Datatypes

1. Una forma simple de declarar un nuevo tipo de dato

Podemos definir un tipo de dato nuevo llamado `fruta` que consista de tres valores posibles: `Pera`, `Naranja`, `Manzana`, de la siguiente manera:

```
- datatype fruta = Pera | Naranja | Manzana;
datatype fruta = Manzana | Naranja | Pera
```

Una vez definido este nuevo tipo de dato, podemos usarlo en una función u otra expresión. Por ejemplo, podríamos definir:

```
- fun esPera(x) = (x = Pera);
val esPera = fn : fruta ->bool
```

Esta función devuelve verdadero si se le pasa como argumento el valor `Pera` y falso para cualquier otro valor de tipo `fruta`. Es un error pasarle como argumento cualquier cosa que no sea alguno de los constructores del tipo de dato `fruta`.

Veamos las siguientes respuestas:

```
- esPera(Manzana);
val it = false : bool
- esPera(Naranja);
val it = false : bool
- esPera(Banana);
stdIn:7.8-7.14 Error: unbound variable or constructor: Banana
```

Esta última respuesta nos indica que `Banana` no es un argumento válido para la función `esPera`.

1.1. Usando expresiones constructoras para definir nuevos tipos de datos

La forma general de una declaración de tipo es:

`datatype (`<*lista-de-parámetros*>`)` <*identificador-nuevo-tipo*> `=`
<*1era-expresión-constructora*> `|`
<*2da-expresión-constructora*> `|`
...
<*últ-expresión-constructora*>

Una expresión constructora consiste en el nombre del constructor, la palabra `of` y una expresión de tipo.

Un ejemplo simple:

```
- datatype b = Banana of int
```
datatype b = Banana of int

Un ejemplo más complejo:

```
- datatype ('a, 'b) elemento =
            P of 'a * 'b |
            S of 'a;
```

```
datatype ('a,'b) elemento = P of 'a * 'b | S of 'a
```

Este tipo de dato representa la unión entre elementos simples y pares ordenados.

Supongamos ahora que tomamos una lista de `(string, int) elemento` y queremos sumar los enteros de las segundas componentes de aquellos elementos que tienen esta segunda componente.

La función `sumaLaLista` hace lo que queremos:

```
- fun sumaLaLista(nil) = 0
| sumaLaLista(S(x)::L) = sumaLaLista(L)
| sumaLaLista(P(x,y)::L) = y + sumaLaLista(L);
val sumaLaLista = fn : ('a,int) elemento list -> int
```

1.2. Tipo de dato recursivo

En muchos casos, necesitamos aplicar constructores de datos de manera recursiva para construir expresiones arbitrariamente grandes.

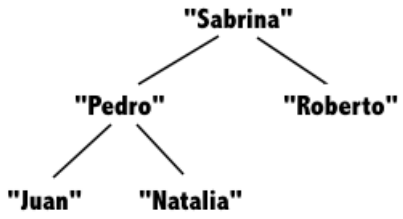
Un árbol binario etiquetado puede definirse recursivamente como sigue:

```
- datatype 'etiqueta arbolbin =
            Vacio |
            Nodo of 'etiqueta * 'etiqueta arbolbin * 'etiqueta arbolbin;
datatype 'a arbolbin = Nodo of 'a * 'a arbolbin * 'a arbolbin | Vacio
```

Un ejemplo de un valor de tipo `string arbolbin`:

```
- Nodo("Sabrina",
        Nodo("Pedro",
                Nodo("Juan", Vacio, Vacio),
                Nodo("Natalia", Vacio, Vacio)
        ),
        Nodo("Roberto", Vacio, Vacio)
);
```

Y representa el siguiente árbol binario:



Vamos a ver ejemplos de funciones que trabajan con este tipo de dato.

1. Defimos una función que tome un árbol `arbolbin` y un elemento `x` y diga si `x` aparece o no en el árbol.

   La función `pertenece` hace lo que queremos:

   ```
   - fun pertenece(Vacio,x) = false
   |     pertenece(Nodo(y,Z,W),x) = if(x=y) then true else (pertenece(Z,x) orelse pertenece(W,x));
   val pertenece = fn : ''a arbolbin * ''a ->bool
   ```

2. Escriba una función que tome un árbol `arbolbin` y un elemento `x` y devuelva la cantidad de veces que `x` aparece en el árbol.

   La función `contar` hace lo que queremos:

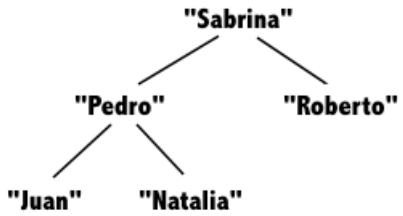
```
- fun contar(Vacio,x)= 0
|     contar(Nodo(y,Z,W),x) = if (x=y) then 1+contar(Z,x)+contar(W,x) else contar(Z,x)+contar(W,x);
val contar = fn : ''a arbolbin * ''a ->int
```

Podemos también definir árboles de tipo más general, es decir, árboles que acepten más de dos hijos por cada nodo.

Una definición para este otro tipo de árbol podría ser:

```
- datatype ('etiqueta) arbol1 =
            Nodo of 'etiqueta * 'etiqueta arbol1 list;
datatype 'a arbol1 = Nodo of 'a * 'a arbol1 list
```

Con este nuevo tipo de datos podemos también representar árboles binarios. Por lo tanto, podemos definir el árbol con el cual ya habíamos trabajado:



de la siguiente manera:

```
- Nodo("Sabrina", [Nodo("Pedro", [Nodo("Juan", [Vacio, Vacio]), Nodo("Natalia", [Vacio, Vacio])]),
                   Nodo("Roberto", [Vacio, Vacio])]
     );
```

Vamos a ver ejemplos de funciones que trabajan con este tipo de dato.

1. Escriba una función que tome un árbol `arbol1` y un elemento `x` y diga si `x` aparece o no en el árbol.

   La función `pertenece2` hace lo que queremos:

```
- fun pertenece2(Nodo(a,nil),x) = (a=x)
|     pertenece2(Nodo(a,t::ts),x) =
                    if (a=x) then true
                             else (pertenece2(t,x) orelse pertenece2(Nodo(a,ts),x));
```

2. Escriba una función que tome un árbol `arbol1` y un elemento `x` y devuelva la cantidad de veces que `x` aparece en el árbol.

   La función `contar2` hace lo que queremos:

```
- fun contar2(Nodo(a,nil),x) = if(x=a) then 1 else 0
|     contar2(Nodo(a,t::ts),x) = contar2(t,x) + contar2(Nodo(a,ts),x);
```