OCAML

Langage typé statiquement sans transtypage implicite.

Déclarations :

Let	Déclaration de variables/fonctions
exception	Déclaration d'exception
type	Déclaration de type

Opérateurs sur les entiers :

+	addition
-	soustraction
*	multiplication
/	division euclidienne
mod	modulo

Opérateurs sur les flottants :

+.	addition
-.	soustraction
*.	multiplication
/.	division euclidienne
**	exponentiation

Opérateurs booléens :

not	&& ou &	|| ou or

Opérateurs d'égalité et de comparaison :

=	égalité structurelle	<	inf.
==	égalité physique	>	sup.
<>	négation de =	<=	inf. ou égal
!=	négation de ==	>=	sup. ou égale

Conversion : float_of_int et int_of_float

Fonctions sur les flottants :

ceil	floor	sqrt
exp	log	log10

Fonctions trigonométriques :

cos	sin	tan
acos	asin	atan

Caractères : type char : 0-255 ; noté 'x'
char_of_int et int_of_char
Chaines : type string : longueur <2^24-6 ; noté "x"
Concaténation : ^
Conversion : int_of_string string_of_int
string_of_float float_of_string

Syntaxe des n-upplet : (X , X);; ou X, X;;
Pour les couples les fonctions fst et snd donne le premier et le second élément.

Syntaxe des listes : [X;X];; ou X::X::[];;
Liste vide : []
Concaténation de listes : [X;X]@[X;X];;
List.hd/List.tl : fournis respectivement la tête/la queue de la liste.
Appliquer une fonction à une liste :
List.map func list;;
Tester tous les éléments d'une liste :
List.for_all func list;;

Expression Conditionnelle :
if expr1 then expr2 else expr3;;
avec expr1 de type bool et expr2 et expr3 de type identique. else peut etre omis si expr3 est de type unit

Déclaration de valeurs :
let X = expr1;;
let X1=expr1 and X2=expr2 and ... Xn=exprn;;
Déclaration locale :
let X = expr1 in expr2;;
pred succ abs

Déclaration de fonctions :
function p -> expr ;; ou let nom = function p -> expr ;;
fun p1 ... pn -> expr;; ou let nom p1 ... pn = expr;;
Pour des function récursives remplacer let par let rec
Application partielle : let f x y =..;; let g = f x;;

Exceptions : Les noms des exceptions commence toujours par un majuscule !
Déclaration :
exception MonException;; exception simple
exception MonException of type;; exception avec argument.
Lancement :
raise MonException;;
raise (MonExecption argument);;
Récupération :
try expr with execption -> instruction ;;
Exception particulière :
raise (Failure "message") ~ failwith "message"

Pattern matching :

```
match var with
motif1 -> expr1
...
motifn -> exprnp
|_ -> expr1p;;
```

Il est possible de matcher plusieurs motifs :

```
motif1 | motif2 | motif3 -> expr1
```

Il est possible de matcher des couples :

```
(x,y) -> expr1
```

Filtrage universel :

```
(x,_) -> expr1
```

Filtrage avec garde :

```
(x,y) when contrainte -> expr1
```

Filtrage d'enregistrements :

```
{ch1=val, ch2=val} -> expr1
```

Fonctions et pattern matching :

```
let f x = match x with motifs ...;;
let f = function x -> match x with motifs...;;
let f = function motifs...;;
```

Déclaration de type :

```
type nom_type = definition;;
```

utilisation forcee du type : `let (x:type)=valeur;;`

Enregistrements :

déclaration : `type nom={ch1:type ; ch2:type} ;;`

utilisation : `let c = {ch1=valeur; ch2=valeur};;`

accès : `c.ch1 c.ch2`

modification d'un champ : `c.ch1<-valeur;;`

pour être modifiable ch1 doit être déclaré mutable : `type nom={mutable ch1:type ; ...} ;;`

Tableaux :

`[| ch1 ; ch2 ; ... ; chn |];;`

accès : `tab.(index)` avec `0<=index<length`

modification : `tab.(i) <- valeur`

```
Array.create taille valeur;;
Array.length tab;;
Array.copy tab;;
Array.append tab1 tab2;;
```

Tableaux en 2D

```
Array.create_matrix taille taille valeur;;
```

Les chaînes de caractères sont un cas particulier des tableaux :

```
let s=" chaine ";;
```

accès : `s.[i]`

modification `s.[i] <- ch;;`

```
String.length str;;
```