Universidad del Valle
Facultad de Ingeniería
Escuela de Ingeniería de Sistemas y Computación
Fundamentos de Programación
Víctor Bucheli, Ph. D.

***Este documento se basa en los capítulos 25, 26, 30, 31 y 32 del libro how to design programs. Es realizado sólo con fines académicos para el curso de Fundamentos de programación de la Escuela de Ingeniería de Sistemas y Computación*.**

Recursión generativa y funciones acumulativas

1. Una nueva forma de recursión

Las funciones desarrolladas hasta ahora entran dentro de dos amplias categorías. Por una parte, funciones que encapsulan conocimiento de un dominio. Por otra parte, funciones que consumen estructuras de datos. Estas funciones típicamente descomponen sus argumentos en sus componentes estructurales inmediatos y después procesan esos componentes. Si uno de los componentes inmediatos pertenece a la misma clase de datos que la entrada, la función es recursiva; y se denomina como **funciones (estructuralmente) recursivas**. En algunos casos, sin embargo, también se necesitan funciones basadas en una diferente forma de recursión, conocida como, ***recursión generativa***.

Un problema puede ser visto como una instancia de una gran clase de problemas y un algoritmo aplica para todos éstos problemas. En general, un algoritmo divide un problema en otros más pequeños y los resuelve. Por ejemplo, un algoritmo para planear un viaje de vacaciones requiere solucionar pequeños viajes, así el viaje de nuestro hogar al aeropuerto, un vuelo que nos lleve a un aeropuerto cercano a nuestro lugar de vacaciones, un recorrido de este aeropuerto hasta nuestro punto de estancia. El problema entero se resuelve combinando las soluciones para cada uno de estos problemas.

Diseñar un algoritmo recursivo distingue dos clases de problemas: los que son **trivialmente solucionables** y los que no lo son. Si un problema dado es trivialmente solucionable, un algoritmo produce la solución correspondiente. Por ejemplo, el problema de ir de nuestro hogar a un aeropuerto próximo puede ser trivialmente solucionable. Podemos ir en carro, tomar un taxi, o pedirle a un amigo que nos lleve. Si no, el algoritmo genera un nuevo problema y una nueva solución

para esos nuevos problemas. Un viaje con varias escalas es un ejemplo de un problema que no es trivial y pueda ser solucionado generando nuevos problemas más pequeños. En alguna circunstancia computacional uno de los problemas más pequeños pertenece a menudo a la misma clase de problemas que el original, y es por esta razón que llamamos a dicho enfoque **Recursión Generativa.**

Sabemos que este proceso (recrusión generativa) es mucho más una actividad ***ad hoc*** que el diseño ***impulsado-por-datos*** de funciones estructuralmente recursivas (vistas anteriormente). De hecho, es mejor llamarlo ***invención*** de un algoritmo que diseño de uno. Inventar un algoritmo requiere una nueva visión –un ***eureka***. Algunas veces se requiere muy poca inspiración. Por ejemplo, solucionar un ***problema*** simplemente podría requerir la enumeración de una serie de números. En otras ocasiones, sin embargo, se requeriría apoyarse en un teorema matemático referente a números. O, puede aprovechar una serie de resultados matemáticos de sistemas de ecuaciones. Para adquirir una buena comprensión de este proceso del diseño, es necesario estudiar ejemplos y desarrollar un sentido para las varias clases de ejemplos.

Terminología: en algunas ocasiones no se distingue entre la recursión estructural y la recursión generativa, y se refieren a ambos tipos de funciones como algoritmos. En otros casos se utiliza la terminología de métodos **recursivos e iterativos.**

1.1. Modelando una pelota en una mesa

Consideremos el problema que parece simple de modelar los movimientos de una pelota a través de una mesa. Se asume que la pelota rueda a una velocidad constante hasta que cae de la mesa, momento en el cual el programa para. Podemos modelar la mesa con un lienzo de anchura y altura fijas (un cuadrado). La pelota es un punto que se mueve a través del lienzo, lo cual expresamos con el dibujo del punto, un tiempo de espera, y borrado del punto, hasta que el punto está fuera de límites.

El siguiente código “Auxiliares para mover-hasta-fuera” muestra las funciones, la estructura de los datos, y las definiciones de variables que modelan la pelota:

1. Una *pelota* es una estructura con cuatro campos: la posición actual y la velocidad en cada dirección *x* y *y*. Es decir, los primeros dos números en la estructura pelota es la posición actual respecto al lienzo, y los dos números siguientes describen cómo y qué tanto la pelota se mueve por pasos en las dos direcciones.
2. La función *mover-pelota* modela el movimiento físico de la pelota. Consume una pelota y devuelve una pelota, modelando un paso.

3. La función *dibujar-y-borrar* dibuja la pelota en su posición actual, después espera un momento, y la borra otra vez.

La definición de variables especifica las dimensiones del lienzo y el delay o tiempo de espera.

```
;; Paquete de enseñanza: draw.ss

(define-struct pelota (x y delta-x delta-y))
;; Una pelota es una estructura:
;; (make-pelota número número número número)

;; dibujar-y-borrar : una-pelota  ->  true
;; dibuja, duerme, borra un disco del lienzo
;; diseño estructural, conocimiento Scheme
(define (dibujar-y-borrar una-pelota)
  (and
   (draw-solid-disk (make-posn (pelota-x una-pelota) (pelota-y una-
pelota)) 5 'red)
   (sleep-for-a-while DEMORA)
   (clear-solid-disk (make-posn (pelota-x una-pelota) (pelota-y una-
pelota)) 5 'red)))

;; mover-pelota : pelota  ->  pelota
;; crear una nueva pelota, modelando un movimiento para una-pelota
;; diseño estructural, conocimiento de física
(define (mover-pelota una-pelota)
  (make-pelota (+ (pelota-x una-pelota) (pelota-delta-x una-pelota))
               (+ (pelota-y una-pelota) (pelota-delta-y una-pelota))
               (pelota-delta-x una-pelota)
               (pelota-delta-y una-pelota)))
;; Dimensión del lienzo
(define ANCHO 200)
(define ALTO 200)
(define DEMORA .1)
```

Auxiliares para *mover-hasta-fuera*

Ahora la parte fácil es definir *fuera-de-limites?* una función que determina si una pelota dada sigue siendo visible en el lienzo:

```
;; fuera-de-limites?  :  una-pelota  -->  booleano
;;  determina si una-pelota está fuera de los límites
;;  conocimiento del dominio, geometría
    (define (fuera-de-limites? una-pelota)
```

```
(not  (and  (<= 0 (pelota-x una-pelota) ANCHO)
            (<= 0 (pelota-y una-pelota) ALTO))))
```

Sin embargo, en contraste a lo visto hasta ahora, escribir una función que dibuje la pelota en el lienzo hasta que esté fuera de los límites ***pertenece a un grupo programas que no se había encontrado hasta el momento***.

```
;; mueve-hasta-afuera : una-pelota --> true
;; modela el movimiento de una pelota hasta que sale de los limites
(define (mueve-hasta-afuera una-pelota) ...)
```

La función consume una pelota y dibuja su movimiento en un lienzo, adicionalmente produce **true.** Diseñar la recursión con la receta para estructuras no tiene sentido, sin embargo, está ya claro cómo vamos a *dibujar-y-borrar* la pelota y cómo moverla, también. Lo que se necesita en cambio es una distinción del caso que verifica si la pelota está fuera de límites o no.

Refinemos el encabezamiento de la función con una expresión condicional apropiada:

```
(define (mueve-hasta-afuera una-pelota)
  (cond
    [(fuera-de-límites? una-pelota) ...]
    [else ...]))
```

La función “estar fuera de límites " es trivial.

Si la pelota consumida por *mueve-hasta-afuera* está fuera de los límites del lienzo, la función puede producir true, siguiendo el contrato. Si la pelota todavía está dentro de los límites, dos cosas pasarán. Primero, la pelota se debe dibujar y borrar del lienzo. En segundo lugar, la pelota debe ser movida, y entonces debemos repetir todo otra vez. Esto implica que después de mover la pelota, aplicaremos *mueve-hasta-afuera* otra vez, esto significa que la función es recursiva.

```
;; mueve-hasta-afuera : una-pelota  ->  true
;; modelar el movimiento de una pelota hasta que este fuera de los limites
(define (mueve-hasta-afuera una-pelota)
  (cond
    [(fuera-de-limites? una-pelota) true]
    [else (and (dibujar-y-borrar una-pelota)
               (mueve-hasta-afuera (mover-pelota una-pelota)))]))
```

Las dos funciones (*dibujar-y-borrar una-pelota*) y (*mueve-hasta-afuera* (*mover-pelota una-pelota*)) producen **true**, y ambas expresiones deben ser evaluadas. Así que podemos combinarlas con una expresión **and**.

Podemos ahora probar la función como sigue:

(**start** ANCHO ALTO)
(*mueve-hasta-afuera (make-pelota 111 20 -3 +1)*)
(**stop**)

Esto crea un lienzo y una pelota que se mueve hasta que sale del lienzo.

Una inspección cuidadosa a la definición de la función revela dos particularidades: primero, aunque la función es recursiva, su cuerpo consiste en una **cond** expresión donde las condiciones no tienen nada que ver con los datos de la entrada, recuerde por ejemplo, que en los casos de recursión ya vistos el caso de parada es cuando llega a vacío. En segundo lugar, la aplicación de la recursividad en el cuerpo no consume una parte de la entrada. En cambio, *mueve-hasta-afuera* genera una completamente nueva y diferente *estructura pelota*, que representa la pelota original después de un paso, y la utiliza para la recursión. Claramente,

Ejercicio 1: ¿Cree usted que ninguna de nuestras recetas del diseño vistas previamente permite la definición recursiva anterior. Hemos encontrado una nueva manera de programar?

Ejercicio 2: Desarrollar *mover-pelotas*. La función consume una lista de pelotas y mueve cada una hasta que todas se han movido fuera de los límites. **Sugerencia:** Es mejor escribir esta función usando **filter**, **andmap** y funciones abstractas similares de la parte IV.

1.2. Ordenamiento rápido

El algoritmo ***ordena-rápido*** o Quiksort es el ejemplo clásico de recursión generativa. Como la función ***ordena*** definida en talleres anteriores, ***ordena-rápido*** es una función que consume una lista de números y produce una lista que contiene los mismos números en orden ascendente. La diferencia entre las dos funciones es que ***ordena*** se basa en recursión estructural o recursión a partir de los datos y ***ordena-rápido*** se basa en recursión generativa.

La idea subyacente del paso generativo es una estrategia para resolver problemas: divide y vencerás. Es decir, dividimos los casos no-triviales del problema en dos más pequeños, relacionando problemas, resolviendo esos problemas más pequeños, y combinando sus soluciones en una solución para el problema original.

En el caso de ***ordena-rápido***, el objetivo intermedio es dividir la lista de números en dos listas: una de ellas contiene todos los ítems estrictamente más pequeños que el primer ítem, y la otra contiene todos los ítems que son estrictamente más grandes que el primer ítem. Después las dos listas más pequeñas son ordenadas usando el mismo procedimiento. Una vez que las dos listas se ordenan, simplemente se yuxtaponen las piezas. Debido a su papel especial, el primer ítem en la lista es a menudo llamado el ítem pivote.

Veamos un ejemplo, supongamos que la entrada es:

(**list** 11 8 14 7)

El ítem pivote es 11, el primero de la lista. Dividiendo la lista en ítems que son más grandes y más pequeños que 11, se producen dos listas:

(**list** 8 7) y (**list** 14)

La segunda ya está ordenada en orden ascendente, ordenando el primero tenemos (**list** 7 8). Esto deja tres pedazos de la lista original:

1. (**list** 7 8), la versión ordenada después de aplicar el mismo procedimiento;
2. 11; y
3. (**list** 14), la versión ordenada de la lista con los números más grandes.

Para producir una versión ordenada de la lista original, se deben concatenar las tres piezas que proporcionen el resultado deseado: (**list** 7 8 11 14).

La ilustración saca a relucir cómo *ordena-rápido* sabe cuando parar? Dado que es una función basada en recursión generativa, la respuesta general es que se detiene cuando el problema de ordenación es trivial (es decir no se puede dividir más). Claramente, **empty** es una entrada trivial para *ordena-rápido*, porque la única versión a ordenar es **empty**.

Ejercicio 3: Simular los pasos de *ordena-rápido* para (**list** 11 9 2 18 12 14 4 1).

Ahora que se tiene un buen entendimiento de los pasos generativos, se puede trasladar el proceso descrito a Scheme. La descripción sugiere que *ordena-rápido* distinga dos casos. Si la entrada es **empty**, produce **empty**. Por otra parte, realiza una recursión generativa, lo que sugiere una **cond**-expresión:

```
;; ordena-rápido : (listade número) ->(listade número)
;; crear una lista de números con los mismos números que lista de números,
;; obtiene una lista ordenada en forma ascendente.
```

```
(define (ordena-rápido uldn)
  (cond
    [(empty? uldn) empty]
    [else ...]))
```

La respuesta para este primer caso está dada. Para el segundo caso, cuando las entradas de *ordena-rápido* no sean **empty**, el algoritmo usa el primer ítem para particionar el resto de la lista en dos sublistas: una lista que tiene todos los ítems más pequeños que el ítem pivote y otra con los ítems más grandes que el ítem pivote.

Dado que el resto de la lista es de tamaño desconocido, queda la tarea de particionar la lista en dos funciones auxiliares: ítems-menores e ítems-mayores. Estas funciones procesan la lista y filtran aquellos ítems que son más pequeños y más grandes respectivamente que el primero (ítem pivote). De ahí que cada función auxiliar acepta dos argumentos, o sea una lista de números y un número. Desarrollar estas funciones es, desde luego, un ejercicio de recursión estructural; su definición se muestra a continuación.

Cada sublista se ordena separadamente usando *ordena-rápido*. Esto implica que el uso de la recursión y más específicamente, el seguimiento de las dos expresiones:

1. (*ordena-rápido* (*ítems-menores uldn* (**first** *uldn*))), que ordena la lista de ítems más pequeños que el pivote, y
2. (*ordena-rápido* (*ítems-mayores uldn* (**first** *uldn*))), que ordena la lista de ítems más grandes que el pivote.

```
;; ordena-rápido : (lista-de números)  ->  (lista-de números)
;; crear una lista de números con los mismos números que uldn,
;; ordenada en orden ascendente
;; asumir que todos los números son distintos
(define (ordena-rápido uldn)
  (cond
    [(empty? uldn) empty]
    [else (append (ordena-rápido (ítems-menores uldn (first uldn)))
                  (list (first uldn))
                  (ordena-rápido (ítems-mayores uldn (first uldn))))]))

;; ítems-mayores : (lista-de números)  ->  (lista-de números)
;; crear una lista con todos los números en uldn
;; que son más grandes que el pivote
(define (ítems-mayores uldn pivote)
  (cond
    [(empty? uldn) empty]
```

```
      [else (if (> (first uldn) pivote)
                (cons (first uldn) (ítems-mayores (rest uldn) pivote))
                (ítems-mayores (rest uldn) pivote))]))

;; ítems-menores : (lista-de números)  ->  (lista-de números)
;; crear una lista con todos los números en uldn
;; que son más pequeños que el pivote
(define (ítems-menores uldn pivote)
  (cond
    [(empty? uldn) empty]
    [else (if (< (first uldn) pivote)
                (cons (first uldn) (ítems-menores (rest uldn) pivote))
                (ítems-menores (rest uldn) pivote))]))
```

Algoritmo de *ordena-rápido*

Una vez que se tienen ordenadas las versiones de las dos listas, se necesita una función que combine las dos listas y el pivote. La función **append** de Scheme logra esto:

```
(append (ordena-rápido (ítems-menores uldn (first uldn)))
        (list (first uldn))
        (ordena-rápido (ítems-mayores uldn (first uldn))))
```

Ejercicio 4: Mientras que *ordena-rápido* reduce rápidamente el tamaño del problema en muchos casos, es inapropiadamente lento para problemas con listas pequeñas. Se emplea a menudo *ordena-rápido* para listas grandes.

Desarrollar una versión de ***ordena-rápido*** que use ***ordena***, si la longitud de la entrada está debajo de algún umbral.

Ejercicio 5: Si la entrada para *ordena-rápido* contiene el mismo número varias veces, el algoritmo regresa una lista que es siempre más corta que la de entrada. ¿Por qué? Resuelva el problema para que la salida sea de la misma longitud que la entrada.

Ejercicio 6: Utilice la función **filter** para definir *ítems-menores* e *ítems-mayores*.

2. Diseño de algoritmos

En primera instancia, los algoritmos *mueve-hasta-afuera* y *ordena-rápido* parecen tener poco en común. Uno procesa estructuras y el otro procesa listas. Uno crea una nueva estructura para el paso generativo; el otro divide una lista en tres partes y recurre en dos de éstas. En pocas palabras, una comparación entre los dos ejemplos de recursión generativa sugiere que el diseño de algoritmos es una actividad con fines específicos y que es imposible tener una receta general de diseño. Un enfoque más atento, sin embargo, sugiere una perspectiva diferente.

Primero, aun cuando se habla de algoritmos como procesos que resuelven problemas no se trata más que de funciones que consumen y producen datos. En otras palabras, se emplean datos para representar un problema y se debe entender definitivamente la naturaleza de los mismos. En segundo lugar, se describe el proceso en términos de datos, por ejemplo: “crear una nueva estructura” o “particionar una lista de números”. Tercero, siempre se distinguen aquellos datos de entrada para los cuales es trivial producir una solución, así como aquellos para los que no lo es. Cuarto, la generación de problemas es la clave para el diseño de algoritmos. Finalmente, una vez que los problemas generados han sido resueltos, las soluciones deben ser combinadas con otros valores.

Examinemos los seis escenarios generales de la receta estructural de diseño para ilustrar esta discusión:

2.1.1. Análisis de datos y diseño

La elección de una representación de datos para un problema normalmente afecta el razonamiento acerca del proceso. Algunas veces la descripción de un proceso determina una representación en particular. En otras ocasiones, es posible y fructífero explorar otras alternativas. En cualquier caso se debe analizar y definir las colecciones de datos.

2.1.2. Contrato, propósito, encabezado

También es necesario un contrato, un encabezado de definición y un propósito. Partiendo de que el paso generativo no tiene conexión con la estructura de la definición de datos, el propósito no solamente debe definir **qué** hace la función sino además incluir un comentario que explique en términos generales **cómo** lo hace.

2.1.3. Ejemplos de la función

En las recetas anteriores, los ejemplos de la función especificaban meramente que dato debía producir la función para determinado dato de entrada. Para los algoritmos, los ejemplos deben ilustrar **cómo** se comporta el algoritmo a partir de una entrada determinada. Para el caso de funciones

como *mueve-hasta-afuera* el proceso es trivial y no necesita más que algunas palabras. Para otras, incluida *ordena-rápido*, el proceso se relaciona con una idea no trivial para su paso generativo y su explicación requiere de un buen ejemplo.

2.1.4. Plantilla

El análisis sugiere un formato general para algoritmos:

```
(define (fun-recursión-generativa problema)
 (cond
  [(trivialmente-solucionable? problema)
   (determinar-solución problema)]
  [else
   (combinar-soluciones
    ... problema ...
    (fun-recursión-generativa
       (generar-problema-1 problema))
     (fun-recursión-generativa
       (generar-problema-n problema)))]))
```

2.1.5. Definición

Desde luego, este formato es simplemente una sugerencia y no una forma definitiva. Cada función en el formato sirve para recordarnos que necesitamos pensar en las siguientes cuatro preguntas:

1. ¿Qué es un problema trivialmente solucionable?
2. ¿Cuál es la solución correspondiente?
3. ¿Cómo generamos nuevos problemas más fáciles de solucionar que el problema original? ¿Se genera un nuevo problema o se generan varios?
4. ¿La solución del problema dado es la solución de los nuevos o alguno de los nuevos problemas?, ó, ¿se necesitan combinar las soluciones para crear una solución para el problema original? Y, si es el caso, se necesita alguno de los datos del problema original?

Para definir el algoritmo, se deben expresar las respuestas a estas cuatro preguntas en términos de la representación de datos elegida.

2.1.6. Prueba

Una vez que se tiene una función completa, debemos probarla. Tal como se hacía antes, el acierto de realizar pruebas es descubrir errores y eliminarlos.

Recuerde que realizar pruebas, no valida que la función trabaja correctamente para todas las posibles entradas. También recuerde que es mejor formular pruebas con forma de expresiones booleanas que automáticamente comparan el valor esperado con el valor producido.

Ejercicio 7: Formular respuestas informales a las cuatro preguntas clave para el problema de modelar el movimiento de una pelota a través de un lienzo mientras se encuentra fuera de los límites.

2.2. Terminación

Desafortunadamente, la receta estándar no es lo suficientemente buena para el diseño de algoritmos. Hasta ahora, una función siempre ha producido una salida para una entrada legítima. Esto es, la evaluación siempre se detiene. Después de todo, por la naturaleza de nuestra receta, cada recursión natural consume una pieza inmediata de la entrada y no la entrada tal cual. Debido a que los datos son generados de manera jerárquica, esto significa que la entrada es reutilizada en cada caso. Por tanto, la función tarde o temprano consume una pieza atómica de datos y se detiene.

Con funciones basadas en recursión generativa, esto ya no es cierto. Las recursiones internas no consumen un componente inmediato de la entrada sino nuevas piezas de datos, que son generadas a partir de la entrada. Como el ejercicio de orden-rapido, lo demuestra, este paso produce la misma entrada una y otra vez impidiendo que la evaluación siempre produzca un resultado. Se dice que el programa entra en un **ciclo infinito** (LOOP).

En suma, incluso el más mínimo de los errores al traducir la descripción del proceso en una función causará un ciclo infinito. El problema se comprende más fácilmente con un ejemplo.

Considerar la siguiente definición de *ítems-menores*, uno de los dos “generadores de problemas” para *ordena-rápido*:

```
;; ítems-menores : (listade número) número  ->  (listadenúmero)
 ;; crea una lista con todos aquellos números en uldn
 ;; menores o iguales al segundo argumento
        (define (ítems-menores uldn límite)
          (cond
           [(empty? uldn) empty]
           [else (if (<= (first uldn) límite)
                     (cons (first uldn) (ítems-menores (rest uldn)límite))
```

(*ítems-menores* (**rest** *uldn*) *límite*))]))

En vez de **<** , ésta emplea **<=** para comparar números. Como resultado, esta función produce (**list** 5) cuando se aplica a (**list** 5) y 5.

Peor aún, si la función *ordena-rápido* es combinada con la nueva versión de *ítems-menores*, ésta no produce ningún resultado para (**list** 5):

```
  (ordena-rápido (list 5))
= (append (ordena-rápido (ítems-menores 5 (list 5)))
          (list 5)
          (ordena-rápido (ítems-mayores 5 (list 5))))
= (append (ordena-rápido (list 5))
          (list 5)
          (ordena-rápido (ítems-mayores 5 (list 5))))
```

La primera llamada recursiva demanda que *ordena-rápido* resuelva el problema ordenando (**list** 5) – pero éste es exactamente el problema con el que empezamos. Dado que esta es una evaluación circular, (*ordena-rápido* (**list** 5)) nunca produce un resultado. Generalmente, no hay garantía de que el tamaño de la entrada para una llamada recursiva nos lleve a una solución más cercana que la de la entrada original.

La lección de este ejemplo es que el diseño de algoritmos requiere un paso más en la receta de diseño: un ARGUMENTO DE TERMINACIÓN, el cual explica por qué el proceso produce una salida para cada entrada y cómo la función implementa esta idea, o una advertencia, la cual explica cuando el proceso podría no terminar. Para *ordena-rápido*, el argumento podría ser como éste:

> En cada paso, *ordena-rápido* divide la lista en dos sublistas empleando *ítems-menores* e *ítems-mayores*. Cada función produce una lista que es menor que la entrada (el segundo argumento), incluso cuando el punto de arranque (el primer argumento) es un elemento de la lista. Por tanto cada aplicación recursiva de *ordena-rápido* consume una lista estrictamente más corta que la lista dada. Eventualmente, *ordena-rápido* recibe y regresa **empty**.

Sin un argumento como éste, un algoritmo debe ser considerado incompleto.

Un buen argumento de terminación también debe revelar casos de paro adicionales. Por ejemplo, (*ítems-menores* N (**list** N)) e (*ítems-mayores* N (**list** N)) siempre producen **empty** para cualquier N. Por tanto se sabe que la respuesta de

ordena-rápido para (**list** N) es (**list** N). Para agregarle este conocimiento a *ordena-rápido*, simplemente agregamos una cláusula **cond**:

```
(define (ordena-rápido uldn)
 (cond
  [(empty? uldn) empty]
  [(empty? (rest uldn)) uldn]
  [else (append
          (ordena-rápido (ítems-menores uldn (first uldn)))
           (list (first uldn))
          (ordena-rápido (ítems-mayores uldn (first uldn))))]))
```

La condición (**empty?** (**rest** *uldn*)) es una manera de preguntar si *uldn* contiene un elemento.

La siguiente tabla resume las sugerencias sobre el diseño de algoritmos. Los puntos indican que el diseño de un algoritmo requiere un nuevo paso: el argumento de paro. Leer la tabla en conjunto con las de los capítulos anteriores.

Fase	Acierto	Actividad
Ejemplos	Caracterizar la Relación entrada-salida y el proceso computacional mediante ejemplos.	• Crear y mostrar ejemplos de problemas solucionables trivialmente. • Crear y mostrar ejemplos que requieren procesamiento recursivo. • Ilustrar como trabajar a través de los ejemplos.
Cuerpo	Definir un algoritmo.	• Formular pruebas para problemas con solución trivial. • Formular respuestas para los casos triviales. • Determinar como generar nuevos problemas a partir del problema dado, posiblemente empleando funciones auxiliares. • Determinar como combinar las soluciones de estos problemas en una solución del problema dado.
Paro	Argumentar que el algoritmo termina para todas las entradas posibles.	Mostrar que las entradas a las aplicaciones recursivas son más pequeñas que la entrada dada.

Diseñando algoritmos

2.3. Recursión estructural contra recursión generativa tomado una decisión

Un usuario no puede distinguir *ordena* y *ordena-rápido*. Ambos consumen una lista de números y ambos producen una lista que consiste en los mismos números arreglados en orden ascendente. Para un observador, las funciones son completamente equivalentes. Esto produce la pregunta de cuál será la que el programador debe proveer?. Más generalmente, si podemos desarrollar una función usando recursión estructural y una equivalente usando recursión generativa, ¿qué se debe hacer?

Para entender esta elección mejor, vamos a discutir otro ejemplo clásico de recursión generativa de las matemáticas. El problema de encontrar el máximo común divisor de dos números naturales positivos. Todos estos números tienen al menos un divisor en común: 1.

En ocasiones, éste es el único común divisor. Por ejemplo, 2 y 3 tienen solamente al 1 como común divisor porque 2 y 3, respectivamente, son los únicos divisores. Sin embargo, 6 y 25 son números con varios divisores:

1. 6 es divisible por 1, 2, 3, y 6
2. 25 es divisible por 1, 5, y 25

En este caso también, el máximo común divisor de 25 y 6 es 1. En contraste, 18 y 24 tienen muchos divisores en común:

1. 18 es divisible por 1, 2, 3, 6, 9, y 18;
2. 24 es divisible por 1, 2, 3, 4, 6, 8, 12, y 24.

El máximo común divisor es 6.

Con base en esto vamos desarrollar las funciones correspondientes, siguiendo la receta de diseño, se inicia con un contrato, un propósito y un encabezado:

```
;; mcd : N[>= 1] N[>= 1]  ->  N
;; encontrar el máximo común divisor de n y m
(define (mcd n m)
  ...)
```

El contrato especifica las entradas: números naturales mayores o iguales a 1 (no 0).

Ahora se necesita tomar una decisión si es que se quiere obtener un diseño basado en recursión estructural o generativa. Dado que la respuesta no está determinada, se desarrollan las dos. Para la versión estructural, se debe considerar qué entrada debe procesar la función: *n*, *m*, o ambos. Una consideración momentánea sugiere

que lo que realmente se necesita es una función que empieza con el menor de los dos y devuelve el primer número menor o igual a dicha entrada que divide a n y m.

```
;; mcd-estructural : N[>= 1] N[>= 1]  ->  N
;; encontrar el máximo común divisor de n y m
;; recursión estructural usando definición de datos N[>= 1]
(define (mcd-estructural n m)
  (local ((define (primer-divisor-<= i)
      (cond
        [(= i 1) 1]
        [else (cond
                [(and (= (remainder n i) 0)
                      (= (remainder m i) 0))
                 i]
                [else (primer-divisor-<= (- i 1))])])))
    (primer-divisor-<= (min m n))))
```

Encontrar el mayor divisor común mediante recursión estructural

Se usa **local** para definir una función auxiliar apropiada. Las condiciones ``igualmente divisible'' han sido codificadas como (**=** (**remainder** n i) 0) y (**=** (**remainder** m i) 0). Las dos aseguran que *i* divide a *n* y a *m* sin residuo. Probando *mcd-estructural* con ejemplos muestra que encuentra las respuestas esperadas.

Aun cuando el diseño de *mcd-estructural* es adecuado, también es poco probado. Simplemente prueba para cada número si divide tanto a *n* como a *m* y devuelve el primer número que cumpla. Para números naturales pequeños, este proceso funciona muy bien.

Considera el siguiente ejemplo:

(*mcd-estructural* 101135853 45014640)

El resultado es 177 y para llegar ahí *mcd-estructural* tuvo que comparar 101135676, esto es, 101135853 - 177, números. Este es un esfuerzo grande y aun computadoras razonablemente rápidas requieren muchos minutos en dicha tarea.

Ejercicio 8 : Inserta la definición de *mcd-estructural* en la ventana de `Definiciones` y evalúa `(`**time** `(`*mcd-estructural* 101135853 45014640`))` en la ventana de `Interacciones`.

Desde que los matemáticos reconocieron la ineficiencia de los “algoritmos estructurales” mucho tiempo atrás. Estudiaron el problema de encontrar los divisores con mayor profundidad. La percepción esencial es de que para dos números naturales *mayor* y *menor*, su común divisor es igual al máximo común divisor de *menor* y el resto (**remainder**) del *mayor* dividido entre *menor*. Así es como podemos colocar esta percepción de la misma manera:

(*mcd* mayor menor)
= (*mcd* menor (**remainder** mayor menor))

Dado que (**remainder** mayor menor) es menor que ambos *mayor* y *menor*, el uso por la derecha de *mcd* consume a *menor* primero.

Aquí está como es que esta percepción es aplicada a nuestro pequeño ejemplo:

1. Los números dados son 18 y 24.
2. De acuerdo con la percepción de los matemáticos, ambos tienen el mismo máximo común divisor en 18 y 6.
3. Y ambos tienen el mismo máximo común divisor como 6 y 0.

Aquí nos vemos sorprendidos debido a que 0 no es algo esperado. Pero, 0 puede ser dividido por cualquier número, así que hemos encontrado nuestra respuesta: 6.

Trabajando a través del ejemplo no solamente explica la idea pero también sugiere como descubrir el caso con solución trivial. Cuando el menor de los dos números es 0, el resultado es el número más grande. Colocando todo junto, se obtiene la siguiente definición:

```
;; mcd-generativa : N[>= 1] N[>=1]  ->  N
;; encontrar el máximo común divisor de n y m
;; recursion generativa: (mcd n m) = (mcd n (remainder m n)) if
(<= m n)
(define (mcd-generativa n m)
  (local ((define (mejor-mcd mayor menor)
            (cond
              [(= menor 0) mayor]
                    [else (mejor-mcd menor (remainder mayor
menor))])))
    (mejor-mcd (max m n) (min m n))))
```

La definición **local** introduce a la función: *mejor-mcd*, una función basada en la recursión generativa. Su primera línea descubre el caso trivial comparando menor con 0 y produce la solución coincidente. El paso generativo emplea a *menor* como su nuevo primer argumento y (**remainder** mayor menor) como su nuevo segundo argumento para *mejor-mcd*, explotando la ecuación de arriba.

Si ahora se emplea *mcd-generativa* con nuestro ejemplo complejo de arriba:

(*mcd-generativa* 101135853 45014640)

Se observa que la respuesta es casi instantánea. Una evaluación manual muestra que *mejor-mcd* recurre solamente nueve veces antes de producir la solución: 177. En breve, la recursión generativa ha ayudado a encontrar una solución más rápida al problema.

Ejercicio 9: Formular respuestas informales a las cuatro preguntas clave para *mcd-generativa*.

Ejercicio 10: Definir *mcd-generativa* y evaluar

(**time** (*mcd-generativa* 101135853 45014640))

En la ventana de **interacciones**.

Evaluar manualmente (*mejor-mcd* 101135853 45014640). Muestra solamente aquellas líneas que introducen una nueva llamada recursiva a *mejor-mcd*.

Ejercicio 20: Formular un argumento de paro para *mcd-generativa*.

Considerando el ejemplo de arriba, es tentador desarrollar funciones usando recursión generativa. Después de todo, estas producen resultados más rápidos. Este juicio es demasiado aventurado por tres razones. Primero, aun el mejor diseñado de los algoritmos no siempre es más rápido que su función equivalente recursiva estructuralmente. Por ejemplo, *ordena-rápido* gana solamente para listas grandes; para las pequeñas, la función ordena estándar es más rápida. Peor aún, un algoritmo mal diseñado puede afectar por completo el desempeño de un programa. Segundo, típicamente es más sencillo diseñar una función empleando la receta para recursión estructural. Contrariamente, diseñar un algoritmo requiere una idea de cómo generar nuevos problemas menores, paso que generalmente requiere de un enfoque matemático profundo. Finalmente, la gente que lee funciones puede entender fácilmente funciones recursivas estructuralmente, aun sin mucha documentación. Para entender un algoritmo, el paso generativo debe ser

bien explicado, y aun con una muy buena explicación, podría ser aun complicado tener clara la idea.

La experiencia muestra que la mayor parte de las funciones en un programa emplean recursión estructural; solamente unas pocas explotan la recursión generativa. Cuando se encuentra con una situación en la que el diseño puede usar tanto recursión estructural como generativa, el mejor enfoque es comenzar con la versión estructural. Si esta es muy lenta, el diseño alternativo usando recursión generativa deberá ser explorado. Si éste es elegido, es importante documentar la generación del problema con buenos ejemplos y dar un buen argumento de paro.

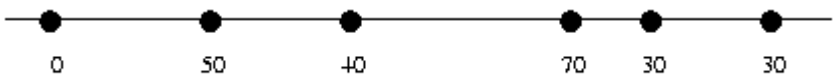
3. La pérdida de conocimiento

Cuando se diseñan funciones recursivas, no se piensa en el contexto de su uso. Si se aplican por primera vez o si están llamadas por centésima vez de una manera recursiva, no importa. Deben trabajar según la declaración que se hace en el propósito, y eso es todo lo que se necesita saber mientras se diseñan los cuerpos de las funciones.

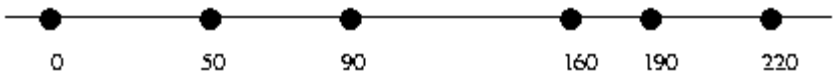
Aunque este principio de independencia de contexto facilita grandemente el desarrollo de funciones, también causa problemas ocasionales. En esta sección, se ilustra un problema que esto genera. Se refieren a la pérdida de conocimiento que ocurre durante una evaluación recursiva. En la subsección se muestra como esta pérdida hace una función estructural recursiva más complicada y menos eficiente.

3.1. Un problema con el proceso estructural

Suponga que se dan las distancias relativas entre una serie de puntos, comenzando en el origen, y supóngase que se deben computar las distancias absolutas desde el origen. Por ejemplo, puede ser una línea como ésta:



Cada número especifica la distancia entre dos puntos. Lo que se requiere está en la figura siguiente, donde cada punto se anota con la distancia al punto extremo izquierdo:



Desarrollar un programa que realice este cálculo es, hasta este punto, un ejercicio de recursión estructural. El programa completo en Scheme podría ser.

```
  ;; relativa-a-absoluta : (listade número) -> (listade número)
;; para convertir una lista de distancias relativas a una lista de distancias  absolutas
    ;; el primer ítem en la lista representa la distancia al origen
    (define (relativa-a-absoluta uldn)
      (cond
        [(empty? uldn) empty]
        [else (cons (first uldn)
                    (agrega-a-todos (first uldn) (relativa-a-absoluta (rest uldn))))]))

     ;; agrega-a-todos : número (listade número) -> (listade número)
    ;; para agregar n a cada número en uldn
    (define (agrega-a-todos n uldn)
      (cond
        [(empty? uldn) empty]
        [else (cons (+ (first uldn) n) (agrega-a-todos n (rest uldn)))]))
```

Convertir distancias relativas a distancias absolutas

Mientras que el desarrollo del programa es fácil, usarlo en listas cada vez mayores revela un problema. Considérese la evaluación de la definición siguiente:

```
(define x (relativa-a-absoluta (list 0 ... N)))
```

Mientras que aumentamos *N*, el tiempo necesario crece más rápido:

N	Tiempo de evaluación
100	220
200	880
300	2050
400	5090
500	7410
600	10420
700	14070
800	18530

En vez de duplicar al pasar de 100 a 200 ítems, el tiempo se cuadruplica. Ésta es también la relación aproximada de ir de 200 a 400, 300 a 600, etc.

Ejercicio 11: Reformular *agrega-a-todos* usando **map** y **local**.

En vista de la simplicidad del problema, la cantidad de “trabajo” que las dos funciones realizan es sorprendente. Si se convirtiera la misma lista manualmente,

se obtendría la distancia total y sólo se agregaría a las distancias relativas cada vez que se diera un paso adicional a lo largo de la línea.

Intentando diseñar una segunda versión de la función que esté más cerca del método manual, la nueva función sigue siendo una función de procesamiento de lista, así se inicia por:

```
(define (rel-a-abs uldn)
  (cond
    [(empty? uldn) ...]
    [else ... (first uldn) ... (rel-a-abs (rest uldn)) …]))
```

El primer ítem de la lista del resultado debe obviamente ser 3, y es fácil construir esta lista. Pero, el segundo debería ser (+ 3 2), entonces la segunda instancia *rel-a-abs* no tiene ninguna manera de “saber” que el primer ítem de la lista original es 3. El “conocimiento” se ha perdido.

El problema es que las funciones recursivas son independientes de su contexto. Una función procesa la lista *L* en (**cons** *N L*) exactamente de la misma manera como *L* en (**cons** *K L*). De hecho, también procesaría *L* de esta manera si se le diera *L* por sí misma. Mientras que esta característica hace fácil el diseño de funciones estructuralmente recursivas, también significa que las soluciones son, en ocasiones, más complicadas que lo necesario, y esta complicación puede afectar el desempeño de la función.

Para compensar la pérdida de “conocimiento", se adiciona la función con un parámetro adicional: *dist-acu*. El nuevo parámetro representa la distancia acumulada, que es el valor guardado cuando se convierte una lista de distancias relativas a una lista de distancias absolutas. Su valor inicial debe ser 0. Mientras que la función procesa los números en la lista, también debe agregarlos a dicho valor.

La función redefinida es:

```
(define (rel-a-abs uldn dist-acu)
  (cond
    [(empty? uldn) empty]
    [else (cons (+ (first uldn) dist-acu)
                (rel-a-abs (rest uldn) (+ (first uldn) dist-acu)))]))
```

La aplicación recursiva consume el resto de la lista y la nueva distancia absoluta del punto actual al origen. Aunque esto significa que dos argumentos están cambiando simultáneamente, el cambio en el segundo depende del primer argumento. La

función sigue siendo un procedimiento sencillo de procesamiento de lista. Un ejemplo:

= (*rel-a-abs* (**list** 3 2 7) 0)
= (**cons** 3 (*rel-a-abs* (**list** 2 7) 3))
= (**cons** 3 (**cons** 5 (*rel-a-abs* (**list** 7) 5)))
= (**cons** 3 (**cons** 5 (**cons** 12 (*rel-a-abs* **empty** 12))))
= (**cons** 3 (cons 5 (**cons** 12 **empty**)))

Cada ítem en la lista se procesa una vez. Cuando *rel-a-abs* alcance el fin de la lista de argumentos, el resultado está completamente determinado y no hay necesidad de más trabajo. En general, la función actúa en orden de *N* pasos de recursiones naturales para una lista con *N* ítems.

Un problema de menor importancia con la nueva definición es que la función consume dos argumentos y no es equivalente a *relativa-a-absoluta*, una función de un argumento. Peor aún, alguien puede emplear mal accidentalmente *rel-a-abs* aplicándolo a una lista de números y a un número que no es 0. Se solucionan ambos problemas con una definición de función que contenga *rel-a-abs* en una definición de **local,** Ahora, *relativa-a-absoluta* y *relativa-a-absoluta2* son indistinguibles.

```
;; relativa-a-absoluta2 : (listade número) -> (listade número)
;; para convertir una lista de distancias relativas a una lista de distancias absolutas
;; el primer ítem en la lista representa la distancia al origen
(define (relativa-a-absoluta2 uldn)
  (local ((define (rel-a-abs uldn dist-acu)
      (cond
        [(empty? uldn) empty]
        [else (cons (+ (first uldn) dist-acu)
                    (rel-a-abs (rest uldn) (+ (first uldn) dist-acu)))]))
    (rel-a-abs uldn 0)))
```

Convirtiendo distancias relativas con un acumulador

4. Diseño de funciones estilo acumulador

En general, agregar un **Acumulador** es decir, un parámetro que acumula conocimiento, es algo que se agrega a una función después de haberse diseñado, no antes. Las claves para el desarrollo de una función estilo-acumulador son:

1. Reconocer que la función se beneficia de, o requiere, un acumulador;
2. Comprender qué representa el acumulador.

4.1. Reconocer la necesidad de un acumulador

Reconocer la necesidad de acumuladores no es una tarea fácil. Se han visto dos razones, las cuales predominan más para agregar parámetros acumuladores. En cualquier caso, es esencial que primero se construya una función completa *con base en una receta de diseño*. Luego al analizar la función para buscar alguna de las siguientes características:

1. Si la función es estructuralmente recursiva y el resultado de una aplicación recursiva es procesado por una función recursiva auxiliar, entonces debe considerarse el empleo de un parámetro acumulador.

Veamos la función invierte como ejemplo:

```
;; invierte : (listade X)  ->  (listade X)
;; construir la inversa de uldx
;; recursión estructural
 (define (invierte uldx)
         (cond
    [(empty? uldx) empty]
    [else (haz-último-ítem (first uldx) (invierte (rest uldx)))]))

;; haz-último-ítem : X (listade X)  ->  (listade X)
;; agrega una-x al final de uldx
;; recursión estructural
       (define (haz-último-ítem una-x uldx)
         (cond
           [(empty? uldx) (list una-x)]
    [else (cons (first uldx) (haz-último-ítem una-x  (rest uldx)))]))
```

El resultado de la aplicación recursiva produce el reverso del resto de la lista. Esto es procesado por *haz-último-ítem*, la cual agrega el primer ítem al reverso del resto y así crea el reverso de la lista entera. Esta segunda función auxiliar es también recursiva. Se tiene así identificado un candidato potencial.

2. Si se está tratando con una función apoyada en recursión generativa, se enfrenta una tarea más difícil. La meta debe ser comprender si el algoritmo puede fallar para producir un resultado para entradas en las cuales esperamos un resultado. En este caso, puede ayudar agregar un parámetro que acumule conocimiento. Estas dos situaciones no representan las únicas, sino las más comunes.

4.2. Funciones estilo acumulador

Cuando se ha decidido que una función estilo acumulador es necesaria se introduce en dos pasos:

4.2.1. Construyendo un acumulador

Primero, se debe comprender qué conocimiento requiere recordar el acumulador acerca de los parámetros apropiados y entonces cómo es recordarlo. Por ejemplo, para la conversión de distancias relativas a absolutas, es suficiente acumular la distancia total encontrada hasta el momento.

La mejor manera de analizar el proceso de acumulación es introducir una plantilla para la función estilo-acumulador mediante una definición **local** y nombrar los parámetros de la función apropiada de modo diferente de aquellos de la función auxiliar.

En el ejemplo de *invierte*:

```
;; invierte : (listade X) -> (listade X)
;; construir el reverso de uldx
(define (invierte uldx0)
(local (;; acumulador ...
            (define (rev uldx acumulador)
              (cond
                [(empty? uldx) ...]
                [else
                 ... (rev (rest uldx) ... ( first uldx) ... acumulador)... ])))
 (rev uldx0 ...)))
```

Se tiene la definición que *invierte* como una función auxiliar *rev* en forma de plantilla. Esta plantilla auxiliar tiene un parámetro adicional: el parámetro acumulador. La aplicación recursiva indica que se requiere una expresión que mantiene el proceso de acumulación y que este proceso dependa del valor actual de *acumulador* y de (**first** *uldx*).

Claramente, *invierte* no puede olvidar nada y revierte el orden de los ítems en la lista. De ahí que se pudiera sólo desear acumular todos los ítems que encuentre *rev*. Lo que significa

1. Que *acumulador* representa una lista, y
2. Que representa todos aquellos ítems en *uldx0* que preceden el argumento *uldx* de *rev*.

Para la segunda parte del análisis, es importante poder distinguir el argumento original, *uldx0*, del actual, *uldx*.

Ya que se conoce de forma aproximada el propósito del acumulador, se considera qué debería ser el primer valor y que debería hacer para la recursión. Cuando se aplica *rev* en el cuerpo de la **expresión local**, recibe *uldx0*, lo cual significa que no ha encontrado alguno de sus ítems. El valor inicial para *acumulador* es **empty**. Cuando recurre *rev*, ha encontrado un ítem extra: (**first** uldx). Para recordarlo, se utiliza **cons** en el valor actual del acumulador.

A continuación la función mejorada:

```
;; invierte : (listade X)  ->  (listade X)
;; construir el reverso de uldx
(define (invierte uldx0)
   (local (;; acumulador es la lista invertida de todos aquellos ítems
                ;; en uldx0 que preceden uldx
                (define (rev uldx acumulador)
                 (cond
                   [(empty? uldx) ...]
                   [else
                       ... (rev (rest uldx) (cons (first uldx) acumulador))
                       ...])))
          (rev uldx0 empty)))
```

Un inspección cuidadosa revela que acumulador no sólo son los ítems en *uldx0* que preceden sino una lista de aquellos ítems en orden inverso.

Una vez se ha decidido qué conocimiento el acumulador mantiene y como lo mantiene, ahora se puede tratar la pregunta de cómo utilizar dicho conocimiento para la función.

En el caso de invierte, la respuesta es casi obvia. Si acumulador es la lista de todos los ítems en uldx0 que preceden uldx en orden inverso, entonces, si uldx es empty, acumulador representa el reverso de uldx0. Dicho de otra manera: si uldx es empty, la respuesta de rev es acumulador, la cual es la respuesta que se desea en ambos casos:

```
;; invierte : (listade X)  ->  (listade X)
;; construir la inversa de uldx
(define (invierte uldx0)
  (local (;; acumulador es la lista inversa de todos aquellos ítems
               ;; en uldx0 que preceden uldx
                 (define (rev uldx acumulador)
                  (cond
                    [(empty? uldx) acumulador]
                       [else
                     (rev (rest uldx) (cons (first uldx) acumulador))])))
```

(*rev uldx0* **empty**)))

4.3. Transformando funciones en estilo-acumulador

La parte más compleja de la receta de diseño es el requerimiento de formular un invariante acumulador. Sin éste no se pueden producir funciones operantes estilo-acumulador. Debido a que la formulación de invariantes es claramente un arte que requiere mucha práctica, veamos un ejemplo par la función suma.

```
;; suma : (listade número)  ->  número
;; calcular la suma de los números en uldn
;; recursion estructural
(define (suma uldn)
  (cond
    [(empty? uldn) 0]
    [else (+ (first uldn) (suma (rest uldn)))]))
```

A continuación el primer paso hacia una versión acumulador:

```
;; suma : (listade número)  ->  número
;; calcular la suma de los números en uldn0
(define (suma uldn0)
  (local (;; acumulador ...
          (define (suma-a uldn acumulador)
            (cond
                [(empty? uldn) ...]
              [else
                    ... (suma-a (rest uldn) ... (first uldn) ... acumulador)
               ... ])))
    (suma-a uldn0 ...)))
```

Como se indicó en el primer paso, se debe poner el formato para *suma-a* en una definición **local**, agregar un parámetro acumulador, y renombrar el parámetro de *suma*.

La meta es desarrollar un invariante acumulador para *suma*. Para hacerlo, se debe considerar cómo procede *suma* y cuál es la meta del proceso. Igual que *rev*, *suma-a* procesa los números en la lista uno a uno. La meta es sumar estos números. Lo que sugiere que *acumulador* representa la suma de los números vistos hasta ahora:

```
...
  (local (;; acumulador es la suma de los números que preceden
          ;; aquellos en uldn en uldn0
            (define (suma-a uldn acumulador)
             (cond
                  [(empty? uldn) ...]
                  [else
```

```
            ... (suma-a (rest uldn) (+ (first uldn) acumulador))
            ... ])))
    (suma-a uldn0 0)))
```

Cuando se aplica *suma-a* se debe emplear 0 como el valor de *acumulador*, debido a que no ha procesado ninguno de los números en *uldn* aún. Para la segunda cláusula, se debe agregar (**first** *uldn*) a **acumulador** de modo que el invariante represente a la aplicación de la función.

Dado un invariante preciso, el resto es nuevamente sencillo. Si *uldn* es **empty**, *suma-a* devuelve *acumulador* debido a que representa ahora la suma de todos los números en *uldn*. La definición final de la versión estilo-acumulador de *suma*.

```
;; suma : (listade número)  ->  número
;; calcular la suma de los números en uldn0
(define (suma uldn0)
  (local (;; acumulador es la suma de los números que preceden
          ;; aquellos en uldn en uldn0
    (define (suma-a uldn acumulador)
      (cond
        [(empty? uldn) acumulador]
        [else (suma-a (rest uldn) (+ (first uldn) acumulador))])))
    (suma-a uldn0 0)))
```

La función suma estilo-acumulador

```
  (suma (list 10.23 4.50 5.27))
= (+ 10.23 (suma (list 4.50 5.27)))
= (+ 10.23 (+ 4.50 (suma (list 5.27))))
= (+ 10.23 (+ 4.50 (+ 5.27 (suma empty))))
= (+ 10.23 (+ 4.50 (+ 5.27 0)))
= (+ 10.23 (+ 4.50 5.27))
= (+ 10.23 9.77)
= 20.0
```

```
  (suma (list 10.23 4.50 5.27))
= (suma-a (list 10.23 4.50 5.27) 0)
= (suma-a (list 4.50 5.27) 10.23)
= (suma-a (list 5.27) 14.73)
= (suma-a empty 20.0)
= 20.0
```

En el lado izquierdo, la función recursiva simple desciende en la lista de números hasta el final y realiza operaciones de suma al final. A la derecha, se ve cómo la versión estilo-acumulador suma los números conforme avanza. Adicionalmente, se ve que para cada aplicación de *suma-a* el invariante se mantiene con respecto a la aplicación de *suma*. Cuando *suma-a* es procesada finalmente a empty, el acumulador es resultado final, y *suma-a* lo devuelve.

Otro ejemplo, la función factorial:

```
;; ! : N -> N
;; calcular n · (n - 1) · ... · 2 · 1
;; recursión estructural
 (define (! n)
  (cond
    [(zero? n) 1]
    [else (* n (! (sub1 n)))]))
```

Mientras que ***relativa-a-absoluta*** e ***invierte*** procesan listas, la función factorial trabaja con números naturales. Esto se da para funciones que procesan **N**.

Procedemos igual que antes creando una definición **local** de ***!:***

```
;; ! : N -> N
;; calcular n · (n - 1) · ... · 2 · 1
(define (! n0)
  (local (;; acumulador ...
          (define (!-a n acumulador)
            (cond
              [(zero? n) ...]
              [else
                 ... (!-a (sub1 n) ... n ... acumulador) ...])))
    (!-a n0 ...)))
```

Este bosquejo sugiere que si *!* es aplicado al número natural *n*, *!-a* procesa *n*, entonces *n* - 1, *n* - 2, y así sucesivamente hasta que llegue a 0. Dado que el objetivo es multiplicar estos números, el acumulador debe ser el producto de todos los números encontrados por *!-a*:

```
...
  (local (;; acumulador es el producto de los números naturales entre
          ;; n0 (inclusive) y n (exclusivo)
          (define (!-a n acumulador)
            (cond
              [(zero? n) ...]
              [else
                 ... (!-a (sub1 n) (* n acumulador)) ...])))
    (!-a n0 1)))
```

Para hacer verdadero el invariante al principio, debemos usar 1 para el acumulador. Cuando recurre *!-a*, debemos multiplicar el valor actual del acumulador con *n* para reestablecer el invariante.

Del propósito del acumulador *!-a*, vemos que si n es 0, el acumulador es el producto de n, . . ., 1. Esto es el resultado deseado. Entonces, como *suma*, *!-a* devuelve *acumulador* en el primer caso y simplemente recurre en el segundo.

```
  (! 3)                                  (! 3)
= (* 3 (! 2))                          = (!-a 3 1)
= (* 3 (* 2 (! 1)))                    = (!-a 2 3)
= (* 3 (* 2 (* 1 (! 0))))              = (!-a 1 6)
= (* 3 (* 2 (* 1 1)))                  = (!-a 0 6)
= (* 3 (* 2 1))                        = 6
= (* 3 2)
= 6
```

La columna izquierda muestra cómo funciona la versión original y la columna derecha cómo procede la función estilo-acumulador. Ambas recorren el número natural hasta que llegan a 0, pero mientras la versión original sólo realiza multiplicaciones, la nueva multiplica los números mientras son procesados. Aunado a ello, la columna derecha muestra cómo la nueva función factorial mantiene el invariante acumulador. Para cada aplicación, el acumulador es el producto de 3 a *n*, donde *n* es el primer argumento de ***!-a***.

Ejercicio 12: Desarrollar una versión *estilo-acumulador* de *producto*, la función que computa el producto de una lista de números. Mostrar las etapas que expliquen lo que representa el acumulador.