Programmation impérative

- Paradigme **impératif** : un programme est une séquence d'instructions

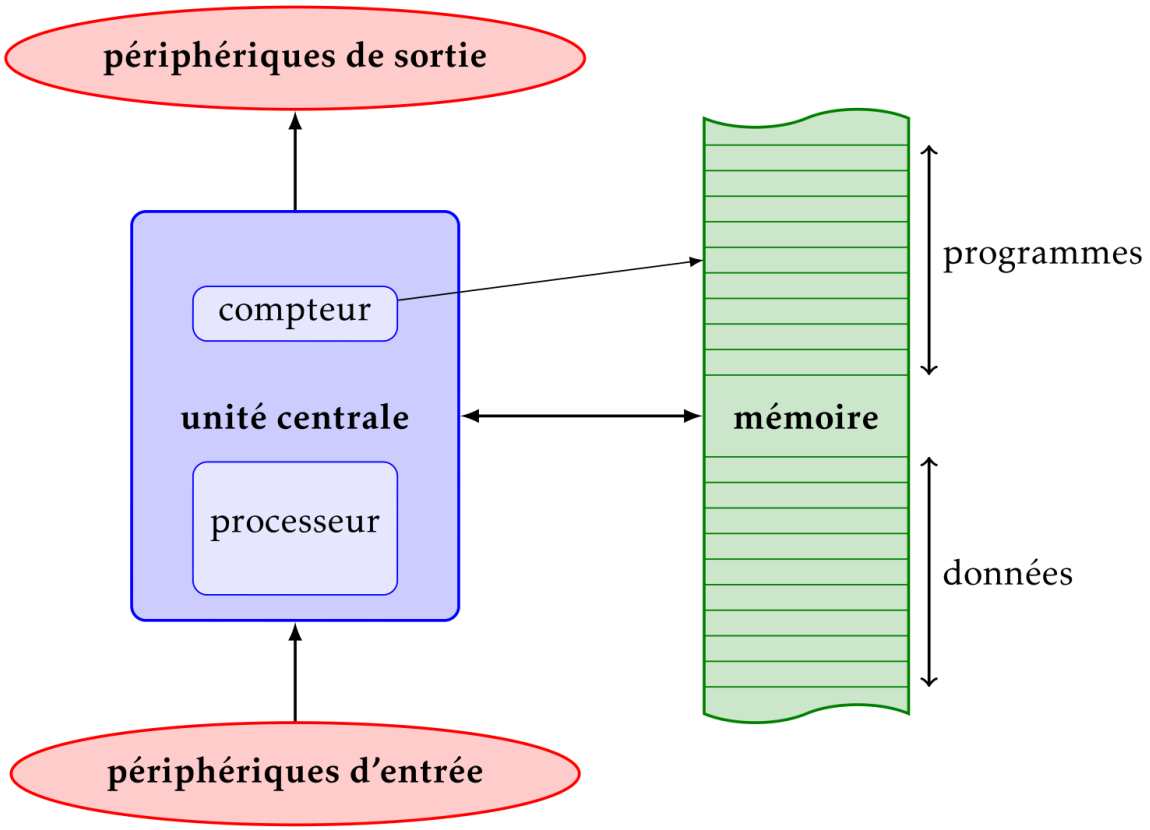
OCAML
```
do_this();
do_that();
print_this();
drop_me();
```

- Modification de **l'état** de l'ordinateur
- Paradigme **fonctionnel** : un programme est l'évaluation d'une expression/fonction.

OCAML
```
drop (eval_that (eval_this obj))
```

- Programme = calcul d'une fonction

Programmation par effets



Deux types d'*effets* :

- intervention avec l'extérieur (entrée ou sortie) ;
- modification de l'état de la mémoire.

Programmation par effets en CAML

- Quizz :

> Quelle « instruction » avons-nous déjà rencontrée en CAML qui ne rentre pas vraiment dans le cadre étudié jusqu'ici ?

- Réponse :

OCAML

```
let add x y = x + y;;
add : int -> int -> int = <fun>

#trace add;;
add is now traced.
```

Les fonctions `print`

- Réponse 2 :

OCAML

```
print_int 3;;
3

print_string "En Python on savait faire...";;
En Python on savait faire...

print_newline ();;
```

De manière plus générale : `print_type`.

Les fonctions `print`

- Comment imprimer les entiers de 0 à 5 ?

OCAML

```
let imprime_entiers n =
  let rec loop i =
    print_string " ";
    print_int i;
    if i = n then
      print_string "."
    else
      loop (i + 1)
  in
  loop 0
;;

imprime_entiers 5;;
0 1 2 3 4 5
```

- Mais... quel est son type ?

Le type unit

On a besoin d'un « nouveau » type :

OCAML

```
();;
- : unit = ()

print_int;;
- : int -> unit = <fun>

print_newline;;
- : unit -> unit = <fun>

print_newline ();;

- : unit = ()
```

Effets de bords

- Fonctions `print_int`, `print_float`, `print_char`, `print_string` et `print_newline`
- agissent (en sortie) sur l'écran
- Toutes de type **`type -> unit`**
- Symétriquement, les fonctions `read_int`, `read_float` et `read_line` agissent en entrée
- demandent à l'utilisateur d'utiliser son clavier pour envoyer au processeur respectivement un entier, un flottant ou une chaîne de caractères
- Type de la forme **`unit -> type`**

Effets de bords

Attention

Le « résultat » d'un programme peut donc dépendre du « moment » où il est exécuté

- Ce n'était pas le cas jusqu'à présent.
- Souvent, ces effets compliquent la lisibilité du comportement des programmes et/ou nuisent à la réutilisabilité des fonctions et procédures.

Séquences en CAML

- « Instruction » ≈ expression de type **`unit`**
- En CAML, séquences d'instructions : expressions séparées par des points-virgules `;`
- L'évaluation de la séquence :

```
expr1; expr2; ...; exprn
```

- c'est une expression
- provoque les effets éventuels des n expressions dans l'ordre
- sa valeur est celle de la dernière expression
- la valeur des autres expressions étant ignorée/supprimée
- le type des autres expressions ~~devrait~~ doit être **`unit`**
- séquence d'instructions sans effets : sans intérêt !

Séquences en CAML

- le `;` est bien un séparateur
- on a souvent besoin de parenthéser :
- `(expr1; expr2; ...; exprn)`
- on utilise plutôt **`begin`** [...] **`end`**
- **`begin`** `expr1; expr2; ...; exprn` **`end`**
- Attention : ~~conseil~~ obligation : une seule instruction par ligne :

OCAML

```
begin
  expr1;
  expr2;
  ...;
  exprn
end
```

Références

- Raccourci pour manipuler les adresses mémoire (pointeur)
- Création : `let identifiant = ref valeur <;; ou in>` :

OCAML
```
let x = ref 0;;
x : int ref = ref 0
```

- `valeur` est la valeur initiale
- Affectation d'une nouvelle valeur : opérateur infixe `:=` :

OCAML
```
x := 2;;
- : unit = ()
x;;
- : int ref = ref 2
```

Références

- Accès au contenu par opérateur de déréférencement (bang) `!x` :

OCAML

```
x := !x + 3;;
- : unit = ()

!x;;
- : int = 5

x;;
- : int ref = ref 5
```

Différence entre liaison et référence

- Liaison permanente entre un nom et une valeur
- Référence : emplacement en mémoire (une boîte)
- Une référence est une valeur de type `ref`
- Différencier
  - `let s = 0;;` : liaison entre le nom `s` et la valeur `0`
  - `let s = ref 0;;` : liaison entre le nom `s` et un emplacement en mémoire contenant la valeur `0`
- L'emplacement mémoire est une valeur qui « contient » une autre valeur (un entier) : `int ref`.
- Différence importante : comportement statique d'une définition contre comportement dynamique d'une référence

Différence entre liaison et référence : exemple

OCAML

```
let a = 1;;
a : int = 1

let incr x = x + a;;
incr : int -> int = fun

let a = 2;;
a : int = 2

incr 0;;
- : int = 1
```

Différence entre liaison et référence : exemple

OCAML

```
let a = ref 1;;
a : int ref = ref 1

let incr x = x + !a;;
incr : int -> int = fun

a := 2;;
-: unit = ()

incr 0;;
- : int = 2
```

Instruction conditionnelle : `if` déjà vu

- l'exécution dépend d'une condition

OCAML
```
if condition then expr1 else expr2
```

- `expr1` et `expr2` doivent être du même type
- En l'absence de `else`, `expr1` doit être du type `unit` et `expr2` est considéré comme `()`.

OCAML
```
if condition then expr1
```

est équivalent à

OCAML
```
if condition then expr1 else ()
```

Attention

Ne jamais oublier **else** si `expr1` n'est pas de type **unit** !

Priorité du `;` par rapport au `if`

Attention

Si on veut mettre une séquence dans un `then` ou un `else` : il **faut** la mettre entre parenthèses, *i.e.* encadrée par les mots-clés `begin` et `end`.

Syntaxe :

OCAML

```
if test then begin
  expr1;
  expr2;
  [...]
end else begin
  expr1';
  expr2';
  [...]
end
```

Priorité du `;` par rapport au `if`

Attention

Attention

Retenir

`;` est prioritaire devant `if then else`

OCAML

```
if cond then expr1; expr2 else expr3
(* identique à *)
begin if cond then expr1 else () end; expr2 else expr3
(* aucun sens *)
```

Boucles inconditionnelles

OCAML

```
for indice = expr1 to expr2 do
  expr3
done
```

- indice de boucle `indice` (type entier) est incrémenté de 1 en 1 entre les valeurs prises par `expr1` et `expr2` en effectuant à chaque fois le calcul de `expr3` sous forme de séquence
- Par exemple :

OCAML

```
for i = 1 to 10 do
  print_int i;
  print_char ' '
done;;
1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

- souvent `expr3` est une séquence !
- souvent `expr3` dépend de `indice` !

Boucles inconditionnelles

OCAML
```
for indice = expr1 downto expr2 do
  expr3
done
```

- indice de boucle `indice` (type entier) est décrémenté de 1 en 1 entre les valeurs prises par `expr1` et `expr2` en effectuant à chaque fois le calcul de `expr3` sous forme de séquence
- Par exemple :

OCAML
```
for i = 10 downto 1 do
  print_int i;
  print_char ' '
done;;
10 9 8 7 6 5 4 3 2 1 - : unit = ()
```

Bornes dans les boucles inconditionnelles

Attention

Ce n'est pas comme `range` en PYTHON. Les bornes sont inclusives.

OCAML

```
let n = 10 in
for i = 1 to n do
  print_int i;
  print_char ' '
done;;
1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

Boucle conditionnelle

OCAML

```
while expr1 do
  expr2
done
```

- On évalue `expr2` tant que la condition `expr1` (type **`bool`**) est vérifiée
- `expr2` doit réaliser un effet (sinon, on voit mal comment la condition pourrait cesser d'être vérifiée...)
- Par exemple :

OCAML

```
let i = ref 1 in
while !i <= 10 do
  print_int (!i);
  print_char ' ';
  i := !i + 1
done;;
1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

Enregistrements à champs mutables

OCAML

```
type etudiant = {numero : int; mutable age : int};;
```

On modifie le champ d'un enregistrement avec la construction <-

OCAML

```
let anniversaire eleve = eleve.age <- eleve.age + 1;;
val anniversaire : etudiant -> unit = <fun>

let eleve = {numero = 12345; age = 18};;
val eleve : etudiant = {numero = 12345; age = 18}

anniversaire eleve;;
- : unit = ()

eleve;;
- : etudiant = {numero = 12345; age = 19}
```

Retour sur le type 'a ref

Les références ne sont rien de plus que des enregistrements à un seul champ mutable.

OCAML

```
type 'a ref = {mutable content : 'a};;

let ref = fun valeur -> {content = valeur};;
let (:=) = fun refer val -> refer.content <- val;;
let (!) = fun reference -> reference.content;;
```

Fermetures et références

OCAML

```
let compteur i =
  let etat = ref i in
  fun () -> incr etat;
  !etat;;
val compteur : int -> unit -> int = <fun>

let compteur1 = compteur 0;;
val compteur1 : unit -> int = <fun>

compteur1 ();;
- : int = 1

compteur1 ();;
- : int = 2

let compteur2 = compteur 10;;
val compteur2 : unit -> int = <fun>

compteur2 ();;
- : int = 11
```

Fermetures et références

- Une fermeture est une fonction avec un **état interne**.
- Dans cet exemple, les fonctions `compteur1` et `compteur2` ont chacune leur état.
- Il faut bien savoir distinguer les deux fonctions suivantes :

OCAML
```
let f =
  let x = ref 3 in
  fun () -> x
;;
```

OCAML
```
let g = fun () ->
  let x = ref 3 in
  x
;;
```

Que contiennent `b` et `d` après les instructions suivantes ?

OCAML
```
let a = f ();;
let b = f ();;
incr a;;
!b;;
```

OCAML
```
let c = g ();;
let d = g ();;
incr c;;
!d;;
```

Typage des références polymorphes

La structure de liste étant par construction une structure de donnée récursive et persistante (non modifiable), **il est interdit de n'avoir ne serait-ce que l'idée même de penser à utiliser des références de listes.**

Mais... Que se passe-t-il si on essaye quand même ?

Les trois instructions suivantes sont-elles bien typées ?

OCAML

```
let liste = ref [];;

liste := 4 :: !liste;;

let f liste =
  match liste with
  | [] -> 0.
  | x :: _ -> x +. 0.5
in
f !liste;;
```

Si c'était le cas, la troisième provoquerait une erreur.

Variables de type monomorphe

Pour résoudre ce problème, on introduit des variables de type **monorphe** qu'on distingue des variables polymorphes à l'aide du caractère _.

OCAML

```
let liste = ref [];;
val liste : '_a list ref = {contents = []}
```

À la différence des variables polymorphes, la valeur d'une variable monomorphe évolue au fur et à mesure que de nouvelles contraintes apparaissent.

OCAML

```
liste := 4 :: !liste;;
- : unit = ()

liste;;
- : int list ref = {contents = [4]}
```

Variables de type monomorphe

C'est un phénomène qui a déjà été rencontré en début d'année. Un application de fonction peut provoquer du monomorphisme :

OCAML

```
let identite x = x;;
val identite : 'a -> 'a = <fun>

identite identite;;
- : '_a -> '_a = <fun>
```

Alors que la construction `let` est susceptible d'introduire du polymorphisme :

OCAML

```
let itere f = f f in itere identite;;
Characters 16-17:
  let itere f = f f in itere identite;;
                  ^
Error: This expression has type 'a -> 'b
       but an expression was expected of type 'a
       The type variable 'a occurs inside 'a -> 'b
```