Ejercicios de standard ML

M. Pilar Nivela

Report LSI–90–37

Ejercicios de Standard ML

Mª. Pilar Nivela Alós
Departamento LSI

1. Escribir las siguientes funciones ML:
   a) **fun** primo *n* = ... que calcule el *n*-ésimo número primo.
   b) **fun** raiz_ent *n* = ... que calcula la parte entera de la raiz cuadrada de un natural (estará en el intervalo [0, *n*] y puede usarse una búsqueda binaria general en un intervalo [a, b])
   c) **fun** Eratóstenes *n* = lista de todos los primos entre 2 y *n* usando el método de la criba de Eratóstenes.

2. a) ¿Qué hace la siguiente función?

```
fun prog(x) = let
                  fun loop(y) = if b(y) then loop(g(y)) else y
              in h(loop(f(x))) end
```

   Escribir un programa imperativo equivalente.
   b) Escribir en ML un programa equivalente al siguiente programa Pascal

```
y := f(x)
repeat
   y := g(y)
until b(y)
z := h(y)
```

3. ¿Qué calcula el siguiente programa?

```
- fun f(n) = if n = 0 then 1 else f(n-1) + g(n-1, n-1)
  and g(n, k) =  if k = 0 then 0 else f(n) + g(n, k-1) ;
- f(2) ;
```

4. Escribir funciones para
   a) quitar las repeticiones adyacentes de una lista
   b) concatenar dos listas
   c) aplanar una lista
   d) hallar la mediana de una lista de enteros (el entero de la lista que tiene tantos mayores como menores)
   e) extraer de una lista la sublista de los elementos en una posición par, es decir, que de la lista [5, 1, 6, 9, 7, 3] proporcione la lista [1, 9, 3].
   f) decidir si una lista es "prefijo" de otra, es decir, la lista [2, 5] es prefijo de [2, 5, 1, 7, 9]
   g) generar la lista de enteros desde *i* hasta *j*, esto es, la lista [i, i+1, ..., j-1, j]

5. a) Programar la función de orden superior *reduce* definida por

$$\text{reduce } f\ b\ [x_1, \ldots, x_n] = f\ x_1\ (f\ x_2\ (\ldots (f\ x_n\ b)\ldots))$$

b) Dar su tipo

c) Utilizando *reduce* definir funciones para:

- sumar una lista de enteros
- concatenar dos listas
- aplanar una lista de listas
- ordenar una lista por el método de inserción directa

6. a) Programar la función de orden superior *map* definida por

$$\text{map } f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n]$$

b) Dar su tipo

c) Definir la función *map* en términos de la función *reduce.*

7. a) ¿Qué hace la función *nosé* definida por

fun nosé a [ ] = [[a]]
| nosé a (x :: *l*) = (a ::(x :: *l*)) :: map (cons x) (nosé a *l*)

donde **fun** cons x *l* = x :: *l*?

b) Utilizar la función *nosé* para programar la función *perms* que devuelve la lista de todas la permutaciones de una lista dada.

8. a) Programar la función de orden superior *acumulate* definida por

$$\text{acumulate } f\ b\ [x_1, \ldots, x_n] = f\ (\ldots (f\ (f\ b\ x_1)\ x_2)\ \ldots)\ x_n$$

b) Dar su tipo

c) Comparar las secuencias de reducciones a que da lugar las siguientes evaluaciones

reduce mas 0 [1, 2, 3]

acumulate mas 0 [1, 2, 3]

siendo *mas* la función definida por **fun** mas x y = (x+y): int.

9. a) Programar la función de orden superior *filter* deinida por

$$\text{filter } f\ [x_1, \ldots, x_n] = [\text{ lista de los } x_i\ ,\ i = 1\ ..\ n, \text{ tales que } f\ x_i \text{ es true }]$$

b) Dar su tipo

c) Definir *filter* en función de *reduce.*

d) Utilizar la función *filter* para:

- calcular la lista de todos los divisores de un natural
- encontrar todos los primos en un intervalo (i, j)

10. La función *reduce* es una función potente para expresar tratamientos recursivos sobre listas. Sin embargo no podría usarse para diseñar una función *repetidos?* que indicara si hay o no elemetos repetidos en una lista. Una posible generalización de *reduce* que tiene en cuenta el resto de la lista sería

fun recur_list f b [] = b
| recur_list f b (x :: ls) = f x ls (recur_list f b ls)

a) Deducir el tipo de esta función.

b) Utilizarla para escribir la función *repetidos?*.

11. Programar:

a) la función de orden superior *zip* definida por

$$\text{zip f } [x_1, \ldots, x_n]\ [y_1, \ldots, y_n] = [\text{f } x_1\ y_1, \ldots, \text{f } x_n\ y_n]$$

y dar su tipo

b) la variante de *zip* que, cuando una lista es mayor en longitud que la otra, añade el resto de lista que sobra

c) la variante de *zip* que, cuando una lista es mayor en longitud que la otra, se olvida de lo que sobra

12. Usar las funciones del ejercicio anterior para programar

a) revonto $[x_1, \ldots, x_n]\ [y_1, \ldots, y_m] = [y_m, y_{m-1}, \ldots, y_1, x_1, \ldots, x_n]$

b) reverse $[x_1, \ldots, x_n] = [x_n, \ldots, x_1]$ (y en función de *revonto*)

c) existe? p x $[x_1, \ldots, x_n]$, que decide si en la lista $[x_1, \ldots, x_n]$ hay un elemento x_i que cumple la propiedad p, es decir, tal que p x_i = true

d)trans $[x_1, \ldots, x_n]\ [y_1, \ldots, y_n]$, la función *trans* de **FP**

e) cubos_dig x = lista de los enteros **n** menores que **x** tales que la suma de los cubos de los dígitos de **m** sea igual a **m**.

13. Escribir variantes de *reduce* y *acumulate* que operan sobre una función en lugar de operar sobre una lista de manera que puedan ser usadas para sumar los valores de una función en un rango dado y para encontrar el mínimo de una función en un rango dado.

14. a) Escribir una función para currificar funciones de dos argumentos. Esto es, dada una función f(x, y), **curry f x y** calcula lo mismo que f (x, y).

b) Deducir su tipo.

c) ¿Cuál es el tipo de **curry** ∘ **curry** ?. (El operador " ∘ " es el operador estandar de ML de composición de funciones).¿Podría utilizarse para la currificación de funciones de tres argumentos?

15. a) Escribir una función para descurrificar funciones de dos argumentos. Esto es, dada una función f x y, **uncurry f (x, y)** calcula lo mismo que f x y.

b) Deducir su tipo.

c) ¿Cuál es el tipo de **uncurry** ∘ **uncurry** ?. (El operador " ∘ " es el operador estándar de ML de composición de funciones). ¿Podría utilizarse para la descurrificación de funciones de tres argumentos?

16. Cuando *map* es seguida de *reduce*

$$\text{reduce g b (map f x)}$$

se hacen dos inspecciones o recorridos sobre la lista x, uno para calcular *map f x* y otro cuando se

aplica *reduce* sobre la lista resultante. Definir una función *reduce_map* que realiza ambas operaciones al mismo tiempo con un único recorrido de la lista.

17. Considerando un diccionario como una función de claves en valores, escribir funciones para crear un diccionario, modificar-añadir un par (clave, valor) y eliminar un par (clave, valor). Dar el tipo de estas funciones.

18. Escribir y dar el tipo de una función tal que dado un conjunto (lista) devuelva el conjunto de todos sus subconjuntos.

19. Definir, dando su tipo, las siguientes funciones currificadas de orden superior

a) **fun** burbuja menor = ... para ordenar una lista por el método de la burbuja

b) **fun** ord_inser menor = ... para ordenar una lista por el método de la inserción directa

c) **fun** quicksort menor = ... para ordenar una lista por el método quicksort

donde el único parámetro sea una función currificada *menor* que decide si su primer argumento es menor que el segundo.

20. Definir una función *merge_sort* que ordena una lista por mezcla de listas. La idea consiste en convertir la lista original en una lista de listas de longitud uno; en cada paso recursivo convertir la lista de listas ordenadas en curso en una lista de listas ordenadas más pequeña mezclándolas dos a dos.

21. Diseñar una función

ord_lex: (string)list * (string)list → bool

que compare lexicográficamente dos listas de *string*. Generalizarla de forma que *ord_lex* sea una intanciación de un orden lexicográfico polimórfico cualquiera.

22. Escribir un programa ML para el *juego de la vida*: se tiene un cuadrado dividido en *celdas*. Cada celda tiene ocho vecinas, excepto las celdas de los bordes si este cuadrado es finito.Sin embargo, puede considerarse también que el cuadrado es infinito en sus dos dimensiones. Una celda puede estar *viva* o *muerta*. Una *generación* es una configuración del cuadrado. A partir de una generaración se clacula la siguiente siguiendo las reglas:

- La siguiente generación es la unión de los *supervivientes* y los *recién nacidos* que se calculan simultáneamente a partir de las celdas vivas de la generación en curso.
- Una celda viva es *superviviente* en la siguiente generación si tiene dos o tres vecinas vivas en la generación en curso. En caso contrario la celda muere por superpoblación o soledad.
- Una celda muerta se convierte en *recién nacida*, esto es, una celda viva en la siguiente generación, si tiene exactamente tres vecinas vivas en la generación en curso.

23. Definir mediante el mecanismo **datatype** los tipos de datos

a) *pila*, con la operación de acceso al *top* de la pila y *pop* un elemento de la cima de de una pila

b) *cola*, con la operación de acceso al *frente* de la cola y *quitar* un elemento de una cola

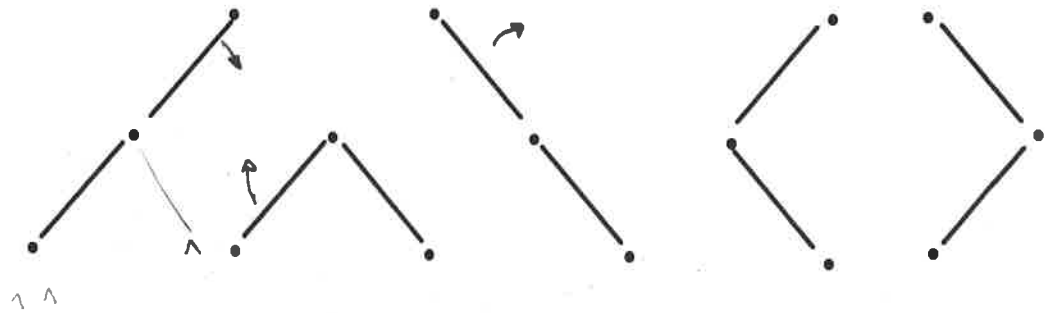
c) *cola con propridad* con las operaciones análogas a las anteriores

d) *arbol n-ario* (no etiquetado) con operaciones de acceso a los subárboles para n= 2, 3, ...

25. Dado el tipo de datos *árbol binario* definido por

datatype árbol_binario = vacío | enraizar **of** árbol_binario × árbol_binario

escribir una función que dado un natural *n* devuelva la lista de todos los árboles binarios con *n* nodos. Por ejemplo, todos los posibles árboles de 3 nodos de este tipo serían los siguientes:

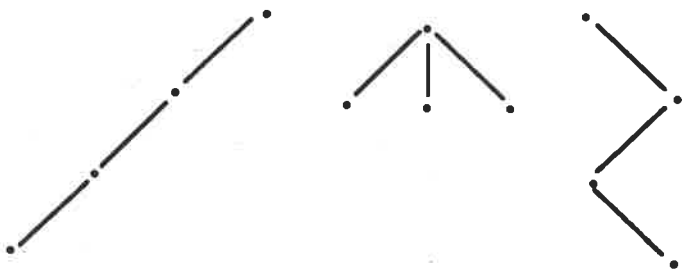


26. a) Definir completamente una función **particiones n x** que dado un natural **n** proporciona la lista de todas sus descomposiciones en **x** sumandos. Por ejemplo

particiones 3 2 = [[3, 0], [2, 1], [1, 2], [0, 3]]

b) Definir un **datatype** para árboles 3-arios no etiquetados (árboles en los que cada nodo tiene como máximo 3 hijos).

c) Aplicar **particiones** del apartado a) para programar una función que dado un natural **n** devuelve todos los 3-árboles con **n** nodos. Por ejemplo, para n = 4 algunos 3-árboles con cuatro nodos que deberán estar en la lista son:



etc, ...

27. El tipo de datos de los *árboles binarios con etiquetas en las hojas* puede definirse de la forma

datatype α árbol = vacío | hoja **of** α | § **of** α árbol x α árbol

infix §

a) Definir las operaciones habituales para manejar este tipo de datos: *subárbol_izq*, *subárbol_der*, *es_hoja?*, *dar_etiq_en_hoja*, ...

b) Definir las funciones

b1) hoja_más_izq: hoja que está más a la izquierda

b2) mismo_contorno: dados dos árboles, la función contesta si ambos tienen el mismo contorno, esto es, si tienen exactamente las mismas hojas de izquierda a derecha. Los árboles

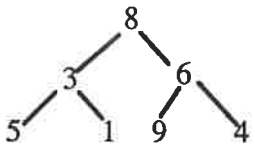
d) Definir una función análoga a la del apartado d) del ejercicio anterior.

e) Definir funciones para recorrer un árbol en preorden, postorden e inorden.

f) Definir una operación

niveles: α árbol $\rightarrow \alpha$ list

de recorrido por niveles de un árbol binario. Por ejemplo, dado el árbol



devuelve la lista [8, 3, 6, 5, 1, 9, 4].

g) Ordenar una lista por el método *tree_sort*. Este método consiste en transformar la lista en un árbol binario de búsqueda para recorrelo después en orden central (o inorden).

29. Los dos tipos de datos árbol de los ejercicios anteriores pueden considerarse casos particulares de

datatype (α, β) árbol = vacío | hoja **of** α | enraizar **of** (α, β) árbol $\times$ β $\times$ (α, β) árbol

donde α es el tipo de las etiquetas de las hojas y β el de las etiquetas de los nodos internos.

a) Definir, al igual que en los ejercicios anteriores, las operaciones habituales para manejar este tipo de datos.

b) Definir, al igual que en los ejercicios anteriores, operaciones de orden superior sobre estos árboles.

c) Este tipo de datos puede ser utilizado para representar expresiones aritméticas con operadores binarios como $(3+2)*(5+8)$. Escribir un programa ML para evaluar estas expresiones.

30. a) Definir mediante el mecanismo **datatype** el tipo de datos *árbol_general* en el que cada nodo tiene un número cualquiera de hijos.

b) Definir, al igual que en los ejercicios anteriores, las operaciones habituales para manejar este tipo de datos.

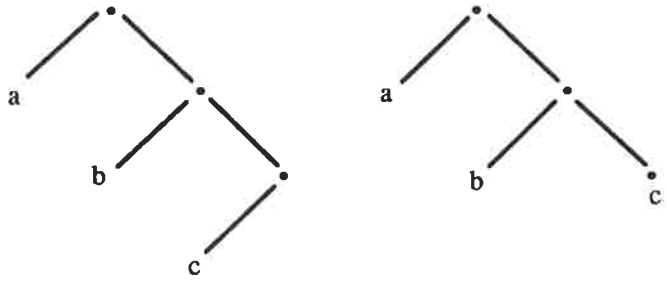
c) Definir, al igual que en los ejercicios anteriores, operaciones de orden superior sobre estos árboles.

d) Definir una operación para transformar un *árbol_general* en un *árbol_binario* (basarse en la representación *hijo izquierdo - hermano derecho*).

31. Definir mediante el mecanismo de **datatype** el tipo de datos *bosque*. Utilizarlo para definir el tipo de datos *árbol_general* del ejercicio anterior.

32. a) Definir es tipo de datos *grafo* mediante el mecanismo **datatype** con las operaciones habituales del tipo: *grafo_vacío:* $\rightarrow \alpha$ *grafo, añadir_arista:* $\alpha \times \alpha \rightarrow \alpha$ *grafo, quitar_arista:* $\alpha \times \alpha \rightarrow \alpha$ *grafo, adyacentes?:* $\alpha \times \alpha \rightarrow$ *bool*, ... Definir funciones para el recorrido en anchura y en profundidad de un grafo.

b) Mismas cuestiones para una definición del tipo *grafo* que use el tipo $\alpha \rightarrow \alpha$ list.



tienen el mismo contorno

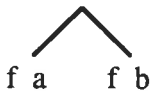
b3) sumar_hojas: suma todos las etiquetas de las hojas

b4) aplanar: lista de sus hojas

b5) reverse: imagen especular de un árbol

c) Definir, dando el tipo, una función *map* sobre este tipo de árboles de forma análoga a la función *map* sobre listas. Es decir, si *ar* es el árbol

entonces *map f ar* es el árbol



d) ¿Qué hace la función *nosé* ?

```
fun nosé g f (hoja x) = f x
  | nosé g f (a1 § a2) = g (nosé g f a1) (nosé g f a2)
```

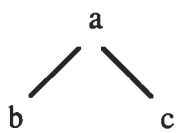
Dar el tipo.

e) Usar la función *nosé* para definir otras versiones de las funciones *hoja_más_izq*, *sumar_hojas*, *aplanar*, *reverse* y *map*.

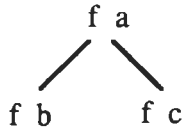
28. Definir mediante el mecanismo **datatype** el tipo de datos *α árbol* de los árboles binarios con etiquetas asociadas a todos los nodos.

a) Definir las operaciones habituales para manejar este tipo de datos: *raiz*, *hijo_izq*, *hijo_der*, *es_hoja?*, *iguales?*.

b) Definir, dando el tipo, una función *map* sobre este tipo de árboles de forma análoga a la definida en el ejercicio anterior. Es decir, si *ar* es el árbol



entonces *map f ar* es el árbol



c) Definir una función simplificar : ((int) list) árbol → (int) árbol que simplifica un árbol cambiando la lista de enteros de cada nodo por la suma de estos enteros

33. Definir mediante el mecanismo **abstype** implementaciones para los siguientes tipos de datos:

a) *polinomios en una variable* con las operaciones de creación de un polinomio, suma, producto y evaluación de un polinomio para un valor dado.

b) *conjuntos* con operaciones de conjunto vacío, añadir un elemento, quitar un elemento, pertenencia de un elemento, igualdad, unión, intersección y diferencia

c) *conjuntos de enteros*, usando árboles binarios de búsqueda, con las mismas operaciones del apartado anterior

d) *conjuntos infinitos* con las mismas operaciones de los apartados anteriores

e) *arrays* de una y dos dimensiomes con operaciones de crear array, añadir /actualizar un valor en una posición de un array, longitud del array, ...

f) *pilas* y *colas* con las operaciones habituales

g)*diccionario* (conjunto de pares *clave - valor*) mediante funciones de *claves* en *valores* con las operaciones de crear un diccionario, añadir/modificar un par *clave - valor* y consultar el *valor* asociado a una *clave*, ...