Paradigmes de programmation

Prof d'info

Lycée Thiers

Crédits

- Wikipedia

Crédits

- Wikipedia
- SQL comme langage utilisant un paradigme logique

Définition

≪ Un paradigme de programmation fournit (et détermine) la vue qu'a le développeur de l'exécution de son programme ≫ (Wikipedia). Les paradigmes de programmation abordés en MP2I sont :

Paradigme impératif structuré Dans l'esprit du programme de MP2I, c'est le langage C qui répond le plus à cette façon de travailler.

Définition

≪ Un paradigme de programmation fournit (et détermine) la vue qu'a le développeur de l'exécution de son programme ≫ (Wikipedia). Les paradigmes de programmation abordés en MP2I sont :

Paradigme impératif structuré Dans l'esprit du programme de MP2I, c'est le langage C qui répond le plus à cette façon de travailler.

Paradigme déclaratif On énonce ce qu'on veut MAIS pas COMMENT on veut l'obtenir. Il se décompose en deux sous-catégories :

Définition

≪ Un paradigme de programmation fournit (et détermine) la vue qu'a le développeur de l'exécution de son programme ≫ (Wikipedia). Les paradigmes de programmation abordés en MP2I sont :

Paradigme impératif structuré Dans l'esprit du programme de MP2I, c'est le langage C qui répond le plus à cette façon de travailler.

Paradigme déclaratif On énonce ce qu'on veut MAIS pas COMMENT on veut l'obtenir. Il se décompose en deux sous-catégories :

- **Paradigme fonctionnel** En MP2I, c'est le langage OCaml qui illustre cette méthode. Tout est fonction.

Définition

« Un paradigme de programmation fournit (et détermine) la vue qu'a le développeur de l'exécution de son programme » (Wikipedia). Les paradigmes de programmation abordés en MP2I sont :

Paradigme impératif structuré Dans l'esprit du programme de MP2I, c'est le langage C qui répond le plus à cette façon de travailler.

Paradigme déclaratif On énonce ce qu'on veut MAIS pas COMMENT on veut l'obtenir. Il se décompose en deux sous-catégories :

- **Paradigme fonctionnel** En MP2I, c'est le langage OCaml qui illustre cette méthode. Tout est fonction.
- **Paradigme logique** Basé sur la logique.

Définition

« Un paradigme de programmation fournit (et détermine) la vue qu'a le développeur de l'exécution de son programme » (Wikipedia). Les paradigmes de programmation abordés en MP2I sont :

Paradigme impératif structuré Dans l'esprit du programme de MP2I, c'est le langage C qui répond le plus à cette façon de travailler.

Paradigme déclaratif On énonce ce qu'on veut MAIS pas COMMENT on veut l'obtenir. Il se décompose en deux sous-catégories :

Paradigme fonctionnel En MP2I, c'est le langage OCaml qui illustre cette méthode. Tout est fonction.

Paradigme logique Basé sur la logique.

- Code plus flexible puisque dépendant de la logique, non de l'ordre des instructions.

Définition

« Un paradigme de programmation fournit (et détermine) la vue qu'a le développeur de l'exécution de son programme » (Wikipedia). Les paradigmes de programmation abordés en MP2I sont :

Paradigme impératif structuré Dans l'esprit du programme de MP2I, c'est le langage C qui répond le plus à cette façon de travailler.

Paradigme déclaratif On énonce ce qu'on veut MAIS pas COMMENT on veut l'obtenir. Il se décompose en deux sous-catégories :

Paradigme fonctionnel En MP2I, c'est le langage OCaml qui illustre cette méthode. Tout est fonction.

Paradigme logique Basé sur la logique.

- Code plus flexible puisque dépendant de la logique, non de l'ordre des instructions.
- Code plus simple à comprendre car suit la pensée logique.

Définition

« Un paradigme de programmation fournit (et détermine) la vue qu'a le développeur de l'exécution de son programme » (Wikipedia). Les paradigmes de programmation abordés en MP2I sont :

Paradigme impératif structuré Dans l'esprit du programme de MP2I, c'est le langage C qui répond le plus à cette façon de travailler.

Paradigme déclaratif On énonce ce qu'on veut MAIS pas COMMENT on veut l'obtenir. Il se décompose en deux sous-catégories :

Paradigme fonctionnel En MP2I, c'est le langage OCaml qui illustre cette méthode. Tout est fonction.

Paradigme logique Basé sur la logique.

- Code plus flexible puisque dépendant de la logique, non de l'ordre des instructions.
- Code plus simple à comprendre car suit la pensée logique.
- Exemples : SQL ; Prolog

Autres paradigmes

D'autres paradigmes sont couramment utilisés par les développeurs professionels. Citons les pour information : programmation orientée objets (POO), programmation orientée processus.

Langages et paradigmes

- « A un langage de programmation donné correspond un unique paradigme de programmation » : Faux.

Langages et paradigmes

- « A un langage de programmation donné correspond un unique paradigme de programmation » : Faux.
- Les langages qui contraignent fortement le choix d'un paradigme sont plutôt rares. Citons quand même Haskell pour la programmation fonctionnelle , Smalltalk pour la POO, et Prolog pour la programmation logique.

Langages et paradigmes

- ≪ A un langage de programmation donné correspond un unique paradigme de programmation ≫ : Faux.
- Les langages qui contraignent fortement le choix d'un paradigme sont plutôt rares. Citons quand même Haskell pour la programmation fonctionnelle , Smalltalk pour la POO, et Prolog pour la programmation logique.
- D'autres langages sont plus souples. Par exemple OCaml, qui est un langage fonctionnel, comporte des traits impératifs et on peut donc écrire un programme en respectant le paradigme impératif structuré dans ce langage.

Langages et paradigmes

- Un paradigme donné est juste une façon d'aborder les problèmes. On peut toujours écrire un programme respectant ce paradigme même si le langage choisi n'est pas conçu pour le supporter.

Langages et paradigmes

- Un paradigme donné est juste une façon d'aborder les problèmes. On peut toujours écrire un programme respectant ce paradigme même si le langage choisi n'est pas conçu pour le supporter.
- Ainsi, il est possible d'écrire un programme s'inspirant de la POO en langage C alors que ce dernier n'est pas conçu pour cela.

Langages et paradigmes

- Un paradigme donné est juste une façon d'aborder les problèmes. On peut toujours écrire un programme respectant ce paradigme même si le langage choisi n'est pas conçu pour le supporter.
- Ainsi, il est possible d'écrire un programme s'inspirant de la POO en langage C alors que ce dernier n'est pas conçu pour cela.
- Dans les faits, le développeur confronté à un problème choisit de le résoudre avec son langage de programmation préféré et adopte un paradigme qu'il pense adapté même s'il doit parfois faire quelques contorsions pour forcer son langage à se conformer au paradigme.

Présentation

- La *programmation impérative* est un paradigme de programmation qui décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'*état* (c'est à dire le contenu de la mémoire) du processus.

Présentation

- La *programmation impérative* est un paradigme de programmation qui décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'*état* (c'est à dire le contenu de la mémoire) du processus.
- Ce type de programmation est le plus répandu parmi l'ensemble des langages de programmation existants

Un paradigme adapté aux processeurs

- La quasi-totalité des processeurs qui équipent les ordinateurs sont de nature impérative : ils sont faits pour exécuter en séquence des instructions élémentaires, codées sous forme d'*opcodes* (pour operation codes).

Un paradigme adapté aux processeurs

- La quasi-totalité des processeurs qui équipent les ordinateurs sont de nature impérative : ils sont faits pour exécuter en séquence des instructions élémentaires, codées sous forme d'*opcodes* (pour operation codes).
- L'ensemble des opcodes forme le *langage machine spécifique* à l'architecture du processeur (la façon dont ce dernier est conçu).

Un paradigme adapté aux processeurs

- La quasi-totalité des processeurs qui équipent les ordinateurs sont de nature impérative : ils sont faits pour exécuter en séquence des instructions élémentaires, codées sous forme d'*opcodes* (pour operation codes).
- L'ensemble des opcodes forme le *langage machine spécifique* à l'architecture du processeur (la façon dont ce dernier est conçu).
- L'*état du processus* à un instant donné est défini par le contenu de la mémoire centrale à cet instant. Le processus passe donc par différents états lors de l'exécution.

Un paradigme adapté aux processeurs

- La quasi-totalité des processeurs qui équipent les ordinateurs sont de nature impérative : ils sont faits pour exécuter en séquence des instructions élémentaires, codées sous forme d'*opcodes* (pour operation codes).
- L'ensemble des opcodes forme le *langage machine spécifique* à l'architecture du processeur (la façon dont ce dernier est conçu).
- L'*état du processus* à un instant donné est défini par le contenu de la mémoire centrale à cet instant. Le processus passe donc par différents états lors de l'exécution.
- La programmation impérative est la plus intuitive. Une recette de cuisine peut être considérée comme de la programmation impérative. Chaque étape de la recette est une instruction et l'état d'avancement du gateau est ce qui correspond le mieux à l'état de la mémoire à un instant donné.

Les instructions de base

- Les langages impératifs de haut niveau comportent 4 types d'instructions principales :

Les instructions de base

- Les langages impératifs de haut niveau comportent 4 types d'instructions principales :
  1. la séquence d'instructions

Les instructions de base

- Les langages impératifs de haut niveau comportent 4 types d'instructions principales :
  1. la séquence d'instructions
  2. l'assignation ou affectation,

Les instructions de base

- Les langages impératifs de haut niveau comportent 4 types d'instructions principales :
  1. la séquence d'instructions
  2. l'assignation ou affectation,
  3. l'instruction conditionnelle,

Les instructions de base

- Les langages impératifs de haut niveau comportent 4 types d'instructions principales :
  1. la séquence d'instructions
  2. l'assignation ou affectation,
  3. l'instruction conditionnelle,
  4. les boucles,

Les instructions de base

- Les langages impératifs de haut niveau comportent 4 types d'instructions principales :
  1. la séquence d'instructions
  2. l'assignation ou affectation,
  3. l'instruction conditionnelle,
  4. les boucles,
- Pour être exhaustif, il convient de mentionner l'existence d'un cinquième type complètement hors programme en MP2I : le *branchement inconditionnel* ou *saut*.

Programmation impérative

Séquences d'instructions

- Une *séquence d'instructions*, (ou bloc d'instruction) désigne le fait de faire exécuter par la machine une instruction, puis une autre, etc..., en séquence.

Programmation impérative

Séquences d'instructions

- Une *séquence d'instructions*, (ou bloc d'instruction) désigne le fait de faire exécuter par la machine une instruction, puis une autre, etc..., en séquence.
- Par exemple :

```
1        ouvrirConnexion(); envoyerMessage(); fermerConnexion();
```

est une séquence d'instructions. Cette construction se distingue du fait d'exécuter en parallèle des instructions.

Programmation impérative

Instructions d'assignations

- Il s'agit d'instructions fortement en lien avec la notion de *variable*. Elles effectuent une opération sur l'information en mémoire et y enregistrent le résultat.

Programmation impérative

Instructions d'assignations

- Il s'agit d'instructions fortement en lien avec la notion de *variable*. Elles effectuent une opération sur l'information en mémoire et y enregistrent le résultat.
- Par exemple :

```
1        x ← 2+3
```

assigne la valeur 2+3 donc 5 à la variable x. Cela signifie quà l'adresse mémoire de x, le système écrit la constante 5.

Programmation impérative

Instructions conditionnelles

- Les instructions conditionnelles permettent à un bloc d'instructions de n'être exécuté que si une condition prédéterminée est réalisée. Dans le cas contraire, les instructions sont ignorées et la séquence d'exécution continue à partir de l'instruction qui suit immédiatement la fin du bloc.

Programmation impérative

Instructions conditionnelles

- Les instructions conditionnelles permettent à un bloc d'instructions de n'être exécuté que si une condition prédéterminée est réalisée. Dans le cas contraire, les instructions sont ignorées et la séquence d'exécution continue à partir de l'instruction qui suit immédiatement la fin du bloc.
- Ainsi

```
1       si connexionOuverte()
2                   alors  {envoyerMessage(); afficher("OK");}
3                   sinon afficher("envoi impossible");
4       fin_si
5       afficher("terminé");
```

n'envoie le message que si la connexion est ouverte. En revanche, l'affichage de `"terminé"` a toujours lieu.

Programmation impérative

Instructions de bouclage

- Les instructions de bouclage répètent plusieurs fois une certain bloc d'instructions.

Programmation impérative

Instructions de bouclage

- Les instructions de bouclage répètent plusieurs fois une certain bloc d'instructions.
- Pour indiquer la fin du bouclage, le développeur peut appliquer 3 stratégies : `pour, répéter, tant que`.

Instructions de bouclage : Pour

Le bouclage s'arrête quand le bloc a été exécuté un certain nombre de fois fixé à l'avance.

- Par exemple, on peut décider d'envoyer 100 fois un message donné puis de s'arrêter (boucle *pour*).

```
1    pour i allant de 1 à 100 faire
2        envoyerLettreDeRelance(numéro=i)
3    fin_pour;
```

Instructions de bouclage : Pour

Le bouclage s'arrête quand le bloc a été exécuté un certain nombre de fois fixé à l'avance.

- Par exemple, on peut décider d'envoyer 100 fois un message donné puis de s'arrêter (boucle *pour*).

```
1    pour i allant de 1 à 100 faire
2        envoyerLettreDeRelance(numéro=i)
3    fin_pour;
```

- Cette stratégie impose de connaître à l'avance le nombre de passages dans la boucle.

Instructions de bouclage : Répéter

- Le bouclage s'arrête quand une certaine condition est vérifiée.

Instructions de bouclage : Répéter

- Le bouclage s'arrête quand une certaine condition est vérifiée.
- Par exemple, on peut décider d'envoyer un message de rappel de paiement tant qu'une certaine personne n'a pas acquitté sa dette (boucle *répéter*).

```
1        repeter
2                envoyerCourrierRappel()
3        jusqua paiementFait();
```

Stratégie plus souple que la précédente mais risque de *boucle infinie*.

Instructions de bouclage : Répéter

- Le bouclage s'arrête quand une certaine condition est vérifiée.
- Par exemple, on peut décider d'envoyer un message de rappel de paiement tant qu'une certaine personne n'a pas acquitté sa dette (boucle *répéter*).

```
1        repeter
2                envoyerCourrierRappel()
3        jusqua paiementFait();
```

  Stratégie plus souple que la précédente mais risque de *boucle infinie*.
- l'instruction est exécutée au moins une fois.

Instructions de bouclage : Répéter

- Le bouclage s'arrête quand une certaine condition est vérifiée.
- Par exemple, on peut décider d'envoyer un message de rappel de paiement tant qu'une certaine personne n'a pas acquitté sa dette (boucle *répéter*).

```
1        repeter
2              envoyerCourrierRappel()
3        jusqua paiementFait();
```

Stratégie plus souple que la précédente mais risque de *boucle infinie*.

- l'instruction est exécutée au moins une fois.
- Variante : Tant qu'une certaine condition est réalisée, on continue :

```
1        tant_que paiementNonFait() faire
2          envoyerCourrierRappel()
3        fin_tant_que
```

L'instruction peut ne jamais être exécutée.

Système Turing-complet

- Tout les programmes écrits avec ces 4 sortes d'instructions peuvent être traduits de façon à fonctionner sur une *machine de Turing*.

Système Turing-complet

- Tout les programmes écrits avec ces 4 sortes d'instructions peuvent être traduits de façon à fonctionner sur une *machine de Turing*.
- Et réciproquement, tout programme réalisable avec une machine de Turing peut être traduit dans un formalisme utilisant ces 4 sortes d'instructions (on parle alors de système *Turing-complet*).

Système Turing-complet

- Tout les programmes écrits avec ces 4 sortes d'instructions peuvent être traduits de façon à fonctionner sur une *machine de Turing*.
- Et réciproquement, tout programme réalisable avec une machine de Turing peut être traduit dans un formalisme utilisant ces 4 sortes d'instructions (on parle alors de système *Turing-complet*).
- Remarque : Tout programme réalisé avec un type de boucle (pour, répéter, tant que), peut s'écrire dans un autre.

Branchement sans condition

Pour info uniquemment :

- Dans les langages *bas niveau* (et même en langage C), on peut exécuter des sauts inconditionnels (instruction `GOTO`).

Branchement sans condition

Pour info uniquemment :

- Dans les langages *bas niveau* (et même en langage C), on peut exécuter des sauts inconditionnels (instruction `GOTO`).
- Il s'agit simplement de dire au système qu'on se rend maintenant à une ligne précise du programme (par exemple `GOTO 103` indique qu'on se rend à la ligne 103 du programme).

Branchement sans condition

Pour info uniquemment :

- Dans les langages *bas niveau* (et même en langage C), on peut exécuter des sauts inconditionnels (instruction `GOTO`).
- Il s'agit simplement de dire au système qu'on se rend maintenant à une ligne précise du programme (par exemple `GOTO 103` indique qu'on se rend à la ligne 103 du programme).
- Avec une telle instruction, on change ainsi le flot de contrôle naturel du programme qui consiste normalement à aller exécuter l'instruction suivante.

Branchement sans condition

Pour info uniquemment :

- Dans les langages *bas niveau* (et même en langage C), on peut exécuter des sauts inconditionnels (instruction `GOTO`).
- Il s'agit simplement de dire au système qu'on se rend maintenant à une ligne précise du programme (par exemple `GOTO 103` indique qu'on se rend à la ligne 103 du programme).
- Avec une telle instruction, on change ainsi le flot de contrôle naturel du programme qui consiste normalement à aller exécuter l'instruction suivante.
- Cette instruction était beaucoup utilisée dans les langages primitifs que sont Fortan pré-90 et BASIC pour réaliser des boucles et d'autres structures de contrôle.

Branchement sans condition

Pour info uniquemment :

- Les instructions de bouclage peuvent être vues comme la combinaison d'un branchement conditionnel et d'un saut.

Branchement sans condition

Pour info uniquemment :

- Les instructions de bouclage peuvent être vues comme la combinaison d'un branchement conditionnel et d'un saut.
- Les appels à une fonction ou une procédure (donc un Sous-programme) correspondent à un saut, complété du passage de paramètres, avec un saut en retour.

Branchement sans condition

Pour info uniquemment :

- Les instructions de bouclage peuvent être vues comme la combinaison d'un branchement conditionnel et d'un saut.
- Les appels à une fonction ou une procédure (donc un Sous-programme) correspondent à un saut, complété du passage de paramètres, avec un saut en retour.
- Or, dans les langages modernes, les structures conditionnelles et de bouclages sont directement disponible : elles rendent inutile l'utilisation de `GOTO`.

Branchement sans condition

Pour info uniquemment :

- Les instructions de bouclage peuvent être vues comme la combinaison d'un branchement conditionnel et d'un saut.
- Les appels à une fonction ou une procédure (donc un Sous-programme) correspondent à un saut, complété du passage de paramètres, avec un saut en retour.
- Or, dans les langages modernes, les structures conditionnelles et de bouclages sont directement disponible : elles rendent inutile l'utilisation de `GOTO`.
- Les développeurs préfèrent souvent éviter l'usage des sauts inconditionnels car ils rendent les programmes peu lisibles (? les avis divergent à ce propos) et surtout difficiles à maintenir (si j'ajoute une ligne de code avant `GOTO 103`, je risque d'avoir un problème).

Branchement sans condition

- Dans certains langages comme Python, des instructions comme `continue` ou `break` remplacent certains usages de `GOTO`.

Branchement sans condition

- Dans certains langages comme Python, des instructions comme `continue` ou `break` remplacent certains usages de `GOTO`.
- En langage C, il existe encore une instruction `goto` (en minuscules) mais elle est hors programme MP2I.

Branchement sans condition

- Dans certains langages comme Python, des instructions comme `continue` ou `break` remplacent certains usages de `GOTO`.
- En langage C, il existe encore une instruction `goto` (en minuscules) mais elle est hors programme MP2I.
- La méthode de programmation qui consiste à utiliser seulement les séquences, affectations, branchements conditionnels et bouclages mais pas l'instruction `GOTO` est appelée *programmation impérative structurée*.
  C'est ce paradigme qui est au programme en MP2I et nous évitons d'écrire explicitement `GOTO`.

Branchement sans condition

- Dans certains langages comme Python, des instructions comme `continue` ou `break` remplacent certains usages de `GOTO`.
- En langage C, il existe encore une instruction `goto` (en minuscules) mais elle est hors programme MP2I.
- La méthode de programmation qui consiste à utiliser seulement les séquences, affectations, branchements conditionnels et bouclages mais pas l'instruction `GOTO` est appelée *programmation impérative structurée*.
  C'est ce paradigme qui est au programme en MP2I et nous évitons d'écrire explicitement `GOTO`.
- Les `GOTO` sont pourtant bien présents : ils sont cachés derrière les appels de fonctions/procédures et la gestion des *exceptions*.

Présentation à effectuer au moment du cours sur OCaml

Présentation

- Cette catégorie de paradigmes consiste à à déclarer les données du problème, puis à demander au programme de le résoudre.

Présentation

- Cette catégorie de paradigmes consiste à à déclarer les données du problème, puis à demander au programme de le résoudre.
- On décrit donc plutôt le *Quoi* que le *Comment*.

Présentation

- Cette catégorie de paradigmes consiste à à déclarer les données du problème, puis à demander au programme de le résoudre.
- On décrit donc plutôt le *Quoi* que le *Comment*.
- On assemble des composants logiciels indépendants et *sans état* (ce qui signifie que l'appel à un de ces composants avec les mêmes arguments produit toujours le même résultat et ce, quel que soit l'historique des commandes au moment de l'appel).

Présentation

- Cette catégorie de paradigmes consiste à à déclarer les données du problème, puis à demander au programme de le résoudre.
- On décrit donc plutôt le *Quoi* que le *Comment*.
- On assemble des composants logiciels indépendants et *sans état* (ce qui signifie que l'appel à un de ces composants avec les mêmes arguments produit toujours le même résultat et ce, quel que soit l'historique des commandes au moment de l'appel).
- Ainsi, les pages HTML sont déclaratives car elles décrivent ce que *contient* une page (texte, titres, paragraphes, etc.) et non *comment* les afficher (positionnement, couleurs, polices de caractères...) (cet affichage étant normalement plutôt dévolu au langage CSS).
  On ne se préoccupe pas de la manière dont le navigateur va respecter les instructions du moment qu'il fait ce qu'on lui demande.

Paradigmes déclaratifs

Programmation descriptive

Pour info.
Il s'agit d'écrire des programmes à l'expressivité réduite en se contentant d'écrire des structures de données (par exemple, HTML ou LaTeX).

Paradigmes déclaratifs

Programmation fonctionnelle

- Un programme est une *fonction* au sens mathématique du terme.

Paradigmes déclaratifs

Programmation fonctionnelle

- Un programme est une *fonction* au sens mathématique du terme.
- Les fonctions y sont sans état (si je calcule deux fois $f(x)$, j'obtiendrai le même résultat quel que soit le nombre et la nature des instructions effectuées entre ces deux appels)

Paradigmes déclaratifs

Programmation fonctionnelle

- Un programme est une *fonction* au sens mathématique du terme.
- Les fonctions y sont sans état (si je calcule deux fois $\mathrm{f}(x)$, j'obtiendrai le même résultat quel que soit le nombre et la nature des instructions effectuées entre ces deux appels)
- Pas d'*effets de bords*

Paradigmes déclaratifs

Programmation fonctionnelle

- Un programme est une *fonction* au sens mathématique du terme.
- Les fonctions y sont sans état (si je calcule deux fois $\mathrm{f}(x)$, j'obtiendrai le même résultat quel que soit le nombre et la nature des instructions effectuées entre ces deux appels)
- Pas d'*effets de bords*
- Exemples : OCaml, Haskell.

Effets de bords

Présentation

- Considérons le programme

```
1    Alice ← description d'Alice;
2    coiffeur(); manucure(); tatoo();
```

Alice est une variable *globale*.

Effets de bords

Présentation

- Considérons le programme

```
1        Alice ← description d'Alice;
2        coiffeur(); manucure(); tatoo();
```

  Alice est une variable *globale*.

- On peut imaginer que les procédures `coiffeur(); manucure(); tatoo();` s'appliquent à `Alice`. Dans un paradigme impératif, la coiffure, les ongles et la peau d'Alice ont été modifiés. On dit qu'elle a subi des *effets de bords* (fait de modifier localement une variable globale). La personne qui sort de ces 3 boutiques est Alice mais son apparence est modifiée.

Effets de bords

Présentation

- Considérons le programme

```
1        Alice ← description d'Alice;
2        coiffeur(); manucure(); tatoo();
```

  Alice est une variable *globale*.

- On peut imaginer que les procédures `coiffeur(); manucure(); tatoo();` s'appliquent à `Alice`. Dans un paradigme impératif, la coiffure, les ongles et la peau d'Alice ont été modifiés. On dit qu'elle a subi des *effets de bords* (fait de modifier localement une variable globale). La personne qui sort de ces 3 boutiques est Alice mais son apparence est modifiée.
- Dans un paradigme fonctionnel, on ne peut pas modifier `Alice`. La personne qui sort de chez le coiffeur est un clone qui ressemble exactement à Alice mais avec une nouvelle coupe.

Effets de bords

Pas d'effet de bords en programmation fonctionnelle

- Les effets de bords compliquent la tâche du développeur dans sa recherche de bugs : il est fréquent d'oublier qu'une procédure a modifié une variable globale ce qui fait qu'il devient difficile de prévoir à quoi ressemble `Alice` après toute ces procédures.

Effets de bords

Pas d'effet de bords en programmation fonctionnelle

- Les effets de bords compliquent la tâche du développeur dans sa recherche de bugs : il est fréquent d'oublier qu'une procédure a modifié une variable globale ce qui fait qu'il devient difficile de prévoir à quoi ressemble `Alice` après toute ces procédures.
- Cela n'arrive jamais en paradigme fonctionnel simplement parce qu'Alice n'est pas modifiable !

Récursivité

- Les langages fonctionnels évitent les boucles (car elles utilisent des variables mutables) et font largement appel à la *récursivité* (fait d'inclure l'appel d'une fonction dans sa définition) (voir transparent suivant)

Récursivité

- Les langages fonctionnels évitent les boucles (car elles utilisent des variables mutables) et font largement appel à la *récursivité* (fait d'inclure l'appel d'une fonction dans sa définition) (voir transparent suivant)
- Cela permet notamment d'éviter de stocker des résultats intermédiaires dans des tableaux (les tableaux sont des structures *mutables* à éviter dans un paradigme fonctionnel).

Récursivité

- Les langages fonctionnels évitent les boucles (car elles utilisent des variables mutables) et font largement appel à la *récursivité* (fait d'inclure l'appel d'une fonction dans sa définition) (voir transparent suivant)
- Cela permet notamment d'éviter de stocker des résultats intermédiaires dans des tableaux (les tableaux sont des structures *mutables* à éviter dans un paradigme fonctionnel).
- Une fonction récursive bien écrite n'est pas moins rapide qu'une fonction impérative et, surtout, elle est plus facile à *analyser* (terminaison, correction, complexité).

Récursivité

- Les langages fonctionnels évitent les boucles (car elles utilisent des variables mutables) et font largement appel à la *récursivité* (fait d'inclure l'appel d'une fonction dans sa définition) (voir transparent suivant)
- Cela permet notamment d'éviter de stocker des résultats intermédiaires dans des tableaux (les tableaux sont des structures *mutables* à éviter dans un paradigme fonctionnel).
- Une fonction récursive bien écrite n'est pas moins rapide qu'une fonction impérative et, surtout, elle est plus facile à *analyser* (terminaison, correction, complexité).
- Mal écrire une récursion : consommation de mémoire. les développeurs cherchent à écrire des fonctions *en récursion terminale* (à suivre)

Paradigme fonctionnel

Récursion

- Version impérative

```
1        fonction factorielle (n:entier) sortie n! :
2          debut
3          var entière r ← 1 /*résultat*/
4          pour i allant de 2 à n faire
5              r← r × i
6          fin_faire
7          renvoyer r
```

Paradigme fonctionnel

Récursion

- Version impérative

```
1   fonction factorielle (n:entier) sortie n! :
2     debut
3     var entière r ← 1 /*résultat*/
4     pour i allant de 2 à n faire
5         r← r × i
6     fin_faire
7     renvoyer r
```

- Version récursive non optimisée

```
1   fonction factorielle (n:entier) sortie n! :
2     debut
3       si n ≤ 1 alors renvoyer n
4       sinon renvoyer n × factorielle (n−1)
5       fin_si
6     fin
```

Il faut garder en mémoire les différentes valeurs de n

Paradigme fonctionnel

Récursion

- Version en récursion terminale

```
1        fonction factorielle (n:entier) sortie n! :
2          debut
3            fonction aux (r,i)  /*fonction auxiliaire interne*/
4              entree : r entier /*résultat intermédiaire entier*/
5                       i entier /*facteur courant entier*/
6              sortie : produit de i! par r
7              {
8              si i vaut 0 ou 1 alors renvoyer r
9              sinon renvoyer aux (r × i, i−1)
10             }
11           renvoyer aux (n, n−1);
12         fin
```

Paradigme fonctionnel

Récursion

- Version en récursion terminale

```
1   fonction factorielle (n:entier) sortie n! :
2     debut
3       fonction aux (r,i)  /*fonction auxiliaire interne*/
4         entree : r entier /*résultat intermédiaire entier*/
5                  i entier /*facteur courant entier*/
6         sortie : produit de i! par r
7         {
8         si i vaut 0 ou 1 alors renvoyer r
9         sinon renvoyer aux (r × i, i−1)
10        }
11      renvoyer aux (n, n−1);
12    fin
```

- Plus besoin de garder en mémoire les différentes valeurs de n.

Paradigme impératif

Résumé

- séquence ;

Paradigme impératif

Résumé

- séquence ;
- affectations (et réaffectation) ;

Paradigme impératif

Résumé

- séquence ;
- affectations (et réaffectation) ;
- instructions conditionnelles ;

Paradigme impératif

Résumé

- séquence ;
- affectations (et réaffectation) ;
- instructions conditionnelles ;
- boucles ;

Paradigme impératif

Résumé

- séquence ;
- affectations (et réaffectation) ;
- instructions conditionnelles ;
- boucles ;
- effets de bords possibles

Paradigme fonctionnel

Résumé

- séquence ;

Paradigme fonctionnel

Résumé

- séquence ;
- affectations ;

Paradigme fonctionnel

Résumé

- séquence ;
- affectations ;
- instructions conditionnelles ;

Paradigme fonctionnel

Résumé

- séquence ;
- affectations ;
- instructions conditionnelles ;
- récursion (terminale si possible) ;

Paradigme fonctionnel

Résumé

- séquence ;
- affectations ;
- instructions conditionnelles ;
- récursion (terminale si possible) ;
- pas d'effets de bords

Langages fonctionnels impurs

- On dit que OCaml est un *langage fonctionnel impur* car il contient des traits impératifs.

Langages fonctionnels impurs

- On dit que OCaml est un *langage fonctionnel impur* car il contient des traits impératifs.
- Par exemple, en OCaml on a des *listes* (au sens usuel du terme en informatique) et des *tableaux* (idem, au sens usuel).

Langages fonctionnels impurs

- On dit que OCaml est un *langage fonctionnel impur* car il contient des traits impératifs.
- Par exemple, en OCaml on a des *listes* (au sens usuel du terme en informatique) et des *tableaux* (idem, au sens usuel).
- Si je veux modifier un élément d'une liste, je suis obligé de recopier la liste intégralement à l'exception de l'item à changer. Il n'y a donc pas d'effet de bord avec les listes.

Langages fonctionnels impurs

- On dit que OCaml est un *langage fonctionnel impur* car il contient des traits impératifs.
- Par exemple, en OCaml on a des *listes* (au sens usuel du terme en informatique) et des *tableaux* (idem, au sens usuel).
- Si je veux modifier un élément d'une liste, je suis obligé de recopier la liste intégralement à l'exception de l'item à changer. Il n'y a donc pas d'effet de bord avec les listes.
- En revanche je peux modifier un élément du tableau sans être contraint de recopier tout le reste des items (il y a alors possibilité d'effet de bord).

Présentation à effectuer au moment du cours sur les BDD

Programmation logique

Partie à aborder au moment du cours sur les BDD

- Elle consiste à exprimer les problèmes et les algorithmes sous forme de *prédicats* ou formules logiques (comme en Prolog). Approche plus souple que donner une succession d'instructions à exécuter.

Programmation logique

Partie à aborder au moment du cours sur les BDD

- Elle consiste à exprimer les problèmes et les algorithmes sous forme de *prédicats* ou formules logiques (comme en Prolog). Approche plus souple que donner une succession d'instructions à exécuter.
- En MPI, le paradigme logique est lié aux bases de données. On peut consulter à ce propos le cours de Philippe Rigaux pour en savoir plus.

Programmation logique

Partie à aborder au moment du cours sur les BDD

- Elle consiste à exprimer les problèmes et les algorithmes sous forme de *prédicats* ou formules logiques (comme en Prolog). Approche plus souple que donner une succession d'instructions à exécuter.
- En MPI, le paradigme logique est lié aux bases de données. On peut consulter à ce propos le cours de Philippe Rigaux pour en savoir plus.
- Le principe du paradigme logique en `SQL` est de décrire les propriétés du point d'arrivée (le résultat) en fonction de celles du point de départ (les données de la base). On est donc bien dans une démarche déclarative.

Programmation logique

Partie à aborder au moment du cours sur les BDD

- Elle consiste à exprimer les problèmes et les algorithmes sous forme de *prédicats* ou formules logiques (comme en Prolog). Approche plus souple que donner une succession d'instructions à exécuter.
- En MPI, le paradigme logique est lié aux bases de données. On peut consulter à ce propos le cours de Philippe Rigaux pour en savoir plus.
- Le principe du paradigme logique en `SQL` est de décrire les propriétés du point d'arrivée (le résultat) en fonction de celles du point de départ (les données de la base). On est donc bien dans une démarche déclarative.
- La façon dont on passe du point de départ à l'arrivée n'est pas décrite (c'est le problème du *Système de Gestion des Bases de Données* (SGBD). Ainsi, on exprime ce qu'on *veut* et ce qu'on *a* (des tables de la base de données) et le SGBD trouve tout seul le *comment*.

Programmation logique

- Dans sa version initiale, le langage SQL étudié en MPI est un langage de programmation logique.

Programmation logique

- Dans sa version initiale, le langage SQL étudié en MPI est un langage de programmation logique.
- La base de données est un ensemble de *relations* au sens mathématiques (sous-ensemble de produits cartésiens).

Programmation logique

- Dans sa version initiale, le langage SQL étudié en MPI est un langage de programmation logique.
- La base de données est un ensemble de *relations* au sens mathématiques (sous-ensemble de produits cartésiens).
- Les *requêtes* SQl (la question posée au SGBD) sont écrites dans une syntaxe particulière mais s'interprètent naturellement comme des formules logiques.