Informe de Proyecto 2: Programación en LISP

Lógica para ciencias de la computación

Dimatz, Juan y Jouglard, Juan Francisco

26/06/2017

Profesores: Falappa, Marcelo y Gómez Lucero, Mauro.

1er. Cuatrimestre del 2017

Tabla de contenidos

Introducción

Se nos dieron los siguientes problemas para ser resuelto usando el lenguaje de programación LISP: prefijo, sublista, reverso recursivo y eliminar el n-ésimo elemento de una lista y todas sus sublistas.
Para llevar a cabo este proyecto usamos el compilador GNU CLISP versión 2.49.
En la siguiente parte del documento se explicarán las estrategias utilizadas para la resolución, así como también, las decisiones que se tomaron. También se agrega el código desarrollado, así como también, ejemplos de ejecuciones.

Prefijo

Funciones utilizadas: prefijo, prefijo_aux y largo.

Esta función recibe como entrada una lista y un número N, y tiene como objetivo devolver una nueva lista con los primeros N elementos de la lista recibida por parámetro. Lo primero que hace la función al ser invocada, es validar la entrada, chequeando que el índice no supere la longitud de la lista, utilizando la función “largo” provista por la cátedra. El índice debe ser un número entero mayor a 0. Si la misma es correcta, llama a una función auxiliar que se encargará de recorrer la lista hasta el elemento n de la siguiente manera:

Dada una lista [X\|Xs] tal que X sea la cabeza y Xs la cola, y N el índice recibido como parámetro.

- Si el índice es 1, quiere decir que ya recorrió la lista hasta el elemento requerido, por lo tanto solo devuelve X.
- Sino, concatena, X con el resultado de invocar recursivamente a la función con los parámetros Xs y N-1.

Si la entrada es inválida, se solicitara por pantalla ingresar la función con valores correctos.

Ejemplos de corridas de la función:

```
[4]> (prefijo '(a b c d e f) 3)
1. Trace: (PREFIJO '(A B C D E F) '3)
2. Trace: (PREFIJO '(B C D E F) '2)
3. Trace: (PREFIJO '(C D E F) '1)
3. Trace: PREFIJO ==> (C)
2. Trace: PREFIJO ==> (B C)
1. Trace: PREFIJO ==> (A B C)
(A B C)
```

```
[5]> (prefijo '(a b c d e f) 10)
1. Trace: (PREFIJO '(A B C D E F) '10)
1. Trace: PREFIJO ==> (INGRESE UN INDICE VALIDO)
(INGRESE UN INDICE VALIDO)
```

```
[6]> (prefijo '(a b c d e f) 6)
1. Trace: (PREFIJO '(A B C D E F) '6)
2. Trace: (PREFIJO '(B C D E F) '5)
3. Trace: (PREFIJO '(C D E F) '4)
4. Trace: (PREFIJO '(D E F) '3)
5. Trace: (PREFIJO '(E F) '2)
6. Trace: (PREFIJO '(F) '1)
6. Trace: PREFIJO ==> (F)
5. Trace: PREFIJO ==> (E F)
4. Trace: PREFIJO ==> (D E F)
3. Trace: PREFIJO ==> (C D E F)
2. Trace: PREFIJO ==> (B C D E F)
1. Trace: PREFIJO ==> (A B C D E F)
(A B C D E F)
```

Sublista

Funciones utilizadas: sublist, sublist_aux y largo.

El objetivo de esta función es similar a “prefijo”, a diferencia que ahora se retornara una lista con los elementos entre la posición X y la posición Y de la lista.
Para la resolución de este problema, utilizamos una función cáscara, que valida los argumentos recibidos, corroborando que X sea mayor a 0, e Y sea mayor a X y menor al tamaño de la lista.

Si son correctos, llama a una función auxiliar, que se encargará de recorrer la lista, hasta encontrar el primer índice, y luego, solo basta con llamar al predicado prefijo con lo que queda de la lista y con el segundo índice. Hay que tener en cuenta que al mismo hay que restarle la cantidad de elementos que recorrimos.

Ejemplos de corridas de la función:

```
[8]> (sublist '(a b c d e f) 2 5)
1. Trace: (SUBLIST '(A B C D E F) '2 '5)
2. Trace: (SUBLIST '(B C D E F) '1 '4)
3. Trace: (PREFIJO '(B C D E F) '4)
4. Trace: (PREFIJO '(C D E F) '3)
5. Trace: (PREFIJO '(D E F) '2)
6. Trace: (PREFIJO '(E F) '1)
6. Trace: PREFIJO ==> (E)
5. Trace: PREFIJO ==> (D E)
4. Trace: PREFIJO ==> (C D E)
3. Trace: PREFIJO ==> (B C D E)
2. Trace: SUBLIST ==> (B C D E)
1. Trace: SUBLIST ==> (B C D E)
(B C D E)
```

```
[9]> (sublist '(a b c d e f) 2 10)
1. Trace: (SUBLIST '(A B C D E F) '2 '10)
1. Trace: SUBLIST ==> (INGRESE INDICES VALIDOS)
(INGRESE INDICES VALIDOS)
```

```
[10]> (sublist '(a b c d e f) 5 3)
1. Trace: (SUBLIST '(A B C D E F) '5 '3)
1. Trace: SUBLIST ==> (INGRESE INDICES VALIDOS)
(INGRESE INDICES VALIDOS)
```

Reversa recursiva

Funciones utilizadas: rec-reverse.

Esta función recibe como entrada una lista cuyos elementos pueden ser “átomos”, u otras sub listas anidadas, y su objetivo es retornar la lista invertida, con la cualidad de que los elementos de la lista que sean sublistas, también estén invertidas.
La estrategia utilizada en este caso, fue diferenciar la forma en que procesamos los elementos de la lista.

Dada la lista [X\|Xs] tal que X es la cabeza y Xs la cola.

Si el X es un átomo (número, letra, etc.), concatenamos el resultado de llamar a la función con Xs y X, en ese orden para que luego el resultado sea el reverso del original. Si en cambio, X es una sublista, concatenamos el resultado de llamar a la función con Xs y el resultado de llamar a la función con X.

Ejemplos de corridas de la función:

```
[4]> (rec-reverse '(a (b c) (d (e f) g) h))
1. Trace: (REC-REVERSE '(A (B C) (D (E F) G) H))
2. Trace: (REC-REVERSE '((B C) (D (E F) G) H))
3. Trace: (REC-REVERSE '((D (E F) G) H))
4. Trace: (REC-REVERSE '(H))
5. Trace: (REC-REVERSE 'NIL)
5. Trace: REC-REVERSE ==> NIL
4. Trace: REC-REVERSE ==> (H)
4. Trace: (REC-REVERSE '(D (E F) G))
5. Trace: (REC-REVERSE '((E F) G))
6. Trace: (REC-REVERSE '(G))
7. Trace: (REC-REVERSE 'NIL)
7. Trace: REC-REVERSE ==> NIL
6. Trace: REC-REVERSE ==> (G)
6. Trace: (REC-REVERSE '(E F))
7. Trace: (REC-REVERSE '(F))
8. Trace: (REC-REVERSE 'NIL)
8. Trace: REC-REVERSE ==> NIL
7. Trace: REC-REVERSE ==> (F)
6. Trace: REC-REVERSE ==> (F E)
5. Trace: REC-REVERSE ==> (G (F E))
4. Trace: REC-REVERSE ==> (G (F E) D)
3. Trace: REC-REVERSE ==> (H (G (F E) D))
3. Trace: (REC-REVERSE '(B C))
4. Trace: (REC-REVERSE '(C))
5. Trace: (REC-REVERSE 'NIL)
5. Trace: REC-REVERSE ==> NIL
4. Trace: REC-REVERSE ==> (C)
3. Trace: REC-REVERSE ==> (C B)
2. Trace: REC-REVERSE ==> (H (G (F E) D) (C B))
1. Trace: REC-REVERSE ==> (H (G (F E) D) (C B) A)
(H (G (F E) D) (C B) A)
```

```
[5]> (rec-reverse '(a (b c) ((e f) g)))
1. Trace: (REC-REVERSE '(A (B C) ((E F) G)))
2. Trace: (REC-REVERSE '((B C) ((E F) G)))
3. Trace: (REC-REVERSE '(((E F) G)))
4. Trace: (REC-REVERSE 'NIL)
4. Trace: REC-REVERSE ==> NIL
4. Trace: (REC-REVERSE '((E F) G))
5. Trace: (REC-REVERSE '(G))
6. Trace: (REC-REVERSE 'NIL)
6. Trace: REC-REVERSE ==> NIL
5. Trace: REC-REVERSE ==> (G)
5. Trace: (REC-REVERSE '(E F))
6. Trace: (REC-REVERSE '(F))
7. Trace: (REC-REVERSE 'NIL)
7. Trace: REC-REVERSE ==> NIL
6. Trace: REC-REVERSE ==> (F)
5. Trace: REC-REVERSE ==> (F E)
4. Trace: REC-REVERSE ==> (G (F E))
3. Trace: REC-REVERSE ==> ((G (F E)))
3. Trace: (REC-REVERSE '(B C))
4. Trace: (REC-REVERSE '(C))
5. Trace: (REC-REVERSE 'NIL)
5. Trace: REC-REVERSE ==> NIL
4. Trace: REC-REVERSE ==> (C)
3. Trace: REC-REVERSE ==> (C B)
2. Trace: REC-REVERSE ==> ((G (F E)) (C B))
1. Trace: REC-REVERSE ==> ((G (F E)) (C B) A)
((G (F E)) (C B) A)
```

Eliminar elemento n recursivamente

Funciones utilizadas: elim-n-rec y eliminar.

Dado un anidamiento de listas, y un número N, esta función permite eliminar los N-ésimos elementos de cada una de las listas.
Para este problema también usamos una cáscara para validar las entradas, y luego llamamos a una función auxiliar, que se encarga de recorrer la lista buscando el índice para eliminar el correspondiente elemento, así como, también eliminando de las sublistas.

Si el elemento de la lista es una sublista, se llama recursivamente a la función con el índice inicial, para también eliminar el n-ésimo elemento de esta. Aunque ya se haya eliminado el elemento, es necesario, seguir recorriendo la lista para eliminar el elemento de una posible sublista dentro de la misma.

Para llevar a cabo este procesamiento es importante diferenciar el comportamiento de la función una vez que ya se eliminó el elemento de la lista, en este caso solo se debe recorrer en busca de sublistas de las que hay que eliminar. Para esto usamos un valor que se le pasa por parámetro a la función que indica si ya se eliminó de la lista que se está recorriendo o si todavía se está buscando el elemento en el índice indicado.

También es imprescindible mantener el índice que se pasa como parámetro inicialmente, para luego poder eliminar esa posición en las sublistas. Por eso, la función inicialmente recibe dos copias del índice, una será la que irá decrementando hasta llegar al índice indicado, y la otra la pasara por parámetro cuando llame a la función con las sublistas.

Ejemplos de corridas de la función:

```
[7]> (elim-n-rec '((1 2) (3 (4 5)) (6 (7 8 9) 10)) 2)
1. Trace: (ELIM-N-REC '((1 2) (3 (4 5)) (6 (7 8 9) 10)) '2)
1. Trace: ELIM-N-REC ==> ((1) (6 10))
((1) (6 10))
```

```
[4]> (elim-n-rec '(a (b c) ((e f) g h)) 2)
1. Trace: (ELIM-N-REC '(A (B C) ((E F) G H)) '2)
1. Trace: ELIM-N-REC ==> (A ((E) H))
(A ((E) H))
```

```
1. Trace: (ELIM-N-REC '(A B C (E (H I J K) G) (L M N)) '4)
2. Trace: (ELIMINAR '(A B C (E (H I J K) G) (L M N)) '4 '4 '0)
3. Trace: (ELIMINAR '(B C (E (H I J K) G) (L M N)) '3 '4 '0)
4. Trace: (ELIMINAR '(C (E (H I J K) G) (L M N)) '2 '4 '0)
5. Trace: (ELIMINAR '((E (H I J K) G) (L M N)) '1 '4 '0)
6. Trace: (ELIMINAR '((L M N)) '1 '4 '1)
7. Trace: (ELIMINAR '(L M N) '4 '4 '0)
8. Trace: (ELIMINAR '(M N) '3 '4 '0)
9. Trace: (ELIMINAR '(N) '2 '4 '0)
10. Trace: (ELIMINAR 'NIL '1 '4 '0)
10. Trace: ELIMINAR ==> NIL
9. Trace: ELIMINAR ==> (N)
8. Trace: ELIMINAR ==> (M N)
7. Trace: ELIMINAR ==> (L M N)
7. Trace: (ELIMINAR 'NIL '1 '4 '1)
7. Trace: ELIMINAR ==> NIL
6. Trace: ELIMINAR ==> ((L M N))
5. Trace: ELIMINAR ==> ((L M N))
4. Trace: ELIMINAR ==> (C (L M N))
3. Trace: ELIMINAR ==> (B C (L M N))
2. Trace: ELIMINAR ==> (A B C (L M N))
1. Trace: ELIM-N-REC ==> (A B C (L M N))
(A B C (L M N))
Break 2 [12]>
```

Decisiones tomadas

- Permitimos que el índice pasado como parámetro para elim-n-rec sea mayor al largo de la lista principal, ya que puede haber sublistas que si tengan esa longitud. Si se da el caso de que se recorrió toda la lista y no se llegó al índice, el resultado es la misma lista (sin eliminar ningún elemento). Esto se da en todas las listas que se evalúan, no solo en la principal.
- Al tratar de resolver el problema de eliminar recursivamente nos encontramos con dos estrategias, la primera era ir recorriendo por niveles las listas e ir eliminando el n-ésimo elemento de cada una. Es decir, primero la lista principal, luego ver cuáles de los elementos de la misma son listas y recorrer esas, y así hasta llegar al final. La segunda y con la cual nos terminamos quedando es la de recorrer recursivamente una lista y todas las sublistas que sean elementos de ella. Es decir, recorremos la lista principal, y si encontramos un elemento que es una lista, nos adentramos en la misma para eliminar el n-ésimo elemento, así hasta recorrer todas las listas. Sería el equivalente a un recorrido en preorden de un árbol.

Código LISP

```
(DEFUN prefijo (L X)
    (COND
        (
            (OR (NOT (ZEROP (mod X 1))) (> X (largo L)) (< X 0)) '(Ingrese un indice valido)
        )
        (
            T (prefijo_aux L X)
        )

    )

)


(DEFUN prefijo_aux (L X)
    (COND
        (
            (> X 1) (APPEND (LIST(CAR L)) (prefijo (CDR L) (- X 1)))
        )

        (
            (EQUAL X 1) (LIST (CAR L))
        )

    )
)
```

```
(DEFUN sublist (L X Y)

   (COND
       (
           (OR (< X 0) (> Y (largo L)) (> X Y) (NOT (ZEROP (mod X 1))) (NOT (ZEROP (mod Y 1)))) '(Ingrese indices validos)
       )
       (
           T (sublist_aux L X Y)
       )

   )

)
(DEFUN sublist_aux (L X Y)

       (COND
           (
               (> X 1) (sublist (CDR L) (- X 1) (- Y 1))
           )
           (
               (EQUAL X 1) (prefijo L Y)
           )
       )

)
```

```
(DEFUN largo (L)
            (COND
                    (
                        (NULL L) 0
                    )
                    (
                        T (+ 1 (largo (CDR L)))
                    )
            )
)
```

```
(DEFUN rec-reverse (L)

        (COND
            (
                (NULL L) NIL
            )
            (
                (ATOM (CAR L)) (APPEND (rec-reverse (CDR L)) (LIST(CAR L)))
            )

            (
                T (APPEND (rec-reverse (CDR L)) (LIST (rec-reverse (CAR L))))
            )


        )
)
```

```
(DEFUN eliminar (L X Y Elimine)

    (COND
            (
                (NULL L) NIL
            )
            (
                (EQUAL X 1) (COND
                                ((ZEROP Elimine) (eliminar (CDR L) 1 Y 1))
                                ((ATOM(CAR L)) (APPEND (LIST (CAR L)) (eliminar (CDR L) 1 Y 1)))
                                (T (APPEND (LIST (eliminar (CAR L) Y Y 0)) (eliminar (CDR L) 1 Y 1)))
                            )
            )
            (
                (ATOM (CAR L)) (APPEND (LIST (CAR L)) (eliminar (CDR L) (- X 1) Y 0))
            )
            (
                T (APPEND (LIST (eliminar (CAR L) Y Y 0)) (eliminar (CDR L) (- X 1) Y 0))
            )
    )

)

(DEFUN elim-n-rec (L N)

    (COND
        (
            (OR (> 0 N) (NOT (ZEROP (mod N 1)))) '(Ingrese un indice valido)
        )
        (
            T (eliminar L N N 0)
        )
    )

)
```