Pontificia Universidad Católica Madre y Maestra

CSTI-1870-5238 - Desarrollo Móvil

Tema:

Procesamiento Lenguaje y Compilación

Integrantes:

1014-3611 Starlin Frías Luna

Profesor:

Juan Felipe

Santiago de los Caballeros, República Dominicana

Miércoles 5 de Febrero del 2025

Introducción:

El objetivo principal de esta tarea T1 es realizar ejercicios con respecto al lenguaje SML.

Este conjunto de ejercicios desarrolla habilidades progresivas en el manejo de fechas y listas en SML, cubriendo conceptos fundamentales de programación funcional como recursión, pattern matching y manipulación de listas.

Ejercicios:

1. Escriba una función is_older función que toma dos fechas y se evalúa como verdadera o falsa. Se evalúa a

verdadero si el primer argumento es una fecha que viene antes del segundo argumento . (Si las dos fechas son iguales, el resultado es falso.)

Solución:

```
fun is_older((y1,m1,d1), (y2,m2,d2)) =

    y1 < y2 orelse

    (y1 = y2 andalso m1 < m2) orelse

    (y1 = y2 andalso m1 = m2 andalso d1 < d2)
```

Explicación: Compara fechas jerárquicamente (año, mes, día) usando operadores lógicos `orelse` y `andalso`.

Prueba:

- is_older((2023,3,15), (2023,3,16)); (* true *)
- is_older((2023,3,15), (2023,3,15)); (* false *)
- is_older((2022,12,31), (2023,1,1)); (* true *)
- is_older((2023,2,28), (2023,3,1));   (* true *)

2. Escriba una función number_in_month que toma una lista de fechas y un mes (es decir, un int) y devuelve el número de fechas en la lista que contienen el mes determinado.

Solución:

```
fun number_in_month(dates, month) =

    case dates of

        [] => 0

      | (y,m,d)::rest => (if m = month then 1 else 0) +
number_in_month(rest, month)
```

Explicación: Usa recursión para contar fechas que coinciden con el mes especificado.

Prueba:

- number_in_month([(2023,3,15),(2023,3,16),(2023,4,1)], 3); (* 2 *)
- number_in_month([(2023,1,1),(2023,2,1),(2023,3,1)], 4); (* 0 *)
- number_in_month([], 3); (* 0 *)

3. Escriba una función number_in_months que toma una lista de fechas y una lista de meses (es decir, una lista int) y devuelve el número de fechas en la lista de las fechas que se encuentran en alguno de los meses en la lista de meses. Supondremos que la lista de los meses no ha repetido ningún número.
Sugerencia: Utilice su respuesta al problema anterior.

Solución:

```
fun number_in_months(dates, months) =

    case months of

        [] => 0
```

```
      | month::rest => number_in_month(dates, month) +
number_in_months(dates, rest)
```

Explicación: Extiende la función anterior para trabajar con múltiples meses usando recursión.

Prueba:

- number_in_months([(2023,3,15),(2023,4,16),(2023,3,1)], [3,4]); (* 3 *)
- number_in_months([(2023,1,1),(2023,2,1)], [3,4,5]); (* 0 *)
- number_in_months([], [1,2,3]); (* 0 *)

4. Escriba un función dates_in_month que toma una lista de fechas y un mes (es decir, un int) y devuelve una lista que contiene las fechas de la lista de argumentos de fechas que están en el mes. La lista devuelta debe contener las fechas en el orden que se les dio en un principio.

Solución:

```
fun dates_in_month(dates, month) =

    case dates of

        [] => []

      | date::rest => if #2 date = month

                      then date::dates_in_month(rest, month)

                      else dates_in_month(rest, month)
```

Explicación: Filtra y retorna las fechas que coinciden con el mes especificado.

Prueba:

- dates_in_month([(2023,3,15),(2023,3,16),(2023,4,1)], 3); (* [(2023,3,15),(2023,3,16)] *)
- dates_in_month([(2023,1,1),(2023,2,1),(2023,3,1)], 4); (* [] *)

- dates_in_month([], 3); (* [] *)

5. Escriba una función dates_in_months que toma una lista de fechas y una lista de meses (es decir, una lista int) y devuelve una lista que sostiene las fechas de la lista de argumentos de las fechas que se encuentran en alguno de los meses en la lista de meses. Supondremos que en la lista de los meses no hay repetido ningún número. Sugerencia: Utilice su respuesta al problema anterior y el operador de lista append (@).

Solución:

```
fun dates_in_months(dates, months) =

    case months of

        [] => []

      | month::rest => dates_in_month(dates, month) @
dates_in_months(dates, rest)
```

Explicación: Combina resultados de `dates_in_month` para múltiples meses usando concatenación de listas.

Prueba:

- dates_in_months([(2023,3,15),(2023,4,16),(2023,3,1)], [3,4]); (* [(2023,3,15),(2023,3,1),(2023,4,16)] *)
- dates_in_months([(2023,1,1),(2023,2,1)], [3,4]); (* [] *)
- dates_in_months([], [1,2,3]); (* [] *)

Más ejercicios SML:

1.- alternate

Escriba una función alternate: int list -> int que tome una lista de números y los agregue con signo alterno. Por

ejemplo, suplente [1,2,3,4] = 1 - 2 + 3 - 4 = -2.

Solución:

```
fun alternate [] = 0
  | alternate (x::xs) =
    let
      fun alt ([], _) = 0
        | alt (x::xs, sign) = (x * sign) + alt(xs, ~sign)
    in
      alt(x::xs, 1)
    end;
```

Prueba:

- alternate [1,2,3,4]; (* Salida esperada: -2 *)
- alternate [5,3,2]; (* Salida esperada: 4 *)
- alternate []; (* Salida esperada: 0 *)

2.- min_max

Escriba una función min_max : int list -> int * int que tome una lista de números no vacía y devuelva un par

(min, max) del mínimo y el máximo de los números de la lista.

Solución:

```
fun min_max [] = raise Empty
  | min_max [x] = (x, x)
  | min_max (x::xs) =
    let
      val (min, max) = min_max xs
    in
      (Int.min(x, min), Int.max(x, max))
    end;
```

Prueba:

- min_max [1,2,3,4,5]; (* Salida esperada: (1,5) *)
- min_max [7]; (* Salida esperada: (7,7) *)
- min_max [3,1,4,1,5]; (* Salida esperada: (1,5) *)

3.- cumsum

Escriba una función cumsum: int list -> int list que tome una lista de números y devuelva una lista de las sumas

parciales de esos números. Por ejemplo cumsum [1,4,20] = [1,5,25].

Solución:

```
fun cumsum [] = []
  | cumsum (x::xs) =
    let
      fun cum ([], _) = []
        | cum (x::xs, acc) = (acc + x)::cum(xs, acc + x)
```

```
in
  cum(x::xs, 0)
end;
```

Prueba:

- cumsum [1,4,20]; (* Salida esperada: [1,5,25] *)
- cumsum [2,3,4]; (* Salida esperada: [2,5,9] *)
- cumsum []; (* Salida esperada: [] *)

4.- saludo

Escriba un función saludo : opción de cadena -> cadena que dada una opción de cadena SOME nombre devuelve la cadena "¡Hola, ...!" donde los puntos serían reemplazados por nombre. Tenga en cuenta que el nombre se proporciona como una opción, por lo que si es None, reemplace los puntos con "usted".

Solución:

```
fun saludo NONE = "¡Hola, usted!"
  | saludo (SOME name) = "¡Hola, " ^ name ^ "!";
```

Prueba:

- saludo (SOME "Juan"); (* Salida esperada: "¡Hola, Juan!" *)
- saludo NONE; (* Salida esperada: "¡Hola, usted!" *)

5.- repetir

Escriba una función repetir: list int * lista int -> lista int que dada una lista de enteros y otra lista de enteros no

negativos, repite los enteros en la primera lista de acuerdo con los números indicados por la segunda lista. Por

ejemplo: repetir ([1,2,3], [4,0,3]) = [1,1,1,1,3,3,3].

Solución:

```
fun repetir (_, []) = []
  | repetir ([], _) = []
  | repetir (x::xs, 0::ns) = repetir(xs, ns)
  | repetir (x::xs, n::ns) = x::repetir(x::xs, (n-1)::ns);
```

Prueba:

- repetir ([1,2,3], [4,0,3]); (* Salida esperada: [1,1,1,1,3,3,3] *)
- repetir ([5,2], [2,1]); (* Salida esperada: [5,5,2] *)
- repetir ([], [1,2,3]); (* Salida esperada: [] *)

6.- addOpt

Escriba una función addOpt: opción int * opción int -> optión int que, dados dos enteros "opcionales", los agrega

si ambos están presentes (devolviendo SOME de su suma), o devuelve NONE si al menos uno de los dos

argumentos es NONE .

Solución:

```
fun addOpt (SOME x, SOME y) = SOME(x + y)
  | addOpt (_, _) = NONE;
```

Prueba:

- addOpt (SOME 5, SOME 3); (* Salida esperada: SOME 8 *)
- addOpt (SOME 1, NONE); (* Salida esperada: NONE *)
- addOpt (NONE, NONE); (* Salida esperada: NONE *)

7.- agregarTodasOpt

Escriba una función agregarTodasOpt: int option list -> int option que, dada una lista de enteros "opcionales",

agrega los enteros que están allí (es decir, agrega todos los SOME i). Por ejemplo: addAllOpt ([SOME 1,

NONE, SOME 3]) = SOME 4. Si la lista no contiene SOME i, es decir, todos son NONE o la lista está vacía, la

función debería devolver NONE.

Solución:

```
fun agregarTodasOpt [] = NONE
  | agregarTodasOpt xs =
    let
      fun sum ([], acc) = acc
        | sum (NONE::xs, acc) = sum(xs, acc)
        | sum (SOME x::xs, NONE) = sum(xs, SOME x)
        | sum (SOME x::xs, SOME acc) = sum(xs, SOME(x + acc))
    in
      sum(xs, NONE)
```

```
    end;
```

Prueba:

- agregarTodasOpt [SOME 1, NONE, SOME 3]; (* Salida esperada: SOME 4 *)
- agregarTodasOpt [NONE, NONE]; (* Salida esperada: NONE *)
- agregarTodasOpt []; (* Salida esperada: NONE *)

8.- cualquiera

Escribe una función cualquiera : bool list -> bool que dada una lista de valores booleanos devuelve verdadero si

hay al menos uno de ellos que es verdadero, de lo contrario devuelve falso. (Si la lista está vacía, debería

devolver falso porque no hay verdadero).

Solución:

```
fun cualquiera [] = false
  | cualquiera (x::xs) = x orelse cualquiera xs;
```

Prueba:

- cualquiera [true, false, true]; (* Salida esperada: true *)
- cualquiera [false, false]; (* Salida esperada: false *)
- cualquiera []; (* Salida esperada: false *)

9.- todo

Escribe una función all : bool list -> bool que dada una lista de valores booleanos devuelve verdadero si todos

son verdaderos, de lo contrario devuelve falso. (Si la lista está vacía, debería devolver verdadero porque no hay

falso).

Solución:

```
fun todo [] = true
  | todo (x::xs) = x andalso todo xs;
```

Prueba:

- todo [true, true, true]; (* Salida esperada: true *)
- todo [true, false, true]; (* Salida esperada: false *)
- todo []; (* Salida esperada: true *)

10.- zip

Escribe una función zip: int list * int list -> int * int que dadas dos listas de enteros crea pares consecutivos y se detiene cuando una de las listas está vacía. Por ejemplo: zip ([1,2,3], [4, 6]) = [(1,4), (2,6)].

Solución:

```
fun zip ([], _) = []
  | zip (_, []) = []
  | zip (x::xs, y::ys) = (x,y)::zip(xs, ys);
```

Prueba:

- zip ([1,2,3], [4,6]); (* Salida esperada: [(1,4), (2,6)] *)
- zip ([1,2], [3,4,5]); (* Salida esperada: [(1,3), (2,4)] *)

- zip ([], [1,2,3]); (* Salida esperada: [] *)

Conclusiones:

- Las funciones demuestran el uso efectivo de recursión y pattern matching
- La complejidad aumenta gradualmente entre ejercicios
- El código mantiene el orden original de las fechas
- Las soluciones son concisas y funcionales

Recomendaciones:

1. Probar con casos límite (listas vacías)
2. Verificar el manejo de fechas inválidas
3. Considerar optimizaciones para listas grandes