NSI premiere programmation

Diversité et unité des langages de programmation

DIU

2019/06

Exemples sur repl

ici

Diversité et unité des langages de programmation

Une rapide comparaison de différents langages de programmation autour d'un même algorithme - Python - Javascript - Java - C - Scheme - Prolog[2]

Les codes sont fournis en respectant autant que possible un même squelette de base. On fait donc apparaître ici plus les similitudes des structures que les spécificités des langages, même si certains points importants apparaissent déjà. Parmi les points communs, on peut donc retrouver les fonctions et leurs paramètres, la déclaration de variables, les structures itératives et conditionnelles, la notion de bloc d'instruction.

Le langage Prolog est ajouté pour montrer une approche très différente de la programmation (par des clauses logiques). Compte-tenu de sa spécificité, on ne compare pas ce langage avec les autres dans le paragraphe suivant.

Quelques éléments de comparaison :

- Python, Javascript et Scheme sont interprétés alors que Java et C sont compilés
- Javascript, Java et C partagent la même syntaxe sur les structures de base (syntaxe issue de C)
- Python, Javascript et Scheme sont dynamiquement typés alors que Java et C sont statiquement typés
- En java les méthodes doivent être placées dans des classes
- En C le code produit par le compilateur est du code binaire, spécifique
- En java le code produit par le compilateur est du bytecode interprétable par une machine virtuelle
- Pour les langages interprétés (Python, Javascript et Scheme) le même code peut être exécuté sur des machines d'architecture et de systèmes d'exploitation différents (comme le bytecode Java)
- l'écriture de code respecte globalement les mêmes règles dans tous les langages, indentation, nommage des variables. Seul Python impose l'indentation comme élément de syntaxe.

Le pseudo code

Voici l'algorithme qui va être décliné dans différent langage. Il est itératif. Il n'est pas optimisé, mais le but est de montrer plus d'aspect des langages (par exemple on aurait pu avoir un test `n < 2` ou `n = 0 ou 1`, ou n'avoir qu'un seul `return`, etc.).

```
factorielle (n) =
   si n = 0
      alors le résultat est 1
   sinon si n = 1
      alors le résultat est 0
   sinon
      on initialise une variable result à 1
      on initialise une variable i à 1
      tant que i < n
         on incrémente i
     on multiplie result par i
```

```
      le résultat est result
```

Python

Fichier factorielle.py.

```
def factorielle(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        result = 1
        i = 0
        while i < n:
            i = i + 1
            result = result * i
        return result
```

On peut utiliser ce code dans l'interprète Python après avoir évalué cette définition.

```
>>> factorielle(5)
120
```

Mais on peut aussi l'utiliser comme un script, il faut compléter ce code des lignes suivantes :

```
if __name__ == '__main__':
   import sys
   n = int(sys.argv[1])
   print( str(factorielle(n)) )
```

Le tout est sauvegardé dans un fichier `factorielle.py`, puis dans une console on exécute :

```
$ python3 factorielle.py 5
120
```

javascript

On présente ici deux versions, une avec la syntaxe "ancienne" de javascript et une avec les évolutions de syntaxe récentes.

old school

Fichier factorielle-old.js.

```
var factorielle = function(n) {
    if ( n === 0)
    return 1;
    else if ( n === 0)
    return 1;
    else {
    var result = 1;
    var i = 0;
    while (i < n) {
        i = i + 1;
        result = result * i;
    }
    return result;
    }
}
```

Le langage javascript a évolué fortement (et dans le bon sens) ces dernières années. Par exemple le mot-clé `var` est en disgrâce. On écrirait donc plutôt maintenant :

version ES6

Fichier factorielle.js.

```
const factorielle =
          n => {
                  if ( n === 0)
                    return 1;
                  else if ( n === 1)
                    return 1;
                  else {
                    let result = 1;
                    let i = 0;
                    while (i < n) {
                      i = i + 1;
                      result = result * i;
                    }
                    return result;
                  }
              }
```

Dans les deux cas on peut utiliser la console javascript fournie avec firefox par exemple (`Ctrl+Shift+K` dans le navigateur, outil également présent dans Chrome après touche `F12`). Il faut évaluer cette définition puis l'expression :

```
>> factorielle(5)
<- 120
```

Java

Fichier Math.java.

```
public class Math {
    public static int factorielle(int n) {
    if ( n == 0 )
        return 1;
        else if  ( n == 1 )
            return 1;
    else {
        int result = 1;
        int i = 0;
        while (i < n) {
        i = i + 1;
        result = result * i;
        }
        return result;
    }
    }
}
```

Pour son utilisation, il faut compléter ce code d'une "méthode `main`" placée avant la dernière accolade fermante (par exemple):

```
public static void main(String[] arg) {
   int n = Integer.parseInt(arg[0]);
   System.out.println(Math.factorielle(n));
}
```

L'ensemble doit être enregistré dans un fichier `Math.java`. Ce fichier doit être compilé pour être utilisé. La compilation (Oracle Open JDK) produit un fichier `Math.class`qui contient du *bytecode java* interprétable par une *machine virtuelle java* (JVM). Le bytecode java peut-être utilisé sans nouvelle compilation dans les différentes architectures et systèmes d'exploitation, grâce à la JVM (slogan *compile once run everywhere* de java).

```
$ javac Math.java
```

```
$ java Math 5
120
```

NB L'usage de `static` pour la méthode `factorielle`n'est pas caractéristique des méthodes en Java, mais il est légitime et pertinent ici car le résultat de `factorielle` ne dépend que de son paramètre et d'aucun objet particulier.

C

Fichier factorielle.c.

```
int factorielle ( int n ) {
  if ( n == 0 )
    return 1;
  else if (n == 1 )
    return 1;
  else {
    int result = 1;
    int i = 0;
    while (i < n) {
      i = i + 1;
      result = result * i;
    }
    return result;
  }
}
```

Pour son utilisation, il faut compléter ce code d'une "fonction `main`" :

```
int main(int argc, char *argv[] ) {
  int n = atoi(argv[1]);
  printf("%d", factorielle(n) );
}
```

L'ensemble placé dans un même fichier `factorielle.c` doit être compilé pour être utilisé. La compilation produit (ci-dessous) le fichier `factorielle`qui contient un code binaire machine. Il est directement exécutable. Il faut cependant produire un exécutable différent pour chaque architecture de machine et chaque système d'exploitation.

```
$ gcc -o factorielle factorielle.c
$ ./factorielle 5
120
```

Scheme

Fichier factorielle.scm.

Le langage Scheme (descendant de Lisp) n'est vraiment pas fait pour de l'itération ni les affectations, le code suivant ferait donc hurler tout progammeur Scheme. Mais on respecte le schéma de pseudo-code fourni initialement.

```
(define factorielle
  (lambda(n)
    (cond ((= n 0) 1)
          ((= n 1) 1)
          (else
           (let ((result 1))
             (do ((i 1 (+ i 1)))
                 ((> i n))
               (set! result (* result i))
               )
             result)
           )
      )
    )
  )
```

NB une structure `if`, ternaire, existe aussi en Scheme, on fait une petite entorse ici au respect strict de l'algorithme donné, mais on a aussi utilisé le `elif` de Python.

Pour utiliser ce code on évalue cette définition dans l'interprète Scheme (par exemple Dr Racket) et on évalue ensuite l'expression

```
> (factorielle 5)
120
```

Prolog

Fichier factorielle.pl.

Et pour montrer qu'il y a des langages qui se distinguent fortement des autres, voici la programmation de `factorielle` en Prolog. Prolog est un langage de *Pro*grammation *Log*ique. Programmer en Prolog consiste à écrire des clauses logiques et exécuter un programme Prolog consiste à faire une preuve d'une formule logique et à trouver une instanciation (*unification*) des éventuelles variables qui permette cette preuve.

```
factorielle(0,1) :- !.
factorielle(1,1) :- !.
factorielle(N, Result) :- N > 0,
                          N_moins_1 is N-1,
                          factorielle(N_moins_1,   Fact_N_moins_1),
                          Result is N * Fact_N_moins_1.
```

Le prédicat `factorielle(N,R)` signifie `factorielle(N,R)` est vrai si *R est le résultat de factorielle(N)*. La première ligne signifie donc "*il est vrai que factorielle 0 vaut 1*. La seconde est similaire. La troisième *clause* peut se lire *Result est le la valeur de factorielle(N) si les clauses qui suivent `:-` sont toutes vérifiées*.

On peut exécuter ce programme avec l'interprète en ligne de swi-prolog.

Pour l'exécuter on demande la preuve de `factorielle(5,Result).`, c'est-à-dire que l'on pose la question "*Est-il vrait que `factorielle(5,Result)` ?*" :

```
?- factorielle(5,Result).
Result = 120
```

Le moteur de résolution de Prolog répond, "*vrai pour l'instanciation Result=120*".