lista con las cuotas a pagar ordenadas de forma creciente.

Por ejemplo:

```
(check-expect (cuotas 10000 0 10)
              empty)
(check-expect (cuotas 10000 1 12)
              (list 10100))
(check-expect (cuotas 30000 3 12)
              (list 30100 30200 30300))
(check-expect (cuotas 100000 4 10)
              (list 25375 25750 26125 26500))
```

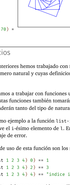
Ejercicio 13. Diseñe una función `circulos` que tome un número natural `n` y devuelva una imagen cuadrada de lado $2*n^2$ con `n` círculos azules centrados y radios: n^2, $(n-1)^2$, ..., 2^2, 1 respectivamente.

Por ejemplo:

```
(circulos 10) ▸
```

Ejercicio 14. Diseñe una función `cuadrados` que tome un número natural `n`, un ángulo `ang` y devuelva una imagen cuadrada de lado `200` con `n` cuadrados azules centrados. Los cuadrados azules tendrán lados de tamaño: n^2, $(n-1)^2$, ..., 2^2, 1 respectivamente. El ángulo `ang` indica la rotación del cuadrado de mayor dimensión. El ángulo que corresponde al cuadrado de lado $(n-1)^2$ debe ser de `20` grados mayor que el que le corresponde al cuadrado de lado n^2, para cualquier valor de `n` mayor o igual a `1`.

Por ejemplo:



```
(cuadrados 10 70) ▸
```

2 Más ejercicios

En los ejercicios anteriores hemos trabajado con funciones que tomaban entre sus argumentos un número natural y cuyas definiciones dependían de si ese natural era cero o positivo.

En esta sección, vamos a trabajar con funciones un poco más complejas. Además de un número natural, estas funciones también tomarán como argumento una lista, y sus definiciones dependerán tanto del tipo de natural como de lista.

Vamos a tomar como ejemplo a la función `list-ith` que recibe una lista `l` y un número natural `i` y devuelve el `i`-ésimo elemento de `l`. En caso de que el índice no sea válido, devuelve un mensaje de error.

Algunos ejemplos de uso de esta función son los siguientes:

```
(list-ith (list 1 2 3 4) 0) ⇒ 1
(list-ith (list 1 2 3 4) 2) ⇒ 3
(list-ith (list 1 2 3 4) 4) ⇒ "indice invalido"
```

Para definir `list-ith` deberemos considerar múltiples condiciones. En primer lugar, si `l` es la lista vacía, tenemos que devolver el mensaje de error. Ahora, si `l` no es vacía e `i` es cero, tenemos que devolver el primer elemento de `l`. El caso no trivial sucede cuando `l` no es vacía e `i` no es cero. Pero esto es equivalente a devolver el (`i`-1)-ésimo elemento de `l` sin su primer elemento, es decir, simplemente tenemos que llamarnos recursivamente sobre `(rest l)` y `(sub1 i)`.

El código queda entonces:

```
; list-ith: (Listof X) Natural -> X
; Dada una lista 'l' y un natural 'i', retorna el i-ésimo elemento de l
(define (list-ith l i)
  (cond
    [(empty? l) "indice invalido"]
    [(zero? i) (first l)]
    [else (list-ith (rest l) (sub1 i))]))
```

A continuación, se proponen ejercicios para poner en práctica estos conceptos.

Ejercicio 15. Diseñe una función `list-invert` que reciba una lista, un elemento `e` y un número natural `n`, e inserte `e` en la `n`-ésima posición de la lista. En el caso de que `i` sea mayor o igual al largo de la lista, entonces `e` se debe agregar al final de la lista.

Por ejemplo:

```
(check-expect (list-invert (list 5 9 10 -2) 8 0) (list 8 5 9 10 -2))
(check-expect (list-invert (list 5 9 10 -2) 8 2) (list 5 9 8 10 -2))
(check-expect (list-invert (list 5 9 10 -2) 8 4) (list 5 9 10 -2 8))
(check-expect (list-invert empty 8 3) (list 8))
```

Ejercicio 16. Diseñe una función `tomar` que reciba una lista `y` un número natural `n` y devuelva la lista que resulta de tomar los primeros `n` elementos de la lista. En el caso de que `n` sea mayor o igual al largo de la lista, entonces se debe devolver la lista original.

Por ejemplo:

```
(check-expect (tomar (list 5 2 1 9) 0) empty)
(check-expect (tomar (list 5 2 1 9) 2) (list 5 2))
(check-expect (tomar (list 5 2 1 9) 4) (list 5 2 1 9))
(check-expect (tomar (list 5 2 1 9) 7) (list 5 2 1 9))
```

Ejercicio 17. Diseñe una función `eliminar-n` que reciba una lista, un elemento `e` y un natural `n` y elimine las primeras `n` ocurrencias del elemento `e` en la lista. En caso de que `n` sea mayor o igual al número de ocurrencias del elemento, debe eliminar todas las ocurrencias.

Por ejemplo:

```
(check-expect (eliminar-n (list 1 2 1 1 2 1) 1 3) (list 2 1 2 1))
(check-expect (eliminar-n (list 1 2 2 1 2) 1 3) (list 2 2 2))
```

Ejercicio 18. Diseñe una función `member-n` que reciba una lista, un elemento `e` y un natural `n` y determine si existen exactamente `n` ocurrencias de `e` en la lista.

Por ejemplo:

```
(check-expect (member-n (list 1 2 1 1 1 2 5) 5 1) #t)
(check-expect (member-n (list 1 2 2 2 1) 1 3) #f)
(check-expect (member-n (list 2 2 2 1 2 2) 2 3) #f)
```