Programmation impérative

Références

- les *"pointeurs"* de Caml
  - `let a = ref 3` a → 3
  - `a` est l'adresse d'une case mémoire
- déréférenciation: opérateur `!` (BANG), lire la case mémoire
  - `f !a` par exemple
- modifier la valeur stockée en `a :=`

  `a := 4`
- programmation *impérative:*
  - `:=` est un *ordre* (une instruction),
  - par opposition à une *déclaration*: `let x = ...`

  "marions-nous" / *"il se trouve que je suis amoureux de toi"*

Références – typage

- `ref` est un *constructeur de type*

  si `x :t ref` alors `!x :t` `x` est une *case mémoire*
- type de l'effet de bord `x := v : unit = ()` *"objet par défaut"*
- combiner effet de bord et valeur: la *séquence* `;`

  `# a := !a+1; 4+3;;` ⇝ `- : int = 7`

  N.B.: la séquence "jette" une valeur: `3;4;;` ⇝ *Warning*
- `unit` est également utilisé pour *déclencher* un effet de bord

```
let x = ref 0;;
let f = function () -> (x := !x+1; !x);;
let g () = (x := 3* !x; !x);;
f ();;   g();;
f() + g();;
```

→ 19 *: le résultat dépend de l'implémentation de +*

Enregistrements modifiables

- rappel: enregistrements

```
type pers = { nom : string; age : int }
```
- le mot-clef `mutable` donne le droit de *modifier en place* un champ d'un enregistrement

```
type pers = { nom : string; mutable age : int };;
let moi = { nom = "daniel"; age = 9 };;
moi.age <- 11;;      modifier en place: enr.champ <- expr;;
# moi;;
- : pers = {nom="daniel"; age=11}
```

  droits en lecture seule, ou lecture et écriture
- la nature d'un `ref`

```
# let a = ref 3;;
val a : int ref = {contents=3}
```

Deux mots sur C

- en C, les variables sont par défaut (presque) toutes mutables

  (pas les tableaux)
- la vie d'une variable est mouvementée en C
  - déclaration `int k;`
  - initialisation `k=0;`
  - modifications `k++;...k=f(x,y);`
  - disparition ("automatique") `{ int a;...}`$_{\dagger a}$
- les variables de Caml
  - déclaration-initialisation et disparition

    `let a = 0 in (...)`$_{\dagger a}$
  - `let li = [3;2;5;2] in (...)`$_{\dagger li}$

    déclaration-allocation-initialisation-désallocation

    (les `malloc`s et les `free`s que ça cache)

C – pointeurs

- `int *p;` `p` est un pointeur sur un entier
  - ''`let a = ref (??  :  int)`''
  - `*p` est un entier

    l'opérateur `*` *déréférence*, comme `!` en Caml
- `int a; int *p; ...  p = &a;`
  - `p` pointe sur `a`, `p` reçoit l'adresse de `a`
  - `&` permet de récupérer l'adresse en mémoire de tout objet (un entier, une fonction, ...)
  - pas de ça en Caml: n'est un `ref` que quelque chose qui a été "créé" comme un `ref`
- autre différence entre les `ref` de Caml et les pointeurs de C: on n'a jamais accès à la "vraie" adresse en Caml
  - du coup Caml peut 'bouger' les `ref` sans qu'on le remarque
  - pas d'arithmétique sur les pointeurs

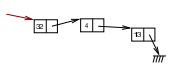
Il faut refaire des pointeurs en C

- mes informateurs me disent que les pointeurs n'ont pas été assimilés en TP de C (la semaine dernière)

```
int *p;        int *p;
p = 3;         *p = 3;
```

  quel est le problème?
- cette semaine, n'hésitez pas à solliciter vos TDmen des groupes du mardi soir et du jeudi pour des éclaircissements sur le TP de C

Du Raymond Queneau avec des listes



- la structure de liste est *récursive*
- pour implémenter des listes, il faut manipuler des pointeurs

  DÉMO `listes_c.c`
- adaptation naïve ("pointeurs en Caml = `ref`")

  DÉMO `listes_ref.ml`
- une solution plus naturelle en Caml

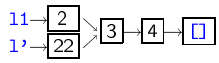
  DÉMO `listes_mutable.ml`
- et bien sûr

  DÉMO `listes_caml.ml`

Sous le tapis

- avoir les types somme et le filtrage permet de se dispenser de beaucoup de manipulations scabreuses et sources d'erreur

  "langage de haut niveau"

- DÉMO `partage.ml`
  - = égalité *structurelle* == égalité *physique*
  - *partage*: 

    on peut voir cela comme une "optimisation": on profite du fait que les listes sont *immutables*

- les valeurs sont *construites* DÉMO `construct.ml`

Égalités structurelle et physique

- rappel: deux manières de regarder les valeurs
  - =: égalité structurelle, les objets en tant que valeur
  - ==: égalité physique: les objets en tant que mot en mémoire

    en Java, == et la méthode `equals` pour définir l'égalité structurelle pour une classe donnée

- phénomènes d'alias, de partage

  DÉMO `alias.ml`

`let x = ref 52;;`	*et en shell?*	on a créé un *alias*:
`let y = x;;`	commande ln	deux noms pour le même
`let z = ref !x;;`	commande cp	objet

- différence entre valeurs **immédiates** et valeurs **référencées**

  DÉMO `immediat.ml`

  un mot fait (souvent) 32 bits, les entiers de Caml sont sur 31 bits

Tableaux en Caml

les tableaux (aussi appelés *vecteurs*) en OCaml:

▷ création:

```
# let t = [| 1 ; 2 |];;
val t : int array = [|1; 2|]
# let v = Array.make 3 true;;
val v : bool array = [|true; true; true|]
```

▷ accès:

```
# v.(1) <- not v.(2);;
- : unit = ()
# v;;
- : bool array = [|true; false; true|]
```

les tableaux sont la structure de données "naturelle" en programmation impérative

Itération

- `for id = e1 to e2 do e3 done`

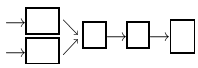
```
# let a = ref 1 in
  for i = 2 to 5 do a := !a * i done;;
- : unit = ()
```

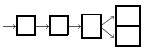
- `while e1 do e2 done`

```
# let a,b = ref 5,ref 1 in
  (while !a>0 do b:= !b * !a; a:= !a-1 done; !b);;
- : int = 120
```

- toutes les constructions impératives vont vers `unit`

Programmation impérative, valeurs en mémoire

- les données en mémoire sont (bien entendu) modifiables
  - en Caml, `mutable`
- les données de types de base (`int`, `float`, ...) sont stockées "comme on croit" en mémoire
  - ou presque: cf. plus tard
- idem pour les enregistrements
  - `{a : int ; b : bool }`
- pour les types somme définis par l'utilisateur, on a vu une technique d'implémentation à base de *cellules* (une cellule par argument)
  - `type toto = C of int*toto*bool | ...` ⇝
- phénomènes de *partage*: 

  (et ça? )

Pile et tas, fonctions en mémoire

- retour sur la pile d'appels des fonctions
  - `let f1 x = x+3`
  - `let f2 y = [y;y;y]`
  - `let f3 z = ref (z*z)`
- `f2` et `f3` renvoient un pointeur, la valeur pointée ne peut être dans la pile
  - à l'autre extrémité de la mémoire: le tas (allocation dynamique)
- en Caml, les fonctions sont des données
  - en argument et en sortie d'autres fonctions
  - `type toto = {a : string; f : int -> int }`
  - les fonctions sont immutables, d'ailleurs
- stocker une fonction en mémoire, c'est en stocker le code *compilé* (cf. plus tard)

Encore du partage

- des matrices comme tableaux de tableaux

```
# let a = Array.make 3 0;; ← on ne crée que 3 entiers
val a : int array = [|0; 0; 0|]
# let b = Array.make 3 a;;
val b : int array array = [|[|0; 0; 0|]; [|0; 0; 0|]; [|0; 0;
0|]|]
# b.(1).(2) <- 3;;
- : unit = ()
# b;;
- : int array array = [|[|0; 0; 3|]; [|0; 0; 3|]; [|0; 0; 3|]|]
```

- il y a *partage* entre les lignes de la matrice
- les matrices en Caml sont naturellement non rectangulaires

Chaînes et tableaux de caractères

- Les chaînes de caractères sont quelque part en mémoire

```
# let hat = "chateau";;
val hat : string = "chateau"
# hat.[3] <- 'p';;
- : unit = ()
# hat;;
- : string = "chapeau"
```

`hat.[3]` et non `hat.(3)`

Exemple: pile impérative

tiré du livre de Chailloux, Manoury, Pagano

"Développement d'applications avec Objective Caml"

DÉMO `stacks.ml`

c'est presque du (C/Pascal), à la syntaxe près

D'autres effets de bord: entrées/sorties

saisie et affichage: effets de bord sur les périphériques (modif. tampon du clavier, modif. d'un fichier, ...)

`in_channel, out_channel`: types pour les *canaux*

```
print_string :  string -> unit
read_line :  unit -> string    read_int,...
```

`Scanf.scanf`

pas de manière simple d'afficher (et encore moins de saisir) des objets ayant un type somme

Impression formatée:

```
 # printf;;
- :  ('a, out_channel, unit) format -> 'a = <fun>
# printf "alors %d et %d\n" 9 11;;
alors 9 et 11
```

ça cache quelque chose: DÉMO `printf_caml.ml`