1 Definición de funciones recurrentes

En el paradigma de *programación funcional* el principal bloque constructivo es la función, en la que hay que distinguir: su definición (el código del programa) que se realiza invocando otras funciones (composición de funciones), de su evaluación (la ejecución del programa). Además, los datos que se manejan son simbólicos (*S-expresiones*), una notación en forma de texto basada en estructuras anidadas. Esta misma notación se utiliza también en la definición e invocación de una función.

En los lenguajes puramente funcionales, la definición de funciones que al evaluarse requieran realizar un cálculo o proceso repetitivo ha de llevarse a cabo, necesariamente, de alguna de las dos formas siguientes:

- Dando una definición recurrente o recursiva.
- Utilizando ciertas funciones de orden superior (FOS); es decir, funciones que reciben como argumento otra función o en las que el resultado de la evaluación es a su vez una función.

Nos ocuparemos aquí de las funciones recursivas, distinguiendo dos casos:

- *Definición de funciones recursivas sobre listas*. Su evaluación permite resolver problemas que sólo afectan al nivel superior de una S-expresión que es una lista. En este caso, es irrelevante que un elemento de una lista sea a su vez otra lista (recuérdese que una S-expresión es una estructura multinivel). Por ejemplo, definir una función que retorne el número de elementos de una lista.
- *Definición de funciones recursivas sobre S-expresiones*. Aunque las expresiones son estructuras anidadas, en la parte de prácticas sólo se van a manipular listas anidadas o multinivel; es decir, todos los problemas se plantearan sobre el subconjunto de S-expresiones que son átomos o listas. Así, a diferencia del caso previo, aquí sí sería relevante que un elemento del nivel superior de la lista fuese a su vez otra lista y así sucesivamente. Por ejemplo, en la definición de una función que retorne el número de átomos de una S-expresión. Es evidente que la definición de tal función no permite obviar que un elemento del nivel superior de la lista sea a su vez otra lista, ya que esta última podrá contener desde cero átomos (si es la lista vacía) a cualquier otro número de éstos.

En ambos casos, se utilizará la definición recurrente del dominio para establecer la definición de una función recursiva.

1.1 Funciones recursivas sobre listas

Se verá a continuación como definir funciones recursivas que permitan resolver un problema cuando la información (o dato) a tratar es una lista en la que resulta irrelevante que sus elementos sean a su vez otras listas. Si este es el caso, una lista es una secuencia de elementos que representaremos entre paréntesis como: $(e_0\ e_1 \ldots e_{n-1})$, con independencia de que cada uno de los elementos e_i sea un átomo, o sea, a su vez, otra lista.

Las listas se pueden definir recurrentemente de la forma siguiente:

a) *Base*: una lista que no contiene elementos, la *lista vacía*: ( )
b) *Recurrencia*: si x es un elemento e y la lista $(e_0\ e_1 \ldots e_{n-1})$, entonces $cons(x, y)$ también es una lista, la lista $(x \cdot y) = (x\ e_0\ e_1 \ldots e_{n-1})$. Obsérvese, que $x = car(cons(x, y))$ e $y = cdr(cons(x, y))$

El esquema de la definición de una función recursiva f sobre una lista l (Fig. 1), o *análisis por casos*, se establecerá utilizando la notación funcional habitual, en lugar de una S-expresión, y se corresponderá con la definición recurrente de su dominio (el indicado previamente):

1. *Base*: el resultado de la función para la lista vacía: ¿ $f(())$?
2. *Recurrencia*: l no es la lista vacía; es decir, $l = cons(car(l), cdr(l))$
    a. *Hipótesis*: se supone conocido $f\big(cdr(l)\big) = H$
    b. *Tesis*: obtener $f(l)$ a partir de la hipótesis en combinación con el elemento $car(l)$ de la lista l, ya que éste no forma parte del argumento de la hipótesis

Fig. 1: Esquema de la definición de una función recursiva sobre una lista

Tanto este análisis por casos, como el que se verá en el siguiente apartado, se establecerá mediante una notación lo más estándar posible e independiente del lenguaje de programación en el que, finalmente, se codificará la definición de la función (en nuestro caso en *Racket*). Así, todas las invocaciones a funciones se declararán con la notación habitual: $f(x_0, x_1, \dots)$, el nombre de la función y entre paréntesis y separados por el carácter ',' la lista de argumentos requeridos.

En las prácticas, el *análisis por casos* **siempre deberá incluirse** como comentarios, antes de dar la definición de una función recursiva en el lenguaje de programación utilizado (*Racket*).

En el caso de que la definición de la función $f(x_0, x_1, \dots)$ dependa de los valores que tomen uno o más de sus argumentos, lo habitual es indicar las distintas definiciones para cada caso (condición que tienen que cumplir sus argumentos) de la forma siguiente:

$$f(x_0, x_1, \dots) = \begin{cases} definición\text{-}1 & \text{si } condición\text{-}1 \\ definición\text{-}2 & \text{si } condición\text{-}2 \\ & \dots \end{cases}$$

Para este tipo de funciones se utilizará una notación similar, pero se utilizarán *funciones condicionales* que se pueden anidar y que expresaremos de forma independiente del lenguaje de programación:

$f(x_0, x_1, \dots) =$ **si** $condición\text{-}1$
entonces $definición\text{-}1$
si_no
    si $condición\text{-}2$
    entonces $definición\text{-}2$
    si_no …

Ejemplo 1

Definir la función $my\text{-}length(l)$ que retorna el número de elementos de la lista dada.

1. *Base*: el resultado de la función para la lista vacía; $my\text{-}length(()) = 0$
2. *Recurrencia*: l no es la lista vacía; es decir, $l = cons(car(l), cdr(l))$
    a. *Hipótesis*: se supone conocido $my\text{-}length\big(cdr(l)\big) = H$
    b. *Tesis*: $my\text{-}length(l) = H + 1$

Obsérvese, que si l es una lista cualquiera, $(e_0\, e_1 e_2 \dots e_{n-1})$, entonces $car(l) = e_0$ y $cdr(l) = (e_1 e_2 \dots e_{n-1})$. Como $my\text{-}length\big(cdr(l)\big) = H$, la lista l tiene un elemento más $(H + 1)$, el elemento $car(l) = e_0$.

En Racket, la definición es exactamente la misma, con la única diferencia de que, en lugar de utilizar la notación funcional habitual: nombre de función y entre paréntesis y separados por el carácter ',' su lista de argumentos, $f(x_0, x_1, \ldots, x_{n-1})$, debe utilizarse la S-expresión equivalente $(f\ x_0\ x_1\ \ldots\ x_{n-1})$:

```
(define (my-length l)
  (if (null? l)
      0
      (+ (my-length (cdr l)) 1)))

(my-length '(a (b c) d)) ⇒ 3
```

Ejemplo 2

Definir la función $remove\text{-}all(x, l)$ que retorna una lista copia de l, pero que no contiene elemento x alguno.

Tal y como ocurre en este caso, puede haber funciones de más de un argumento que se podrán definir de forma totalmente análoga a las funciones que reciben como argumento sólo una lista. Esto es así cuando, de los múltiples argumentos que puede recibir la función, el problema es tal que únicamente afecta de forma relevante a uno de ellos y éste es una lista. Para este ejemplo, el argumento relevante es el segundo, que se corresponde con el parámetro l de la función y es una **lista**. Sin embargo, el propio enunciado, da al primer argumento de la función el tratamiento de un posible **elemento** de la lista dada como segundo argumento.

El *análisis por casos* se realiza exactamente igual que para el ejemplo 1, pero teniendo en cuenta que la lista sobre el que ha de realizarse éste es el segundo argumento:

1. *Base*: el resultado de la función para la lista vacía; $remove\text{-}all(x, ()) = ()$
2. *Recurrencia*: l no es la lista vacía; es decir, $l = cons(car(l), cdr(l))$
   a. *Hipótesis*: se supone conocido $remove\text{-}all\big(x, cdr(l)\big) = H$
   b. *Tesis*: $remove\text{-}all(x, l) =$ **si** $car(l) = x$ **entonces** H **si_no** $cons(car(l), H)$

En Racket:

```
(define (remove-all x l)
  (cond [(null? l) l]
        [(equal? (car l) x) (remove-all x (cdr l))]
        [else (cons (car l) (remove-all x (cdr l)))]))

(remove-all '(a) '(b (a) (c (a)) (a) d)) ⇒ (b (c (a)) d)
```

Ejemplo 3

Definir la función $my\text{-}append(l1, l2)$ que retorna la lista concatenación de las dos listas que se reciben como argumento.

Si la función a definir tiene dos argumentos que son listas de igual relevancia para el problema, tal y como ocurre en este caso, el *análisis por casos* se hace de forma análoga a como se ha realizado en los ejemplos previos, aunque ahora hay que considerar, al menos a priori, las posibles combinaciones de casos base e hipótesis de recurrencia:

Base	**Hipótesis de recurrencia**
$l1$ es la lista vacía, $my\text{-}append(),l2) = l2$	Se conoce $my\text{-}append(cdr(l1),l2) = H1$
$l2$ es la lista vacía, $my\text{-}append(l1,()) = l1$	Se conoce $my\text{-}append\big(l1,cdr(l2)\big) = H2$
$l1$ y $l2$ son listas vacías, $my\text{-}append((),()) = ()$	Se conoce $my\text{-}append\big(cdr(l1),cdr(l2)\big) = H3$

Para una función que tiene dos argumentos y ambos son lista, las posibles combinaciones de casos base e hipótesis de recurrencia son las que se muestran en la tabla previa. No hay más posibilidades porque ambos se establecen para el mismo argumento, el primero ($l1$) o el segundo ($l2$), o bien para ambos ($l1$ y $l2$).

Ahora deben analizarse los tres posibles casos base e hipótesis de recurrencia y, dependiendo del problema, elegir la opción más adecuada. No hay una regla general que nos permita seleccionar el caso base e hipótesis más adecuados porque, como se ha indicado, depende del problema a resolver y, así, nos podemos encontrar con diferentes escenarios:

1. Que el problema tenga solución de igual complejidad para cualquiera de las tres posibilidades.
2. Que el problema tenga solución con cualquiera de las tres posibilidades, pero sea más complejo de resolver con alguna de éstas.
3. Que el problema sólo tenga solución con una y sólo una de las tres posibilidades.
4. E incluso, que el problema sólo tenga solución utilizando más de uno de los posibles casos base e hipótesis de recurrencia.

Por ejemplo, para la función $my\text{-}append(l1,l2)$ nos encontramos en el segundo de los escenarios. Si se optara por la segunda o tercera de las posibilidades, para completar la concatenación de $l1$ y $l2$ desde las hipótesis $H2$ o $H3$ (ambas listas), es necesario construir una nueva lista a partir de la hipótesis que deberá tener el $car(l2)$ en un sitio concreto en medio de ésta, tal y como se indica en las figuras siguientes.

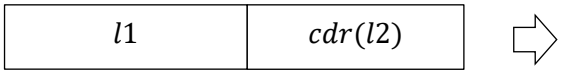

Fig 2.1: $my\text{-}append\big(l1,cdr(l2)\big) = H2$

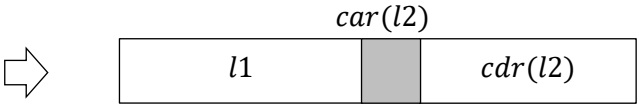

Fig 2.2: $my\text{-}append(l1,l2)$

Fig 3.1: $my\text{-}append\big(cdr(l1),cdr(l2)\big) = H3$

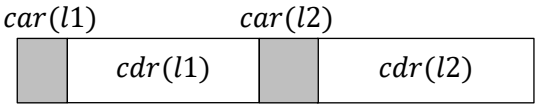

Fig 3.2: $my\text{-}append(l1,l2)$

Crear una nueva lista a partir de otra con un elemento adicional en una posición determinada es algo que se podría hacer, aunque no es precisamente trivial, es más, resulta un problema más complejo que el de partida. Sin embargo, obtener $my\text{-}append(l1,l2)$ a partir de la hipótesis $H1$ es bastante simple, lo único que se requiere es construir una nueva lista que como primer elemento tenga el $car(l1)$ y como resto la lista $H1$; es decir, $cons(car(l1),H1)$, tal y como se muestra a continuación gráficamente:

Fig 4.1: $my\text{-}append(cdr(l1),l2) = H1$

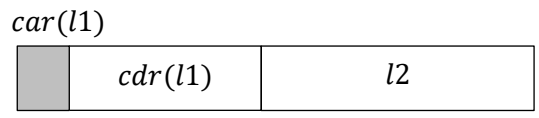

Fig 4.2: $my\text{-}append(l1,l2)$

1.2 Funciones recursivas sobre S-expresiones

Se verá aquí como realizar el análisis por casos para definir funciones recursivas sobre el subconjunto de S-expresiones que son listas, que son la S-expresiones con las que se trabajará en prácticas tal y como ya se anticipó al principio del documento. Estas funciones permiten resolver ciertos problemas que requieren un cálculo o proceso repetitivo cuando la información (o dato) a tratar es una lista multinivel (S-expresión) y es absolutamente necesario resolver el mismo problema para todas las S-expresiones del nivel superior y así sucesivamente. Obsérvese, que a diferencia de lo que se ha visto en el apartado previo, si la S-expresión es una lista multinivel entonces los elementos de la lista en el nivel superior también son S-expresiones igual de relevantes que la S-expresión principal.

Las S-expresiones se definen recurrentemente como:

a) *Base*: un átomo es una S-expresión
b) *Recurrencia*: si x e y son S-expresiones entonces el par $(x \cdot y) = cons(x, y)$ también es una S-expresión. De las S-expresiones que se pueden construir de esta forma, en prácticas sólo se manejaran las listas: *secuencias de S-expresiones entre paréntesis* y, como ya se ha visto en teoría, el par $(x \cdot y)$ será una lista siempre que y también lo sea.

1. *Base*: el resultado de la función para un átomo: ¿$f(átomo)$? Es necesario también, analizar el resultado para el caso particular $f(())$, ya que la lista vacía es átomo y lista simultáneamente.
2. *Recurrencia*: $Sexp$ no es un átomo; es decir, $Sexp = cons(car(Sexp), cdr(Sexp))$
   a. *Hipótesis*: se conocen $f(car(Sexp)) = H1$ y $f\big(cdr(Sexp)\big) = H2$
   b. *Tesis*: obtener $f(Sexp)$ combinando ambas hipótesis, $H1$ y $H2$

Fig. 5: Esquema de la definición de una función recursiva sobre una S-expresión

El esquema de la definición de una función recursiva f sobre una S-expresión $Sexp$ (Fig. 5), o *análisis por* casos, se corresponde con la definición recurrente de su dominio y, al igual que en el apartado previo, se establecerá utilizando la notación funcional habitual.

Ejemplo 1

Definir la función $atoms(Sexp)$ que retorna el número de átomos que de la S-expresión que se proporciona.

1. *Base*: el resultado de la función para un átomo; $atoms(átomo) = 1$; excepto si el átomo es la lista vacía, ya que $atoms(()) = 0$.
2. *Recurrencia*: $Sexp$ no es un átomo; es decir, $Sexp = cons(car(Sexp), cdr(Sexp))$
   a. *Hipótesis*: se conocen $atoms\big(car(Sexp)\big) = H1$ y $atoms(cdr(Sexp)) = H2$
   b. *Tesis*: $atoms(Sexp) = H1 + H2$

En Racket:

```
(define (atoms Sexp)
  (cond [(null? Sexp) 0]
        [(atom? Sexp) 1]
        [else (+ (atoms (car Sexp)) (atoms (cdr Sexp)))]))

(atoms '((b (c) a) d)) ⇒ 4
```

Ejemplo 2

Definir la función $replace(S1, S2, Sexp)$ que retorna la S-expresión $Sexp$ reemplazando cada aparición de la S-expresión $S1$ por la S-expresión $S2$.

En las funciones recursivas sobre S-expresiones debe tenerse muy presente la diferencia entre las dos partes que componen una S-expresión no atómica (su car y su cdr), en particular en lo que se refiere al nivel de anidamiento de ambas estructuras. Así, mientras que el nivel de anidamiento del cdr es el mismo que tiene en la S-expresión, su car tiene un nivel de anidamiento una unidad inferior con respecto al nivel que tiene dentro de la S-expresión (el car se saca de la S-expresión). No es que esto sólo ocurra en funciones de este tipo, ocurre exactamente lo mismo si la S-expresión se maneja como una lista de elementos, pero aquí esta diferenciación se extiende a las hipótesis de recurrencia que se establecen sobre las partes, su car y su cdr: $H1$ y $H2$, respectivamente.

Lo anteriormente indicado, también conlleva, que la comprobación de condiciones en las S-expresiones que contiene una S-expresión dada, se deban realizar sobre su car. Exactamente igual, que cuando es necesario verificar si los elementos de una lista cumplen cierta condición. Por ejemplo, en la función $replace$ es necesario identificar las S-expresiones $S1$ que contiene la S-expresión $Sexp$ y, por tanto, si $Sexp$ no es un átomo, habrá que comprobar si $car(Sexp)$ coincide o no con la S-expresión $S1$ para proceder en consecuencia. En caso afirmativo, deberá reemplazarse el $car(Sexp)$ por $S2$ y, en tal caso, no ha lugar el tener en cuenta la hipótesis $H1$.

1. *Base*: el resultado de la función para un átomo;
   $replace(S1, S2, átomo) =$ **si** $átomo = S1$
   entonces $S2$
   si_no $átomo$
   $replace(S1, S2, ()) = ()$
2. *Recurrencia*: $Sexp$ no es un átomo; es decir, $Sexp = cons(car(Sexp), cdr(Sexp))$
   a. *Hipótesis*: se conocen $replace\big(S1, S2, car(Sexp)\big) = H1$ y
   $replace(S1, S2, cdr(Sexp)) = H2$
   b. *Tesis*: $replace(S1, S2, Sexp) =$ **si** $car(Sexp) = S1$
   entonces $cons(S2, H2)$
   si_no $cons(H1, H2)$

En Racket:

```
(define (replace S1 S2 Sexp)
  (cond [(null? Sexp) Sexp]
        [(atom? Sexp)
         (if (eq? S1 Sexp) S2 Sexp)]
        [(equal? (car Sexp) S1)
         (cons S2 (replace S1 S2 (cdr Sexp)))]
        [else
         (cons (replace S1 S2 (car Sexp))
               (replace S1 S2 (cdr Sexp)))]))

(replace '(f) 0 '((a f (f)) (d a (f)) f)) ⇒ ((a f 0) (d a 0) f)
```