Chapitre 5 : Langages de programmation

Prof. Assma Azeroual

Faculté Polydisciplinaire Ouarzazate
SMI1 et IGE1

2019 / 2020

Sommaire I

1 Histoire

2 Transformation du code source

3 Paradigmes

4 Quelques notions principales de la programmation

Summary I

1 Histoire
- Évolution des langages informatiques
- Hello world !

2 Transformation du code source

3 Paradigmes

4 Quelques notions principales de la programmation

Histoire I

La programmation consiste à créer une séquence d'instructions pour un ordinateur afin qu'il puisse résoudre un problème ou exécuter une tâche.
À partir des années 50 les premiers langages de programmation modernes commençaient à apparaittre. Voici les créateurs des langages les plus utilisés :

- John Backus, inventeur de Fortran (1957)
- John McCarthy, inventeur de LISP (1959)
- Grace Hopper, surnommée « la mère du langage COBOL » (1960)
- John George Kemeny, concepteur du BASIC (1964)
- Dennis Ritchie et Ken Thompson, inventeurs du langage C (1972)

Histoire II

- Niklaus Wirth inventeur de Pascal (1970) et Modula-2
- Bjarne Stroustrup, développeur de C++ (1983)
- Guido van Rossum, créateur de Python (1991)
- James Gosling et Patrick Naughton, créateurs de Java (1995).
-

Il existe des centaines de langages de programmation, certains disent des milliers. Les plus populaires en 2018 sont C, C++, C#(C sharp), Objective-C, Java, Python, Ruby, Visual Basic, PHP, Javascript, Delphi, SQL, Perl,..

Première génération

On distingue aujourd'hui cinq générations de langages. La première génération est le langage machine, ou code machine. Il est composé d'instructions et de données à traiter codées en binaire. C'est le seul langage qu'un ordinateur peut traiter directement. Voici à quoi peut ressembler un programme en langage machine :

A1 01 10 03 06 01 12 A3 01 14

Il s'agit de la représentation hexadécimale d'un programme permettant d'additionner les valeurs de deux cases mémoire et de stocker le résultat dans une troisième case. On voit immédiatement la difficulté d'un tel langage...

Deuxième génération

La deuxième génération est le langage assembleur : le code devient lisible et compréhensible par un plus grand nombre d'initiés. Il existe en fait un langage assembleur par type de processeur. Le programme précédent écrit en assembleur donnerait ceci :

MOV AX, [0110]
ADD AX, [0112]
MOV [0114], AX

Il reste utilisé dans le cadre d'optimisations, mais a été supplanté en popularité par les langages plus accessibles de troisième génération.

La troisième génération

La troisième génération utilise une syntaxe proche de l'anglais. Proposés autour de 1960, ces langages ont permis un gain énorme en lisibilité et en productivité. Ils ne dépendent plus du processeur, comme c'était le cas des générations précédentes, mais d'un compilateur spécifique du processeur. L'idée de portabilité des programmes était lancée.

La troisième génération

La plupart des langages de programmation actuels sont de troisième génération. On trouve dans cette catégorie tous les grands langages : Ada, Basic, Cobol, Fortran, C, C++, Java, Perl, Pascal, Python, ... Cette génération couvre d'ailleurs tant de langages qu'elle est souvent subdivisée en catégories, selon le paradigme particulier des langages.

La quatrième génération

Les langages de quatrième génération, abrégés L4G, souvent associée à des bases de données, se situent un niveau au-dessus, en intégrant la gestion de l'interface utilisateur et en proposant un langage moins technique, plus proche de la syntaxe naturelle.
Ils sont conçus pour un travail spécifique : gestion de base de données (Microsoft Access, SQL), production graphique (Postscript), création d'interface (4D).

La cinquième génération I

La cinquième génération de langages sont des langages destinés à résoudre des problèmes à l'aide de contraintes, et non d'algorithmes écrits. Ces langages reposent beaucoup sur la logique et sont particulièrement utilisés en intelligence artificielle. Parmi les plus connus, on trouve Prolog, dont voici un exemple :

frère_ou_soeur(X,Y) :- parent(Z,X), parent(Z,Y), X Ȳ.
parent(X,Y) :- père(X,Y).
parent(X,Y) :- mère(X,Y).
mère(trude, sally).
père(tom, sally).
père(tom, erica).
père(mike, tom).

La cinquième génération II

Il en résulte que la demande suivante est évaluée comme vraie :

?- frère_ou_soeur(sally, erica).
oui.

Ce qui signifie que Sally et Erica sont sœurs. En effet, Sally et Erica ont le même père (Tom).

Hello world !

C’est dans un memorandum interne de Brian Kernighan, Programming in C : A tutorial, écrit en 1974 dans les laboratoires Bell, que l’on trouve la première version d’un mini-programme affichant à l’écran « Hello World ! ». Voici comment cela s’écrit dans divers langages :

Hello world ! : Ada

```
with Ada.Text_IO ;
use Ada.Text_IO ;
procedure Bonjour is
begin – Bonjour
Put("Hello world !") ;
end Bonjour ;
```

Hello world ! : Assembleur X86 sous DOS

```
cseg segment
assume cs :cseg, ds :cseg
org 100h
main proc
jmp debut
mess db 'Hello world !$'
debut :
mov dx, offset mess
mov ah, 9
int 21h
ret
main endp
cseg ends
end main
```

Hello world ! : BASIC

```
10 PRINT "Hello world !"
20 END
```

Hello world ! : C

```
#include <stdio.h>
int main()/* ou int argc, char *argv[ ] */
{
printf("Hello world ! \n") ;
return 0 ;
}
```

Hello world ! : C++

```
#include <iostream>
int main()
{
std : :cout « "Hello world !" « std : :endl ;
return 0 ;
}
```

Hello world ! : FORTRAN 77

```
PROGRAM BONJOUR
WRITE (*,*) 'Hello world !'
END
```

Hello world ! : Java

```
public class HelloWorld {
public static void main(String[] args) {
System.out.println("Hello world !") ;
}
}
```

Python 3

```
print("Hello world !")
```

Summary I

1 Histoire

2 Transformation du code source
- Compilation
- Interprétation
- Avantages et inconvénients

3 Paradigmes

4 Quelques notions principales de la programmation

Transformation du code source

Le code source n'est (presque) jamais utilisable tel quel. Il est généralement écrit dans un langage « de haut niveau », compréhensible pour l'homme, mais pas pour la machine. Il existe deux stratégies de traduction, ces deux stratégies étant parfois disponibles au sein du même langage (1).

Transformation du code source

- Le langage traduit les instructions au fur et à mesure qu'elles se présentent. Cela s'appelle la compilation à la volée, ou l'interprétation.
- Le langage commence par traduire l'ensemble du programme en langage machine, constituant ainsi un deuxième programme (un deuxième fichier) distinct physiquement et logiquement du premier. Ensuite, et ensuite seulement, il exécute ce second programme. Cela s'appelle la compilation.

Transformation du code source

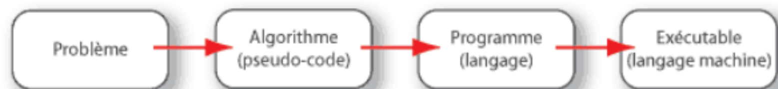

FIGURE 1 – Transformation du code source

Compilation

•Certains langages sont "compilés". En toute généralité, la compilation est l'opération qui consiste à transformer un langage source en un langage cible. Dans le cas d'un programme, le compilateur va transformer tout le texte représentant le code source du programme, en code compréhensible pour la machine, appelé code machine.
•Dans le cas de langages compilés, ce qui est exécuté est le résultat de la compilation. Une fois effectuée, l'exécutable obtenu peut être utilisé sans le code source.

Interprétation

•D'autres langages ne nécessitent pas de phase spéciale de compilation. La méthode employée pour exécuter le programme est alors différente. Le programme entier n'est jamais compilé. Chaque ligne de code est compilée "en temps réel" par un programme. On dit de ce programme qu'il interprète le code source. Par exemple, Python est un langage interprété.

•Cependant, ce serait faux de dire que la compilation n'intervient pas. L'interprète produit le code machine, au fur et à mesure de l'exécution du programme, en compilant chaque ligne du code source.

Avantages et inconvénients

•Les avantages généralement retenus pour l'utilisation de langages "compilés", est qu'ils sont plus rapides à l'exécution que des langages interprétés, car l'interprète doit être lancé à chaque exécution du programme, ce qui mobilise systématiquement les ressources.
•Les langages interprétés offrent en revanche une certaine portabilité, ainsi qu'une facilité pour l'écriture du code. En effet, il n'est pas nécessaire de passer par la phase de compilation pour tester le code source.

Summary I

1 Histoire

2 Transformation du code source

3 Paradigmes
- Programmation impérative
- Programmation structurée (ou procédurale)
- Programmation orientée objet
- Programmation fonctionnelle

4 Quelques notions principales de la programmation

Paradigmes

En informatique, un paradigme est une façon de programmer, un modèle qui oriente notre manière de penser pour formuler et résoudre un problème.
Certains langages sont conçus pour supporter un paradigme en particulier (Java supporte la programmation orientée objet), alors que d'autres supportent des paradigmes multiples (à l'image de C++, Python).

Programmation impérative

C'est le paradigme le plus ancien. Les recettes de cuisine et les itinéraires routiers sont deux exemples familiers qui s'apparentent à de la programmation impérative. La grande majorité des langages de programmation sont impératifs. Les opérations sont décrites en termes de séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme.

Programmation structurée

La programmation procédurale est un paradigme de programmation basé sur le concept d'appel procédural (Exemple : C++, Python). Une procédure, aussi appelée routine, sous-routine, méthode ou fonction (à ne pas confondre avec les fonctions de la programmation fonctionnelle reposant sur des fonctions mathématiques) contient simplement une série d'étapes à réaliser. N'importe quelle procédure peut être appelée à n'importe quelle étape de l'exécution du programme, incluant d'autres procédures voire la procédure elle-même (récursivité).

Exemple

Dans certains langages anciens comme le BASIC, les lignes de programmation portent des numéros, et les lignes sont exécutées par la machine dans l’ordre de ces numéros. Dans tous ces langages, il existe une instruction de branchement qui envoie directement le programme à la ligne spécifiée (GOTO). Inversement, ce type de langage ne comporte pas d’instructions comme "Fin Tant Que", ou "Fin Si", qui ferment un bloc. Prenons l’exemple d’une structure "Si ... Alors ... Sinon" (**??** :

Exemple

Programmation Structurée	Programmation non structurée

Si condition Alors
    instructions 1
Sinon
    instructions 2
FinSi
	1000 Si condition Alors Aller En 1200
	1100 instruction 2
	1110 etc.
	1120 etc.
	1190 GOTO 1400
	1200 instruction 1
	1210 etc.
	1220 etc.
	1400 suite de l'algorithme

FIGURE 2 – Programmation structurée et non structurée

Programmation orientée objet

La programmation orientée objet (POO) ou programmation par objet, est un paradigme de programmation informatique qui consiste en la définition et l'assemblage de "briques logicielles" appelées objets. Un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre.
Quelques langages à objets : Ada, Java, Python, C++, PHP...

Programmation fonctionnelle

Le principe général de la programmation fonctionnelle est de concevoir des programmes comme des fonctions mathématiques que l'on compose entre elles. A la différence des programmes impératifs organisés en instructions produisent des effets de bords, les programmes fonctionnels sont bâtis sur des expressions dont la valeur est le résultat du programme.

Programmation fonctionnelle

Un programme fonctionnel consiste en une expression E (représentant l'algorithme et les entrées). Cette expression E est sujette à des règles de réécriture : la réduction consiste en un remplacement d'une partie de programme fonctionnel par une autre partie de programme selon une règle de réécriture bien définie. Ce processus de réduction sera répété jusqu'a l'obtention d'une expression irréductible (Exemple : MIRANDA).

Summary I

1 Histoire

2 Transformation du code source

3 Paradigmes

4 Quelques notions principales de la programmation
- L'affectation
- Les tests
- Les boucles
- Compteur

Les notions principales de la programmation

La plupart des langages de programmation ont des caractéristiques communes que nous allons passer rapidement en revue ici.

L'affectation

Une affectation est une opération qui permet d'attribuer une valeur à une variable. Pour expliquer ce qui se passe lors d'une affectation, il faut imaginer qu'il existe, dans un recoin de l'ordinateur, une case mémoire appelée x. L'instruction x=t consiste à remplir la case x avec la valeur de l'expression t. Si x contenait déjà une valeur, celle-ci est écrasée.

Incrémentation / décrémentation

Il arrive fréquemment que l'on doive incrémenter (ou décrémenter) une variable, c'est-à-dire augmenter (ou diminuer) sa valeur d'une ou plusieurs unités. Dans ce cas, on peut utiliser l'instruction x=x+1. On prend la valeur contenue dans x, on y ajoute 1, puis on remet le nouveau résultat dans la variable x. L'instruction x=x+1 est tellement fréquente qu'il existe souvent des raccourcis : x+=1 (Python) ou x++ (C).

Les tests

Un test est une instruction du genre si... alors... sinon. La suite du programme dépendra du résultat du test.
Par exemple, en Python, on aura :

```
if x>=10 :
x=x-20
else :
x=x+2
```

Les boucles

Une boucle est une structure de contrôle permettant de répéter une ou un ensemble d’instructions plusieurs fois, tant qu’une condition est satisfaite.
Par exemple, en Python 3, ce programme écrira tous les entiers de 0 à 9 :

x=0
while x<10 :
print(x)
x+=1

Compteur

Un compteur permet de réaliser une boucle associée à une variable entière qui sera incrémentée (ou décrémentée) à chaque itération.
En BASIC :
FOR i = depart TO fin instruction NEXT i
En Pascal :
for i := depart to fin do instruction ;

Informatique (presque) débranchée, Chapitre 6 Programmation et langages, Didier Müller.

https ://www.techno-science.net/definition/11446.html

http ://hameur.perso.univ-pau.fr/Cours/Para/Pgfonct.pdf