Lenguajes de Programación, 2023-1
Notas del laboratorio 1

Profesora: Karla Ramírez Pulido
Ayud. Lab.: Manuel Ignacio Castillo López
Facultad de Ciencias, UNAM

Manipulación de listas para recursión

En las notas A, se presenta la definición de listas y algunos ejemplos de recursión sobre listas. En esta sección, profundizaremos brevemente en las funciones que nos ayudan a manipular una lista con la intención de explorar su contenido recursivamente.

first y rest

Recordemos que de acuerdo con la definición de listas en Racket, una lista es vacía o es el elemento de un conjunto seguido de una lista (que puede ser vacía). Las funciones `first` y `rest` nos permiten extraer estos elementos [2]; por ejemplo, si x es un elemento de un conjunto y xs es una lista y ambos elementos conforman la lista *(cons x xs)* [1], podemos extraer estos componentes de la lista de la siguiente forma:

```
1 > (first (cons x xs))
2 x
3 > (rest (cons x xs))
4 xs
```

Estas dos funciones son alias de las funciones `car` y `cdr` [2]; que tienen exactamente el mismo comportamiento:

```
1 > (car (cons x xs))
2 x
3 > (cdr (cons x xs))
4 xs
```

Con `first`/`car` y `rest`/`cdr`, podemos manipular listas de forma recursiva. Por ejemplo, podemos definir la siguiente función, que toma una lista de números y suma su contenido en un único escalar.

```
1 (define (suma-con-lista listnum)
2     (if (empty? listnum)
3         0
4         (+ (first listnum) (suma-con-lista (rest listnum)))
5     )
6 )
```

Es importante aclarar que podríamos crear una versión de esta función usando `foldl`/`foldr`; lo que nos permitiría abstraer la recursión (puesto que la suma es conmutativa, no importa si sumamos los elementos de izquierda a derecha o viceversa).

Funciones de aridad no específica y funciones con múltiples resultados

En RACKET, una función no necesariamente devuelve un único valor. Es posible definir funciones que regresan múltiples valores. RACKET incluye varias herramientas que nos ayudan trabajar con este tipo de funciones; en particular, la función `values` nos ayuda a devolver una serie de valores como diferentes resultados [2].

```
1 > (values 1 2 3)
2 1
3 2
4 3
```

Es importante mencionar que los parámetros de `values` NO son una lista; sino varios valores. Esto es porque `values` es una función de *aridad no específica* (también conocidas como *Rest Argument* en RACKET); es decir, valores (`values`) que puede recibir como parámetros (también es posible llamarla sin parámetros, pero el resultado es análogo a una no-operación; es decir, no tiene ningún efecto). Por su parte, la función `apply` nos permite pasar como parámetros los valores en una lista a una función *Rest Argument*; pues como vimos en el ejemplo anterior, esperan recibir una serie de parámetros y tomarían una lista en su totalidad como un solo parámetro. apply recibe una función *Rest Argument* y una o varias listas cuyos valores pasará a la función dada [2].

```
1 > (apply values (list 1 2 3))
2 1
3 2
4 3
```

Declaraciones locales

Dentro de una función; sea nombrada o anónima, podemos definir variables locales, incluyendo otras funciones [2]. Podemos declarar valores que nos ayuden a mantener resultados de forma que no tengamos que estar repitiendo operaciones; y hacer más eficientes nuestros programas (respecto al tiempo). Para declarar variables locales, podemos usar las siguientes instrucciones [2]:

`let`

`let` es la instrucción básica que nos permite definir variables locales en una definición [2]. Ejemplo de uso:

```
1 (define (tengo-variables)
2     (let (
3         [x 1]
4         [y 2]
5         [z (+ 1 2)])
6         (values x y z)
7     )
8 )
```

Podemos declarar tantas variables como sea necesario. Es importante considerar que no se permite reutilizar nombres de variables; si en el mismo bloque de definición `let` aparecen declaraciones con el mismo nombre de variable, el intérprete de RACKET dispara un error. Tampoco podemos hacer referencia a los valores en una variable para definir una siguiente, para hacer esto debemos usar `let*` [2].

`let*`

`let*` es muy similar a `let`, con la diferencia de que nos permite utilizar una variable definida anteriormente para definir otra [2].

```
1 (define (tengo-variables)
2     (let* (
3         [x 1]
4         [y 2]
5         [z (+ x y)])
6         (values x y z)
7     )
8 )
```

En este caso, `let*` sí nos permite sobre-escribir una variable definida anteriormente, si tratamos de definir dos variables con el mismo nombre, la variable tendrá el último valor que le hubiera sido asignado [2].

`let-values`

Podemos almacenar los valores devueltos por una función que regresa varios valores con `let-values`. Además, cuando usamos `let-values`, es importante encerrar las variables que se definen con paréntesis [2]. Por ejemplo, la función definida en los ejemplos anteriores `tengo-variables` devuelve tres resultados: 1, 2 y 3. Podríamos almacenar estos valores en una declaración de otra función de la siguiente manera:

```
1 (define (uso-variables)
2     (let-values (
3         [(x y z) (tengo-variables)]
4         [(unico-val) 5])
5         (+ x y z unico-val)
6     )
7 )
```

`let*-values`

Por último, `let*-values` combina la funcionalidad de `let*` con `let-values`; habilitando la declaración de múltiples variables en una misma sentencia cuyo valor puede ser leído o sobre-escrito en una declaración siguiente [2].

Definiendo funciones dentro de otras funciones

Por otro lado, también podemos definir funciones dentro de otra función. Esto nos puede ayudar a tener funciones auxiliares para una función que se expone al módulo [2]. Por ejemplo, la siguiente función toma una lista numérica y obtiene el promedio de la lista. Para ello, utiliza una función auxiliar que realiza la suma de los valores en la lista.

```
(define (promedio-lista listnum)
    (define (suma-listnum listnum)
        (apply + listnum))
    (/ (suma-listnum listnum) (length listnum))
)
```

Retomando las funciones *Rest Argument*, es posible definir nuestras propias funciones de aridad no específica, pero para ello debemos usar la sintaxis de lambda (funciones anónimas); puesto que la sintaxis de funciones nombradas no soporta este tipo de declaraciones. Sin embargo, podemos aprovechar el hecho de que podemos definir una función anónima dentro de otra función como alternativa para el soporte de funciones *Rest Argument* nombradas [2].

Cuando definimos una función *Rest Argument*, definimos un solo parámetro que representa cero o más valores que puede recibir; el parámetro los almacenará en forma de lista. Note que si la función recibe una lista, el argumento se evaluará como una lista que contiene la lista dada. [2]. Por ejemplo, la siguiente función toma una serie de números; que puede ser vacía, y devuelva la suma de los números en la serie *nums*.

```
(define suma-nums (lambda nums
    (let (
        [num-nums (length nums)])
        (cond
            [(= num-nums 0) 0]
            [(= num-nums 1) (first nums)]
            [else (+ (first nums) (apply suma-nums (rest nums)))]
        )
    )
))
```

Observe que la función `suma-nums` es equivalente a la función + incluida en Racket.

Diccionarios

Podemos definir diccionarios en Racket en forma de tablas `hash`. Las tablas `hash` se encuentran implementadas como un tipo de datos; por lo que como con otros tipos en Racket, podemos preguntar si un valor es una tabla `hash` con el predicado `hash?`. Podemos declarar tablas `hash` mutables o inmutables; la principal diferencia entre éstas es que las tablas `hash` inmutables ofrecen un mejor desempeño respecto al tiempo. Para crear una tabla `hash` inmutable usamos la función `hash`, la cual recibe un número arbitrario de valores (es una función *Rest Argument*) que toma en pares; clave y valor [2]:

```
(hash "uno" 1 "dos" 2 "tres" 3)
```

En el ejemplo anterior, `hash` está asociando la cadena *uno* con el valor *1*, la cadena *dos* con el valor *2* y *tres* con *3*.

Por su parte, creamos tablas mutables con la función `make-hash`, que devuelve una tabla `hash` que podemos manipular con las funciones `hash-set!` para agregar elementos y `hash-remove!` para quitarlos [2].

```
> (define tabla-mutable (make-hash))
> (hash-set! tabla-mutable "uno" 1)
> (hash-set! tabla-mutable "dos" 2)
> (hash-set! tabla-mutable "tres" 3)
> (hash-remove! tabla-mutable "uno")
```

En ambos casos, para obtener el valor asociado a una llave en una tabla `hash`, utilizamos la función `hash-ref`, la cual recibe la tabla que queremos consultar, la llave cuyo valor queremos conocer y opcionalmente un valor que será devuelto en caso de que la llave no exista en la tabla. Por defecto, `hash-ref` dispara un error cuando se intenta consultar una llave no definida en la tabla dada [2].

```
1 > (hash-ref tabla-mutable "dos" #f)
2 2
3 > (hash-ref tabla-mutable "uno" #f)
4 #f
```

Existen otros tipos de tablas `hash` en Racket; como las tablas `hash` de enlaces débiles, y otras operaciones que podemos hacer sobre ellos, como crear un conjunto a partir de una tabla `hash` o usar una función para actualizar el valor asociado a una clave en la tabla [2].

Por último, no es exclusivamente necesario que definamos una tabla `hash` como miembro de un módulo directamente, si solo se requiere dentro de una función, podemos declararlo en ella; incluyendo con la notación `let` [2].

Explorando un diccionario

Recordemos que Racket es un lenguaje principalmente funcional [1]. En este paradigma, la recursión suele ser una de las principales herramientas para realizar iteraciones; a diferencia de en otros paradigmas, como el estructurado u orientado a objetos, en donde por el contrario, se suele recomendar evitar la recursión; pues su implementación en este tipo de lenguajes suele involucrar pilas de ejecución y segmentos de memoria que deben ser administrados de forma más minuciosa que en lenguajes funcionales. Por lo anterior, en muchos programas escritos en Racket no será necesario utilizar las estructuras iterativas, puesto que por diversas razones se prefieren estrategias recursivas. Sin embargo, aún existen casos en los que puede ser conveniente realizar iteraciones [1, 2].

En el caso en el que necesitemos recorrer una tabla `hash`, podemos hacerlo de forma recursiva de manera accesible con las funciones `hash-iterate-first`, `hash-iterate-next`, `hash-iterate-key`, `hash-iterate-value`, `hash-iterate-pair` y `hash-iterate-key+value`:

- **`hash-iterate-first` y `hash-iterate-next`**
  Nos ayudan a obtener un índice que podemos pasar a las otras cuatro funciones para obtener las claves y/o los valores asociados en la tabla `hash`. `hash-iterate-first` únicamente recibe una tabla `hash`, ésta regresa un índice que apunta a su primer elemento, mientras que `hash-iterate-next`; además de la tabla `hash`, necesita un índice válido en la tabla. Devuelve el índice que apunta al siguiente elemento al que apunta el índice que se le haya proporcionado. En caso de que la tabla `hash` sea vacía, o no exista ningún otro elemento después del índice que se de a `hash-iterate-next`; estas funciones devuelven `#f` (falso).
  En el algoritmo 1, se utiliza `hash-iterate-next` para iterar sobre un diccionario de manera recursiva. La función del algoritmo, `obtener-freq-mayor`, debe ser llamada inicialmente con `hash-iterate-first` para obtener el índice del primer elemento del diccionario; o `#f` si éste es vacío.

- **`hash-iterate-key` y `hash-iterate-value`**
  Devuelven la clave o el valor (respectivamente) en una tabla `hash` que se les dé como parámetro. Esta clave o valor (dependiendo del método que usemos), son referenciados por un índice que debemos proporcionar también a estas funciones como segundo parámetro. Podemos obtener el índice de esta clave o valor con las funciones `hash-iterate-first` y `hash-iterate-next`; dependiendo si queremos la clave o valor del primer elemento almacenado en el diccionario, o de un elemento subsecuente al iterar sobre el diccionario.
  En el algoritmo 1, el diccionario `tablahs` contiene valores numéricos (número - frecuencia) como entradas (clave - valor). Se utiliza `hash-iterate-value` para obtener la frecuencia de cada número (el valor de cada entrada del diccionario) y obtener la frecuencia mayor en el diccionario.

- **`hash-iterate-pair` y `hash-iterate-key+value`**
  Combinan los resultados que devolverían `hash-iterate-key` y `hash-iterate-value` en una sola operación de la siguiente manera:
  - `hash-iterate-pair` equivale a crear el par `(cons hash-iterate-key hash-iterate-value)`
  - `hash-iterate-key+value` devuelve los *dos* valores que arrojarían `hash-iterate-key` y `hash-iterate-value`. Note que en ambos casos `hash-iterate-key` y `hash-iterate-value` serían llamadas con la misma tabla `hash` e índice para construir el resultado.

En el algoritmo 1, se presenta la función obtener-freq-mayor que dada una tabla `hash` que almacena frecuencias de elementos de alguna otra colección, determina cual es la mayor frecuencia de dicha colección; según el `hash`:

```
1  (define (obtener-freq-mayor tablahs i freq-mayor)
2      (if (eq? i #f)
3          freq-mayor
4          (obtener-freq-mayor tablahs (hash-iterate-next tablahs i)
5              (if (> (hash-iterate-value tablahs i) freq-mayor)
6                  (hash-iterate-value tablahs i)
7                  freq-mayor
8              )
9          )
10     )
11 )
```

Algoritmo 1: Ejemplo de iteración de un diccionario para determinar la frecuencia mayor según una tabla `hash` con entradas (clave - valor)

Para que esta función se comporte como lo esperamos, tendría que ser llamada de la siguiente forma; por ejemplo usando la tabla `hash` definida en los ejemplos anteriores `tabla-mutable`:

```
1  (obtener-freq-mayor tabla-mutable (hash-iterate-first tabla-mutable) 0)
```

Pruebas unitarias con plai

La variante de Racket `#plai` incluye herramientas con las que podemos implementar pruebas unitarias. Una prueba unitaria, es aquella donde un componente unitario del programa; como puede ser una función (rutina/método) o una sola clase (en el caso de Programación Orientada a Objetos) [3]. En `#plai`, la función `test` nos ayuda a crear validaciones en las que dos expresiones regresan el mismo resultado. Usando una expresión como la función que queremos probar, y otra expresión como el resultado esperado, podemos implementar casos de prueba que nos ayudan a validar el funcionamiento correcto de nuestros programas. Por ejemplo, podemos validar que la función `suma-nums` en efecto que comporta igual que la función + definida por defecto en el lenguaje de la siguiente manera:

```
1  (test (suma-nums 1 2 3) (+ 1 2 3))
```

Con lo anterior, podríamos definir una función de prueba como la siguiente:

```
1  (define (prueba-suma-nums)
2      (test (suma-nums 1 2 3) 6)
3  )
```

Diseño guiado por pruebas

En Ingeniería de Software, la estrategia de diseño guiado por pruebas, se basa en comenzar la implementación definiendo pruebas que verifican que las funcionalidades a implementar se comporten de la manera deseada, con el fin de mejorar la calidad del producto de software que se construye, así como su documentación; ya que las pruebas ayudan a entender el comportamiento de las funciones que evalúan y su interacción con otros componentes. En esta estrategia, se comienza por identificar las funciones más relevantes que van a implementarse durante una etapa del proyecto; al inicio de dicha etapa [3].

Después se escriben las pruebas que toman la firma de la función que verifican junto con los resultados que se espera arrojen dichas funciones. Puesto que una función puede regresar una gran gama de resultados diferentes; por ejemplo una función que suma podría arrojar todos los enteros válidos para la plataforma con la que se trabaja, se elige un conjunto mínimo de casos relevantes para validar su resultado. Por ejemplo, al probar una función que obtiene potencias de dos, podríamos validar que la potencia cero devuelve 1, la potencia 1 devuelve 2 y algunos casos donde se conozca el resultado de la operación; como 2^5=32 y 2^{10}=1024 [3].

De esta forma, al inicio de las tareas de implementación de estas funciones, se busca que al finalizar la implementación se satisfagan las pruebas definidas; pues éstas deberían garantizar que las funciones que se prueban únicamente deberían pasar las pruebas si se apegan a su especificación. Así, al final de la etapa del proyecto, todas las pruebas definidas al inicio deberían ejecutarse correctamente y la versión resultante del producto de software puede continuar en la siguiente etapa del proyecto o de su ciclo de vida [3].

Una recomendación al definir pruebas es NO repetir el código de la función que se está probando; ni usar una serie de instrucciones similares a la de función que se prueba. Es mejor utilizar un resultado esperado como constante o que pueda ser construido con otras funciones cuyo comportamiento sea confiable.

Referencias

[1] K. Ramírez Pulido y M. Soto Romero, «Notas de clase A: Introducción a Racket», Facultad de Ciencias, UNAM, sep. 2021.

[2] M. Flatt y R. B. Findler, «The Racket Guide», PLT Research Group, 8.5.0.8, jul. 2022.

[3] I. Sommerville, Software engineering. Boston: Addison-Wesley, 2011.