Algoritmos y Estructuras de Datos I

Apuntes suplementarios de las clases presenciales

Prof. Ing. Fernando A. Villar

2008
Rev.1 – 2009
Rev.2 - 2010

Breve Inicio en ML

Características

ML es un lenguaje de programación relativamente nuevo con unas características extremadamente interesantes.

ML *es esencialmente un lenguaje funcional*, esto significa que el modo básico de la programación esta basado en la definición y aplicación de funciones. Estas funciones pueden ser definidas por el usuario, como en los lenguajes convencionales, mediante la escritura de código para la función. Pero también es posible utilizar a las funciones como valores y calcular nuevas funciones a partir de ellas mediante operadores como la composición de funciones.

Es libre efectos colaterales. La consecuencia del estilo funcional es que el cálculo procede mediante la evaluación de expresiones, no mediante la asignación de variables. De todos modos hay formas de construir expresiones que afectan o cambian de manera permanente el valor de variables u otros objetos observables.

ML *soporta funciones de orden superior*. Funciones que toman a otras funciones como argumentos de manera rutinaria y con generalidad.

ML *soporta el polimorfismo*, lo cual es la habilidad de una función de tomar argumentos de varios tipos.

ML *recomienda la recursión* preferentemente a las iteraciones como bucles *for* y *while*, de uso común en lenguajes imperativos como Pascal y C.
La recursión permite expresar las ideas computacionales de manera mas clara.

Hay una manera muy fácil en **ML** de realizar *programación basada en reglas*, donde las acciones están basadas en reglas del tipo if-then.
La idea central es la construcción de acciones patrones, donde un valor es comparado a la vez con varios patrones. El primer patrón que concuerda provoca la ejecución de la acción asociada.
Ejemplo:

```
fun     factorial (0) = 1
|       factorial (n) = n * factorial(n-1);
```

En este caso podemos ver que se establecen dos patrones con acciones asociadas, cuando se invoca a la función *factorial* con un argumento igual a cero (0), la concordancia se establece con el primer patrón por lo tanto el valor de la funcion es 1. Para cualquier otro caso se disparará la ejecución de la segunda acción (llamada recursiva …).

ML es un *lenguaje fuertemente tipado*, significa que todos los valores y variables tienen un tipo de dato que puede ser determinado en tiempo de compilación (examinando el programa sin necesidad de ejecutarlo).
Esta característica es una ayuda muy importante en la depuración de programas, ya que le permite al compilador capturar varios errores en lugar de que aparezcan errores misteriosos mientras el programa se ejecuta.
Al contrario de los demás lenguajes fuertemente tipados, ML trata de inferir a través de las expresiones utilizadas, el tipo único que tiene cada variable, y sólo espera una declaración cuando le es imposible deducir el tipo.

Empezando en ML

Expresiones:

Por ejemplo en modo interactivo tipeamos:

```
> 1+2*3;
val it = 7 : int
```

(En itálica siempre me referiré a la respuesta del sistema ML)

Observando este ejemplo se desprenden dos puntos interesantes

- Una expresión debe estar seguida por un “punto y coma” para indicarle al intérprete que la instrucción ha terminado.

- La respuesta del intérprete ML a una expresión es:
  - La palabra *val* representando la palabra “valor”.
  - El nombre de variable *it*, la cual referencia a la expresión previa
  - Un signo igual (=)
  - El valor resultante de la expresión (7 en nuestro caso)
  - Un signo dos puntos, el cual en ML es el símbolo que asocia un valor con su tipo de dato
  - Una expresión que denota el tipo del valor. En nuestro ejemplo el valor de la expresión es un entero, por tal motivo el tipo *int* sigue a los dos puntos.

Constantes:

Como en cualquier otro lenguaje, las expresiones en ML estan compuestas por operandos y operadores. Estos operandos pueden ser tanto variables como constantes. A este punto no tiene sentido hablar de variables, asi que consideramos a todos los valores como constantes, ya sean literales o asociados a un identificador.
ML provee varios tipos de datos simples.

Enteros:
Los enteros se representan en ML de la misma manera que en cualquier otro lenguaje, con la excepción del signo negativo-
Un entero positivo es una secuencia de uno o mas digitos numericos como 0, 1234, 111111. Un entero negativo se representa adicionando el signo unario ~ delante de los digitos como ~1234.
También se los puede representar en notación hexadecimal, 0x19AF ó 0X56fb.

Ejemplos

```
> 0x1234
val it = 4660 : int
```

como se puede observar la respuesta del intérprete ML es la conversión a decimal del valor

```
> ~0XaA
val it = ~170 : int
```

Reales:
Los numeros reales se representan de manera convencional, a excepción del signo negativo ya visto.

Una constante del tipo real en **ML** consiste en:

1. Opcionalmente **~**
2. Una secuencia de uno o mas digitos
3. Uno de los siguientes elementos
   a) Un punto decimal seguido de uno o mas digitos
   b) La letra **E** ó **e**, opcionalmente ~, y uno o mas digitos

Ejemplos:

```
~123.0      (* -123.0 *)
3E~3        (* 0.003 *)
3.14e12     (* 3.14 x 10^12 *)
~3.14e12    (* -3.14 x 10^12 *)
```

Booleanos:
Hay dos valores booleanos **true** y **false** . Recordar que ML es sensible a la capitalización de caracteres, así que siempre deberán escribirse en minuscula.

Strings (Cadenas de caracteres):
Los valores del tipo string son secuencias de caracteres encerradas en comillas dobles como “hola” , “R2D2” o “1234”.
Algunos caracteres especiales de control estan representados por una secuencia de caracteres precedida por las barra hacia atrás (backslash).

\n	newline
\t	tab

Ejemplo:
El string “A\tB\tC\nD\tE\tF\n” se imprimiría de la siguiente forma

A	B	C
D	E	F

Caracteres
Lo mismo que en otros lenguajes de programación existe una diferencia entre el estring de longitud 1 y un carácter.
La representacion de un carácter en ML es poco usual consiste en el carácter # seguido por una cadena de caracteres de longitud 1. Así #”x” representa al carácter x.

Operadores
Los operadores aritméticos de ML son identicos a los de cualquier lenguaje, a excepcion del signo negativo el cual es notado como un tilde **~**.

El operador de concatenacion de strings es **^**. Asi la expresión “ Hola “ ^ “Gente” da como resultado la cadena “Hola Gente”.

Tuplas

Una tupla esta formada por una lista de ***dos o mas expresiones de distinto tipo***, separadas por comas y encerradas entre parentesis.
Tiene un aspecto similar al record de Pascal o la struct de C, pero los distintos valores se referencian por su posicion dentro de la tupla

val t = (“4, 5.0, “siete”);
*val t = (4, 5.0, “siete”) : int * real * string*

Accediendo a los componentes de la tupla

#1(t);
val it = 4 : int
#3(t);
val it = “siete” : string

Listas

ML provee una notación simple para las listas, ***cuyos elementos son todos del mismo tipo***, Tomamos una lista de elementos separados por comas y encerrados en corchete cuadrados.

[1, 2, 3];
val it = [1, 2, 3] : int list

Notacion y operadores

Lista vacia: la lista vacia o lista sin elementos esta representada tanto por la palabra *nil* como por el par de corchetes *[]*.

Head y Tail: Cualquier lista, a excepción de la lista vacia, esta compuesta de un *head*, el cual es el primer elemento de la lista, y un *tail*, que consiste en la lista de todos los elementos excepto el primero, en el mismo orden.

Ejemplo: Si L es la lista [2, 3, 4], el head de L es 2 y el tail de L es la lista [3, 4]. Si M es la lista [5], entonces el head de M es 5 y el tail de M es la lista vacia, [] ó nil .

Las funciones **hd()** y **tl()** extraen el *head* y el *tail,* respectivamente, de la lista que se proporcione como argumento

val L = [2, 3, 4];
hd(L);
val it = 2 : int

tl(L);
val it = [3, 4] : int list

Concatenación de listas
Mientras hd and tl, separan listas. Existen dos operadores que construyen listas.

El operador de ***concatenación*** para listas, @ , toma dos listas, cuyos elementos son del mismo tipo y produce una lista consistiendo en los elementos de la primera lista seguidos de los de la segunda.

[1, 2] @ [3, 4]
val it = [1, 2, 3, 4] : int list

El operador ***cons***, representado por ***::*** , toma un elemento y una lista del mismo tipo que este elemento, y produce una lista donde el primer operando es el head de la lista. Es decir agrega elementos simples por el head de una lista.

2 :: [3, 4];
val it = [2, 3, 4] : int list

5 :: []
val it = [5] : int list

Observación:
La expresión 1 :: 2 :: 3 :: nil, se poroduce el agrupamiento de derecha a izquierda.
Es decir

1 :: (2 :: (3 :: nil))
1 :: (2 :: [3])
1 :: [2, 3]
[1, 2, 3]

Pattern Matching

Uno de los origenes del poder de ML es la definicion de funciones sobre la base de patrones de sus parámetros. Esto reemplaza el concepto de "... si los parámetros satisfacen cierta condicion hacer una cosa sino hacer otra ". Cada patrón es una expresión con variables, y cuando el patrón se corresponde con los argumentos, a esas variables se les asignan los valores que corresponden, La misma variable luego puede ser usada en la expresión que define el valor de la función.

Ejemplo:
Uno de los patrones mas comunes es x :: xs. Ya que :: representa ***cons***, este patron se corresponderá con cualquier lista no vacía. En el match, **x** tendra el valor del elemento head de la lista y **xs** tendrá el valor del tail.

La forma general para una función definida por patrones involucar el símbolo |, el cual nos permite listar las formas alternativas para los argumentos de la función :

```
fun <identificador>  (<primer patron>)   = <primera expresión>
  | <identificador>  (<segundo patron>)  = <segunda expresión>
  |     …
  | <identificador>  (<ultimo patron> )  = <ultima expresión>;
```

El identificador debe ser siempre el mismo (es el nombre de la función), los valores producidos por todas las expresiones del lado derecho de los signos igual, **deben ser todas del mismo tipo de dato**. Asi mismo el tipo de dato de los patrones debe se el mismo, pero no necesariamente igual al de las expresiones, ya que una funcion va de un conjunto dominio a un conjunto imagen. (fn: 'a → 'b).

Ejemplo:

```
fun reverse ( nil ) = nil
|   reverse ( x:: xs ) = reverse(xs) @[x];
val 'a reverse = fn : 'a list -> 'a list

fun suma ( nil ) = 0
|   suma ( x :: xs ) = x + suma(xs);
val suma = fn : int list -> int
```

Bibliografía:
Jeffrey D. Ullman ,***"Elements of ML Programming"***, 1998

INTRODUCCIÓN AL TRABAJO CON LISTAS EN ML

FACTORIAL

```
fun    fact (0) = 1
      |fact (n ) = n * fact ( n - 1);
```

CUENTA ELEMENTOS EN UNA LISTA

```
fun cuenta ( [] ) = 0
|cuenta (x::xs) = 1 + cuenta(xs);
```

MINIMO

```
exception ListaVacia;

fun min ([]) = raise ListaVacia
|min ([x::nil) = x
|min (x::xs) = if x < hd(xs) then min(x::tl(xs))
               else min(xs);
```

(*otra version identica escrita de manera distinta*)

```
fun min ([]) = raise ListaVacia
|min ([x]) = x
|min (x::y::t) = if x < y then min(x::t)
                 else min(y::t);
```

MAXIMO

```
exception ListaVacia;

fun    max ([]) = raise ListaVacia
      |max (x::nil) = x
      |max (x::y::tl) = if x > y then max(x::tl)
                        else max (y::tl);
```

Ejercicios: Escriba las siguientes funciones, las cuales devolverán una lista
Take (n, L) devuelve una lista con los primeros "n" elementos de la lista "L"
Drop (n, L) devuelve una lista similar a "L" en la cual se han eliminado los primeros "n" elementos

WC() : CUENTA DE PALABRAS (EN EL INICIO DE LA PALABRA)

```
fun     wci([] ) = 0
      | wci( x::L) =
                        if null L then 0
                        else
                        let val y = hd(L)
                        in
                              if ((x = #" ") andalso (y <> #" ")) then
                                    wci (y::tl(L)) + 1

                              else  wci (y::tl(L))
                        end;
fun    wc( nil) = 0
      |wc(x::L) = if x <> #" " then wci(L) + 1
                  else wci(x::L);
```

Ejemplo de uso:
```
wc ( explode("Esta es una frase de prueba que consta de 11 palabras"));
```

Funciones Anónimas
Definicion IN-Line de un criterio como función

Cuando una expresión requiere un criterio (expresión lógica que se evalúa como Verdadero o Falso).

fn x => x < 10
fn x => x <> 0
fn x => p(x) *Donde p es cualquier función que devuelve V o F (**PREDICADO**)*

Ejercicios: Escriba las siguientes funciones
takewhile(p , L) : Devuelve una lista con los elementos de L que cumplen el predicado p
dropwhile(p, L) : Devuelve una lista que contiene los elementos de L que no cumplen el predicado p
delete (x , L) : Elimina al elemento x de la lista L

Funciones de Orden Superior

map : La función *map* toma una función ***F*** y una lista **[a1, a2, a3, … , an]** y produce como resultado una lista **[F(a1), F(a2), F(a3), … , F(an)]** . Incluida en la biblioteca standard , `map = fn:(´a ->´b) * ´a list -> ´b list`.

Ejemplo de implementación de una función map

```
fun mi_map(F, []) = []
| mi_map ( F, x::xs) = F(x):: mi_map(F, xs)


(*Ejemplo de utilización*)

fun cuadrado x = x*x;

val milista = [1.0, 2.0, 3.0, 4.0];

map(cuadrado, milista);
```

val it = [1.0, 4.0, 9.0, 16.0]: real list

```
(* O utilizando funciones in-line *)

map(fn x => x*x, milista);
```
val it = [1.0, 4.0, 9.0, 16.0]: real list

filter: La función *filter* toma un predicado ***P*** y una lista ***L***, y produce una lista con los elementos de ***L*** que satisfacen el predicado ***P***.
El modulo *List*, incluye una versión propia de filter, `List.filter = fn:(´a->bool)*´a list->´a list.`

Veamos como seria la forma de la función filter.

```
fun filter ( _ , [] ) = []
| filter ( P , x::xs) =
      if P(x) then x:: filter( P, xs)
      else filter(P, xs);
```

PROBLEMA DEL PALINDROMO

```
load "Array";

fun pali s =
      let
            val L = Array.fromList(filter (fn x => x <> #" ") (explode(s)) );

            fun aux(a, i, f) =
                  if i<f then
                        if Array.sub(a, i)=Array.sub(a, f) then aux(a, i+1, f-1)
                        else false
                  else true
      in
            aux(L, 0, Array.length(L)-1 )
      end
```

ejemplo:

```
pali "dabale arroz a la zorra el abad";
```

Notas:

explode(s): dado el string s, devuelve el string convertido en una lista de caracteres. Funcion inversa : implode(x::xs)

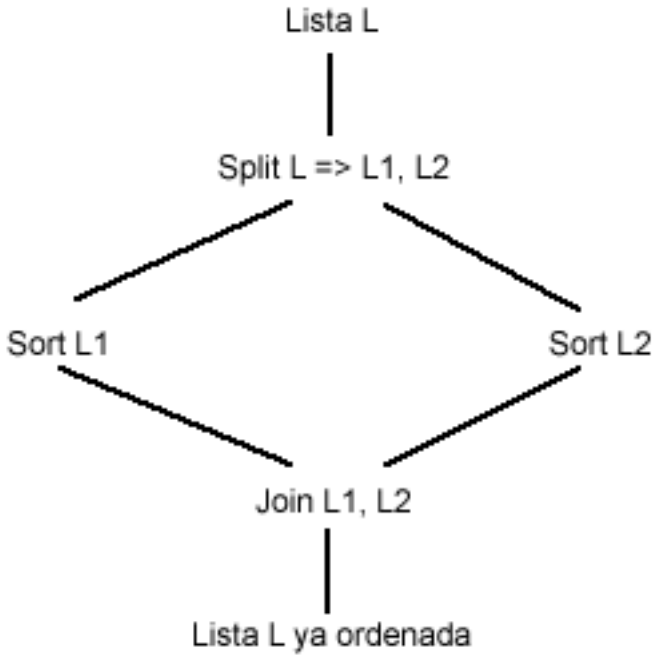
Para poder comparar se deben eliminar los espacios en blanco dentro de la frase, esto se lleva a cabo mediante la aplicación de filter.

Para poder recorrer la expresión de derecha a izquierda, simultaneamente con un recorrido de izquierda a derecha, es necesario convertir estos datos en un array. Para ello se utiliza Array.fromList.
Como sabemos si es palindromo?. Se debe comparar el primer carácter con el ultimo, el segundo con el anteultimo y asi sucesivamente.
En el codigo se ve como un indice que crece de izquierda a derecha a partir de cero (i), y otro indice que arranca desde la ultima posicion y cada vez que i se incrementa este indice se decrementa.

CLASIFICACION INTERNA

METODOS DE CLASIFICACION DE LISTAS

Los algoritmos de clasificación basados en comparación se basan en el mismo principio:
*"… dividir (**split**) la lista original L en dos listas L1 y L2, ordenar esas listas recursivamente y luego unir (**join**) esas listas para componer el resultado…"*



Dependiendo de cómo se definen esas operaciones y si cada operación resulta *"difícil" (**hard**)*o *"fácil" (**easy**)*, tenemos distintos algoritmos de clasificación con diferentes propiedades.
Los algoritmos pueden ser caracterizados por las siguientes propiedades:

Hard/easy splits : refiere a la complejidad de la operación de división, por ejemplo es mas fácil dividir una lista tomando el primer elemento que, encontrar el mínimo-maximo de una lista y luego dividirla en función de ellos.

Singleton/equal size list: refiere a la comparación entre la longitud de L1 y la de L2.

Hard/easy join: refiere a la complejidad de la operación de union de L1 y L2, por ejemplo es más fácil concatenar dos sublistas, que recorrerlas en paralelo y siempre tomar el menor de cada una.

	Singleton	**Equal Size**
Hard split/easy join	✓ Clasificación por Selección (*Selection sort*) ✓ *Heapsort*	✓ *Quicksort*
Easy split / hard join	✓ Clasificación por Inserción (*Insertion sort*) ✓ *Tree sort*	✓ *Mergesort*

- ✗ **Selection Sort** : Extrae del conjunto original la clave con valor mas bajo y la inserta en el resultado. Esto se repite hasta que el conjunto original está vacío.

- ✗ **Insertion Sort:** Toma el primer elemento y lo inserta en el lugar correcto dentro de la secuencia resultante.

- ✗ **QuickSort:** Selecciona al primer elemento como un *"pivot"* y divide al resto en dos subconjuntos: Uno conteniendo todas las claves que son menores o iguales al pivot y el otro conteniendo las claves mayores que el pivot. Esos subconjuntos se ordenan recursivamente y luego se unen con el pivot en el centro.
  Nota: para efectuar la división en dos listas, escribimos una función auxiliar llamada "partition".

- ✗ **MergeSort :** Divide la secuencia original en dos mitades que son recursivamente ordenadas y fusionadas para componer la secuencia ordenada.

Selection Sort

```
fun   ssort ([]) = []
|     ssort ( L ) =
            let
                  val m = minimo ( L )
            in
                  m :: (delete(m, L))
            end;
```

Insertion Sort

```
fun  inserta (x, [] ) = [x]
|    inserta( x, L as y::ys) =
        if x < y then x::L
        else y :: inserta(x, ys)

fun  isort ([]) = []
|    isort (x::xs ) =
        inserta(x, isort(xs));
```

QuickSort

fun partition (*p, l*) :

p : criterio de partición expresado bajo la forma de una función lógica *(predicado)*
l : Lista sobre la cual se aplicara el criterio

Resultado: 2 listas.
La primera todos los elementos de ***l*** que cumplen ***p***, la segunda todos los que no cumplen ***p***.

```
fun partition (_ , []) = ([] , [])
|     partition ( p , L) =
           ( List.filter(p ,L) , List.filter(fn x => not(p(x)), L)
```

O en forma recursiva

```
fun partition (_ , []) = ([] , [])
|     partition ( p , h::t) =
           let
              val(V , F) = partition (p,t)
           in
              If p(h) then (h::V , F) else (V, h::F)
           end;
```

Finalmente

```
fun  qsort ([]) = []
|    qsort (pivot::resto) =
        let
          val ( Menores, Mayores) = partition ( fn x => x < pivot , resto)
        in
          qsort(menores) @ [pivot] @ qsort(mayores)
        end;
```

MergeSort

Todo el proceso se desarrolla con dos acciones básicas split() y merge() :

```
fun   split [] = ([],[])
     |split [a] =([a],[])
     |split (x1::x2::resto) =
      let
          val (M,N)= split(resto)
      in
          (x1::M,x2::N)
        end;


fun    merge (a,  nil) = a
      |merge (nil, a) = a
      |merge (L1 as x::xs,L2 as y::ys) =
            if x < y then x::merge(xs,L2) else y::merge(L1,ys);
```

Finalmente la clasificación

```
fun msort1 [] = []
   |msort1 [a] = [a]
   |msort1 lista =
      let val (M,N) = split lista;
          val M = msort1 M;
          val N = msort1 N
      in
          merge(M,N)
      end;
```

Veamos como esto puede hacerse de manera mas compacta (sólo a modo de ejemplo)

```
fun unflat L = map (fn x => [x]) L;

(* Version mejorada utilizando fold (acumulador) *)


fun msort2 [] = []
    | msort2 [a] = [a]
    | msort2 L = foldr merge [] (unflat L);
```

Tipos de Datos Abstractos (TDAs)

Que es un TDA?. Trataremos de explicarlo de la manera mas sencilla posible, para ello pensemos primero en lo que es un tipo de dato.
Es una definicion de un dominio de valores y de un conjunto de operaciones que es posible realizar con ellos.
Por ejemplo, pensemos en el tipo *string*, es una secuencia de caracteres encerrada entre comillas, pero ¿ que operaciones es posible realizar con ellos?. Por ejemplo la division de dos strings no esta definida y en realidad no se que podria significar dividir entre si dos cadenas de caracteres.
Sin embargo dos cadenas pueden ser concatenadas (^) o pueden ser comparadas logicamente para establecer una relacion de orden entre ellas .

Un TDA entonces debe ser similar en comportamiento a un tipo simple ... sera una definicion de valores pòsibles y operaciones. Pero en este caso será una construccion de alto nivel (con esto queremos decir que es un concepto matematico de organización de datos o información tan complejo como se nos ocurra) que mantendra informacion de manera predeterminada y permitirá manipulaciones de estos datos.
Ejemplo: Un arbol binario, un grafo.

Como ML maneja los TDAs

ML posee un mecanismo poderoso para la creación de nuevos tipos de datos llamados *datatypes*. Una definición de datatype involucra dos clases de identificadores.

- Un `constructor de tipo`, que es el nombre del tipo de dato. El constructor de tipo se utiliza para construir tipos justamente como un nombre de tipo.
  Ej.:
  type ('d , 'r) *mapeo* = ('d, 'r) list;

- Uno o mas constructores de datos los cuales son identificadores utilizados como operadores para construir valores pertenecientes al nuevo tipo de dato.

Ahora vamos a dedicarnos a la forma mas general de una definicion de tipo de dato, donde:

1. Variables de tipo pueden ser usadas para parametrizar el nuevo tipo de dato.
2. Los constructores de datos pueden tomar argumentos

Así, la declaracion de un tipo de dato es:

datatype (<lista de parametros de tipo>) <identificador> =
<Expresión del primer constructor> |
<Expresión del segundo constructor> |
<Expresión del tercer constructor> |
.
.
.
<Expresión del ultimo constructor>

Entonces será la palabra reservada ***datatype*** seguida por una lista de cero o más argumentos de tipo. Los parentesis son opcionales si hay un parámetro de tipo e ilegales si hay cero argumentos de tipo. Luego sigue un identificador, que es el constructor de tipo, a continuacion un signo = y una lista de expresiones constructores de datos separados por una barra vertical.
Una expresión de constructor consiste en un nombre de constructor, la palabra reservada ***of*** y una expresión de tipo.

Ej: Banana **of** int

Esta expresion de constructor dice que los valores del tipo de dato, que esta siendo definido, pueden tener la forma de Banana(23) o mas generalmente Banana(i), donde i es un entero. Observemos puntos importantes al respecto:

- Notemos que el constructor de datos es usado para envolver el dato con parentesis. En una expresion como Banana(23) no sólo tenemos el dato entero 23, en su lugar estamos diciendo

a traves del constructor Banana algo sobre la forma, o tal vez el origen, o tal vez estemos diciendo que es un manojo de 23 bananas.

- Los constructores de datos son aplicados a los datos como si fueran funciones, pero no lo son. Preferimos usar constructores para formar expresiones simbólicas cuya apariencia es similar a la de una expresion que involucra la aplicación de una función.

- No confundir constructor de datos con el constructor de tipos. Los constructores de datos son utilizados para construir expresiones que son valores para el tipo en cuestión. Los constructores de tipos son utilizados en expresiones que denotan a los tipos mismos.

Veamos unos TDAs básicos las **pilas** y las **colas**.

Pilas (Stacks)

Una pila es una estructura de datos que almacena informacion con la particularidad que cuando quiero extraer un dato de la pila, extraeré el ultimo que ingresó (LIFO: Last In First Out).
Las pilas son de extrema utilidad en cualquier proceso que deba "volver sobre sus pasos" (la pila del Sist.Operativo es lo que permite que los programas sean recursivos por ejemplo).

Las operaciones basicas que se pueden realizar con una pila son

Poner: Ingresa un dato a la pila
Tope: Devuelve el valor del tope de la pila
Sacar: Saca el valor tope de la pila

La forma mas sencilla de implementar una pila es con listas. Veamos como sería una pila de enteros.

datatype Pila = Stack **of** int list;

Observacion:
Si en el entorno de ML escribimos lo siguiente:

```
datatype Pila = Stack of  int list;
New type names: =Pila
datatype Pila = (Pila,{con Stack : int list -> Pila})
con Stack = fn : int list -> Pila
```

luego si declaramos:

```
val t = Stack([2,7,1];
val t = Stack [2, 7, 1] : Pila
```

Veamos que nos esta diciendo que hay un nuevo nombre de tipo llamado "Pila". Mas alla de que luego indica cuales son los constructores del tipo Pila, ver que Stack es una funcion que transforma a una int list en Pila (*fn: int list -> Pila*).
Luego al declarar un valor t, vemos que usamos el constructor de datos Stack, pero el tipo del valor t creado es Pila.

Ahora desarrollemos las funciones necesarias antes mencionados.

```
datatype Pila = Stack of int list;
exception Pila_Vacia;

fun poner (x, Stack([])) = Stack([x])
|   poner (x, Stack(l)) = Stack (x::l);

fun tope(Stack([])) = raise Pila_Vacia
  | tope (Stack(x::xs)) = x;
```

```
fun sacar (Stack([])) = raise Pila_Vacia
  | sacar (Stack(x::xs)) = Stack(xs);
```

La pregunta que nos hacemos es ¿ Cada vez que necesito una pila, que maneja objetos de tipo distinto a los enteros, debo reescribir todas las declaraciones y funciones cambiando el tipo de lista manejada?

La respuesta es NO. Ya que puedo escribir una version generalizada de este datatype pasando un tipo de dato como argumento.

```
datatype 'a Pila = Stack of 'a list;

val s = ("Pepe", 33756221, "Pellegrini 333", 4223451);
```
*val s = ("Pepe", 33756221, "Pellegrini 333", 4223451) : string * int * string * int*
```
val miPila = Stack([]);
```
val 'a miPila = Stack[] : 'a Pila
```
val miPila = poner(s, miPila);
```
*val miPila = Stack [("Pepe", 33756221, "Pellegrini 333", 4223451)] : (string * int * string * int) Pila*

Colas (Queues)

Al igual que la pila, una cola es una estructura de datos que almacena informacion con la particularidad que cuando quiero extraer un dato de la cola, extraeré el primero que ingresó (FIFO: First In First Out).
Las colas son utilizadas para generar un lista de orden de “llegada”, como por ejemplo la cola de impresión.

Las funciones para manipular una cola son similares a las de una pila: Poner(), Siguiente(), Sacar()

Nota: Escribiremos la version generalizada del datatype.

```
datatype 'a Cola = Queue of 'a list;

exception Cola_Vacia;

fun poner (x, Queue([])) = Queue([x])
  |   poner (x, Queue(L)) = Queue( L @ [x]);

fun siguiente (Queue([])) = raise Cola_Vacia
  |   siguiente (Queue( x :: t )) = x;

fun sacar (Queue([])) = raise Cola_Vacia
  |   sacar (Queue( x :: t)) = Queue(t);
```

Estructuras de Diccionario

Una estructura de diccionario es un tipo de dato abstracto (TDA) cuyo propósito es mantener un conjunto de datos, los cuales son identificables por un valor en particular que denominamos *"Clave"*. Esta clave es la que sirve para localizar e identificar al elemento dentro del conjunto. (Así como en un diccionario utilizamos el término como clave de búsqueda y este tiene asociado otro tipo de información, su significado).
Las operaciones de mantenimiento de este conjunto comprenden tres operaciones básicas:

- Insertar
- Eliminar
- Buscar

Con respecto a la implementación de este TDA, nos resultan de interés abordar tres posibles soluciones, mediante listas ordenadas, Arboles Binarios de Búsqueda y mediante Tablas de dispersión (Hashing).

Vamos a analizar brevemente las ventajas y desventajas de cada estrategia.

Listas enlazadas: Ya hemos visto que las listas son muy fáciles de implementar y manejar, sólo nos hace falta considerar algunos aspectos de eficiencia (tiempo de procesamiento) en las operaciones de inserción y búsqueda.
El tiempo optimista de búsqueda de un elemento dentro de una lista, se da cuando este se encuentra en la entrada de la misma, en este caso se llega a él en un tiempo igual a 1 operación.

$T_{min} = 1$

El tiempo máximo de búsqueda o tiempo pesimista se da en el caso de tener que recorrer la totalidad de la lista "secuencialmente", lo cual para un conjunto de N miembros nos dará N operaciones.

$T_{max} = N$

Por tanto podríamos decir que el tiempo promedio de búsqueda, para una estructura secuencial como las listas, está dado por:

$T = (T_{min} + T_{max}) / 2 = (1 + N)/2$

Es decir el tiempo está dado por un polinomio de grado1 en N (N: cantidad de datos), en este caso diremos que el tiempo promedio de búsqueda (o inserción) es ***"del orden de N" u O(n)*.**

Arboles binarios: Sabemos que un árbol binario, es una forma de organizar datos. Esta organización está dada por una regla simple y recursiva que expresa: *"Dado un nodo con un valor determinado x, el subárbol izquierdo contiene nodos con valores inferiores a x y el subárbol derecho contiene valores mayores a x"*.
(Propiedad BST)

Si los valores que se insertaron en el árbol se proporcionaron con la suficiente aleatoriedad podemos pensar que el árbol se encuentra balanceado, esto significa que cuando nos paramos sobre la raíz del mismo, la profundidad del subárbol derecho es igual a la profundidad del subárbol izquierdo.
Veamos si podemos deducir los tiempos o cantidad de operaciones en la inserción/búsqueda de un elemento en el árbol.
Observar que la profundidad del árbol es directamente proporcional a la cantidad de operaciones (consulta de etiqueta de nodo y bifurcación de camino).
Sea K, la cantidad de operaciones necesarias y n la cantidad de datos (nodos) acomodados en estas K operaciones.

```
K = 0 , n = 1
K = 1 , n = 2
K = 2 , n = 4
.
.
.
K = i , n = 2^i (*)
```

Si el conjunto posee N datos, ¿Cuántas operaciones serian necesarias para acomodar estos datos en el árbol?

De (*) tendríamos

$$N = 2^T$$

Aplicando Log_2 a ambos miembros obtenemos:

$$Log_2 (N) = T * Log_2(2)$$

$$\mathbf{Log_2(N) = T}$$

Es decir el tiempo máximo que lleva acomodar N elementos en un árbol binario de búsqueda es aproximadamente proporcional al Log_2 de N.

Para tener una idea aproximada de lo que esto significa, consideremos lo siguiente. Comparemos cuanto es el tiempo que lleva acomodar en una lista ordenada y en un árbol binario de búsqueda un lote de 1024 (2^{10}) datos.

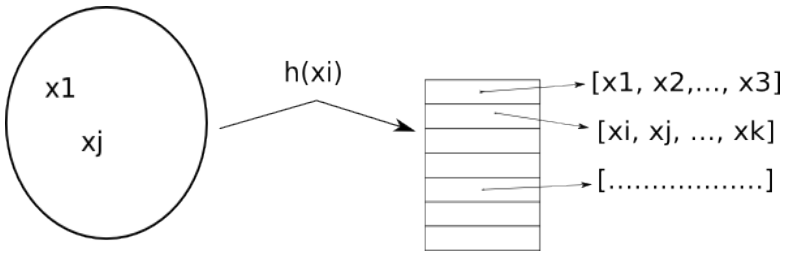
Lista ordenada: $(1 + 1024)/2 \approx 512$ operaciones
Árbol Binario de búsqueda: $Log_2 (1024) = Log_2 (2^{10}) = 10$ operaciones

Tabla de Dispersión (Hash Table): Las tablas de dispersión requieren un tiempo promedio constante para acceder a un elemento del conjunto. Con un diseño cuidadoso podemos conseguir un tiempo de acceso lo suficientemente pequeño. En el peor caso tendremos un tiempo proporcional al tamaño del conjunto, como las listas.
Debemos considerar dos tipos de dispersion: Abierta y Cerrada.

La dispersión abierta o externa permite que el conjunto sea guardado en un espacio virtualmente ilimitado.
La idea es particionar el potencialmente infinito conjunto de miembros en un numero finito de clases. Si tenemos **B** clases numeradas de **0** a **B-1**, luego tenemos una ***funcion de dispersión h()*** , tal que para cada **x** del tipo de dato del conjunto que se va a representar, **h(x)** es un entero de **0** a **B-1**. El valor de h(x) es naturalmente la clase a la x pertenece. Generalemente a x lo denominamos *clave* y a h(x) *valor de dispersion de x*.

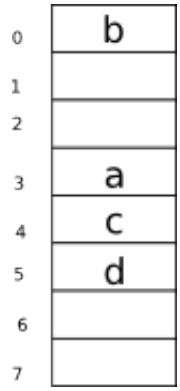
Consideremos que esas clases la representamos con un arreglo de B componentes, numeradas de 0 a B-1. En este caso el valor h(x), numero de clase corresponde a un numero de componente.
En cada componente almacenaremos una lista con todos los elementos del conjunto que pertenecen a la misma clase es decir todos los x / h(x) = i.

Si el conjunto a clasificar tiene N elementos, es nuestra esperanza que en cada clase haya N/B elementos.
Este deseo sólo se hará realidad si la función de dispersión tiene una distribución uniforme en el intervalo 0, B-1. El secreto del buen funcionamiento de la dispersión está en la elección de la función de dispersión h(x). Para conjuntos particulares pueden elegirse determinado tipo de funciones que tienen en cuenta características particulares del dominio.



La dispersión cerrada, coloca todos los elementos del conjunto en el arreglo mismo, no crea listas, esto significa que sólo podemos acomodar conjuntos de N = B.
Cuando a dos elementos les corresponde la misma clase h(x1) = h(xj) = m, se aplica lo que denominamos "*estrategia de redispersión*".
Si tratamos de colocar el elemento x en la posición h(x) y encontramos que ya está ocupada, ocurre lo que se denomina una colisión, y la estrategia de redispersión elige una secuencia de lugares alternativos h1(x), h2(x), h3(x)... dentro de la tabla donde debemos colocar a x. Se intenta cada una de las ubicaciones en secuencia hasta encontrar una "Vacia". Sino x NO PUEDE SER INSERTADO.

Inicialmente asumimos que la tabla está vacía, esto significa que en el arreglo tenemos guardado un valor especial “vacio”, que debe ser distinto a cualquiera de los datos que se quieren insertar.
Supongamos que queremos insertar los valores a, b, c y d cuyos valores de dispersión son h(a) = 3, h(b) = 0, h(c) = 4 y h(d) =3. Como estrategia de redispersión utilizaremos una forma sencilla llamada *“dispersión lineal”* , cuya ley sería $h_i(x) = (h(x) + i) \bmod B$). Si nuestra tabla en cuestion tiene 8 componentes, B = 8.
Vemos que al insertar “a”, se coloca en la posición 3, “b” en la cero y “c” en la 4.



Cuando tratamos de insertar “d” vemos que h(d)=3 esta ocupado, entonces luego probamos h1(d)=4 y tambien esta ocupado, h2(d) = 5 lo encontramos vacío entonces lo insertamos en esa posición.

El test de pertenencia de un elemento x (es lo quedenominamos buscar(x)) requiere que se examinen h(x),h1(x),h2(x), ..., hi(x), hasta que encontremos x o una posición “vacio”.
Primero supongamos que no se permiten las eliminaciones.
Digamos que buscando x, se da que h3(x) es la primera posicion que encontramos vacia, entonces sería imposible que x estuviera en h4(x), h5(x) o cualquier posicion subsiguiente en la secuencia, ya que esto sólo sería posible, si h3(x) estuvo lleno en el momento de insertar x.

Notar que si permitimos eliminaciones, nunca podremos estar seguros que x no está en el conjunto, al encontrar una cubeta vacía. Ya que ésta podria haber estado ocupada al momento de insertar x, y luego se eliminó el elemento que la ocupaba.

Para subsanar esta situacion, cuando eliminamos un elemento del conjunto, la posicion se marca con otro valor especial, “borrado”. Este valor tiene distinto significado según se esté insertando o buscando.
En el caso de una inserción la posición “vacio” y “borrado” tienen la misma significación, ya que significa que la posicion dentro del array está disponible.

En el caso de la búsqueda, una posicion marcada como “vacio” significa que nunca hubo nada en ese lugar, por lo tanto debe parar el proceso de busqueda. Por otro lado una posicion marcada con “borrado” significa que hubo elemento en esa posicion, por lo tanto a los efectos de la busqueda es como si estuviera ocupado, en consecuencia el proceso continua.

Implementación de Arboles Binarios de Búsqueda

En una primera aproximación utilizaremos un tipo de datos simple (char ó int) que almacenaremos en cada nodo del árbol, este valor funcionará como clave.
Primero debemos definir el tipo de dato necesario para construir el árbol utilizando **`datatype`**.
Para ello, nos basaremos en la definición recursiva del árbol.

Regla base: Un árbol vacío es un árbol binario.
Ley inductiva: Si $\mathbf{T_1}$ y $\mathbf{T_2}$ son árboles binarios, y **a** es un valor (tal que $\forall\ \mathbf{t_1} \in \mathbf{T_1} \wedge \forall \mathbf{t_2} \in \mathbf{T_2}$ se verifica que : $\mathbf{t_1} < \mathbf{a} < \mathbf{t_2}$). Entonces podemos conformar otro árbol binario llamado **T** creando un nuevo nodo que contenga al valor **a**, el subárbol izquierdo $\mathbf{T_1}$ y el subárbol derecho $\mathbf{T_2}$. El nuevo nodo creado es la raíz de **T**.

```
datatype  bintree = Vacio | Nodo of char * bintree * bintree
```

Búsqueda:
La búsqueda de un elemento x dentro del árbol comenzará siempre en la raíz. Si el dato a buscar es menor que el valor del nodo se continuará buscando en el subárbol izquierdo, si es mayor en el subárbol derecho.
Si llegamos a buscar en un árbol `Vacio`, la búsqueda ha fallado, no es posible encontrar a x en el árbol.
Si en un nodo con valor y, se verifica y=x, la búsqueda ha sido exitosa.
En este caso construiremos una función que retornara verdadero o falso según halle o no el valor indicado dentro del árbol.

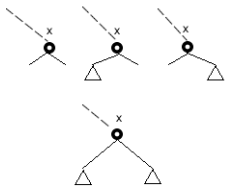
```
fun buscar ( _ , Vacio) = false
|   buscar ( x , Nodo(y , izq , der) ) =
      if x < y then buscar (x , izq)
      else x > y then buscar ( x, der)
      else true;(* no es mayor ni menor, por lo tanto es igual *)
```

Inserción:
La función insertar, retornará un árbol binario en el cual se ha agregado un nodo con el valor indicado como parámetro (el nodo agregado siempre estará en una hoja o nodo externo del árbol).

Caso Base: Para insertar **x** en un árbol **`Vacio`**, se devolverá un nodo que contendrá a **x** y tendrá los subárboles izquierdo y derecho **`Vacio`**.
Inducción: Para insertar **x** en un árbol con valor en la raíz y, donde $\mathbf{x} \neq \mathbf{y}$, insertar **x** recursivamente en el subárbol izquierdo si $\mathbf{x} < \mathbf{y}$ o insertar **x** recursivamente en el subárbol derecho si $\mathbf{x} > \mathbf{y}$. Devolver la copia del árbol con sus respectivos subárboles izquierdo o derecho modificados.

```
fun insertar ( x, Vacio) = Nodo(x, Vacio, Vacio)
|   insertar ( x , T as Nodo(y, izq, der)) =
      if x < y then Nodo(y, insertar(x, izq), der)
      else x > y then Nodo(y, izq, insertar(x,der))
      else T; (* No insertar x ya estaba en el árbol*)
```

Eliminación o Borrado:
Para escribir el proceso de borrado, debemos hacer algunas consideraciones con respecto a las distintas situaciones que nos podemos encontrar.
En este gráfico hemos representado las situaciones que se pueden dar con respecto a la eliminación del valor **x**, del árbol binario.
Como vemos en el caso (a), **`Nodo(x, Vacio, Vacio)`**, la eliminación de **x** puede hacerse sencillamente eliminando el nodo que lo contiene y devolviendo el valor **`Vacio`** al nodo precedente(marcado con líneas de puntos).



En los casos (b) y (c), **Nodo(x, izquierdo, Vacio)** o **Nodo (x, Vacio, derecho)**, vemos que la eliminación de **x** significa que su lugar será ocupado por el subárbol izquierdo o por el derecho respectivamente..

En el caso (d), vemos que no se puede hacer desaparecer al nodo físicamente, ya que alteraría la estructura del árbol , por lo tanto se debe eliminar **x** lógicamente del árbol, y la manera adecuada es encontrar un valor de reemplazo para ocupar el nodo de **x**.

Para ello observemos los siguiente, veamos el árbol que contiene a **x** en la raíz (**Nodo(x, T1, T2)**). Sean T1 y T2 sus subárboles, izquierdo y derecho respectivamente.

Sea $\mathbf{t_{1max}}$ el valor máximo contenido en T1 y sea $\mathbf{t_{2min}}$ el valor mínimo contenido en T2. En este caso se verifica que:

$$t_1 < t_{1max} < x < t_{2min} < t_2$$
$$\forall\ t_1 \in T_1 - \{t_{1max}\} \ \wedge \ t_2 \in T_2 - \{t_{2min}\}$$

En esta relación de orden sin alterar la estructura lógica de los datos vemos que:

$$t_1 < t_{1max} < t_{2min} < t_2$$

esto significa que todo elemento de T1, es menor que $\mathbf{t_{2min}}$, o que todo elemento de T2 es mayor que $\mathbf{t_{1max}}$.
Por lo tanto podemos usar a cualquiera de esos valores para reemplazar a **x**, sin que se altere la estructura del árbol, nos inclinaremos por la versión de encontrar el mínimo en el subárbol derecho. Para esto necesitaremos desarrollar una función que denominaremos `deleteMin` , la cual aplicada a un árbol, devuelve el valor mínimo almacenado en el mismo, y el árbol modificado donde este mínimo se ha eliminado del conjunto (la función devolverá una tupla).
El valor más chico siempre se encuentra buscándolo recursivamente hacia la izquierda, y lo hallaremos cuando se arribe a un nodo cuyo subárbol izquierdo es **Vacio**.

```
exception Arbol_Vacio;


fun deleteMin (Vacio) = raise Arbol_Vacio;
| deleteMin (Nodo(y, Vacio, derecho)) = (y, derecho)
| deleteMin (Nodo(y, izquierdo, derecho)) =
            let
                  val (w, L) = deleteMin(izquierdo)
            in
                  ( w, Nodo(y, L, derecho))
            end;
```

Ahora sí podemos analizar el algoritmo de eliminación de un valor que se encuentra en un árbol binario.

Caso Base: No se necesita hacer nada para eliminar a **x** de un árbol `Vacio`, retornar el árbol tal cual esta.

Para borrar a **x** de un árbol cuya raíz contiene al valor **x**, modificar el árbol manteniendo la propiedad BST.

- Si la raíz tiene al menos un subárbol `Vacio`, reemplazar todo el árbol por el otro subárbol.
- Si la raíz tiene dos subárboles no `Vacio`:
  Reemplazar a **x** por el resultado de la función `deleteMin`, aplicada al subárbol derecho y retornar un árbol que tiene a **x** reemplazado convenientemente, el subárbol izquierdo como estaba y el derecho modificado.

Inducción: Para borrar a **x** de un árbol cuya raíz tiene el valor **y**, donde **x≠y**, recursivamente borrar a **x** del subárbol izquierdo si **x<y** o del derecho si **x>y**.

```
fun eliminar (x, Vacio) = Vacio
|   eliminar (x, Nodo(y, izq, der)) =
        if x < y then eliminar( x, izq)
        else if x>y then eliminar(x, der)
        else   (* x = y *)
            case (izq, der) of
            (Vacio, d) => d |
            (i, Vacio) => i |
            (i, d) =>
                  let
                        val (z, r1) = deleteMin(d)
                  in
                        Nodo(z, i, r1)
                  end;
```

Bien, ahora que hemos analizado el funcionamiento del árbol binario de búsqueda, podemos afrontar la problemática de su especialización, es decir como haríamos un árbol que almacene en sus nodos algo más que un simple `char` o un `int`, un TDA.

Para ello, no sólo veremos cómo es la declaración del `datatype`, sino que además necesitamos proveer un mecanismo de comparación de estos TDA. El mecanismo en ML es similar a la creación de Templates en C++.

```
datatype ´tda bintree = Vacio | Nodo of ´tda * ´tda bintree * ´tda bintree;
```

Esta declaración define un tipo de dato `bintree` el cual almacena en sus nodos, además de sus subárboles izquierdo y derecho, una etiqueta de tipo `´tda`.

Para poder trabajar con un tipo genérico `´tda`, es necesario proveer una función que permita comparar a dos objetos del tipo `´tda`. Ya que seguramente los operadores (`>`, `<`, `=`), utilizados para tomar decisiones dentro del árbol, no estarán definidos para este tipo de dato.
Supongamos que la función utilizada para compararlos la denominemos simbólicamente `lt(x,y)` (less than), entonces se deberá modificar todas las funciones ya escritas. Éstas recibirán un argumento más, que será la función `lt`, y en el código se reemplazarán a los operadores de comparación por una llamada a esta función.
Ejemplo:

```
fun buscar ( x, Nodo(y,izq,der), lt) =
      if lt(x,y) then buscar(x, izq)
      else …
```

en este caso `x` e `y` son del tipo `´tda` , `izq` y `der` son del tipo `´tda bintree` y la función `lt` recibe dos instancias del tipo `´tda`, devolviendo `true` si `x` es menor que `y` o `false` si `x` es mayor que `y`.

Ejemplo probado:

```
datatype 'tda bintree = Vacio | Nodo of 'tda * 'tda bintree * 'tda bintree;


fun insertar ( x, Vacio, lt) = Nodo(x, Vacio, Vacio)
  | insertar ( x, T as Nodo(y, iz, de), lt) =
       if lt(x,y) then Nodo(y, insertar(x, iz, lt), de)
       else if lt(y,x) then Nodo(y,iz, insertar(x, de, lt))
       else T;
```

```
(* definimos un tipo Alumno que tiene nombre, domicilio y edad *)

datatype alumno = Nadie | registro of string * string * int;


(* una function que compara dos alumnos utilizando su nombre como clave *)

fun cmpAlu(Nadie,_) = true
  | cmpAlu(_, Nadie) = false
  | cmpAlu(registro(nombreX, domicilioX, edadX), registro(nombreY,domicilioY, edadY)) =
        if nombreX < nombreY then true
        else false;

(* Ahora intentemos insertar alumnos en el árbol *)

val al1 = registro("Felix Luna","Pellegrini 250 - Rosario", 19);
val T = insertar(al1, Vacio, cmpAlu);
val T = insertar(registro("Jose Maria Rosa","Rioja 1146 - Rosario", 21), T, cmpAlu);
val al2 = registro(“Damian Alvarez”,”Cordoba 1749 - Rosario”,22);
val T = insertar(al2, T, cmpAlu);
```