Standard ML

1. Tipos básicos y sus operaciones

1.1. Número entero: `int`

- **Valores:** `4`, `~2`, `78`, ...
- **Operadores:** `+`, `-`, `*`, `div`, `mod`, `=`, `<>`, `<`, `<=`, `>`, `>=`, ...

`div`: división entera

Ejemplos:
```
- 4 div 2;
val it = 2 : int
- 5 div 2;
val it = 2 : int
```

`mod`: resto de la división entera

Ejemplos:
```
- 4 mod 2;
val it = 0 : int
- 5 mod 2;
val it = 1 : int
```

1.2. Número real: `real`

- **Valores:** `3.14`, `2.17`, `0.1e6`, `~5.89`, ...
- **Operadores:** `+`, `-`, `*`, `/`, `<`, `<=`, `>`, `>=`, ...

`/`: división real

Ejemplos:
```
- 4.0 / 2.0;
val it = 2.0 : real
- 5.0 / 2.0;
val it = 2.5 : real
```

1.3. Caracter: `char`

- **Valores:** `#"a"`, `#"b"`, `#"\n"`, `#"\t"`, ...
- **Operadores:** `=`, `<>`, `<`, `<=`, `>`, `>=`, ...

1.4. Cadena de Caracteres: `string`

- **Valores:** `"HOLA"`, `""`, `"HO LA"`, ...
- **Operadores:** `^`, `size`, `=`, `<>`, `<`, `<=`, `>`, `>=`, ...

^: concatenación de cadenas de caracteres

Ejemplos:
```
- "Hola"^"Juan";
val it = "HolaJuan": string
- "Hola "^"Juan";
val it = "Hola Juan": string
```

size: tamaño de una cadena de caracteres

Ejemplos:
```
- size("Hola");
val it = 4 : int
- size("HolaJuan");
val it = 8 : int
- size("Hola Juan");
val it = 9 : int
```

=: igualdad de cadenas de caracteres

Ejemplos:
```
- "Hola"="Juan";
val it = false : bool
- "Juan"="Juan";
val it = true : bool
```

<>: desigualdad de cadenas de caracteres

Ejemplos:
```
- "Hola"<>"Juan";
val it = true : bool
- "Juan"<>"Juan";
val it = false : bool
```

<, <=, >, >=: orden lexicográfico de cadenas

Ejemplos:
```
- "abc"<"ab";
val it = false : bool
- "abc"<"adc";
val it = true : bool
- "abc"<"aabc";
val it = false : bool
- "ab"<="abc";
val it = true : bool
- "abc"<="abc";
val it = true : bool
```

1.5. Valor Booleano: `bool`

- **Valores:** `true` y `false`
- **Operadores:** `andalso`, `orelse`, `not`, `if` <*expresión0*> `then` <*expresión1*> `else` <*expresión2*>

orelse: 'o' lógico

Ejemplos:
```
- 1<2 orelse 3>4;
```

```
val it = true : bool
- 1>2 orelse 3>4;
val it = false : bool
```

`andalso`: 'y' lógico

Ejemplos:
```
- 1<2 andalso 3<4;
val it = true : bool
- 1<2 andalso 3>4;
val it = false : bool
```

`if...then...else...`: condicional

Usamos `if` <*expresión0*> `then` <*expresión1*> `else` <*expresión2*> cuando el valor depende de un valor booleano.

Tiene tipo `T` si <*expresión0*> es de tipo `bool` y tanto <*expresión1*> y <*expresión2*> son de tipo `T`.

Ejemplos:

```
- if 1<2 then "less" else "greater";
val it = "less": string
- if 1<2 then 0 else (1 div 0); (* los tipos son correctos, 1 div 0 no se evalúa *)
val it = 0 : int
- if 1<2 then 0 else "abc"; (* error: tipos diferentes en las dos ramificaciones *)
stdIn:4.1-4.25 Error: types of if branches do not agree [literal]
then branch: int
else branch: string
in expression:
if 1 <2 then 0 else "abc"
```

2. Tipos de datos más complejos

2.1. Tuplas

Las tuplas son elementos de un producto cartesiano.

Ejemplo:
```
- type fraction=int*int;
type fraction = int * int
- val foo:fraction=(44,100);
val foo = (44,100) : fraction
- #1(foo);
val it = 44 : int
- #2(foo);
val it = 100 : int
```

2.2. Listas

Los objetos en una lista deben ser del mismo tipo.

- **Valores:** `[1,2,3]`, `["dog","cat","mouse"]`, `nil`, `[]`, ...
- **Operadores:** `@` y `::`

@: concatenación de dos listas del mismo tipo

Ejemplos:
```
- [1,2,3]@[4,5];
val it = [1,2,3,4,5] : int list
- ["dog","cat"]@["mouse"];
val it = ["dog","cat","mouse"] : string list
```

:: : agrega un elemento al comienzo de una lista

Ejemplos:
```
- 1::[2,3];
val it = [1,2,3] : int list
- "dog"::["cat","mouse"];
val it = ["dog","cat","mouse"] : string list
```

3. Conversiones de tipos

3.1. Cómo convertir un `int` a un `real`

Usamos la función `real`.

Ejemplo:
```
- real(1);
val it = 1.0 : real
```

3.2. Cómo convertir un `real` a un `int`

Podemos usar las siguientes funciones: `floor`, `ceil`, `round` o `trunc`.

Ejemplos:

```
- floor(3.5);
val it = 3 : int
- floor(~3.5);
val it = ~4 : int
- floor(3.4);
val it = 3 : int
- floor(~3.6);
val it = ~4 : int

- ceil(3.5);
val it = 4 : int
- ceil(~3.5);
val it = ~3 : int
- ceil(3.4);
val it = 4 : int
- ceil(~3.6);
val it = ~3 : int

- round(3.5);
val it = 4 : int
- round(~3.5);
val it = ~4 : int
- round(3.4);
val it = 3 : int
```

```
- round(~3.6);
val it = ~4 : int

- trunc(3.5);
val it = 3 : int
- trunc(~3.5);
val it = ~3 : int
- trunc(3.4);
val it = 3 : int
- trunc(~3.6);
val it = ~3 : int
```

3.3. Cómo convertir un `char` a un `string`

Usamos la función `str`.

Ejemplo:

```
- str(#"a");
val it = "a": string
```

3.4. Cómo convertir un `char` a un `int` (entre 0 y 255)

Usamos la función `ord`.

Ejemplos:

```
- ord(#"a"); (* devuelve el código ASCII del caracter*)
val it = 97 : int
- ord(#"A");
val it = 65 : int
```

3.5. Cómo convertir un `int` (entre 0 y 255) a un `char`

Usamos la función `chr`.

Ejemplos:

```
- chr(97); (* convierte un código ASCII al caracter que corresponde *)
val it = #"a": char
- chr(65);
val it = #"A": char
```

3.6. Cómo convertir un `string` a un `list`

Usamos la función `explode`.

Ejemplos:

```
- explode("abcd");
val it = [#"a",#"b",#"c",#"d"] : char list
- explode("Hola");
val it = [#"H",#"o",#"l",#"a"] : char list
```

3.7. Cómo convertir un `list` a un `string`

Usamos la función `implode`.

Ejemplos:

```
- implode( [#"a",#"b",#"c",#"d"] );
```

```
val it = "abcd": string
- implode([#"H",#"o",#"l",#"a"]);
val it = "Hola": string
```

4. Funciones

4.1. Función `use`

La función `use: string -> unit` se usa para cargar y ejecutar el código fuente de un programa escrito en Standard ML usando el `string` que contiene el nombre del archivo (o el `path` en la sintaxis correspondiente al sistema operativo usado).

Supongamos contamos con un archivo llamado `foo.sml` que contiene el siguiente código:

```
fun double (x:int):int = 2 * x;
fun square (x:int):int = x * x;
fun power (x:int,y:int):int = if (y=0) then 1 else x * power (x,y-1);

double(4);
square(4);
power(2,3);
power(2,1);
```

Podemos entonces ejecutarlo de la siguiente manera:

```
- use "foo.sml";
[opening foo.sml]
val double = fn : int ->int
val square = fn : int ->int
val power = fn : int * int ->int
val it = 8 : int
val it = 16 : int
val it = 8 : int
val it = 2 : int
[closing foo.sml]
val it = () : unit
```

4.2. Cómo definimos funciones propias

`fun` <*identificador*> `(`<*lista-de-parámetros*>`) =` <*expresión*>`;`

Ejemplo 1: *Definimos una función que convierte una letra minúscula en la misma letra mayúscula.*

```
- fun mayuscula(c)= chr(ord(c)-32);
val mayuscula = fn : char ->char
- mayuscula(#"a");
val it = #"A": char
```

Ejemplo 2: *Definimos una función que calcula el cuadrado de un número real.*

```
- fun cuadrado(x:real) = x*x;
val cuadrado = fn : real ->real
```

Sino aclaramos que el parámetro es `x` es de tipo real, ML usa el tipo por defecto:

```
- fun cuadrado(x) = x*x;
val cuadrado = fn : int ->int
```

4.3. Funciones con más de un parámetro

```
- fun max3(a:real, b, c) = (* máximo de 3 reales *)
   if(a >b) then
      if (a >c) then a
      else c
   else
      if (b >c) then b
      else c;
val max3 = fn : real * real * real ->real
```

ML considera que es una función que toma un solo parámetro que es una tupla de 3 reales.

```
- val t=(1.0, 2.0, 3.0);
val t = (1.0,2.0,3.0) : real * real * real
- max3(t);
val it = 3.0 : real
```

4.4. Funciones que usan variables externas

```
- val x=3;
val x = 3 : int
- fun sumax(a)=a+x;
val sumax = fn : int ->int
- val x=10;
val x = 10 : int
- sumax(2); (* la definición de sumax está ligada a la primera x definida *)
val it = 5 : int
```

4.5. Funciones Recursivas

Una función recursiva es aquella que se llama a sí misma, de forma directa o indirecta.

En general, una función recursiva consiste en:

- Uno o más casos bases, aquellos casos donde los argumentos son pequeños y para los cuales se puede calcular el valor sin usar llamadas recursivas;
- Uno o más pasos inductivos, donde los argumentos no pueden ser manejados por los casos bases y llamamos recursivamente a la función una o más veces, con argumentos más pequeños.

Ejemplo: *Escribir una función que tome una lista y devuelva la lista inversa. Por ejemplo, que tome* `[1,2,3]` *y devuelva* `[3,2,1]`.

```
- fun inversa(L) =
   if L=nil then nil (* caso base *)
   else inversa (tl(L))@[hd(L)]; (* caso inductivo *)
val inversa = fn : "a list ->"a list
- inversa([1,2,3]);
val it = [3,2,1] : int list
```

Cómo evalúa ML `inversa([1,2,3])`*:*

```
inversa([1,2,3]) = inversa(tl[1,2,3]) @ [hd[1,2,3]]
                 = inversa([2,3]) @ [1]
                 = inversa(tl[2,3]) @ [hd[2,3]] @ [1]
                 = inversa([3]) @ [2] @ [1]
                 = inversa(tl[3]) @ [hd[3]] @ [2] @ [1]
                 = inversa([]) @ [3] @ [2] @ [1]
                 = [] @ [3] @ [2] @ [1]
                 = [3,2,1].
```

4.6. Funciones Mutuamente Recursivas

Ocasionalmente, necesitamos escribir dos o más funciones que son mutuamente recursivas, es decir, que cada una llama al menos una vez a otra función de ese grupo de funciones. En ML esto puede hacerse de forma bastante directa. Lo veremos a través del siguiente ejemplo.

Ejemplo: *Escribir una función que toma una lista L como argumento y produce una lista de elementos de L salteando uno de por medio.*

Naturalmente hay dos versiones posibles de esta función. Una que devuelve los elementos en las posiciones impares y otra que devuelve los elementos en las posisiciones pares.

Podemos definirlas de la siguiente manera:

```
- fun impares(L) =
        if L=nil then nil
        else hd(L)::pares(tl(L))
  and
       pares(L) =
        if L=nil then nil
        else impares(tl(L)];
```

En general, la forma de definir funciones mutuamente recursivas es la siguiente:

```
- fun
        <definición 1er función>
  and
        <definición 2da función>
  and
        ...
  and
        <definición n-ésima función>;
```

4.7. Patrones en la definición de funciones: `Pattern Matching`

Una de las mayores virtudes de ML es la definición de funciones usando patrones sobre sus parámetros.

Hasta ahora habíamos visto que la forma típica de definir funciones era: *"Si el argumento satisface una condición entonces hace tal cosa, sino otra cosa"*

Otra forma es definir funciones mostrando todos los patrones que puede cumplir el argumento y describiendo el valor que debe producir en cada caso.

Ejemplo: *Un patrón que usamos frecuentemente es* `x::xs`. *Como* `::` *representa el agregado de un elemento a una lista, este patrón concuerda con cualquier lista no vacía.*

La forma general de una función definida usando patrones usa el símbolo `|` que nos permite listar las distintas alternativas para los argumentos de una función como sigue:

`fun` <*identificador*> `(`<*1er_patron*>`) =` <*1era_expresion*>
`|` <*identificador*> `(`<*2do_patron*>`) =` <*2da_expresion*>
`|` ...
`|` <*identificador*> `(`<*ult_patron*>`) =` <*ult_expresion*>

Ejemplo: *Podemos redefinir la función que tome una lista y devuelva la lista inversa.*

```
fun inversa2(nil) = nil
| inversa2(x::xs) = inversa2(xs)@[x];
val inversa2 = fn : 'a list ->'a list
- inversa2[1,2,3];
val it = [3,2,1] : int list
```

Ejemplo: *Podemos también definir una función que sume dos enteros.*

```
fun suma(x,0)=x
| suma(x,y) = 1+ suma(x,y-1);
val suma = fn : int * int ->int
- suma(2,3);
val it = 5 : int
```