Qu'est-ce que la «Programmation» ?

D'une manière générale,
l'objectif de la programmation est de
permettre l'**automatisation** d'un certain nombre de tâches,
à l'aide de machines particulières:

les **automates programmables**

Automates programmables

☞Un **automate** est un dispositif capable d'assurer, sans intervention humaine, un enchaînement d'opérations, correspondant à la réalisation d'une tâche donnée.
Comme exemple d'automates, on peut citer:
la montre, le réveil-matin, le «ramasse-quilles» (du bowling).

☞ L'automate est dit «**programmable**» lorsque l'enchaînement d'opérations effectuées peut être modifié à volonté, pour permettre un changement de la tâche à réaliser. Dans ce cas, la **description de la tâche** à accomplir se fait par le biais d'un **programme**, c'est-à-dire une séquence d'instructions et de données susceptibles d'être traitées (i.e. «comprises» et «exécutées») par l'automate.

Comme exemples d'automates programmables, citons:

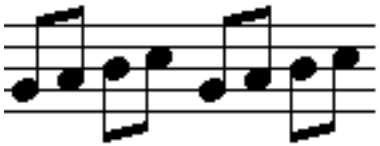
le métier à tisser Jacquard,
l'orgue de barbarie,
…,
et naturellement l'ordinateur.

Exemple d'automate programmable

automate

programme

tâche



Programmer c'est...

En résumé, programmer c'est donc
décomposer la tâche à automatiser sous la forme
d'une **séquence d'instructions et de données**
adaptées à l'automate utilisé.

Dès lors, voyons quelles sont ces
instructions et données «adaptées»,
dans le cas où l'automate programmable est
un ordinateur.

Instructions et langage machine

Nous l'avons vu, un ordinateur est, en schématisant à l'extrême, constitué:

- d'un [**micro**]**processeur**, capable d'exécuter (réaliser) un jeu donné d'opérations élémentaires.
- d'une **mémoire centrale**, dans laquelle sont stockées les données en cours de traitement, ainsi que le programme lui-même;
- de bus, ports d'entrées-sorties et périphériques;

Le **jeu d'instructions** (~ langage) que l'ordinateur est capable de traiter est donc tout naturellement déterminé par le processeur.

☞ Les instructions comprises par un processeur sont appelées les ***instructions machine*** de ce processeur.

☞ Le langage de programmation qui utilise ces instructions est appelé le ***langage machine***.

Langage machine: format interne

☞Pour que les opérations et les données manipulées soient compréhensibles par le processeur, elles doivents être exprimées dans le seul format qu'il peut prendre en compte:

le *format interne*,

qui est [presque[1]] toujours un ***format binaire***.

⇨ Ce format n'utilise que deux *symboles élémentaires* (généralement «*0*» et «*1*») appelés «*bits*»[2].

☞ Mettre en correspondance la représentation externe des opérations et des données avec leur représentation sous la forme d'une séquence de bits s'appelle le **codage**.

⇨ Le codage permet donc à l'ordinateur de manipuler des données de nature et de type divers sous la forme d'une représentation unique.

1. Il y eu quelques tentatives pour construire des machines supportant d'autre formats, en particulier en URSS le format ternaire. Si au niveau électrique il y a un intérêt certain à disposer ainsi de trois états (actif, inactif, neutre), cela ne concerne pas le niveau logique. L'avantage de la représentation binaire est qu'elle est techniquement facile à réaliser (au moyen de bistables), que les opérations fondamentales sont relativement simple à effectuer sous forme de circuits logiques et que de plus, comme l'a démontré Shannon, tous les calculs logiques et arithmétiques peuvent être réalisés à partir de l'arithmétique binaire.
2. Contraction de l'expression anglaise «binary digit».

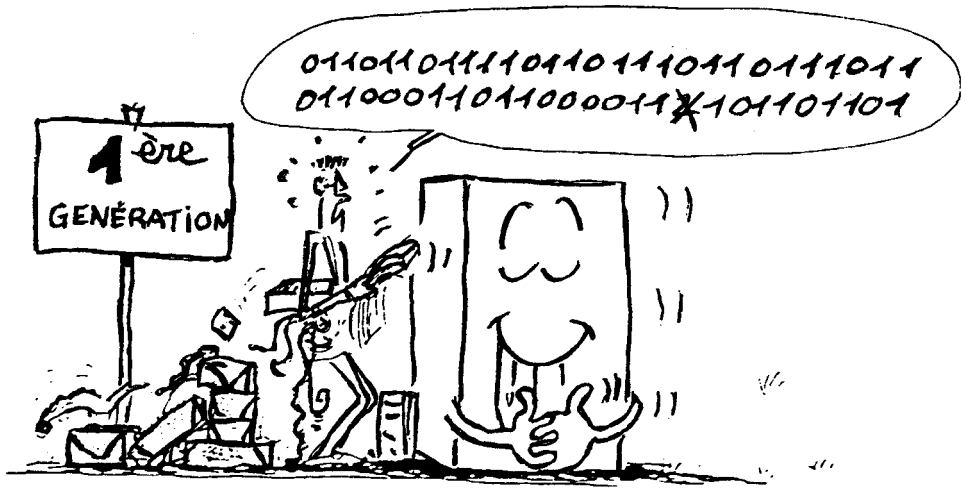
Les générations de langage

Au cours du temps, différents types de langages de programmation firent leur appartition,
et furent regroupés [entre autres] en **générations**

Bien qu'il s'agisse d'une classification différente de celle des machines, les générations de langages sont, du moins pour les premières, liées sur le plan chronologique aux générations de machines, et aux performances de leurs composants

1ère Génération

Au début était le **langage** [de la] **machine**.



Dans le cas des ordinateurs de premières générations, la mémoire n'est pas encore utilisée pour le stockage du programme.

Celui-ci est placé sur des cartes et/ou rubans perforés, voir même totalement **volatile**,[3] i.e. entré à la main à l'aide de panneaux de commutateurs.

On l'imagine facilement, la production des programmes était fastidieuse, et leur relecture et modification presque impossible.
Par ailleurs, les programmes écrits pour une machine ne pouvaient pas être exécutés sur une autre.

⟹ Remarquons que la technique des cartes ou rubans perforés a perduré jusqu'à nos jours (dans des secteurs très ciblés): c'est un moyen de stockage de l'information passablement souple, qui offre surtout une très **grande fiabilité en milieu hostile**.

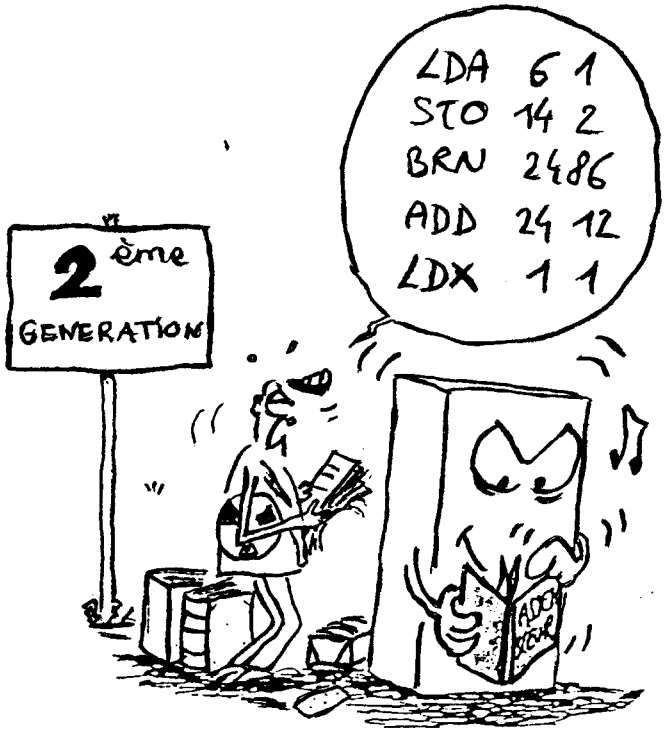
3. Notamment en ce qui concerne l'amorçage du système, et la connexion/prise en charge du système de lecture de cartes.

2ème génération (1)

Puis vint le **langage d'assemblage**.

Afin de rendre les programmes plus faciles à écrire et à corriger, on a rapidement songé à remplacer les séquences de bits par des symboles, les instructions machines étant codées par des *mnémoniques* et les données par les caractères alphanumériques associés.

Le nouveau langage ainsi produit s'appelle un **langage d'assemblage**, ou **langage assembleur**.



Exemple

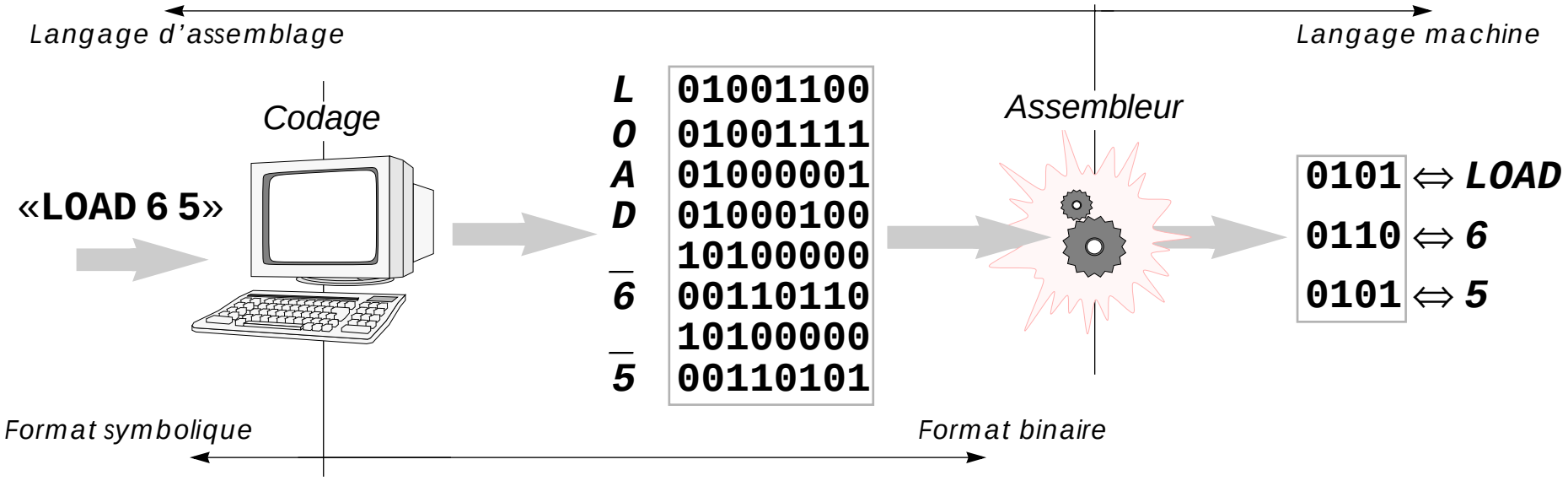
0101	0110	0101
↓	↓	↓
«LOAD»	«6»	«5»

2ème génération (2)

☞ L'apparition de périphériques de saisie (clavier) et de visualisation (écran) permet une écriture – respectivement lecture – plus ergonomique des séquences de 0 et de 1, sous la forme de séquences alphanumériques.

☞ Avec l'idée de *Von Neumann* d'enregistrer dans la mémoire le programme à exécuter ainsi que ses données, il devient également envisageable de faire subir à ce programme des traitements avant son exécution (prétraitements).

⇨ La représentation en mémoire du programme en langage d'assemblage devient ainsi elle-même une donnée, pouvant être traitée par un programme spécifique nommé «assembleur», dont le rôle est sa traduction en langage machine.



Exemple de langage d'assemblage (1)

Exemple de langage d'assemblage: *(machine à mots de 4 bits et alignement sur 12)*

Mnémonique	Code	Syntaxe	Définition
NOP	0000	NOP	Opération neutre (ne fait rien)
CMP	0001	CMP [*adresse*] [*valeur*]	Compare la valeur stockée à l'adresse [*addresse*] avec la valeur [*valeur*]. L'exécution se poursuit à l'adresse courante + 1 en cas d'égalité, + 2 sinon.
DECR	0011	DECR [*adresse*]	Décrémente de 1 la valeur stockée en [*addresse*]
JUMP	0100	JUMP [*saut*]	L'exécution se poursuit à l'adresse courante + [*saut*]
LOAD	0101	LOAD [*adresse*] [*valeur*]	Charge la valeur [*valeur*] à l'adresse [*adresse*]
END	1111	END	Fin d'exécution

Codage des données: *(codage sur 4 bits, en complément à 2)*

Données	Code machine
-3	1101
0	0000
2	0010
3	0011
5	0101
6	0110

Exemple de langage d'assemblage (2)

Programme
(langage assembleur)

```
LOAD   6 5    ; ...
CMP    6 0    ; ...
JUMP   +3
DECR   6
JUMP   -3
END
```

commentaires

mnémoniques
instructions
opérateurs

données
arguments
opérandes

Assembleur

Programme
(langage machine)

segment de code

00	010101100101
01	000101100000
02	010000110000
03	001101100000
04	010011010000
05	111100000000
06	000000000000

adresses

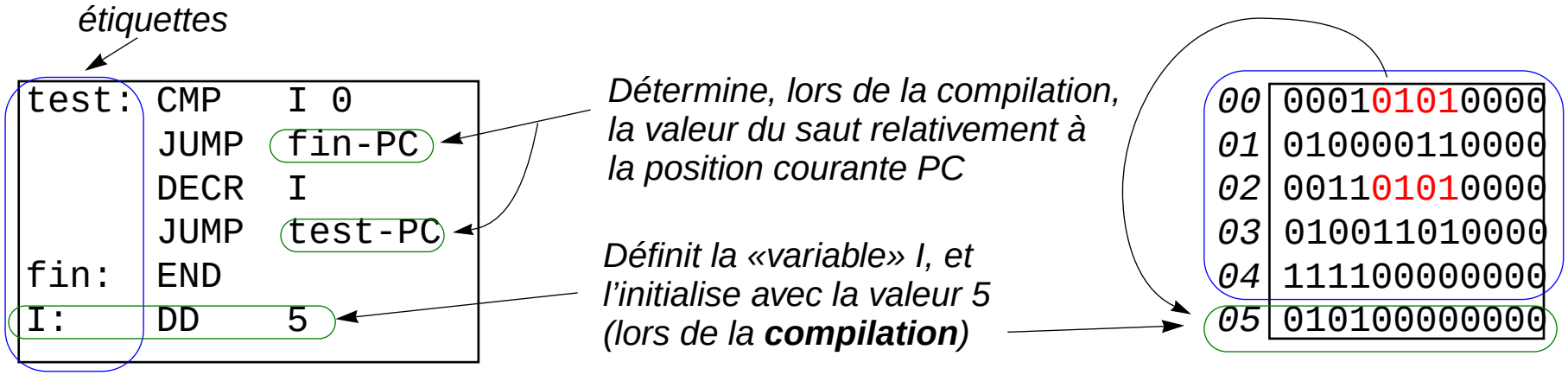
segment de données

Exemple de langage d'assemblage (3)

☞Un langage d'assemblage tel que celui décrit précédemment est appelé **langage d'assemblage pur**, c'est à dire qu'il y a biunivocité entre les instructions machine et les mnémoniques (le code source en langage d'assemblage comportent le même nombre d'élements (d'instructions) que le code machine résultant).

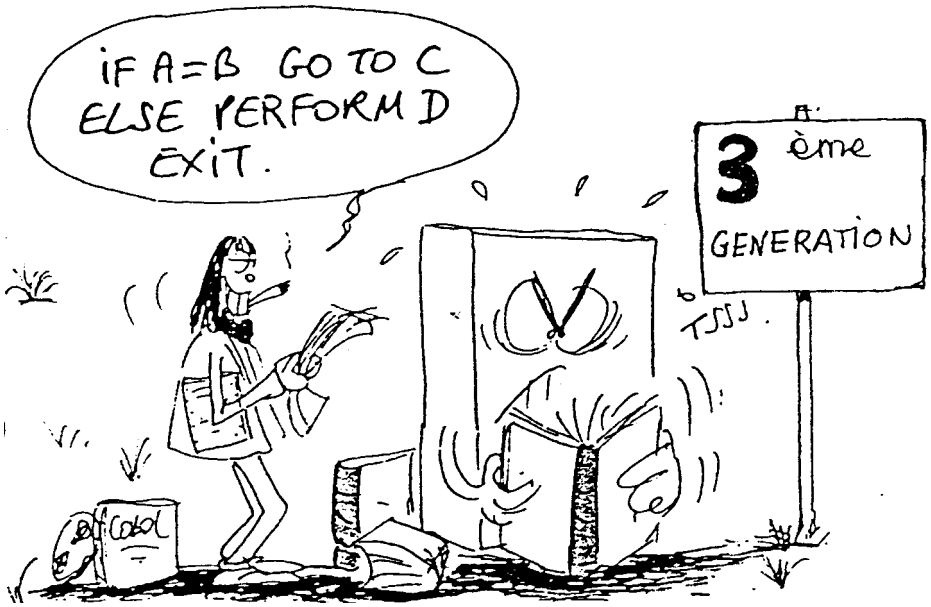
Les tout premiers assembleurs étaient *purs*, mais cette «propriété» disparut rapidement, avec l'apparition des langages offrant des *macros-commandes* (pour définir une seule fois des portions de codes fréquentes) ainsi que des *pseudo-instructions* (réservation & initialisation de mémoire, chargement de modules séparés,...).

La traduction effectuée par les assembleurs correspondants a permis de rendre le code **portable** (i.e. réutilisable, moyennant son assemblage, sur une autre machine), en masquant la couche matérielle (**abstraction de la machine**).



3ème génération

Dans la lancée apparurent les **langages évolués**.



Le but est de fournir au programmeur la possibilité d'utiliser des **instructions** et des **structures de données de** plus **haut niveau**, afin de rendre plus facile l'écriture des programme et d'améliorer la productivité.

⇨ Ces langages sont plus accessibles, plus proches de notre manière de **penser et de conceptualiser les problèmes.**

Les langages évolués sont nombreux, et il ne cesse d'en apparaître de nouveaux. Citons à titre d'exemple: BASIC, Fortran, Cobol, C, Pascal, Modula-2, Ada, OccamII, Portal, ...

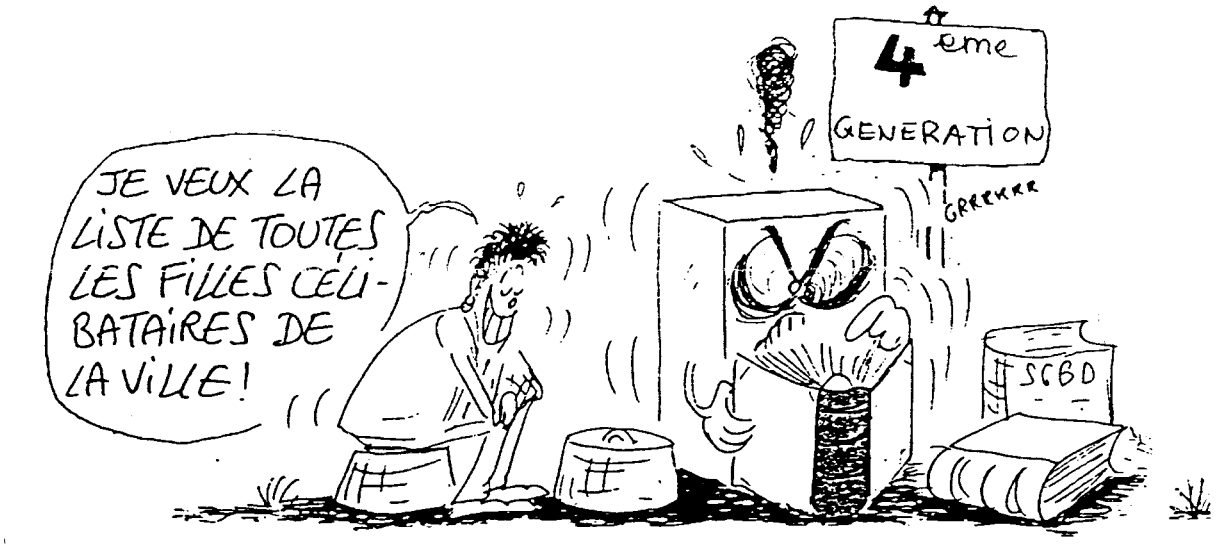
Exemples de langages évolués

Exemples de langage de programmation de haut niveau:

«Vieux» BASIC	*ANSI C*
`1  REM Ceci est un commentaire`	`/* Ceci est un commentaire */`
`4  LET N = 5`	`int n = 5;`
`5  IF (N = 0) THEN GOTO 10`	`while (n != 0) do --n;`
`6  ELSE LET N = N-1`	
`7  GOTO 5`	
`10 END`	

4ème génération

Parallèlement aux langages évolués, des langages encore **plus spécialisés**, voir des **méta-langages**, ont fait leur apparition.



On peut par exemple citer les langages d'intelligence artificielle (Lisp, Prolog), les langages objets (Smalltalk, Eiffel), certains langages de gestion (L4G), ...

Langage de programmation (1)

Cependant, si nous utilisons des intructions sophistiquées, comment les rendre compréhensible pour l'ordinateur qui doit les exécuter ?

☞ Nous l'avons vu, une solution est de disposer d'un programme capable de **transformer** une séquence d'instructions de haut niveau (aussi appelée **code source**) en une séquence d'instructions machine (aussi appelée **code objet**[4] ou **binaire**).

Les programmes qui convertissent un programme quelconque écrit dans un langage source en un programme écrit dans un langage cible sont appelés ***traducteurs***.

Selon ses caractéristiques, un programme *traducteur* est appelé un ***assembleur***, un ***compilateur***[5] ou un ***interpréteur***; l'ensemble des instructions et données de plus haut niveau que le traducteur est capable de traiter constitue un **langage de programmation.**

4. Ce qui n'a rien à voir avec la Programmation [Orientée] Objet !

5. Equivalent d'un assembleur, mais fonctionnant avec un langage source évolué (non assembleur).

Langage de programmation (2)

Un **langage de programmation** est
donc un moyen formel permettant de décrire
des traitements (i.e. des tâches à effectuer) sous la forme
de programmes (i.e. de séquences d'instructions et de données
de haut niveau, c'est-à-dire compréhensibles par le programmeur)
et pour lequel il existe un traducteur permettant l'exécution
effective des programmes par un ordinateur.

Les aspects syntaxiques (règles d'écriture des programmes)
et sémantiques (définition des instructions) d'un langage
de programmation doivent être spécifiés de manière précise,
généralement dans un manuel de référence.

Interpréteur v.s. Compilateur (1)

Il est important de bien comprendre la différence entre traduction effectuée par un assembleur ou un compilateur et celle réalisée un interpréteur.

⇨ Les **compilateurs** et **assembleurs traduisent** tous deux les programmes **dans leur ensemble**: tout le programme doit être fourni au compilateur pour la traduction. Une fois cette traduction effectuée, son résultat (code objet) peut être soumis au processeur pour traitement.

Un langage de programmation pour lequel un compilateur est disponible est appelé un **langage compilé**.

⇨ Les **interpréteurs** traduisent les programmes instruction par instruction, et soumettent immédiatement chaque instruction traduite au processeur, pour exécution.

Un langage de programmation pour lequel un interpréteur est disponible est appelé un **langage interpreté**.

Remarquons que pour certains langages (par exemple Lisp), il existe à la fois des compilateurs et des interpréteurs.

Interpréteur v.s. Compilateur (2)

Naturellement, chacune de ces techniques possède des avantages et des inconvénients, qui la rend plus ou moins adaptée suivant le contexte:

☞ De manière générale, on peut dire que les langages interprétés sont bien adaptés pour le **développement rapide et le prototypage**:

- le cycle de test est plus court qu'avec les langages compilés,
- il est souvent possible de modifier/rectifier le programme en cours d'exécution (test),
- et ces langages offrent généralement une **plus grande liberté d'écriture**.

Interpréteur v.s. Compilateur (3)

☞ A l'inverse, les langages compilés sont à utiliser de préférence pour les **réalisations opérationnelles**, ou les **programmes de grande envergure**:

- Les programmes obtenus sont **plus efficaces**:

  → d'une part, le compilateur peut effectuer des optimisations plus facilement que l'interpréteur, puisqu'il possède une visibilité globale sur le programme,

  → et d'autre part, l'effort de traduction n'est fait qu'une seule fois, qui plus est en prétraitement.

- Par ailleurs, la visibilité globale offerte au compilateur, allié à une structuration plus rigoureuse et un typage plus ou moins contraint, permet une **meilleur détection des erreurs**, lors de la compilation.

⟹ Remarquons finalement que la compilation permet la diffusion du programme sous sa forme opérationnelle, sans imposer pour autant sa diffusion sous forme conceptuelle (lisible et compréhensible par un humain)

Variété des applications

L'utilisation de l'informatique dans les 3 types d'applications – calcul scientifique, gestion d'informations et commande de processus – à conduit à la production d'une grande diversité de programmes:

- **Petits utilitaires**, «drivers» (librairie de fonctions permettant de piloter un matériel), et autre mini-programmes.

  → Effort: ~ 1 à 2 *personnes* × *mois*, de ~1000 à 10'000 lignes de code (source), équipes d'une à deux personnes.

- **Petits logiciels** («maileur», gestionnaire de fichiers, agenda électronique, ...)

  → Effort: ~ 3 à 12 *personnes* × *mois*, de 1'000 à 50'000 lignes, petites équipes de développement (1 à 5 personnes).

- **Progiciels**, logiciels complet (traitement de textes, commande de central téléphonique,...)

  → Effort: ~ 1 à 10 *personnes* × *ans*, de 10'000 à 1'000'000 lignes, équipes de développement moyennes (de 5 à 20 personnes)

- **Gros systèmes** (aéronautique, systèmes d'exploitation, systèmes experts, ...)

  → Effort: ~ plus de 100 *personnes* × *ans*, plusieurs centaines de millions de lignes de code, grosses équipes de développement (plusieurs milliers de personnes).

Le système d'exploitation

Au fil des années, une spécialisation progressivedes logiciels s'est réalisée:

- logiciels d'application:

  → résolution de problèmes spécifiques (traitement de textes, PAO, tableurs, logiciels de comptabilités, CAO (conception, design et simulation), ...)

- logiciels utilitaires:

  → logiciels qui servent au développement des applications (assembleur, compilateurs, dévermineur, ... , mais aussi gestionnaire de versions, gestionnaire de fenêtres, librairie de dessin, outils de communications...)

- logiciels systèmes (regroupés dans le *système d'exploitation*):

  → présents au cœur de l'ordinateur, ces logiciels sont à la base de toute exploitation, coordonnant les tâches essentielles à la bonne marche du matériel. Nous allons voir plus en détails leurs caractéristiques.

C'est du système d'exploitation que dépend la qualité
de la gestion des ressources (processeur, mémoire, périphériques)
et la convivialité de l'utilisation d'un ordinateur.

Définition d'un système d'exploitation

☞ Le système d'exploitation est l'ensemble des programmes qui se chargent de résoudre les problèmes relatifs à l'exploitation de l'ordinateur.

Plus concrètement, on assigne généralement deux tâches distinctes à un système d'exploitation:

- Gérer les ressources physiques de l'ordinateur

  → assurer l'exploitation **efficace, fiable et économique** des ressources critiques (processeur, mémoire)

- Gérer les informations manipulées par les utilisateurs

  → faciliter le travail des utilisateurs en leur présentant une machine plus simple à exploiter que la machine réelle (concept de *machine virtuelle*)

Apparition des systèmes d'exploitation

Les premières machines étaient dépourvues de système d'exploitation; a cette époque, toute programmation était l'affaire de l'utilisateur.

Dès lors, le «passage en machine» (exécution d'un programme) nécessitait un ensemble d'opérations longues et fastidieuses. Par exemple, lorsque la machine s'arretait (suite à une panne), il fallait à nouveau programmer à la main l'amorcage.[1]

Avec les machines de seconde génération, on commenca à automatiser les opérations manuelles, ce qui améliora l'exploitation des différentes unités.

Pour cela, des programmes spécifiques appelés *moniteurs* ou *exécutifs* firent leur apparition, leur rôle étant d'assurer la bonne marche des opérations (séquencement des travaux des utilisateurs).

⇨ le système d'exploitation était né.

1. Entre autre l'accès à la mémoire secondaire (lecteur de cartes ou de bandes), puis le chargement des utilitaires de bases, tels l'assembleur, et finalement le programme lui-même.

Caractéristiques fondamentales

Il existe aujourd'hui un nombre très important de systèmes d'exploitation (plusieurs centaines). La tendance actuelle est toutefois à la standardisation, en se conformant aux systèmes existant plutôt qu'en en développant de nouveaux.

⇨ un avantage essentiel est la portabilité du code existant.

Classification fréquente des systèmes d'exploitation:

<table>
<tr><th>Mono-tâche</th><th colspan="2">Multi-tâches</th></tr>
<tr><td rowspan="3">A tout instant, un seul programme est exécuté, et un suivant ne démarrera, sauf conditions exceptionnelles, que lorsque le premier aura terminé.</td><td colspan="2">Plusieurs processus (un programme en cours d'exécution) peuvent s'exécuter simultanément (systèmes multi-processeurs) ou en quasi parallélisme (systèmes à temps partagé)</td></tr>
<tr><td>mono-utilisateurs</td><td>multi-utilisateurs</td></tr>
<tr><td>A tout moment présence d'au plus un utilisateur</td><td>Plusieurs utilisateurs peuvent travailler simultanément sur la même machine.</td></tr>
<tr><td>Exemple: DOS</td><td>Exemple: Windows</td><td>Exemple: VMS, Unix</td></tr>
</table>

Structure en couche d'un SE moderne

Les couches permettent de clairement identifier les niveaux d'interdépendance entre les fonctions fournies par le S.E.

Le noyau

Les fonctions principales du noyau (d'un SE multi-tâches) sont:

- Gestion du processeur:
  → reposant sur un allocateur (*dispatcher*) responsable de la répartition du temps processeur entre les différents processus,
  → et un planificateur (*scheduler*) déterminant les processus à activer, en fonction du contexte.

- Gestion des *interruptions*:
  → les *interruptions* sont des signaux envoyés par le matériel, à destination du logiciel, pour signaler un évènement.

- Gestion du multi-tâches:
  → simuler la simultanéité des processus coopératifs (devant se synchroniser pour échanger des données)
  → gérer les accès concurrents aux ressources (fichiers, imprimantes, ...)

Du fait de la fréquence élevée des interventions du noyau, il est nécessaire qu'il réside en permanence et en totalité dans la mémoire centrale. Son codage doit donc être particulièrement soigné, pour être à la fois performant et de petite taille.

Mémoire virtuelle (1)

La mémoire centrale à toujours été une ressource critique: initialement très coûteuse et peu performante (torres magnétiques), elle était de très faible capacité.

Avec l'apparition des mémoires électroniques, la situation s'est bien améliorée, mais comme parallèlement la taille des programmes à considérablement augmenté, la mémoire demeure, même de nos jours, une ressource rare et à économiser.

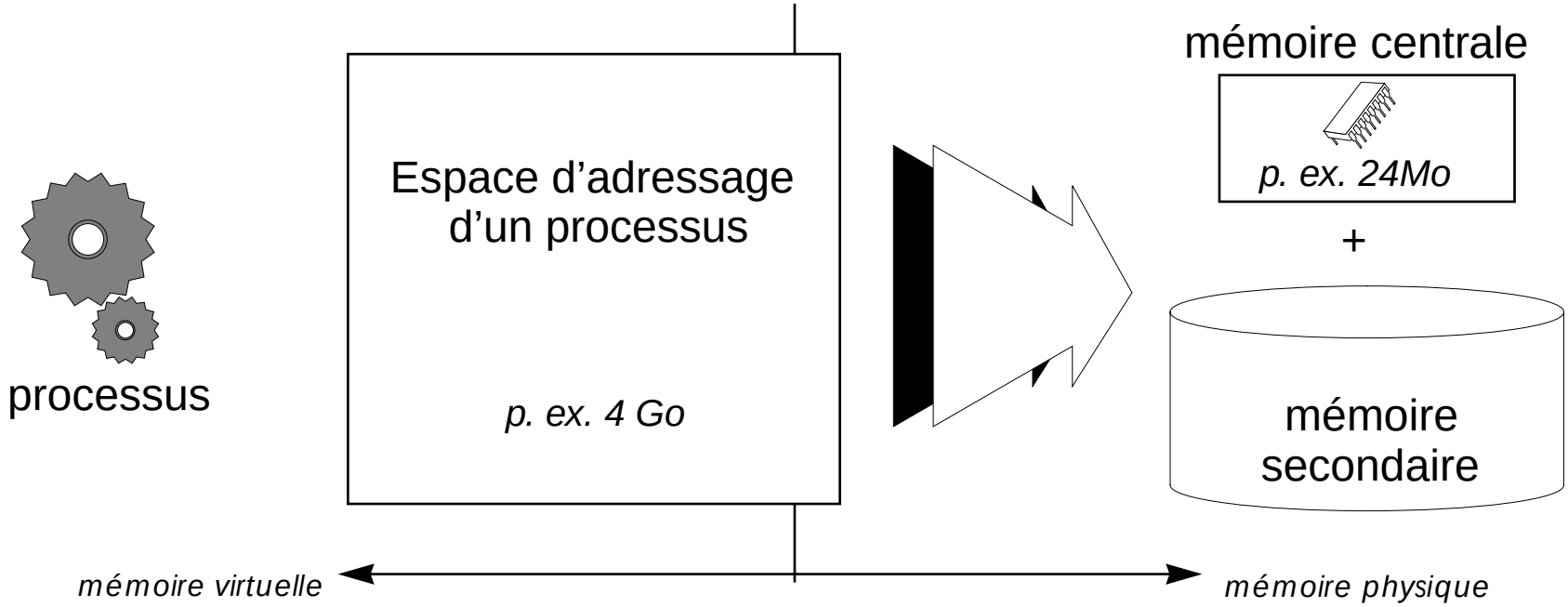
Pour suppléer ce manque crucial de mémoire, l'idée est venue d'utiliser la mémoire secondaire, plus lente mais de beaucoup plus grande capacité, et avec elle apparut le concept de **mémoire virtuelle**.

Mémoire virtuelle (2)

L'idée de mémoire virtuelle est à la fois simple et élégante.

Elle consiste en une **décorellation** entre la **mémoire physique**, présente sur la machine, et celle mise à disposition des processus par le système d'exploitation (la **mémoire virtuelle**, ou **logique**).

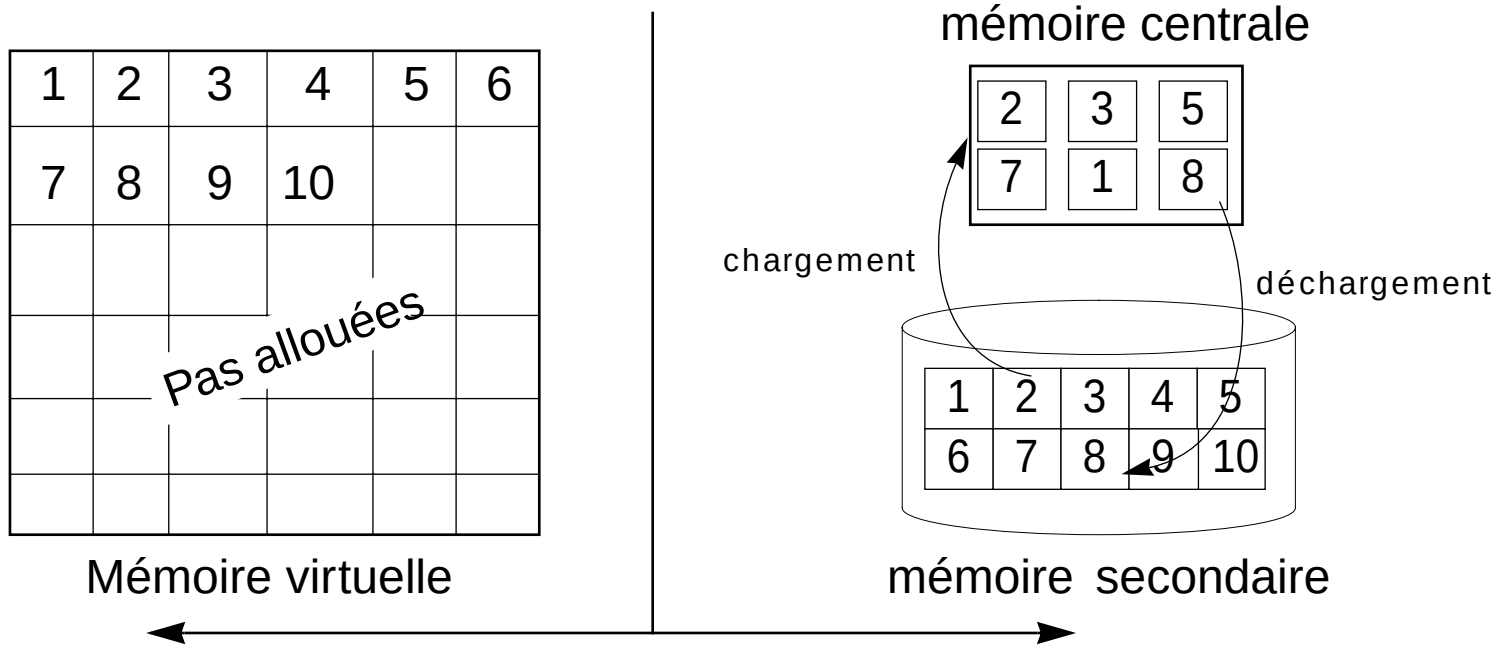
Pour cela, on effectue un traitement séparé des adresses (*virtuelles*) référencées dans un programme et des adresses (*réelles*) de la mémoire présente.



Mémoire virtuelle (3)

Une technique fréquemment utilisée pour mettre en œuvre ce mécanisme est la **pagination**.

La mémoire vue par le processus (**mémoire virtuelle**, aussi grande que le permet le système) est segmentée en pages.
A tout instant, **seul un nombre limité** de ces pages est physiquement présent dans la mémoire centrale, le reste [des pages allouées, i.e. utilisée] est conservé en mémoire secondaire, et **chargé** en mémoire physique au besoin.



Système de Fichiers

Le concept de fichiers est une structure adaptée aux mémoires secondaires et auxilliaires permettant de regrouper des données.

Le rôle d'un système d'exploitation est de donner corps à ce concept de fichiers (les gérer, c'est-à-dire les créer, les détruires, les écrire (modifier) et les lires, en offrant la possibilité de les désigner par des noms symboliques).

Dans le cas de systèmes multi-utilisateurs, il faut en plus assurer la **confidentialité** de ces fichiers, en protégant leur contenu du regard des autres utilisateurs.

Remarquons que cet aspect des systèmes multi-utilisateurs implique l'autentification de l'utilisateur en début de session de travail, généralement au moyen d'un ***login*** (nom d'utilisateur + mot de passe)

Entrées-Sorties

La gestion des entrées-sorties est un domaine délicat
dans la conception d'un système d'exploitation:

il s'agit de permettre le dialogue (échange d'informations)
avec l'extérieur du système.

La tâche est rendue ardue, de part la diversité des périphériques
d'entrées-sorties et les multiples méthodes de codage des informations
(différentes représentations des nombres, des lettres, etc.)

Concrètement, la gestion des E/S implique que le SE mette à disposition
de l'utilisateur des procédures standard pour l'émission et la réception des données,
et offre des traitements appropriés aux multiples conditions d'erreurs susceptibles
de se produire (plus de papier, erreur de disque, débit trop différents, ...)