Programmation

Nicolas Barnier, Pascal Brisset

ENAC

Février 2009

Plan

- Présentation historique
- Langage (lexique, syntaxe, compilation)
- Valeurs, contrôle de base (conditionnelle, séquence, appel fonctionnel), affectation
- Contrôle, fonctions récursives, boucles
- Structure de données, gestion de la mémoire
- Modularité, compilation séparée, entrées-sorties

Classification par modèle de calcul

Langage	Impératif	Fonctionnel	Déclaratif	Objet
Exemples	Assembleur Fortran Pascal Basic **C** Ada	LISP **ML** Miranda	Prolog CLP λProlog	C++ Eiffel Smalltalk Ada95 Java
Caract.	Mémoire Séquence	Fonction Application	Relation Recherche	Héritage
Modèle	Mach. de Turing	λcalcul	Logique	

Deux langages pour illustrer ce cours

- **C**
  - Langage **impératif**, Kernighan & Ritchie, 1978
  - Utilisé pour le développement des systèmes Unix

«

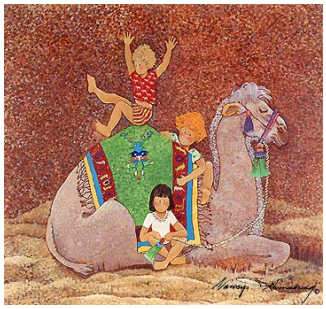
  - Assembleur » de haut niveau
- **Objective Caml**
  - Langage **fonctionnel**, Mac Carthy, 1958
  - Dialecte **Caml** (INRIA 1984) de **ML** (pour Meta Language, Milner 78)
  - Implémentation pour « micros » de X. Leroy, 1990

Pourquoi deux ?

- Les méthodes de programmations sont indépendantes des langages
  - **C** :
    - Leader pour la programmation “système”
    - Base pour **C**++ et Java
    - Efficacité
  - **O’Caml** :
    - Fondements théoriques sûrs
    - Nombreux aspects des langages : typage, modularité, objets, ...
    - Langage de référence pour l’initiation en France
- Pour aller deux fois plus vite...

Implémentations

- **C** : GCC, `http://gcc.gnu.org`
  - GNU C Compiler
  - GNU Compiler Collection : C, C++, Fortran, Java, Ada, ...
  - GNU : GNU's Not Unix (1984), « unix like free software »
  - Licence GPL
- **Objective Caml** : `http://www.ocaml.org`
  - INRIA : `http://www.inria.fr`
  - Multi-plateforme, multi-OS
  - Licence GPL



Mon premier programme

C	O'Caml
`/* Commentaire */`	`(* Commentaire *)`
`/* Integer global variable */` `int a = 1729;`	`(* Integer global variable *)` `let a = 1729;;`

`/* Integer function */`
`int f(int x) {`
`  return (x + a + 42);`
`}`
	`(* Integer function *)`
	`let f = fun x ->`
	`  x + a + 42;;`

Lexique

- Variable : `a`
- Fonctions, opérateurs : `f`, `+`
- Mot-clés : `int`, `return`, `let`, `fun`
- Caractères clés : `=`, `;`, `(`, `)`, `,` ,`->`, `;;`

- `1729` : nombre entier (séquence de chiffres)
- `3.14`, `1.5e9` : nombre flottant (séquence commencant par un chiffre ...)
- `"mach31"` : chaîne de caractères (suite de caractères encadrés par des guillemets)
- `'x'` : caractères (encadré par des *quotes*)
- `pi314` : identificateur (suite de caractères, lettre ou chiffre)
- Espace, tabulation, passage à ligne : séparateurs

Tokens

Description formelle (chapitre 6 pour **O'Caml**) :

ident : := (letter| _) { letter| 0...9| _ }

letter : := A ... Z — a ... z

integer-literal : := [-] { 0...9 }+

: := | [-] (0x| 0X) 0...9| A...F| a...f +

: := | [-] (0o| 0O) 0...7 +

: := | [-] (0b| 0B) 0...1 +

Valeurs

Valeurs manipulées par les langages de haut niveau :

- Valeurs de base
  - entier (32 bits)
  - nombre flottant, norme IEEE 754 (64 bits)
  - caractère (8 bits)
- Valeurs composées (tableau, structure)
- Fonction
- ...

Types

L'ensemble des valeurs est structuré en *types*.

- Booléen : `bool` (en **O'Caml**)
- Entiers : `int`
- Flottants : `float`
- Caractères : `char`
- Tableau d'entiers :
  - **C** : `int t[]`
  - **O'Caml** : `int array`
- Fonctions :
  - **C** : `int (int)`
  - **O'Caml** : `int -> int`

Typage

L'association entre identificateur (variable, fonction) et type est

- Explicite en **C** :
  - **Déclaration** des variables
  - **Déclaration** des fonctions : résultat et paramètres
- Implicite en **O'Caml** :
  - **Définitions** des variables
  - **Inférence** automatique du type des fonctions

Syntaxe (grammaire)

Construction syntaxique :

- Séquence de définitions de variables et de fonctions
- Une fonction est une séquence de définitions de variables puis d'instructions
- Instruction :
  - affectation de la valeur d'une expression à une variable
  - affichage de la valeur d'une expression
  - appel de fonction
- Expression :
  - valeur immédiate (entier, flottant, chaîne, ...)
  - appel de fonction

Sémantique : expressions

- entier, flottant, ... : interprété par sa valeur (dans $\mathbb{N}$, $\mathbb{R}$, ...)
- fonctions primitives (+, *) : valeur (dans $\mathbb{N} \rightarrow \mathbb{N}$, ...)
- appel fonctionnel : corps de la fonction où les paramètres sont remplacés par les arguments

```
let f = fun x -> x + 3;;
let a = 123 + f 12;;
```

$$\begin{aligned} E[\![\texttt{a}]\!] &= E[\![123 + \texttt{f } 12]\!] \\ &= E[\![123]\!] + E[\![\texttt{f } 12]\!] \\ &= 123 + E[\![12 + 3]\!] \\ &= 123 + 12 + 3 \\ &= 138 \end{aligned}$$

Sémantique : instructions

L'interprétation d'une séquence d'instructions nécessite

- une mémoire (état, environnement, ...)
- la lecture et la modification de cet état

```
int x = 1;
int y = 2;
x = y + x + 3;
y = x;
```

$$
\begin{aligned}
I[\![\texttt{int x ... x;}]\!]_{\{\}} &= I[\![\texttt{int y = 2; ...x;}]\!]_{\{x=1\}} \\
&= I[\![\texttt{x = y + x + 3; y = x;}]\!]_{\{x=1,y=2\}} \\
&= I[\![\texttt{x =}E[\![y + x + 3]\!]\texttt{; y = x;}]\!]_{\{x=1,y=2\}} \\
&= I[\![\texttt{y = x;}]\!]_{\{x=6,y=2\}} \\
&= I[\![]\!]_{\{x=6,y=6\}}
\end{aligned}
$$

Exécution

Différentes façons d'exécuter :

- Interprétation : évaluation directe du code
- Compilation : transformation du code vers
  - du code assembleur pour un processeur donné (*code natif*)
  - du code pour un pseudo-processeur, une machine virtuelle (*bytecode*)
- Compilation à la volée (*Just In Time*)

Le pour et le contre

	Interprétation	Compilation	
		Bytecode	Natif
Avantages	Simplicité Portabilité	Portabilité	Efficacité
Inconvénients	Performance		Spécificité
Compilateurs	`ocaml`	`ocamlc`	`ocamlopt` `gcc`

Activité de programmation

1. Écriture du programme *source* à l'aide d'un *éditeur* (Emacs)
2. Compilation : contrôle de propriétés statiques
   - Erreurs : doivent êtres corrigées
   - *Warnings* : 99.99% doivent êtres corrigés
3. Exécution : test de propriétés dynamiques

Style impératif

Modèle de Turing :

- Cases mémoire (numérotées : *adresse*)
- Lecture
- Écriture : *effets de bord*
- Séquence d'instructions ; l'ordre d'exécution est primordial

Désignation

On désigne les cases mémoire avec des variables.
La taille de la donnée est fixée par son type.

- **C** : initialisation non indispensable (*mais fortement recommandée*)

```
int a ; /* 32 bits */
float b = 3.14 ; /* 32 ou 64 bits */
char c ; /* 8 bits */
```

- **O'Caml** : initialisation nécessaire, on fabrique une *référence* vers une case mémoire

```
let a = ref 123 ; ; (* de type int, 32 bits *)
let b = ref 3.14 ; ; (* de type float, 64 bits *)
let c = ref 'x' ; ; (* de type char, 8 bits *)
```


Le choix des noms doit faciliter la lecture du programme

Un identificateur de variable (ou de fonction)

- commence par une lettre minuscule[1] ou un souligné
- contient
  - des lettres minuscules ou majuscules
  - des chiffres
  - des soulignés

Exemples :

- **Oui** : `x`, `x1`, `vitesse_avion`, `_anonyme`
- **Non** : `1x`, `position-navette`, `Georges`

[1] Les identificateurs du C peuvent commencer par une majuscule.

Lecture d'une variable

Le nom de la variable permet de retrouver la valeur associée

- **C** : simple mention du nom

```
int main() {
  printf("a = %d\n", a);
  printf("b = %f\n", b);
  printf("c = %c\n", c);
  return 0;
}
```

- **O'Caml** : avec l'opérateur préfixe ! pour une référence

```
Printf.printf "a = %d\n" !a;;
Printf.printf "b = %d\n" !b;;
Printf.printf "c = %d\n" !c;;
```


Affectation d'une variable

Une variable qui désigne un emplacement mémoire peut être **affectée**

- **C** : opérateur = appliquable à une *left value*

```
int main() {
  a = 124;
  b = b * 2.0;
  c = 'y';
  return 0;
}
```

- **O'Caml** : opérateur := appliquable à une *référence*

```
a := 124;;
b := !b *. 2.0;;
c := 'y';;
```

Portée des variables

Une variable a une **portée** limitée. Elle doit être la plus **locale** possible :

- Variable globale (en dehors d'une fonction) :
  - Visible (utilisable) dans toute la suite du fichier
  - **Sauf** si elle est *masquée* par une variable locale
- Variable locale
  - **C** : déclarée en début et visible dans un *bloc* (e.g. corps de fonction)

```
int x = 1;
int f() { int x = 2; x = 3; }
int g() { x = 4;  int x = 5;  }
int main() { f(); g(); printf("%d\n", x); return 0;}
```

  - **O'Caml** : définie avec un `let in`, visible jusqu'à la fin de la structure syntaxique englobante

```
let x = ref 1;;
let f = fun () -> let x = ref 2 in x := 3;;
let g = fun () -> x := 4; let x = ref 5 in ();;
f (); g (); Printf.printf "%d\n" !x;;
```


Contrôle

Enchaînement des instructions :

- Séquence
- Boucle
  - Bornée : `for`
  - Non bornée : `while`
- Conditionnelle
- Appel fonctionnel
- Exception

Séquence

- **C** : le ; est le **terminateur** d'instruction

```
int x = 2; x = x + 1; y = x + 1;
```

- **O'Caml** :
  - le ; est l'**opérateur** de séquence
  - le `let in` définit **et** réalise une séquence

```
let x = ref 2 in x := !x + 2; y := !x + 1
```


Boucle bornée

Répétition *n* fois, *n* fixé au départ, d'une suite d'instructions. Une boucle bornée termine (presque) toujours.

- **C** : (l'opérateur ++ incrémente un entier, -- décrémente)

```
int i;
for(i = 0; i < n; i++) { printf("%d\n", i); }
for(i = n; i > 0; i--) { printf("%d\n", i); }
```

Ne **jamais** modifier l'indice de la boucle **dans** la boucle.

- **O'Caml** :

```
for i = 0 to n - 1 do Printf.printf "%d\n" i done;
for i = n downto 1 do Printf.printf "%d\n" i done;
```

Une boucle non bornée ne termine pas si elle ne contient pas d'effet de bord.

Répétition tant qu'une condition est vérifiée.

- **C**

```
int i = 10;
while (i >= 0) i--;
```

- **O'Caml** (la fonction `decr` décrémente un entier, `incr` incrémente)

```
let i = ref 10 in
while !i >= 0 do decr i done
```


Conditionnelle

Base de la programmation : test puis exécution d'une instruction ou d'une autre

- **C** (% est l'opérateur de reste dans la division euclidienne)

```
if (x % 2 == 0) x = x + 2; else x++;
```

- **O'Caml** (`mod` est l'opérateur de reste dans la division euclidienne)

```
if !x mod 2 = 0 then x := !x + 2 else incr x
```

Appel fonctionnel

Fonction **appliquée** à des **arguments**

- **C** : parenthèses et virgule

```
x = f(x, y+1) + 2;
```

- **O'Caml** : juxtaposition

```
x := f !x (y+1) + 2
```


Définition d'une fonction

Paramètres et **corps**

- **C** : parenthèses et virgule

```
int f(int a, int b) { return ((a+b)/2); }
```

Le type de chacun des paramètres et du résultat doit être indiqué.

- **O'Caml** : *lambda* (théorie du λ-calcul)

```
fun a b -> (a + b) / 2
```

Aucun type n'est indiqué : le type entier est *inféré* grâce à la constante 2 et aux opérateurs + et /.

Mon premier programme impératif en C

```
#include <stdio.h>
#include <math.h>
float f = 1729.0;
float racine(float x, float epsilon) {
  float y = x;

  while (fabs(y * y - x) > epsilon) {
    y = (y + x / y) / 2.0;
  }
  return y;
}
int main() {
  printf("Racine de %.2f =~ %.2f\n", f, racine(f, 1e-2));
  return 0;
}
```


#include

Permet d'obtenir la visibilité sur des noms (variables, fonctions, types, ...) définis ailleurs.

- `stdio.h` : librairie standard d'entrées-sorties (*Input/Output*). Indispensable pour `printf`
- `math.h` : librairie standard mathématique. Pour `fabs`.

La documentation (man) mentionne les `include` nécessaires

Arithmétique

Typage : on ne mélangera pas les choux et les radis.

En **C** :

- les opérateurs arithmétiques sont les mêmes pour les nombres entiers et les nombres flottants
- en cas de mélange des deux dans une même expression, des conversions automatiques sont faites
- il est **fortement déconseillé** de faire de tels mélanges

```
# include <stdio.h>
int main() {
  printf("%f %f\n", 1.0 + 9/10, 1.0 + 9.0/10);
  return 0;
}
```


`main`

Un programme doit contenir une et une seule fonction `main`.

- C'est la première fonction appelée
- Elle peut avoir comme paramètres le nombre et les arguments du programme

  ```
  int main(int argc, char **argv) {
    printf("%d %s\n", argc, argv[1]);
    return 0;
  }
  ```

  affiche le nombre d'argument du programme (plus 1) et le premier.

printf

Fonction d'affichage formatté (`stdio.h`).

- Premier argument : chaîne de caractères contenant des directives
  - un caractère ordinaire **autre que** % est affiché simplement
  - le caractère `\n` est le passage à la ligne
  - `%d` permet d'afficher un entier en décimal
  - `%c` permet d'afficher un caractère
  - `%s` permet d'afficher une chaîne (string)
  - `%f` permet d'afficher un flottant
  - `%.2f` : flottant avec 2 chiffres après la virgule
  - ... (voir documentation)
- Les autres arguments sont les valeurs qui correspondent aux directives **dans le même ordre**.

Compilation et exécution (fichier `r.c`)

```
sepia[137]% gcc -o r r.c
sepia[138]% ./r
Racine de 1729.00 =~ 41.58
```

Mon premier programme impératif en **O'Caml**

```
let f = 1729.0;;

let racine = fun x epsilon ->
  let y = ref x in
  while abs_float (!y *. !y -. x) > epsilon do
    y := (!y +. x /. !y) /. 2.0
  done;
  !y;;

Printf.printf "Racine de %.2f =~ %.2f\n" f (racine f 1e-2);;
```


; ;

Un programme **O'Caml** est une séquence

- de *liaisons* `let`
  - de variables
  - de fonctions
- d'instructions

terminées par ; ; et évaluées dans l'ordre.

Une fonction (`fun`) n'est jamais évaluée
(si on ne l'applique pas à des arguments)

Variables

On utilise `ref` uniquement pour les variables que l'on veut pouvoir modifier.

- `f` est une constante
- les paramètres d'une fonction ne sont pas des références

On utilisera des références uniquement pour les valeurs modifiables

Arithmétique

- Les entiers et les flottants sont incompatibles
- Il n'y a jamais de conversion automatique
  (cf. `float : int -> float` et `truncate : float -> int`)
- Les opérateurs arithmétiques sur les flottants se terminent par le caractère `'.'`

Modules

- Chaque fichier est un *module* (réciproque fausse)
- Le fichier `f` se nomme module `F`
- Pour accéder à un nom `x` dans un module `M`, on écrit `M.x`
- La fonction `printf` est dans le module `Printf` de la librairie standard.

Compilation et exécution (fichier `r.ml`)

```
sepia[105]% ocamlc -o r r.ml
sepia[106]% ./r
Racine de 1729.00 =~ 41.58
```

Il faut distinguer

- Le langage
  - Lexique, syntaxe
  - Opérations basiques (arithmétique)
  - Mode d'exécution
  - Compilateur
- Les librairies : des fonctions, types, ... déjà écrits et *packagés*
  - Librairie de base : disponible automatiquement
  - Librairies *standard* : disponibles avec le compilateur
  - Librairies *autres* : au cas par cas

stdio.h	Entrées-sorties (`printf()`)
ctype.h	Tests de catégories de caractères (`isdigit()`)
string.h	Traitement des chaînes de caractères (`strcpy()`)
math.h	Mathématiques (`acos()`)
stdlib.h	Utilitaires (`atoi()`)
assert.h	Assertions (`assert()`)
stdarg.h	Nombre variables d'arguments
setjmp.h	Exceptions (`setjmp()`)
signal.h	Signaux (`signal()`)
time.h	Date (`time()`)
limit.h	Constantes (`INT_MAX`)

Les librairies d'**O'Caml** sont réparties en trois groupes :

- *Core* contient l'arithmétique sur les entiers (`int`), et sur les réels (`float`), le calcul booléen (`bool`) avec notamment les fonctions de comparaison, et des opérations d'entrée-sortie.
- *Standard* contient une trentaine de modules qui implémentent toutes les structures de données classiques, les affichages, ... ; les plus importants dans l'immédiat sont `Array`, `List`, `Printf` et `String`.
- Les *autres* librairies sont dépendantes de l'architecture : `Unix`, `Num` (nombres avec une précision arbitraire), `Graphics`. Elles nécessitent une commande de compilation particulière pour être utilisées (voir le manuel).

Une fonction `f` dans un module `M` est notée `M.f` .

- Pendant la compilation, il faut avoir *visibilité* sur la librairie :
  - `#include <stdio.h>`
  - `open Printf` **déconseillé** ou `Printf.printf` **recommandé**
- À la fin de la compilation, le programme est *lié* avec les librairies.
- Il est parfois nécessaire de mentionner explicitement une librairie à la compilation :
  - `gcc programme_qui_utilise_math.c -lm`
  - `ocamlc unix.cma programme_qui_utilise_unix.ml`

Caractères

- Type : `char`
- Allocation : un octet (8 bits)
- Constantes : entre apostrophes :
  - `'a'`, `'$'`, ...
  - `'\n'` (fin de ligne), `'\t` (tabulation), `'\\'`, ...
  - `'\`*xxx*`'` où *xxx* est un nombre
    - en octal en **C**
    - en décimal en **O'Caml**

man ascii

```
...
Oct   Dec   Hex   Char                Oct   Dec   Hex   Char
------------------------------------------------------------
000   0     00    NUL '\0'            100   64    40    @
001   1     01    SOH                 101   65    41    A
002   2     02    STX                 102   66    42    B
003   3     03    ETX                 103   67    43    C
004   4     04    EOT                 104   68    44    D
005   5     05    ENQ                 105   69    45    E
006   6     06    ACK                 106   70    46    F
007   7     07    BEL '\a'            107   71    47    G
010   8     08    BS  '\b'            110   72    48    H
011   9     09    HT  '\t'            111   73    49    I
012   10    0A    LF  '\n'            112   74    4A    J
...
```


Entiers

- Type : `int`
- Allocation : un *mot* de la machine (32 bits) dans $-2^{31}..2^{31}$
- Constantes
  - `31` en décimal
  - `0x1f` ou `0X1F` en hexadécimal
- Opérations : `+`, `-`, `*`, `/` et `%` (`mod` en **O'Caml**)

Nombres Flottants

Représentation approchée

- Type : `float` (`double` préférable en **C**)
- Allocation : 64 bits
- Constantes :
  - `3.14`
  - `6.55e-10`
- Opérations :
  - **C** : `+`, `-`, `*`, `/`
  - **O'Caml** : `+.`, `-.`, `*.`, `/.` et `**`
- Fonctions : `sin`, `log`, ...

Norme IEEE754 pour les flottants

Tous les calculs produisent un résultat :

- `1./0.` : *infinity*
- `1./0. - 1e10` : *infinity*
- `-1./0.` : *-infinity*
- `0./0.` : *nan*, *Not A Number*
- `sqrt(-1)` : *nan*
- `sqrt(-1) + 1.` : *nan*
- `1./0. * 0.` : *nan*

Booléens

- Type :
  - **C** : pas de type spécifique : `int`
  - **O'Caml** : `bool`
- Allocation : un *mot* de la machine
- Constantes :
  - **C** : `0` (faux), toute valeur non nulle (vrai)
  - **O'Caml** : `false`, `true`
- Opérations : `&&`, `||` et `!` (`not` en **O'Caml**)
- Comparaisons sur les types basiques : <=, <, >, >=, == (= en **O'Caml**), != (<> en **O'Caml**)

Conversions en C : *casting*

- Conversion implicite (À ÉVITER) : toujours vers le type le plus *grand*.
  - `('a' + 3)` convertit le `char` en `int`
  - `(3 + 3.14)` convertit le `int` en `float`
  - `float x = 'x';` convertit le `char` en `float`
  - `(3 / 2)` ne convertit rien
- Conversion explicite : `(`*nom-de-type*`)` *expression*
  - `(int)'a'` convertit le `char` en `int`
  - `(char)123` convertit le `int` en `char`
  - `(int)3.14` tronque le `float` en `int`

Conversions en **O'Caml** : fonctions classiques

Il n'y a JAMAIS de conversion implicite en **O'Caml**.

- `(Char.code 'a')` donne le code d'un caractère
- `(float 3)` transforme un `int` en `float`
- `(3 / 2)` est évalué à 1
- `(Char.chr 123)` donne le caractère d'un code donné
- `(int_of_float 3.14)` tronque le `float` en `int`
- `(fun x -> x <> 0)` convertit un `int` en `bool`
- `(fun x -> if x then 1 else 0)` convertit un `bool` en `int`

Chaîne de caractères en **C** : tableau de `char`

Valeur de taille quelconque.

- Type : `char []`
- Allocation : $n+1$ octets pour n caractères, le dernier est `'\0'` (implicite pour les constantes)
- Constantes : `"Hello\n"`, `"Coucou \"Georges\""`
- Accès : indexation par un entier, le premier d'indice 0.
- Ne contient pas sa taille.
- Exemple de conversion (tant qu'on a des chiffres)

```
int atoi(char s[]) {
  int i, n = 0;
  while(s[i] >= '0' && s[i] <= '9') {
    n = 10 * n + (s[i] - '0');
    i++;
  }
  return(n);
}
```

Chaîne de caractères en **O'Caml** : `string`

- Type : `string`
- Allocation : $n + 8$ octets pour n caractères
- Constantes : `"Hello\n"`, `"Coucou \"Georges\""`
- Accès : indexation par un entier, le premier d'indice 0.
- Taille : `String.length`
- Exemple de conversion (tous les caractères)

```
let int_of_string = fun s ->
  let n = ref 0 in
  for i = 0 to String.length s - 1 do
    n := 10 * !n + (Char.code s.[i] - Char.code '0')
  done;
  !n;;
```


Type sans information

Type des intructions, des boucles, ...

- **C** : `void`
  - Aucune valeur dans ce type
  - Type des fonctions sans `return` (procédures)
- **O'Caml** : `unit`
  - Une seule valeur : `()` (prononcer *unit*)
  - Type de ce qui se trouve à gauche d'un `;`
  - Type du paramètre d'une fonction qui n'en n'a pas

Récursivité

Répétition :

- Boucle bornée
- Boucle non bornée
- Appel fonctionnel : fonction récursive

Un appel fonctionnel à une fonction `f` est *récursif* s'il est situé dans le corps de la fonction `f`.
La plupart des langages de programmation autorise les appels récursifs : nécessité d'une pile.

Boucle infinie

- **C**

```
int main () {
  main ();
  return 0;
}
```

- **O'Caml**

```
let rec main = fun () ->
  main ();;
```

Le `rec` permet d'augmenter la portée du `let` à la définition elle-même.

⟶ `Stack Overflow`

Boucle contrôlée avec une conditionnelle

- **C**

```
int fact(int n) {
  if (n == 0)
    return 1;
  else
    return n * fact(n-1);
}
```

- **O'Caml**

```
let rec fact = fun n ->
  if n = 0 then 1 else n * fact (n - 1);;
```

Déduit de la définition récursive de la factorielle :

$$\begin{cases} 0! & = & 1 \\ n! & = & n * (n-1)! \end{cases}$$

Exécution

```
fact 2
if 2 = 0 then 1 else 2 * fact (2 - 1)
2 * fact (2 - 1)
2 * fact 1
2 * (if 1 = 0 then 1 else 1 * fact (1 - 1))
2 * (1 * fact (1 - 1))
2 * (1 * fact 0)
2 * (1 * (if 0 = 0 then 1 else 0 * fact (0 - 1)))
2 * (1 * 1)
2
```


Preuve de terminaison

Une fonction récursive doit **toujours** posséder un cas **non** récursif.

- Théorème de récurrence
- Induction
- Nécessité d'un ordre *noethérien* : pas de chaîne infinie décroissante

Fibonacci

$$\begin{cases} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-2} + F_{n-1} \end{cases}$$

- **C**

```
int fib(int n) {
  if (n == 0 || n == 1)
    return(1);
  else
    return(fib(n - 2) + fib(n - 1));
}
```

- **O'Caml**

```
let rec fib = fun n ->
  if n = 0 || n = 1
  then 1 else fib (n - 2) + fib (n - 1);;
```


Écrire une fonction récursive

- Plus *puissant* que les boucles bornées
- Ne nécessite pas d'affectation (pas de références)
- Raisonnement simplifié sur les programmes : pas d'*état* du calcul
- Consommation de mémoire *cachée* par l'usage de la pile

$\longrightarrow$ Style *fonctionnel*

- **Ne pas** imaginer les appels récursifs
- **Supposer** que la fonction renvoie le résultat attendu **avant** de l'avoir écrite : hypothèse de récurrence.

Itération : récursivité terminale

On a $\lim_{n\to\infty} u_n = \sqrt{x}$ avec

$$\begin{cases} u_0 &= 1 \\ u_{n+1} &= \frac{1}{2}(u_n + \frac{x}{u_n}) \end{cases}$$

C

```
float u(int n, float x) {
  if (n == 0) return 1;
  else {
    float un1 = u(n-1, x);
    return((un1 + x / un1) / 2.);
  }
}
```

O'Caml

```
let rec u = fun n x ->
  if n = 0 then 1.
  else
    let un1 = u (n-1) x in
    (un1 +. x/.un1)/.2.;;
```


Répétition (avec une fonction récursive) d'une transformation élémentaire :

$$u \to \frac{(u + \frac{x}{u})}{2}$$

où x reste constant mais est nécessaire :

$$(x, u) \to (x, \frac{(u + \frac{x}{u})}{2})$$

On obtient

```
void iter(float x, float u) {
  iter(x, (u + x / u) / 2.);
}
```

Comment s'arrête-t-on ?

Tant qu'une condition n'est pas atteinte

```
float iter(float x, float u) {
  if (fabs(u*u - x) > 0.01)
    return(iter(x, (u + x / u) / 2.));
  else
    return(u);
}

int main() {
  printf("%f\n", iter(2., 1.));
  return 0;
}
```


En comptant

$$(x, n, u) \rightarrow (x, n-1, \frac{(u + \frac{x}{u})}{2})$$

```
float iter(float x, int n, float u) {
  if (n > 0)
    return(iter(x, n-1, (u + x / u) / 2.);
  else
    return(u);
}

int main() {
  printf("%f\n", iter(2., 10, 1.));
  return 0;
}
```

Itération pour la factorielle

$$(0,1) \longrightarrow \ldots \longrightarrow (k, fk) \longrightarrow (k+1, (k+1) * fk) \ldots \longrightarrow (n, n!)$$

```
let rec iterer = fun k fk n ->
  if k = n
  then fk
  else iterer (k+1) (fk*(k+1)) n;;
let fact = fun n ->
  iterer 0 1 n;;
```


Imbrication

En utilisant une définition *locale* : la fonction `iterer` n'est utile que pour la fonction `fact`

```
let fact = fun n ->
  let rec iterer = fun k fk n ->
    if k = n
    then fk
    else iterer (k+1) (fk*(k+1)) n in
  iterer 0 1 n;;
```

Et en profitant du *contexte* de la définition, suppression du `n` de `iterer` :

```
let fact = fun n ->
  let rec iterer = fun k fk ->
    if k = n
    then fk
    else iterer (k+1) (fk*(k+1)) in
  iterer 0 1;;
```

Fonctions locales possible également en **C**

- Ne fait pas partie de la norme du langage
- Extension proposée par le compilateur `gcc` : dialecte
- Limitations
- Non recommandé

```
int fact(int n) {
  int iterer(int k, int fk) {
    if (k == n)
      return(fk);
    else
      return(iterer(k+1, fk*(k+1)));
  }
  return(iterer(0, 1));
}
```


La récursivité terminale est *gratuite*

Autre solution pour l'itération : test de fin simplifié

$$(n, 1) \longrightarrow \ldots \longrightarrow (p, np) \longrightarrow (p-1, np*p) \longrightarrow \ldots \longrightarrow (1, n!)$$

```
let fact = fun n ->
  let rec iterer = fun p np ->
    if p = 1
    then np
    else iterer (p-1) (p*np) in
  iterer n 1;;
```

Exécution :

```
fact 3
iterer 3 1
if 3 = 1 then 1 else iterer (3-1) (3*1)
iterer (3-1) (3*1)
iterer 2 3
if 2 = 1 then 3 else iterer (2-1) (2*3)
iterer (2-1) (2*3)
iterer 1 6
if 1 = 1 then 6 else iterer (1-1) (1*6)
6
```

Contrairement à la solution récursive directe, la taille de l'expression lors de l'évaluation est constante : *récursivité terminale*.

Assembleur généré : C

Les compilateurs *reconnaissent* les récursivités terminales.
Version récursive : `gcc -S fact.c` produit `fact.s`

```
fact:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
        cmpl $0,8(%ebp)
        jne .L3
        movl $1,%eax
        jmp .L2
.L3:
        addl $-12,%esp
        movl 8(%ebp),%eax
        decl %eax
        pushl %eax
        call fact
        addl $16,%esp
        movl %eax,%eax
        movl %eax,%edx
        imull 8(%ebp),%edx
        movl %edx,%eax
        jmp .L2
.L2:
        leave
        ret
```

Assembleur généré : **O'Caml**

Version itérative : `ocamlopt -S fact.ml` produit `fact.s`

```
F__iterer_56:
.L101:
        cmpl    $1, %eax
        jne     .L100
        movl    %ebx, %eax
        ret
.L100:
        movl    %ebx, %ecx
        movl    %eax, %ebx
        imull   %ecx, %ebx
        addl    $-1, %eax
        jmp     .L101
```


Exercice

Écrire :

- Une itération pour calculer Fibonacci
- Le code **C** et **O'Caml** correspondant
- Comparer avec la version récursive
  - Nombre d'étapes de calcul : complexité temporelle
  - Espace mémoire utilisé : complexité spatiale

Recursivité croisée

Récursivité *cachée* : `f` appelle `g` et `g` appelle `f`

- **C** : déclaration préalable de l'une des fonctions

```
int est_impair(int);
int est_pair(int  n) {
  return (n >= 0 && (n == 0 || est_impair (n-1)));
}
int est_impair(int  n) {
  return (n >= 0 && (n == 1 || est_pair (n-1)));
}
```

- **O'Caml** : `let` *en parallèle* avec le mot clé `and` :

```
let rec est_pair = fun n ->
  n >= 0 && (n = 0 || est_impair (n-1))
and est_impair = fun n ->
  n >= 0 && (n = 1 || est_pair (n-1));;
```


Itérateurs

Soit f l'étape élémentaire d'une itération, z l'élément initial. La répétition n fois s'exprime

$$f(f(...f(z)...) = f^n(z)$$

Une telle itération peut être *générique* : la fonction f et l'élément initial z deviennent les paramètre d'un itérateur général.

En **C** :

```
int iter(int f(int), int n, int z) {
  if (n == 0)
    return(z);
  else
    return(f(iter(f, n-1, z)));
}
int fois2(int x) { return(2*x); }
int puissance_de_2(int n) { return(iter(fois2, n, 1)); }
```

Itérateurs

En **O'Caml** :

```
let rec iter = fun f n z ->
  if n = 0 then z else f (iter f (n-1) z);;
let fois2 = fun x -> 2 * x;;
let puissance_de_2 = fun n -> iter fois2 n 1;;
```

O'Caml est mieux adapté au style fonctionnel :

- Une fonction peut être anonyme

  ```
  let puissance_de_2 = fun n -> iter (fun x -> 2 * x) n 1;;
  ```

- L'itérateur est *polymorphe* :

  ```
  let interets_a_10pourcents = fun n ->
    iter (fun x -> 1.10 *. x) n 1.;;
  ```


Itérateur *vs* boucle bornée

L'itérateur `iter` peut remplacer une boucle `for`. L'instruction

`for` *i* `=` *a* `to` *b* `do` *instruction sur i* `done`

est équivalente à

`iter (fun` *i* `->` *instruction sur i*`; i+1) (b-a+1) a`

En revanche, l'itérateur est plus puissant que les boucles bornées. La fonction d'Ackermann

```
let rec ack = fun m n ->
  if m = 0 then n + 1
  else if n = 0 then ack (m-1) 1
  else ack (m-1) (ack m (n-1));;
```

n'est pas calculable avec des boucles bornées mais s'exprime par une double itération.

Itérateur *vs* fonction récursive

L'avantage d'utiliser un itérateur par rapport à une fonction récursive est double :

- Concision d'écriture
- Preuve de terminaison *gratuite*

NB : l'itérateur `iter` peut être écrit sour forme récursive terminale :

```
let iter = fun f n z ->
  let rec iter = fun k fk ->
    if k = n
    then fk
    else iter (k+1) (f fk) in
  iter 0 z;;
```


Autre itérateur

On définit F à partir de f de la façon suivante (système T de Gödel) :

$$\begin{cases} F(0) & = & z \\ F(n+1) & = & f(n, F(n)) \end{cases}$$

```
let rec recurseur = fun f n z ->
  if n = 0 then z
  else f (n-1) (recurseur f (n-1) z);;
```

La factorielle est alors directement définie par :

```
let fact = fun n -> recurseur (fun x y -> (x+1)*y) n 1;;
let exp = fun e n -> recurseur (fun x y -> y * e) n 1;;
```

Tableaux

Ensemble d'éléments :

- de taille quelconque n *constante*
- homogènes : tous les éléments sont de même type
- indexés par des entiers : 0, ..., $n-1$
- modifiables

Type

- **C**
  - `int t[]` : pour un tableau `t` d'entiers
  - `int *` : dans certains cas, pour « simuler » un tableau (type *pointeur* vers une zone mémoire contenant des entiers consécutifs, cf. section 12)
- **O'Caml**
  - `int array` : pour un tableau d'entiers
  - `'a array` : (prononcer « α array ») pour un tableau d'éléments du même type quelconque `'a` (*variable de type*)

Accès

Tous les éléments sont accessibles avec leur index en temps constant
Les structures de contrôle adaptées sont

- La boucle bornée
- Les itérateurs

- **C** : un tableau ne contient pas sa taille

```
int somme(int n, int t[]) {
  int i, s = 0;
  for(i = 0; i < n; i++) s = s + t[i];
  return(s);
}
```

- **O'Caml** : un tableau contient sa taille

```
let somme = fun t ->
  let s = ref 0 in
  for i = 0 to Array.length t - 1 do s := !s + t.(i) done;
  !s;;
```

Le module `Array` de la librairie standard fournit les fonctions de manipulation sur les tableaux.
Itérateur `iter` : application en séquence d'une fonction à tous les éléments.

```
let somme = fun t ->
  let s = ref 0 in
  Array.iter (fun ti -> s := !s + ti) t;
  !s;;
```


Modification

Les éléments de tableaux sont modifiables en temps constant

- **C**

```
void cumul(int n, int t[]) {
  int i;
  for(i = 1; i < n; i++)
    t[i] = t[i] + t[i-1];
}
```

Modification *en place* de l'argument de la fonction.

- **O'Caml**

```
let cumul = fun t ->
  for i = 1 to Array.length t - 1 do
    t.(i) <- t.(i) + t.(i-1)
  done;;
```

Création, initialisation

Tableau statique : taille connue à la compilation

- **C**

```
int t[10];
float a[3] = { 1.5, 2.5, 3.5 };
```

Attention : un tableau alloué dans une fonction ne peut pas être renvoyé en résultat

```
int *fausse_allocation() { int t[10]; return(t); }
```

- **O'Caml**

```
let t = Array.create 10 0;;
let a = [| 1.5; 2.5; 3.5 |];;
```

Initialisation grâce à une fonction :

```
let carres = Array.init 10 (fun i -> i * i);;
```


Tableau dynamique : taille connue à l'exécution

```
int* entiers(int n) {
  int *t = (int*) malloc(n * sizeof(int));
  int i;
  for(i = 0; i < n; i++)
    t[i] = i;
  return(t);
}
```

La mémoire allouée avec `malloc` devra être libérée explicitement.

- `malloc` : allocation de mémoire (`stdlib.h`)
- `sizeof` : taille de la valeur d'un type donné (opérateur)
- `(int*)` : conversion explicite
- `free` : libération de mémoire allouée avec `malloc` (`stdlib.h`)

Allocation en **O'Caml**

Gestion automatique de la mémoire : *Garbage Collector*.

```
let entiers = fun n ->
  let t = Array.create n 0 in
  for i = 0 to n - 1 do
    t.(i) <- i
  done;
  t;;
```


Matrice : tableau de tableaux

En **C** : Types `int**`, `float**`, ...

```
float m[10][5];  /* Allocation statique */
 /* Allocation dynamique */
float **cree_matrice(int n, int m) {
  float **mat = (float**) malloc(n * sizeof(float*));
  int i;
  for(i = 0; i < n; i++) mat[i] = (float*) malloc(m * sizeof(float));
  return mat;
}
```

Accès par double indexation :

```
float trace(int taille_mat, float **mat) {
  float t = 0.;
  int i;
  for(i = 0; i < taille_mat; i++) t = t + mat[i][i];
  return t;
}
```

Manipulation des lignes :

```
void haut_en_bas(int taille_mat, float **mat) {
  int i;
  for(i = 0; i < taille_mat / 2; i++) mat[i] = mat[taille_mat - 1 - i];
}
```

Matrice

En **O'Caml** : Types `int array array`, `'a array array`, ...

```
let cree_fausse_matrice = fun n m ->
  Array.create n (Array.create m 0);;
let cree_matrice = fun n m init ->  (* Array.create_matrix *)
  let mat = Array.create n [||] in
  for i = 0 to n - 1 do
    mat.(i) <- Array.create m init
  done;
  mat;;

let trace = fun mat ->
  let t = ref 0. in
  for i = 0 to Array.length mat - 1 do
    assert(Array.length mat.(i) > i);
    t := !t +. mat.(i).(i)
  done;
  !t;;
let haut_en_bas = fun mat ->  (* De type 'a array -> unit *)
  let n = Array.length mat in
  for i = 0 to n / 2 - 1 do
    mat.(i) <- mat.(n - 1 - i)
  done;;
```


Un itérateur est souvent préférable à un `for`

Aucun risque de se tromper sur le nombre d'éléments :

- `(Array.map f [|x1; ...; xn|])` renvoit `[|f x1; ...; f xn|]`
- `(Array.fold_right f [|x1; ...; xn|] z)` renvoit `f x1 (f x2 (f ... (f xn z)...)`
- `(Array.fold_left f z [|x1; ...; xn|])` renvoit `(f (... (f (f z x1) x2) ... xn)`

Exemples :

```
let carres = fun a -> Array.map (fun x -> x * x) a;;
let produit = fun a -> Array.fold_right (fun x r -> x * r) a 1;;
let produit = fun a -> Array.fold_left (fun acc x -> x * acc) 1 a;;
let somme = fun a -> Array.fold_left (+) 0 a;;
```

Le copier-coller est rarement profitable en programmation

- Le nommage permet d'éviter les répétitions :
  - Utilisation d'une variable intermédiaire
  - La complexité du calcul peut être affectée drastiquement
  - Attention aux effets de bord
- Deux expressions semblables peuvent et doivent être factorisées :
  - fonction intermédiaire ;
  - passage en paramètre.

Expressions identiques

```
int f(int x, float y) { if (y > 0.) return (x*x+g(y)+1) ; else
return (x*x+g(y)-1) ; }
```

peut devenir

```
int f(int x, float y) { int x2gy = x*x+g(y) ; if (y > 0.) retu
(x2gy+1) ; else return (x2gy-1) ; }
```


En **C**, le *qualificatif de type* `const` indique qu'une variable est non modifiable :

```
int f(int x, float y) {
  const int x2gy = x*x+g(y);
  if (y > 0.)
    return (x2gy+1);
  else
    return (x2gy-1);
}
```

semblable en **O'Caml** à

```
let f = fun x y ->
  let x2gy = x*x + g y in
  if y > 0. then x2gy+1 else x2gy-1;;
```

Avec une fonction intermédiaire

- globale en **C**

```
int si(float y, int x2gy) {
  if (y > 0.)
    return (x2gy+1);
  else
    return (x2gy-1);
}
int f(int x, float y) {
  return (si(y, x*x+g(y)));
}
```

- anonyme en **O'Caml** :

```
let f = fun x y ->
  (fun x2gy -> if y > 0. then x2gy+1 else x2gy-1)
  (x*x + g y);;
```


La sémantique de

`let` *x* `=` *e1* `in` *e2*

peut s'exprimer

`(fun` *x* `->` *e2*`)` *e1*

Donc un `let` (ou plusieurs) peut (peuvent) être remplacé(s) par une fonction à un paramètre (plusieurs).

```
let f = fun x -> x+1;;
let g =
  fun y -> let y1 = y+1 in f (y1*y1);;
```

peut s'écrire

```
let g =
  fun y -> (fun f y1 -> f (y1*y1)) (fun x -> x+1) (y+1);;
```

Expression conditionnelle

Une conditionnelle n'est pas nécessairement une instruction mais peut être une expression :

```
let f = fun x y ->
  x*x + g y + (if y > 0. then 1 else -1);;
```

En **C**, la syntaxe est distincte :

- Instruction : `if ( condition ) alors else sinon;`
- Expression : `( condition ? alors : sinon )`

```
int f(int x, float y) {
  return (x*x+g(y) + ((y > 0.) ? 1 : -1));
}
```


Effets de bord

Les factorisations précédentes sont impossibles si l'expression factorisée effectue un *effet de bord* :

```
int h(int x, float y) { return(x*x+g(y)+1 + x*x+g(y)-1); }
```

- Affichage

```
int g(float y) {
  printf("y vaut %f", y);
  return (2 * y);
}
```

- Modification d'une variable globale :

```
int n = 0;
int g(float y) {
  n = n + 1;
  return (2 * y);
}
```

Expressions similaires

```
void h(float x, float y) {
  printf("%f\n", (x + k(x)) / sqrt(2 * x*x));
  printf("%f\n", (y + k(y)) / sqrt(2 * y*y));
}
```

doit devenir :

```
float xkx2x2(float x) {
  return ((x + k(x)) / sqrt(2 * x*x));
}
void h(float x, float y) {
  printf("%f\n", xkx2x2(x));
  printf("%f\n", xkx2x2(y));
}
```


Abstraction :

- nouvelle fonction ou fonction plus générale
- paramétrée par ce qui diffère dans les deux expressions semblables

Intérêts :

- Concision
- Lisibilité : le nom de la fonction *commente*
- Maintenabilité : une expression écrite 2 fois est 2 fois fausse

Ordre supérieur

On peut abstraire par rapport à n'importe quoi (ou presque) : un paramètre de la nouvelle fonction peut être une fonction

```
let produit = fun tab ->
  let p = ref 1. in
  for i = 0 to Array.length tab - 1 do
    p := !p *. tab.(i)
  done;
  !p;;
let somme = fun tab ->
  let s = ref 0. in
  for i = 0 to Array.length tab - 1 do
    s := !s +. tab.(i)
  done;
  !s;;
```


On abstrait par rapport à la différence :

```
let agregat = fun init op tab ->
  let p = ref init in
  for i = 0 to Array.length tab - 1 do
    p := op !p tab.(i)
  done;
  !p;;
let produit = fun tab -> agregat 1. ( *. ) tab;;
let somme = fun tab -> agregat 0. ( +. ) tab;;
```

Notation : la fonction correspondant à un opérateur infixe *op* se note (*op*).
Donc 1+2*3 peut s'écrire (+) 1 ((*) 2 3)

Même processus en **C** :

```
float agregat(float init, float op(float, float),
              int taille_tab, float tab[]) {
  float p = init; int i;
  for(i = 0;  i < taille_tab; i++)
    p = op(p, tab[i]);
  return(p);
}
float mult(float x, float y) { return(x * y); }
float produit(int taille_tab, float tab[]) {
  return agregat(1., mult, taille_tab, tab);
}
float plus(float x, float y) { return(x + y); }
float somme(int taille_tab, float tab[]) {
  return agregat(0., plus, taille_tab, tab);
}
```

Modularité

Un programme sera *structuré*

- en *fichiers* : unité de compilation
- en *modules* (équivalent à un fichier en première approximation)
- en *librairies* : ensemble de fichiers

Avantages :

- Taille raisonnable pour chaque fichier
- Espace de nommage
- Compilation séparée

Compilation, édition de liens

Deux phases (implicites jusqu'à maintenant) :

- Compilation : production d'un *objet* à partir du *source* ; option `-c` du compilateur
  - **C** : `gcc -c tp7.c` compile `tp7.c` et produit `tp7.o`
  - **O'Caml** : `ocamlc -c tp7.ml` compile `tp7.ml` et produit `tp7.cmo`
- Édition de liens : production d'un *exécutable* en *liant* un ou plusieurs *objets* et des librairies.
  - **C** : `gcc -o tp7 tp7.o` *lie* `tp7.o` à la librairie standard et produit `tp7`
  - **O'Caml** : `ocamlc -o tp7 tp7.cmo` lie `tp7.cmo` à la librairie standard et produit `tp7`

L'édition de liens fait la correspondance entre les appels aux fonctions de la librairie (`printf`, `atoi`, ...) et le code de ces fonctions.

Compilation séparée

Plusieurs fichiers `f1`, `f2`, `f3` ;

- **C** : **une seule** fonction `main()` pour tous les fichiers

```
gcc -c f1.c
gcc -c f2.c
gcc -c f3.c
gcc -o f f1.o f2.o f3.o
```

- **O'Caml** :

```
ocamlc -c f1.ml
ocamlc -c f2.ml
ocamlc -c f3.ml
ocamlc -o f f1.cmo f2.cmo f3.cmo
```


Édition de liens

Liens entre unités de compilation : chaque unité de compilation constitue un *espace de nommage* distinct : le nom `x` dans `f1` est différent du nom `x` dans `f2`.

Il est nécessaire de pouvoir utiliser une variable, une fonction, ... définie dans un autre fichier.

- **O'Caml** :
  - le nom `x` défini (avec un `let` **global**) dans le fichier `fic1.ml` est désigné `Fic1.x` partout en dehors de `fic1.ml`.
  - la directive `open Fic1` donne la *visibilité* sur tous les noms définis dans `fic1.ml`. Non recommandé.
- **C** : le mot clé `extern` permet de déclarer la visibilité sur un nom défini dans un **autre** fichier **sans préciser lequel**.

```
int main() {
  extern int x;
  extern float f(int, int);
  printf("%f", f(x, 12));
  return 0;
}
```


Gestion des dépendances en **O'Caml**

Soit `fic1.ml` définissant `x` et `fic2.ml` utilisant `Fic1.x`.

- Le fichier `fic1.ml` doit être compilé (avec l'option `-c`) **avant** `fic2.ml`
- Lors de la compilation de `fic2.ml`, le compilateur **vérifie** que le nom `x` est bien défini dans `Fic1`
- Lors de l'édition de liens de `fic2.cmo`, `fic1.cmo` doit être présent et placé **avant** dans la commande de compilation :

  `ocamlc -o executable ... fic1.cmo ... fic2.cmo ...`
- Conséquence : pas de dépendances croisées entre fichiers.

Fonctionnement : lors de la compilation de `fic1.ml`, `ocaml` produit `fic1.cmi` qui contient la description (i.e. le type) de tous les noms définis dans `Fic1`.

Gestion des dépendances en **C**

Soit `fic1.c` définissant `x` et `fic2.c` déclarant `x` externe.

- Les fichiers `fic1.c` et `fic2.c` peuvent être compilés (avec l'option `-c`) dans n'importe quel ordre (éventuellement en même temps)
- Lors de la compilation de `fic2.c` aucune vérification n'est faite sur `x`
- Lors de l'édition de liens de `fic2.o`, `fic1.o` doit être présent.
- La dépendance croisée est possible (mais n'est pas forcément souhaitable)

Fonctionnement : la description des noms est présente uniquement dans les objets (`.o`).

Restrictions sur les noms exportés

Par défaut, en **C** et **O'Caml** les noms globaux sont visibles par n'importe quel autre fichier.
Il est utile de pouvoir restreindre la visibilité sur un fichier.
C : déclaration avec le mot clé `static` des noms que l'**on ne veut pas exporter** :

```
static int ma_variable_a_moi;
static float ma_fonction_a_moi(int x, ...
```

O'Caml : définition d'une *interface* des valeurs exportées : `.mli` :

```
(* fic1.ml *)
let pi = 4 *. atan 1.;;
let ma_variable_a_moi = 7;;
let fonction_utile = fun x -> x + 1;;
let ma_fonction_a_moi = fun x y -> ...


(* fic1.mli *)
val pi : float;;
(** Que j'aime a faire apprendre ce nombre utile aux sages *)
val fonction_utile : int -> int;;
(** Cette fonction vous permettra d'arriver au sommet *)
```

Compilation :

1. `ocamlc fic1.mli`
2. Compilation de tout ce qui dépend de `Fic1`, `fic1.ml` y compris
3. Édition de liens

Fichier d'entête en **C**

Pour faciliter l'usage s'un fichier `fic1.c` son auteur devra écrire un fichier d'entête `fic1.h` (un *header*) correspondant :

- Ce fichier contient toutes les déclarations, avec le mot clé `extern` des noms qu'il veut exporter, et les commentaires correspondant
- Ce fichier est inclus (avec `#include`) dans `fic1.c` : ceci permet au compilateur de vérifier la compatibilité entre ce qui est déclaré dans le `.h` et défini dans le `.c`
- Les utilisateurs incluent cet entête pour utiliser ce qui est exporté par `fic1.c`

```c
/* fic1.c */
#include "fic1.h"
float pi = 3.14;
static int ma_variable_a_moi = 7;;
int fonction_utile(int x) { return x+1; }
static float ma_fonction_a_moi(int x, float y) { ... }
```

```c
/* fic1.h */
extern float pi;
/* Que j'aime a faire apprendre ce nombre utile aux sages */
extern int fonction_utile(int);
/* Cette fonction vous permettra d'arriver au sommet */
```

```c
/* fic2.c */
#include <stdio.h>
#include "fic1.h"
float surface(float r) { return pi*r*r; }
void test(int n) { printf("%d", fonction_utile(n)); }
```


Préprocesseur en C

Le `#include` est géré par un *prétraitement*, cpp, qui analyse les directives # avant que le source soit compilé :

- `#include` : insertion d'un fichier (habituellement un `.h`)
- `#define` : définition d'un *alias* qui va remplacer toutes les occurrences du nom par la valeur correspondante :

```c
#define TRUE 1
#define FALSE 0
#define bool int
bool est_pair(int n) {
   if (n % 2 == 0) return TRUE; else return FALSE;
}
```

- `#define` définition d'une *macro* : semblable à une fonction mais l'application (remplacement des paramètres par les arguments) est fait avant la compilation :

```
#define Est_pair(x) (x % 2 == 0)
int syracuse(int n, int k) {
  if (Est_pair(n)) syracuse(n/2, k+1); else ...
}
```

NB : Un entête (`.h`) peut définir des macros.
Convention :

- Macro pour une constante : en majuscules
- Macro avec des paramètres : commencant par une majuscule

- `#if`, `#else` `#endif` permettent de sélectionner à la compilation des lignes de code :

```
#if QUESTION1
int main() { printf("%d", exp1(...
#endif
#if QUESTION2
int main() { printf("%d", exp2(...
#endif
...
```

pourra se compiler en sélectionnant la fonction `main()` voulue.

```
gcc -DQUESTION1 -o tp7 tp7.c
```

ou

```
gcc -DQUESTION2 -o tp7 tp7.c
```

(Re)Compiler sans effort : Makefile

- Les commandes de compilations peuvent être longues et fastidieuses à saisir.
- Pour un programme constitué de plusieurs fichiers, il est rare de devoir **tout** recompiler après une modification
- On saisira dans un fichier `Makefile` (il s'appellera toujours ainsi) la description des opérations nécessaires pour la compilation.
- Ce fichier sera **exécuté** avec la commande `make`.
- Un `Makefile` est consitué de *règles* cible-dépendances-commande


```
# Makefile (les commentaires commencent par un dièse)
tp7 : tp7_1.o tp7_2.o
<tabulation>gcc -o tp7 tp7_1.o tp7_2.o

tp7_1.o : tp7_1.c
<tabulation>gcc -c tp7_1.c

tp7_2.o : tp7_2.c
<tabulation>gcc -c tp7_2.c
```

sera exécuté simplement avec la commande `make`.
Une règle :

```
cible :  dependances
         commande
```

exprime que les *dépendances* sont nécessaires pour fabriquer la *cible*. Si la *cible* est plus ancienne (comparaison des dates des fichiers) que l'une des *dépendances*, alors la *commande* est exécutée.

Structure de données : types utilisateur en **O'Caml**

- Pour chaque structure de données, on utilisera un **nouveau** type.
- Il faut pouvoir compléter les types de bases (types scalaires, chaînes, tableaux) avec des types nouveaux définis par le programmeur.

Type produit : conjonction de types

Représentation de **plusieurs** valeurs simultanément.
Exemple : les tableaux pour des valeurs de même type.
Prédéfini : **produit cartésien**

- Valeur (tuple) : $(val_1, val_2, \ldots, val_n)$
- De taille maximale 4194303
- Type : produit cartésien : $\tau_1 * \tau_2 * \ldots * \tau_n$
- Couples, triplets, quadruplets...

Exemples avec le *toplevel* :

```
#1, 2, 3;;
- : int * int * int = (1, 2, 3)

#let t = (2, "deux", '2');;
val t : int * string * char = (2, "deux", '2')

#let q = (1, "un", '1', t);;
val q : int * string * char * (int * string * char) =
  (1, "un", '1', (2, "deux", '2'))

#let div_et_reste = fun a b -> (a / b, a mod b);;
val div_et_reste : int -> int -> int * int = <fun>
```

Décomposition d'un tuple : *pattern matching*

On peut utiliser un *pattern* dans un `let` : il s'agit d'un *motif* de structure dont on nomme, avec des variables, les éléments que l'on veut récupérer (_ pour une variable anonyme) :

```
#let (a, b, c) = t;;
val a : int = 2
val b : string = "deux"
val c : char = '2'
#let (_, _, _, d) = q;;
val d : int * string * char = (2, "deux", '2')
```

Un *pattern* peut être arbitrairement complexe :

```
#let f = fun x ->
#  let (_, ((_,a,_), b), _, _) = x in a + b;;
val f : 'a * (('b * int * 'c) * int) * 'd * 'e -> int = <fun>
```

Le paramètre d'une fonction peut être également un *pattern* :

```
#let f = fun (_, ((_,a,_), b), _, _) -> a + b;;
val f : 'a * (('b * int * 'c) * int) * 'd * 'e -> int = <fun>
```


Tuples *vs* tableaux

- Taille statique
- Non homogène
- Non modifiable
- Accès *statique* aux éléments : pas d'index calculé

Enregistrement, structure, *record*

Inconvénient du produit cartésien (pour un annuaire) : confusion possible entre champs de même type :

```
#("Jean", "10 rue Balzac", 31000, "Toulouse");;
- : string * string * int * string =
("Jean", "10 rue Balzac", 31000, "Toulouse")
```

Généralisation du produit cartésien

- champs nommés
- un *nom* pour le type

Définition avec le mot clé `type` :

```
#type individu = {
#  nom    : string;
#  rue    : string;
#  cp     : int;
#  ville : string
#};;
type individu = { nom : string; rue : string; cp : int; ville : str
```

`individu` est un nouveau type. L'ordre des champs n'est pas significatif. `nom`, `rue`...sont les *étiquettes*. Une valeur pour ce type s'écrit :

```
#{ rue="Belin"; ville="Pau"; nom="Jean"; cp=31000 };;
- : individu = {nom = "Jean"; rue = "Belin"; cp = 31000; ville = "Pa
```

Un tuple ne peut pas être incomplet :

```
# { nom = "Toto" };;
 Some record field labels are undefined: rue cp ville
```

Une même étiquette ne peut être utilisée dans deux types différents :

```
# type numero = { nom : string; numero : int };;
# { nom = "J"; rue = "Belin"; cp=31000; ville="Pau"};;
 The record field label rue belongs to the type individu
but is here mixed with labels of type numero
```

Cette restriction :

- est nécessaire pour l'inférence de type :
- est résolue quand on répartit le code dans plusieurs fichiers :
  - le type `t` dans le fichier `f.ml` s'appelle `F.t`,
  - l'étiquette `e` du type `t` dans le fichier `f.ml` est notée `F.e`.

Accès aux champs

- Pattern-matching (filtrage de motif) :

```
#let code_postal = fun {nom=_; rue=_; cp=cp; ville=_} -> cp;;
 val code_postal : individu -> int = <fun>
```

ou

```
#let code_postal = fun {cp=code} -> code;;
val code_postal : individu -> int = <fun>
```

- Sélection :

```
#let code_postal = fun i -> i.cp;;
val code_postal : individu -> int = <fun>
```

Types somme (disjonction de types)

Comment regrouper dans un même type des valeurs de types distincts ?
Un type *somme* est une *union* finie étiquetée de types. Chaque étiquette de l'union est appelée *constructeur*. Chaque constructeur est un identificateur **commençant par une majuscule**.
Ex : identification d'une personne par un nom **ou** son numéro de sécu :

```
#type identification =
#    Nom of string
#  | Ss  of int64;;


#let i1 = Nom "Toto";;
val i1 : identification = Nom "Toto"

#let i2 = Ss 1760173623128L;;
val i2 : identification = Ss 1760173623128L
```

`i1` et `i2` sont de même type.

Supposons que le nom soit accompagné de la date de naissance et que le numéro de sécu soit associé à une clé.

```
#type numero_ss = {num: int64; cle:int};;

#type identification =
#    Nom of string * int
#  | Ss  of numero_ss;;
```

Les deux valeurs suivantes sont de même type :

```
#Nom ("Toto", 080176);;
- : identification = Nom ("Toto", 80176)

#Ss {num = 1760173623128L; cle = 23};;
- : identification = Ss {num = 1760173623128L; cle = 23}
```

On peut parler d'*injection de type* de `string * int64` (ou `numero_ss`) vers `identification`.

```
#type couleur = Coeur | Carreau | Pique | Trefle
#type carte = Valet of couleur | Dame of couleur
#           | Roi of couleur | Petite of int * couleur;;


#let tetes = fun c -> [|Valet c; Dame c; Roi c|];;
val tetes : couleur -> carte array = <fun>

#let petites = fun c -> Array.map (fun n -> Petite (n,c)) [|1;
val petites : couleur -> carte array = <fun>


#Array.append (tetes Trefle) (petites Trefle);;
- : carte array =
[|Valet Trefle; Dame Trefle; Roi Trefle; Petite (1, Trefle);
  Petite (7, Trefle); Petite (8, Trefle); Petite (9, Trefle);
  Petite (10, Trefle)|]
```


Reconnaissance des cas d'un type somme : *pattern-matching* (filtrage de motif)

La construction :

```
match expression with
   pattern_1 ->  result_1
|  pattern_2 -> result_2
...
|  pattern_n -> result_n
```

permet *d'essayer* plusieurs *patterns* : le premier qui réussit est sélectionné et l'expression correspondante est évaluée.

```
#let rouge_ou_noir = fun couleur ->
#  match couleur with
#    Coeur   -> "rouge"
#  | Carreau -> "rouge"
#  | _       -> "noir";;
val rouge_ou_noir : couleur -> string = <fun>
```

```
#let valeur = fun atout carte ->
#  match carte with
#    Petite (1,_)  -> 11
#  | Roi _         -> 4
#  | Dame _        -> 3
#  | Valet c       -> if c = atout then 20 else 2
#  | Petite (10,_) -> 10
#  | Petite (9,c)  -> if c = atout then 14 else 0
#  | _             -> 0;;
val valeur : couleur -> carte -> int = <fun>
```


Généricité

Rappel : **le copier-coller est rarement profitable en programmation**

Structure de données dont le type des éléments est indifférent :

```
#type paire_int = {e1 : int; e2 : int};;
```

```
#type paire_float = {r1 : float; r2 : float};;
```

```
#type paire_string = {c1 : string; c2 : string};;
```

Application d'une fonctions aux deux éléments d'une paire :

```
#let appl_int = fun f {e1=x; e2=y} -> f x y;;
```

```
#let appl_float = fun f {r1=x; r2=y} -> f x y;;
```

```
#let appl_string = fun f {c1=x; c2=y} -> f x y;;
```

On abstrait les types paire par rapport au type de leurs éléments.

```
#type 'a paire = {e1 : 'a; e2 : 'a};;
```

On a défini un type *générique*.

```
#let appl = fun f {e1 = x; e2 = y} -> f x y;;
val appl : ('a -> 'a -> 'b) -> 'a paire -> 'b = <fun>
```

On a défini une fonction *polymorphique paramétrique*.
Remarque : on obtient finalement des types différents

```
# e1 = 1; e2 = 2 = e1 = 1.; e2 = 2.;;
 This expression has type float paire but is here used with
type int paire
```



```
# type vache_qui_rit =
       {taille : float; boucle_d_oreille : vache_qui_rit};;
# {taille = 10.; boucle_d_oreille = {taille = 1. ; boucle_d...
```

Règle : un type récursif possède un cas non récursif.

```
#type vache_qui_rit =
#    Infinitesimale
#  | Boite of float * vache_qui_rit;;

#Boite (10., Boite (1., Boite (0.1, Infinitesimale)));;


#let rec profondeur = fun v ->
#  match v with
#    Infinitesimale -> 0
#  | Boite (_,b)    -> 1 + profondeur b;;
val profondeur : vache_qui_rit -> int = <fun>
```


Type récursif générique

Séquence d'éléments de même type :

```
#type 'a liste =
#    Nil
#  | Cons of 'a * 'a liste;;
```

Arbre binaire générique avec des feuille en `'a` et des nœud en `'b` :

```
#type ('a,'b) arbre2 =
#    Feuille of 'a
#  | Noeud of ('a,'b) noeud_binaire
#and ('a,'b) noeud_binaire = {
#    etiquette : 'b;
#    gauche:('a,'b) arbre2;
#    droit:('a,'b) arbre2
#};;
```

Arbre quelconque générique avec des feuille en `'a` et des nœud en `'b` :

```
#type ('a,'b) arbre =
#    F of 'a
#  | N of ('a,'b) noeud
#and ('a,'b) noeud = {
#  etiq : 'b;
#  fils : (('a,'b) arbre) liste
#};;
```

On préfèrera définir un nouveau type pour chaque utilisation.
Arbre pour une expression boolénne :

```
#type eb =
#    Vrai
#  | Faux
#  | Var of string
#  | Et of eb * eb
#  | Ou of eb * eb;;
```


Itérateurs

À chaque type correspond une fonction *intrinsèque*, son *itérateur* qui, pour un type à n constructeurs :

- est d'arité $n+1$;
- dont les n premiers paramètres sont des fonctions d'arité égale aux arités des n constructeurs ;
- est récursif si le type est récursif ;
- possède n cas.

L'itérateur permet de faire un traitement uniforme d'une donnée du type correspondant.

```
#let eb_iter = fun vrai faux do_var do_et do_ou expression ->
#  let rec iter = fun e ->
#    match e with
#      Vrai         -> vrai
#    | Faux         -> faux
#    | Var v        -> do_var v
#    | Et (e1, e2) -> do_et (iter e1) (iter e2)
#    | Ou (e1, e2) -> do_ou (iter e1) (iter e2) in
#  iter expression;;
#
#let compte_variable = fun e ->
#   eb_iter 0 0 (fun _ -> 1) (+) (+) e;;


#compte_variable (Et (Vrai, (Ou (Var "a", Var "b"))));;
- : int = 2
```


En résumé

- Il est possible de définir des nouveaux types
- Les types peuvent être paramétrés
- Les types peuvent être récursifs
- Type produit : produit cartésien étiqueté de types
- Type somme : disjonction de types
- À chaque type correspond un itérateur

Pointeurs

Adresse

En **C**, on peut gérer l'espace mémoire précisément. Il est toujours possible de savoir **où** une donnée est stockée, *i.e.* connaître son **adresse**.

- Une variable qui contient une adresse s'appelle un **pointeur**
- Le type d'un pointeur vers une donnée de type τ est $\tau*$
- On obtient l'adresse d'une donnée avec l'opérateur préfixe &
- On obtient la valeur à une adresse donnée avec l'opérateur préfixe $*$

Un $*$ sur une mauvaise adresse termine toujours mal.

Accès et modification d'une donnée

Par son adresse :

```
int x = 4;
int* adresse_de_x = &x;
int copie_de_x = *adresse_de_x
 *adresse_de_x = 5;
```

NB : on ne peut obtenir l'adresse que d'une variable, i.e. une *left-value*, quelque chose qui peut apparaître à gauche de l'opérateur d'affectation (=). Les expressions suivantes

- `&123`
- `&(2+2)`
- `&(fact(10))`

n'ont pas de sens.

Passage de paramètres *par valeur*, *par adresse*

```
void echange_rien(int x, int y) {
  int tmp = x;
  x = y;
  y = tmp;
}
void echange(int* x, int* y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}
int main() {
  int a = 12, b = 42;
  echange_rien(a, b);
  echange(&a, &b);
  return 0;
}
```

Le cas particulier du tableau

- Un tableau est en fait désigné par un pointeur vers son premier élément.
- Donc un tableau d'entiers est de type `int*`.
- Une chaîne de caractères est un tableau de caractères : `char*`.
- Le second argument de `main` est un tableau de chaînes : `char**`.

Arithmétique sur les pointeurs

Soit `p` de type `int*` : `*(p+1)` désigne l'entier qui suit l'entier `*p` dans la mémoire.
Donc pour `tab` de type `int*`, `tab[k]` et `*(tab+k)` sont identiques.
Calcul de la longueur d'une chaîne de caractères (dans `string.h`) :

```
int strlen(char* s) {
  int n = 0;
  char *p = s;
  while(*p != '\0') {
    p++;
    n++;
  }
  return(n);
}
```

On peut également calculer la différence de 2 pointeurs. Donc

```
int strlen(char* s) {
  char *p = s;
  while(*p != '\0')
    p++;
  return(p-s);
}
```

On remarque que `'\0'` correspond à l'entier `0` qui est le codage du booléen faux, donc on peut écrire :

```
int strlen(char* s) {
  char *p = s;
  while(*p)
    p++;
  return(p-s);
}
```


Types produit : structures

Équivalent aux enregistrements d'**O'Caml**.
Mot clé : `struct`.
Allocation à la main.
Désignation avec des pointeurs.

```
struct individu {
  char* nom;
  char* rue;
  int cp;
  char* ville;
};
```

Le type `struct individu` est appelé un type *produit*. L'ordre des étiquettes **est significatif** .

Initialisation statique :

```
struct individu id = {"Jean","10 r Balzac",31000,"Toulouse"};
```

Le compilateur accepte une initialisation partielle :

```
struct individu id2 = { "Paul", "6 allee Hugo" };
```

Une même étiquette **peut** être utilisée dans deux types différents.

```
struct numero {
  char* nom;
  int numero;
};
```

Contrairement à **O'Caml**, les champs d'une structure **sont modifiables**.

Accès aux champs par sélection

- Sélection dans la structure : opérateur `.` (point)
- Sélection à partir d'un pointeur vers la structure : opérateur `->` (flèche)

Donc `pointeur->etiquette` est un raccourci pour `(*pointeur).etiquette`.

```
int code_postal(struct individu* i) {
  return (i->cp);
}

void affiche(struct individu* i) {
  printf("%s\n%s\n%d%s\n", i->nom, i->rue, i->cp, i->ville);
  fflush(stdout);
}
```

`fflush` (dans `<stdio.h>`) vide le *buffer* d'un canal.

```
struct individu*
make_individu(char* n, char* r, int c, char* v) {
  struct individu* i =
    (struct individu*)malloc(sizeof(struct individu));
  i->nom = n; i->rue = r; i->cp = c; i->ville = v;
  return i;
}
```

On manipule en général un pointeur vers la structure et non pas la structure elle-même.
En résultat, ou en paramètre d'une fonction, on ne peut manipuler **que** des pointeurs.

Alias de type

Le mot-clé `typedef` permet de **nommer** un type, ce qui peut simplifier l'écriture.

```
typedef struct individu* ind;

ind make_individu(char* n, char* r, int c, char* v) {
  ind i = (ind)malloc(sizeof(struct individu));
  i->nom = n; i->rue = r; i->cp = c; i->ville = v;
  return i;
}
```

La construction est

`typedef` *type* *nom*;

Comparer avec

```
#define ind (struct individu*)
```

Types somme : énumération, union

Deux cas :

- Disjonction de constantes

  ```
  enum booleen { faux, vrai };
  ```

- Disjonction de types quelconques : *union*
  - Taille du type le plus gros de l'union
  - Pas de discriminant automatique

  Une personne est identifiée par son nom ou son numéro de sécu :

  ```
  union identification {
      char* nom;
      int ss;
  };
  ```

  regroupe sous un même type une chaîne et un entier ... mais ne permet pas de les distinguer.


```
enum nom_ou_ss { Nom, Ss };
struct ident_par_un_nom { char* nom; int date_de_naissance; };
struct ident_par_un_ss { int ss; int cle; };
struct identification {
  enum nom_ou_ss discriminant;
  union {
    struct ident_par_un_nom* par_un_nom;
    struct ident_par_un_ss* par_un_ss;
  } valeur ;
};

struct ident_par_un_nom toto = { "Toto", 230303 };
struct ident_par_un_ss monsieurX = { 176017362, 23 };
struct identification i1 = { Nom, &toto};
struct identification i2 = { Ss, &monsieurX};
```

i1 et i2 sont de même type.

Choix multiple : switch

Comparaison d'une valeur *scalaire* avec différentes constantes : valeurs testées dans l'ordre :

```
switch ( expression) {
  case valeur_1 :  traitement_1;
  case valeur_2 :  traitement_2;
...
  case valeur_n :  traitement_n;
  default :  traitement_par_défaut;
}
```

Attention : il faut terminer explicitement le traitement d'un cas sinon l'exécution se poursuit avec le cas suivant :

- `break` pour sortir du `switch` ;
- `return` pour sortir d'une fonction" ;
- fonction `exit` pour sortir d'un programme.


```
void affiche_identification(struct identification* i) {
  switch (i->discriminant) {
  case Nom : {
    struct ident_par_un_nom* ipn = i->valeur.par_un_nom;
    printf("%s/%d\n", ipn->nom, ipn->date_de_naissance);
    break;
  }
  case Ss : {
    struct ident_par_un_ss* ipss = i->valeur.par_un_ss;
    printf("%d/%d\n", ipss->ss, ipss->cle);
    break;
  }
  default:
    printf("Inconnu\n"); exit(1);
  }
}
```

```
enum couleur { Coeur, Carreau, Pique, Trefle };
enum genre_de_carte { Valet, Dame, Roi, Petite };
struct carte {
  enum couleur coul;
  enum genre_de_carte genre;
  int hauteur; /* utile seulement pour genre Petite */
};
char* rouge_ou_noir(struct carte* c) {
  switch (c->coul) {
  case Coeur:
  case Carreau:
    return "rouge";
  default:
    return "noir";
  }
}
```



```
int valeur(enum couleur atout, struct carte* c) {
  switch (c->genre) {
  case Petite:
    switch (c->hauteur) {
    case 1: return 11;
    case 10: return 10;
    case 9: return (c->coul == atout ? 14 : 0);
    default: return 0;
    }
  case Roi: return 4;
  case Dame: return 3;
  case Valet: return (c->coul == atout ? 20 : 2);
  }
}
int main() {
  struct carte valet_d_atout = { Coeur, Valet };
  printf("%d\n", valeur(Coeur, &valet_d_atout));
  return 0;
}
```

Généricité

Pas de généricité explicite en **C** ... mais une possibilité de casting.

```
struct paire = { void* e1; void* e2; };
struct paire p1 = { (void*) 12, (void*) 14 };
struct paire p2 = { (void*) "Hello", (void*) "Coucou" };

void *appl(void* f(void*, void*),  struct paire* p) {
  return f(p->e1, p-> e2);
}
```

Remarque : p1 et p2 ont le même type.

Types récursifs

La récursivité d'un type s'exprime grace à un pointeur :

```
struct vache_qui_rit {
  float taille;
  struct vache_qui_rit* boucle_d_oreille;
};
```

Un cas terminal nul s'exprime habituellement avec un pointeur nul :

```
typedef struct vache_qui_rit* vqr;
int profondeur(vqr v) {
  if (v) {
    return (1 + profondeur(v->boucle_d_oreille));
  } else /* pointeur nul : boite infinitesimale */
    return 0;
}
```

Listes

```
struct cons { int car ; struct cons* cdr; };
typedef struct cons* list;
const list nil = (list)0;
list cons(int x, list y) {
  list c = (list)malloc(sizeof(struct cons));
  c->car = x; c->cdr = y; return(c);
}
int car(list c) { return(c->car); }
list cdr(list c) { return(c->cdr); }
int main() {
  list c = cons(34, cons(25, nil));
  printf("%d %d\n", car(c), car(cdr(c)));
  return 0;
}
```

Gestion de la mémoire

Les données manipulées sont différemment représentées en mémoire :

- *Classe de stockage statique* (éventuellement *externe*)
- *Classe de stockage automatique* (pile)
- Allocation dynamique

Gestion de la mémoire allouée dynamiquement :

- *À la main* en **C**
- Automatique en **O'Caml**

Allocation statique en C

L'adresse de la donnée est fixée à la compilation. La mémoire correspondante est accessible depuis toutes les fonctions.

- Toutes les variables globales (déclarées en dehors des fonctions)
- Les variables locales qualifiées `static` : leur valeur est conservée d'un appel à l'autre.

Remarque : la *classe de stockage* `extern` déclare une variable globale déclarée *ailleurs*.

```
void f(int x) {
  static int var;
  printf("var=%d", var);
  var = x;
}
int main() {
  f(12); f(34); f(25);
  return 0;
}
```

produit l'affichage suivant

```
var=0
var=12
var=34
```

Attention : `static` sur une variable globale signifie qu'elle n'est pas utilisable depuis un autre fichier.

« Allocation statique » en **O'Caml**

```
#let f =
#  let var = ref 0 in
#  fun x ->
#    Printf.printf "var=%d\n" !var;
#    var := x;;
val f : int -> unit = <fun>
```

Comparer :

```
#let s1 =
#  fun r ->
#    let pi = acos (-1.) in
#    pi *. r *. r;;
#let s2 =
#  let pi = acos (-1.) in
#  fun r -> pi *. r *. r;;
```

Pile (allocation automatique)

L'adresse de la donnée dépend de la fonction dans laquelle elle est définie.

- Paramètre d'une fonction
- Variable locale (déclarée à l'intérieur d'une fonction)

Attention : la mémoire correspondante à une allocation automatique n'est utilisable que dans la fonction où la variable est déclarée.
Donc une fonction ne doit pas renvoyer l'adresse d'une variable déclarée localement ou d'un paramètre.

```
int* a_ne_pas_faire(int x) {
  return (&x);
}
float* a_ne_pas_faire_non_plus() {
  float pi = 3.14;  return (&pi);
}
```


Tas (*heap*) en **C**

L'adresse de la donnée est allouée dans une zone de mémoire associée au processus.
En **C**, cette mémoire :

- est allouée avec `malloc()` (dans `stdlib.h`) ;
- est accessible par toutes les fonctions ;
- doit être libéré explicitement avec `free` :

```
void supprime_premier(list* l) {
  if (*l) { /* Sinon c'est la liste vide */
    list p = *l;
    *l = (*l)->cdr;
    free(p);}}
```

Tas (*heap*) en **O'Caml**

En **O'Caml**, cette mémoire est allouée et libérée automatiquement (GC) :

```
#let f = fun n -> Array.create n 54;;
#let g = fun n -> (n, 3, 3.14);;
#type t = { x : int; y : int };;
#let zero = { x = 0; y = 0 };;
#type t' = Int of int | Float of float;;
#let to_int = fun i -> Int i;;
```


Récupérateur de mémoire (*Garbage Collector*)

Sommeil réparateur :

- Arrêt du programme quand toute la mémoire est consommée
- Marquage de la mémoire *utile* : parcours des pointeurs
- Suppression de la mémoire *inutile*
- Compactage de ce qui reste (la mémoire utile)

Quand ça ne marche pas

Pour *debugger* :

1. on peut ajouter des affichages pour observer le comportement du programme
2. il peut être nécessaire d'exécuter pas à pas.
   - Compilation et édition de lien avec l'option `-g`
   - Exécution avec le *debuggeur* :
     - **C** : `gdb program` ; voir `man gdb`
     - **O'Caml** : `ocamldebug program` ; voir `http://rafale`

Langage de commande

- `help` : pour obtenir de l'aide
- `run` : lancement du programme
- `step` : exécution d'une instruction **en rentrant** dans les appels fonctionnels
- `next` : exécution d'une instruction **sans rentrer** dans les appels fonctionnels
- `print` : affichage d'une variable
- `break` : positionnement d'un point d'arrêt (*breakpoint*)
- `quit` : pour sortir

Abréviation possible des commandes par un préfixe : par exemple `r` pour `run`.

gdb

Debugger indépendant du langage. Possibilité de tracer l'assembleur exécuté (si on a oublié l'option -g par exemple).

f.c

```
void echange(int* x, int* y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}
int main() {
  int a = 12, b = 42;
  echange(&a, &b);
  *(int*)1729 = 42;
  return 0; }
```

```
sepia[124]% gcc -g -o f f.c
sepia[125]% gdb ./f
...
(gdb) break echange
Breakpoint 1 at 0x80483aa: file f.c, line 2.
(gdb) run
Starting program: /home/sepia/brisset/cours/programmation/cours/f
Breakpoint 1, echange (x=0xbffff914, y=0xbffff910) at f.c:2
2         int tmp = *x;
(gdb) print *x
$1 = 12
(gdb) step
3         *x = *y;
(gdb) break f.c:4
Breakpoint 2 at 0x80483bc: file f.c, line 4.
(gdb) c
Continuing.
Breakpoint 2, echange (x=0xbffff914, y=0xbffff910) at f.c:4
4         *y = tmp;
(gdb) p y
$2 = (int *) 0xbffff910
(gdb) quit
A debugging session is active.
Do you still want to close the debugger?(y or n) y
```


En cas d'erreur d'exécution, un *état* est stocké dans un fichier core. Ce fichier peut être utilisé **a posteriori** avec gdb.

```
sepia[134]% ./f
Segmentation fault (core dumped)
sepia[135]% gdb ./f core
...
Core was generated by `./f'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  main () at f.c:9
9           *(int*)1729 = 42;
(gdb) where
#0  main () at f.c:9
```

ocamldebug

Debugger pour les programmes **O'Caml** compilés avec *ocamlc*.
Possibilité d'exécution *en avant* et *en arrière*.

f.ml

```
let a_zero = fun t ->
  for i = 0 to Array.length t do
    t.(i) <- 0
  done;;
let x = [|4;5;6|];;
a_zero x;;
```



```
sepia[141]% ocamlc -g -o f f.ml
sepia[142]% ocamldebug ./f
        Objective Caml Debugger version 3.04

(ocd) help break
break : Set breakpoint at specified line or function.
Syntax: break function-name
break @ [module] linenum
break @ [module] # characternum
(ocd) break @ F 2
Loading program... done.
Breakpoint 1 at 4996 : file F, line 2 column 3
(ocd) run
Time : 12 - pc : 4996 - module F
Breakpoint : 1
2   <|b|>for i = 0 to Array.length t do
(ocd) step
Time : 13 - pc : 5024 - module F
3     <|b|>t.(i) <- 0
(ocd) p i
i : int = 0
(ocd) p t
t : int array = [|4; 5; 6|]
```

```
(ocd) r
Time : 17
Program end.
Uncaught exception: Invalid_argument "Array.set"
(ocd) back
Time : 16 - pc : 5024 - module F
3     <|b|>t.(i) <- 0
(ocd) p i
i : int = 3
(ocd) goto 13
Time : 13 - pc : 5024 - module F
3     <|b|>t.(i) <- 0
(ocd) p i
i : int = 0
(ocd) quit
The program is running. Quit anyway ? (y or n) y
```

Structure de donnée de LISP (LISt Processing)

Le type :

```
#type 'a list =
#    Nil
#  | Cons of 'a * 'a list;;
```

est prédéfini :

- `Nil` est noté `[]`
- `Cons` est noté `::` **infixe**

```
#1::2::3::[];;
- : int list = [1; 2; 3]
```

```
#[[1] ; [1; 2]];;
- : int list list = [[1]; [1; 2]]
```



```
#[(+) ; (-) ; (/)];;
- : (int -> int -> int) list = [<fun>; <fun>; <fun>]
```

```
#[1] = 1::[];;
- : bool = true
```

```
#let rec fact = fun n ->
#  if n = 0 then 1 else n * fact (n-1) in
#[fact 1; fact 2; fact 3; fact 4];;
- : int list = [1; 2; 6; 24]
```

```
#[1; 2 ; 3] @ [4; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Attention : la concaténation n'est pas gratuite !

Parcours d'une liste

Une liste est vide ou ne l'est pas !

Le test doit se faire par *pattern-matching* :

```
#let premier = fun l ->
#  match l with
#    [] -> failwith "liste vide !"
#  | tete :: queue -> tete;;
val premier : 'a list -> 'a = <fun>
```


Type récursif et fonction récursive

La récursivité de la fonction est égale à la récursivité du type :

```
#let rec longueur = fun l ->
#  match l with
#    [] -> 0
#  | tete :: queue -> 1 + longueur queue;;
val longueur : 'a list -> int = <fun>
```

Construction d'une liste

Une liste se construit avec la tête et la queue.
Carré des éléments d'une liste (parcours et construction) :

```
#let rec carres = fun l ->
#  match l with
#    []    -> []
#  | n::ns -> n*n :: carres ns;;
val carres : int list -> int list = <fun>
```

Concaténation : $(a :: l_1) \cup l_2 = a :: (l_1 \cup l_2)$

```
#let rec conc = fun l1 l2 -> match l1 with
#    []    -> l2
#  | x::xs -> x :: conc xs l2;;
val conc : 'a list -> 'a list -> 'a list = <fun>
```


Renversement

Renversement naïf (complexité quadratique) : $\overline{(a :: l)} = \bar{l} \cup (a)$

```
#let rec renverse = fun l ->
#  match l with
#    []    -> []
#  | x::xs -> renverse xs @ [x];;
```

Renversement itératif (complexité linéaire) :

$$(l, ()) \longrightarrow \ldots \longrightarrow (a :: l, r) \longrightarrow (l, a :: r) \longrightarrow \ldots \longrightarrow ((), \bar{l})$$

```
#let renverse = fun l ->
#  let rec rev = fun l r ->
#    match l with
#      []    -> r
#    | x::xs -> rev xs (x::r) in
#  rev l [];;
val renverse : 'a list -> 'a list = <fun>
```

Itérateurs

Deux fonctions similaires sont des instances (cas particulier) d'une troisième.

```
#let rec carres = fun l ->
#  match l with [] -> [] | (n::ns) -> n*n :: carres ns;;
val carres : int list -> int list = <fun>
```

```
#let rec racines = fun l ->
#  match l with [] -> [] | (n::ns) -> sqrt n :: racines ns;;
val racines : float list -> float list = <fun>
```


On abstrait l'opération effectuée :

```
#let rec map = fun f l ->
#  match l with
#    []    -> []
#  | x::xs -> f x :: map f xs;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
#let carres = map (fun x -> x*x) and racines = map sqrt;;
```

Prédéfini :

```
#List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

L'itérateur à [x1;x2;...;xn] associe c x1 (c x2 ... (c xn n)...)

```
#let rec iter = fun cons nil l ->
#  match l with
#    []    -> nil
#  | x::xs -> cons x (iter cons nil xs);;
val iter : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>

#
#let somme = fun l -> iter (+) 0 l;;
val somme : int list -> int = <fun>

#somme [1;2;3];;
- : int = 6
```

Prédéfini :

```
#List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```


Version récursive terminale

```
#let rec fold_left f accu l =
#  match l with
#    [] -> accu
#  | a::l -> fold_left f (f accu a) l;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fu
```

List.fold_left f a [b1; ...; bn] calcule
f (... (f (f a b1) b2) ...) bn.

```
#List.fold_left ( * ) 1 [1;2;3];;
- : int = 6
```

Liste d'associations

La liste de couples est une représentation simple d'une association clé-valeur.
Exemple : annuaire de nom-numéro.

```
#let annuaire = [("A", 4053); ("B", 4142); ("C", 4282)];;
```

Recherche de la valeur associée à une clé :

```
#let rec assoc = fun k l ->
#  match l with
#    [] -> raise Not_found
#  | (k', v) :: kvs -> if k = k' then v else assoc k kvs;;
val assoc : 'a -> ('a * 'b) list -> 'b = <fun>

#assoc "B" annuaire;;
- : int = 4142
```

La fonction `raise` abandonne le calcul et *lève* une erreur.

On peut ajouter une condition quelconque à un pattern. Cette condition peut porter sur les variables du pattern : *pattern* `when` *condition*

```
#let rec assoc = fun k l ->
#    match l with
#    [] -> raise Not_found
#  | (k', v) :: kvs when k = k' -> v
#  | _ :: kvs -> assoc k kvs;;
val assoc : 'a -> ('a * 'b) list -> 'b = <fun>
```

Préfédini :

```
#List.assoc;;
- : 'a -> ('a * 'b) list -> 'b = <fun>
```

Traitement exceptionnel

- Division par zéro ?
- Échec du pattern-matching ?
- Situation innatendue détectée par le programme ?

Une *exception* est *levée* (*raised*).
Une exception est une valeur de type `exn`, prédéfinie ou déclarée :

```
#Failure "Ca c'est mal passe";;
- : exn = Failure "Ca c'est mal passe"

#exception Erreur of int;;
exception Erreur of int

#Erreur 1729;;
- : exn = Erreur 1729
```

Lever une exception

```
#raise;;
- : exn -> 'a = <fun>

#raise (Erreur 7);;
Exception: Erreur 7.
```

La levée d'une exception provoque l'abandon de l'évaluation courante :

- pour traiter une erreur ;
- pour sortir rapidement d'une évaluation " profonde ".

```
#let rec dernier = fun l ->
#  match l with
#    []    -> raise (Failure "liste vide")
#  | [x]   -> x
#  | x::xs -> dernier xs;;
val dernier : 'a list -> 'a = <fun>
```


Exceptions prédéfinies pour la librairie standard :

- `Match_failure (file, firstcharpos, lastcharpos)`
- `Assert_failure (file, firstcharpos, lastcharpos)`
- `Failure string` : erreur " générale "
- `Invalid_argument string` : par exemple pour l'indexation
- `Not_found` pour les fonctions de recherche
- `End_of_file` pour les fonctions de lectures
- `Division_by_zero` pour l'arithmétique

Fonctions prédéfinies :

```
let invalid_arg =  fun s -> raise (Invalid_argument s)
let failwith = fun s -> raise (Failure s)
```

```
#exception Zero;;

#let produit = fun t ->
#  let p = ref 1 in
#  for i = 0 to Array.length t - 1 do
#    if t.(i) = 0 then raise Zero;
#    p := !p * t.(i)
#  done;
#  !p;;


#produit [|1;2;3;0;4;5|];;
Exception: Zero.
```


Récupérer une exception

Pour pouvoir continuer :

- en rattrapant une erreur ;
- quand un calcul est interrompu par une exception.

`try` *expression* `with` *traitement par cas de type* `exn`

```
#let produit_ruse = fun l ->
#  try
#    produit l
#  with
#    Zero -> 0;;
 val produit_ruse : int array -> int = <fun>

#produit_ruse [|1;2;3;0;4;5|];;
- : int = 0
```

Une exception non récupérée est propagée à l'extérieur du `try with`.

```
#exception TropPetit;;

#exception TropGrand;;

#let f = fun x ->
#  if x < 0 then raise TropPetit
#  else if x > 0 then raise TropGrand
#  else failwith "f: nul";;

#let g = fun x ->
#  try (f x) with
#    TropGrand -> Printf.printf "g: %d trop grand\n" x;;

#let h = fun x ->
#  try (g x) with
#    TropPetit -> Printf.printf "f: %d trop petit\n" x;;
```



```
#h (-1);;
f: -1 trop petit
- : unit = ()

#h 1;;
g: 1 trop grand
- : unit = ()

#h 0;;
Exception: Failure "f: nul".
```

Typage : attention les cas « exceptionnels » doivent être de même type que les cas « normaux »

```
exception Resultat of int;;
let f = fun ... ->
  try
    while true do
      ...
      raise (Resultat n);
      ...
    done;
    0 (* ou failwith "inaccessible" *)
  with
    Resultat r -> r;;
```

Parce que les programmes manipulent des données

Lecture et écriture sur des *streams* ou *channels*.
Séquentialité : position implicite mise à jour automatiquement :

- ce qui a été lu ne peut pas être relu ;
- ce qui a été écrit ne peut pas être effacé.

Opérations :

- ouverture de fichier ;
- lecture, écriture ;
- déplacement ;
- fermeture de fichier.

Canaux

- **C**
  - *Stream* : type `FILE *`, par exemple `stdin`, `stdout` et `stderr` et 2
- **O'Caml**
  - Canal d'entrée : type `in_channel`, par exemple `stdin`
  - Canal de sortie : type `out_channel`, par exemple `stdout` et `stder`

Ouverture

- **C** : `man fopen`

```
FILE *fopen(const char *path, const char *mode);
```

  - `path` : nom de fichier
  - `mode` : `"r"` pour *read*, `"w"` pour *write*, `"a"` pour *append*, ...
  - le pointeur `NULL` est renvoyé en cas d'erreur : **doit être testé**

```
#include <stdio.h>
FILE * ouvre_pour_lire(char *nom_de_fichier) {
  FILE *f = fopen(nom_de_fichier, "r");
  if (f != NULL) return f;
  printf("%s ne s'ouvre pas en lecture\n", nom_de_fichier)
  exit(1);
}
```


- **O'Caml** : `core library`

```
#open_in;;
- : string -> in_channel = <fun>

#open_out;;
- : string -> out_channel = <fun>
```

Une exception est levée en cas d'erreur

```
#let ouvre_pour_lire = fun nom_de_fichier ->
#  try open_in nom_de_fichier with
#    exc ->
#      Printf.printf "%s ne s'ouvre pas en lecture\n" nom_de_f
#      raise exc;;
val ouvre_pour_lire : string -> in_channel = <fun>

#ouvre_pour_lire "toto";;
toto ne s'ouvre pas en lecture
Exception: Sys_error "toto: No such file or directory".
```

Écriture formatée

La fonction `printf` est un cas particulier de `fprintf` pour la sortie standard.

- **C** : `man fprintf`

  ```
  int fprintf(FILE *stream, const char *format, ...);
  ```

- **O'Caml** : module `Printf`

```
#Printf.fprintf;;
- : out_channel -> ('a, out_channel, unit) format -> 'a = <fu
```

On trouve dans `printf.ml` :

```
let printf = fun format -> fprintf stdout format;;
```


Lecture non formatée en **C**

- **C** : `man getc`

  ```
  int getc(FILE *stream);
  ```

  lit un caractère. Renvoie `EOF` sur une fin de fichier.

  ```
  char *gets(char *s);
  ```

  lit des caractères jusqu'à une fin de ligne ou la fin de fichier et les stocke à l'adresse `s` **qui doit pointer vers une zone mémoire allouée assez grande**. Le caractère de retour à la ligne n'est pas copié.
  Extrêmement dangereux : permet les attaques du type *buffer overflow*. Préférer :

  ```
  char *fgets(char *s, int size, FILE *stream);
  ```

  où on précise le nombre **maximal** de caractères à lire.

```
void wc(char * fichier) {
  int lines = 0, chars = 0;
  FILE * f = fopen(fichier, "r");
  if (f == NULL) {
    fprintf(stderr, "j'peux pas ouvrir %s\n",  fichier);
    return;
  }
  while (1) {
    char c = getc(f);
    if (c == EOF) {
      printf("%d caractères, %d lignes\n", chars, lines);
      fclose(f);
      return;
    }
    chars++;
    if (c == '\n') lines++;
  }
}
```


Lecture non formatée en **O'Caml**

- **O'Caml** : `core library`

```
#input_char;;
- : in_channel -> char = <fun>

#input_line;;
- : in_channel -> string = <fun>
```

lèvent l'exception `End_of_file` en fin de fichier. Ici, c'est `input_line` qui alloue la mémoire nécessaire.

Copie l'entrée standard jusqu'à la première ligne vide :

```
#let copie_jusqu_a_vide = fun () ->
#  let rec encore = fun () ->
#    let l = input_line stdin in
#    if l <> "" then begin
#      Printf.printf "%s\n" l;
#      encore ()
#    end in
#  try encore () with
#    End_of_file -> ();;
```

Lecture formatée en C

Similaire à `fprintf` : `man fscanf`

```
int fscanf(FILE *stream, const char *format, ...);
```

Format :

- conversion : `%d` pour un entier, `%f` pour un flottant, ...
- espace : pour un nombre quelconque d'espaces, de tabulations, de passages à la ligne
- autre caractère : pour lui-même

Attention : les arguments sont tous des pointeurs **valides** (passage par adresse).
Exemple : lecture d'une date écrite `"7  avril 2003  11h35"`

```
int jour, annee, heure, minute;
char mois[20];
fscanf(stdin, "%d %s %d %dh%d", &jour, mois, &annee, &heure,
```


Lecture formatée en O'Caml

Module `Scanf` (version $\geq$ `3.06`).
Utilisation de la librairie `Str` d'expressions régulières :

```
Str.regexp : string -> Str.regexp
Str.split : Str.regexp -> string -> string list
```

Lecture de la date :

```
let lit_date = fun f ->
  match Str.split (Str.regexp "[ \t]+") (input_line f) with
    [jour; mois; annee; heure; minute] ->
       let jour = int_of_string jour
       and annee  = int_of_string annee
       ...
  | _ -> failwith "Erreur lecture dans lit_date";;
```

Positionnement

Dans certains cas, il est possible de positionner la lecture ou l'écriture n'importe où dans le fichier : `seek`.

- **C** : `man fseek`

  ```
  long ftell(FILE *stream);  /* pour obtenir la position cou
  int fseek(FILE *stream, long offset, int whence);  /* pour
  ```

  Le `whence` précise par rapport à quoi on compte :
  - `SEEK_SET` : par rapport au début
  - `SEEK_CUR` : par rapport à la position courante
  - `SEEK_END` : par rapport à la fin
- **O'Caml** : *core library*

  `pos_in, pos_out, seek_in, seek_out`

Fermeture

- **C** :

  ```
  int fclose(FILE *stream);
  ```
- **O'Caml** :

```
#close_in;;
- : in_channel -> unit = <fun>

#close_out;;
- : out_channel -> unit = <fun>
```

- Tous les fichiers sont fermés à la sortie du programme
- Toutes les écritures sont bufferisées ; il peut être nécessaire de *flusher*
  - **C** :

    ```
    int fflush(FILE *stream);
    ```
  - **O'Caml** :

```
#flush;;
- : out_channel -> unit = <fun>
```