

INF201

Algorithmique et Programmation Fonctionnelle

Cours 1: Identificateurs, types de base et fonctions

Année 2023



Identificateurs

La notion d'identificateur

Associer un **nom** (**un identificateur**)
aux
valeurs, fonctions, types, ... d'un programme

Identificateurs

La notion d'identificateur

Associer un **nom** (**un identificateur**)
aux
valeurs, fonctions, types, ... d'un programme

Quelques règles de définition des identificateurs en OCaml :

- ▶ longueur maximale : 256 caractères
- ▶ doivent débuter par une **lettre minuscule**
- ▶ pas d'espace
- ▶ sensible à la casse

Identificateurs

La notion d'identificateur

Associer un **nom** (un identificateur)
aux
valeurs, fonctions, types, ... d'un programme

Quelques règles de définition des identificateurs en OCaml :

- ▶ longueur maximale : 256 caractères
- ▶ doivent débuter par une **lettre minuscule**
- ▶ pas d'espace
- ▶ sensible à la casse

Exemple : Identificateurs valides et non valides

- | | |
|---------------------|-----------------|
| ▶ vitesse ✓ | ▶ s ✓ |
| ▶ Vitesse ✗ | ▶ 3m ✗ |
| ▶ vitesse moyenne ✗ | ▶ temporaire3 ✓ |
| ▶ vitesse_moyenne ✓ | |

Identificateurs : définition globale

Syntaxe d'une *définition globale*

```
let identificateur = expression
```

↔ la **valeur** de `expression` est attachée/liée à `identificateur`

↔ le **type** de `identificateur` est le type de `expression`

Identificateurs : définition globale

Syntaxe d'une *définition globale*

```
let identificateur = expression
```

↔ la **valeur** de `expression` est attachée/liée à `identificateur`

↔ le **type** de `identificateur` est le type de `expression`

Cette définition est **globale** : elle peut être utilisée

- ▶ dans d'autres définitions
- ▶ dans le reste du programme

Identificateurs : définition globale

Syntaxe d'une *définition globale*

```
let identificateur = expression
```

↔ la **valeur** de `expression` est attachée/liée à `identificateur`

↔ le **type** de `identificateur` est le type de `expression`

Cette définition est **globale** : elle peut être utilisée

- ▶ dans d'autres définitions
- ▶ dans le reste du programme

Possibilités de **définitions simultanées** :

```
let ident1 = expr1 and ident2 = expr2 and ... identn = exprn
```

Identificateurs : définition globale

Syntaxe d'une *définition globale*

```
let identificateur = expression
```

↪ la **valeur** de `expression` est attachée/liée à `identificateur`

↪ le **type** de `identificateur` est le type de `expression`

Cette définition est **globale** : elle peut être utilisée

- ▶ dans d'autres définitions
- ▶ dans le reste du programme

Possibilités de **définitions simultanées** :

```
let ident1 = expr1 and ident2 = expr2 and ... identn = exprn
```

Exemple :

- ▶ `let x = 1`
- ▶ `let i = 1`
- ▶ `let y = 2 + 12`
- ▶ `let i = i+1`

DEMO: Définitions globales

Identificateurs : définition locale

Exemple : Comment calculer $e = (2*3*4)*(2*3*4)+(2*3*4)+2$?

$\hookrightarrow \text{prod} = (2*3*4)$

$\hookrightarrow e = \text{prod} * \text{prod} + \text{prod} + 2$

$\hookrightarrow \text{prod}$ est *locale* à e

Identificateurs : définition locale

Exemple : Comment calculer $e=(2*3*4)*(2*3*4)+(2*3*4)+2$?

$\hookrightarrow \text{prod} = (2*3*4)$

$\hookrightarrow e = \text{prod} * \text{prod} + \text{prod} + 2$

$\hookrightarrow \text{prod}$ est *locale* à e

Syntaxe d'une définition *locale* :

`let identificateur = expression1 in expression2`

\hookrightarrow la valeur de `expression1` est attachée/liée **de manière permanente** à `identificateur` **mais seulement** lors de l'évaluation de `expression2`

Identificateurs : définition locale

Exemple : Comment calculer $e=(2*3*4)*(2*3*4)+(2*3*4)+2$?

$\hookrightarrow \text{prod} = (2*3*4)$

$\hookrightarrow e = \text{prod} * \text{prod} + \text{prod} + 2$

$\hookrightarrow \text{prod}$ est *locale* à e

Syntaxe d'une définition *locale* :

`let identificateur = expression1 in expression2`

\hookrightarrow la valeur de `expression1` est attachée/liée **de manière permanente** à `identificateur` **mais seulement** lors de l'évaluation de `expression2`

Possibilité d'imbrications :

```
let id1=expr1 in
  let id2=expr2 in
...
  let idn = exprn ... in expr
```

Avec définitions simultanées :

```
let id1=expr1
  and id2=expr2
...
  and idn = exprn ... in expr
```

De l'importance des types ...

Programmation = traitement de données

De l'importance des types ...

Programmation = traitement de **données**

“traitement” ?

- ▶ faire des calculs
- ▶ mémoriser, rechercher, trier, associer ...
- ▶ échanger avec l'extérieur (afficher, capter, ...)
- ▶ etc.

Un facteur de **qualité**, de **correction** et d'**efficacité** d'un programme =

↪ choisir une “bonne” représentation pour ces données ...

De l'importance des types ...

Programmation = traitement de **données**

“traitement” ?

- ▶ faire des calculs
- ▶ mémoriser, rechercher, trier, associer ...
- ▶ échanger avec l'extérieur (afficher, capter, ...)
- ▶ etc.

Un facteur de **qualité**, de **correction** et d'**efficacité** d'un programme =
↔ choisir une “bonne” représentation pour ces données ...

Dans les langages de programmation :

→ notion de **type** (de données)

- ▶ types de base : fournis par le langage
- ▶ constructeurs de types : permettre au programmeur de définir ses propres types ...

Types de base et expressions

`int`: les entiers

- Un sous-ensemble fini des entiers signés \mathbb{Z} , ex., $-10, 2, 0, 3, 9 \dots$

Types de base et expressions

`int`: les entiers

- Un sous-ensemble fini des entiers signés \mathbb{Z} , ex., $-10, 2, 0, 3, 9 \dots$

- **Opérations usuelles :**

$-i$	opposé
$i + j$	addition
$i - j$	soustraction
$i * j$	multiplication
i / j	division (entière)
$i \bmod j$	modulo

DEMO: integers

Types de base et expressions

`float`: les “nombres à virgule”

- Un sous-ensemble fini des réels \mathbb{R}
- Leur notation nécessite :
 - ▶ soit un “point” (représente la virgule)
 - ▶ soit un exposant (base 10), prefixé par *e* ou *E*

Remarque Pas de calculs exacts (limites de précision)



Exemple :

0.2, 2e7, 1E10, 10.3E2, 33.23234E(-1.5), 2.

Types de base et expressions

`float`: les “nombres à virgule”

- Un sous-ensemble fini des réels \mathbb{R}
- Leur notation nécessite :
 - ▶ soit un “point” (représente la virgule)
 - ▶ soit un exposant (base 10), prefixé par *e* ou *E*

Remarque Pas de calculs exacts (limites de précision)



Exemple :

0.2, 2e7, 1E10, 10.3E2, 33.23234E(-1.5), 2.

• **Opérateurs usuels :**

<code>-.x</code>	opposé
<code>x +. y</code>	addition
<code>x -. y</code>	soustraction
<code>x *. y</code>	multiplication
<code>x /. y</code>	division
<code>int_of_float x</code>	conversion réel vers entier
<code>float_of_int x</code>	conversion entier vers réel

DEMO: float

Types de base et expressions

bool: les booléens

- L'ensemble des valeurs $\mathbb{B} = \{\text{true}, \text{false}\}$
- Quelques opérateurs sur les booléens:

not	négation
&&	"et puis"
	"ou alors"

- Quelques opérateurs à résultat booléen

<code>x = y</code>	x est <i>égal</i> à y
<code>x <> y</code>	x est non égal à y
<code>x < y</code>	x est plus petit que y
<code>x <= y</code>	x est plus petit ou égal à y
<code>x >= y</code>	x est plus grand ou égal à y
<code>x > y</code>	x est plus grand que y

DEMO: opérateurs à résultat booléens

Types de base et expressions

char: les caractères

- L'ensemble des caractères (du code ASCII)

$char \subseteq \{ 'a', 'b', \dots, 'z', 'A', \dots, 'Z' \}$

- Contient aussi les éléments suivants :

' \ \ '	le caractère backslash
' \ " '	le caractère apostrophe
' \ t '	le caractère tabulation
' \ r '	le caractère retour-chariot
' \ n '	le caractère fin-de-ligne
' \ b '	le caractère backspace

Types de base et expressions

char: les caractères

- L'ensemble des caractères (du code ASCII)

$char \subseteq \{ 'a', 'b', \dots, 'z', 'A', \dots, 'Z' \}$

- Contient aussi les éléments suivants :

' \ '	le caractère backslash
' \'	le caractère apostrophe
' \t '	le caractère tabulation
' \r '	le caractère retour-chariot
' \n '	le caractère fin-de-ligne
' \b '	le caractère backspace

- Conversion entre entier et caractère (et réciproquement): un caractère peut être représenté par son code ASCII :

- ▶ Char.code: renvoie le code ASCII d'un caractère
- ▶ Char.chr: renvoie le caractère correspondant à un code ASCII

- Des minuscules aux majuscules et réciproquement :

- ▶ Char.lowercase_ascii
- ▶ Char.uppercase_ascii

DEMO: char

Types de base et expressions

`string`: les chaînes de caractères

- chaînes=séquences de symboles placée entre guillemets: `"toto"`, `"23"`, `"INF201"` `"x=2"`.
- Un `char` n'est pas un `string`: `'a'` et `"a"` sont différents.

Types de base et expressions

`string`: les chaînes de caractères

- chaînes=séquences de symboles placée entre guillemets: `"toto"`, `"23"`, `"INF201"` `"x=2"`.
- Un `char` n'est pas un `string`: `'a'` et `"a"` sont différents.
- **Opérateurs usuels :**

<code>" "</code>	chaîne vide
<code>x ^ y</code>	concaténation
<code>String.length x</code>	longueur

DEMO: string

Compléments sur les opérateurs

Les opérateurs ont un type

Les opérateurs sont des fonctions (donc des valeurs), ils ont un type

→ contraintes sur les arguments et le résultat

- ▶ ordre
- ▶ nombre

↔ la “signature” d’un opérateur

Compléments sur les opérateurs

Les opérateurs ont un type

Les opérateurs sont des fonctions (donc des valeurs), ils ont un type

→ contraintes sur les arguments et le résultat

- ▶ ordre
- ▶ nombre

↔ la “signature” d’un opérateur

Etant donné un opérateur op :

arg1	$type_1$	
arg2	$type_2$	
...	...	
argn	$type_n$	
result	$type_r$	

$$\Rightarrow type_1 \rightarrow type_2 \rightarrow \dots \rightarrow type_n \rightarrow type_r$$

=

type de op

Compléments sur les opérateurs

Les opérateurs ont un type

Les opérateurs sont des fonctions (donc des valeurs), ils ont un type

→ contraintes sur les arguments et le résultat

- ▶ ordre
- ▶ nombre

↔ la “signature” d’un opérateur

Etant donné un opérateur op :

arg1	$type_1$	
arg2	$type_2$	
...	...	
argn	$type_n$	
result	$type_r$	

\Rightarrow

$$type_1 \rightarrow type_2 \rightarrow \dots \rightarrow type_n \rightarrow type_r$$

$=$
type de op

Exemple : Types de certains opérateurs

$+: int \rightarrow int \rightarrow int$
 $=: int \rightarrow int \rightarrow bool$
 $<: int \rightarrow int \rightarrow bool$
...

DEMO: type des opérateurs

Compléments sur les opérateurs

priorité et associativité

Rappel sur l'associativité:

- ▶ associativité à droite : $a \text{ op } b \text{ op } c$ signifie $a \text{ op } (b \text{ op } c)$
- ▶ associativité à gauche : $a \text{ op } b \text{ op } c$ signifie $(a \text{ op } b) \text{ op } c$

Priorité des opérateurs sur les types de base, par **ordre croissant**:

Opérateurs					Associativité
	&&				gauche
+	-	+	-		gauche
*	/	*	/	mod	gauche

Une construction du langage

`if ... then ... else ...`

Une **expression conditionnelle**

`if cond then expr1 else expr2`

- ▶ le résultat est une valeur
- ▶ `cond` doit être une expression booléenne
- ▶ `expr1` et `expr2` doivent être de même type

Exemple

`let m = if a > b then a else b`

Remarque La branche `else` est obligatoire ... (en INF201)



DEMO: `if...then...else...`

Compléments sur les type

A propos du système de type de OCaml

Le typage est un mécanisme/concept qui a pour but :

- ▶ d'éviter des erreurs
- ▶ de favoriser *l'abstraction*
- ▶ de vérifier que les expressions ont un sens, par ex.
 - ▶ `1 + yes`
 - ▶ `true * 42`

Compléments sur les type

A propos du système de type de OCaml

Le typage est un mécanisme/concept qui a pour but :

- ▶ d'éviter des erreurs
- ▶ de favoriser *l'abstraction*
- ▶ de vérifier que les expressions ont un sens, par ex.
 - ▶ `1 + yes`
 - ▶ `true * 42`

La vérification des types en OCaml : **typage statique strict**

- ▶ strict : pas de conversion implicite entre types
- ▶ statique : vérification avant exécution

Compléments sur les type

A propos du système de type de OCaml

Le typage est un mécanisme/concept qui a pour but :

- ▶ d'éviter des erreurs
- ▶ de favoriser *l'abstraction*
- ▶ de vérifier que les expressions ont un sens, par ex.
 - ▶ `1 + yes`
 - ▶ `true * 42`

La vérification des types en OCaml : **typage statique strict**

- ▶ strict : pas de conversion implicite entre types
- ▶ statique : vérification avant exécution

Inférence de type : pour toute expression e , OCaml calcule (automatiquement et systématiquement) le type de e

Compléments sur les type

A propos du système de type de OCaml

Le typage est un mécanisme/concept qui a pour but :

- ▶ d'éviter des erreurs
- ▶ de favoriser *l'abstraction*
- ▶ de vérifier que les expressions ont un sens, par ex.
 - ▶ `1 + yes`
 - ▶ `true * 42`

La vérification des types en OCaml : **typage statique strict**

- ▶ strict : pas de conversion implicite entre types
- ▶ statique : vérification avant exécution

Inférence de type : pour toute expression e , OCaml calcule (automatiquement et systématiquement) le type de e

Exemple : Typage des entiers et réels

- ▶ Deux ensembles disjoints d'opérateurs :
 - ▶ entiers (+, -, *)
 - ▶ réels (+., -., *.)
- ▶ Pas de conversion implicite entre eux, `1 + 0.42` produit une erreur

Compléments sur le typage

A propos du système de type de OCaml (suite)

OCaml est un langage de programmation **sûr** :

- ▶ Jamais d'erreur mémoire à l'exécution
- ▶ Nombreuses erreurs détectées avant exécution

Remarque Comparaison avec C :

- ▶ C est *faiblement typé* : conversions de type implicites
- ▶ Erreurs possibles à l'exécution : segmentation-fault, bus-error, etc. . .



Fonctions

Introduction

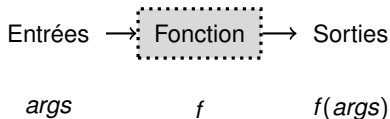
Jusqu'ici : expressions basées sur des opérateurs prédéfinis ...

Fonctions

Introduction

Jusqu'ici : expressions basées sur des opérateurs prédéfinis ...

Fonctions



Intérêt :

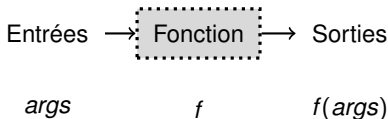
- ▶ définir de nouvelles opérations
- ▶ pouvoir les ré-utiliser à plusieurs endroits d'un programme
- ▶ lisibilité du code

Fonctions

Introduction

Jusqu'ici : expressions basées sur des opérateurs prédéfinis ...

Fonctions



Intérêt :

- ▶ définir de nouvelles opérations
- ▶ pouvoir les ré-utiliser à plusieurs endroits d'un programme
- ▶ lisibilité du code

Les fonctions dans les langages fonctionnels

- ▶ Proches des fonctions mathématiques
- ▶ Pas d'effets de bords (contrairement à C)
- ▶ Des expressions comme les autres, avec une *valeur* et un *type*

Définition des Fonctions

Exemple : la fonction valeur absolue

<i>nom</i>	<i>param. formel</i>	<i>type du résultat</i>	<i>corps de la fonction</i>
let	abs (a:int)	:int	= if a < 0 then -a else a
	↓	↓	↑
type:	int	->	int

Définition des Fonctions

Exemple : la fonction valeur absolue

<i>nom</i>	<i>param. formel</i>	<i>type du résultat</i>	<i>corps de la fonction</i>
<code>let</code>	<code>abs (a:int)</code>	<code>:int</code>	<code>= if a < 0 then -a else a</code>
	↓	↓	↑
	type: <code>int</code>	<code>-></code>	<code>int</code>

Autres écritures possibles :

`let abs = fun (a:int) -> if a < 0 then -a else a`

ou `let abs a = if a < 0 then -a else a`

ou `let abs (a:int) = if a < 0 then -a else a`

Définition des Fonctions

Exemple : la fonction valeur absolue

<i>nom</i>	<i>param. formel</i>	<i>type du résultat</i>	<i>corps de la fonction</i>
<code>let</code>	<code>abs (a:int)</code>	<code>:int</code>	<code>= if a < 0 then -a else a</code>
	↓	↓	↑
	type: <code>int</code>	<code>-> int</code>	

Autres écritures possibles :

`let abs = fun (a:int) -> if a < 0 then -a else a`

ou `let abs a = if a < 0 then -a else a`

ou `let abs (a:int) = if a < 0 then -a else a`

Exercice

Définir la fonction `carre: int -> int`

Fonctions

Comment les utiliser

$f(x)$, le résultat de l'application de f à x sera noté en CAML $(f\ x)$

Exemple :

- ▶ `(abs 2)`
- ▶ `(abs(2 - 3))`
- ▶ `abs 2` (les parenthèse peuvent ici être omises ...)

Fonctions

Comment les utiliser

$f(x)$, le résultat de l'application de f à x sera noté en CAML $(f\ x)$

Exemple :

- ▶ $(\text{abs } 2)$
- ▶ $(\text{abs}(2 - 3))$
- ▶ $\text{abs } 2$ (les parenthèses peuvent ici être omises ...)

Application d'une fonction

(expr1 expr2)

Typage :

si $\left. \begin{array}{l} \text{expr1 a pour type } t1 \rightarrow t2 \\ \text{et } \text{expr2 a pour type } t1 \end{array} \right\} \text{ alors } (\text{expr1 expr2}) \text{ a pour type } t2$

Fonctions: Généralisation aux fonctions à plusieurs arguments

Exemple : Aire d'un rectangle

- ▶ Nécessite 2 paramètres : longueur et largeur (floats)
- ▶ définition: `let aire (x:float) (y:float) : float = x *. y`
- ▶ appel de la fonction : `(aire 2.3 1.2)`

Fonctions: Généralisation aux fonctions à plusieurs arguments

Exemple : Aire d'un rectangle

- ▶ Nécessite 2 paramètres : longueur et largeur (floats)
- ▶ définition: `let aire (x:float) (y:float) : float = x *. y`
- ▶ appel de la fonction : `(aire 2.3 1.2)`

Définition d'une fonction à n paramètres

`let nom_fonction (p1:t1) (p2:t2) ... (pn:tn) : t = expr`

- ▶ p_1, \dots, p_n sont les *paramètres formels*
- ▶ Le type de `nom_fonction` est `t1 -> t2 -> ... -> tn -> t`

Fonctions: Généralisation aux fonctions à plusieurs arguments

Exemple : Aire d'un rectangle

- ▶ Nécessite 2 paramètres : longueur et largeur (floats)
- ▶ définition: `let aire (x:float) (y:float) : float = x *. y`
- ▶ appel de la fonction : `(aire 2.3 1.2)`

Définition d'une fonction à n paramètres

`let nom_fonction (p1:t1) (p2:t2) ... (pn:tn) : t = expr`

- ▶ p_1, \dots, p_n sont les *paramètres formels*
- ▶ Le type de `nom_fonction` est `t1 -> t2 -> ... -> tn -> t`

Appeler une fonction à n paramètres

`(nom_fonction e1 e2 ... en)`

- ▶ e_1, \dots, e_n sont les *paramètres effectifs*
- ▶ Le type de `nom_fonction e1 e2 ... en` est `t`
si `ti` est le type de `ei` et `nom_fonction` est de type
`t1 -> t2 -> ... -> tn -> t`

Fonctions anonymes

Cela nous servira plus tard ...

→ on définit une fonction, mais sans lui donner de nom ...

Exemple : retour sur la valeur absolue

```
fun a → if a < 0 then -a else a  
ou function a → if a < 0 then -a else a  
ou function (a:int) → if a < 0 then -a else a
```

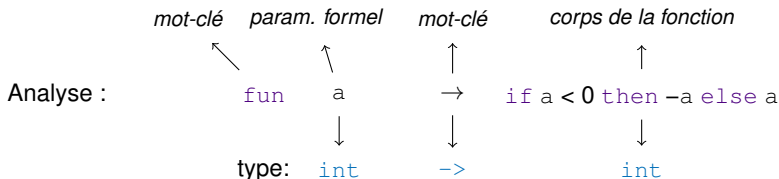
Fonctions anonymes

Cela nous servira plus tard ...

→ on définit une fonction, mais sans lui donner de nom ...

Exemple : retour sur la valeur absolue

```
fun a → if a < 0 then -a else a  
ou function a → if a < 0 then -a else a  
ou function (a:int) → if a < 0 then -a else a
```



Déclarations, expressions et contexte

- ▶ Déclaration = définition globale (`let` = ;;) de valeur, de fonction ou définition de type (voir prochain cours `type` = ;;).
 - ▶ `let x=2;;`
 - ▶ `let f (x:int):int = 2+x;;`
 - ▶ `type euro=float;;`
- ▶ Expression = phrase Ocaml qui n'est pas une déclaration:
 - ▶ `2`
 - ▶ `2+3`
 - ▶ `2+3*x`
 - ▶ `if x>0 then f x else g x`
 - ▶ `let x=3 in (if x>0 then f x else x)`

Une expression a une certaine valeur dans un certain contexte.

Exemple

- ▶ `2+3*x` a pour valeur 5 dans le contexte $\{x \mapsto 1\}$.
- ▶ `(if x>0 then (f x) else x)` est une expression de type `int` qui a pour valeur 25 dans le contexte $\{x \mapsto 5, f \mapsto \text{carré}\}$

Le “let ... = ... in ...” est une expression comme une autre

Exemple :

- ▶ `(let x=3 in x+2)` est une expression de type `int` qui vaut 5.
- ▶ `(let carre x → x*x in (carre 2)+(carre 3))` est une expression de type `int` qui vaut 25.

On peut utiliser des définitions locales à l'intérieur du corps d'une autre fonction.

Exemple :

Soit f la fonction qui aux entiers x et y associe $(x^2 + y^2)^2$.

```
let f (x:int) (y:int): int =  
  let carre (a:int): int → a*a in carre ((carre x)+(carre y));;
```


Fonctions: SPECIFICATION et IMPLEMENTATION

Il est très important de distinguer 2 concepts/étapes lors de la définition d'une fonctions (et d'un programme en général)

Fonctions: SPECIFICATION et IMPLEMENTATION

Il est très important de distinguer 2 concepts/étapes lors de la définition d'une fonctions (et d'un programme en général)

Spécification:

Une description de **ce qui est attendu**

- ▶ à un certain niveau d'abstraction
- ▶ doit être suffisamment précis
- ▶ proche d'une description "mathématique"
- ▶ peut utiliser des exemples *pertinents*

Un contrat



Consiste en:

- ▶ 1 description
- ▶ 1 signature
- ▶ des exemples

Fonctions: SPECIFICATION et IMPLEMENTATION

Il est très important de distinguer 2 concepts/étapes lors de la définition d'une fonction (et d'un programme en général)

Spécification:

Une description de **ce qui est attendu**

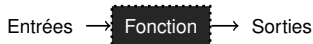
- ▶ à un certain niveau d'abstraction
- ▶ doit être suffisamment précis
- ▶ proche d'une description "mathématique"
- ▶ peut utiliser des exemples *pertinents*

Implémentation:

La description de **comment le réaliser**

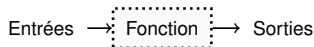
- ▶ le code OCaml

Un contrat



Consiste en:

- ▶ 1 description
- ▶ 1 signature
- ▶ des exemples



Fonctions: SPECIFICATION et IMPLEMENTATION

Il est très important de distinguer 2 concepts/étapes lors de la définition d'une fonctions (et d'un programme en général)

Spécification:

Une description de **ce qui est attendu**

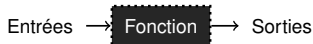
- ▶ à un certain niveau d'abstraction
- ▶ doit être suffisamment précis
- ▶ proche d'une description "mathématique"
- ▶ peut utiliser des exemples *pertinents*

Implémentation:

La description de **comment le réaliser**

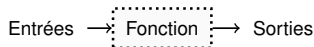
- ▶ le code OCaml

Un contrat



Consiste en:

- ▶ 1 description
- ▶ 1 signature
- ▶ des exemples



Définir une fonction : Spécification **PUIS** Implémentation

Définir des fonctions : quelques exemples

Exemple : Définir la fonction valeur absolue

- ▶ Spécification:
 - ▶ La fonction valeur absolue *abs* prend un entier n en paramètre et renvoie n si cet entier est positif et $-n$ si cet entier est négatif. La fonction valeur absolue renvoie toujours un entier positif.
 - ▶ Profil : $\mathbb{Z} \rightarrow \mathbb{N}$
 - ▶ Exemple : $abs(1) = 1$, $abs(0) = 0$, $abs(-2) = 2$
- ▶ Implémentation: `let abs (a:int) : int = if a < 0 then -a else a`

Définir des fonctions : quelques exemples

Exemple : Définir la fonction valeur absolue

- ▶ Spécification:
 - ▶ La fonction valeur absolue *abs* prend un entier n en paramètre et renvoie n si cet entier est positif et $-n$ si cet entier est négatif. La fonction valeur absolue renvoie toujours un entier positif.
 - ▶ Profil : $\mathbb{Z} \rightarrow \mathbb{N}$
 - ▶ Exemple : $abs(1) = 1$, $abs(0) = 0$, $abs(-2) = 2$
- ▶ Implémentation: `let abs (a:int) : int = if a < 0 then -a else a`

Exemple : Définir la fonction carre

- ▶ Spécification:
 - ▶ La fonction carre *sq* prend un entier n en paramètre et renvoie $n * n$.
 - ▶ Profil : $\mathbb{Z} \rightarrow \mathbb{N}$
 - ▶ Exemple : $sq(1) = 1$, $sq(0) = 0$, $sq(3) = 9$, $sq(-4) = 16$
- ▶ Implémentation: `let sq (n:int) : int = n*n`

Exercices

Un peu d'algorithmique

Exercice

Définir la fonction `my_max` qui renvoie le maximum de deux entiers

Exercices

Un peu d'algorithmique

Exercice

Définir la fonction `my_max` qui renvoie le maximum de deux entiers

Exercice

Définir les fonctions:

- ▶ `carre: int → int`
- ▶ `somme_carre: int → int → int`

tq. `somme_carre` calcule la somme des carrés de deux entiers

Exercices

Un peu d'algorithmique

Exercice

Définir la fonction `my_max` qui renvoie le maximum de deux entiers

Exercice

Définir les fonctions:

- ▶ `carre: int → int`
- ▶ `somme_carre: int → int → int`

tq. `somme_carre` calcule la somme des carrés de deux entiers

Problème : moyenne olympique

Calcule la moyenne de 4 valeurs, sans prendre en compte la plus petite ni la plus grande

1. Proposez un type pour la fonction `moy_olymp`
2. Proposez un algorithme, en supposant que l'on dispose de 2 fonctions `min4` et `max4`, qui calculent respectivement le minimum et le maximum de 4 entiers
3. Définissez les fonctions `min4` et `max4`, en utilisant `min` et `max`

Résumé et Travail à faire

Résumé

- ▶ Identificateurs
- ▶ Types de base et opérateurs

type	opérateurs	constantes
Booleans	<code>not, &&, </code>	<code>true, false</code>
integers	<code>+, -, *, /, mod</code>	<code>..., -1, 0, 1, ...</code>
floats	<code>+. , -. , *. , /. </code>	<code>0.4, 12.3, 16. , 64.</code>

- ▶ construction `if...then...else`
- ▶ Système de type OCaml
- ▶ Fonctions
 - ▶ définition et utilisation
 - ▶ spécification et réalisation d'une fonction

Travail à faire

Consulter la page web du cours

Revoir ces transparents !

Jouer avec l'interpréteur OCaml : <https://try.ocamlpro.com/>