Guia Completo de OCaml

1. Tipos Básicos e Declarações

OCaml é **fortemente tipado** e usa **type inference** (deduz tipos automaticamente).

```ocaml
(* Inteiros *)
let x = 5;;
let y = x + 3;;

(* Floats - repara que usa . para operações *)
let pi = 3.14159;;
let result = pi *. 2.0;;  (* .* é multiplicação de floats *)

(* Strings *)
let name = "OCaml";;
let greeting = "Hello " ^ name;;  (* ^ concatena strings *)

(* Booleans *)
let is_true = true;;
let is_false = false;;

(* Type annotations (opcional, mas útil) *)
let age : int = 25;;
let height : float = 1.75;;
```

Operadores importantes:

- Inteiros: `+`, `-`, `*`, `/`, `mod`
- Floats: `+.`, `-.`, `*.`, `/.`
- Comparação: `=`, `<>`, `<`, `>`, `<=`, `>=`

2. Listas (Crucial para Autómatos!)

Listas são a estrutura de dados mais importante em OCaml:

ocaml

```ocaml
(* Criar listas *)
let empty_list = [];;
let numbers = [1; 2; 3; 4; 5];;
let mixed = [1; 2; 3];;  (* Todos do mesmo tipo *)

(* Construir com :: (cons - "head :: tail") *)
let list1 = 1 :: [2; 3; 4];;  (* Resultado: [1; 2; 3; 4] *)
let list2 = 1 :: 2 :: 3 :: [];;  (* Resultado: [1; 2; 3] *)

(* Concatenar listas com @ *)
let combined = [1; 2] @ [3; 4];;  (* Resultado: [1; 2; 3; 4] *)

(* Pattern matching em listas *)
let head_tail list =
  match list with
  | [] -> "Lista vazia"
  | head :: tail ->
      Printf.sprintf "Cabeça: %d, Cauda: %s" head (string_of_int (List.length tail));;

(* Funções de lista úteis *)
let length = List.length [1; 2; 3];;  (* 3 *)
let reversed = List.rev [1; 2; 3];;  (* [3; 2; 1] *)
let mapped = List.map (fun x -> x * 2) [1; 2; 3];;  (* [2; 4; 6] *)
let filtered = List.filter (fun x -> x > 2) [1; 2; 3; 4];;  (* [3; 4] *)
let folded = List.fold_left (fun acc x -> acc + x) 0 [1; 2; 3];;  (* 6 *)
```

3. Tuples (Duplas, Triplas, etc.)

Diferentes do listas - podem ter tipos diferentes:

ocaml

```ocaml
let pair = (1, "hello");;  (* (int, string) *)
let triple = (5, 3.14, true);;  (* (int, float, bool) *)

(* Desempacotar tuples *)
let (x, y) = (10, 20);;
let (a, b, c) = (1, 2.5, "text");;

(* Aceder elementos (menos comum, usar pattern matching é melhor) *)
let first = fst pair;;  (* 1 *)
let second = snd pair;;  (* "hello" *)
```

4. Pattern Matching (Muito Importante!)

É como `switch` mas muito mais poderoso:

ocaml

```ocaml
let classify x =
  match x with
  | 0 -> "Zero"
  | 1 -> "Um"
  | n when n > 0 -> Printf.sprintf "Positivo: %d" n
  | _ -> "Negativo";;

(* Com listas *)
let describe_list lst =
  match lst with
  | [] -> "Vazia"
  | [x] -> Printf.sprintf "Um elemento: %d" x
  | [x; y] -> Printf.sprintf "Dois: %d e %d" x y
  | head :: tail -> Printf.sprintf "Começa com %d, tem %d elementos" head (List.length tail);;

(* Com tuples *)
let process_pair p =
  match p with
  | (0, y) -> Printf.sprintf "X é zero, Y é %d" y
  | (x, y) when x = y -> "Iguais"
  | (x, y) -> Printf.sprintf "Diferentes: %d, %d" x y;;
```

5. Funções e Recursão

ocaml

```ocaml
(* Função simples *)
let add x y = x + y;;

(* Função com type annotation *)
let multiply (x : int) (y : int) : int = x * y;;

(* Função lambda/anónima *)
let double = fun x -> x * 2;;
let apply_twice f x = f (f x);;

(* Recursão - factorial *)
let rec factorial n =
  if n <= 1 then 1
  else n * factorial (n - 1);;

(* Recursão com listas - soma *)
let rec sum_list lst =
  match lst with
  | [] -> 0
  | head :: tail -> head + sum_list tail;;

(* Recursão com listas - filtrar *)
let rec filter_positives lst =
  match lst with
  | [] -> []
  | head :: tail when head > 0 -> head :: filter_positives tail
  | head :: tail -> filter_positives tail;;

(* Currying - funções que retornam funções *)
let add_curried x y = x + y;;  (* Funciona com múltiplos argumentos *)
let add_five = add_curried 5;;  (* Aplicação parcial *)
let result = add_five 3;;  (* 8 *)
```

6. Tipos Personalizados (Type Variants)

Perfeito para representar estados de autómatos:

ocaml

```ocaml
(* Tipo simples *)
type color = Red | Green | Blue;;
let my_color = Red;;

(* Tipo com dados *)
type state = Initial | Processing of int | Done of string;;
let s1 = Initial;;
let s2 = Processing 5;;
let s3 = Done "Finished";;

(* Pattern matching em tipos personalizados *)
let describe_state st =
  match st with
  | Initial -> "A começar"
  | Processing n -> Printf.sprintf "Processando: %d" n
  | Done msg -> Printf.sprintf "Terminado: %s" msg;;

(* Tipo com múltiplos campos (Record) *)
type person = {
  name : string;
  age : int;
  email : string
};;

let john = { name = "João"; age = 30; email = "joao@mail.com" };;
let name = john.name;;

(* Atualizar record (cria novo) *)
let john_older = { john with age = 31 };;
```

7. Records (Estruturas)

```ocaml
type node = {
  value : int;
  next : node option
};;

type automaton = {
  states : string list;
  initial : string;
  accepting : string list;
  transitions : (string * char * string) list
};;

let fa = {
  states = ["q0"; "q1"; "q2"];
  initial = "q0";
  accepting = ["q2"];
  transitions = [("q0", 'a', "q1"); ("q1", 'b', "q2")]
};;
```

8. Options (Valores que podem não existir)

Muito útil para lidar com casos onde algo pode estar ausente:

```ocaml
let find_in_list lst x =
  if List.mem x lst then Some x else None;;

(* Pattern matching *)
let process_option opt =
  match opt with
  | Some value -> Printf.sprintf "Encontrei: %d" value
  | None -> "Nada encontrado";;

(* Função prática *)
let safe_divide x y =
  if y = 0 then None else Some (x / y);;
```

9. Refs (Mutabilidade - Use com Cuidado!)

OCaml favorece imutabilidade, mas às vezes precisa:

ocaml

```ocaml
let counter = ref 0;;  (* Cria referência *)
let get_value = !counter;;  (* ! obtém o valor *)
counter := !counter + 1;;  (* := atualiza *)
```

10. Estruturas de Controlo

ocaml

```ocaml
(* If-then-else *)
let max_value x y =
  if x > y then x else y;;

(* If-then (sem else retorna unit) *)
let () = if true then print_endline "Hello";;

(* For loop *)
for i = 1 to 5 do
  Printf.printf "%d " i
done;;

(* While loop *)
let i = ref 1;;
while !i <= 5 do
  Printf.printf "%d " !i;
  i := !i + 1
done;;
```

11. Entrada/Saída

ocaml

```ocaml
(* Print *)
print_endline "Hello";;  (* Com quebra de linha *)
print_string "Hello";;   (* Sem quebra *)
Printf.printf "Número: %d\n" 42;;

(* Input *)
let input_line = read_line ();;  (* Lê uma linha *)
let input_int = int_of_string (read_line ());;
```

12. Conversões de Tipos

```ocaml
let s = string_of_int 42;;      (* int -> string *)
let i = int_of_string "42";;    (* string -> int *)
let f = float_of_int 42;;       (* int -> float *)
let b = bool_of_string "true";; (* string -> bool *)
```

13. Dica Especial para Autómatos

```ocaml
(* Representar transição *)
type transition = {
  from_state : string;
  symbol : char;
  to_state : string
};;

(* Representar autómato simples *)
type dfa = {
  alphabet : char list;
  states : string list;
  start : string;
  accept : string list;
  delta : transition list
};;

(* Função para simular *)
let rec simulate automaton word pos current_state =
  if pos = String.length word then
    List.mem current_state automaton.accept
  else
    let symbol = word.[pos] in
    let next_transitions =
      List.filter (fun t ->
        t.from_state = current_state && t.symbol = symbol
      ) automaton.delta
    in
    match next_transitions with
    | [] -> false
    | [t] -> simulate automaton word (pos + 1) t.to_state
    | _ -> false;;
```

Resumo de Syntax Essencial

Conceito	Sintaxe
Comentário	`(* comentário *)`
Variável	`let x = 5;;`
Função	`let f x = x + 1;;`
Lista vazia	`[]`
Cons	`head :: tail`
Concatenar	`list1 @ list2`
Tuple	`(1, 2, 3)`
Pattern match	`match expr with pattern -> result`
Recursão	`let rec func ...`
Tipo custom	`type name = Constructor;;`
Record	`type r = { field : type; };;`
Option	`Some x` ou `None`
Referência	`ref value`, `!ref`, `:=`