Apuntes-Examen-PP.pdf

casaalo

Paradigmas de Programación

2º Grado en Ingeniería Informática

Facultad de Informática
Universidad de A Coruña

Reservados todos los derechos.
No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

Apuntes de OCaml

1. Tipos de Datos Básicos

Enteros y Flotantes

```
let entero = 10
let flotante = 3.14
```

String

```
let saludo = "Hola, mundo"
```

Boolean

```
let es_verdadero = true
let es_falso = false
```

Operaciones:

- Negado: `not <e>`
- and: `<e1> && <e2>`
- or: `<e1> || <e2>`
- Igualdad: `<e1> = <e2>`

2. Operadores Aritméticos y Lógicos

Aritméticos

```
let suma = 5 + 3
let producto = 4 * 6
let division = 10 / 2
```

Lógicos

```
let y = true && false
let o = true || false
```

3. Comentarios

Usando `(* comentario *)`.

```
(* Esto es un comentario *)
let x = 5  (* Este es otro comentario *)
```

WUOLAH

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

4. If-Then-Else

Se utilizan `if`, `then` y `else` para decisiones condicionales.

```
let resultado = if x > 10 then "Mayor" else "Menor"
```

5. Operadores de Comparación

OCaml incluye operadores comunes de comparación, como `=`, `<>`, `>`, `<`, `>=`, `<=`.

```
let es_igual = (5 = 5)  (* True *)
let es_distinto = (5 <> 3)  (* True *)
```

6. Asignaciones:

NO SON VARIABLES, ya que son inmutables por defecto. Se declaran usando la palabra clave `let`.

```
let x = 5  (* Declara una variable 'x' con valor 5 *)
val x : int = 5
let y = 10 (* Declara una variable 'y' con valor 10 *)
val y : int = 10
```

Asignaciones anidadas

```
let x = 8 in
    let y = 2 in
        x + y;;
- : int = 10(*Si ahora queremos usar x e y van a tener sus valores originales, 5
y 10 respectivamente*)
```

Asignaciones en paralelo

```
# let x = x + y
  and y = x - y;;
```

Esto tiene utilidad en los casos en los que queramos hacer que esas asignaciones sean equivalentes aún cambiando el orden, de otra forma el resultado dependería de cuál se realizase primero

7. Funciones

WUOLAH

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

Definición de Funciones Básicas

Las funciones en OCaml se definen con la palabra clave `let` y pueden tener uno o más parámetros. El valor de retorno es implícito: se devuelve el valor de la última expresión evaluada dentro de la función.

```
function <p1> -> <exp1>
        |<p2> -> <exp2>
        ...
        |<pn> -> <expn>
```

Funciones totales: cada elemento tiene un elemento de destino.

Funciones parciales: no todos los elementos tienen imagen.

Funciones Recursivas

Hay que tener en cuenta la **RECURSIVIDAD TERMINAL**: hacer que la función recursiva sea lo que se ejecute al final de cada iteración.

```
let rec factorial n =
    if n = 0 then 1
    else n*(factorial (n-1)) (* val factorial : int -> int = <fun> *)
let rec potencia n x =
    if n = 0 then
    1.0
    else x*. (potencia (n-1) x) (* val potencia : int -> float -> float = <fun>
*)
factorial 10 (* - : int = 3628800 *)
potencia 4 5.0 (* - : float = 625. *)
```

La palabra clave `rec` es necesaria para definir funciones recursivas en OCaml.

Funciones Parciales(Currificación)

En OCaml, las funciones son **curried** por defecto. Esto significa que una función que toma múltiples parámetros puede ser llamada con uno solo, devolviendo otra función que espera los parámetros restantes.

```
let suma x y = x + y (* val suma : int -> int -> int = <fun> *)
let suma' = fun x -> (fun y -> x + y) (* val suma' : int -> int -> int = <fun> *)
let suma'' x = fun y -> x + y (* val suma'' : int -> int -> int = <fun> *)
```

Parcialización

```
let siguiente = suma 1 (* val siguiente : int -> int = <fun> *)
siguiente 4 (* - : int = 5 *)
```

WUOLAH

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

8. Patrones

Match with

El `match` es uno de los elementos más poderosos de OCaml. Permite hacer comparaciones complejas.

```
match expr with
    patron_1 -> expr1
    | patron_2 -> expr2
    ...
    | patron_n -> exprn
```

Definiciones con match with

```
let rec fib n =
    match n with
        0 -> 0
        |1 -> 1
        |n -> fib (n-1) + fib (n-2) (* val fib : int -> int = <fun> *)
fib 10 (* - : int = 55 *)
```

Asignaciones con patrones

```
let x = (1,'a') (* val x : int * char = (1, 'a') *)
let izq,_ = x (* val izq : int = 1 *)
let _,der = x (* val der : char = 'a'*)
```

9. Tuplas

Las **tuplas** son estructuras de datos que permiten agrupar varios elementos de diferentes tipos en un solo contenedor. Las tuplas no tienen un tamaño fijo ni una estructura específica como las listas.

Definición

Las tuplas se definen usando paréntesis `( )` y sus elementos están separados por comas.

```
let tupla = (1, "Hola", 3.14)
```

Acceder a los elementos de una tupla

Puedes acceder a los elementos de una tupla destructurando la tupla.

```
let (a, b, c) = tupla (* a = 1, b = "Hola", c = 3.14 *)
```

WUOLAH

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

Limitaciones de las tuplas

Las tuplas son útiles para agrupar elementos, pero no puedes cambiar su tamaño ni acceder a elementos por índice de manera directa como en las listas.

Operaciones

- `fst` : → devuelve el primer componente.
- `snd` → devuelve el segundo componente.

```
let p = 1, "Hola" (* val p : int * string = (1, "Hola") *)
fst p (* - : int = 1 *)
snd p (* - : string = "Hola" *)
```

10. Listas

Las listas se definen con corchetes `[]` y sus elementos se separan por puntos y coma.

Lista de Enteros

```
let lista_enteros = [1; 2; 3; 4; 5]
```

Lista de Cadenas

```
let lista_cadenas = ["Hola"; "Mundo"; "OCaml"]
```

Operaciones con Listas

OCaml proporciona varias funciones de la biblioteca estándar para trabajar con listas.

```
let lista = [1; 2; 3; 4; 5]
```

- `List.hd` : Devuelve el primer elemento de la lista. Si la lista está vacía, lanza una excepción `Failure`

```
let primero = List.hd [1; 2; 3] (* primero es 1 *)
```

- `List.tl` : Devuelve la lista sin el primer elemento. Si la lista está vacía, lanza una excepción `Failure` .

```
let resto = List.tl [1; 2; 3] (* resto es [2; 3] *)
```

- `List.length` : Devuelve la longitud de la lista, nº de elementos que contiene.

```
let longitud = List.length lista (* longitud es 5 *)
```

- `List.compare_lengths` : Compara el tamaño de 2 listas (-1, 0 o 1).

WUOLAH

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

```
let list1 = [1; 2; 3; 4]
let list2 = [1; 2; 3]
let comparation = List.compare_lengths list1 list2 (* comparation es 1 *)
```

- `List.nth`: Devuelve el elemento enésimo de la lista, las posiciones empiezan en 0.

```
List.nth [1; 2; 4; 8; 16] 3 (* int = 8 *)
```

- `List.append`: Concatena 2 listas, .

```
let list1 = [1; 2; 3; 4]
let list2 = [5; 6; 7]
List.append list1 list2 (* - : int list = [1; 2; 3; 4; 5; 6; 7] )
```

- `List.rev_append`: concatena 2 listas dándole la vuelta a la primera.

```
let list1 = [1; 2; 3; 4]
let list2 = [1; 2; 3]
List.rev_append list1 list2 (* - : int list = [4; 3; 2; 1; 1; 2; 3] *)
```

- `List.concat` o `List.flatten`: Convierte una lista de listas en una lista plana (una sola lista con todos los elementos).

```
let list = [[1; 2; 3;]; [4; 5; 6; 7]; [8; 9]; [0]]
List.concat list (* - : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 0]*)
List.flatten list (* - : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 0]*)
```

- `List.rev`: Devuelve la lista invertida.

```
let lista_invertida = List.rev [1; 2; 3] (* lista_invertida es [3; 2; 1] *)
```

- `List.map`: Aplica una función a cada elemento de la lista, generando una nueva lista con los resultados.

```
let lista_doblada = List.map (fun x -> x * 2) [1; 2; 3] (* lista_doblada es
[2; 4; 6] *)
```

- `List.filter`: devuelve todos los elementos de la lista que cumplen un predicado.

```
let lista_filtrada = List.filter (fun x -> x > 2) [1; 2; 3; 4] (*
lista_filtrada es [3; 4] *)
```

- `List.for_all`: Devuelve true si todos los elementos de la lista satisfacen una condición o false si no.

```
List.for_all (fun x -> x > 2) [1; 2; 3; 4] (* - : bool = false *)
List.for_all (fun x -> x > 2) [3; 4; 5; 6] (* - : bool = true *)
```

- `List.exists`: Comprueba si al menos 1 elemento satisface la condición.

WUOLAH

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

```
List.exists (fun x -> x < 2) [1; 2; 3; 4] (* - : bool = true *)
List.exists (fun x -> x < 2) [4; 3; 2] (* - : bool = false *)
```

- `List.find`: Devuelve el primer elemento de la lista que satisfaga una condición.

```
List.find (fun x -> x > 2) [1; 2; 3; 4; 5; 6] (* - : int = 3 *)
```

- `List.fold_left`: Realiza una acumulación de izquierda a derecha sobre los elementos de la lista, utilizando una función de acumulación. Empieza desde el primer elemento de la lista.

```
let suma = List.fold_left (fun acc x -> acc + x) 0 [1; 2; 3; 4] (* suma es 10
*)
```

- `List.fold_right`: Realiza una acumulación de derecha a izquierda sobre los elementos de la lista. Es similar a `fold_left`, pero procesa los elementos de la lista en orden inverso.

```
let suma_derecha = List.fold_right (fun x acc -> x + acc) [1; 2; 3; 4] 0 (*
suma_derecha es 10 *)
```

- `List.mem`: Devuelve `true` si un elemento está presente en la lista.

```
let contiene_3 = List.mem 3 [1; 2; 3; 4] (* contiene_3 es true *)
```

- `List.split`: Divide una lista de tuplas en dos listas. La primera lista contiene los primeros elementos de las tuplas, y la segunda lista contiene los segundos elementos.

```
let lista_de_tuplas = [(1, "uno"); (2, "dos"); (3, "tres")]
let (numeros, palabras) = List.split lista_de_tuplas (* numeros es [1; 2; 3],
palabras es ["uno"; "dos"; "tres"] *)
```

- `List.combine`: Combina dos listas en una lista de tuplas, donde el primer elemento de la primera lista se empareja con el primer elemento de la segunda lista, el segundo con el segundo, y así sucesivamente.

```
let lista1 = [1; 2; 3]
let lista2 = ["uno"; "dos"; "tres"]
let combinada = List.combine lista1 lista2 (* combinada es [(1, "uno"); (2,
"dos"); (3, "tres")] *)
```

- `List.sort`: Ordena los elementos de una lista utilizando una función de comparación. Por defecto, ordena de menor a mayor.

```
let nombres = ["Carlos"; "Ana"; "Pedro"; "Beatriz"]
let nombres_ordenados = List.sort String.compare nombres (* val
nombres_ordenados : String.t list = ["Ana"; "Beatriz"; "Carlos"; "Pedro"] *)
```

- Concatenar listas: `@`.

WUOLAH

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

```
let lista1 = [1; 2; 3; 4; 5]
let lista2 = [6; 7; 8; 9; 0]
let lista = (lista1 @ lista2) (lista = [1; 2; 3; 4; 5; 6; 7; 8; 9; 0])
```

- Añadir elementos a la lista `::`.

```
let lista = [1; 2; 3; 4; 5]
0 :: lista (* lista = [0; 1; 2; 3; 4; 5] *)
```

11. Definiciones de Tipos de Datos

- Ejemplo de **árbol binario**:

```
type 'a arbol =
Hoja
| Nodo of 'a * 'a arbol * 'a arbol;;
```

- **Expresiones aritméticas**

```
type expression =
    Var of string
    | Const of int
    | Add of expression * expression
    | Mul of expression * expression
let expr1 = Add(Mul(Const 2,Var "x"),Var "y")
```

- **Fórmulas**

```
type ('a)formula =
    False
    | True
    | Not of ('a)formula
    | And of ('a)formula * ('a)formula
    | Or of ('a)formula * ('a)formula;;
```

- **Registros**

  En otros lenguajes son conocidos como **structs**.

```
type persona = {
    nombre : string;
    apellido : string;
    edad : int
}
let nombre = {nombre = n} = n
let nombre = p.nombre
```

WUOLAH

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

12. Datos importantes de Ocaml

- Los nombres de los valores siempre van en minúsculas.
- Los nombres de los módulos siempre van en mayúsculas.
- **Tiempos**: Sys.time(): devuelve el tiempo consumido por el CPU en segundos.
- int → int → int = int → (int → int)
- x+1: succ
- x-1: pred
- `()`: unit, similar a void en c.

WUOLAH

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.