Programmation fonctionnelle

Quoi? Pourquoi? Comment?

Jean-Michel Begon

Université de Liège

22 février 2017

http://www.montefiore.ulg.ac.be/~jmbegon/?pf2016_2017

Plan

- Programmation impérative
  - Programmation procédurale (ex. fortran, C)
  - Programmation orientée objet (ex. Java, C#)
- Programmation déclarative
  - Programmation fonctionnelle (ex. Lisp, **Scheme**)
  - Programmation logique (ex. Prolog)

> *Un paradigme de programmation est un style fondamental de programmation informatique qui traite de la manière dont les solutions aux problèmes doivent être formulées [...].*
>
> — *Wikipedia, 2015*

Bref : Une manière de penser et de concevoir la programmation et la résolution de problèmes.

Programmation	Alphabet
C $\leftrightarrow$ fortran	latin $\leftrightarrow$ grec
impératif $\leftrightarrow$ déclaratif	latin $\leftrightarrow$ chinois

Attention : tant que vous ne penserez pas dans le bon paradigme, les chances de réussites seront faibles.

- Limitation des effets de bord
  - Déterminisme
  - Simplicité $\leftrightarrow$ erreurs
- Lisibilité
- Nouvelles mécaniques
  - Encapsulation et modularité
  - Constructeurs
  - Retour d'objets complexes
  - *Garbage collection* et abstraction de la mémoire
  - ...

Les langages orientés objets (1960s) ont repris certaines de ces caractéristiques et les ont parfois même améliorées...

Avantages

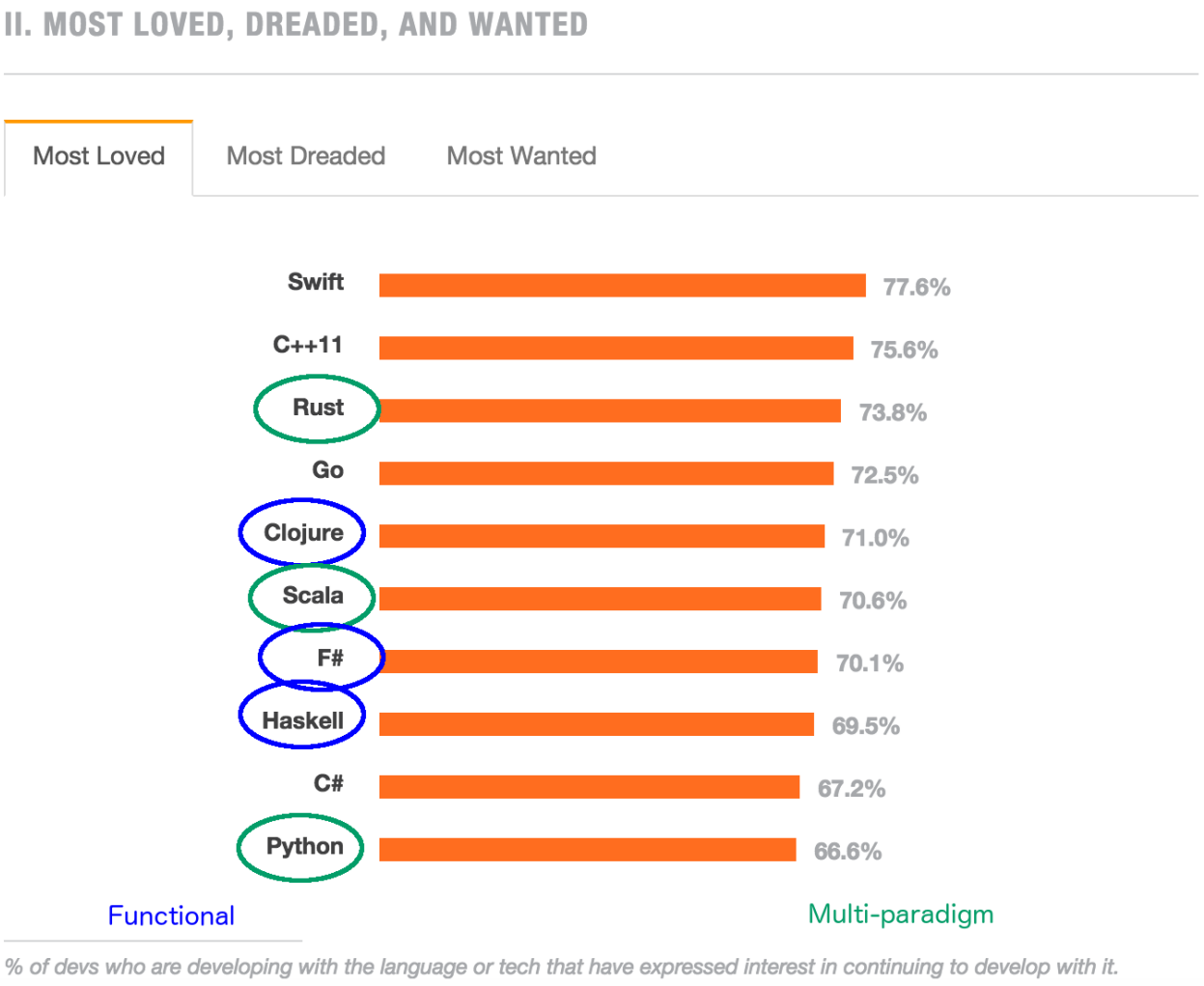
- Limitation des effets de bords.
- Parallélisation.
- Simplicité.
- Robustesse.

Inconvénients

- Performances
  - Pas de tableaux, tables de hachage, etc.
  - Coût de la récursion (pas toujours moins efficace).
- Pas d'I/O "natif"

Domaines d'intérêt : *High scientific/Distributed computing* (ex. : MapReduce), backend et réseaux (*cf.* Erlang), compilateurs et calcul symbolique, etc.

Certains problèmes et certaines structures de données se manipulent naturellement en programmation déclarative !



http://stackoverflow.com/research/developer-survey-2015#tech

Pourquoi la programmation fonctionnelle ?

- Apprendre et maîtriser un nouveau paradigme.
- Reconnaître les situations dans lesquelles l'approche fonctionnelle est plus avantageuse que l'approche impérative.
- Apprendre à écrire des spécifications correctes.
- (Maîtriser un nouveau langage)

Afin :

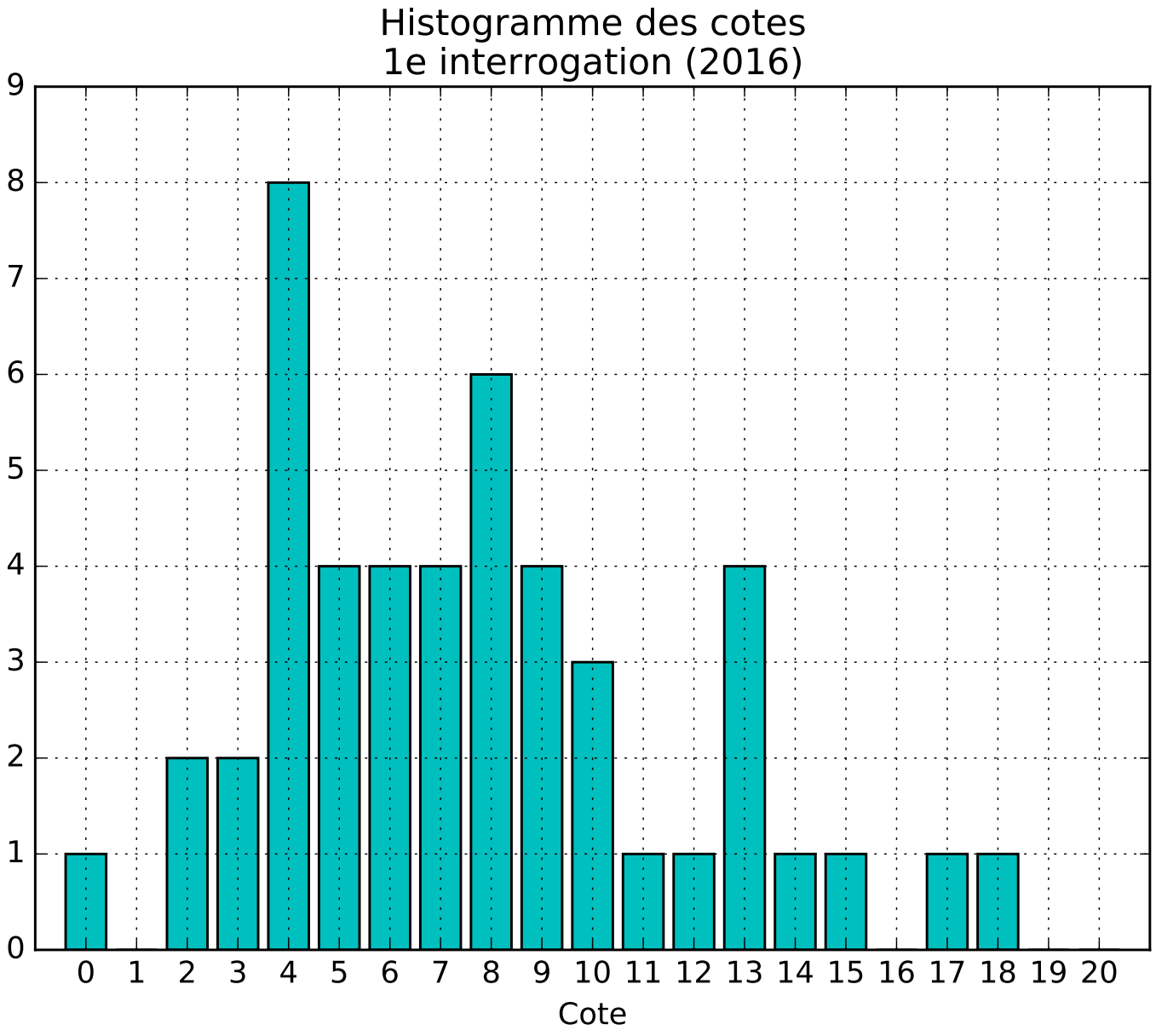
- D'être critique dans ses choix de méthode de résolution.
- D'être plus rapide, plus efficace.
- De s'ouvrir la porte de certains domaines spécifiques.

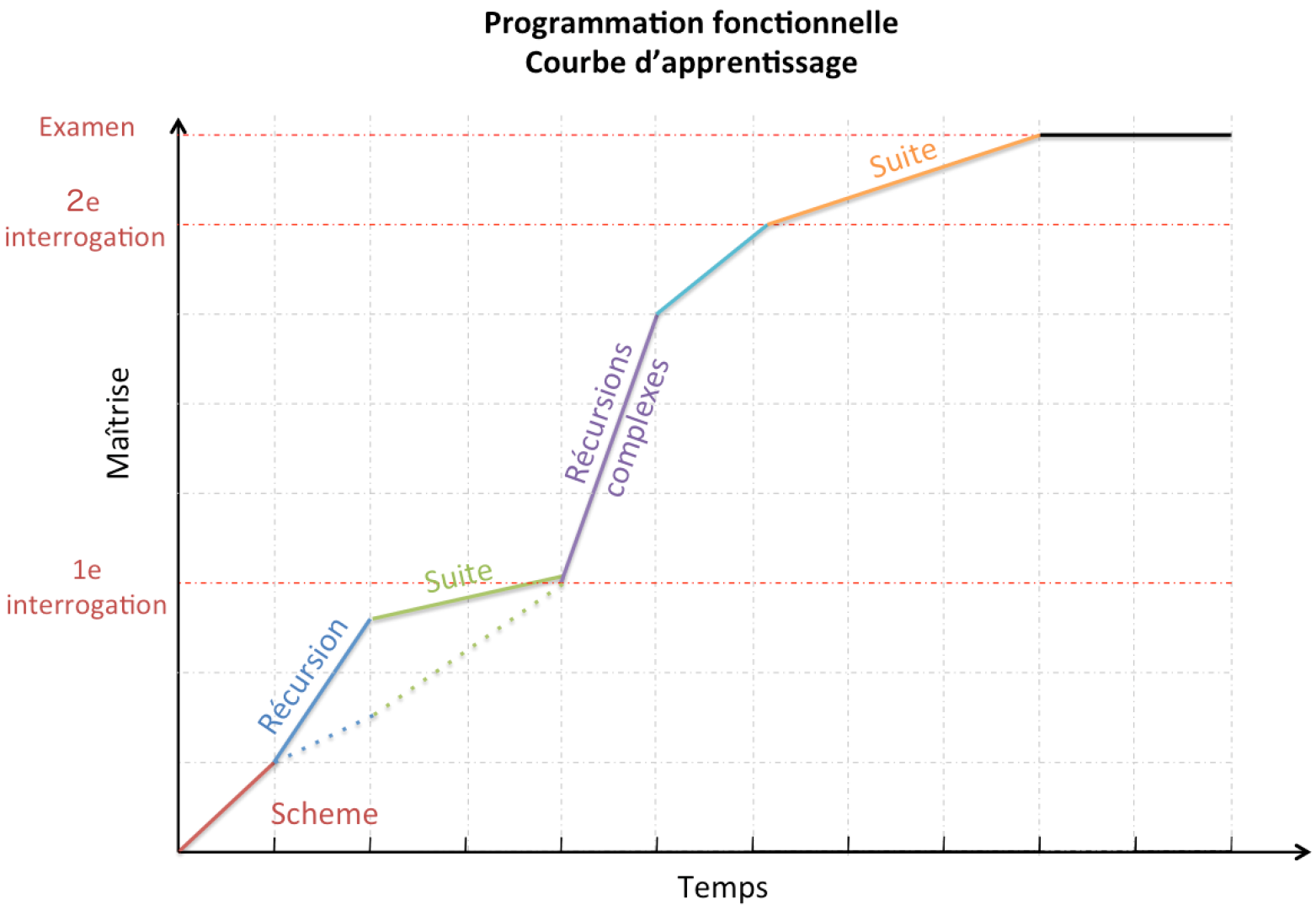
Mais aussi de profiter de l'élégance d'un nouveau paradigme (pour des problèmes appropriés).

Où en sommes-nous ?

1. Quoi ?
2. Pourquoi ?
3. Comment ?
   - Infos pratiques
   - Evaluation
4. Re-quoi ?
5. Conclusion

- 8-9 séances de répétition.
  - Participation active !
  - Faites des erreurs !
- 3 interrogations (16 mars, 20 avril et 11 mai).
  - Encourager une progression constante.
  - Sources de feedbacks !
  - Faites moins d'erreurs.
- 1 projet.
  - Interpréteur `Scheme` : `http://racket-lang.org/download/`
- 1 examen oral, deux questions tirées au sort.
  - Même niveau de difficulté que la dernière interrogation.
  - Ne faites plus d'erreur.


Histogramme des cotes
1e interrogation (2016)
0
1
2
3
4
5
6
7
8
9
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Cote


Programmation fonctionnelle
Courbe d'apprentissage
Examen
2e interrogation
1e interrogation
Maîtrise
Scheme
Récursion
Suite
Récursions complexes
Suite
Temps

Procédure d'évaluation :

Parcours de la spécification

- Si OK, parcours du code
  - Si OK, regard sur l'efficacité
    - Si OK, regard sur le temps de réalisation

1. Mauvaise utilisation des primitives
   - cons, list, append
   - map, apply
   - ...
2. Spécification absente
3. Spécification incorrecte
4. Spécification incomplète
   - Tous les arguments ne sont pas évoqués
5. Champ lexical impératif
6. Code déraisonnablement inefficace
7. Lenteur pour écrire une solution (manque d'entrainement)
8. Code inutilement complexe

Trois nouveautés :

- Langage interprété (typage dynamique) et notation préfixée :
  $\Rightarrow$ Perturbant au début.
- Programmation déclarative :
  $\Rightarrow$ La logique est différente ; la partie la plus difficile.
- Programmation déclarative fonctionnelle :
  $\Rightarrow$ Quelques nouveautés ; la partie fun (mais quand même difficile).

Où en sommes-nous ?

1. Quoi ?
2. Pourquoi ?
3. Comment ?
4. Re-quoi ?
   - Le paradigme déclaratif
   - Un peu de blabla
   - Ouf ! Un exemple
   - Conséquences de la démarche déclarative
   - Spécifications : ennuyeux mais utile !
5. Conclusion

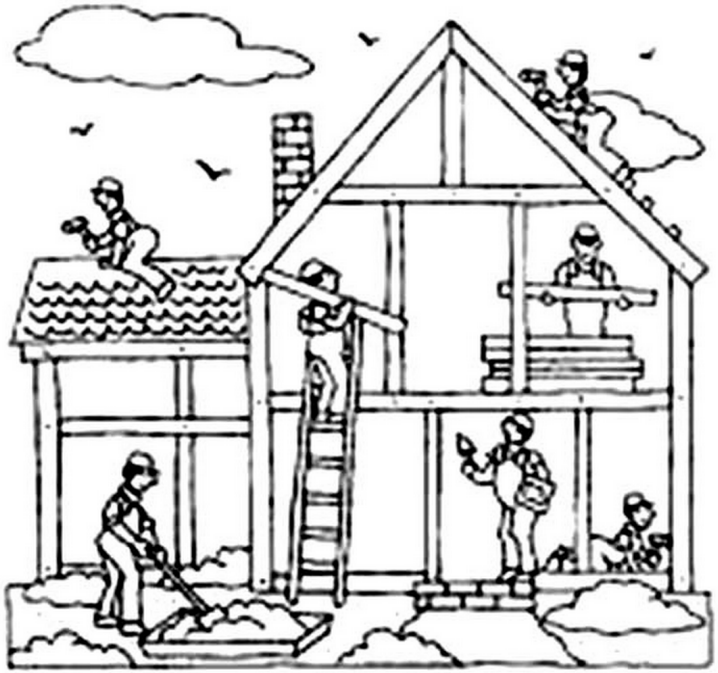
- Programmation impérative
  - Programmation procédurale (ex. fortran, C)
  - Programmation orientée objet (ex. Java, C#)
- Programmation déclarative
  - Programmation fonctionnelle (ex. Lisp, **Scheme**)
  - Programmation logique (ex. Prolog)

> *Un paradigme de programmation est un style fondamental de programmation informatique qui traite de la manière dont les solutions aux problèmes doivent être formulées [...].*
>
> — *Wikipedia, 2015*

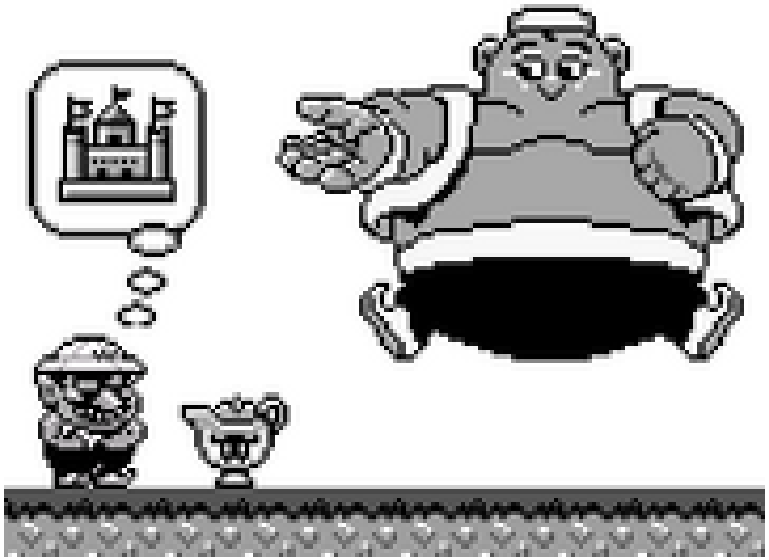
Bref : Une manière de penser et de concevoir la programmation et la résolution de problèmes.

Programmation impérative vs. programmation déclarative

Programmation impérative



Programmation déclarative



Declarative programming is a programming paradigm [...] that expresses the logic of a computation without describing its control flow.
Many languages applying this style attempt to minimize or ***eliminate side effects*** *by describing* ***what*** *the program* ***should accomplish*** *in terms of the problem domain,* ***rather than*** *describing* ***how*** *to go about accomplishing it as a sequence of the programming language primitives.*
This is in contrast with imperative programming, in which algorithms are implemented in terms of explicit steps.

— *Wikipedia, 2015*

Functional programming is a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

— *Wikipedia, 2015*

Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these languages are formed by using functions to combine basic values.

— *Graham Hutton*

$$\texttt{sum\_n}(n) = \sum_{i=0}^{n-1} i$$

Programmation impérative
(procédurale)

```
uint sum_n(uint n)
{
    uint s = 0;
    for (uint i=1; i<n; i++)
        s += i;
    return s;
}
```

Comment calcule-t-on *sum_n* ?

1. Initialise une variable s à 0
2. Pour i allant de 1 à n, ajoute i à s
3. Retourne s

$$\texttt{sum\_n}(n) = \sum_{i=0}^{n-1} i$$

Programmation impérative
(procédurale)

```
uint sum_n(uint n)
{
    uint s = 0;
    for (uint i=1; i<n; i++)
        s += i;
    return s;
}
```

Programmation déclarative
(fonctionnelle)

```
uint sum_n(uint n)
{
    if (n == 0)
        return 0;

    return n + sum_n(n-1)
}
```

$$\texttt{sum\_n}(n) = \sum_{i=0}^{n-1} i$$

C'est quoi *sum_n*(*n*) ?

La somme des n premiers naturels c'est

- 0 si n est nul
- n plus la somme des $n-1$ premiers naturels sinon

Rem. : $sum\_n : \mathbb{N} \to \mathbb{N}$ est une fonction mais *sum_n*(*n*) est un nombre

Programmation déclarative
(fonctionnelle)

```
uint sum_n(uint n)
{
    if (n == 0)
        return 0;

    return n + sum_n(n-1)
}
```

$$\texttt{sum\_n}(n) = \sum_{i=0}^{n-1} i$$

Programmation impérative :

Comment calcule-t-on *sum_n* ?

1. Initialise une variable s à 0
2. Pour i allant de 1 à n, ajoute i à s
3. Retourne s

Programmation déclarative :

C'est quoi *sum_n*(n) ?

La somme des n premiers naturels c'est

- 0 si n est nul
- n plus la somme des $n-1$ premiers naturels sinon

> *A functional language is a language that supports and encourages programming in a functional style.*
>
> — *Graham Hutton (again)*

La programmation déclarative/fonctionnelle est un paradigme ; une manière de penser et d'approcher les problèmes. Le langage permet d'aider plus ou moins à travailler dans ce sens.

Soit le prédicat `path?` $(G, N*, n2, S)$:

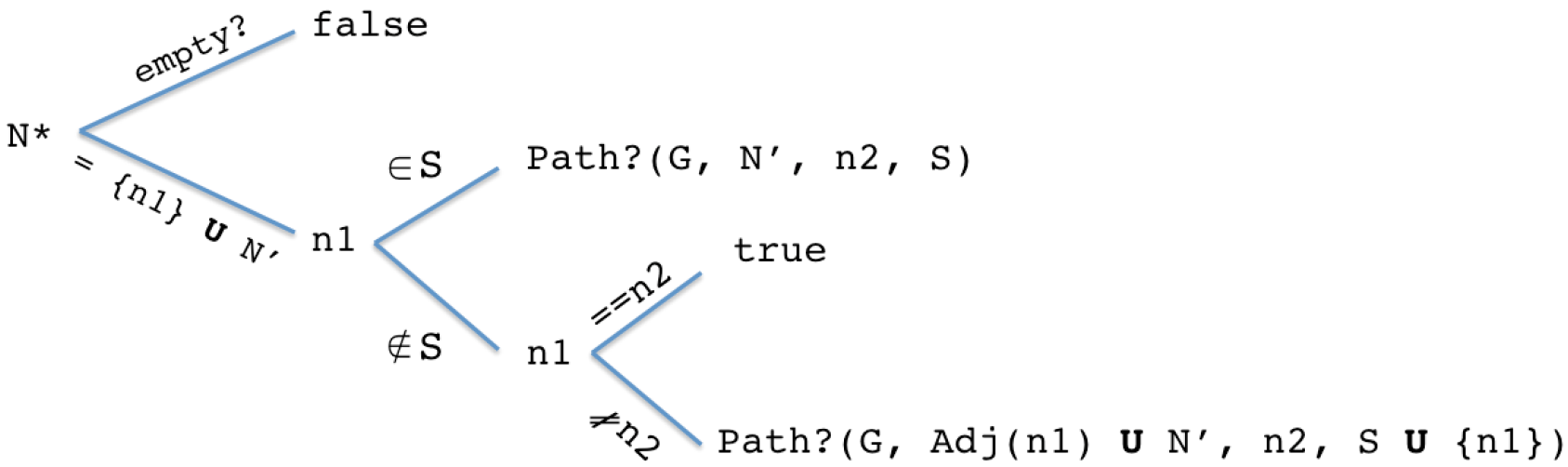
G un graph : $G = (V, E)$

$N*$ un ensemble de nœuds de G : $N* \subseteq V$

$n2$ un nœud de G : $n2 \in V$

S un ensemble de nœuds de G : $S \subseteq V$

$true \iff \exists n \in (N* \setminus S) : n \rightarrow_G n2$



$Adj(n) = \{n_i \in V \mid (n, n_i) \in E\}$

Effet de bord (*side effect*)

Conséquences de la démarche déclarative

> *En informatique, une fonction est dite à effet de bord [...] si elle modifie un état autre que sa valeur de retour. Par exemple, une fonction pourrait modifier une variable statique ou globale, modifier un ou plusieurs de ses arguments, [...]*
>
> — *Wikipedia, 2015*

Exemple :

```
static int counter;
...
int addAndReturn(int n)
{
    counter += n;
    return counter;
}
...
```

```
int main()
{
...
    counter = 0;
    x = addAndReturn(5);
    y = addAndReturn(5);
...
}
```

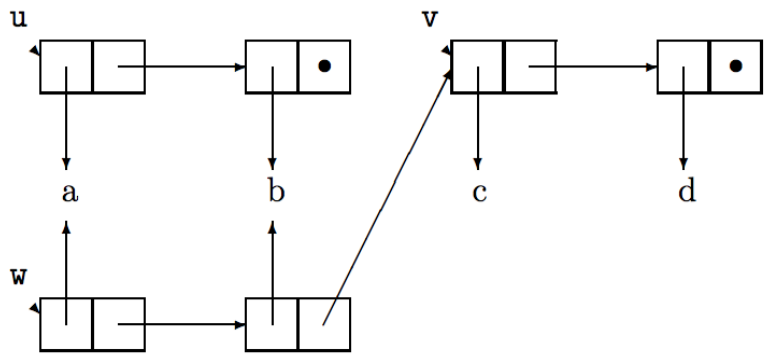
Problèmes ?

Problèmes :

- $x \neq y$
- Impossible de décrire viablement des objets à partir d'autres si ceux de référence changent...

Solution : **immuabilité** !

- On ne modifie rien, on crée de nouveaux objets



$$w = append(u, v)$$

- Pas de tableau (on utilise des listes à la place)

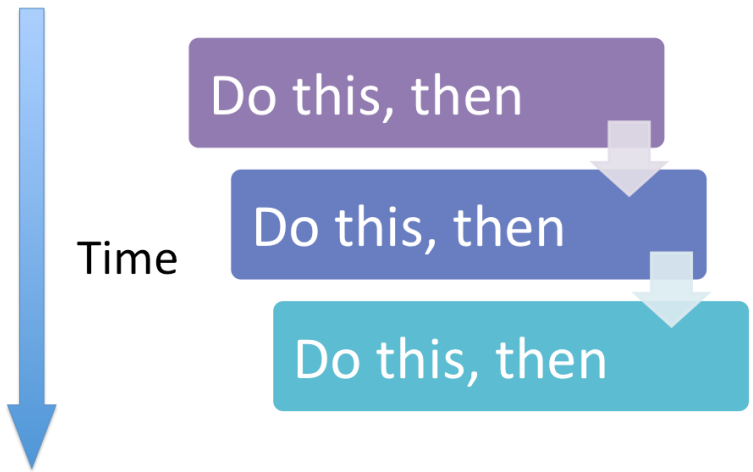
$$\Rightarrow \; O(1) \rightarrow O(n)$$

$$\texttt{sum\_n}(n) = \sum_{i=0}^{n-1} i$$

Programmation impérative
(procédurale)

```
uint sum_n(uint n)
{
    uint s = 0;
    for (uint i=1; i<n; i++)
        s += i;
    return s;
}
```

Modification des variables



Programmation déclarative :

1. Immuabilité $\Rightarrow$ la valeur de retour est **toujours** la même pour des arguments donnés.
2. L'ordre des clauses n'a plus d'importance :

```
uint sum_n(uint n)
{
    if (n == 0)
        return 0;

    return n + sum_n(n-1)
}
```

```
uint sum_n(uint n)
{
    if (n > 0)
        return n + sum_n(n-1)
    else
        return 0;
}
```

Parallélisation possible sans devoir prendre de précautions !

Remarque : champ lexical
Conséquences de la démarche déclarative

On ne dit pas...

- "On fait varier..."
- "Au départ la liste est vide"
- "Les nœuds déjà visités"
- "Initial", "final", "avant", "plus tôt", "puis", ...

Puisqu'on ne dicte pas à la machine comment elle doit s'y prendre, on ne connaît pas la séquence d'opérations. Du point de vue du programmeur fonctionnel, les fonctions sont des boîtes noires. Il connaît juste la relation entrée-sortie et la complexité des fonctions.

En poussant le vice jusqu'au bout, on devrait éviter de dire "`insert(T, x)` ajoute l'élément `x` à l'arbre `T`", puisqu'on construit un nouvel arbre.

- Spécification d'une fonction

  But (gen.) Décrire comment la sortie dépend des entrées (pas comment elle s'y prend pour y arriver)

- Programmation déclarative

  But Exprimer un objet en fonction d'autres objets

Lien fort entre spécification et programmation déclarative !

Une bonne spécification aide grandement l'emploi de la programmation déclarative.

Tableau initial: [5, 2, 4, 6, 1, 3]

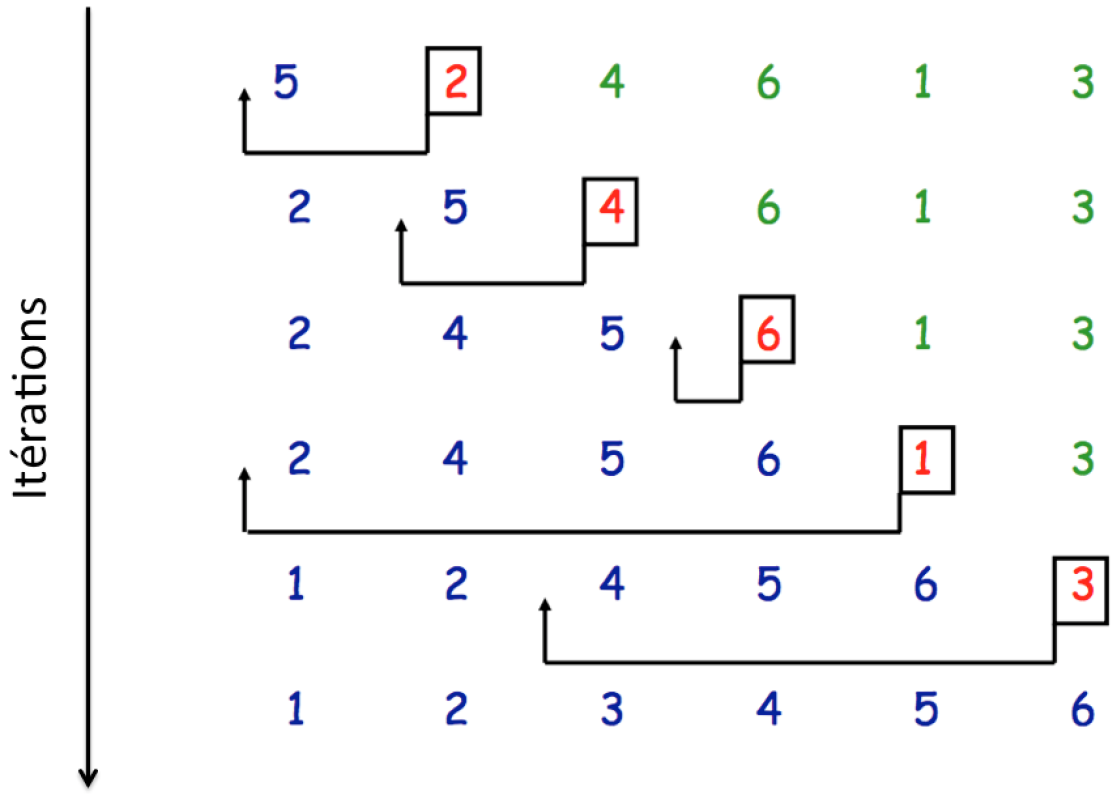


Tableau final: [1, 2, 3, 4, 5, 6]

```
Insertion-Sort(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

- Invariant : (pour la boucle externe) le sous-tableau $A[1..j-1]$ contient les éléments du tableau original $A[1..j-1]$ ordonnés.
- On doit montrer que
  - l'invariant est vrai avant la première itération
  - l'invariant est vrai avant chaque itération suivante
  - En sortie de boucle, l'invariant implique que l'algorithme est correct

insert(ls, x) si ls est une liste de nombres d'ordre croissant et x est un nombre, insert(ls, x) **est** la liste ls où x a été ajouté afin de respecter la propriété d'ordre.

sort(ls, ts) si ls est une liste de nombres d'ordre croissant et ts est une liste de nombres, sort(ls, ts) **est** la fusion triée de ls et ts. (sort(*liste vide*, ts) trie donc ts).

```
sort(ls, ts)
1   if ts is empty
2         return ls
3   return sort(insert(ls, ts.val), ts.next)
```

- L'invariant est devenu partie intégrante de la spécification !
- La démonstration est devenue une preuve par induction.

insert(ls, x) si ls est une liste de nombres d'ordre croissant et x est un nombre, insert(ls, x) **est** la liste ls où x a été ajouté afin de respecter la propriété d'ordre.

sort(ls, ts) si ls est une liste de nombres d'ordre croissant et ts est une liste de nombres, sort(ls, ts) **est** la fusion triée de ls et ts. (sort(*liste vide*, ts) trie donc ts).

```
sort(ls, ts)
1   if ts is empty
2         return ls
3   return sort(insert(ls, ts.val), ts.next)
```

Les arguments de sort sont du bon type lors de l'appel récursif : insert($ls, ts.val$) est bien une liste triée de nombres et ts est bien une liste de nombres.

insert(ls, x) si ls est une liste de nombres d'ordre croissant et x est un nombre, insert(ls, x) **est** la liste ls où x a été ajouté afin de respecter la propriété d'ordre.

sort(ls, ts) si ls est une liste de nombres d'ordre croissant et ts est une liste de nombres, sort(ls, ts) **est** la fusion triée de ls et ts. (sort(*liste vide*, ts) trie donc ts).

```
sort(ls, ts)
1  if ts is empty
2        return ls
3  return sort(insert(ls, ts.val), ts.next)
```

Le type de retour de sort est correct si les entrées sont du bon type : trivialement vrai pour le cas de base. Vrai par induction puisque insert renvoie une liste.

insert(ls, x) si ls est une liste de nombres d'ordre croissant et x est un nombre, insert(ls, x) **est** la liste ls où x a été ajouté afin de respecter la propriété d'ordre.

sort(ls, ts) si ls est une liste de nombres d'ordre croissant et ts est une liste de nombres, sort(ls, ts) **est** la fusion triée de ls et ts. (sort(*liste vide*, ts) trie donc ts).

```
sort(ls, ts)
1   if ts is empty
2         return ls
3   return sort(insert(ls, ts.val), ts.next)
```

On atteindra le cas de base puisque la taille de la liste en second argument décroît à chaque appel récursif.

insert(ls, x) si ls est une liste de nombres d'ordre croissant et x est un nombre, insert(ls, x) **est** la liste ls où x a été ajouté afin de respecter la propriété d'ordre.

sort(ls, ts) si ls est une liste de nombres d'ordre croissant et ts est une liste de nombres, sort(ls, ts) **est** la fusion triée de ls et ts. (sort(*liste vide*, ts) trie donc ts).

```
sort(ls, ts)
1   if ts is empty
2         return ls
3   return sort(insert(ls, ts.val), ts.next)
```

Au vu de la spécification de insert, sort(ls, ts) et sort(insert($ls, ts.val$), $ts.next$) renvoient bien la même liste.

Plan

Trois nouveautés :

- Langage interprété (typage dynamique) et notation préfixée :
  $\Rightarrow$ Perturbant au début.
- Programmation déclarative :
  $\Rightarrow$ La logique est différente ; la partie la plus difficile.
- Programmation déclarative fonctionnelle :
  $\Rightarrow$ Quelques nouveautés ; la partie fun (mais quand même difficile).

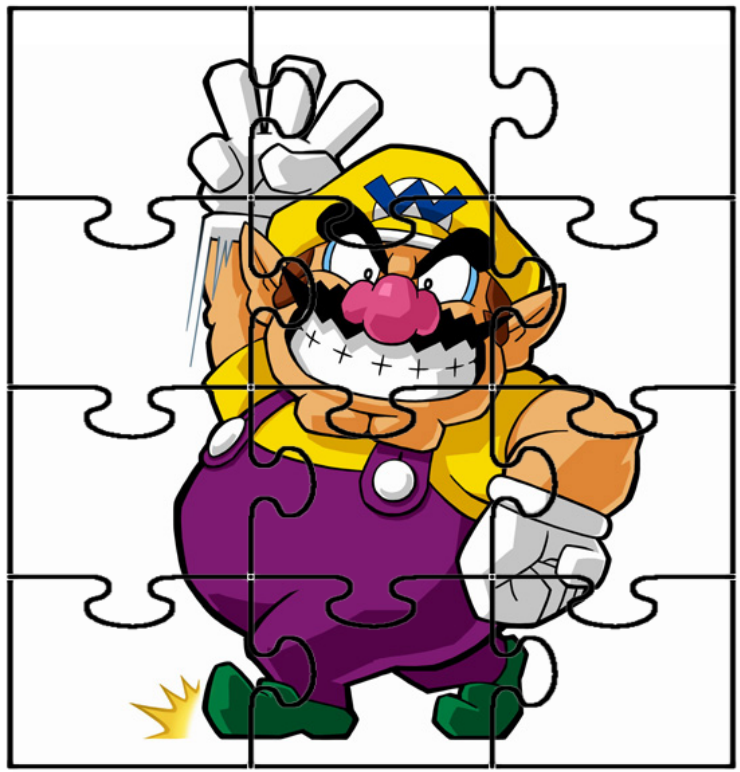
Penser déclarativement.

1. Percevoir les relations entre les entités.

2. Exprimer correctement ces relations.
3. ... avec le bon vocabulaire et rapidement !

Penser déclarativement.

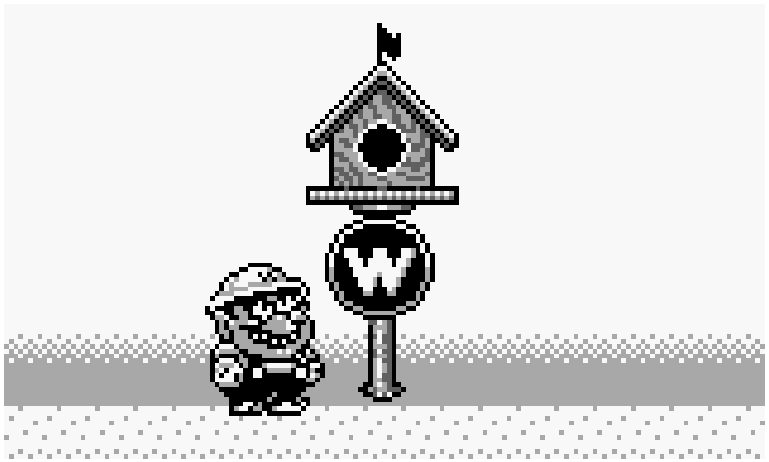
1. Percevoir les relations entre les entités.



2. Exprimer correctement ces relations.
3. ... avec le bon vocabulaire et rapidement !

Penser déclarativement.

1. Percevoir les relations entre les entités.
2. Exprimer correctement ces relations (en code et en spécification).



3. ... avec le bon vocabulaire et rapidement !

Penser déclarativement :

1. Percevoir les relations entre les entités.
2. Exprimer correctement ces relations.
3. ... avec le bon vocabulaire et rapidement !

C'est quoi la programmation fonctionnelle ?

1. Programmation déclarative
   - Description de la logique, pas de la suite d'instructions
     - Variables immuables
     - Pas d'effet de bord
     - Pas de notion explicite de temps
2. Programmation déclarative fonctionnelle
   - La logique est décrite à l'aide de fonctions

Objectifs

- Apprendre et maitriser un nouveau paradigme.
  - Savoir écrire **rapidement** un programme fonctionnel **correct**.
    - → Ecrire une spécification correcte.
- Reconnaître les situations dans lesquelles l'approche fonctionnelle est plus avantageuse que l'approche impérative.


SAFELY ENDANGERED
SWEET JESUS, POOH!
THAT'S NOT HONEY
YOU'RE EATING
RECURSION
SAFELY ENDANGERED
SWEET JESUS, POOH!
THAT'S NOT HONEY
YOU'RE EATING
RECURSION