

PROGRAMMATION FONCTIONNELLE : LANGAGES, PARADIGME ET ÉCOSYSTÈME

EN QUELQUES MOTS

- Paradigme construit sur la notion de "fonction"
- Eviter les effets de bord et les mutations de variables
- Privilégier la récursivité aux structures itératives comme les boucles

POURQUOI ?

- Une approche polyvalente et compositionnelle
- Facilite le développement de programmes complexes
- Vers des programmes plus fiables et maintenables

```
(* Coq *)
Fixpoint map (A B : Type) (f : A -> B) l := 
  match l with
  | [] => []
  | a :: l => f a :: map A B f l
end.
```

IMPACT ET PERSPECTIVES

PROGRAMMATION PARALLÈLE

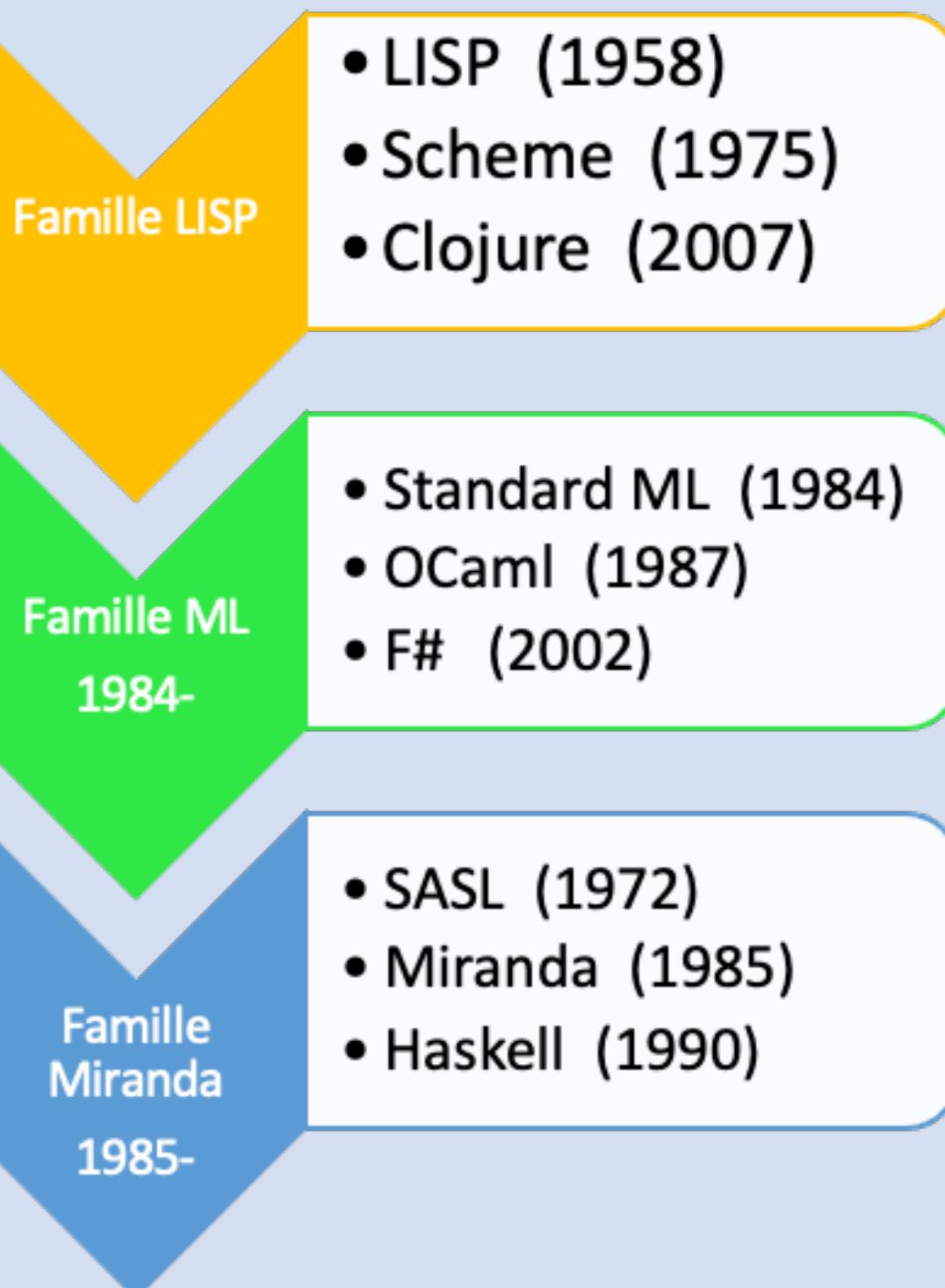
Le parallélisme dans un programme fonctionnel est implicite grâce à :

- La transparence référentielle
- L'indépendance de l'ordre d'exécution des fonctions en fonction du résultat final
- La détection et l'allocation automatiques aux processeurs du code parallélisable

GÉNIE LOGICIEL

- De plus en plus de langages multiparadigmes adoptent le fonctionnel
- La composition de fonctions permet de réduire la complexité globale d'un projet en le découplant en sous-problèmes moins complexes

FAMILLES DE LANGAGES FONCTIONNELS, CERTAINS MULTIPARADIMES



Au fil des années les langages fonctionnels ont commencé à sortir du milieu académique et de la recherche pour se tourner vers le milieu industriel avec des langages comme OCaml. Ce paradigme a commencé à s'intégrer de façon native dans la définition des langages impératifs et orientés objet.

LES PRINCIPAUX CONCEPTS DU PARADIGME FONCTIONNEL

Evaluer l'argument de la fonction avant d'évaluer l'appel de la fonction

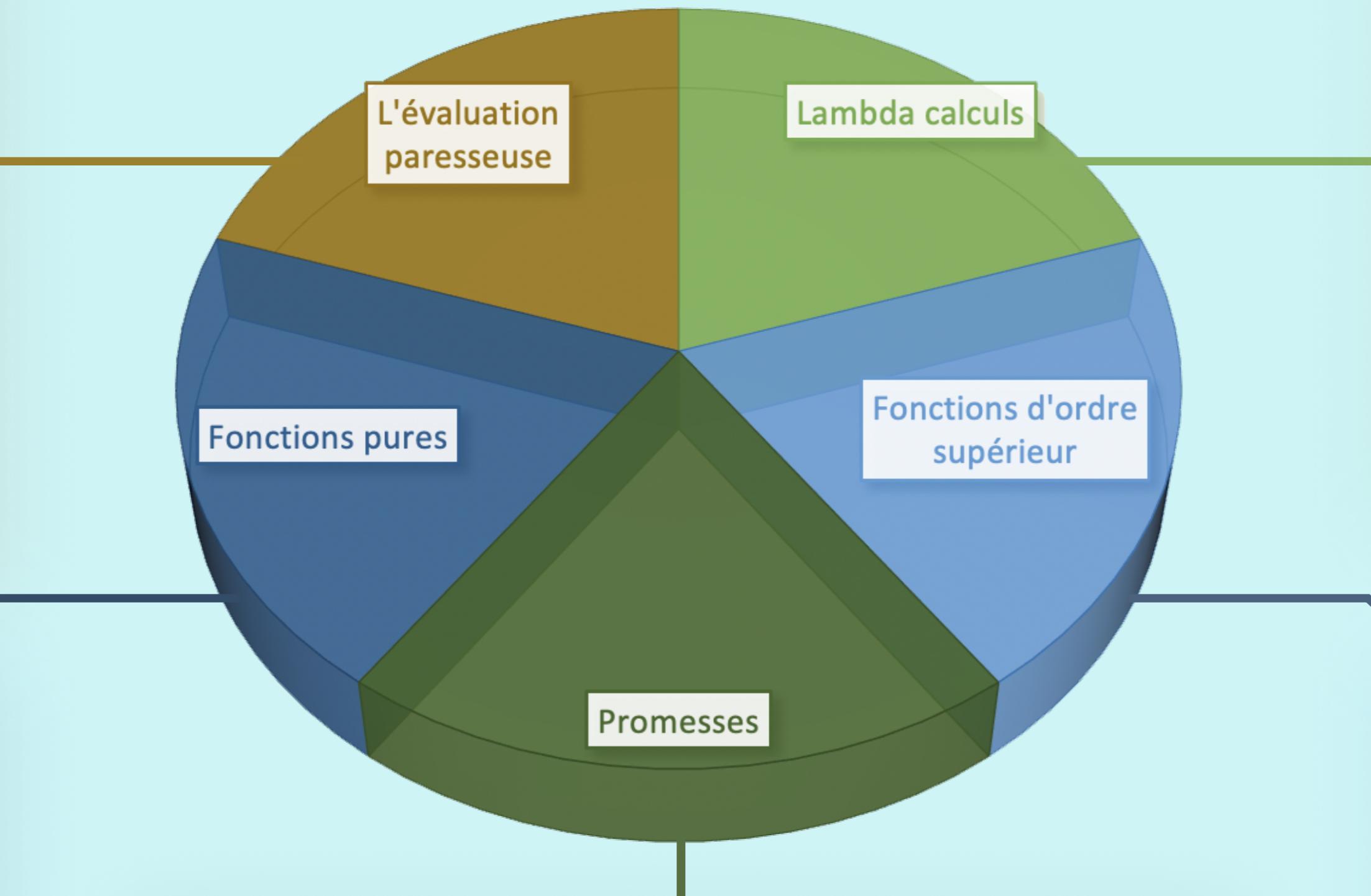
```
-- Haskell
(\f -> \x -> \y -> f (f x))
(\n -> n * n) 16
(Let loop n = loop n in loop 0)

→ 65536
```

Fonctions pures : Sans effet de bord observable

Transparence référentielle : La valeur de retour est la même pour les mêmes arguments

```
(* OCaml *)
let plus a b = a + b;;
```



Approche se servant du système de fermeture pour gérer des requêtes asynchrones, en retardant l'évaluation de certains traitements (callbacks)

```
//JavaScript
let promesse = new Promise( resolve => {
  setTimeout(()=>{resolve("response")},10000)
})
promesse.then(res => alert(res))
```

- Calcul effectif
- Calculabilité
- Récursivité

```
-- Haskell
(\x -> x + 1)
⇒ 3+1
⇒ 4
```

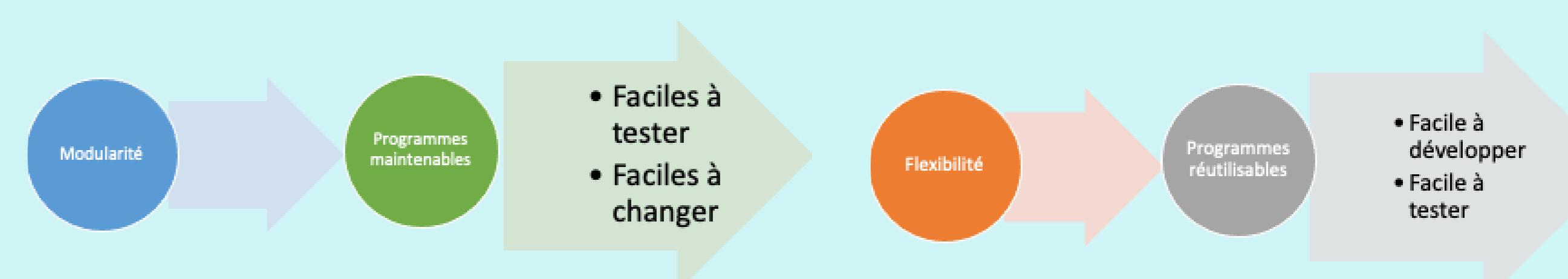
- Une fonction qui prend une ou plusieurs fonctions en arguments
- OU
- Une fonction qui renvoie une fonction

```
(* OCaml *)
let rec sigma f = function
| [] -> 0
| x :: l -> f x + sigma f l;;
```

CONCLUSIONS

Ce paradigme s'appuyant sur des fondements logico-mathématiques, cette approche théorique est quelque fois perçue comme difficile à appréhender

Néanmoins, ce paradigme a trouvé une place très importante dans l'industrie du génie logiciel grâce à sa modularité et sa flexibilité



Références

J.Hughes. *Why Functional Programming Matters*. The Computer Journal32.2 (jan. 1989)

Philip Wadler. *The essence of functional programming*. In: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '92

Eric Thivierge et Marc Feeley. *Efficient compilation of tail calls and continuations to JavaScript*. En. In: Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming - Scheme '12.

Henk Barendregt. *The impact of the lambda calculus in logic and computer science*. (1997)

Zhenjiang Hu, John Hughes et Meng Wang. *How functional programming mattered*. In: National Science Review2.3 (sept. 2015).

Alexander Sobolev et Sergey Zykow. *Functional Programming Patterns in JavaScript*. En. Intelligent Decision Technologies 2019.

Guillaume Claret. *Program in Coq*

Mark Priestley. *AI and the Origins of the Functional Programming Language Style*. En. Minds & Machines27.3 (sept. 2017)