Programmation Fonctionnelle Avancée

Cours 1

kn@lri.fr
http://www.lri.fr/~kn

université
PARIS-SACLAY

Un mot sur l'organisation

- 12 séances de cours et TP
- Cours le mardi après-midi Petit Amphi du PUIO
- TP les vendredi après-midi et lundi
- Examen (en présentiel) en fin de semestre (décembre ou janvier)

TP Noté(s), interros : 40%, examen : 60% (à vérifier)

Au second semestre (UE séparée) Projet PFA parmi les choix possibles de projet

Plan

Définitions

Un *langage* est un système de communication **structuré**. Il permet **d'exprimer une pensée** et de **communiquer** au moyen d'un système de signes (vocaux, gestuel, graphiques, ...) doté **d'une sémantique**, et le plus souvent **d'une syntaxe**.

Un *langage de programmation* est un système de communication **structuré**. Il permet **d'exprimer un algorithme** de façon à ce qu'il soit **réalisable par un ordinateur**. Il est doté **d'une sémantique**, et **d'une syntaxe**.

Syntaxe et sémantique

La *syntaxe* est l'ensemble des **règles de bonne formation du langage**.

Exemple avec du code OCaml:

```
1 + 4          (* est syntaxiquement correct *)

1 / 'Bonjour' (* est syntaxiquement correct *)

1 +            (* est syntaxiquement incorrect *)
```

La *sémantique* est l'ensemble des règles qui donne le **sens** des programmes **bien formés**

```
1 + 4          (* est sémantiquement correct *)

1 / "Bonjour" (* est sémantiquement incorrect *)
```

Quels langages de programmation ?

La tour de Babel, Pieter Brueghel l'Ancien, 156

Quels sont les caractéristiques des langages de programmation ?

- Généralistes ou dédiés : certains langages ont un but spécifique (par exemple SQL pour interroger les bases de données) d'autres sont généralistes (comme C++, Python ou Java)

- Compilés ou interprétés : des langages compilés sont traduits par un **compilateur** en instructions machines (C, C++, Java, ...).
Les langages interprétés disposent d'un **interprète** qui permet d'évaluer des phrases du langages (le Shell, Python, JavaScript, ...).

- Typés dynamiquement ou statiquement

- Impératifs, fonctionnels, orienté objets, logiques, ...

- ...

Est-ce une bonne chose ?

Oui !

- Il n'y a pas un unique langage de programmation qui soit le meilleur
- Un bon programmeur ou une bonne programmeuse se doit de connaître **plusieurs** langages
- Connaître plusieurs langages permet d'aborder des problèmes de façon différentes
- Cela permet aussi de choisir le meilleur outil pour résoudre son problème

Sur le cycle de Licence, au moins 5 langages

- C++ (programmation impérative)
- Python (programmation impérative et objet)
- Java (programmation Orientée Objet)
- **OCaml** (programmation fonctionnelle)
- SQL (interrogation de bases de données)

Plan

Programmation Fonctionnelle ?

Traduction de l'anglais "Functional Programming" qui recouvre plusieurs concepts :

- Les fonctions sont des valeurs de première classe du langage :
  - On peut prendre des fonctions en argument
  - On peut renvoyer des fonctions comme résultat
  - On peut stocker des fonctions dans des structures de données
- Programmation récursive
  - Utilisation de fonctions récursives plutôt qu'utilisation de boucles
  - Utilisations de structures de données récursives (listes, arbres) plutôt que « plates » (tableaux)
- Programmation pure (sans effets de bords)
  - On prend des arguments et on renvoie des résultats plutôt que modifier des paramètres passés par référence/pointeurs
  - On renvoie des copies (partielles) des structures de données plutôt que des les modifier en place

Pourquoi est-ce utile ?

- Paramétrer un « objet » par un comportement :
  - Fonction de tri qui prend en argument la comparaison
  - Fonction qui permet de mettre en « cache » des résultats déjà calculés par une autre fonction (ex: resolution de noms)
  - Configurer une interface graphique pour appeler une certaine fonction lorsque l'on clique sur un bouton
- Être plus proche de l'algorithmique (ou des mathématiques) :
  - Écrire simplement des algorithmes complexes (type diviser pour régner)
  - Se convaincre (ou prouver !) qu'une implémentation est correcte
- Être efficace en mémoire (par le partage des structures communes)
- Introduire moins de bugs dans les programmes concurrents

En français, on appelle cela la programmation *applicative*

Programmation Fonctionnelle Avancée?

- Structures de données complexes tirant parti de la PF (arbres)
- Les ordinateurs fonctionnent par effet de bord : structures de données mutables, mélanges de programmation fonctionnelle et impérative
- Paramétrer chaque fonction (ex: des ensembles) par une fonction de comparaison ? Système de modules et de foncteurs d'OCaml
- Simple et efficaces : algorithmes récursifs et mémoisation
- Concepts théoriques pour la programmation avancée : monades (erreurs, prog. concurrente), transformations CPS (toute fonction récursive *terminale*)
- Éléments de génie logiciel: factorisation et réutilisatsion de code, outils de build, utilisation du typage comme outil de débuggage

Plan

Caractéristiques du langage

- Langage généraliste : traitement de données, interfaces graphiques, réseau, jeux, calcul scientifique, intelligence artificielle, ...
- Langage compilé (comme C++ ou Java)
- Langage typé **statiquement** (comme C++ ou Java)

Le langage supporte différents paradigmes : fonctionnel, impératif, orienté objet.

Dans ce rappel, on n'utilise que **le fragment fonctionnel**

Un premier programme

On considère le fichier `salut.ml`

```
let limit = 40                    (* On définit une variable globale *)
let () = Printf.printf "Quel est votre age ?\n" (* On affiche un message *)
let age = read_int ()     (* On lit un entier sur l'entrée standard *)
let msg =
  if age >= limit then            (* On teste la valeur  *)
     "vieux"
  else
     "toi"
let () = Printf.printf "Salut, %s!\n" msg
```

On peut compiler ce programme dans un terminal :

```
$ ocamlc -o salut.exe salut.ml
$ ./salut.exe
Quel est votre age ? 41
Salut, vieux !
$
```

Qu'y a t'il dans ce programme ?

- Définitions de variables
- Entrées (de l'utilisateur) et affichages (dans la console)
- Manipulation de constantes (nombres, textes, ...)
- Tests de valeurs
- Des commentaires

Le programme est **compilé** (comme Java ou C++)

Comment programmer avec OCaml ?

On privilégie pour les TPs l'éditeur VSCode

1. Ouvrir un terminal
2. Créer et se placer dans un répertoire pour le TP (par exemple `PFA/TP1`)
3. Lancer un VSCode :

```
$ code .
$
```

4. Créer des fichiers `.ml`, éditer le code, sauver le fichier
5. Tester le programme :

```
$ ocamlc -o exo1.exe exo1.ml
$ ./exo1.exe
```

Le site du cours contient des instructions pour installer OCaml sur votre machine.

La boucle d'interaction

Le langage OCaml possède aussi un **mode interactif** qui permet d'évaluer des instructions, comme un shell.

Il suffit de lancer la commande `ocaml` sans argument.

```
$ ocaml
        OCaml version 4.12.1

Down v0.0.4 loaded. Type Down.help () for more info.
# 1 + 1 ;;
- : int = 2
# 3 * 10 ;;
- : int = 30
# let x = 42 ;;
val x : int = 42
# x + 10 ;;
52
#
```

On peut quiter avec `CTRL-d`

Compilation vs. mode intéractif

Mode intéractif

- attend des expressions ou des définitions OCaml et les exécute au fur et à mesure.
- peut être utilisé pour tester des petits morceaux de programmes
- phrase rentrée dans ce mode interactif doit **obligatoirement** se terminer par ;;

Compilation

- Le programme `ocamlc` compile les fichiers sources en code-objet interprété par une machine virtuelle (comme Java)
- Le programme `ocamlopt` compile les fichiers sources en exécutables natifs (« binaires ») et effectue de nombreuses optimisations.

Programmation fonctionnelle

C'est un paradigme de programmation dans lequel :

- On ne fait pas d'effets de bords : pas de mise à jour de variables, de tableaux (sauf pour les entrées/sorties)
- On n'écrit pas de boucles, mais des fonctions **récursives**
- Les fonctions sont des objets de première classe (comme les entiers et les chaînes de caractères) :
  - On peut passer des fonctions en argument à d'autres fonctions
  - On peut renvoyer des fonctions comme résultat d'autres fonctions
  - On peut stocker des fonctions dans des structures de données
  - On peut définir des fonctions à n'importe quel endroit du code (dans d'autres fonctions en particulier)

C'est une **façon de programmer** particulièrement concise, puissante et qui peut être efficace. Elle vient compléter les autres styles de programmation : impératifs et orienté objet.

Programmation fonctionnelle en OCaml

Pourquoi faire de la programmation fonctionnelle **en OCaml** ?

TOUS les langages de programmation modernes supportent le paradigme fonctionnel :

- C++ (depuis C++11)
- Java (depuis Java 8, 2013)
- Javascript (proprement depuis 2015)
- Python (depuis Python 3)

Mais :

- Ils sont inutilement verbeux (Java, C++)
- Ils se contortionnent pour faire faire rentrer tout ça dans leur concepts de base comme les classes et les interfaces (Java)
- Ils ne sont pas typés (Javascript, Python)
- Ils limitent arbitrairement la récursion (Python)

Au début ...

Les premiers TPs vont peut être paraître arides :

- On se concentre sur une syntaxe nouvelle
- On doit penser différement

Ils deviendront plus sexy au fur et à mesure qu'on avencera dans le langage (programmation système, graphique, ...)

Plan

Les entiers (`int`)

En OCaml, les entiers ont une taille fixe : 63bits sur une architecture 64bits ou 31bits sur une architecture 32bits (un bit est reservé dans chaque entier **en plus du bit de signe**) :

```
# 1 ;;
- : int = 1
# -149 ;;
- : int = -149
# 1234567891011 ;;
- : int = 1234567891011
```

Opération sur les entiers

Symbole	Description
+	addition
-	soustraction
*	multiplication
/	division entière
mod	modulo

```
# 1 - 9 ;;
- : int = -8
# 3 * 4 ;;
- : int = 12
# 5 / 3 ;;
- : int = 1
# 10 mod 2 ;;
- : int = 2
# 4 + 3.5 ;;
Error: This expression has type float but an
  expression was expected of type int
```

Erreur de type ?

En OCaml, les expressions ont **un type et un seul**. C'est aussi valable pour les fonctions et les opérateurs. `+` est l'addition entre entiers.

À l'inverse d'autres langages il n'y a pas de conversion implicite entre types, il faut utiliser des conversion explicites.

Appels de fonctions

En OCaml, on appelle une fonction en donnant simplement son nom, suivi des arguments sans parenthèse :

```
f 1 2 3 ;;   (* on appelle la fonction f sur 3 arguments *)

g 4 ;;       (* on appelle la fonction g sur un seul argument *)
g (2 + 2) ;; (* on appelle la fonction g sur 1 seul argument *)
```

Cette notation étrange sera justifiée dans le prochain cours

Les nombres à virgule (`float`)

En OCaml, les « nombres à virgule » ont une précision limitée. On les représente en utilisant la notation scientifique :

```
# 1.5 ;;
- : float = 1.5
# -12.3423e13 ;;
- : float = -123423000000000.0
# 1.55555555555555555555555
- : float = 1.5555555555555558
```

Remarque : `-12.3423e13` = -12.3423×10^{13} = `-123423000000000.0`

Attention : En OCaml, comme dans de nombreux langages, calculer avec des nombres à virgule (nombres *flottants*) peut provoquer des erreurs d'arrondi.

OCaml (comme C, C++, Java, Python, Javascript, ...) utilise le standard IEEE-754 pour les flottants. C'est aussi celui implémenté en matériel par les processeurs et les cartes graphiques.

Opérations sur les flottants

Symbole	Description
`+.`	addition
`-.`	soustraction
`*.`	multiplication
`/.`	division
`**`	puissance
`float` *i*	conversion `int`→`float`
`float_of_int` *i*	conversion `int`→`float`
`int_of_float` *f*	conversion `float`→`int`
`sqrt` *f*	racine carrée
`sin` *f*	sinus
`cos` *f*	cosinus
`tan` *f*	tangeante
`log` *f*	logarithme (naturel)
`log10` *f*	logarithme (base 10)

Opérations sur les nombres flottants

```
# 1.5 +. 1.5 ;;
 - : float = 3.
# 3.141592653589793 *. 2.0 ;;
- : float = 6.2831853071795862 3
# 10.5 /. 3.0 ;;
 - : float = 3.5
# 1.2 +. 1.2 +. 1.2 ;;
 - : float = 3.5999999999999964
# 4.5 ** 100.0 ;;
2.0953249170398633e+65
#  1.0 /. 0.0 ;;
 - : float = infinity
```

Chaînes de caractères (`string`)

On représente les « textes » par des **chaînes de caractères**.

```
# "Bonjour, ça va bien ?"
 - : string = "Bonjour, ça va bien ?"
```

On ne montre que quelques opérations sur les chaînes de caractères :

Symbole	Description
`^`	concaténation
`String.length` *s*	longueur

Entrées/sorties

On se contentera d'entrées et sorties simples :

- Affichage formatté avec `Printf.printf`
- Lecture d'entrées sur le terminal
- Accès aux arguments du programme

Dans un second temps, on verra comment lire et écrire des fichiers.

Printf.printf

La fonction `Printf.printf` est similaire à la fonction C du même nom. C'est une fonction `variadique` (nombre arbitraire d'arugments)

Le premier argument doit être une *chaîne de format* qui indique combien d'arguemnts lire ensuite et comment les afficher.
Dans cette chaîne les séquences suivantes sont spéciales :

- `%s` lit l'argument suivant qui doit être une chaîne et l'insère à cet endroit.
- `%d` lit l'argument suivant qui doit être un `int` et l'insère à cet endroit.
- `%f` lit l'argument suivant qui doit être un `float`

Exemple :

```
Printf.printf "Un entier: %d, une chaîne: \"%s\", un flottant: %f\n"
   42 "foo" 3.14;;
```

```
Un entier: 42, une chaîne: "foo", un flottant: 3.14
```

Quel type pour la fonction `Printf.printf` ?

Si on exécute la fonction `Printf.printf` dans le terminal quel est le type du résultat ?

```
# Printf.printf "1+1 ça fait %d!\n" 2 ;;
1+1 ça fait 2!
- : unit = ()
#
```

Le résultat est du type `unit`. Ce type contient une seule valeur spéciale notée `()`.

Il est utilisé par les fonctions qui ne renvoient pas de résultats (affichage par exemple) ou qui ne prennent aucun argument.

On peut le voir comme un équivalent de `void` en Java.

Lecture au clavier

Plusieurs fonctions permettent de lire des données saisies au clavier :

- `read_int` : permet de lire un entier
- `read_float` : permet de lire un flottant
- `read_line` : permet de lire une ligne de texte

Ces fonctions prennent `()` en argument

Arguments d'un programme

Dans les langages comme C ou Java, il y a une fonction principale `main`

Cette dernière reçoit en argument un tableau contenant les arguments passés au programme sur la ligne de commande.

Dans les langages sans fonction principale comme OCaml (mais aussi Python ou Javascript), les arguments sont stockés dans un tableau global. En OCaml se tableau est dans la variable globale. `Sys.argv`.

On peut accéder aux éléments d'un tableau avec la notation `t.(i)`.
Exemple :

```
if Array.length Sys.argv >= 1 then
   Printf.printf "Le premier argument est %s\n" Sys.argv.(1)
```

Le tableau contient toujours au moins une case, le nom du programme dans lequel on est (dans `Sys.argv.(0)`)

Plan

Structures d'un programme

Un programme OCaml est consituté d'une suite d'éléments, terminés par `;;`. Ces éléments peuvent être :

- Des définitions de variables gloables de la forme `let v = e`
- Des expressions sans résultats (par exemple des affichages) de la forme `let () = i`
- Des définitions de fonctions (voir plus loin)

Il n'y a pas de point d'entrée, un programme est exécuté dans l'ordre du fichier.

En OCaml il n'y a pas de notion de « d'instruction », il n'y a que des expressions. Certaines de ces instructions renvoient `()`, pour indiquer qu'elles ont eu un effet (affichage, écriture dans un fichier, ...)

if/then/else

Un test `if/then/else` est une **expression** dont l'évaluation renvoie la valeur de l'expression dans la branche `then` ou `else`

Les deux expressions de chaque branche doivent avoir **le même type**

Ainsi, on peut écrire :

```
1 + (if x > 42 then 3 else 4)
```

Cette expression renvoie `4` si x est plus grand que 42 et `5` sinon.
Si on compare du code C++/Java et du code OCaml

```
let y =
    if x > 42 then
      4
    else
      5
```

```
int y;
if (x > 42) {
    y = 4;
} else {
    y = 5;
}
```

if/then/else (2)

Si la branche `then` est du type `unit` (pas de résultat), alors on peut omettre la branche `else`

```
if e > 10 then
    Printf.printf "e est plus grand que 10!\n"
```

Si on veut mettre plusieurs instructions de type `Unit` à la suite, on peut utiliser les mots clés `begin` et `end` et **séparer** les expressions par des `;`.

```
if e > 10 then begin
   Printf.printf "e est plus grand que 10!\n";
   Printf.printf "Si si je vous jure !\n";
   Printf.printf "Il est vraiment plus grand!\n"  (* pas de ; ici *)
end
```

`begin` et `end` jouent le même rôle que `{` et `}` en Java.

Les booléens

L'algèbre de Boole (George Boole, 1847) est une branche de l'algèbre dans laquelle on ne considère que deux valeurs : `true` et `false`.
Les opérations sur ces valeurs sont la négation (`not`), le « ou logique » (`||`) et le « et logique » (`&&`).

On peut manipuler ces objets en OCaml, comme on le fait avec des entiers, des nombres à virgule ou des chaînes de caractères.

```
# true ;;
- : bool = true
# false ;;
- : bool = false
# not true ;;
- : bool = false
# true || false ;;
- : bool = true
# true && false ;;
- : bool = false
```

Les comparaisons

Les booléens servent à exprimer le résultat d'un **test**. Un cas particulier de test sont les comparaisons. Les opérateurs de comparaisons en OCaml sont :

Symbole	Description
=	égal
<>	différent
<	inférieur
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal

Attention : dans les premiers cours on ne comparera que des nombres. Les comparaisons d'autres types (chaînes de caractères par exemple) seront expliquées plus tard. Les comparaisons `==` et `!=` existent aussi, mais on les verra plus tard.

Les comparaisons (exemples)

Le résultat d'une comparaison est toujours un booléens (`True` ou `False`) :

```
# 1 + 1 = 2;;
- : bool = true
# 3 <= 10 ;;
- : bool = true
# let x = 4;;
val x : int = 4
# x > 3 && x < 8;;
- : bool = true
# x <> 4 ;;
- : bool = false
```

Définition de variables

Une **variable** est un moyen de donner un **nom** au résultat d'un calcul.
En OCaml, une variable est une suite de caractères qui commence par une lettre minuscule ou un « _ » et contient des lettres, des chiffres ou des « _ ».
On définit une variable avec le mot clé « `let` ».

```
# let x = 2 ;;
val x : int = 2
# let y = 3 ;;
val y : int = 3
# let z = x + y;;
val z : int = 5
```

Définition de variables locales

On peut définir des variables locales à une expression avec les mots clés `let … in`

```
# let x = 2  in x + x;;
- : int = 4
# let y = 3 ;;
val y : int = 3
# let z = 4 in z + y;;
- : int = 7
# x + y;;
Error: Unbound value x
```

L'expression `let x =` e_1 `in` e_2 permet de définir la variable `x` uniquement le temps du calcul de e_2. Elle prend tout son sens lorsqu'on la combine à d'autres expressions comme le `if/then/else`.

Définitions de variables locales (exemple)

On peut comparer les deux codes OCaml et Java :

```
let norm =
  if z > 10 then
        let x2 = x *. x in
        let y2 = y *. y in
        sqrt (x2 +. y2)
  else
     -1.0
```

```
double norm;
if (z > 10) {
   double x2 = x * x;
   double y2 = y * y;
   norm = Math.sqrt (x2 + y2);
} else {
   norm = -1.0;
}
```

Dans les deux cas, les variables `x2` et `y2` ne sont plus visibles en dehors du bloc `then`.

Plan

Définitions de fonctions

En OCaml, on définit une fonction aussi avec le mot clé `let`

```
let carre n = n * n

let aire_triangle base hauteur =
       base *. hauteur * 0.5

let a = aire_triangle 5.0 14.5
```

La syntaxe générale d'une fonction est :

```
let f x1 … xn =
                    e
```

où `e` est l'expression dont la valeur est renvoyée.
⇒ il n'y a pas de mot-clé `return` en OCaml.
Bien sûr, un fonction peut avoir un corps complexe avec des `let … in`, des `if/then/else`

Exemple : formattage d'une heure

On veut écrire une fonction qui prend en argument un nombre de secondes et renvoie une chaîne de caractères au format : `j h min s`

```
let format_time t =
    let j = string_of_int (t / (24 * 3600)) in
    let t = t mod (24 * 3600) in
    let h = string_of_int (t / 3600) in
    let t = t mod 3600 in
    let m = string_of_int (t / 60) in
    let s = string_of_int (t mod 60) in
    j ^ "j " ^ h ^ "h " ^ m ^ "m " ^ s ^ "s"

let s = format_time 145999
let () = Printf.printf "%s\n" s
 (* affiche 1j 16h 33m 19s *)
```

Fonctions récursives

On n'a pas vu comment faire des boucles. Hors la répetition de code est un pilier important de la programmation (et sa raison d'être initiale).

On peut contourner l'absence de boucles en écrivant des fonctions **récursives**. Une fonction récursive est une fonction qui s'appelle elle même.
Commençons par l'exemple standard de la factorielle, écrit en OCaml :

```
let rec fact n =
  if n <= 1 then
     1
   else
     n * fact (n-1)

let () = Printf.printf "fact 10 = %d\n" (fact 10)
(* Affiche 3628800 *)
```

On introduit des fonctions récursives avec le mot clé `let rec`

Écritures de fonctions récursives

Lorsqu'on écrit une fonction récursive, on distingue **TOUJOURS** deux types de cas

- Le ou les cas **de base** : ce sont les cas pour lesquel on n'effectue pas d'appel récursif, ils sont calculables directement
- Le ou les cas **récursifs** : ce sont les cas qui dépendent du calcul récursif

Lorsque l'on fait un appel récursif, l'argument doit toujours **« se rapprocher »** du cas de base.

```
let rec fact n =
  if n <= 1 then (* cas de base *)
     1
   else (* cas récursif *)
     n * fact (n-1)   (* on se rappelle sur n-1, donc on
                         arrivera à 1 ou 0 à un moment *)
```

Pour les premiers cours, les fonctions récursives seront toujours sur des entiers

Écriture de fonctions récursives (2)

On donne un autre exemple, la fonction `fizzbuzz` (utilisée comme « échauffement » dans beaucoup d'interviews techniques)

- La fonction énumère les entiers entre 1 et n
- Si `n` est un multiple de `3` la fonction affiche `Fizz`
- Si `n` est un multiple de `5` la fonction affiche `Buzz`
- Si `n` est un multiple de `3` et de `5` la fonction affiche `FizzBuzz`
- Dans les autres cas on n'affiche rien

```
let rec fizzbuzz_aux i n =
  if i <= n then (* cas récursif *)
     let i3 = i mod 3 = 0 in
     let i5 = i mod 5 = 0 in
     begin
      if i3 && i5 then Printf.printf "FizzBuzz\n"
      else if i3 then Printf.printf "Fizz\n"
      else if i5 then Printf.printf "Buzz\n";
      fizzbuzz_aux (i+1) n   (* on se rappelle sur (i+1) → n *)
     end
;;
```

Écritures de fonctions récursives (2)

Dans un premier temps, les fonctions récursives auront **toujours** la forme (pseudo-code) :

```
let rec f n … =
   if test sur n then
       cas de base
    else
       cas récursif, appel sur f (n±e)
```

Rappel

```
let rec fact n =
if n <= 1 then (* cas de base *)
 1
else (* cas récursif *)
 n * fact (n-1)   (* on se rappelle sur n-1, donc on
                     arrivera à 1 ou 0 à un moment *)
```

Observons l'exécution « `fact 6` ».

```
fact 6
6 * fact 5
5 * fact 4
    4 * fact 3
        3 * fact 2
            2 * fact 1
                1 ← cas de base
            2
        6
    24
  120
720
```

En mémoire

Dans les architectures modernes, une zone de la mémoire est allouée pour le programme et utilisée de façon particulière : **la pile d'exécution**.
C'est dans cette zone que sont allouée les variables locales à la fonction. Lorsque l'on fait un appel de fonction, le code machine effectue les instructions suivantes

- Empile les arguments de la fonction sur la pile
- Place l'adresse de l'instruction sur la pile
- Saute à l'adresse de la fonction

La fonction appelée :

- S'exécute pour calculer son résultat
- Lit l'adresse sauvegardée sur la pile et y revient

Conventions x86_64

On va illustrer le comportement des fonctions récursives avec du code assembleur intel 64 bits

Le code est tout à fait similaire sur d'autres architectures (MIPS, ARM, ...)

Quelques points à savoir :

- La pile « grandit » vers les adresses décroissantes (i.e. le sommet de la pile est une grande adresse et plus on empile, plus on se rapproche de 0)
- L'adresse du sommet de pile est stockée dans le registre `rsp`
- Le registre `rax` est celui dans lequel les fonctions écrivent leur valeur de retour

En mémoire

- L'appel à fact dans fact `fact` se trouve à l'adresse `0x012345`
- L'appel à fact dans main se trouve à l'adresse `0xabcde`.

```
pile          rax
 6
               720
```

```
fact:
    cmpq 1, %rsp[+8]     #compare n à 1
    jle   Lthen          #si <=, then
    movq %rsp[+8], %rax  # sinon copier n dans rax
    subq 1, %rax         # rax := rax - 1
    push %rax            # place n-1 sur la pile
    call fact            # appelle fact, la valeur de
                    # retour est dans rax
                    # ici : adresse 0x12345
    addq 8, %rsp         # on nettoie la pile
    imulq %rsp[+8], %rax #
    ret                  # on revient
Lthen:
    movq 1, %rax
    ret
main:
        push 6
        call fact
►        ...         #située à l'adresse 0xabcde
```

Stack overflow ?

La pile grandit à chaque appel récursif.

Si on fait trop d'appels (en particulier mauvais cas de base, on ne s'arrête jamais), la pile dépasse sa taille maximale autorisée ⇒ Erreur Stack Overflow

Par défaut sous linux, la pile fait 8192 octets.

Elle peut être agrandie par le système ou l'utilisateur (command `ulimit -s`)

Pour la factorielle, la solution n'est pas satisfaisante, on utilise 16000ko pour calculer `fact 1000` !

Récursion terminale

Considérons la fonction `fact_alt` :

```
let rec fact_alt n acc =
if n <= 1 then
 acc
else
 fact_alt (n - 1) (n * acc)
fact_alt 6 1
       fact_alt 5 6
           fact_alt 4 30
               fact_alt 3 120
                   fact_alt 2 360
                       fact_alt 1 720
                       720 ← cas de base
                   720
               720
           720
       720
 720
```

Récursion terminale (2)

La fonction `fact_alt` calcule son résultat « en descendant » :

- Elle utilise un argument auxiliaire `acc` dans lequel elle accumule les résultats partiels
- Lorsqu'on arrive au cas de base, le calcul est terminé
- Les « retours » d'appels récursifs ne font que propager le résultat

`fact_alt` est une fonction récursive terminale.

Une fonction est récursive terminale si tous les appels récursifs sont terminaux.

Un appel récursif est terminal si c'est la dernière instruction à s'exécuter.

Récursion terminale (3)

```
let rec fact_alt n acc =
if n <= 1 then
 acc
else
 fact_alt (n - 1) (n * acc)
```

```
let rec fact n =
if n <= 1 then
 1
else
 n * fact (n-1)
```

- Dans `fact_alt` l'appel récursif est terminal, c'est la dernière chose que l'on fait dans la fonction
- Dans `fact` l'appel récursif est non-terminal : il faut prendre la valeur qu'il renvoie et la multiplier par `n`. La multiplication s'effectue après l'appel récursif.

Le compilateur OCaml optimise les fonctions recursives terminales en les compilant comme des boucles, ce qui donne du code très efficace et qui consomme une quantité de mémoire bornée (et non pas une pile arbitrairement grande)

Récursion terminale (4)

On peut appliquer une technique générale pour transformer une boucle while en fonction récursive terminale. Soit le pseudo code (Java) :

```
 int i = 0;
int res = 0;
while (i < 1000) {
res = f (i, res);  //on calcule un résultat
                   //en fonction des valeurs
                   //précédentes et de l'indice de boucle
i = i + 1;
}
return res;
```

Le code OCaml correspondant :

```
let rec loop i limit res =
if i >= limit then res     (* return final *)
else
  loop (i+1) limit (f i res)

let r = loop 0 1000 0
```

Récursion terminale (fin)

- La programmation récursive a souvent la réputation d'être inefficace
- C'est lié à une utilisation abusive de récursion non terminale
- Il faut écrire des fonctions récursives terminales dès que l'on essaye de programmer des boucles `for` ou `while` sur des entiers.

Attention, certains problèmes nécessitent forcément d'utiliser de la mémoire. Une fonction récursive non-terminale pourra alors être élégante, alors qu'un code impératif devra utiliser une boucle et une pile explicite.

Plan

Inférence de types

Le compilateur OCaml effectue une inférence de types :

- Il devine le type des variables
- Il en déduit le type des expressions
- Il en déduit le type des fonctions

```
# let x = 2 ;;
val x : int = 2
# let y = 3 ;;
val y : int = 3
# let f n = n + 1;;
val f : int -> int = <fun>
# let g x = sqrt (float x);;
val g : int -> float = <fun>
```

Inférence de types (2)

Le compilateur affecte à chaque variable de programme une variable de type.

Il pose ensuite des équations entre ces variables de types

Si des équations sont contradictoires, OCaml indique une erreur de type

```
let f n  =
if n <= 1 then
 42
else
 n + 45
αn : type de n
αf : type de retour de f
αn = int (car n <= 1)
αn = int (car n + 45)
αf = int (car on renvoie 42)
αf = int (car on n + 45)
n : αn= int
f : αn -> αf = int -> int
```

```
let g n =
if n <= 1 then
 42
else
 "Boo!"
αn : type de n
αg : type de retour de g
αn = int (car n <= 1)
αg = int (car on renvoie 42)
αg = string (car on renvoie "Boo!")
n : int ≠ string ⇒ erreur de typage
```

Type des fonctions

Soit une fonction :

```
let f x1 … xn =
   ef
```

Le type de `f` se note : `t1 -> … -> tn -> tf` où :

- t_i est le type du paramètre x_i (pour $1 \leq i \leq n$)
- t_f est le type de retour de la fonction

Types de fonctions (exemples)

```
  let mult_add a b c =  a * b + c;; (* int -> int -> int -> int *)

let carte n =
if n = 1 then "As"
else if n = 11 then "Valet"
else if n = 12 then "Reine"
else if n = 13 then "Roi"
else if n > 13 then "invalide"
else string_of_int n
 (* int -> string *)

let us_time h m =
let s = if h > 12 then "pm" else "am" in
let h = if h > 12 then h - 12 else h in
Printf.printf "%d:%d %s" h m s
 (* int -> int -> unit *)

let to_seq j h m s =
float (j * 3600 * 24 + h * 3600 + m * 60) +. s
 (* int -> int -> int -> float -> float *)
```

Plan

Type structuré

Un **type structuré** est un type permettant d'associée plusieurs données de façon à les traîter comme un tout.

Un exemple de type structuré vu en L1 est le tableau :

- collection ordonnée d'éléments, de taille fixe
- les éléments sont accessibles par décalage par rapport au premier élément (indice)
- les « cases » du tableau sont modifiables

Nous allons voir plusieurs types structurés proposés en standard par OCaml et adaptés à la programmation fonctionnelle.

Type produit

Le type structuré le plus simple est celui des produits, aussi appelés n-uplets.
En OCaml, une expression n-uplet se note $(e_1, \ldots, e_n)$:

```
let div_mod a b =
   (a / b, a mod b)
```

Le type des n-uplets se note t_1 `*` … `*` t_2. Le nom de produit vient du « produit Cartésien » $A \times B$ entre deux ensembles.
La fonction `div_mod` ci-dessus a le type

```
int -> int -> int * int
```

Elle prend en argument deux entiers et renvoie une paire d'entiers.

Type produit (2)

Une façon commode de travailler avec les n-uplet est d'utiliser un `let` multiple.

```
let x, y = div_mod 10 3
let () = Printf.printf "%d/%d = %d et il reste %d\n" 10 3 x y
```

Une autre façon de faire est d'utiliser les fonctions prédéfinies : `fst` (pour *first*) et `snd` (pour *second*) :

```
let res = div_mod 10 3
let () = Printf.printf "%d/%d = %d et il reste %d\n"
           10 3 (fst res) (snd res)
```

Type produit (3)

Les n-uplets ne sont pas limités aux paires :

```
let hms s =
  let h = s / 3600 in
  let s = s / 3600 in
  let m = s / 60 in
  let s = s mod 60 in
    (h, m, s)

(* val hms : int -> int * int * int *)
```

Attention, dans ce cas on ne peut plus utiliser les fonctions `fst` et `snd` qui sont réservées aux paires. La notation `let` est à privilégier :

```
let h, m, s = hms 4932
let () = Printf.printf
"%d secondes font %d heure(s), %d minute(s) et %d seconde(s)\n"
 4932 h m s
(*Affiche:
4932 secondes font 1 heure(s) 22 minute(s) et 12 seconde(s)
*)
```

Produit nommé

Une variante du type produit est le **produit nommé** aussi appelé **enregistrement** (ou *struct* ou *record*).
En OCaml, un produit nommé doit être déclaré au moyen de l'instruction `type` :

```
type point = { x : float; y : float }

let origin = { x = 0.0; y = 0.0 }

let dist p1 p2 =
let dx = p1.x -. p2.x in
let dy = p1.y -. p2.y in
sqrt (dx *. dx + dy *. dy)

(* dist : point -> point -> float *)

let p1 = { x = -1.0; y = 5.5 }
let p2 = { x = 10.0; y = 3.4 }
```

Produit nommé (2)

La syntaxe pour définir un produit nommé est :

```
type nom_type = {
lab1 : type1 ;
  …
labn : typen ; (* le dernier ; est optionnel *)
}
```

Les $\texttt{lab}_i$ sont des étiquettes ou des champs. Ils permettent d'accéder aux composantes du produit nommé. Les expressions s'écrivent :

```
{
lab1 = e1 ;
  …
labn = en ; (* le dernier ; est optionnel *)
}
```

On peut ensuite accéder à un champs par : $\texttt{e.lab}_i$.

Produit nommé (3)

Les produits nommés sont un peu moins souple que les n-uplets :

- On doit d'abord définir le type avant de l'utiliser
- Deux types différents ne peuvent pas partager le même nom de champ.

```
type personne = { nom : string; prenom : string }
type pays = { nom : string; lat : float; long : float }

let get_nom p = p.nom
(* de type:
 pays -> string
 l'étiquette nom du types pays « masque » celle du type personne.
*)
```

Pour les TPs, on utilisera toujours des noms d'étiquettes uniques.

Type sommes

Il est courant en informatique d'avoir un type composé de plusieurs « cas » différents. Pour gérer cela, OCaml propose des types « sommes » (aussi appelés types algébriques ou variants). On va prendre l'exemple d'un jeu de cartes que l'on souhaite modéliser :

```
type couleur = Coeur | Pique | Trefle | Carreau
```

Le code ci-dessus permet de définir quatres constantes, du type couleur. C'est une approche plus robuste que d'utiliser des entiers ou des chaînes de caractères. On veut aussi pouvoir définir la valeur d'une carte :

```
type valeur = Roi | Dame | Valet | Valeur of int
```

Ici, on indique que la valeur d'une carte peut être soit l'une des trois constantes `Roi`, `Dame`, `Valet` soit une un entier « décoré » par l'étiquette `Valeur`. On peut enfin définir une carte, par exemple en utilisant un produit nommé :

```
type carte = { valeur : valeur ; couleur : couleur }
let roi_pique = { valeur = Roi; couleur = Pique }
let sept_coeur = { valeur =  Valeur (7); couleur = Coeur  }
```

Type sommes (2)

On peut manipuler les types sommes comme n'importe quelle valeur OCaml :

```
let est_tete c =
  c.valeur = Roi || c.valeur = Dame || c.valeur = Valet

(* est_tete : carte -> bool *)
```

Cette utilisation est cependant inélégante. De plus, on ne peut pas utiliser les « entiers étiquetés » directement :

```
let as_trefle = { valeur = Valeur 1; couleur = Trefle}
let () = Printf.printf "valeur : %d\n" as_trefle.valeur
(*
Error: This expression has type valeur but an expression was
       expected of type int
*)
```

Plan

Filtrage ?

Les types produits (n-uplets ou produits nommés) possèdent une notation simple pour accéder à leur composantes (`let x, y = …`, `e.x`, ...).
Comment manipuler des valeurs d'un type somme ?

```
let string_of_valeur v =
     match v with
     Roi -> "Roi"
     | Dame -> "Dame"
     | Valet -> "Valet"
     | Valeur (1) -> "As"
     | Valeur (n) -> string_of_int n

(* string_of_valeur : valeur -> string *)
```

Dans un type sommes, il faut tester tous les cas possibles (étant donné une `valeur` de carte, on ne sait pas a priori dans quel cas on est). Cette construction est similaire au `switch` de C/C++/Java mais est plus sophistiquée.

Construction `match with`

Un type somme peut être inspecté en OCaml au moyen de la construction `match … with`. Cette dernière a la syntaxe suivante :

```
match e with
  p1 -> e1
| p2 -> e2
  …
| pn -> en
```

`e` est l'expression à analyser. Chaque `pi -> ei` est appelé une branche. Les `pi` sont appelés des motifs (*pattern* en anglais). Les `ei` sont des expressions.

En première approximation, un motif est soit une constante d'un type somme, soit une valeur étiquetée. Dans ce cas on peut capturer des sous-valeurs au moyen de variables (comme le `n` dans le transparent précédent).

Construction `match with` (2)


```
match e with
 p1 -> e1
| p2 -> e2
 …
| pn -> en
```

Dans l'expression ci-dessus, `e` est évalué pour donner une valeur `v`. Puis, `v` est comparée tour à tours aux motifs :

- si `v` est compatible avec p_1, alors e_1 est évalué
- sinon si `v` est compatible avec p_2, alors e_2 est évalué
- sinon si `v` est compatible avec p_3, alors e_3 est évalué
- ...
- sinon `v` est compatible avec p_n e_n est évalué

Exhaustivité du filtrage

Le compilateur OCaml détecte si un filtrage est incomplet ou au contraire redondant. Dans les deux cas un « warning » est émis :

```
# let dix = Valeur (10);;
val dix : valeur = Valeur 10
# match dix with
  Roi -> "Roi"
  | Valeur (n) -> string_of_int n;;
Warning 8: this pattern-matching is not exhaustive.
  Here is an example of a case that is not matched:
  (Dame|Valet)
# match dix with
  Roi -> "Roi"
  | Dame -> "Dame"
  | Valet -> "Valet"
  | Valeur (n) -> string_of_int n
  | Valeur (1) -> "As";;
Warning 11: this match case is unused.
```

On fera en sorte de toujours avoir du code sans *warning*.

Exemples complets

On illustre l'utilisation du filtrage avec d'autres fonctions

```
let string_of_couleur c =
       match c with
       Pique -> "Pique"
       | Coeur -> "Coeur"
       | Trefle -> "Trefle"
       | Carreau -> "Carreau"

(* string_of_couleur : couleur -> string *)

let string_of_carte c =
  (string_of_valeur c.valeur) ^ " de " ^ (string_of_couleur c.couleur)

let as_pique = { valeur = Valeur (1); couleur = Pique}

let s = string_of_carte as_pique
(* s : string = "As de Pique"  *)
```

Exemples complets (2)

```
(* renvoie -1 si c1 < c2, 0 si c1 = c2 et 1 si c1 > c2 *)
let compare_cartes c1 c2 =
  match c1.valeur, c2.valeur with
  | Roi, Roi | Dame, Dame | Valet, Valet -> 0
  | Roi, _ -> 1
  | _  , Roi -> -1
  | Dame, _ -> 1
  | _, Dame -> -1
  | Valet, _ -> 1
  | _, Valet -> -1
  | Valeur v1, Valeur v2 ->
      if v1 < v2 then -1
      else if v1 = v2 then 0
      else 1

(* compare_cartes : carte -> carte -> int  *)
(* Pas de warning, on sait qu'on a traité tous les cas *)
```

Remarque : _ est une variable spéciale qui signifie «n'importe quelle valeur».

On peut factoriser les motifs qui ont la même expression: p_1 | p_2 | p_3 -> e