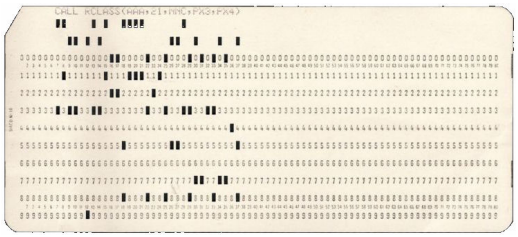
1 Introduction à la programmation

1 Évolution des systèmes d'exploitation

Les premiers ordinateurs étaient capables d'accomplir une seule tâche à la fois. Dans ce mode de fonctionnement, souvent appelé **traitement par lots mono-utilisateur**, l'ordinateur exécutait un programme à la fois et traitait les données en groupes, appelés lots. Les utilisateurs de ces systèmes présentaient généralement leur travail à un centre informatique sous la forme de **cartes perforées**. Souvent, des heures ou même des jours devaient s'écouler avant que les résultats puissent être renvoyés aux utilisateurs.

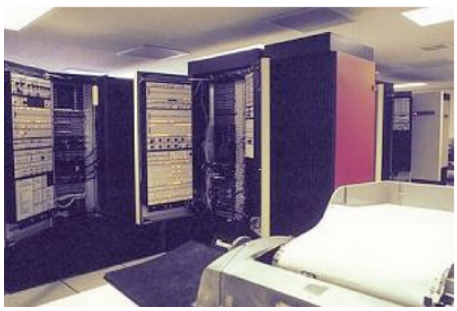
(a) Centre de calcul



(b) Carte perforée

FIGURE 1.1 – Ordinateurs des années 1955-1965 : traitement par lot

Comme les ordinateurs devinrent plus puissants, le traitement par lots mono-utilisateur fut vite dépassé. Les ingénieurs constatèrent que les ordinateurs passaient plus de temps à attendre qu'un périphérique termine son travail qu'à exécuter des instructions. Afin de permettre l'exécution de plusieurs programmes à la fois, des logiciels appelés **systèmes d'exploitation** furent développés. Les programmes étaient d'abord chargés en mémoire pour y résider simultanément. Le premier programme était exécuté, puis, lorsqu'il avait besoin d'attendre qu'un périphérique se libère, le système d'exploitation enchaînait sur l'exécution du deuxième programme, puis du troisième et ainsi de suite. Cette technique, dite de **multiprogrammation**, est le précurseur du multitâche des systèmes d'exploitation d'aujourd'hui.



(a) Ordinateurs supportant Multics

(b) Créateurs de UNIX

FIGURE 1.2 – Premiers systèmes d'exploitation : Multics et UNIX

En 1965, le *Massachusetts Institute of Technology* (MIT) se lance dans la réalisation d'un système d'exploitation **à temps partagé** : *Multics*. Sur le principe de la multiprogrammation, *Multics* autorisait le chargement de plusieurs programmes en mémoire et gérait le passage de l'un à l'autre, mais cette fois-ci sans attendre le blocage d'un programme. Chaque programme était exécuté pendant une durée de quelques millisecondes, puis le système passait au suivant. Ce temps, très court, donnait l'illusion que les programmes s'exécutaient simultanément – une illusion qui existe encore aujourd'hui. Les systèmes à temps partagé furent plus avantageux que les systèmes informatiques précédents en ce sens que les utilisateurs recevaient des réponses rapides à leurs demandes plutôt que d'attendre de longues périodes pour obtenir des résultats.

Ecrit en langage de programmation PL/1 et trop complexe pour les ordinateurs de l'époque, *Multics* fut un échec commercial. En 1969, les ingénieurs Dennis Ritchie et Ken Thompson des laboratoires Bell, se lance dans l'écriture d'une version allégée de *Multics*. Le système d'exploitation est baptisé *UNIX*. Rapidement reprogrammé dans un langage de programmation plus approprié, le C, UNIX se révéla particulièrement simple à porter sur de nouvelles plateformes, ce qui assura son succès. En outre, distribuant librement leur code source afin que d'autres programmeurs puissent l'utiliser, le modifier et l'étendre, Ritchie et Thompson contribuèrent pleinement à la création d'une large communauté d'utilisateurs UNIX. Grâce à un effort de collaboration entre de nombreux chercheurs et développeurs, UNIX devint un système d'exploitation flexible et puissant, capable de gérer n'importe quel type de tâche requis par l'utilisateur. De nombreuses versions d'UNIX furent développées, parmi lesquelles le système d'exploitation open-source *Linux*.

En 1975, l'*Altair 8800* fut l'un des premiers micro-ordinateurs au monde à être commercialisé. Vendu en kit au prix approximatif de 400 $, il se programmait en inversant une série d'interrupteurs à bascule situés sur la face de l'appareil. Le prix de l'*Alter 8800* le rendait accessible aux particuliers mais son utilisation compliquée, qui rappelait d'ailleurs les ordinateurs centraux des années 1940, limitait son utilisation à des passionnés. En outre, sa puissance de calcul ne lui permettait pas d'être équipés d'un système d'exploitation comme UNIX.

En 1980, IBM prend contact avec Bill Gates, co-fondateur de la société *Microsoft*, pour l'adaptation du langage BASIC à son nouveau micro-ordinateur, le *Personal Computer* (abrégé PC). IBM est également à la recherche d'un système d'exploitation. Gates décide alors de racheter QDOS, un système d'exploitation pour les processeurs Intel 8086, qu'il modifie quelque peu à la demande d'IBM. Le nouveau système est appelé MS-DOS.

C'est à la même époque que Steve Jobs et Steve Wozniak fondent la startup *Apple Computer* avec comme optique de faire du micro-ordinateur un produit grand public aussi facile d'usage que n'importe quel appareil électrique. *Xérox*, une des sociétés majeurs de l'époque, s'intéresse à l'optique de Jobs et réunit une poignée de scientifiques et d'ingénieurs dans son centre de recherche afin de développer le premier micro-ordinateur équipé d'une interface utilisateur graphique. Le résultat des recherches, le *Xerox Star*, ne sera jamais commercialisé mais c'est *Apple* qui, avec le *Macintosh*, popularisa les recherches effectuées par *Xerox*.

(a) Altair 8800 (1975) (b) IBM PC 5150 (1981) (c) Macintosh 128K (1984)

FIGURE 1.3 – Premiers micro-ordinateurs personnels

Les premiers ordinateurs mis au point par *Apple* et *IBM* étaient des unités indépendantes : les gens travaillaient sur leur machine et transportaient des disquettes pour partager leurs données. Bien qu'au début, les ordinateurs personnels n'étaient pas suffisamment puissants pour les utiliser en temps partagé, il était néanmoins possible de les relier ensemble sur des lignes téléphoniques ou via des réseaux locaux (LAN). Ces réseaux conduisirent au **calcul distribué**, dans lequel un traitement est réparti entre plusieurs ordinateurs reliés en réseaux plutôt que d'être effectué sur une seule machine.

De nos jours, la plupart des ordinateurs sont aussi puissants que les machines utilisées il y a une trentaine d'années et qui coûtaient alors près d'un million de dollars. Ces performances fournissent à chaque utilisateur d'énormes capacités de calcul. De plus, l'information est facilement transmissible via de nombreux réseaux sur lesquels des ordinateurs appelés **serveurs** stockent des programmes et des données que peuvent utiliser d'autres machines appelées **client**. Ce type de configuration a donné lieu à l'architecture client-serveur qu'utilisent tous les systèmes d'exploitation actuels comme UNIX, Solaris, MacOS, Windows ou Linux.

Pour plus d'informations : http://fr.wikipedia.org/wiki/Chronologie_informatique

2 Organisation interne d'un ordinateur

Un **ordinateur** est un automate à composants électriques capable d'effectuer des calculs et prendre des décisions logiques à des vitesses nettement supérieures à celles des êtres humains. Ainsi, les ordinateurs actuels les plus rapides peuvent effectuer des centaines de milliards d'additions par seconde correspondant au nombre de calculs qu'accompliraient ensemble des centaines de milliers de personnes en une année !

Un ordinateur est composé de divers dispositifs, tels que le clavier, l'écran, la souris, les disques, la mémoire et les unités de traitement, connus sous le nom de **matériel informatique** ou **hardware**. Les programmes qui s'exécutent sur un ordinateur sont quant à eux appelés **logiciels** ou **softwares**. Les coûts du matériel ont diminué de façon spectaculaire durant ces dernières années, au point que les ordinateurs personnels sont devenus des marchandises commercialisées même dans les grandes surfaces. Les coûts des logiciels, cependant, n'ont cessé d'augmenter en raison du développement d'applications de plus en plus puissantes et complexes. Le but de ce cours est de vous apprendre des méthodes éprouvées de réalisation de logiciels capables de réduire les coûts du développement. Un accent particulier sera mis sur la programmation orientée objet qui permet d'améliorer grandement la productivité des programmeurs.

2.1 Composants principaux d'un ordinateur

Tous les ordinateurs, indépendamment des différences qui composent leur apparence physique, peuvent être envisagés comme étant divisés en six **unités logiques** :

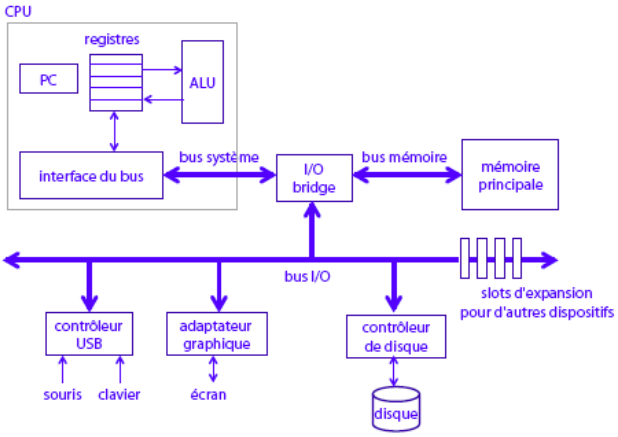


FIGURE 1.4 – Architecture classique d'un ordinateur

(1) L'**unité d'entrée** réceptionne les informations (données et programmes d'ordinateur) à partir de différents périphériques d'entrée et les met ensuite à la disposition des autres unités pour faciliter le traitement de l'information. Aujourd'hui, la plupart des utilisateurs transmettent des informations aux ordinateurs via le clavier, la souris et de plus en plus l'écran tactile. Parmi les autres dispositifs d'entrée, nous pouvons également mentionner le microphone (pour parler à l'ordinateur), le scanner (pour la numérisation de documents) et les caméras numériques (pour charger des photographies ou des vidéos).

(2) L'**unité de sortie** prend l'information que l'ordinateur a traitée et la place sur divers périphériques de sortie afin de la rendre disponible en dehors de la machine. Les ordinateurs peuvent extraire de l'information en l'affichant sur un écran, en la transmettant à des appareils audio ou vidéo, en l'imprimant sur du papier ou en l'utilisant pour contrôler d'autres appareils.

(3) L'**unité de mémoire** facilite le stockage temporaire des données en conservant l'information transmise par l'unité d'entrée sur un support de faible capacité mais d'accès rapide qui permet un traitement immédiat. En outre, l'unité de mémoire conserve l'information traitée jusqu'à ce qu'elle puisse être transmise aux dispositifs de sortie. L'unité de mémoire est souvent appelée **mémoire** ou **mémoire principale** - la mémoire à accès aléatoire (RAM) en est un exemple notoire. La mémoire principale est généralement **volatile**, ce qui signifie qu'elle est effacée lorsque la machine n'est plus alimentée en électricité.

(4) L'**unité arithmétique et logique** (ALU) est responsable de l'exécution de calculs tels que l'addition, la soustraction, la multiplication et la division. Elle contient également des mécanismes décisionnels permettant à l'ordinateur d'effectuer des tâches telles que déterminer si deux éléments stockés dans la mémoire sont égaux.

(5) L'**unité centrale de traitement** (CPU) est le coordonnateur de l'ordinateur, chargé de superviser le fonctionnement des autres unités. Le CPU, appelé également **processeur**, avertit l'unité d'entrée lorsque l'information doit être transmise à l'unité de mémoire, indique à l'ALU quand utiliser l'information de l'unité de mémoire pour les calculs et indique à l'unité de sortie quand envoyer l'information provenant de l'unité de mémoire sur certains périphériques de sortie.

(6) L'**unité de stockage secondaire** correspond à un entrepôt de grande capacité, capable de conserver sur des disques les programmes et les données qui ne sont pas actuellement utilisés par l'ordinateur. Ce dernier peut néanmoins récupérer ces informations lorsque celles-ci lui sont nécessaires et ceci des jours, des mois voire même des années après leur mise en mémoire. L'accès aux informations contenues dans l'unité de stockage secondaire est beaucoup plus long que l'accès aux informations situées dans la mémoire primaire. Toutefois, le prix par unité de mémoire secondaire est nettement inférieur à celui de la mémoire principale. Le stockage en mémoire secondaire est généralement non volatile : les données stockées sont conservées même lorsque l'ordinateur est éteint.

Pour plus d'informations : http://www.apprendre-en-ligne.net/info/comment/index.html

3 Langage machine, langage de programmation

La programmation d'un ordinateur consiste à *expliquer* en détail à une machine ce qu'elle doit faire, en sachant d'emblée qu'elle ne peut seulement qu'effectuer un traitement automatique sur des séquences de caractères. Il s'agit donc la plupart du temps de convertir un souhait en un raisonnement parfaitement structuré que l'on appelle un **algorithme**. Si les instructions mises en lumière par l'algorithme sont suffisamment simples, le programmeur pourra alors les encoder dans la machine en respectant de manière très stricte un ensemble de conventions fixées à l'avance, que l'on appelle un **langage informatique**. Pour *comprendre* celui-ci, la machine sera pourvue d'un mécanisme qui décode ces instructions en associant à chaque *mot* du langage une action précise. Ainsi seulement, le souhait pourra s'accomplir.

À strictement parler, un ordinateur n'est rien d'autre qu'une machine effectuant des opérations simples sur des séquences de signaux électriques, lesquels sont conditionnés de manière à ne pouvoir prendre que deux états seulement (un potentiel électrique maximum ou minimum). Ces séquences de signaux peuvent donc être considérés conventionnellement comme des suites de nombres ne prenant que les deux valeurs 0 et 1. Un système numérique ainsi limité à deux chiffres est appelé **système binaire**.

Sachez dès à présent que dans son fonctionnement interne, un ordinateur est totalement incapable de traiter autre chose que des nombres binaires. Toute information d'un autre type doit être convertie, ou codée, **en format binaire**. Ce procédé est vrai non seulement pour les données que l'on souhaite traiter (les textes, les images, les sons, les nombres, etc.), mais aussi pour les programmes, c'est-à-dire les séquences d'instructions que l'on va fournir à la machine pour lui dire ce qu'elle doit faire.

Le seul *langage* que l'ordinateur puisse *comprendre* est donc très éloigné de ce que nous utilisons nous-mêmes. C'est une longue suite de 1 et de 0, appelés *bits*, souvent traités par groupes de 8 (les *octets*), 16, 32, ou même 64. Ce *langage machine* est évidemment presque incompréhensible pour nous. Pour *parler* à un ordinateur, il nous faudra utiliser des systèmes de traduction automatiques, capables de convertir en nombres binaires des suites de caractères formant des mots-clés.

Le système de traduction proprement dit s'appellera **interpréteur** ou bien **compilateur**, suivant la méthode utilisée pour effectuer la traduction. On appellera **langage de programmation** un ensemble de mots-clés associé à un ensemble de règles très précises indiquant comment on peut assembler ces mots pour former des *phrases* que l'interpréteur ou le compilateur pourra ensuite traduire en langage machine.

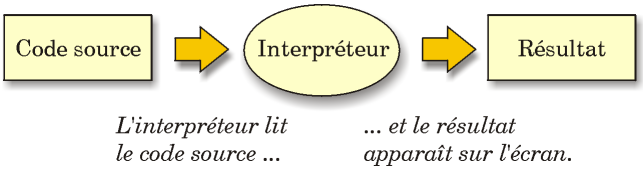
Suivant son niveau d'abstraction, on pourra dire d'un langage qu'il est **de bas niveau** (ex : *assembleur*) ou **de haut niveau** (ex : *Pascal*, *Perl*, *Java*, *Python*, *C++*...). Un langage de bas niveau est constitué d'instructions très élémentaires, *proches de la machine*, alors qu'un langage de haut niveau comporte des instructions plus abstraites, plus *puissantes*. Cela signifie que chacune de ces instructions pourra être traduite par l'interpréteur ou le compilateur en un grand nombre d'instructions machine élémentaires.

Le langage que vous avez allez apprendre dans ce cours est *Python*. Il s'agit d'un langage de haut niveau, dont la traduction en code binaire est complexe et prend donc toujours un certain temps. Cela pourrait paraître un inconvénient. En fait, les avantages que présentent les langages de haut niveau sont énormes : il est beaucoup plus facile d'écrire un programme dans un langage de haut niveau ; l'écriture du programme prend donc beaucoup moins de temps, la probabilité d'y faire des fautes est nettement plus faible, la maintenance (c'est-à-dire l'apport de modifications ultérieures) et la recherche des erreurs (les **bugs**) sont grandement facilitées. De plus, un programme écrit dans un langage de haut niveau sera souvent **portable**, c'est-à-dire qu'on pourra le faire fonctionner sans guère de modifications sur des machines ou des systèmes d'exploitation différents. Un programme écrit dans un langage de bas niveau ne peut jamais fonctionner que sur un seul type de machine : pour qu'une autre l'accepte, il faut le réécrire entièrement.

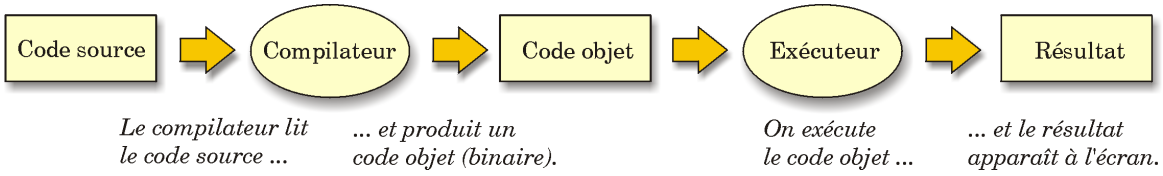
3.1 Compilation et interprétation

Le programme tel que nous l'écrivons à l'aide d'un éditeur (une sorte de traitement de texte) est appelé **code source**. Comme déjà signalé plus haut, il existe deux techniques principales pour effectuer la traduction d'un tel code source en code binaire exécutable par la machine : l'**interprétation** et la **compilation**.

- Dans la technique appelée **interprétation**, le logiciel interpréteur doit être utilisé chaque fois que l'on veut faire fonctionner le programme. Dans cette technique, chaque ligne du programme source est traduite au fur et à mesure en quelques instructions du langage machine, qui sont ensuite directement exécutées.



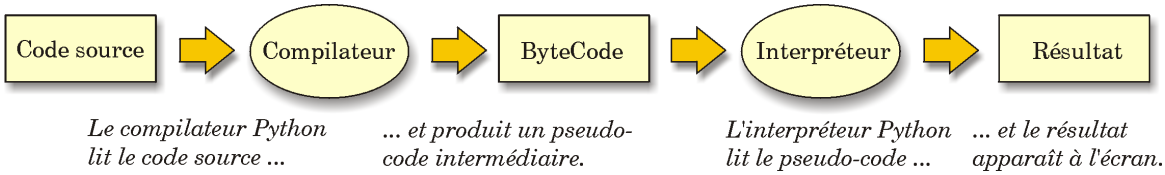
- La **compilation** consiste à traduire la totalité du texte source en une fois. Le logiciel compilateur lit toutes les lignes du programme source et produit une nouvelle suite de codes que l'on appelle **code objet**. Celui-ci peut alors être exécuté et est conservé dans un fichier, appelé **fichier exécutable**.



L'interprétation est idéale lorsque l'on est en phase d'apprentissage du langage, ou en cours d'expérimentation sur un projet. Avec cette technique, on peut en effet tester immédiatement toute modification apportée au programme source, sans passer par une phase de compilation qui demande toujours du temps.

Par contre, lorsqu'un projet comporte des fonctionnalités complexes qui doivent s'exécuter rapidement, la compilation est préférable : il est clair en effet qu'un programme compilé fonctionnera toujours nettement plus vite que son homologue interprété, puisque dans cette technique l'ordinateur n'a plus à (re)traduire chaque instruction en code binaire avant qu'elle puisse être exécutée.

Certains langages modernes tels que Python ou Java tentent de combiner les deux techniques afin de retirer le meilleur de chacune. Lorsque vous lui fournissez un programme source, Python commence par le compiler pour produire un code intermédiaire, similaire à un langage machine, que l'on appelle **bytecode**, lequel sera ensuite transmis à un interpréteur pour l'exécution finale. Du point de vue de l'ordinateur, le *bytecode* est très facile à interpréter en langage machine. Cette interprétation sera donc beaucoup plus rapide que celle d'un code source.



Les avantages de cette méthode sont appréciables :

- Le fait de disposer en permanence d'un interpréteur permet de tester immédiatement n'importe quel petit morceau de programme. On pourra donc vérifier le bon fonctionnement de chaque composant d'une application au fur et à mesure de sa construction.
- L'interprétation du *bytecode* compilé n'est pas aussi rapide que celle d'un véritable code binaire, mais elle est très satisfaisante pour de très nombreux programmes, y compris graphiques.
- Le *bytecode* est portable. Pour qu'un programme Python ou Java puisse s'exécuter sur différentes machines, il suffit de disposer pour chacune d'elles d'un interpréteur adapté.

Tout ceci peut vous paraître un peu compliqué, mais la bonne nouvelle est que tout est pris en charge automatiquement par l'environnement de développement de Python. Il vous suffira d'entrer vos commandes au clavier, de frapper <`Enter`>, et Python se chargera de les compiler et de les interpréter pour vous.

4 Mise au point d'un programme - Recherche des erreurs (debug)

La programmation est une démarche très complexe, et comme c'est le cas dans toute activité humaine, on y commet de nombreuses erreurs. Pour des raisons anecdotiques, les erreurs de programmation s'appellent des **bugs** (ou *bogues*, en Français) [1], et l'ensemble des techniques que l'on met en œuvre pour les détecter et les corriger s'appelle **debug** (ou *débogage*).

En fait, il peut exister dans un programme trois types d'erreurs assez différentes, et il convient que vous appreniez à bien les distinguer.

4.1 Erreurs de syntaxe

Python ne peut exécuter un programme que si sa syntaxe est parfaitement correcte. Dans le cas contraire, le processus s'arrête et vous obtenez un message d'erreur. Le terme **syntaxe** se réfère aux règles que les auteurs du langage ont établies pour la structure du programme.

Tout langage comporte sa syntaxe. Dans la langue française, par exemple, une phrase doit toujours commencer par une majuscule et se terminer par un point. ainsi cette phrase comporte deux erreurs de syntaxe

Dans les textes ordinaires, la présence de quelques petites fautes de syntaxe par-ci par-là n'a généralement pas d'importance. Il peut même arriver (en poésie, par exemple), que des fautes de syntaxe soient commises volontairement. Cela n'empêche pas que l'on puisse comprendre le texte.

Dans un programme d'ordinateur, par contre, la moindre erreur de syntaxe produit invariablement un arrêt de fonctionnement ainsi que l'affichage d'un message d'erreur. Au cours des premières semaines de votre carrière de programmeur, vous passerez certainement pas mal de temps à rechercher vos erreurs de syntaxe. Souvenez-vous que **tous** les détails ont de l'importance. Il faudra en particulier faire très attention à la **casse** (c'est-à-dire l'emploi des majuscules et des minuscules) et à la **ponctuation**. Toute erreur à ce niveau peut modifier considérablement la signification du code, et donc le déroulement du programme.

4.2 Erreurs sémantiques

Le second type d'erreur est l'erreur sémantique ou erreur de logique. S'il existe une erreur de ce type dans un de vos programmes, celui-ci s'exécute parfaitement, en ce sens que vous n'obtenez aucun message d'erreur, mais le résultat n'est pas celui que vous attendiez : vous obtenez autre chose.

En réalité, le programme fait exactement ce que vous lui avez dit de faire. Le problème est que ce que vous lui avez dit de faire ne correspond pas à ce que vous vouliez qu'il fasse. La séquence d'instructions de votre programme ne correspond pas à l'objectif poursuivi. La sémantique (la logique) est incorrecte.

Rechercher des fautes de logique peut être une tâche ardue. Il vous faudra analyser patiemment ce qui sort de la machine et tâcher de vous représenter une par une les opérations qu'elle a effectuées, à la suite de chaque instruction.

1. *bug* est à l'origine un terme anglais servant à désigner de petits insectes gênants, tels les punaises. Les premiers ordinateurs fonctionnaient à l'aide de *lampes* radios qui nécessitaient des tensions électriques assez élevées. Il est arrivé à plusieurs reprises que des petits insectes s'introduisent dans cette circuiterie complexe et se fassent électrocuter, leurs cadavres calcinés provoquant alors des court-circuits et donc des pannes incompréhensibles.

4.3 Erreurs à l'exécution

Le troisième type d'erreur est l'erreur en cours d'exécution (**Run-time error**), qui apparaît seulement lorsque votre programme fonctionne déjà, mais que des circonstances particulières se présentent (par exemple, votre programme essaie de lire un fichier qui n'existe plus). Ces erreurs sont également appelées des **exceptions**, parce qu'elles indiquent que quelque chose d'exceptionnel s'est produit. Vous rencontrerez davantage ce type d'erreurs lorsque vous programmerez des projets de plus en plus volumineux, et vous apprendrez plus loin dans ce cours qu'il existe des techniques particulières pour les gérer.

4.4 Recherche des erreurs et expérimentation

L'une des compétences les plus importantes à acquérir au cours de votre apprentissage est celle qui consiste à **déboguer** efficacement un programme. Il s'agit d'une activité intellectuelle parfois énervante mais toujours très riche, dans laquelle il faut faire montre de beaucoup de perspicacité.

Cette activité s'apparente au travail expérimental en sciences. Vous vous faites une première idée de ce qui ne va pas, vous modifiez votre programme et vous essayez à nouveau. Vous avez émis une hypothèse, qui vous permet de prédire ce que devra donner la modification. Si la prédiction se vérifie, alors vous avez progressé d'un pas sur la voie d'un programme qui fonctionne. Si la prédiction se révèle fausse, alors il vous faut émettre une nouvelle hypothèse.

Pour certaines personnes, *programmer* et *déboguer* signifient exactement la même chose. Ce qu'elles veulent dire par là est que l'activité de programmation consiste en fait à modifier, à corriger sans cesse un même programme, jusqu'à ce qu'il se comporte finalement comme vous le vouliez. L'idée est que la construction d'un programme commence toujours par une ébauche qui fait déjà quelque chose (et qui est donc déjà déboguée), à laquelle on ajoute couche par couche de petites modifications, en corrigeant au fur et à mesure les erreurs, afin d'avoir de toute façon à chaque étape du processus un programme qui fonctionne.

Par exemple, vous savez que Linux est un système d'exploitation (et donc un gros logiciel) qui comporte des milliers de lignes de code. Au départ, cependant, cela a commencé par un petit programme simple que Linus Torvalds avait développé pour tester les particularités du processeur *Intel 80386*. D'après Larry Greenfield (*The Linux user's guide* , beta version 1) : « L'un des premiers projets de Linus était un programme destiné à convertir une chaîne de caractères AAAA en BBBB. C'est cela qui plus tard finit par devenir Linux ! ».

Ce qui précède ne signifie pas que nous voulions vous pousser à programmer par approximations successives, à partir d'une vague idée. Lorsque vous démarrerez un projet de programmation d'une certaine importance, il faudra au contraire vous efforcer d'établir le mieux possible un **cahier des charges** détaillé, lequel s'appuiera sur un plan solidement construit pour l'application envisagée.

5 Les 5 règles d'or de la programmation

Programmer ne suffit pas. Il faut **bien** programmer de telle sorte qu'une personne désirant réutiliser votre programme puisse le faire facilement. En d'autres termes, vos programmes doivent être **compréhensible** pour votre relecteur. La première chose à faire consistera à décomposer un problème compliqué en plusieurs sous-problèmes simples, c'est-à-dire à décomposer votre programme en plusieurs sous-programmes. Voici ensuite les règles à respecter :

- **Règle 1.**
  Ne pas écrire de longs sous-programmes (pas plus de 10-15 lignes de code).
- **Règle 2.**
  Chaque sous-programme doit avoir un objectif clair.
- **Règle 3.**
  Ne jamais utiliser les fonctionnalités du langage dont on n'est pas sûr du résultat ou du rôle.
- **Règle 4.**
  Eviter à tout prix le copier/coller, source d'innombrables erreurs.
- **Règle 5.**
  Ecrire des commentaires pour expliquer les parties les plus subtiles du programme.

6 Glossaire du chapitre

La liste suivante répertorie dans l'ordre alphabétique les principaux termes introduits dans ce chapitre. La connaissance de leur définition est indispensable à la suite de votre apprentissage :

ALU : organe de l'ordinateur chargé d'effectuer les calculs, le plus souvent inclus dans le microprocesseur.

bit : chiffre de la numération binaire pouvant prendre que deux valeurs, notées conventionnellement 0 et 1.

bug : erreur intervenant dans un programme.

casse : distinction entre les lettres minuscules et majuscules.

code binaire : système de numération utilisant la base 2.

code exécutable : code machine généré à la compilation d'un code source.

code source : ensemble d'instructions écrites dans un langage de haut niveau permettant d'obtenir un programme informatique une fois compilé.

compilateur : outil informatique capable de traduire un programme écrit dans un langage de haut niveau en un langage de bas niveau avant de l'exécuter.

CPU : appelé également processeur, ce composant de l'ordinateur exécute les programmes informatiques.

debugging : opération visant à trouver et corriger un bug.

erreur de sémantique : erreur engendrant un comportement du programme différent de ce que le programmeur attendait.

erreur de syntaxe : erreur grammaticale empêchant l'interprétation ou la compilation d'un programme.

exception : erreur détectée à l'exécution du programme lorsque des circonstances exceptionnelles se présentent.

interpréteur : outil informatique capable d'exécuter dynamiquement un programme écrit dans un langage de haut niveau.

langage de bas niveau : langage de programmation difficile à manier car rudimentaire et abstrait qui oblige le programmeur à se soucier de concepts proches du fonctionnement de la machine.

langage de haut niveau : langage de programmation proche du langage humain permettant à un programmeur de s'abstraire des détails inhérents au fonctionnement de l'ordinateur.

langage de programmation : langage informatique composé de mots-clés et de règles grammaticales permettant à un être humain de former des phrases que l'interpréteur ou le compilateur pourra traduire en langage machine exploitable par l'ordinateur.

langage machine : suite de bits interprétée par le processeur d'un ordinateur exécutant un programme informatique.

logiciel : ensemble d'informations relatives à des traitements effectués automatiquement par un appareil informatique. Y sont inclus les instructions de traitement, regroupées sous forme de programmes, des données et de la documentation. Le tout est stocké sous forme d'un ensemble de fichiers dans une mémoire.

octet : unité de mesure de données informatiques composée de 8 bits.

périphérique : dispositif connecté à un système informatique dans le but de lui rajouter des fonctionnalités.

portabilité : capacité d'un programme à fonctionner dans différents environnement d'exécution.

programme informatique : suite d'opérations pré-déterminées destinées à être exécutées de manière automatique par un appareil informatique en vue d'effectuer des travaux, des calculs arithmétiques ou logiques, ou simuler un déroulement.

sémantique : étude de la signification d'un programme informatique.

syntaxe : ensemble des règles grammaticales d'un langage de programmation établies pour structurer un programme.

système d'exploitation : ensemble de programmes central d'un appareil informatique qui sert d'interface entre le matériel et les logiciels applicatifs. Un système d'exploitation peut servir à coordonner l'utilisation du ou des processeur(s), à accorder un certain temps pour l'exécution de chaque processus, à réserver de l'espace dans les mémoires pour les besoins des programmes et à organiser le contenu des disques durs en fichiers et répertoires. Il peut aussi servir à créer l'interface homme-machine des différents programmes et à l'envoyer à l'écran ainsi qu'à recevoir les manipulations effectuées par l'utilisateur via le clavier, la souris ou d'autres périphériques et les transmettre aux différents programmes.