PFON

Notions

Paradigme :

- Affecte la manière de penser
- Affecte l'expressivité
- Manière de s'exprimer

Une fonction est dite à **effet de bord** si elle modifie un état en dehors de son environnement local.
Typiquement, les fonctions qui:

- Modifient une variable locale statique
- Modifient une variable globale (*non locale*)
- Modifient une variable passée en argument par référence
- Appellent des fonctions à *effet de bord*

Langage fonctionnel **pur** :

- Calcul de la valeur de sortie (*retour*) en fonction des valeurs d'entrée (*arguments*)
- Que des constantes
- Pas d'effet de bord
- Pour une entrée, toujours la même sortie

Types d'expressions:

- Expressions **littérales** : s'évaluent à elles-mêmes
- Epressions **combinées** : application de procédures (*opérateurs, fonctions*) à des arguments (*expressions*)
- Expressions **abstraites** : nommage et assignation d'expressions

Abstraction **syntaxique** ⟶ déclarations ⟶ Haskell
Abstraction **fonctionnelle** ⟶ expressions ⟶ Lisp

Variadicité (fonctions à nb d'args variables) en Lisp (*utilisation de &*). Exemple :

```
(defun mklist (head &rest tail)
    (cons head tail))
;; (mklist 'a 'b 'c)

(defun msg (str &optional (prefix "error: ") postfix)
    (concatenate 'string prefix str postfix))
;; (msg "hello" nil "!")
```

Typage **dynamique** :

- **Valeurs** sont typées
- Vérification de type à l'**éxecution**
- ⟶ Lisp

Typage **statique** :

- **Variables** sont typées
- Vérification de type à la **compilation**
- ⟶ Haskell

Typage **faible**: Autorise l'affectation de variable avec des valeurs ne correspondant pas à son type déclaré ⟶ erreur de type difficilement détectable

Typage **fort**: Interdit l'affectation de variable avec des valeurs ne correspondant pas à son type déclaré ⟶ erreur de type facilement détectable

Typage **implicite**: Pas obligé lors de la déclaration d'une variable de donner son type

Typage **explicite**: Obligé lors de la déclaration d'une variable de donner son type

Polymorphisme : définition unique ∀ type

```
length :: [a] -> Int -- qqsoit le type de [a], la définition de la fonction reste la même
```

Surcharge ou *overloading*: différentes définitions selon le type

Modèle de substitution: Remplacement des paramètres formels par les arguments correspondants

```
(defun exemple (a) (+ a a))
;; (exemple 2)
;; -> (+ 2 2)
```

Evaluation **stricte**

- Arguments (expressions) évalués d'abord
- Modèle de substitution
- Lisp :

```
(defun sq (x) (* x x))
(defun ssq (x y) (+ (sq x) (sq y)))
(defun f (a)
    (ssq (+ a 1) (* a 2)))

;; (f 5)
;; (ssq (+ a 1) (* a 2))
;; (ssq (+ 5 1) (* 5 2))
;; (ssq 6 (* 5 2))
;; (ssq 6 10)
;; (+ (sq x) (sq y))
;; (+ (sq 6) (sq 10))
;; (+ (* x x) (sq 10))
;; (+ (* 6 6) (sq 10))
;; (+ 36 (sq 10))
;; (+ 36 (* x x))
```

```
;; (+ 36 (* 10 10))
;; (+ 36 100)
;; 136
```

Lazy évaluation

- Expressions évaluées seulement quand le besoin s'en fait sentir
- Modèle de substitution
- Seulement dans un langage fonctionnel pur
- Haskell :

```
sq :: Float -> Float
sq x = x * x
ssq :: Float -> Float -> Float
ssq x y = sq x + sq y
f :: Float -> Float
f a = ssq (a + 1) (a * 2)

{-
f 5
ssq (a + 1) (a * 2)
ssq (5 + 1) (5 * 2)
sq x + sq y
sq (5 + 1) + sq (5 * 2)
(x * x) + (y * y)
(5 + 1) * (5 + 1) + (5 * 2) * (5 * 2)
6 * (5 + 1) + (5 * 2) * (5 * 2)
6 * 6 + (5 * 2) * (5 * 2)
36 + (5 * 2) * (5 * 2)
36 + 10 * (5 * 2)
36 + 10 * 10
36 + 100
136

-}
```

Ordre applicatif/normal : sémantique des langages
Strict : se dit surtout d'une procédure / fonction
Paresseux : se dit surtout d'un évaluateur
Dans un langage d'***ordre applicatif***, toutes les procédures sont strictes.
Dans un langage d'***ordre normal***, toutes les procédures non primitives sont non strictes (puisque l'évaluateur est paresseux), et les procédures primitives peuvent être strictes, ou pas.

Contextes/environnements locaux **implicites**:

- Args des fonctions (α-conversion, imbrication)

Contextes/environnements locaux **explicites**:

- Données locales (évite la redondance d'éval)
- Fonctions locales (évite pollution des espaces de noms)

Variable **liée**: définie dans le contexte local (définition locale)
Variable **libre**: non définie localement

Scoping: capture d'une variable libre

1. Scoping **lexical**:
    - recherche dans l'environnement de *définition*
    - retour de fcts créées à la demande en toute sécurité

```
(let ((x 10))
    (defun foo () x))
(let ((x 20))
    (foo)) ;; => 10
```

1. Scoping **dynamique**:
    - recherche dans l'environnement d'*appel*
    - retour fonctionnel limité aux fcts constantes

```
(defparameter x 10)
(defun foo () x)
(let ((x 20))
    (foo)) ;; => 20
```

Fermetures lexicales: Combinaison entre une fonction et son environnement de définition (valeurs des variables libres au moment de la définition):

- Opérations génériques par fcts anonymes

```
(defun list+ (lst n)
    (mapcar (lambda (x) (+ x n))
            lst))
```

```
(+++) :: [Int] -> Int -> [Int]
(+++) lst n = map (\x -> x + n) lst
```

- Création dynamique de fcts à état local

```
(defun make-adder (n)
    (lambda (x) (+ x n)))
```

```
makeAdder :: Int -> Int -> Int
makeAdder n = \x -> x + n
```

- Encapsulation (portée restreinte)
- Etat local modifiable (**Lisp**)

```
(let ((cnt 0))
    (defun newtag () (incf cnt))
    (defun resettag () (setq cnt 0)))
```

Curryfication : passage d'une fct n-aire à une fct unaire
Décurryfication: inverse de *curryfication*

> Note: Fonctions Haskell sont unaires. ⟶ curryfication

A retenir des annales

Fonction ordre supérieur:

- Une ou n fcts en entrée
- Renvoie une fct (via création de fct anonyme)

Dans un langage fonctionnel pur:

- Que des constantes
- Pas d'effet de bord
- Pour une entrée, toujours la même sortie

Typage statique ⟶ type des variables connues à la compilation

Offside rule : Indentation comme syntaxe (comme en python)
En Haskell, il existe un séparateur implicite qui remplace l'offside rule lors du parsing ⟶ `;`

En Lisp, les valeurs booléennes sont représentées par: *nil* ou la liste vide (**false**) et tout sauf *nil* (**true**)

Valeur de l'expression suivante en Haskell:

```
[ x == 3 | x <- [2, 3, 4]]
-- which gives [False, True, False]
```

Tree-accumulation: Evaluation récursive de gauche à droite de toutes les sous-expressions d'une expression fonctionnelle

Opérateur spécial en Lisp: Fonction n'obéissant pas aux règles de l'évaluation stricte

Que signifie l'expression "*Lisp-2*" ? ⟶ Qu'il existe des espaces de noms distincts pour les variables et les fonctions

mapping: application d'une fonction à tous les éléments d'une liste

Fonction `complement` de Lisp: Prend une fct booléene et renvoie une fct produisant le résultat inverse