Programmation et langages

version 1.1
09/2016

SI 4 - Base de la programmation

1 PROGRAMMATION 3
ÉVOLUTION DES LANGAGES INFORMATIQUES 3
2 ALGORITHME ET PSEUDO-CODE 7
EXEMPLE DE PSEUDO-CODE 7
3 TRANSFORMATION DU CODE SOURCE 8
COMPILATION 8
INTERPRÉTATION 8
AVANTAGES, INCONVÉNIENTS 8
4 PARADIGMES 9
PROGRAMMATION IMPÉRATIVE 9
PROGRAMMATION STRUCTURÉE (OU PROCÉDURALE) 10
PROGRAMMATION ORIENTÉE OBJET 11
PROGRAMMATION FONCTIONNELLE 11
5 LES NOTIONS PRINCIPALES DE LA PROGRAMMATION 12
L'AFFECTATION 12
AFFECTATION ET COMPARAISON 12
INCRÉMENTATION/ DÉCRÉMENTATION 12
LES TESTS 12
LES BOUCLES 13
BOUCLE « TANT QUE » 14
BOUCLES IMBRIQUÉES 14
BOUCLE « JUSQU'À CE QUE » 15
COMPTEUR 15
ITÉRATEUR 16
LES SOUS-PROGRAMMES 16
PORTÉE D'UNE VARIABLE 16
PASSAGE DE PARAMÈTRES 17

1 Programmation

La programmation consiste à créer une séquence d'instructions pour un ordinateur afin qu'il puisse résoudre un problème ou exécuter une tâche.
Si les concepts et les termes de programmation et de programme sont dûs à Alan Mathison Turing, dans un article de 1936, c'est à partir des années 50 que l'on verra apparaître les premiers langages de programmation modernes.

Évolution des langages informatiques

On distingue aujourd'hui cinq générations de langages :

- La **première génération** est le langage machine, ou code machine. On parle aussi de langage natif. 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.
- 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** utilise une syntaxe plus proche du langage naturel (en pratique 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 de celui-ci. L'idée de portabilité des programmes était lancée. La plupart des langages de programmation actuels sont de troisième génération. On trouve dans cette catégorie tous les grands langages : Ada, Algol, Basic, Cobol, Eiffel, Fortran, C, C++, Java, Perl, Pascal, Python, Ruby... Cette génération couvre d'ailleurs tant de langages qu'elle est souvent subdivisée en catégories, selon le paradigme particulier des langages.
- Les langages de **quatrième génération**, ou L4G, souvent associés 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 (SQL), production graphique (Postscript), création d'interface (4D).

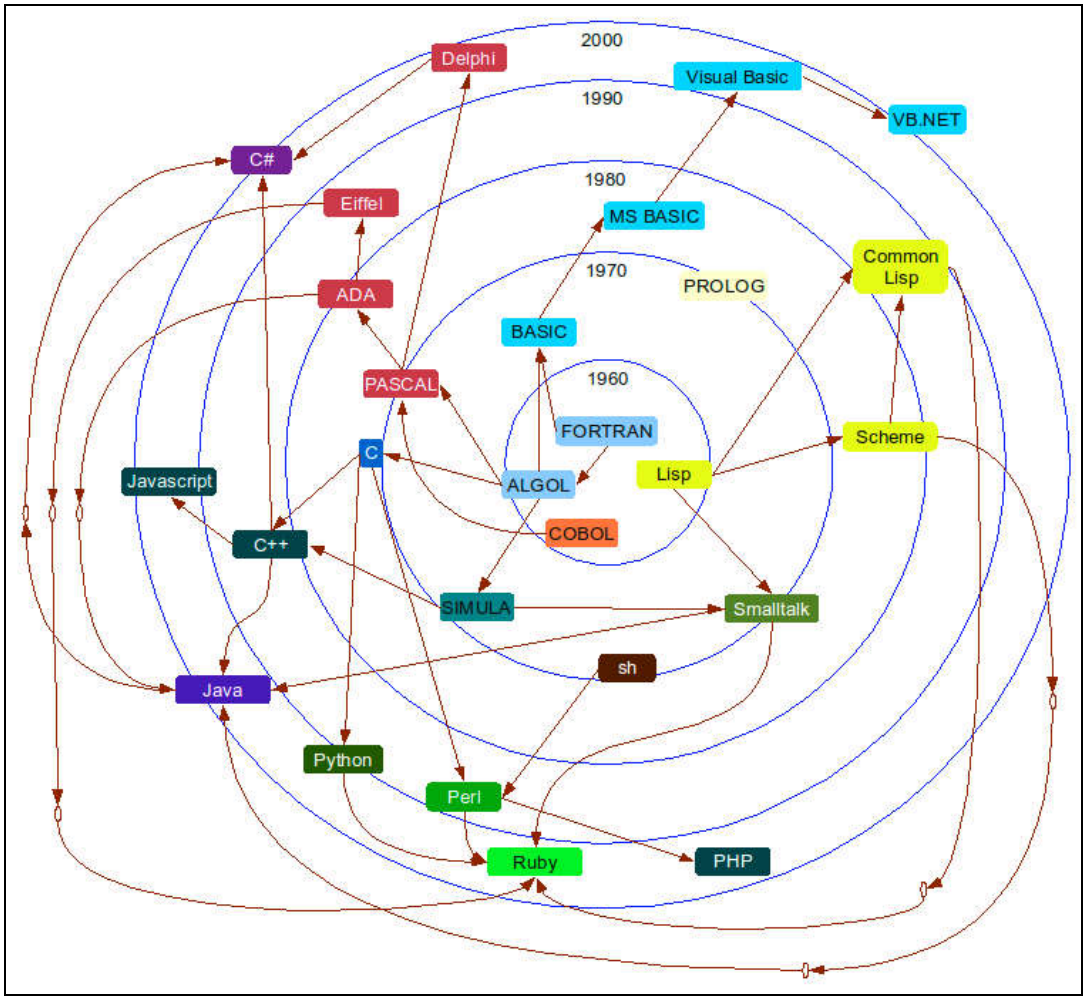
- La **cinquième génération** de langages sont des langages destinés à résoudre des problèmes à l'aide de contraintes, et non d'algorithmes écrits. Ces langages reposent beaucoup sur la logique et sont particulièrement utilisés en intelligence artificielle. Parmi les plus connus, on trouve Prolog, dont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

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

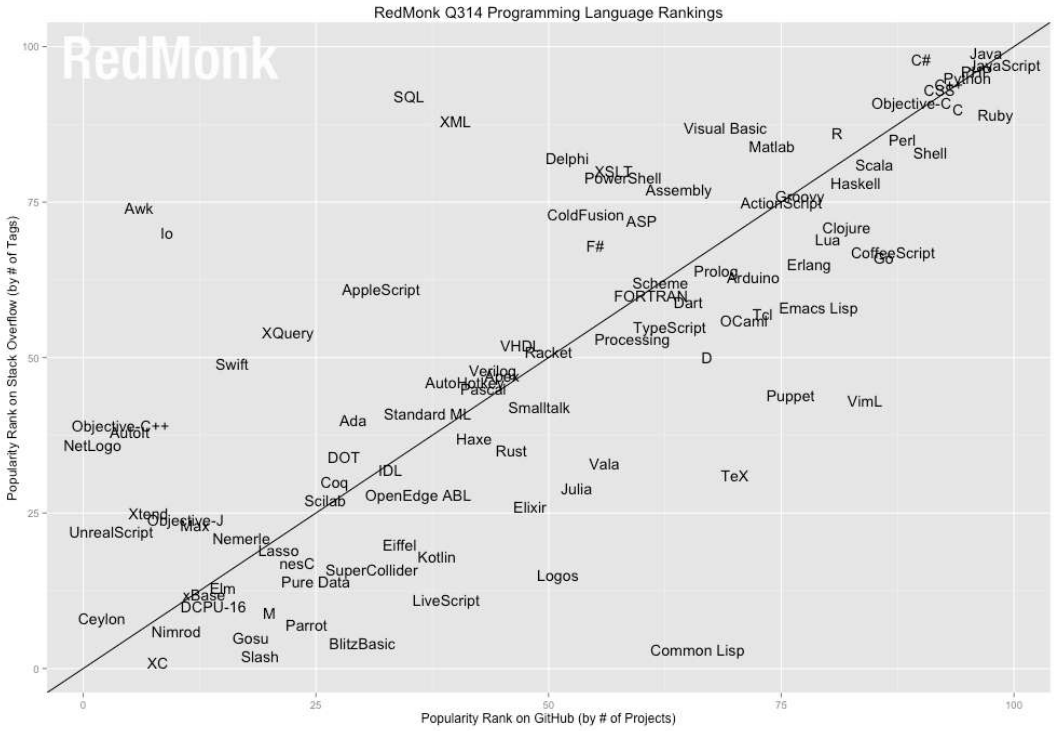
Il en résulte que la demande suivante est évaluée comme vraie :

```
?- frère_ou_soeur(sally, erica).
oui.
```

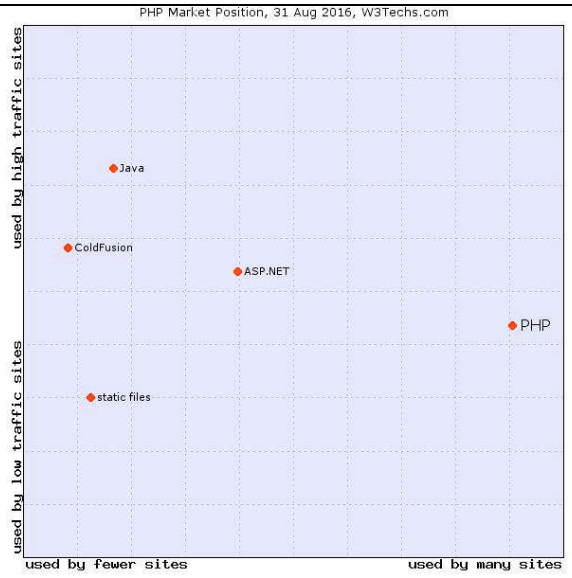
Ce qui signifie que Sally et Erica sont sœurs. En effet, Sally et Erica ont le même père (Tom).
Il existe environ 2500 langages de programmation, certains étant très généraux (C, Java), d'autres hyperspécialisés. Par exemple, RobotProg permet de déplacer un robot virtuel, Rest un langage pour la statistique, etc. Par comparaison, on recense environ 6800 langues humaines parlées et/ou écrites de par le monde.



Généalogie simplifiée des langages informatiques



Langages de programmation Web côté serveur les plus populaires (source W3Tech, 31 août 2016)	
PHP	82 %
ASP.NET	15.7 %
Java	2.8 %
Fichiers statiques	1.5 %
Coldfusion	0.7 %
Ruby	0.6 %
Perl	0.4 %
Javascript	0.3 %
Python	0.2 %
Erlang	0.1 %
(Total supérieur à 100% car un même site peut employer plusieurs langages)	



2 Algorithme et pseudo-code

Tout programme doit commencer sa vie sous forme d'algorithme, avant d'être "traduit" en un programme dans un langage de programmation précis. En programmation, le **pseudo-code** est une façon de décrire un algorithme en respectant certaines conventions, mais sans référence à un langage de programmation en particulier. L'écriture en pseudocode permet de développer une démarche structurée, en « oubliant » temporairement la syntaxe rigide d'un langage de programmation.
Vous devez garder à l'esprit certaines règles (largement vues et revues en algorithmique) :
• Le pseudo-code doit être suffisamment précis pour que quelqu'un d'autre l'ayant lu sans connaître l'algorithme soit capable de le coder.
• L'objectif est d'avoir le pseudo-code le plus simple possible, afin de le coder facilement et sans bug. Le pseudo-code d'un petit algorithme prend en général une douzaine de lignes. Pour un problème plus complexe, il faut compter jusqu'à 20 ou 25 lignes.
• Prévoyez assez de place pour faire tenir tout le pseudo-code sur une même page.
• Lorsqu'on commence à écrire le pseudo-code, on ne sait pas encore précisément ce qui va venir ensuite. Il est judicieux de laisser des lignes blanches régulièrement pour pouvoir ajouter des choses ensuite tout en gardant quelque chose de propre.
• Pour rendre compte visuellement des imbrications des boucles, indentez vers la droite le corps des boucles et des fonctions. Sur le papier, on peut ajouter de grandes barres verticales pour faire ressortir encore plus les différents blocs d'instructions.

Exemple de pseudo-code

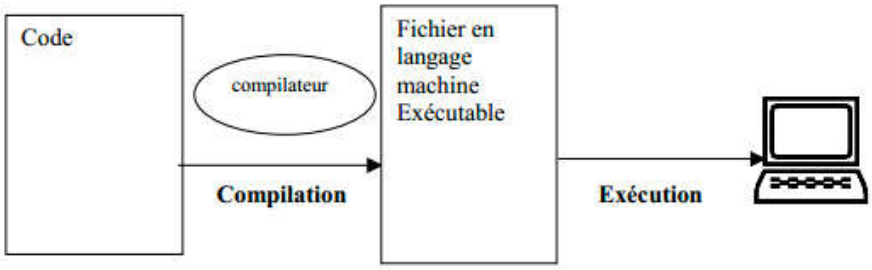
```
Algorithme : Tri d'une liste de nombres
Donnée :Tableau A de n nombres
Résultat :Tableau A de n nombres triés par ordre croissant
DEBUT
  permut := 1
  TANT QUE permut = 1 FAIRE
     permut := 0
     POUR i ALLANT DE 0 À n-2 FAIRE
        SI A[i] > A[i+1] ALORS
            echanger A[i] et A[i+1]
            permut := 1
        FIN SI
     FIN POUR
  FIN TANT QUE
FIN
```

3 Transformation du code source

Quel que soit le langage utilisé pour écrire un programme, ce dernier devra toujours être traduit en langage machine pour pouvoir être exécuté par le processeur. Celui-ci ne connaît qu'une seule langue. 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.

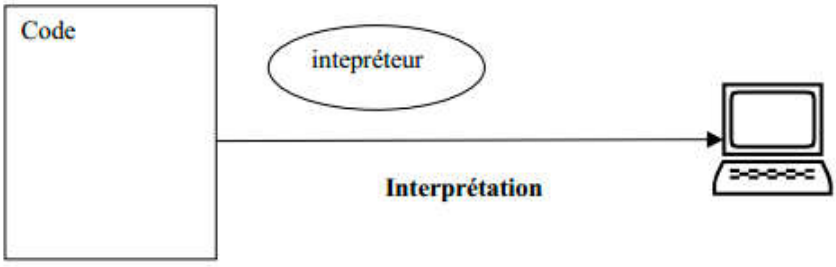
Compilation

Certains langages sont « compilés ». En toute généralité, la compilation est l'opération qui consiste à transformer un langage source en un langage cible. Le compilateur va transformer tout le texte représentant le code source du programme préalablement à l'exécution, pendant une étape appelée **compilation**, 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. C'est le cas de C/C++, C#, Pascal…. Un fichier du programme traduit en langage machine est généré (souvent, un .exe), et c'est ce fichier qui sert de base à l'exécution.



Interprétation

D'autres langages ne nécessitent pas de phase spéciale de compilation. La méthode employée est différente. Chaque ligne de code est compilée « en temps réel » par un programme nommé interpréteur. Les instructions sont traduites au fur et à mesure. Cela s'appelle compilation à la volée, ou interprétation (Basic, Javascript). Il serait faux de prétendre que la compilation n'intervient pas : l'interprètteur produit le code machine au fur et à mesure de l'exécution du programme en compilant chaque ligne du code source.



Avantages, inconvénients

Le tableau suivant résume les avantages et inconvénients des deux techniques.

Type de langage	Avantages	Inconvénient
Compilé	Rapide à l'exécution. Code source inutile pour l'exécution : on peut vendre un programme compilé sans donner le code : on ne risque pas (ou moins) de se faire copier.	Un programme est compilé dans un langage machine donné : il ne va fonctionner que sur les ordinateurs dont le processeur connaît ce langage machine. Un programme compilé pour Mac ne marchera pas sur PC. Un programme compilé n'est pas portable.
Interprété	Peut être exécuté sur n'importe quelle machine possédant un interpréteur du langage, quel que soit le type de langage machine : c'est l'interpréteur qui traduit dans le langage de la machine.	Lent car chaque instruction doit d'abord être traduite par l'interpréteur avant d'être exécutée. L'exécution est fondée sur le code source donc celui-ci doit être fourni au client qui peut s'en servir à mauvais escient

Certains langages, comme Java, emploient une compilation initiale (produisant ce que l'on nomme du byte-code), suivie d'une interprétation sur la machine cliente, garantissant ainsi une portabilité optimale.

4 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 (Smalltalk et Java supportent la programmation orientée objet, tandis que Haskell est conçu pour la programmation fonctionnelle), alors que d'autres supportent des paradigmes multiples (C++, Common Lisp, Python).
Il est à noter toutefois une convergence systèmatique de la plupart des langages au fil du temps : si un paradigme montre une supériorité incontestable, les autres langages reprenent et adaptent ce paradigme dans leur propre langage. C'est ainsi par exemple que PHP s'est doté de capacités d'orientation objet.

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 (ou procédurale)

La programmation structurée constitue un sous-ensemble de la programmation impérative. C'est un paradigme important de la programmation, apparu vers 1970.Elle est célèbre pour son essai de suppression de l'instruction GOTO ou du moins pour la limitation de son usage.
La programmation structurée est possible dans n'importe quel langage de programmation procédural, mais certains, comme le Fortran IV , s'y prêtaient très mal. Vers 1970, la programmation structurée devint une technique populaire, et les langages de programmation procéduraux intégrèrent des mécanismes rendant aisée la programmation structurée. Parmi les langages de programmation les plus structurants, on trouve PL/I, Pascal et, plus tardivement pour les projets de très grande taille, Ada.

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, notée « aller à »en pseudo-code, instruction qui envoie directement le programme à la ligne spécifiée. 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`»

```
Programmation Structurée
Si condition Alors
   instructions 1
Sinon
   instructions 2
FinSi
```

```
Programmation non structurée
1000 Si condition Alors Aller En 1200
1100 instruction 1
1110 etc.
1120 etc.
1190 Aller en 1400
1200 instruction 2
1210 etc.
1220 etc.
1400 suite de l'algorithme
```

Les programmeurs décomposent leur code en procédures ne dépassant guère 50 lignes, afin d'avoir le programme en entier sous leurs yeux. Une procédure, aussi appelée routine, sous-routine, module ou fonction, contient 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).
Il n'y a que des avantages à découper un programme en procédures :

- Un même code réutilisable à différents emplacements d'un programme sans avoir à le retaper ;
- Evolution du programme plus simple à suivre (absence d'instructions « GOTO ») ;
- Code créé plus modulaire et structuré ;
- Chaque programmeur peut développer son bout de code de son côté.

Programmation orientée objet

La programmation orientée objet (POO) 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. Le langage Simula-67 jette les prémisses de la programmation objet, résultat des travaux sur la mise au point de langages de simulation informatique dans les années 1960 dont s'inspira aussi la recherche sur l'intelligence artificielle dans les années 1970-80. Mais c'est réellement par et avec Smalltalk 72 puis Smalltalk 80, inspiré en partie par Simula, que la POO débute et que sont posés les concepts de base de celle-ci : objet, messages, encapsulation, polymorphisme, héritage, etc.
À partir des années 1980, commence l'effervescence des langages à objets : Objective C (début des années 1980), C++ en 1983, Eiffelen 1984, Common Lisp Object System dans les années 1980, etc. Les années 1990 voient l'âge d'or de l'extension de la POO dans les différents secteurs du développement logiciel : Ada, Java, C#, Objective C, Eiffel, Python, C++, PHP , Smalltalk...

Programmation fonctionnelle

La programmation fonctionnelle est un paradigme de programmation qui considère le calcul en tant qu'évaluation de fonctions mathématiques et rejette le changement d'état et la mutation des données. Elle souligne l'application des fonctions, contrairement au modèle de programmation impérative qui met en avant les changements d'état.
Le langage fonctionnel le plus ancien est Lisp, créé en 1958 par McCarthy. Lisp a donné naissance à des variantes telles que Scheme (1975) et Common Lisp (1984). Haskell (1987) est aussi un langage à paradigme fonctionnel. La programmation fonctionnelle s'affranchit de façon radicale des effets secondaires en interdisant toute opération d'affectation.
Le paradigme fonctionnel n'utilise pas de machine d'états pour décrire un programme, mais un emboîtement de « boîtes noires » imbriquées les unes dans les autres. Chaque boîte possédant plusieurs paramètres en entrée mais une seule sortie, elle ne peut sortir qu'une seule valeur possible pour chaque combinaison de valeurs présentées en entrée. Ainsi, les fonctions n'introduisent pas d'effets de bord. Un programme est donc une application, au sens mathématique, qui ne donne qu'un seul résultat pour chaque ensemble unitaire de valeurs en entrée. Cette façon de penser, qui est très différente de la pensée habituelle en programmation impérative, est l'une des causes principales de la difficulté qu'ont les programmeurs formés aux langages impératifs pour aborder la programmation fonctionnelle. Cependant, elle ne pose généralement pas de difficultés particulières aux débutants qui n'ont jamais été exposés à des langages impératifs.

5 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 valeurde l'expression `t`. Si `x` contenait déjà une valeur, celle-ci est écrasée.

Exemples

```
x = 3 // après cette affectation, la variable x contient la valeur 3.
y = 4+x // après cette instruction, la variable y contient la valeur 7.
```

Affectation et comparaison

Il ne faut pas confondre l'affectation avec la comparaison. Par exemple, la comparaison `x==3` permet de savoir si oui ou non la valeur de la variable `x` est `3`. Dans la plupart des langages, ces deux opérations sont nettement différenciées. En Python, en Javascript et en Java (entre autres), l'affectation est exprimée par un « `=` », tandis que la comparaison est exprimée par « `==` ». D'autres langages (Pascal par exemple) utilisent respectivement les opérateurs « `:=` » et « `=` ».

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`
Voici ce qui se passe : 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.

```
SI Test 1 ALORS
   Instruction 1
SINON
   Instruction 2
FIN SI
Instruction 3
```

Par exemple, en Python, on aura :

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

Le même test en Java serait :

```
if (x>=10) x=x-20 else x=x+2
```

On peut enchaîner autant d'instructions « sinon si » que l'on veut : seule la première dont la condition sera vérifiée sera exécutée. On peut généralement associer une clause sinon qui ne sera exécutée que si aucune clause sinon si n'a été vérifiée.

```
SI Test 1 ALORS
  Instruction 1
SINON SI Test 2 ALORS
    Instruction 2
FIN SI
Instruction 3
```

Par exemple, en Python 3:

```
if x==0:
   print("x est nul")
elif x<0:
   print("x est négatif")
else:
   print("x est positif")
```

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.

Boucle « Tant que »

```
TANT QUE Test
  Instruction 1
FIN TANT QUE
Instruction 2
```

Par exemple, en Python 3, ce programme écrira tous les entiers de 0 à 9 :

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

Le grand danger est ce qu'on appelle des boucles infinies, c'est-à-dire des boucles qui ne finissent jamais. Cela arrive quand la condition est toujours satisfaite, typiquement quand on oublie d'incrémenter un compteur.

Voici une boucle qui écrira une infinité de 0, car on a oublié d'incrémenter x :

```
x=0
while x<10:
  print(x)
```

Boucles imbriquées

Il est tout à fait possible, et parfois nécessaire, d'imbriquer une boucle dans une autre boucle. Par exemple, imaginons que l'on veuille écrire toutes les heures et minutes d'une journée. Cela commencera par 0 heure 0 minute, 0 heure 1 minute, 0 heure 2 minutes, ..., 0 heure 59 minutes, 1heure 0 minute, 1 heure 1 minute, ... On voit qu'une première boucle parcourra les heures, tandis que la deuxième parcourra les minutes, et que l'on incrémentera l'heure seulement quand 60 minutes seront passées. Voilà ce que cela donnera en Python 3:

```
h=0
while h<24:
  m=0
  while m<60:
    print(h,"heure(s) ",m,"minute(s)")
    m+=1
  h+=1
```

Si l'on veut ajouter les secondes, on imbriquera une troisième boucle :

```
h=0
while h<24:
  m=0
  while m<60:
    s=0
    while s<60:
      print(h,"heure(s) ",m,"minute(s)",s,"seconde(s)")
      s+=1
    m+=1
  h+=1
```

Boucle « Jusqu'à ce que »

Contrairement à une boucle « Tant que », une boucle « Jusqu'à ce que » exécute au moins une fois les instructions comprises dans la boucles.

```
REPETER
  Instruction 1
JUSQU'A CE QUE condition
Instruction 2
```

Ce type de boucle n'existe pas en Python, mais on le trouve en Pascal:

```
x=0;
repeat
  x=x+1;
  print(x)
until x=10;
```

Compteur

Un compteur permet de réaliser une boucle associée à une variable entière qui sera incrémentée (ou décrémentée) à chaque itération.

```
POUR compteur DE 0 A fin
  Instruction 1
FIN POUR
Instruction 2
```

En BASIC :

```
FOR i = depart TO fin instruction NEXT i
```

En Java :

```
for (i = depart; i = fin; i++) {instruction};
```

Itérateur

Un itérateur est un objet qui permet de réaliser une boucle parcourant tous les éléments contenus dans une structure de données (par exemple une liste).

```
POUR CHAQUE valeur DANS collection
  Instruction 1
FIN POUR CHAQUE
Instruction 2
```

Exemple en Python:

```
for animal in ["chat", "chien", "poule"] :
print(animal)
```

Les sous-programmes

Un sous-programme est un ensemble d'instructions pouvant être appelé depuis plusieurs endroits du programme. Les sous-programmes sont utilisés pour améliorer la structure du programme et sa lisibilité. Ces constructions ajoutent la notion de passage de paramètres et aussi la notion de variables locales qui évite que le sous-programme ait un effet de bord sur la routine appelante. Une fonction peut d'une part effectuer une action (par exemple afficher un résultat), et d'autre part retourner une valeur. Une **fonction** qui ne renvoie pas de valeur est appelée **procédure**.

Portée d'une variable

En Python, une variable définie au moment où elle est affectée d'une valeur. Une variable définie dans le programme principal est visible de l'intérieur des fonctions, mais une variable définie dans une fonction n'est pas visible de l'extérieur (à moins d'utiliser l'instruction `global`). Deux variables peuvent donc avoir le même nom mais être différentes selon l'endroit où elles sont définies. Dans une fonction, Python utilise une variable définie localement. S'il elle n'est pas définie localement, Python recherche la variable au niveau global, mais dans ce cas, il n'est pas possible de modifier la variable globale.

Prenons un exemple en Python (attention : tous ces exemples sont très mauvais du point de vue de la lisibilité. Ils sont juste là pour préciser le concept de portée d'une variable) :

```
def ecrire():
  n = 3
  print(n,end=' ')


n = 5
ecrire()
print(n)
```

Ce programme écrira `3 5`. On voit bien que les deux `n` sont des variables différentes, puisque l'instruction `n=3` n'a pas modifié la valeur de l'autre `n`. Le `n` de la deuxième ligne est une variable locale à la procédure `ecrire`, tandis que celui de la cinquième ligne est une variable locale. Modifions légèrement le programme :

```
def ecrire():
  n += 3
  print(n,end=' ')

n = 5
ecrire()
print(n)
```

Ce programme produira une erreur, car le `n` local dans la procédure `écrire` n'a pas été initialisé.

Passage de paramètres

On peut passer des paramètres à une procédure ou une fonction sous forme d'une liste de paramètres formels entre parenthèses. A l'intérieur du corps de la procédure, les paramètres sont des variables locales. Il ne faut pas redéclarer le nom des paramètres dans la section des déclarations locales. Voici un exemple en Pascal, qui calcule x^n :

```
var n : integer;
function puissance(x:real; n:integer) : real;
var p:real;

begin
  p:=1;
  while n>0 do
    begin
      p:=p*x;
      n:=n-1
    end;
  puissance:=p
end;

begin
  n:=3;
  writeln('Pi au cube =',puissance(3.14159,n));
  writeln(n) (*la valeur de n n'a pas été modifiée*)
end.
```

C'est la valeur du paramètre qui est passé à la procédure. Si la valeur est modifiée à l'intérieur de la procédure, la modification n'est pas répercutée à la sortie de la procédure. On appelle ce type de passage de paramètre un passage par valeur.
Dans certains langages, il est possible de passer des variables comme paramètres. On appelle cela un passage par variable ou passage par référence. Toute modification du paramètre dans la fonction appelée entraîne alors la modification de la variable passée en paramètre.
Voici un autre programme en Pascal, qui échange les valeurs de deux variables :

```
var x,y : real;
procedure echanger(var a,b:real)
  var aux:real;
  begin
    aux:=a; a:=b; b:=aux
  end;

begin
 x:=3; y:=4;
 echanger(x,y);
 writeln(' x=',x, ', y=',y)
end.
```

Ce programme écrira x=4, y=3.
Puisque la procédure attend des variables en paramètres, on ne peut pas appeler `echanger` avec des valeurs : `echanger(3,4)` est interdit, car 3et 4 ne sont pas des variables : on ne peut pas les modifier.
En Python, le passage des paramètres se fait toujours par référence, MAIS certains types sont « immutables », c'est-à-dire non modifiables. Les règles sont les suivantes :

- si un paramètre transmis à une fonction est une valeur immutable (les nombres et les chaînes de caractères), la fonction n'aura aucun moyen de modifier l'original. La valeur de départ est conservée ;
- si un paramètre est une valeur mutable (les listes par exemple), la fonction pourra modifier l'original, et avoir ainsi un effet de bord (désiré ou non).

Prenons un exemple (vicieux) :

```
def ecrire(x):
  x=5
  print(x,end=' ')


x=0
ecrire(x)
print(x)
```

Le résultat sera 5 0, alors que l'on s'attendait plutôt à 5 5. Cela vient du fait que le type entier est immutable.

En revanche :

```
def ecrire(x):
  x[0]=5 # on modifie le 1er élément de la liste x
  print(x[0],end=' ')


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

produira 5 5, car une liste est mutable. D'une manière générale, il vaut mieux éviter les effets de bords, afin d'avoir un programme plus lisible. Une fonction est dite à effet de bord si elle modifie un état autre que sa valeur de retour.