DIU-EIL Bloc 4
Langages de programmation: un panorama

Romuald THION, Emmanuel COQUERY

https://forge.univ-lyon1.fr/diu-eil/bloc4

21 mai 2020

Plan

Généralités sur les langages de programmation

Un langage de programmation est défini par

- une syntaxe
  - en générale textuelle, mais il en existe des graphiques et des formelles
- une sémantique expliquant l'exécution d'un programme ;

Pour son évaluation il peut être

- interprété :
  - un interpréteur va se charger de son exécution
- compilé :
  - un compilateur va se charger de sa traduction, soit en langage machine, soit dans un autre langage interprété (ex : dans le cas des machines virtuelles)

Dans tous les cas, des vérifications vont être effectuées pour minimiser le nombre d'erreurs à l'exécution (syntaxe, définitions, typage, etc.)

Généralités sur les langages de programmation

Un langage de programmation est défini par

- une syntaxe
  - en générale textuelle, mais il en existe des graphiques et des formelles
- une sémantique expliquant l'exécution d'un programme ;

Pour son évaluation il peut être

- interprété :
  - un interpréteur va se charger de son exécution
- compilé :
  - un compilateur va se charger de sa traduction, soit en langage machine, soit dans un autre langage interprété (ex : dans le cas des machines virtuelles)

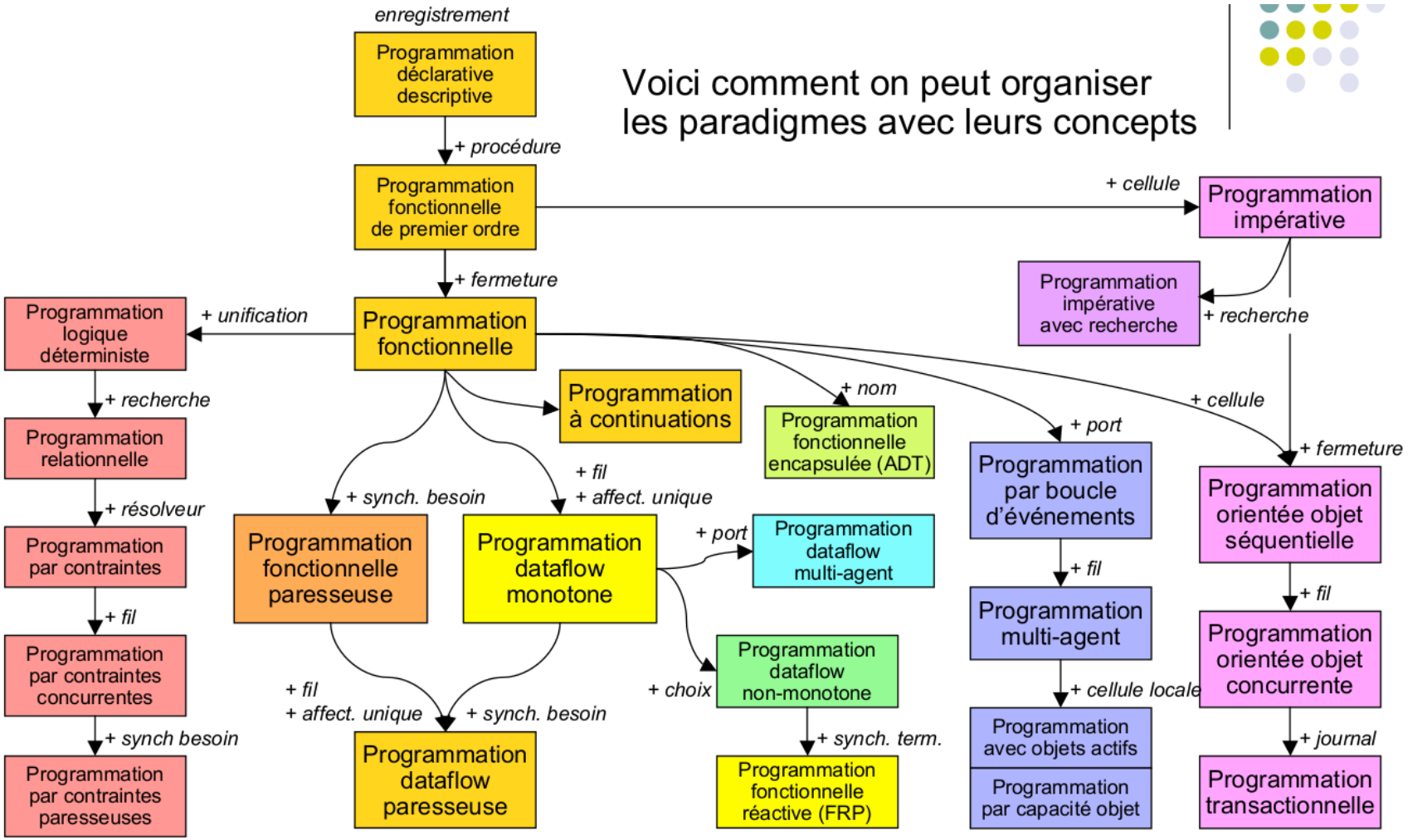
Dans tous les cas, des vérifications vont être effectuées pour minimiser le nombre d'erreurs à l'exécution (syntaxe, définitions, typage, etc.)

Paradigmes de programmation

Paradigme : représentation du monde ; manière de voir les choses ; modèle cohérent de pensée, de vision du monde qui repose sur une base définie, sur un système de valeurs.

- Façon d'exprimer le résultat / l'effet désiré pour un programme :
  - par une liste d'instructions
  - par des expressions à évaluer
  - comme devant respecter une certaine spécification
  - par transformation successives
  - ...
- Le paradigme a une influence forte sur la manière de concevoir un programme
  - il constitue un des critères principaux pour le choix d'un langage

Quels paradigmes de programmation ?



Peter Van Roy

Introduction

Quelques remarques générales

- Les langages sont très rarement « *purs* », ils intègrent plusieurs paradigmes.
- La taxonomie n'est pas consensuelle, voir par exemple Wikipedia

Le cas de Python `https://docs.python.org/3/`

- Impératif : The Python Language Reference : 7. Simple statements et 8. Compound statements
- Fonctionnel : Functional Programming HOWTO
- Objet : The Python Tutorial : 9. Classes

Généralités

- Principe :
  - Un programme est un ensemble d'instructions (actions à réaliser) et de déclarations (fonctions, procédures, etc).
  - L'exécution du programme correspond à celle des instructions.
- Une variable correspond à un emplacement mémoire dans lequel on range / on lit une valeur.
- Les calculs sont réalisés à travers ces instructions
  - même s'il existe des notions d'expression et de fonctions

Mots-clefs : affectation, mémoire, instruction (*statement*), boucle, mutabilité, effets de bord (*side effects*)

Exemples : C (paradigmatique), bash, Pascal, Python, Java, ASM, ...

Usages typiques

- Lorsque le programme à réaliser s'exprime sous forme d'actions à réaliser
  - Les actions reflètent les changements dans l'environnement (mémoire) du programme
- Couramment utilisé dans le cadre de la *programmation système*
  - Pour une gestion manuelle, potentiellement fine, de l'allocation mémoire impératifs

Un exemple filé : le PGCD

L'algorithme d'Euclide pour le calcul du Plus Grand Commun Diviseur (PGCD, *gcd* en anglais), voir Wikipedia et rosettacode.org

```
function gcd(a, b)
    while b != 0
        t := b
        b := a mod b
        a := t
    return a

function gcd(a, b)
    gcd(a, 0) = a
    gcd(a, b) = gcd(b , a mod b)

function gcd(a, b)
    if b = 0
        return a
    else
        return gcd(b, a mod b)
```

PGCD en Python

```python
# traduction directe du pseudo-code
def pgcd_imperatif(a, b):
    while (b != 0):
        t = b
        b = a%b
        a = t
    return a

# version "Pythoniste", avec l'idiome du swap
def pgcd_pythoniste(a, b):
    while b:
        a, b = b, a%b
    return a

# version "des développeurs"
import gcd from math
```

./code/pgcd.py

PGCD en C

```c
/* gcc -Wall pgcd.c && ./a.out */
#include <stdio.h>

int pgcd(int a, int b) {
  int t;
  while (b != 0){
    t = b;
    b = a % b;
    a = t;
  }
  return a;
}

int main() {
  printf("pgcd de %d et %d: %d\n", 84, 18, pgcd(84,18));
}
```

./code/pgcd.c

Généralités

- Principe :
  - Un programme fonctionnel est un ensemble de déclarations (fonctions, etc) suivi d'une expression ;
  - l'exécution du programme consiste à évaluer l'expression.
  - Pas d'affectation [1] : les variables sont des résultats intermédiaires dans le calcul.
  - Gestion de la mémoire automatisée de l'allocation mémoire
- Ordre supérieur :
  - les fonctions sont des « citoyennes de première classe », manipulables au même titre que les autres valeurs :
    - une variable peut prendre une fonction pour valeur,
    - une fonction peut prendre en argument une autre fonction,
    - une fonction peut renvoyer une fonction comme résultat [2]

1. $v = expr$ définit la variable v, *une et une seule fois*
2. quitte à la « fabriquer » au passage avec une λ expression par exemple

Usages typiques

Exemples de logiciels écrit en Haskell : Pandoc.

Mots-clefs : déclaratif, expressions, immutabilité, ordre supérieur, λ-calcul

Exemples : Haskell (paradigmatique), Lisp, Scheme, OCaml, XQuery...

- Manipulations de haut niveau
- Calculs algébriques, abstraits
- Analyse de structures
- Transformations complexes
- Écriture de compilateurs

PGCD récursif en Python

```python
# intuition
# on essaie de programmer sans affectation

def pgcd_rec(a, b):
  if b == 0 :
    return a
  else :
    return pgcd_rec(b, a%b)
```

./code/pgcd_rec.py

PGCD récursif en C

```c
#include <stdio.h>

int pgcd(int a, int b)
{
    if (b == 0)
        return a;
    return pgcd(b, a%b);
}

int main() {
  printf("pgcd de %d et %d: %d\n", 84, 18, pgcd(84,18));
}
```

./code/pgcd_rec.c

PGCD en Haskell

```haskell
-- avec filtrage de motif : la définition au
-- plus proche de celle mathématique
pgcd :: Integer -> Integer -> Integer
pgcd a 0 = a
pgcd a b = pgcd b (a `mod` b)

a, b :: Integer
a = 132
b = 105

main :: IO ()
main = putStrLn $ "le pgcd de " ++ show a ++ " et " ++ show  b
    ++ " est "  ++ show  (gcd a b) ++ "\n"
```

./code/pgcd.hs

Ordre supérieur en Python – 1

```
ma_liste = [1, 2, 0, 4, 3]
print(ma_liste)

# on peut affecter une fonction a une variable
def carre(x):
  return x*x
g = carre

# la fct standard map prend une fct en parametre et une liste
# elle applique cette fonction à tous les éléments
# pour produire une nouvelle liste
# map(f, [a, b, c, ...]) = [f(a), f(b), f(c), ...]
# https://docs.python.org/3/library/functions.html#map
mes_carres = map(g, ma_liste)
print(list(mes_carres))

# on peut passer une lambda expression
mes_carres_lambda = map(lambda x: x*x, ma_liste)
print(list(mes_carres_lambda))
```

Ordre supérieur en Python – 2

```python
# on peut créer des fonctions et les retourner
def compose(f, g):
    return (lambda x: f(g(x)))

f = compose(lambda x: x + 1, lambda x: x * x)
print(f(3))

# on peut aussi définir la composition uniquement
# avec des expressions fonctionnelles (lambda)
comp = lambda f, g : lambda x : f(g(x))

g = comp(lambda x: x + 1, lambda x: x * x)
print(g(3))
```

./code/fun_prog2.py

Ordre supérieur en Python – 3

```
# une variante où on a "curryfiée" la composition :
#   * comp n'attend pas 2 paramètres mais un seul (f)
#   * comp(f) retourne une fonction qui attend un paramètre (g)
#   * comp(f)(g) retourne enfin une fonction qui attend un
    paramètre (x)
comp = lambda f : lambda g : lambda x : f(g(x))

# donc on écrit comp(f)(g), noter les parenthèses supplé
    mentaires
g = comp(lambda x: x + 1)(lambda x: x * x)
print(g(3))
```

./code/fun_prog3.py

Généralités

- Objet :
  - ensemble de champs (ou de variables, ou de valeurs, etc) auquel on associe des méthodes ;
  - une méthode est un comportement associé à l'objet, typiquement une fonction ou une procédure ;
  - une méthode peut accéder aux champs de l'objet auquel elle est rattachée : c'est son contexte d'exécution
- Un langage est dit Orienté Objet (OO) lorsqu'il intègre une notion d'objet
  - ce langage peut par ailleurs être impératif ou fonctionnel

Classes et héritage

- Classe :
  - Patron permettant de fabriquer des objets
    - notion présente dans de nombreux langages OO
- Héritage :
  - une classe *fille* hérite d'une classe *mère*
  - les méthodes et les champs de classe mère sont automatiquement définis dans la classe fille
  - la classe fille *peut* avoir des champs et des méthodes supplémentaires
  - la classe fille *peut* changer la définition de certaines méthodes provenant de la classe mère
  - toute instance de la classe fille peut être utilisée là où une instance de la classe mère est attendue (sous-typage)

Usages :

- Avantageux dans le cadre du développement de gros logiciels :
  - réutilisation du code
  - découpage et isolation du code (encapsulation)
  - donne lieu à des méthodes de conception dédiées (UML), en particulier dans le développement des systèmes d'information

Mots-clefs : classes ou prototype, encapsulation, héritage

Exemples : C++ et Java (paradigmatiques), Python, OCaml, ...

Exemples de classes en Python

```
class Personne:
    def __init__(self, un_nom, un_prenom):
        self.nom = un_nom
        self.prenom = un_prenom
    def affiche(self):
        print(self.prenom + " " + self.nom)

class Eleve(Personne):
    def __init__(self, un_nom, un_prenom, un_groupe):
        Personne.__init__(self, un_nom, un_prenom)
        self.groupe = un_groupe
    def affiche(self):
        print (self.prenom + " " + self.nom + ": " + self.
            groupe)
```

./code/classes_personnes.py

Exemples de classes en Python - 2

```
class Prof(Personne):
    def __init__(self, un_nom, un_prenom, une_matiere):
        Personne.__init__(self,un_nom,un_prenom)
        self.matiere = une_matiere
    def affiche(self):
        print (self.prenom + " " + self.nom + " (" + self.
            matiere +")")

un_gars  = Personne('Thion', 'Romuald')
un_eleve = Eleve('Hunting', 'Will', '1ereA')
un_prof  = Prof('Maguire','Sean.','Mathematiques')

for p in [un_gars, un_eleve, un_prof]:
    p.affiche()
```

./code/classes_personnes.py