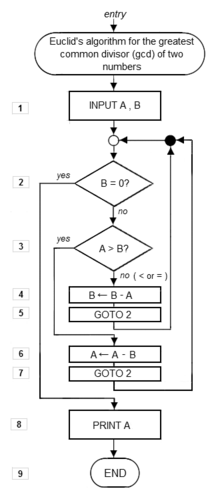
**Algorithmique**

[](http://commons.wikimedia.org/wiki/File:Euclid_flowchart_1.png)

Organigramme de programmation représentant l'algorithme d'Euclide

L'**algorithmique** est l’ensemble des règles et des techniques qui sont impliquées dans la définition et la conception d'algorithmes, c'est-à-dire de processus systématiques de résolution d'un problème permettant de décrire les étapes vers le résultat. En d'autres termes, un algorithme est une suite finie et non-ambiguë d’instructions permettant de donner la réponse à un problème.

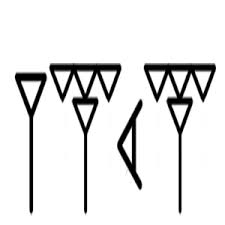
Si les instructions d'un algorithme s’exécutent les unes après les autres, l'algorithme est dit *séquentiel*, si elles s’exécutent en même temps, il est *parallèle*. Si l'algorithme exploite des tâches s’exécutant sur un réseau de processeurs on parle d’algorithme réparti, ou distribué.

Le mot « algorithme » vient du nom du mathématicien Al Khuwarizmi (latinisé au Moyen Âge en *Algoritmi*), qui, au IXe siècle écrivit le premier ouvrage systématique sur la solution des équations linéaires et quadratiques.

**Histoire**

**Antiquité**

Les premiers algorithmes dont on a retrouvé des descriptions datent des Babyloniens, au IIIe millénaire av. J.-C.. Ils décrivent des méthodes de calcul et des résolutions d'équations à l'aide d'exemples.



Calcul antique de numération babylonienne - 3854

Un algorithme célèbre est celui qui se trouve dans le livre 7 des *Éléments d'Euclide*, et appelé algorithme d'Euclide. Il permet de trouver le plus grand diviseur commun, ou PGCD, de deux nombres. Un point particulièrement remarquable est qu’il contient explicitement une itération et que les propositions 1 et 2 démontrent sa convergence.

**Étude systématique**

Le premier à avoir systématisé des algorithmes est le mathématicien arabophone Al Khuwarizmi, actif entre 813 et 833. Dans son ouvrage *Abrégé du calcul par la restauration et la comparaison*, il étudie toutes les équations du second degré et en donne la résolution par des algorithmes généraux. Il utilise des méthodes semblables à celles des Babyloniens, mais se différencie par ses explications systématiques là où les Babyloniens donnaient seulement des exemples.

Le savant arabe Averroès (1126-1198) évoque une méthode de raisonnement où la thèse s’affine étape par étape, itérativement, jusqu’à une certaine convergence et ceci conformément au déroulement d’un algorithme. À la même époque, au XIIe siècle, le moine Adelard de Bath introduit le terme latin de *algorismus*, par référence au nom de Al Khuwarizmi. Ce mot donne *algorithme* en français en 1554.

Au XVIIe siècle, on pourrait entrevoir une certaine allusion à la méthode algorithmique chez René Descartes dans la méthode générale proposée par le Discours de la méthode (1637), notamment quand, en sa deuxième partie, le logicien français propose de « diviser chacune des difficultés que j’examinerois, en autant de parcelles qu’il se pourroit, et qu’il seroit requis pour les mieux résoudre. » Sans évoquer explicitement les concepts de boucle, d’itération ou de dichotomie, l’approche de Descartes prédispose la logique à accueillir le concept de programme, mot qui naît en français en 1677.

L’utilisation du terme *algorithme* est remarquable chez Ada Lovelace, fille de Lord Byron et assistante de Charles Babbage (1791-1871).

**Les temps modernes**

Avec l'informatique, l'algorithmique s'est beaucoup développée. Donald Knuth (né en 1938), auteur du traité The Art of Computer Programming, décrit de très nombreux algorithmes et posa des fondements mathématiques rigoureux pour l'analyse des algorithmes. La revue Communications de l'ACM (Association for Computing Machinery) a été consacrée principalement à la description et à l'analyse des algorithmes pendant plus de 20 ans.

**Vocabulaire**

Le substantif *algorithmique* désigne la méthode utilisant des algorithmes. Le terme est également employé comme adjectif.

Un algorithme énonce une résolution sous la forme d’une *série d’opérations à effectuer*. La mise en œuvre de l’algorithme consiste en l’écriture de ces opérations dans un langage de programmation et constitue alors la brique de base d’un programme informatique.

Les informaticiens utilisent fréquemment l’anglicisme *implémentation* pour désigner cette mise en œuvre. L’écriture en langage informatique est aussi fréquemment désignée par le terme « codage », qui n’a ici aucun rapport avec la cryptographie, mais qui se réfère au terme « code source » pour désigner le texte, en langage de programmation, constituant le programme. L’algorithme devra être plus ou moins détaillé selon le niveau d’abstraction du langage utilisé, de même qu'une recette de cuisine doit être plus ou moins détaillée selon l’expérience du cuisinier.

**Étude formelle**

De nombreux outils formels ou théoriques ont été développés pour décrire les algorithmes, les étudier, exprimer leurs qualités, pouvoir les comparer :

* Ainsi, pour décrire les algorithmes, des structures algorithmiques ont été mises en évidence : structures de contrôle et structures de données.
* Pour justifier de la qualité des algorithmes, les notions de correction, de complétude et de terminaison ont été mises en place.
* Enfin, pour comparer les algorithmes, une théorie de la complexité des algorithmes a été définie.

**Structures algorithmiques**

Les concepts en œuvre en algorithmique, par exemple selon l'approche de N. Wirth pour les langages les plus répandus (Pascal, C, etc.), sont en petit nombre. Ils appartiennent à deux classes :

* les structures de contrôle
  + séquences
  + conditionnelles
  + boucles
* les structures de données
  + constantes
  + variables
  + tableaux
  + structures récursives (listes, arbres, graphes)

Ce découpage est parfois difficile à percevoir pour certains langages (Lisp, Prolog, …) plus basés sur la notion de récursivité où certaines structures de contrôle sont implicites et, donc, semblent disparaître.

**Correction, complétude, terminaison**

Ces trois notions « correction », « complétude », « terminaison » sont liées, et supposent qu'un algorithme est écrit pour résoudre un problème.

La terminaison est l'assurance que l'algorithme terminera en un temps fini. Les preuves de terminaison font habituellement intervenir une fonction entière positive strictement décroissante à chaque « pas » de l'algorithme.

Étant donnée la garantie qu'un algorithme terminera, la preuve de correction doit apporter l'assurance que si l'algorithme termine en donnant une proposition de solution, alors cette solution est correcte — c'est-à-dire qu'elle est effectivement une solution au problème posé.

La preuve de complétude garantit que, pour un espace de problèmes donné, l'algorithme, s'il termine, donnera toutes les solutions.

**Complexité algorithmique**

Les principales notions mathématiques dans le calcul du coût d’un algorithme précis sont les notions de domination (notée *O(f(n))*, « grand o »), où *f* est une fonction mathématique de *n*, variable désignant la quantité d’informations (en bits, en nombre d’enregistrements, etc.) manipulée dans l’algorithme. En algorithmique on trouve souvent des complexités du type :

|  |  |
| --- | --- |
| **Notation** | **Type de complexité** |
| O(1) | complexité constante (indépendante de la taille de la donnée) |
| O(log(n)) | complexité logarithmique |
| O(n) | complexité linéaire |
| O(n log(n)) | complexité quasi-linéaire |
| O(n^{2}) | complexité quadratique |
| O(n^{3}) | complexité cubique |
| O(n^p) | complexité polynomiale |
| O(n^{\log(n)}) | complexité quasi-polynomiale |
| O(2^{n}) | complexité exponentielle |
| O(n!) | complexité factorielle |

Sans entrer dans les détails mathématiques, le calcul de l’efficacité d’un algorithme (sa *complexité algorithmique*) consiste en la recherche de deux quantités importantes. La première quantité est l’évolution du nombre d’instructions de base en fonction de la quantité de données à traiter (par exemple, pour un algorithme de tri, il s'agit du nombre de données à trier), que l’on privilégiera sur le temps d'exécution mesuré en secondes (car ce dernier dépend de la machine sur laquelle l'algorithme s'exécute). La seconde quantité estimée est la quantité de mémoire nécessaire pour effectuer les calculs. Baser le calcul de la complexité d’un algorithme sur le temps ou la quantité effective de mémoire qu’un ordinateur particulier prend pour effectuer ledit algorithme ne permet pas de prendre en compte la structure interne de l’algorithme, ni la particularité de l’ordinateur : selon sa charge de travail, la vitesse de son processeur, la vitesse d’accès aux données, l’exécution de l’algorithme (qui peut faire intervenir le hasard) ou son organisation de la mémoire, le temps d’exécution et la quantité de mémoire ne seront pas les mêmes.

Souvent, on examine les performances "au pire", c'est-à-dire dans les configurations telles que le temps d'exécution (ou l'espace mémoire) est le plus grand. Il existe également un autre aspect de l'évaluation de l'efficacité d'un algorithme : les performances "en moyenne". Cela suppose d'avoir un modèle de la répartition statistique des données de l'algorithme, tandis que la mise en œuvre des techniques d'analyse implique des méthodes assez fines de combinatoire et d'évaluation asymptotique, utilisant en particulier les séries génératrices et des méthodes avancées d'analyse complexe. L'ensemble de ces méthodes est regroupé sous le nom de combinatoire analytique.

On trouvera dans l’article sur la théorie de la complexité des algorithmes d’autres évaluations de la complexité qui vont en général au-delà des valeurs proposées ci-dessus et qui répartissent les problèmes (plutôt que les algorithmes) en classes de complexité.

**Quelques indications sur l’efficacité des algorithmes**

Souvent, l’efficacité d’un algorithme n’est connue que de manière asymptotique, c’est-à-dire pour de grandes valeurs du paramètre *n*. Lorsque ce paramètre est suffisamment petit, un algorithme de complexité supérieure peut en pratique être plus efficace. Ainsi, pour trier un tableau de 30 lignes (c’est un paramètre de petite taille), il est inutile d’utiliser un algorithme évolué comme le Tri rapide (l’un des algorithmes de tri les plus efficaces en moyenne) : l’algorithme de tri le plus trivial sera suffisamment efficace.

Entre deux algorithmes dont la complexité est identique, on cherchera à utiliser celui dont l’occupation mémoire est la plus faible. L’analyse de la complexité algorithmique peut également servir à évaluer l’occupation mémoire d’un algorithme. Enfin, le choix d’un algorithme plutôt qu’un autre doit se faire en fonction des données que l’on s’attend à lui fournir en entrée. Ainsi, le *Quicksort* (ou tri rapide), lorsque l’on choisit le premier élément comme pivot, se comporte de façon désastreuse si on l’applique à une liste de valeurs déjà triée. Il n’est donc pas judicieux de l’utiliser si on prévoit que le programme recevra en entrée des listes déjà presque triées.

Un autre paramètre à prendre en compte est la localité de l’algorithme. Par exemple pour un système à mémoire virtuelle qui dispose de peu de mémoire (par rapport au nombre de données à traiter), le Tri rapide sera normalement plus efficace que le Tri par tas car le premier ne passe qu’une seule fois sur chaque élément de la mémoire tandis que le second accède à la mémoire de manière discontinue (ce qui augmente le risque de *swapping*).

Enfin, il existe certains algorithmes dont la complexité est dite amortie. Cela signifie que, pour certaines exécutions de l’algorithme (cas marginaux), la complexité de l’algorithme sera très supérieure au cas moyen. Bien sûr, on n’utilise la notion de complexité amortie que dans les cas où cette réaction est très marginale.

**Approches pratiques**

L'algorithmique a développé quelques stratégies pour résoudre les problèmes :

* algorithme glouton : un premier algorithme peut souvent être proposé en ne regardant que les cas simples, ou ceux apparaissant le plus souvent. On parle alors d'algorithme glouton. L'algorithme glouton n'est souvent qu'une première étape dans la rédaction d'un algorithme plus performant.
* diviser pour régner : pour améliorer les performances des algorithmes, une technique usuelle consiste à diviser les données d'un problème en sous-ensembles de tailles plus petites, jusqu'à obtenir des données que l'algorithme pourra traiter au cas par cas. Une seconde étape dans ces algorithmes consiste à « fusionner » les résultats partiels pour obtenir une solution globale. Ces algorithmes sont souvent associés à la récursivité.
* recherche exhaustive (ou combinatoire) : une méthode utilisant l'énorme puissance de calcul des ordinateurs consiste à regarder tous les cas possibles. Cela n'est pour autant possible que dans certains cas particuliers (la combinatoire est souvent plus forte que l'énorme puissance des ordinateurs, aussi énorme soit-elle)
* aléatoire, ou par approximations successives : certains algorithmes utilisent des recherches aléatoires, ou par approches successives, donnant de meilleurs résultats (en moyenne) que des recherches directes ou explicites.
* décomposition top-down / bottom-up : les décompositions top-bottom consistent à essayer de décomposer le problème en sous-problèmes à résoudre successivement, la décomposition allant jusqu'à des problèmes triviaux faciles à résoudre. L'algorithme global est alors donné par la composée des algorithmes définis au cours de la décomposition. La démarche bottom-up est la démarche inverse, elle consiste à partir d'algorithmes simples, ne résolvant qu'une étape du problème, pour essayer de les composer pour obtenir un algorithme global.
* pré-traitement / post-traitement : parfois, certains algorithmes comportent une ou deux phases identifiées comme des pré-traitements (à faire avant l'algorithme principal), ou post-traitement (à faire après l'algorithme principal), pour simplifier l'écriture de l'algorithme général.

**Les heuristiques**

Pour certains problèmes, les algorithmes ont une complexité beaucoup trop grande pour obtenir un résultat en temps raisonnable, même si l’on pouvait utiliser une puissance de calcul phénoménale. On est donc amené à rechercher une solution la plus proche possible d’une solution optimale en procédant par essais successifs. Puisque toutes les combinaisons ne peuvent être essayées, certains choix stratégiques doivent être faits. Ces choix, généralement très dépendants du problème traité, constituent ce qu’on appelle une heuristique. Le but d’une heuristique n'est donc pas d'essayer toutes les combinaisons possibles afin de trouver celle répondant au problème, mais de trouver une solution approchée convenable (qui peut être exacte dans certains cas) dans un temps raisonnable. C’est ainsi que les programmes de jeu d’échecs ou de jeu de go (pour ne citer que ceux-là) font appel de manière très fréquente à des heuristiques qui modélisent l’expérience d’un joueur. Certains logiciels antivirus se basent également sur des heuristiques pour reconnaître des virus informatiques non répertoriés dans leur base, en s’appuyant sur des ressemblances avec des virus connus.

# Langage de programmation

[](http://commons.wikimedia.org/wiki/File:Source_code_in_Javascript.png)

langage de programmation JavaScript.

En informatique, un **langage de programmation** est une notation conventionnelle destinée à formuler des algorithmes et produire des programmes informatiques qui les appliquent. D'une manière similaire à une langue naturelle, un langage de programmation est fait d'un alphabet, un vocabulaire, des règles de grammaire, et des significations1,2.

Les langages de programmation permettent de décrire d'une part les structures des données qui seront manipulées par l'appareil informatique, et d'autre part d'indiquer comment sont effectuées les manipulations, selon quels algorithmes. Ils servent de moyens de communication par lesquels le programmeur communique avec l'ordinateur, mais aussi avec d'autres programmeurs; les programmes étant d'ordinaire écrits, lus, compris et modifiés par une communauté3.

Un langage de programmation est mis en œuvre par un traducteur automatique: compilateur ou interpréteur. Un compilateur est un programme informatique qui transforme dans un premier temps un code source écrit dans un langage de programmation donné en un code cible qui pourra être directement exécuté par un ordinateur, à savoir un programme en langage machine ou en code intermédiaire2, tandis que l'interpréteur réalise cette traduction 'à la volée'.

Les langages de programmation offrent différentes possibilités d'abstraction, et une notation proche de l'algèbre, permettant de décrire de manière concise et facile à saisir les opérations de manipulation de données et l'évolution du déroulement du programme en fonction des situations. La possibilité d'écriture abstraite libère l'esprit du programmeur d'un travail superflu, notamment de prise en compte des spécificités du matériel informatique, et lui permet ainsi de se concentrer sur des problèmes plus avancés2.

Chaque langage de programmation reflète un *paradigme*, un ensemble de notions qui orientent le travail de réflexion du programmeur, sa technique de programmation et sa manière d'exprimer le fruit de ses réflexions dans le langage choisi.

Les premiers langages de programmation ont été créés dans les années 1950. De nombreux concepts de l'informatique ont été lancés par un langage, avant d'être améliorés et étendus dans les langages suivants. La plupart du temps la conception d'un langage de programmation a été fortement influencée par l'expérience acquise avec les langages précédents4.

## Définition

Sous un angle théorique, tout langage informatique peut être qualifié de langage de programmation s'il est Turing-complet c'est-à-dire qu'il permet de représenter toutes les fonctions calculables au sens de Turing et Church (en admettant néanmoins pour exception à la théorie que la mémoire des ordinateurs n'est pas un espace infini)5.

### Éléments du langage de programmation

Un langage de programmation est construit à partir d'une grammaire formelle, qui inclut des symboles et des règles syntaxiques, auxquels on associe des règles sémantiques. Ces éléments sont plus ou moins complexes selon la capacité du langage. Les modes de fonctionnement et de définition de la complexité d'un langage de programmation sont généralement déterminé par leur appartenance à l'un des degrés de la Hiérarchie de Chomsky.

Les *règles de syntaxe*

Définies par une Grammaire formelle, elles régissent les différentes manières dont les éléments du langage peuvent être combinés pour obtenir des programmes2. La ponctuation (par exemple l'apposition d'un symbole **;** en fin de ligne d'instruction d'un programme) relève de la syntaxe.

Le *vocabulaire*

Parmi les éléments du langage, le vocabulaire représente l'ensemble des instructions construites d’après des *symboles*. L'instruction peut être mnémotechnique ou uniquement symbolique comme quand elle est représentée par des *symboles d'opérations* tels que des opérateurs arithmétiques (« + » et « - ») ou booléens (&& pour le et logique par exemple). On parle aussi parfois de *mot clé* pour désigner une instruction (par abus de langage car le concept de mot clé ne recouvre par celui des symboles qui font pourtant eux aussi partie du *vocabulaire*).

La *sémantique*

Les règles de *sémantique* définissent le sens de chacune des phrases qui peuvent être construites dans le langage, en particulier quels seront les effets de la phrase lors de l'exécution du programme2. La science l’étudiant est la sémantique des langages de programmation.

L’*alphabet*

l'alphabet des langages de programmation est basé sur les normes courantes comme ASCII, qui comporte les lettres de A à Z sans accent, des chiffres et des symboles6, ou Unicode pour la plupart des langages modernes (dans lesquels l'utilisation se limite en général aux chaînes de caractères littérales et aux commentaires, avec quelques exceptions notables comme C♯ qui autorisent également les identifiants unicode).

La plupart des langages de programmation peuvent prévoir des éléments de structure complémentaires, des méthodes procédurales, et des définitions temporaires et variables et des identifiants:

Les *commentaires*

Les *commentaires* sont des textes qui ne seront pas traduits. Ils peuvent être ajoutés dans les programmes pour y laisser des explications. Les commentaires sont délimités par des marques qui diffèrent d'un langage de programmation à l'autre tel que « -- », « /\* » ou « // »6.

Les *identifiants*

Les éléments constitutifs du programme, tels que les *variables*, les *procédures*, ou les *types* servent à organiser le programme et son fonctionnement. On peut ainsi, par exemple, diviser un programme en fonctions, ou lui donner une structure par objets : ces éléments de structure sont définis par des identifiants ou des procédures par *mot clé* selon le langage.

## Utilisation

Un langage de programmation offre un cadre pour élaborer des algorithmes et exprimer des diagrammes de flux7,6. Il permet en particulier de décrire les structures des données qui seront manipulées par l'appareil informatique et quelles seront les manipulations. Un langage de programmation sert de moyen de communication avec l'ordinateur mais aussi entre programmeurs : les programmes étant d'ordinaire écrits, lus et modifiés par une équipe de programmeurs3.

Un langage de programmation offre un ensemble de notions qui peuvent être utilisées comme primitives pour développer des algorithmes. Les programmeurs apprécient que le langage soit clair, simple, et unifié, qu'il y ait un minimum de notions qui peuvent être combinées selon des règles simples et régulières. Les qualités d'un langage de programmation influent sur la facilité avec laquelle les programmes pourront être écrits, testés, puis plus tard compris et modifiés6.

Les langages de programmation offrent différentes possibilités d'abstraction, et une notation proche de l'algèbre, permettant de décrire de manière concise et facile à saisir les opérations de manipulation de données et l'évolution du déroulement du programme en fonction des situations. Cette possibilité d'écriture abstraire libère l'esprit du programmeur d'un travail superflu, et lui permet de se concentrer sur des problèmes plus avancés2.

La facilité d'utilisation, la portabilité et la clarté sont des qualités appréciées des langages de programmation. La facilité d'utilisation, qui dépend de la syntaxe, du vocabulaire et des symboles, influence la lisibilité des programmes écrits dans ce langage et la durée d'apprentissage. La portabilité permet à un programme écrit pour être exécuté par une plateforme informatique donnée (un système d'exploitation) d'être transféré en vue d'être exécuté sur une autre plateforme7.

Les programmeurs apprécient que la syntaxe permette d'exprimer la structure logique inhérente au programme. Un des soucis en programmation est d'éviter des pannes, qu'il soit possible de les détecter, les éviter et les rectifier; ceci est rendu possible par des mécanismes internes des langages de programmation. Des vérifications implicites sont parfois effectuées en vue de déceler des problèmes7.

Les programmeurs apprécient qu'un langage de programmation soit aligné avec les bonnes pratiques de programmation et d'ingénierie, qu'il encourage la structuration du programme, facilite la maintenance des programmes et qu'il dissuade voire interdise les mauvaises pratiques7. L'utilisation de l'instruction goto, qui existe depuis les premiers langages de programmation, est considérée comme une mauvaise pratique. Son utilisation est déconseillée, voire impossible dans les langages de programmation récents8.

L'alignement sur les standards industriels, la possibilité d'utiliser des fonctionnalités écrites dans un autre langage de programmation et l'exécution simultanée de plusieurs threads sont des possibilités appréciées des langages de programmation7.

### Notions courantes

Un langage de programmation permet de décrire les structures des données qui seront manipulées par l'appareil informatique et quelles seront les manipulations. Il offre un ensemble de notions telles que les instructions, les variables, les types, et les procédures, qui peuvent être utilisées comme primitives pour développer des algorithmes9.

Une *instruction*

Un ordre donné à un ordinateur10.

Une *variable*

Un nom utilisé dans un programme pour faire référence à une donnée manipulée par programme.

Une *constante*

Un nom utilisé pour faire référence à une valeur permanente.

Une expression *littérale*

Une valeur mentionnée en toutes lettres dans le programme9.

Un *type*

Chaque donnée a une classification, celle-ci influe sur la plage de valeurs possibles, les opérations qui peuvent être effectuées, et la représentation de la donnée sous forme de bits9. Chaque langage de programmation offre une gamme de types *primitifs*, incorporés dans le langage. Certains langages offrent la possibilité de créer des nouveaux types.

Les types de données primitifs courants sont les nombres entiers, les nombres réels, le booléen, les chaînes de caractères et les pointeurs.

Le type booléen

Un type qui n'a que deux valeurs : *vrai* et *faux*.

Le type pointeur

Une référence à une donnée, qui se trouve quelque part en mémoire9.

Une structure de données

Une manière caractéristique d'organiser un ensemble de données en mémoire, qui influe sur les algorithmes utilisés pour les manipuler. Les structures courantes sont les tableaux, les enregistrements, les listes, les piles et les arbres11.

Une *déclaration*

Une phrase de programme qui sert à renseigner au traducteur (compilateur, interpréteur, ...) les noms et les caractéristiques des éléments du programme tels que des variables, des procédures, de types3...

Des vérifications sont effectuées au moment de la compilation, ou au moment de l'exécution du programme, pour assurer que les opérations du programme sont possibles avec les types de données qui sont utilisés. Dans un langage *fortement typé*, chaque élément du programme a un type unique, connu et vérifié au moment de la compilation, ce qui permet de déceler des erreurs avant d'exécuter le programme3.

Les *procédures*, *fonctions*, *méthodes*

Divers langages de programmation offrent la possibilité d'isoler un fragment de programme, et d'en faire une opération générale, paramétrable, susceptible d'être utilisée de façon répétée. Ces fragments sont appelés *procédures*, *fonctions* ou *méthodes*.

Les *modules*

Les langages de programmation peuvent également offrir la possibilité de découper un programme en plusieurs pièces appelées *modules*, chacune ayant un rôle déterminé, puis de combiner les pièces3.

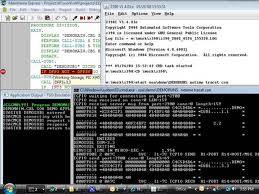
Les notions de *procédure* et de *module* sont destinées à faciliter la création de programmes complexes et volumineux en assistant la prise en charge de cette complexité. Ces fonctions permettent en particulier la modularité et l'abstraction3.

## Paradigmes

Chaque langage de programmation reflète un *paradigme* de programmation. C'est un ensemble de notions qui oriente le travail de réflexion du programmeur et peut être utilisé pour obtenir une solution à un problème de programmation. Chaque paradigme amène une technique différente de programmation; une fois qu'une solution a été imaginée par le programmeur selon un certain paradigme, un langage de programmation qui suit ce paradigme permettra de l'exprimer12.

Impératif

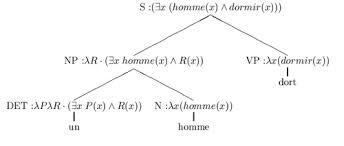
Le paradigme *impératif* ou *procédural* est basé sur l'idée d'une exécution étape par étape semblable à une recette de cuisine. Il est basé sur le principe de la machine de Von Neumann. Un ensemble de structures permet de contrôler l'ordre dans lequel sont exécutées les commandes qui décrivent les étapes. L'abstraction est réalisée à l'aide de procédures auxquelles sont transmises des données. Il existe une procédure principale, qui est la première à être exécutée, et qui peut faire appel à d'autre procédures pour effectuer certaines tâches ou certains calculs. Les langages de programmation C, Pascal, Fortran et COBOL sont en paradigme impératif12.



Fenêtres de développement

Fonctionnel

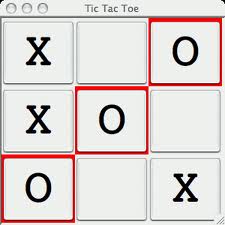
Le paradigme *fonctionnel* est basé sur l'idée d'évaluer une formule, et d'utiliser le résultat pour autre chose, selon le modèle du lambda-calcul. Tous les traitements sont faits en évaluant des expressions et en faisant appel à des fonctions, et l'exécution étape par étape n'est pas possible dans le paradigme fonctionnel. Le résultat d'un calcul sert de matière première pour le calcul suivant, et ainsi de suite, jusqu'à ce que toutes les fonctions aient produit un résultat. ML et Lisp sont des langages de programmation en paradigme fonctionnel12.



Exemple de Lambda calcul

Logique

Le paradigme *logique* est basé sur l'idée de répondre à une question par des recherches sur un ensemble, en utilisant des axiomes, des demandes et des règles de déduction. L'exécution d'un programme est une cascade de recherches de données dans un ensemble, en faisant usage de règles de déduction. Les données obtenues, et associées à un autre ensemble de règles peuvent alors être utilisées dans le cadre d'une autre recherche. L'exécution du programme se fait par évaluation, le système effectue une recherche de toutes les affirmations qui, par déduction, correspondent à au moins un élément de l'ensemble. Le programmeur exprime les règles, et le système pilote le processus. Prolog est un langage de programmation en paradigme logique12.



Prolog - répondre à une question

Orienté objet

Dans le paradigme *orienté objet*, chaque objet est une entité active, qui communique avec d'autres objets par échange de messages. Les échanges de message entre les objets simulent une évolution dans le temps d'un phénomène réel. Les procédures agissent sur les données et le tout est cloisonné dans des objets. Les objets sont groupés en *classes* ; les objets d'une même classe sont similaires du point de vue de leur structure et de leur comportement. La programmation consiste à décrire les classes. Les classes sont organisées selon une structure hiérarchique où il y a de l'héritage : de nouveaux objets peuvent être créés sur la base d'objets existants. Ce paradigme a été développé pour parer aux limitations de son prédécesseur, le paradigme procédural avec les très grands programmes, et surtout avec les programmes interactifs. Le paradigme orienté objet aide le programmeur à créer un modèle organisé du problème à traiter et à gérer les interactions. Il permet d'associer fortement les données avec des procédures adaptées pour réaliser des fonctions abstraites. Simula, Smalltalk, C++ et Java sont des langages de programmation en paradigme orienté objet.



Logo de Java