

# INTRODUCTION A LA PROGRAMMAT<sup>°</sup> FONCTIONNELLE

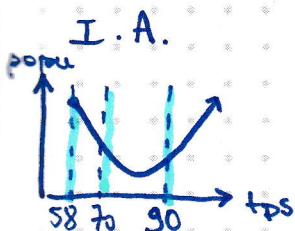
la pgmt fonctionnelle est un paradigme parce → s'utilise très rarement seul.

- I. Paradigme
- II. Tlse en rel<sup>t</sup>e avec les autres paradigmes
- III. lisp & Haskell
- IV. Evaluation
- V. Fonctions = Objet

## Pourquoi Haskell et lisp?

2 lgg car paradigme  $\neq^+$  selon lgg et paradigme =  $\varphi^0$   
On a déjà vu OCaml + pas typique du fct<sup>n</sup>  
2 lgg opposés : Haskell • syntaxe riche | lisp • formalisme lambda  
• formalisme | calcul  
• mathématique | • syntaxe minimaliste  
• très strict | • plus expressif

lisp : père de tous les lgg fortement typés  
1958 → Naissance de la 1<sup>re</sup> dérivat<sup>o</sup> du lisp  
pr intell<sup>igence</sup> artificielle



Courbe de popularité  
de l'IA et du lisp

lisp = famille de lgg : {scheme, common, emacs lisp, clojure, ...} Svt confusion entre lisp et common lisp

- Scheme : lisp pédagogique (SICP)
- Common lisp : Utilisé pdt le cours std en 85 par des industriels
- Clojure : JVM. Récent
- Emacs lisp : le pire, mélange C et lisp

## PARADIGME

"Quoi faire" plutôt que "Comment faire" → on se préoccupe du pb et pas de l'implém  
Très vieux, lgps ignorer, relient mtm

Paradigme → affecte expressivité et manière de penser  
s'en inspirer pour les utiliser dans tous les lgg

Basé sur 3 piliers:

- Expressions → On exprime des choses, on ne donne pas des ordres
- Définitions → On nomme les expressions
- Evaluations → def ou expr

En fait on réduit au maximum les effets de bord  
→ expression et pas instruct

Noyau réduit au minimum et on dup tout autour

ex: Somme des carrés des entiers entre 1 et N  
→ slide 5

Syntax Lisp:

S Expr  $(fn\ x\ y\ z)$   $x, y, z$  étant les args de fn

Atoms 1 "foo" ...

$(1 - x)$  décrémente x de 1  
Pas de return → fondamental

Lisp

```
(defun ssg (n)
  (if (= n 1)
      1
      (+ (* n n) (ssg (- n)))))
```

valeur de ssg(n) = val du if  
valeur du if =  
valeur de 1 si n=1  
valeur de l'expr sinon

ssq :: Int → Int

ssq 1 = 1

ssq n = n \* n + ssq(n-1)

Haskell

Def de ssg qui prend Int et ret Int  
val de ssg(1) = 1  
val de ssg(n) = expr

→ + clair et + concis que de l'impératif ou algo

ex: racine carrée de la somme des carrés de a et b  
→ slide 6

le fonctionnel c'est un peu de l'impératif à l'envers  
→ On s'occupe d'abord du haut niveau puis des détails

## 1<sup>o</sup> ORDRE

Fonction = Objet de 1<sup>o</sup> classe (1<sup>o</sup> ordre, ordre supérieur, etc)  
→ Christopher Strachey

- nommage (var)
- aggrégation (struct)
- arg de fct
- retour de fct
- manipulat anonyme (lambda)
- construct<sup>o</sup> dyn<sup>o</sup>
- ... → Plus d'expressivité (clarté et concision)

⇒ On peut faire avec des fonctions tout ce qu'on peut faire avec les objets built-in du lang

On donne des noms aux fct comme aux variables.  
De même on peut faire des affectations de fonction.

En lisp, un nom peut être à la fois pour une fct et une var en même temps

name 

var	fct
-----	-----

Ex: On peut écrire (foo foo)  
où foo est une fct et une var

On peut passer des fct en arg de fct

- mapping : traiter individuellement les élmt d'une liste par une fct
- folding : combiner les élmt d'une liste par une fct

ex: slide 10

Fonctions anonymes: on peut ne pas les nommer (lambda fct)  
On l'utilise comme une fct normale: ((lambda fct) arg)

ex: slide 11

On peut aussi retourner des fonctions comme n'importe quelle valeur de retour

On peut du coup generer des pgm avec des pgm. → M<sup>o</sup>ta pgm<sup>o</sup>

ex: slide 12

## PURETÉ

Pgm<sup>o</sup> fct pure

- pas d'effet de bord
- la fct a un sens mathématique  
→ Formalisme mathématique
- pas d'influence du contexte
- calcul d'une valeur de sortie en fct de valeurs d'entrée: une fct renverra tjs la m<sup>e</sup> chose si on lui passe tjs les m<sup>e</sup> argut
- une variable est une constante: on donne un nom à une valeur, pas l'inverse

Pureté = plus de sûreté

- parallélisme: pas besoin de verrou car pas de conflit d'accès
- sémantique locale aux fcts: débug local
- preuve de pgm

ex: slide 17

## EVALUATION

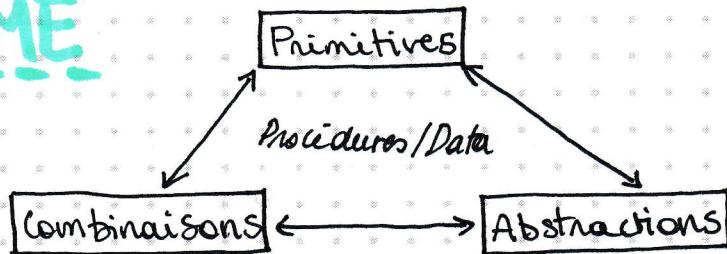
À quel moment évaluer une expr?  
→ stricte / paresseuse

- Stricte: On évalue les args puis on applique la fonction
- Paresseuse: les arguments sont évalués uniquement s'il y en a besoin.

Stricte : lisp      Paresseuse : Haskell

Avantage de l'évaluation paresseuse: + d'abstraction  
 Contrainte " " " : pureté fct<sup>L</sup> reprise

## RÉSUMÉ



les 3 caractéristiques des langages

- Moins de distinction entre procédure et donnée
- Plus de puissance dans la combinaison
- Plus de puissance dans l'abstraction

## EXPRESSION

Mise en forme :

- Lisp:
- Parenthèses
  - Aucun mot réservé
  - Modifiable par ReaderMacro

- Haskell:
- Indentation
  - Séparateur implicite ;
  - Quelques mots réservés

Nommage :

- Lisp:
- N'importe quoi ???
  - ` ` pour escape les noms

- Haskell:
- Identique au C + apostrophe
  - 1<sup>re</sup> lettre capitalisée pour les types

Opérateurs et fonctions

- Lisp:
- Pas de distinct°
  - (+ args...)
  - Exclusivem<sup>t</sup> préfixe
  - Opérateurs variadiques
  - Imbricat° naturelle

- Haskell:
- Distinct°
  - Pont entre les notations  
 $3+4 \Leftrightarrow (+) 3 4$   
 $\text{div } 3 4 \Leftrightarrow 3 \text{'dir'} 4$
  - Pb d'associativité et précédence
  - Définit° d'opérateur

Attribution

- Lisp:
- defvar \* var \* val
  - defun fct (args...)
  - setq var val/expr/var
  - abstract° fonctionnelle

- Haskell:
- def cst et fct :: type/prototype
  - = val
  - = calcul / appel de fct
  - = var
  - abstract° syntaxique

## Développement interactif :

Boucle d'évaluation REPL

- read : saisir une expr

- eval : calculer (évaluer) sa valeur

- print : présenter le résultat sous forme affichable

Lisp : • interprétat° Haskell : • REPL limité (expr vs decl)  
• byte [JIT]

# TYPAGE

Lisp : dynamique & fort Haskell : statique & fort

Pb orthogonal à la pgm<sup>n</sup> fct<sup>L</sup>

## typage dyn<sup>q</sup>:

- valeurs typées

- vérification de type à l'exec

## typage stat<sup>q</sup>:

- variables typées

- vérif<sup>o</sup> de type à la compil°

Var non typée : var = ptr vers un obj Lisp

Val typée : info contenue ds chaque objet

Info de typage accessible : typeof / typep

Lisp

Lisp peut aussi être typé en statique (mot clé à préciser)

la légende de la lenteur du typage dyn<sup>q</sup>

typage dyn<sup>q</sup> → vérif au val → lenteur

Sauf que :

- struct de données modernes

- decl explicite de type

- niveau d'opti des compilateurs (speed, safety, debug)

Lisp:

- liberal

- polymorphisme de facto

- ou explicite*

- compromis efficacité/sûreté

Haskell:

- contrainte (gma) nt

- polymorphisme rigide

- sûreté de facto

- sans compromis*

# EXPRESSIONS CONDITIONNELLES

Booléen : Haskell

- type

- True et False

- && || not

- let and, or :: [Bool] → Bool

- Lisp pas de type

- val t, tout sauf nil (vrai) et nil, () (faux)

- opérat<sup>o</sup>rs and or not

If then else : Haskell

| if (...) then ...

| else if (...)

| then ...

| else ...

Lisp | (if (...))

| (if (...))

| ...)

Gardes & cond : Haskell

...  
| (...) = ...  
| (...) = ...  
| otherwise = ...

lisp (cond ((...))

...  
((...))  
...  
(t ...))

Pattern matching : écriture équationnelle en Haskell (cf seq)  
→ préférable au if et aux gardes.

case : lisp

| (case ...  
| ((...)) ...)  
| ((...)) ...  
| (otherwise ...)

Possible sur objet types (typecase)

Transtypage : implicite en lisp

## LISTES

Type natif des ts des lang. for

- Haskell [ , , ]
- lisp ( , , )

homogènes

hétérogènes

[ ]

() ou nil

(:) :: a → [a] → [a]

cons / consp

lisp: [1] (cons 1 nil)

[1,2] 1:(2:[ ])

[1 NIL]



Haskell: 1:[ ]

→ listes emboîtées.

## EVALUATION p.2

Eval stricte → ordre applicatif

Déroulement : + tree accumulat - processus récursif

Feuilles = données et opé primutifs

$$(2 + 4 * 6) * (x + 5 + 7) \rightarrow (* (+ 2 (* 4 6)) (+ x 5 7))$$

