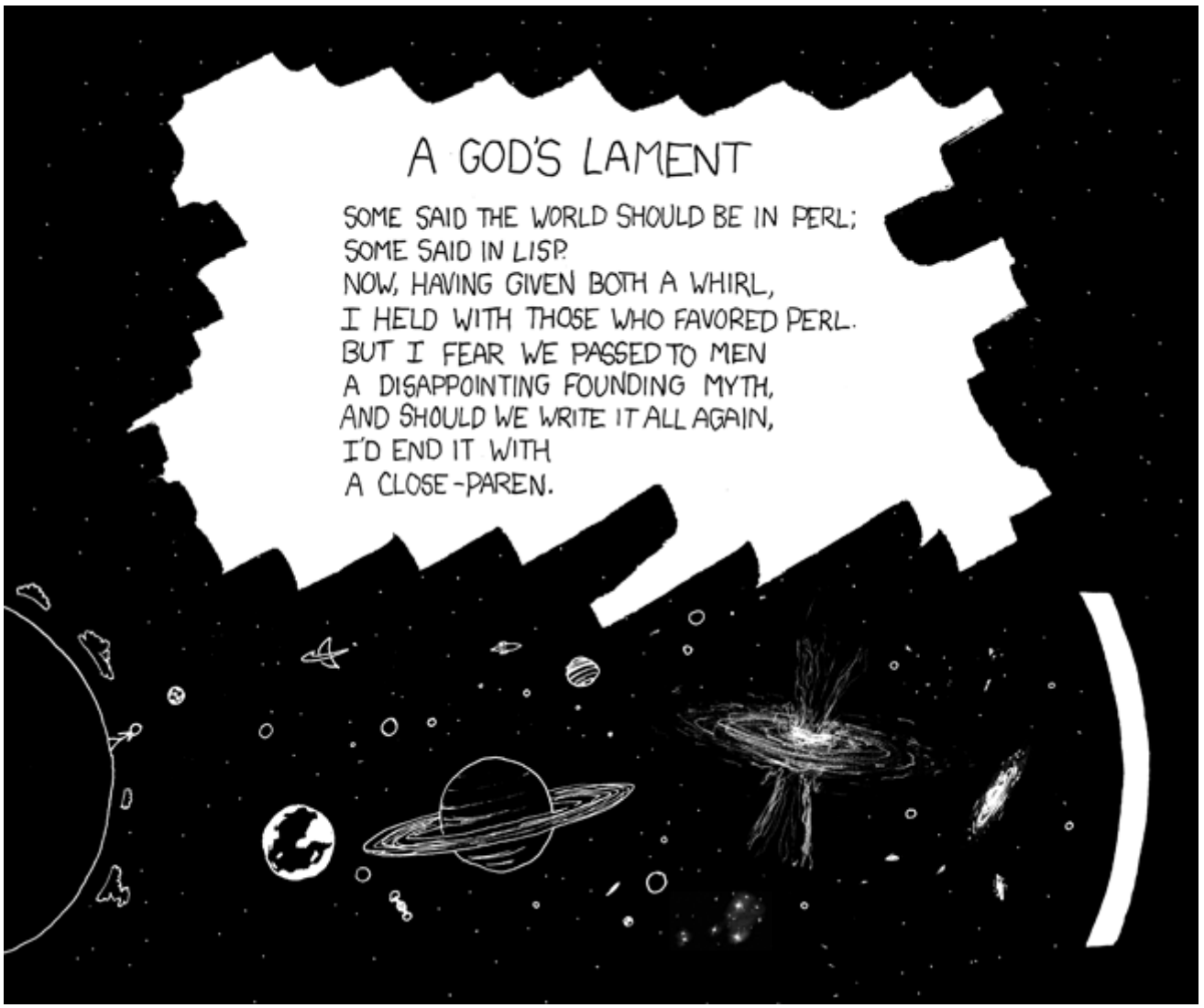
Programmation Fonctionnelle

Manuel Clergue

Programmer doit être amusant !
Programmer doit être beau !



Title text: Some say the world will end in fire; some say in segfaults.
XKCD 312

Chapitre 1 : Introduction à la programmation fonctionnelle

1 - Modèles de calcul

a - Informatique, Algorithmes et Problèmes

L'informatique *est la science et la technique de la représentation de l'information d'origine artificielle ou naturelle, ainsi que des processus algorithmiques de collecte, stockage, analyse, transformation, communication et exploitation de cette information, exprimés dans des langages formels ou des langues naturelles et effectués par des machines ou des êtres humains, seuls ou collectivement.*
Le Conseil scientifique de la Société informatique de France-2014

Un algorithme est la formalisation d'une suite finie d'opérations permettant de résoudre un problème par
décomposition du traitement à effectuer en actions, ou étapes, élémentaires, en précisant l'ordre de leur exécution.

Un problème (algorithmique) pose une question, ou un ensemble de questions, auxquelles il doit être répondu, étant donné un objet initial et un ensemble de spécifications sur la réponse ou le résultat.

Une longue lignée scientifique : Euclide (~ -300), Archimède (-287 -- -212), Brahmagupta (598–670), Al Khuwārizmī (~780 -- ~850), Fibonacci (~1175 -- ~1250), Hilbert, Turing, Church, Gödel, Von Neumann, Knuth, ...

b - Pourquoi un modèle de calcul ?

Tout n'est pas calculable ! Il existe des fonctions (ou des nombres) qui bien que parfaitement définis, ne sont pas calculables.

Un modèle de calcul est le moyen d'exprimer un algorithme (procédure mécanique !) dans un cadre formel (mathématique, théorique et abstrait).

Cela permet de donner une définition précise du concept d'algorithme (Art d'organiser un calcul complexe en partant d'opérations simples) :

- Décrire : un langage commun / structures algorithmique (contrôle/données)
- Exprimer leurs qualités : correction, complétude et terminaison
- Comparer : complexité des algorithmes

c - Machine de Turing

C'est un dispositif (abstrait, bien que …) conçu par Alan Turing (1912-1954)

- Un ruban infini divisé en case portant des symboles dans un alphabet fini
- Une tête de lecture/écriture lisant /écrivant les symboles sur le ruban et pouvant se déplacer sur

la droite ou la gauche de celui-ci
- Un registre d'états (nombre fini d'états, au moins 1 : l'état initial)
- Une table d'actions : qui indique quoi faire (écrire, aller à gauche ou à droite) et quel est le nouvel état, en fonction de l'état courant et du symbole sous la tête

Turing prouve l'existence de machine de turing universelle : une machine de turing qui "reçoit" le code d'une autre machine de turing (T) et de données (D) et qui est capable de simuler le fonctionnement de T (c'est à dire qu'elle produit le même résultat que T appliquée aux données D). [cela est quand même proche d'un ordinateur !]

Turing prouve que tout ce qui est calculable peut être calculé par une machine de turing universelle. Cela induit la notion de turing-complétude.

c - Lambda-calcul

Le lambda-calcul (ou λ-calcul) est un système formel inventé par Alonzo Church (1903-1995) qui fonde les concepts de fonction et d'application. En lambda-calcul tout est fonction, et le calcul se fait par réécriture symbolique (substitution/réduction).

d - Thèse de Church-Turing

Church–Turing thesis states that if some method (algorithm) exists to carry out a calculation, then the same calculation can also be carried out by a Turing machine (as well as by a recursively definable function, and by a λ-function).

2 - Paradigme (principes) de programmation

Programmation impérative : rester près de la machine, en particulier pour l'efficacité, mais la rendre plus humaine -> FORTRAN, COBOL, BASIC, PL/1, Pascal, C, ...

Programmation fonctionnelle : privilégier la justesse, en s'appuyant sur le socle mathématique solide des fonctions, de la logique et du lambda-calcul -> LISP (1958 !), ALGOL, SCHEME, ML, Caml, Haskell, Coq, ... et Clojure !

Programmation objet : privilégier l'architecture mentale en groupant données et fonctions (méthodes) dans des classes -> Simula, Smalltalk, C++, Objective C, Java, ...

Programmation logique / par contraintes : spécifier ce qu'il faut calculer, mais pas comment le calculer -> Prolog, ...

+ langages de script : sh, bash, Perl, ..., langages spécialisés (DSL = Domain-Specific Languages) : Excel, make, LaTex, HTML, lex/yacc, ...

En général les langages mélangent plusieurs principes comme Scala et Python (impératif + fonctionnel + objet)

3 - Programmation fonctionnelle

a - Introduction

Programmer par la définition de fonctions. L'exécution d'un programme se résume à appliquer des fonctions à leurs paramètres, et à en renvoyer leur(s) résultat(s). Il s'agit d'évaluer des lambda-expressions.

Exemple 1 : Calculer 5*2 + 1

Modèle impératif:
Définition

```
int a,b;       // des registres pour porter les résultats des calculs intermédiaires
a = 5 * 2;     // première étape du calcul
b = a + 1;     // deuxième étape du calcul
return b;      // opération spécifique de renvoi du résultat
               //(si on ne fait pas cela, ça de fait "rien"
```

Calcul
1 création des registres : a=?,b=?
2 première étape : a=10, b=?
3 deuxième étape : a=10, b = 11
4 la valeur 11 est "renvoyée"

Modèle fonctionnel:
Définition

```
add(1,mul(5,2)) ;; on suppose que deux fonctions sont définies add et mul)
```

ou en notation Lisp :

```
(add 1 (mul 5 2))
```

Calcul : évaluation récursive (les parenthèses intérieures avant)

```
(add 1 (mul 5 2)) ;; une règle de réécriture pour (mul 5 2)
(add 1 10) ;; une règle de réécriture pour (add 1 10)
11 ;; le résultat est l'expression finale
```

Exemple 2 : factorielle de 3

```
int n,f;           // a
n= 3;              // b
f=1;               // c
while(n>0){
  f = n * f;      // d
  n = n - 1;      // e
}
```

```
return f;
```

calcul :

```
a : n = ?, f = ?
b : n = 3 , f = ?
c : n = 3, f = 1
d : n = 3, f= 3
e : n = 2, f= 3
d : n = 2, f= 6
e : n = 1, f= 6
d : n = 1, f= 6
e : n = 0, f= 6
-> renvoie 6
```

En fonctionnel:

```
(f 0) -> 1                    ;;  définition
(f n) -> (* n (f (- n 1))     ;; définition (récursivité)
(f 3)
```

Calcul :

```
(f 3)
(* 3 (f 2))
(* 3 (* 2 (f 1)))
(* 3 (* 2 (* 1 (f 0))))
(*3 (* 2 (* 1 1)))
(*3 (* 2 1))
(*3 2)
6
```

Tout est fonction !! pas de structure de boucle : remplacée par la récursivité.
Cela ne veut pas dire qu'on ne peut pas faire de calcul itératif !!

Intérêt principal : plus formel, plus facile à prouver, pas (peu) d'effets de bord, rigueur, syntaxe simplissime.
Principal défaut : trop formel, lent, pas (peu) d'effet de bord, rigueur, complexité des mécanismes.

b - Principes généraux

Fonctions pures

Une fonction pure est une fonction dont la valeur de retour ne dépend que de ses paramètres et dont les opérations ne déclenchent aucun effet de bord (pas de manipulation de la mémoire, pas d'entrées/sorties). Une fonction pure est déterministe : un même jeu de paramètres produira toujours le même résultat.
Evidemment toutes les fonctions ne peuvent pas être pures ... sauf dans un monde purement abstrait et mathématique. Il est nécessaire de gérer les interactions avec l'environnement du programme (ex : entrées/sorties) ou avoir une simulation du hasard (ex : générateurs de nombres pseudo-aléatoires, dont le déterminisme est par essence masqué par une périodicité très importante). Pour cela, certains langages (lisp, clojure, …) autorisent l'utilisation de fonctions impures au côté de fonctions pures. D'autres (haskell) imposent que les fonctions impures soient circonscrites par des mécanismes spéciaux (comme les monades en haskell).

Immutabilité
La philosophie de la programmation fonctionnelle suppose que les objets manipulés ne sont pas modifiables (ils sont immutables). Certains langages tolèrent la mutabilité pour des raisons d'efficacité. Dans d'autres langages (haskell toujours, et clojure) interdisent purement et simplement la modification des objets.
En pratique, l'immutabilité suppose que les arguments d'une fonction ne sont pas modifiés par la fonction (ce qui pourrait être considéré comme un effet de bord).

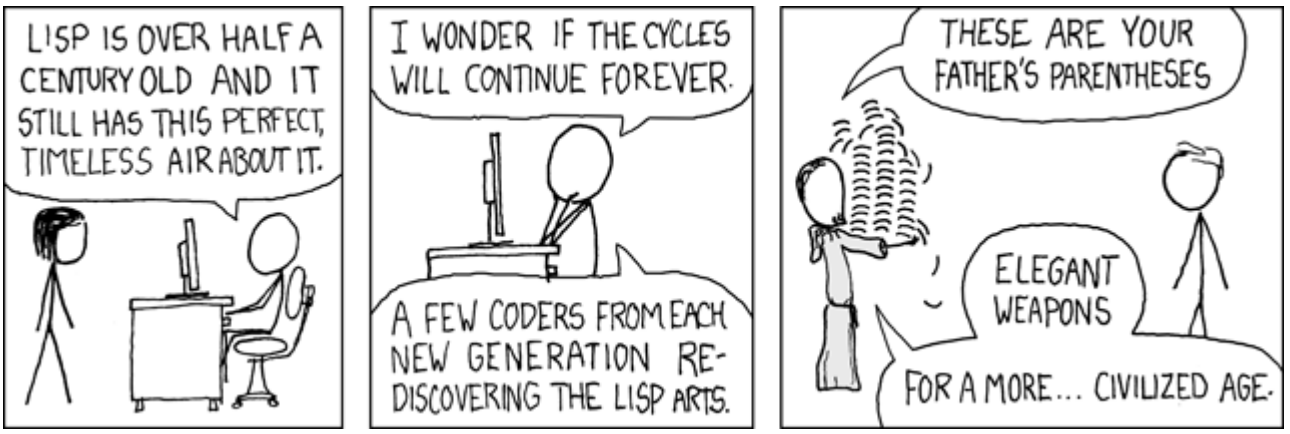
Fonctions de première classe
La programmation fonctionnelle est basée sur le fait que les fonctions doivent être considérées comme des objets manipulés par le programme, au même titre que les nombres, par exemple. Ainsi, elles peuvent être associées à des variables, ou être paramètres ou valeurs de retour d'une fonction.
Les fonctions sont des données comme les autres.

Fonctions d'ordre supérieur
Un fonction d'ordre supérieur est une fonction qui prend en paramètre ou qui renvoie une fonction. Elle n'est pas seulement paramétrée par des valeurs comme des entiers, mais également par des fonctions. C'est une manière élégante d'écrire des algorithmes génériques, des classes d'algorithmes.
Une fonction d'ordre supérieur peut également être un générateur paramétré de fonctions.

c - Typage

Les langages de programmation fonctionnelle peuvent avoir des philosophies différentes en matière de typage. Soit un typage fort (c'est le cas de haskell, qui impose des contraintes de types très fortes) soit un typage faible (les types des variables et en particulier des paramètres des fonctions ne sont pas spécifiés). Dans les deux cas, un mécanisme d'inférence de type permet de déterminer les types en fonction des opérations qui sont réalisées sur des variables (ex : `x+y` suppose que x et y supportent une opération +).

4 - Premier pas avec clojure



XKCD 297 : *I've just received word that the Emperor has dissolved the MIT computer science program permanently.*

Clojure est un dialecte lisp qui vise à en gommer les défauts. C'est un langage de programmation fonctionnelle qui compile vers du ByteCode pour JVM, qui impose l'immutabilité et les fonctions de première classe, qui n'impose pas un typage statique fort, et qui autorise le polymorphisme.

C'est un langage dynamique :
Cela signifie que le langage définit un environnement dont tous les éléments sont réifiés : accessibles, et modifiables.

les points critiques :

- les parenthèses …. bcp de parenthèses …. cependant, c'est la seule "syntaxe" (plus qques formes spéciales … au moins lambda)
- la notation polonaise inversée :-(

les points agréables :

- la gestion des collections (listes, vecteurs, ensembles et tables associatives)
- implémentation aisée de certains algos (surtout ceux donnés sous forme mathématique)

Le cycle d'évaluation la boucle REPL : read-eval-print-loop
Un programme clojure peut s'exécuter dans un programme java, ou comme un script standalone, mais l'interface primaire d'exécution est la boucle REPL.
La compilation se fait à la volée.
L'unité de base d'un programme clojure est une **expression clojure**. En clojure, tout est une expression qui s'évalue, et qui renvoie une valeur.

expression : `(f x1 …. xn)`
chaque terme est évalué (ordre non défini): évaluation récursive !
puis on applique f à x1 …. xn -> une valeur de retour !
ex

```
(+ 1 2) -> 3
(+ (- 4 3) 5) -> (+ -1 5) -> 4
(+ 1 2 3) -> 6
```

[spoiler alert] L'expression (+ 1 2) est structurellement une liste dont le premier élément est un symbole et les deux suivants des nombres. Sa sémantique est une sémantique d'invocation : le premier élément correspond à une fonction et les suivants les paramètres sur lesquels est appliquée la fonction.

a - Eléments de base du langage

Les types numérique :

```
42        ; entier
-1.5      ; nombres réels
22/7      ; fraction
```

Les entiers sont en précision fixée sur 64 bits, tant qu'ils restent dans l'intervalle de définition [2^-63;2^63 - 1]. Au délà, ils passent automatiquement en précision arbitraires.
Les nombres réels sont également en précision sur 64 bits, mais on peut forcer la précision arbitraire en les suffixant par M.
Les fractions sont un type à part entière :

```
(+ 1/3 1/4) ----> 7/12
```

Les types caractères :

```
"hello"          ; chaîne de caractères
\e               ; caractère
#"[0-9]+"        ; expression régulière
```

Il existe certains caractères spéciaux comme `\newline` ou `\tab`.
Les expressions régulières sont directement compilé dans un objet java :
`java.util.regex.Pattern`

Symboles :
A peu près toutes séquences de lettres, chiffres, ou signes de ponctuation (délimitées par des espaces ou les parenthèses).
Il existe 3 symboles spéciaux : **`nil`** (la valeur nulle, ou absence de valeur), et les booléens **`true`** et **`false`**.

Collections :

```
'(1 2 3)      ; liste
[1 2 3]       ; vecteur
#{1 2 3}      ; ensemble
{:a 1, :b 2} ; table associative (dictionnaire)
```

`Affichage :`

```
(println “le nombre deux” 2)
un texte 2
nil

(print “le nombre” 2)
un texte 2nil

(prn "un texte" 2)
"un texte" 2
nil

(pr "un texte" 2)
"un texte" 2nil
```

b - formes spéciales

Définition de variables :

```
(def x 1)
#'user/x
```

Le #’ est la marque des variables, user est l’espace de nommage par défaut et x est le symbole associé à la variable. Une fois définie la variable, elle peut être utilisée :

```
(+ 1 x)
2
```

Notez que l'utilisation d’une variable non définie génèrera une erreur :

```
(+ x y)
Syntax error compiling at (REPL:1:1).
Unable to resolve symbol: y in this context
```

Expression conditionnelle simple :

```
(if (even? 1) "pair" "impair")
"impair"

(if (even? 2) "pair" "impair")
"pair"
```

Notez au passage l'utilisation de la fonction `even?` (le ? fait partie du nom de la fonction, il est mis par convention pour signaler que la fonction renvoie un booléen).
Seule la branche utile est évaluée, et la valeur de l'expression (if … est la valeur renvoyée par l'évaluation de cette branche. L'expression doit absolument renvoyer une valeur, si la branche correspondante est absente, on renvoie la valeur nil :

```
(if (even? 1) "pair")
nil
```

Définition de fonction :

Pour définir une fonction, on utilisera la forme spéciale defn. Par exemple :

```
(defn bonjour [] "Bonjour")
#'user/bonjour
```

définit une fonction à 0 paramètre. Que l'on peut appeler comme ceci :

```
(bonjour)
"Bonjour"
```

Notez la différence entre l'utilisation d'une fonction d'arité 0 et une variable … tout est dans l'invocation !

```
bonjour
#object[user$bonjour 0xd103887 "user$bonjour@d103887"]
```

Le compilateur vérifie que l'appel de la fonction respecte le contrat (essentiellement l'arité) de la fonction:

```
(bonjour 1)
Execution error (ArityException) at user/eval146 (REPL:1).
Wrong number of args (1) passed to: user/bonjour
```

Pour définir des fonctions avec un ou plusieurs paramètres :

```
(defn bienvenue [nom] (str "Bonjour " nom " !"))
#'user/bienvenue
(bienvenue "Manuel")
"Bonjour Manuel !"

(defn addition [x y] (+ x y))
#'user/addition
(addition 3 4)
7
```