# Références

- les *"pointeurs"* de Caml
  - `let a = ref 3`   $\texttt{a} \rightarrow \boxed{3}$
  - `a` est l'adresse d'une case mémoire

# Références

- les *"pointeurs"* de Caml
  - `let a = ref 3`    a $\longrightarrow$ $\boxed{3}$
  - `a` est l'adresse d'une case mémoire
- déréférenciation: opérateur `!`  (BANG), lire la case mémoire
  - `f !a`  par exemple

# Références

- les *"pointeurs"* de Caml
  - `let a = ref 3`    a $\longrightarrow \boxed{3}$
  - `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
```

# Références

- les *"pointeurs"* de Caml
  - `let a = ref 3`    a $\longrightarrow \boxed{3}$
  - `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"*

# 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 };;
```

# 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

# 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}
  ```

# Références – typage

- `ref` un *constructeur de type*

si `x : t ref` alors `!x : t`

# Références – typage

- `ref` 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"*

# Références – typage

- `ref` 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* $\boxed{\texttt{;}}$

  `# a := !a+1; 4+3;;` $\rightsquigarrow$ `- : int = 7`

  N.B.: la séquence "jette" une valeur: `3;4;;` $\rightsquigarrow$ *Warning*

# Références – typage (suite)

- **Corollaire:** en OCaml, seul un `t ref` s'évalue en une référence
  - *conséquence de la réduction du sujet*

# Références – typage (suite)

- **Corollaire:** en OCaml, seul un `t ref` s'évalue en une référence
  - *conséquence de la réduction du sujet*
  - les deux côtés d'une affectation peuvent résulter d'une évaluation en Caml

    `(f x) := (g y)` est ok si `(f x)` s'évalue en un `t ref`
    ...et `(g y)` en un `t`

# Références – typage (suite)

- **Corollaire:** en OCaml, seul un `t ref` s'évalue en une référence
  - *conséquence de la réduction du sujet*
  - les deux côtés d'une affectation peuvent résulter d'une évaluation en Caml

    `(f x) := (g y)` est ok si `(f x)` s'évalue en un `t ref`
    …et `(g y)` en un `t`

- `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();;
```

# Références – typage (suite)

- **Corollaire:** en OCaml, seul un `t ref` s'évalue en une référence
  - *conséquence de la réduction du sujet*
  - les deux côtés d'une affectation peuvent résulter d'une évaluation en Caml

    `(f x) := (g y)` est ok si `(f x)` s'évalue en un `t ref`
    ...et `(g y)` en un `t`

- `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();;
```

$\longrightarrow$ 19 *: le résultat dépend de l'implémentation de* `+`

# Un certain style avec les références

des compteurs:

```
let new_compt init =
 let compt = ref init in
 let reset () = compt:=0 in
 let incr () = compt := !compt+1 in
 let get () = !compt in
reset,incr,get
```

# Un certain style avec les références

des compteurs:

```
let new_compt init =
 let compt = ref init in
 let reset () = compt:=0 in
 let incr () = compt := !compt+1 in
 let get () = !compt in
reset,incr,get
```

on simule un style *“orienté objet”* par l’intermédiaire d’une notion d’*état interne;*

le mécanisme de *portée* est utilisé pour *encapsuler* l’état

# Deux mots sur C

- 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;...}`$_{\dagger a}$

# Deux mots sur C

- 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 (...)` $_{\dagger a}$

# Deux mots sur C

- 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

# 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, . . . )

# 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`

# 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

# Tableaux en Caml

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

- création:

```
 # let t = [| 1 ; 2 |];;
val t :  int array = [|1; 2|]
```

# 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|]
```

# 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 = ()
```

# 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|]
```

# 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 = $e_1$ to $e_2$ do $e_3$ done

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

# 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`

# Exemple: pile impérative

tiré du livre de Chailloux, Manoury, Pagano

*"Développement d'applications avec Objective Caml"*

`http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/index.html`

*(ce lien est disponible depuis la page du cours)*

DÉMO `stacks.ml`

# Exemple: pile impérative

tiré du livre de Chailloux, Manoury, Pagano

*“Développement d’applications avec Objective Caml”*

`http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/index.html`

*(ce lien est disponible depuis la page du cours)*

$\boxed{\text{Démo}}$ `stacks.ml`

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

# Chaînes et tableaux de caractères

- Les chaînes de caractères sont quelque part en mémoire

```
# let hat = "chateau";;
val hat :  string = "chateau"
```

# 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 = ()
```

# 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)`

# 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

# Transparence référentielle

- retour sur l'évaluation: deux manières de parler de l'exécution des programmes
  - à petits pas: $P \rightarrow P'$
  - à grands pas: $\mathrm{eval}(P) = V$

# Transparence référentielle

- retour sur l'évaluation: deux manières de parler de l'exécution des programmes
  - à petits pas: $P \rightarrow P'$
  - à grands pas: $\mathrm{eval}(P) = V$
- implicitement, une notion d'*égalité*
  - on peut remplacer $P$ par $P'$ ou $V$
  - `32+26*2`, `32+52` et `84` *dénotent* la même valeur (entière)
- *une notion d'égalité sur les programmes?*
  - on sait raisonner en maths sur $aX^2 + bX + c = 0$, et sur des équations bien plus compliquées
  - que signifie écrire (voire résoudre) des équations entre programmes?

# Transparence référentielle

- retour sur l'évaluation: deux manières de parler de l'exécution des programmes
  - à petits pas: $P \rightarrow P'$
  - à grands pas: $\text{eval}(P) = V$
- implicitement, une notion d'*égalité*
  - on peut remplacer $P$ par $P'$ ou $V$
  - `32+26*2`, `32+52` et `84` *dénotent* la même valeur (entière)
- *une notion d'égalité sur les programmes?*
  - on sait raisonner en maths sur $aX^2 + bX + c = 0$, et sur des équations bien plus compliquées
  - que signifie écrire (voire résoudre) des équations entre programmes?
- si on avait un moyen de prouver que deux fonctions `f` et `g` sont égales *extensionnellement* ($\forall x.\, \mathtt{f}\, x = \mathtt{g}\, x$), on pourrait remplacer `f` par `g`

  on appelle cela la *transparence référentielle*

# Transparence référentielle

- retour sur l'évaluation: deux manières de parler de l'exécution des programmes
  - à petits pas: $P \rightarrow P'$
  - à grands pas: $\mathsf{eval}(P) = V$
- implicitement, une notion d'*égalité*
  - on peut remplacer $P$ par $P'$ ou $V$
  - `32+26*2`, `32+52` et `84` *dénotent* la même valeur (entière)
- *une notion d'égalité sur les programmes?*
  - on sait raisonner en maths sur $aX^2 + bX + c = 0$, et sur des équations bien plus compliquées
  - que signifie écrire (voire résoudre) des équations entre programmes?
- si on avait un moyen de prouver que deux fonctions `f` et `g` sont égales *extensionnellement* ($\forall x.\, \mathtt{f}\, x = \mathtt{g}\, x$), on pourrait remplacer `f` par `g`

  on appelle cela la *transparence référentielle*
- tout ceci n'est valable qu'en l'absence d'effets de bord

  `fun x -> x+1` $\neq$ `fun x -> (print_int x; x+1)`

# ML: fonctionnel et impératif

- en OCaml, on distingue entre aspects fonctionnels et aspects impératifs

- on peut en principe adopter un point de vue fonctionnel en C, mais rien n'est fait pour

- il existe une vision purement fonctionnelle de la chose:
  cf. monades et Haskell

# Dans l'approche, on est loin de la machine

- pas d'*instruction*: l'exécution est vue différemment, la récursivité est naturelle pour répéter des opérations
- *transparence référentielle:* égalités gravées dans le marbre, vraies éternellement (compositionalité)
- on "croit" à beaucoup de choses
  - les fonctions sont des objets de première classe
  - les booléens existent
  - on programme avec des définitions inductives (types somme)
  - on ne voit pas la mémoire (des références, mais pas d'adresses)
- on ne plaisante pas avec les **types**

# Dans l'approche, on est loin de la machine

- pas d'*instruction*: l'exécution est vue différemment, la récursivité est naturelle pour répéter des opérations
- *transparence référentielle:* égalités gravées dans le marbre, vraies éternellement (compositionalité)
- on "croit" à beaucoup de choses
  - les fonctions sont des objets de première classe
  - les booléens existent
  - on programme avec des définitions inductives (types somme)
  - on ne voit pas la mémoire (des références, mais pas d'adresses)
- on ne plaisante pas avec les **types**

  au total, Caml offre beaucoup d'abstractions ($\neq$ C)

voyons (un peu) ce qui se passe en-dessous

# Égalités structurelle et physique

- deux manières de regarder des objets
  - `=`: 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

  $\boxed{\text{Démo}}$ `alias.ml`, `recursive_record.ml`

# Égalités structurelle et physique

- deux manières de regarder des objets
  - `=`: 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`, `recursive_record.ml`

```
let x = ref 52;;
```

# Égalités structurelle et physique

- deux manières de regarder des objets
  - `=`: 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`, `recursive_record.ml`

```
let x = ref 52;;
let y = x;;
```

*et en shell?*

# Égalités structurelle et physique

- deux manières de regarder des objets
  - `=`: 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`, `recursive_record.ml`

```
let x = ref 52;;
let y = x;;
let z = ref !x;;
```

*et en shell?*
commande `ln`

# Égalités structurelle et physique

- deux manières de regarder des objets
  - `=`: 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`, `recursive_record.ml`

| | |
|---|---|
| `let x = ref 52;;` | *et en shell?* |
| `let y = x;;` | commande `ln` |
| `let z = ref !x;;` | commande `cp` |

# Égalités structurelle et physique

- deux manières de regarder des objets
  - `=`: 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`, `recursive_record.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 |

# Encore du partage

des matrices comme tableaux de tableaux

```
# let a = Array.make 3 0;;
```

# Encore du partage

des matrices comme tableaux de tableaux

```
# let a = Array.make 3 0;;
val a :  int array = [|0; 0; 0|]
```

# Encore du partage

des matrices comme tableaux de tableaux

```
# let a = Array.make 3 0;;
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|]|]
```

# Encore du partage

des matrices comme tableaux de tableaux

```
# let a = Array.make 3 0;;
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;;
```

# Encore du partage

des matrices comme tableaux de tableaux

```
# let a = Array.make 3 0;;
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|]|]
```

# 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

# Représentations internes

- les objets sont *construits* en mémoire

  DÉMO `construct.ml`

# Représentations internes

- les objets sont *construits* en mémoire

  Démo `construct.ml`
- différence entre valeurs **immédiates** et valeurs **référencées**

  Démo `immediat.ml`

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