Lenguajes de Programación, 2022-1
Nota de clase 06: Recursión

Karla Ramírez Pulido Manuel Soto Romero

29 de noviembre de 2021
Facultad de Ciencias UNAM

Como recordarás de tu curso de Estructuras Discretas, la recursión es un método de definición en el cual el concepto definido figura en la definición misma. Este método está presente en algunos lenguajes de programación como JAVA, HASKELL o RACKET y permite definir estructuras o tipos de datos como son las listas o los árboles binarios. En esta sección se presentan los conceptos necesarios para escribir un intérprete recursivo y además tomar en cuenta el desempeño (en espacio y tiempo) de la definición de este tipo de funciones para poder realizar optimizaciones.

6.1. Registros de activación

Para que una función recursiva sea válida y no genere ciclos infinitos de evaluación, debe constar de dos partes:

- Un conjunto de casos base, también llamados cláusulas de escape.
- Un conjunto de reglas recursivas definidas en términos de valores previamente definidos.

Por ejemplo, la función factorial, se define en RACKET como se muestra en el Código 1.

```
1 ;; fact: number → number
2 (define (fact n)
3    (if (zero? n)
4        1
5        (* n (fact (sub1 n)))))
```

Código 1: Factorial de un número

Uno de los principales problemas al usar funciones recursivas es el consumo excesivo de memoria. Internamente, en la pila de llamadas a función se crea un *registro de activación* que contiene información sobre la función a ejecutar. Por cada llamada a función que ocurra durante la ejecución del programa, se genera un nuevo registro y éste no se libera de la pila de llamadas a función hasta que termina el proceso correspondiente.

Por ejemplo, para la llamada `(fact 3)`, se generan cuatro registros de activación, (ver Figura 1), cada espacio en memoria se libera conforme la función genera resultados para cada llamada (ver Figura 2). Cada registro de activación almacena:

- El nombre de la función.

- El argumento con el que fue llamada la función.
- El cuerpo de la función.
- El valor de regreso de la función

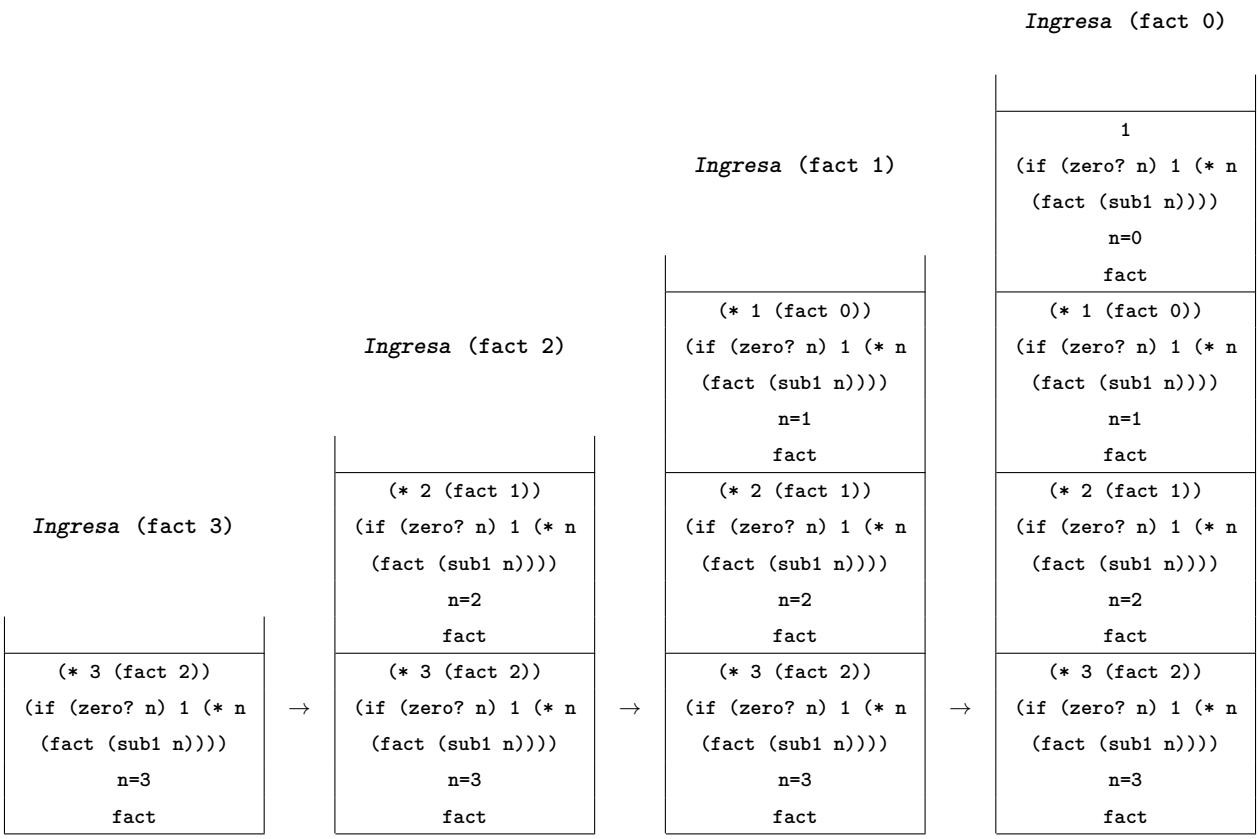

Figura 1: Ingreso de registros de activación

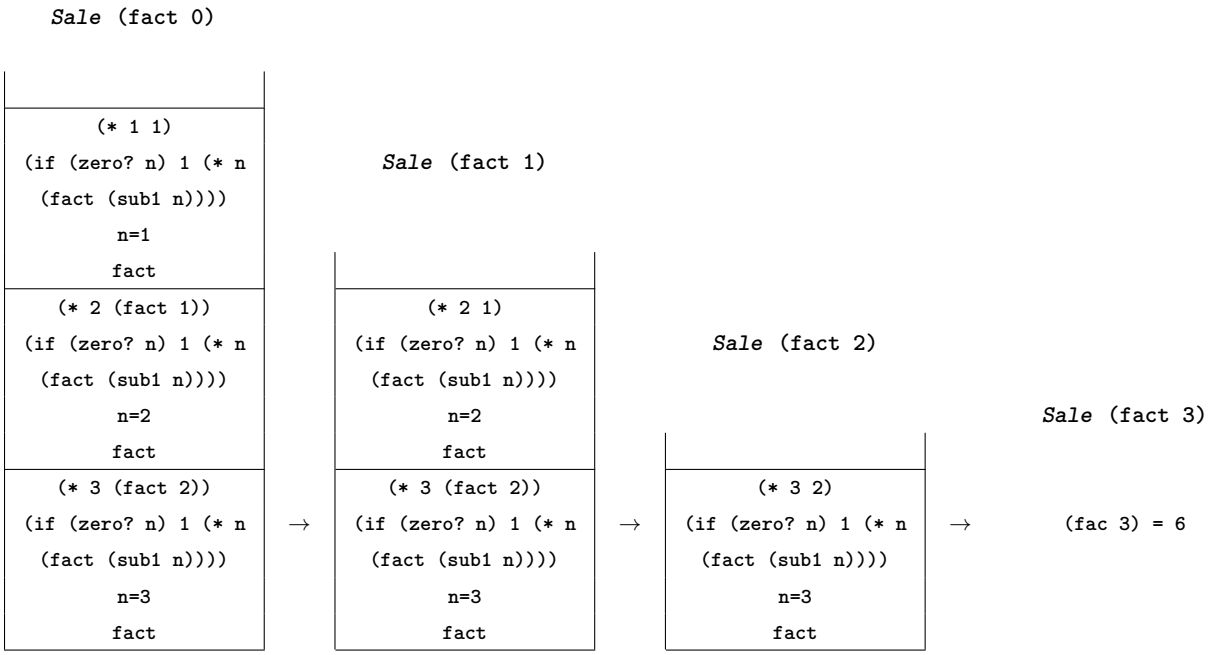

Figura 2: Salida de registros de activación

La Figura 3 muestra el árbol de llamadas para **(fact 3)**. Se aprecia la dependencia de llamadas pendientes en los registros de activación de las Figuras 1 y 2.

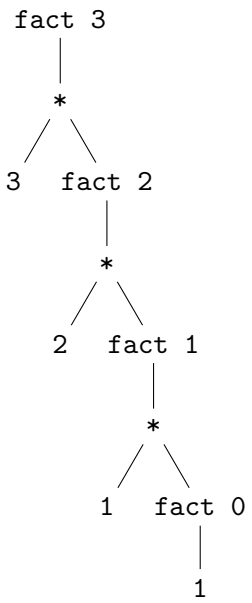

Figura 3: Árbol de llamadas de **(fact 3)**

De esta forma, se aprecia que mientras mayor sea la entrada de la función factorial, se crearán un mayor número de regristros de activación y (como se tiene espacio de memoria finito), ocurrirá un *desbordamiento de pila*[1].

[1]Del inglés *Stack Overflow*

6.2. Recursión de cola

En el ejemplo de la sección anterior, se crearon cuatro registros debido a las operaciones pendientes de cada resultado.

```
;; fact: number → number
(define (fact n)
   (if (zero? n)
       1
       (* n (fact (sub1 n)))))
```

Por ejemplo, la llamada `(* 3 (fact 2))` queda pendiente mientras no se conozca el valor de evaluar la llamada a función `(fact 2)`, con lo cual el registro de activación se quedará en espera hasta que pueda aplicar la operación correspondiente.

Una forma de mejorar el rendimiento en espacio es eliminando las posibles llamadas pendientes dejando que el resultado de una función recursiva sea expresado exclusivamente mediante la llamada recursiva a la función sin involucrar operaciones que dejen en espera los registros. Las operaciones pendientes son realizadas a través de los parámetros de la función y se modifican llamada tras llamada hasta llegar a un caso base. Este mecanismo es conocido como *recursión de cola*[2].

Por ejemplo, se puede modificar la función factorial para que use esta técnica como se muestra en el Código 2. Se agrega un parámetro extra a la función original que lleve el control de las operaciones al cual se le conoce como *acumulador*. Las líneas 2 a 4 muestran la llamada original a la función y llaman a la versión de cola de la función con el acumulador iniciado en 1.

Las líneas 6 a 9 definen la función factorial optimizada con recursión de cola. El caso base de la función, verifica si el número recibido es cero, en cuyo caso se devuelve el valor actual del acumulador. En caso contrario, se llama a la función recursivamente, quitando uno al parámetro recibido y a diferencia de la versión original de la función, se realiza un producto dentro del acumulador, mismo que se irá cargando hasta llegar a un caso base.

```
1 ;; fact: number → number
2 (define (fact n)
3    (fact-tail n 1))
4
5 ;; fact-tail: number number → number
6 (define (fac-tail n acc)
7    (if (zero? n)
8        acc
9        (fact-tail (sub1 n) (* n acc))))
```

Código 2: Factorial de un número

De esta forma la creación y liberación de registros de activación se comporta como se muestra en la Figura 4. Observa que aunque, en este caso, se crean el mismo número de registros, se ocupa únicamente uno por cada llamada de cola.

[2]Del inglés *Tail Recursion*

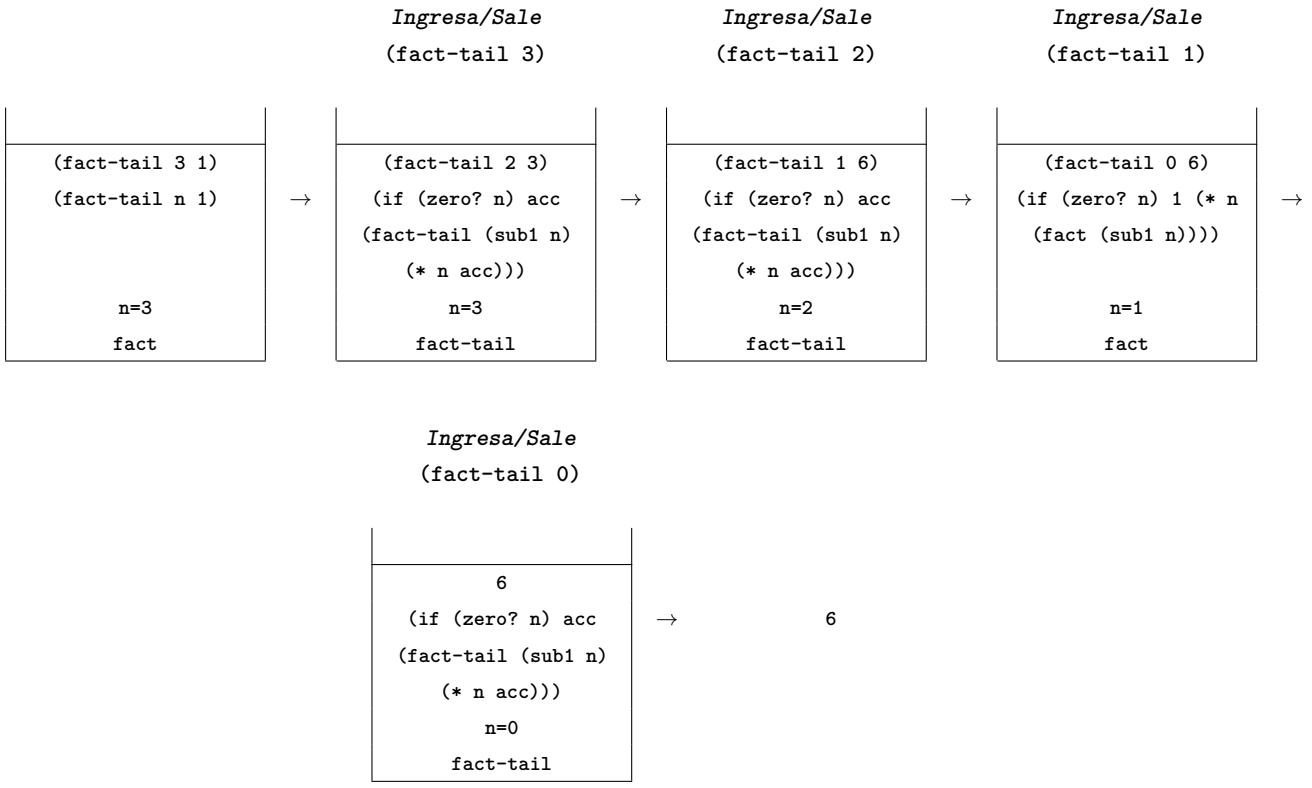

Figura 4: Registros de activación para `fact-tail`

Es importante notar que aunque se siguen generando varios registros, únicamente queda activo uno. Este método de definir funciones recursivas es un mecanismo de optimización que usan algunos intérpretes y compiladores para mejorar el rendimiento de las funciones recursivas.

A continuación se presentan algunos ejemplos de transformación de funciones recursivas a su respectiva versión usando recursión de cola. Se recomienda en general, seguir los siguientes pasos, aunque puede haber excepciones:

- **Reescribir la función.** Se reescribirá agregando parámetros para el acumulador, en un principio, es necesario, agregar tantos acumuladores como llamadas pendientes tenga la función original.
- **Reescribir el caso base.** Dado que el acumulador va guardando el resultado parcialmente, cuando se llegue a un caso base, es necesario devolver el resultado del acumulador.
- **Reescribir la llamada recursiva.** Se reescribirá quitando la operación que ocasiona llamadas pendientes y realizando la misma con ayuda de el o los acumuladores.
- **Reescribir la función principal.** Dado que el usuario de la función no debe interactuar con esta nueva función. Debe haber una función principal que llame a la versión optimizada, asignándole un valor inicial al acumulador. El valor inicial debe ser, en un principio, el valor que regresaba en el caso base la función original.

Ejemplo 6.1. Definir la función `map` tal que `(map f lst)` construye una nueva lista aplicando la función `f` a cada elemento de la lista `lst`. Se define la función mediante la técnica de recursión de cola.

Versión original

```
;; map: procedure (listof any) → (listof any)
(define (map f lst)
   (if (empty? lst)
       empty
       (cons (f (car lst)) (map f (cdr lst)))))
```

Código 3: Función map

Ejecución original

```
(map add1 '(1 2 3)) =
(cons 2 (map add1 '(2 3)) =
(cons 2 (cons 3 (map add1 '(3)))) =
(cons 2 (cons 3 (cons 4 (mapp add1 empty)))) =
(cons 2 (cons 3 (cons 4 empty))) =
'(2 3 4)
```

Versión con recursión de cola

```
;; map: procedure (listof any) → (listof any)
(define (map f lst)
   (map-tail f lst empty))

;; map-tail: procedure (listof any) (listof any) → (listof any)
(define (map-tail f lst acc)
   (if (empty? lst)
       acc
       (map-tail f (cdr lst) (append acc (list (f (car lst)))))))
```

Código 4: Función map de cola

Ejecución con recursión de cola

```
(map add1 '(1 2 3)) =
(map-tail add1 '(1 2 3) empty) =
(map-tail add1 '(2 3) '(2) =
(map-tail add1 '(3) '(2 3)) =
(map-tail add1 '() '(2 3 4)) =
'(2 3 4)
```

¿Cuántos registros de activación son creados con la función **`add1`** *y la lista* **`'(1 2 3 4 5)`** *con la versión de* **`map`** *original y con la versión optimizada con recursión de cola?* □

Ejemplo 6.2. Definir la función **`fibo`** tal que **`(fibo n)`** obtiene el **`fibonacci`** del número **`n`**. Se define la función mediante la técnica de recursión de cola.

Versión original

```
;; fibo: number → number
(define (fibo n)
   (if (< n 2)
       1
       (+ (fibo (sub1 n)) (fibo (- n 2)))))
```

Código 5: Función fibo

Ejecución original

```
(fibo 3) =
(+ (fibo 2) (fibo 1)) =
(+ (+ (fibo 1) (fibo 0)) 1) =
(+ (+ 1 1) 1) =
(+ 2 1) = 3
```

Versión con recursión de cola

```
;; fibo: number → number
(define (fibo n)
   (fibo-cola n 1 1))

;; fibo-tail: number number number → number
(define (fibo-tail n acc1 acc2)
   (if (< n 2)
       acc1
       (fibo-tail (sub1 n) (+ acc1 acc2) acc1)))
```

Código 6: Función `fibo` de cola

Ejecución con recursión de cola

```
(fibo 3) =
(fibo-tail 3 1 1) =
(fibo-tail 2 2 1) =
(fibo-tail 1 3 2) =
3
```

¿Cuántos registros de activación son creados con el número `5` *con la versión de* `fibo` *original y con la versión optimizada con recursión de cola?* □

6.3. El Lenguaje RCFWAE

Existen diversas formas de implementar recursión, a continuación se revisa una forma mediante la definición de ambientes cíclicos que permitan la autoreferencia de expresiones recursivas. Para ejemplificar esto, se agrega la primitiva `rec` al lenguaje como se muestra en la siguiente gramática del lenguaje RCFWAE:

```
<expr> ::= <id>
         | <num>
         | {+ <expr> <expr>}
         | {- <expr> <expr>}
         | {* <expr> <expr>}
         | {if0 <expr> <expr> <expr>}
         | {with {<id> <expr>} <expr>}
         | {rec {<id> <expr>} <expr>}
         | {fun {<id>} <expr>}
         | {<expr> <expr>}
```

Es importante notar el parecido sintáctico entre las primitivas **`with`** y **`rec`**. Su parecido es equivalente al de las primitivas **`let`** y **`letrec`** de RACKET. Adicionalmente se añade la operación de multiplicación con fines prácticos.

La implementación consiste en apoyarse en las estructuras de datos mutables del lenguaje de programación anfitrión, en este caso, cajas[3] de RACKET. La idea consiste en modificar los ambientes de evaluación para que acepten definiciones recursivas. La idea detrás de este proceso, consiste en aplicar una simple regla de tres pasos[3]:

1. Nombrar un contenedor vacío.
2. Mutar el contenedor para que su contenido sea él mismo.
3. Para obtenerlo usar el nombre previamente definido.

Ejemplo 6.3. Dada la expresión del Código 7 que calcula el factorial del número 5.

```
1 (let ([fact (λ(n)
2            (if (zero? n) 1 (* n (fact (sub1 n))))))])
3    (fact 5))
```

Código 7: Factorial de 5

La expresión dada en el Código 7, genera un error al ejecutarse en RACKET, pues al intentar aplicar la definición del factorial que se encuentra en el cuerpo de la función, se tiene una variable libre. Aplicando la regla de tres pasos es posible arreglar este error:

1. Nombrar un contenedor vacío

```
(let ([fact (box 1729)])
   ...)
```

2. Mutar el contenedor para que su contenido sea el mismo.

```
(let ([fact (box 1729)])
   (begin
      (set-box! fact
         (λ(n)
            (if (zero? n) 1 (* n (fact (sub1 n))))))
   ...))
```

[3] *Boxes*

3. Para obtenerlo usar el nombre previamente definido.

```
(let ([fact (box 1729)])
   (begin
      (set-box! fact
         (λ(n)
            (if (zero? n) 1 (* n ((unbox fact) (sub1 n))))))
   ((unbox fact) 5)))
```

Al ejecutar la expresión anterior en RACKET, se puede apreciar la recursión. □

Como se puede observar, puede ser es menos tedioso para algún programador usar directamente la primitiva **letrec**. Ésta técnica para implementar recursión en RCFWAE, aplicando la regla de tres pasos sobre expresiones **rec**. Para ello, se muestra en el Código 8 la definición del tipo de dato **RCFAE** que representa la sintaxis abstracta de la primitiva **rec**.

```
1 (define-type RCFAE
2    [id  (i symbol?)]
3    [num (n number?)]
4    [add (izq RCFAE?) (der RCFAE?)]
5    [sub (izq RCFAE?) (der RCFAE?)]
6    [plus (izq RCFAE?) (der RCFAE?)]
7    [if0 (test-expr RCFAE?) (then-expr RCFAE?) (else-expr RCFAE?)]
8    [rec (id symbol?) (value RCFAE?) (body RCFAE?)])
```

Código 8: RCFAE

Se deben modificar los ambientes de evaluación, para distinguir aquellos que almacenan una expresión recursiva. Es en este punto donde se incluyen las cajas en el intérprete. La nueva definición del tipo **Env** se muestra en el Código 9. Las líneas 2 a 3 definen un predicado que verifica que el ambiente en efecto contenga una caja y que ésta en efecto almacene una expresión **RCFAE-Value**.

```
1 ;; boxed-RCFAE-Value?: any → boolean
2 (define (boxed-RCFAE-Value? v)
3    (and (box? v) (RCFAE-Value? (unbox v))))
4
5 (define-type Env
6    [mtSub]
7    [aSub    (id symbol?) (value RCFAE-Value?) (rest-env Env?)]
8    [aRecSub (id symbol?) (value boxed-RCFAE-Value?) (rest-env Env?)])
```

Código 9: Ambiente recursivo

La interpretación consiste en evaluar el cuerpo de **rec** en el ambiente recursivo correspondiente.

Para construir el ambiente recursivo, se usa la regla de tres pasos. El Código 10 muestra estas definiciones.

```
1  ;; cyclically-bind-and-interp: symbol RCFAE Env → Env
2  (define (cyclically-bind-and-interp id value env)
3     (let* ([contenedor (box (numV 1729))] ;; Paso 1
4            [ambiente   (aRecSub id contenedor env)]
5            [valor      (interp value ambiente)])
6        (begin
7           (set-box! contenedor valor) ;; Paso 2
8           ambiente)))
9
10 ;; interp: RCFAE Env → RCFAE-Value
11 (define (interp expr env)
12    (type-case expr RCFAE
13       ...
14       [rec (id value body)
15          (interp body (cyclically-bind-and-interp id value env))]))
16
17 ;; lookup: symbol Env → RCFAE-Value
18 (define (lookup sub-id env)
19    (type-case env Env
20       ...
21       [aRecSub (id value rest-env)
22          (if (symbol=? id sub-id)
23              (ubox-value) ;; Paso 3
24              (lookup sub-id rest-env))])
```

Código 10: Interpretación de rec

En el Listado de código 10:

- Las líneas 2 a 8 definen la función **cyclcally-bind-and-interp** que construye un ambiente recursivo dado un identificador, un valor y el resto del ambiente. Se crea un contenedor vacío (Paso 1), se asocia el contenedor al nuevo ambiente, se calcula el valor a guardar en el ambiente y finalmente se asocia ese valor al contenedor (Paso 2) y se regresa el nuevo ambiente.
- La interpretación de **rec** se muestra en la línea 15 y se realiza sobre el nuevo ambiente construido.
- Las líneas 18 a 24 muestran la definición de la función **lookup**. Para obtener el valor de un ambiente recursivo debe abrirse el contenedor correspondiente (Paso 3).

Se deja como ejercicio al lector probar que la siguiente expresión se puede ejecutar en este intérprete:

```
{rec {f {fun {n} {if0 n 1 {* n {f {- n 1}}}}}}
   {f 5}}
```

6.4. Ejercicios

6.4.1. Registros de Activación / Recursión de Cola

1. Define las funciones solicitadas en Racket. Para cada una:

- Muestra el número de registros de activación generados para la llamada que se indica.
- Transfórmala usando recursión de cola.

(a) Define una función recursiva que calcule la suma de los primeros *n* números naturales. Muestra el número de registros generado por la entrada `n = 5`.

(b) Define una función recursiva que calcule la potencia n^m. Muestra el número de registros generado por las entradas `n = 2, m = 3`.

(c) Define una función recursiva que encuentre la longitud de un número, es decir, el número de dígitos que tiene. Muestra el número de registros generado por la entrada `n = 1729`.

(d) Define una función recursiva que calcule la suma de los dígitos de un número. Muestra el número de registros generado por la entrada `n = 1729`.

(e) Define una función recursiva que dada una lista de números sume todos sus elementos. Muestra el número de registros generado por la entrada `l = '(1 7 2 9)`.

(f) Define una función recursiva que dada una lista de números multiplique todos sus elementos. Muestra el número de registros generado por la entrada `l = '(1 7 2 9)`.

(g) Define una función recursiva que dada una lista obtenga la longitud de la misma. Muestra el número de registros generado por la entrada `l = '(1 7 2 9)`.

(h) Define una función recursiva que dada una lista de números obtiene el elemento más pequeño de la misma. Muestra el número de registros generado por la entrada `l = '(1 7 2 9)`.

(i) Define una función recursiva que dado un número *n* y una lista, toma los primeros *n* elementos de la lista. Muestra el número de registros generado por las entradas `n = 2, l = '(1 7 2 9)`.

6.4.2. El Lenguaje RCFWAE

1. Dada la siguiente expresión, modifícala para que use recursión correctamente usando la técnica de los tres pasos:

```
(let ([mapea (λ(f l)
               (if (empty? l) l (cons (f (car l)) (mapea f (cdr l)))))])
  (map add1 '(1 2 3)))
```

2. Dada la sintaxis concreta del lenguaje RCFWAE y su sintaxis abstracta:

```
(define-type ASA
  [id  (i symbol?)]
  [num (n number?)]
  [add (i ASA?) (d ASA?)]
  [sub (i ASA?) (d ASA?)]
  [if0 (cond-expr ASA?) (then-expr ASA?) (else-expr ASA?)]
  [with (id symbol?) (value ASA?) (body ASA?)]
  [rec (id symbol?) (value ASA?) (body ASA?)]
  [fun (param symbol?) (body ASA?)]
  [app (fun-expr ASA?) (arg ASA?)])
```

Construye una expresión para calcular la suma de los primeros *n* números naturales y muestra su sintaxis abstracta.

Referencias

[1] Favio E. Miranda, Elisa Viso, *Matemáticas Discretas*, Segunda Edición, Las Prensas de Ciencias, 2016.

[2] Matthew Butterick, *Beautiful Racket: An Introduction to language-oriented programming using Racket*, Versión 1.4. [En línea: `https://beautifulracket.com/`]

[3] Shriram Krishnamurthi, *Programming Languages: Application and Interpretation*, Primera edición, 2007. [En línea: `https://plai.org/`].