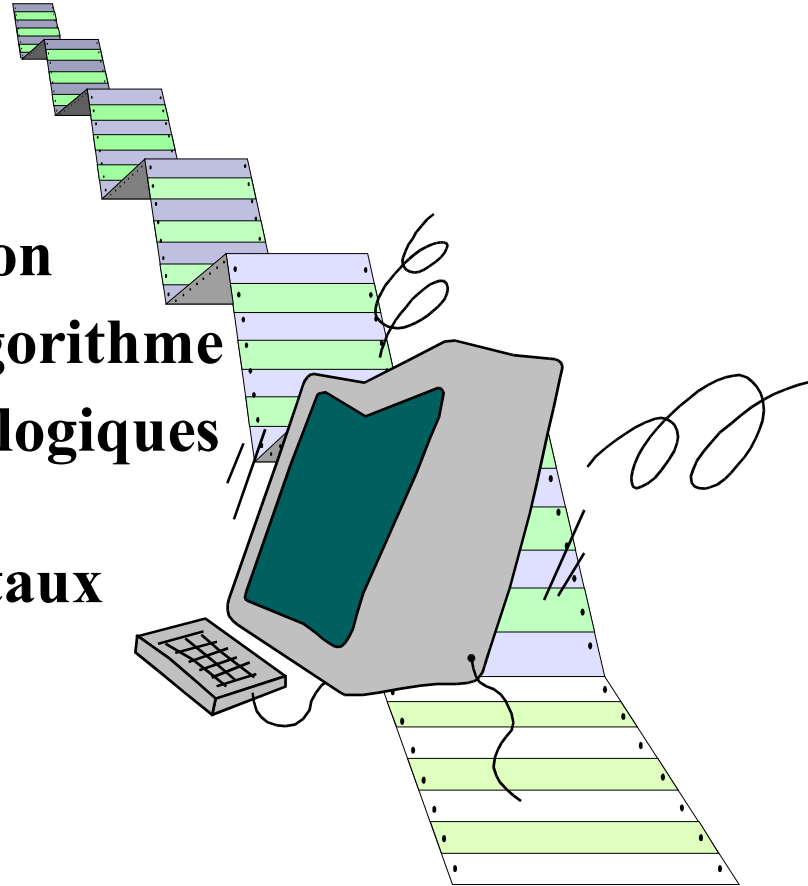


Introduction à la programmation

- **Présenter l'activité de programmation**
- **Introduire et justifier la notion d'algorithme**
- **Donner quelques principes méthodologiques**
 - Diviser pour régner
- **Donner quelques repères fondamentaux**
 - Complexité d'un algorithme
 - Langages

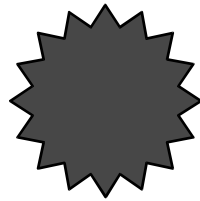


Vous savez compter ! L'ordinateur aussi...

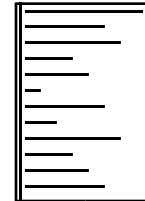
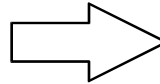
- **Votre programme s'exécute, mais...**
 - Connaissez-vous les mécanismes utilisés ?
 - Etes vous sûr que le résultat soit juste ?
 - Combien de temps devrez vous attendre la fin du calcul ?
 - Y a-t-il un moyen pour obtenir le résultat plus vite ?
 - » Indépendamment de la machine, du compilateur...
- **Un ordinateur ne s'utilise pas comme un boulier !**
 - => Connaître des algorithmes
 - => Apprendre à les construire, les améliorer...



La programmation



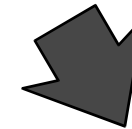
Problème



Programme



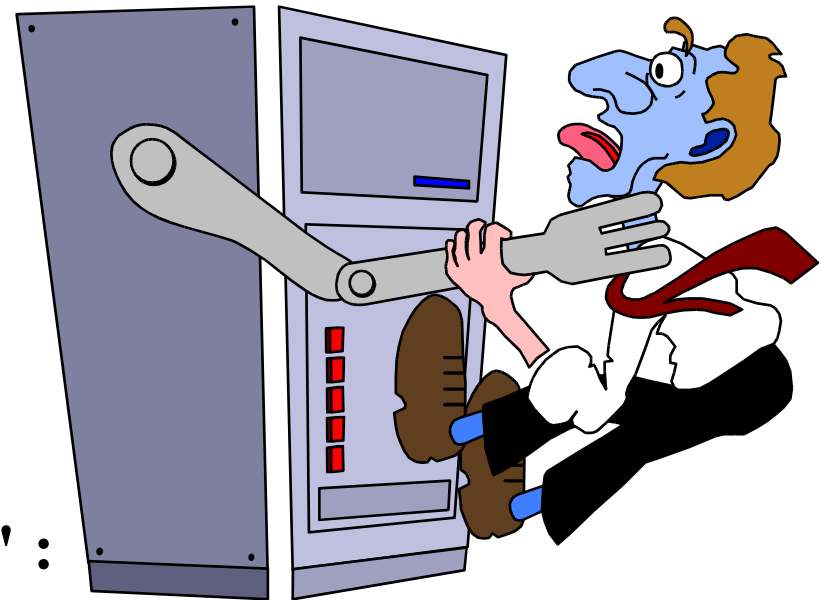
- Question à résoudre par une solution informatique
- Instance d'un problème = entrée nécessaire pour calculer une solution du problème



- Ensemble de données
- Ensemble de résultats
= solution informatique au problème
- Description d' un ensemble d'actions
- Exécution dans un certain ordre

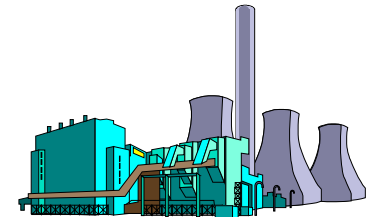
Critères de qualité des programmes

- **Lisibles**
- **Fiables**
- **Maintenables**
- **Réutilisables**
- **Portables**
- **Corrects (preuve)**
- **Efficaces (complexité)**
- **Contraintes "économiques" :**
 - Exécution la plus courte possible
 - Espace mémoire nécessaire le plus petit possible...

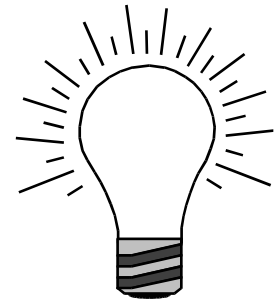


Raisons d'être des méthodes de programmation

- **Augmentation de la taille et de la complexité des logiciels**
 - Travail en équipe
 - Capacité de l'être humain à s'occuper de problèmes simultanément (5 à 9 problèmes)
 - **Nécessité de construire des programmes corrects, vérifiables et modifiables**
 - Conséquences humaines, économiques... de plus en plus coûteuses
- => Méthodologie de conception des programmes**
- Garder la maîtrise de la conception du logiciel
 - Canaliser la créativité



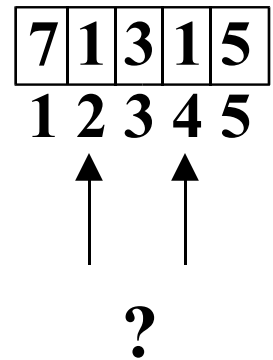
Principes méthodologiques



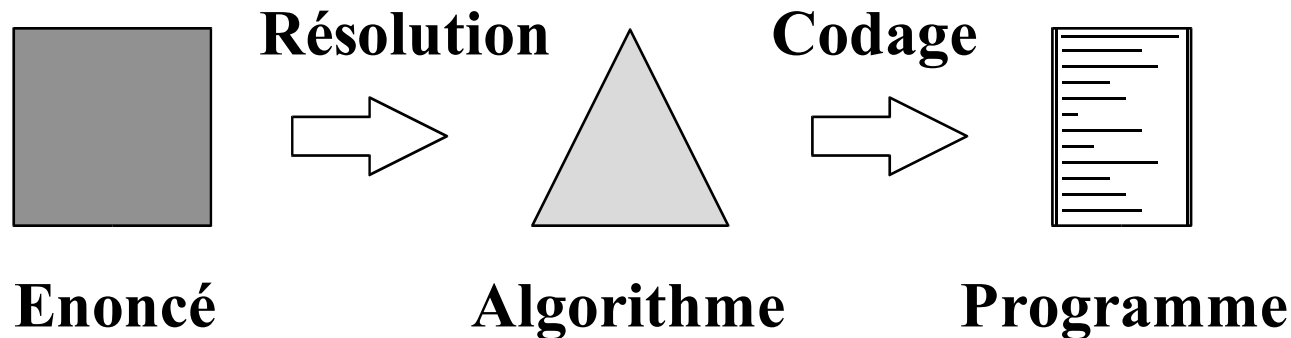
- **Abstraire**
 - Retarder le plus longtemps possible l'instant du codage
- **Décomposer**
 - *"...diviser chacune des difficultés que j'examinerais en autant de parties qu'il se pourrait et qu'il serait requis pour les mieux résoudre."* Descartes
- **Combiner**
 - Résoudre le problème par combinaison d'abstractions
- **Mais aussi :**
 - Vérifier, modulaire, réutiliser...

Notion d'énoncé (du problème)

- Souvent le problème est "mal posé" ...
 - Rechercher l'indice du plus petit élément d'une suite
=> Spécifier = produire un énoncé
- Énoncé = texte où sont définies sans ambiguïté :
 - L'entrée (données du problème)
 - La sortie (résultats recherchés)
 - Les relations (éventuelles) entre les données et les résultats
- Que dois-je obtenir ?
 - Soit I l'ensemble des indices des éléments égaux au minimum d'une suite. Déterminer le plus petit élément de I .



Notion d'algorithme



- = Description d'un processus de résolution d'un problème bien défini
- = Succession d'actions qui, agissant sur un ensemble de ressources (entrée), fourniront la solution (sortie) au problème
- Comment faire pour l'obtenir ?

Pseudo code

Faire la différence entre les contraintes propres à un langage et les difficultés inhérentes à un problème donné

```
lire (n)
pour i ← 0 à n
  si (i mod 2) # 0
    alors afficher(i)
```

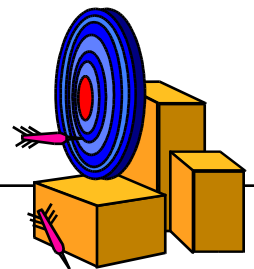
```
#include <stdio.h>
main () {
  int n, i;
  scanf ("%d", &n);
  for (i=0; i<=n; i++) {
    if (i%2) {
      printf ("%d\n", i);
    }
  }
}
```

Plus abstrait, plus lisible, plus concis...

Met en avant l'essence de l'algorithme



La programmation



Complexité d'un algorithme

- **Caractériser un algorithme**
 - Indépendamment de la machine, du compilateur...
 - Complexité
 - » Taille du problème : n
 - » Nombre d'opérations significatives : $T(n)$
 - » Taille mémoire nécessaire : $M(n)$
- **Au mieux, au pire, en moyenne**
- **Notations asymptotiques**
 - $f(n) = O(g(n))$: borne asymptotique supérieure
 - $f(n) = \Omega(g(n))$: borne asymptotique inférieure
 - $f(n) = \Theta(g(n))$: borne approchée asymptotique

Comparaison de temps d'exécution

- 10^6 opérations par seconde
- N = nombre de données à traiter
- C = complexité de l'algorithme de traitement

$N \times C$	1	$\log_2 n$	n	$n \lg_2 n$	n^2	n^3	2^n
10^2	$<1\mu s$	$6,6\mu s$	$0,1ms$	$0,66ms$	$10ms$	$1s$	$4.10^{16}a$
10^3	$<1\mu s$	$9,9\mu s$	$1ms$	$9,9ms$	$1s$	$16,6ms$?
10^4	$<1\mu s$	$13,3\mu s$	$10ms$	$0.13s$	$1,5mn$	$11,5j$?
10^5	$<1\mu s$	$16,6\mu s$	$0,1s$	$1,66s$	$2,7h$	$31,7a$?
10^6	$<1\mu s$	$19,9\mu s$	$1s$	$19,9s$	$11,5j$	$31700a$?

L'algorithmique

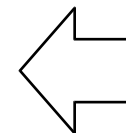
- **Permis de conduire informatique**

- **Produit de matrices carrées $n \times n$**

- Nombre de multiplications

- Algorithme classique : $T(n) = O(n^3)$

- » Trop d'opérations


$$c_{i,j} = \sum_{k=1,n} a_{ik} \times b_{kj}$$

- Meilleure borne inférieure connue : $T(n) = \Omega(n^2)$

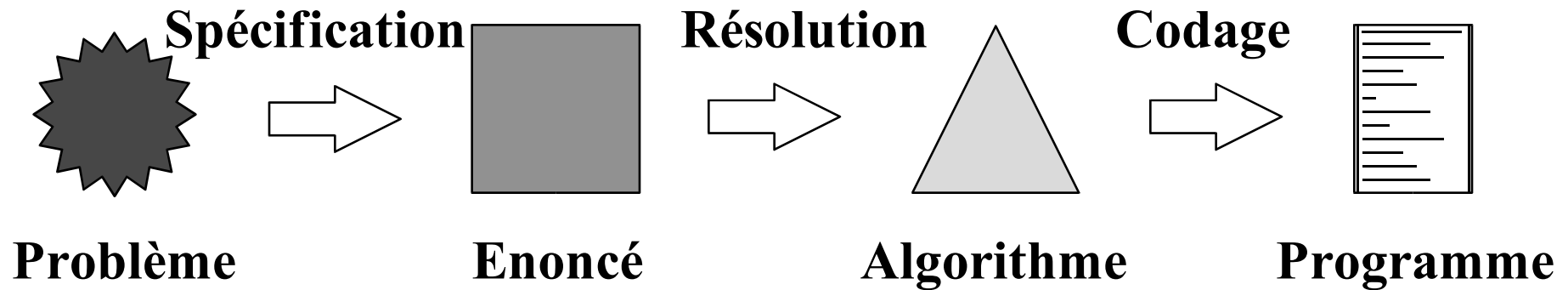
- Algorithme de Strassen : $T(n) = \Theta(n^{\lg 7}) = O(n^{2,81})$

- Meilleur algorithme connu : $T(n) = O(n^{2,376})$

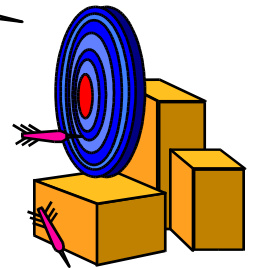
- **Programme**

- Algorithme destiné à la machine

Conception d'un programme

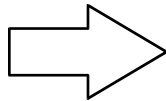


**Ne pas se laisser aveugler par l'objectif final :
le codage !**



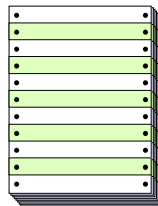
Programmer, c'est communiquer

- Avec la machine
- Avec soi même
- Avec les autres

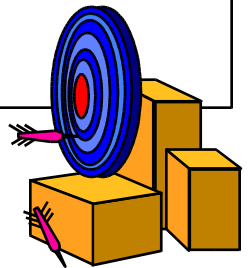


Désignations évocatrices

**Algorithmes en pseudo-code
concis et clairs**



**Programmes indentés
et commentés**



Cycle de vie d'un programme (1)

Analyse + spécification

Conception

Codification

Un processus itératif



Maintenance

Vérification

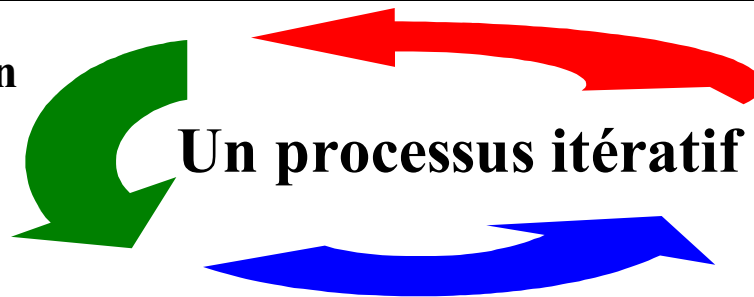
- **Analyse + spécification**
 - Définir clairement le problème
 - Recenser les données
 - Dégager les grandes fonctionnalités
- **Conception**
 - Organiser les données
 - Concevoir l'algorithme en pseudo-code
- **Codification**
 - Traduire l'algorithme dans un langage de programmation

Cycle de vie d'un programme (2)

Analyse + spécification

Conception

Codification



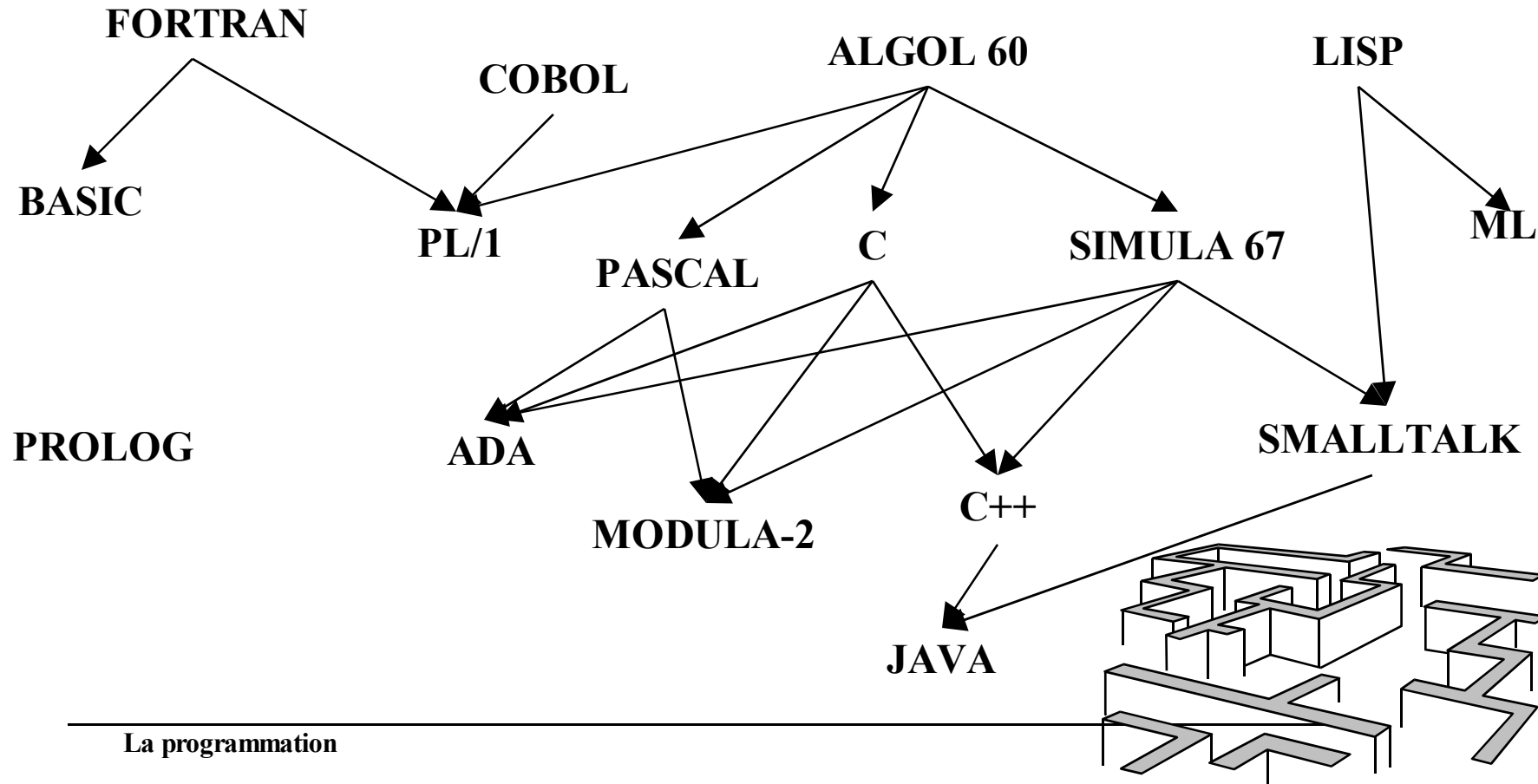
Maintenance

Vérification

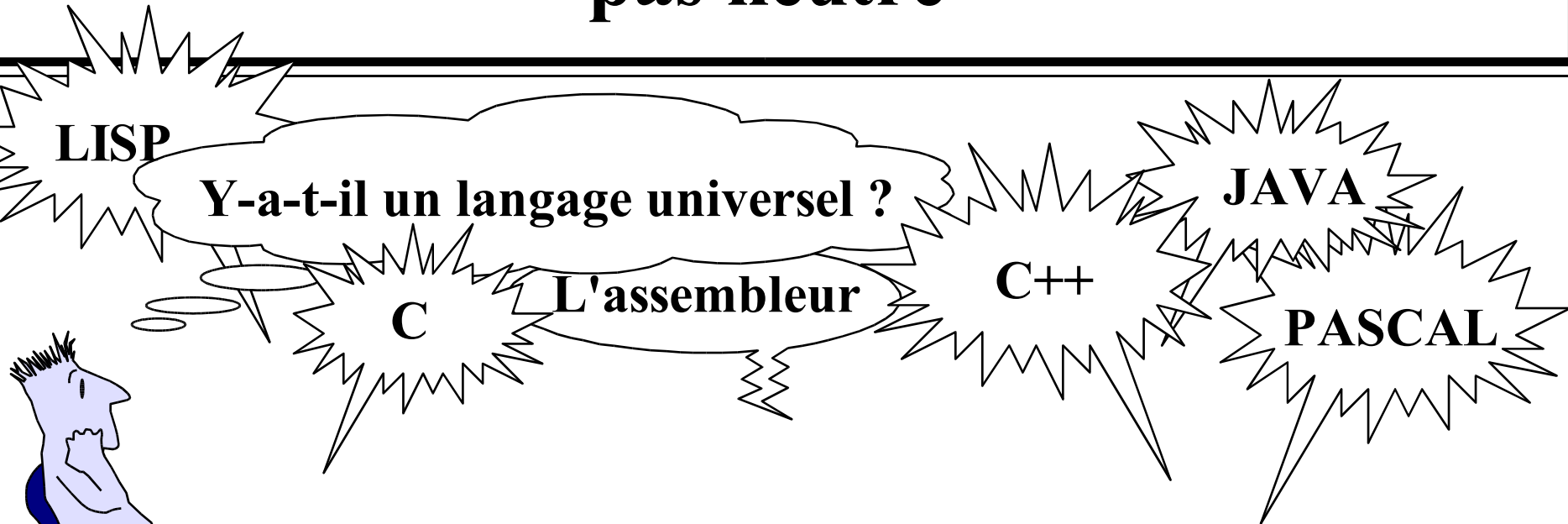
- **4. Vérification (test & mise au point)**
 - Utiliser le programme avec des entrées spécifiques
 - Utiliser un outil de mise au point
- **5. Maintenance**
 - Adapter le programme existant pour de nouvelles fonctionnalités et/ou pour corriger les erreurs
- **Une documentation doit être associée à chaque étape**

Généalogie partielle des langages de programmation

- Plus de 4000 langages



Le choix d'un langage n'est pas neutre



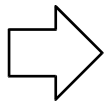
- Un langage facilite la résolution de classes de problème
 - C : systèmes d'exploitation (Unix)...
 - C++ : applications de grande taille...
 - JAVA : applications de grande taille...
 - LISP : prototypage, systèmes experts...

Paradigmes des langages de haut niveau (1)

- **Désigner**
 - Expliciter une entité, en la nommant et en lui associant une définition au moins intuitive
- **Typer**
 - Connaître précisément les propriétés pertinentes d'une entité
- **Paramétrer**
 - Traiter un problème plus général que le problème posé
 - Améliorer la résistance de la solution aux changements
 - Réutiliser

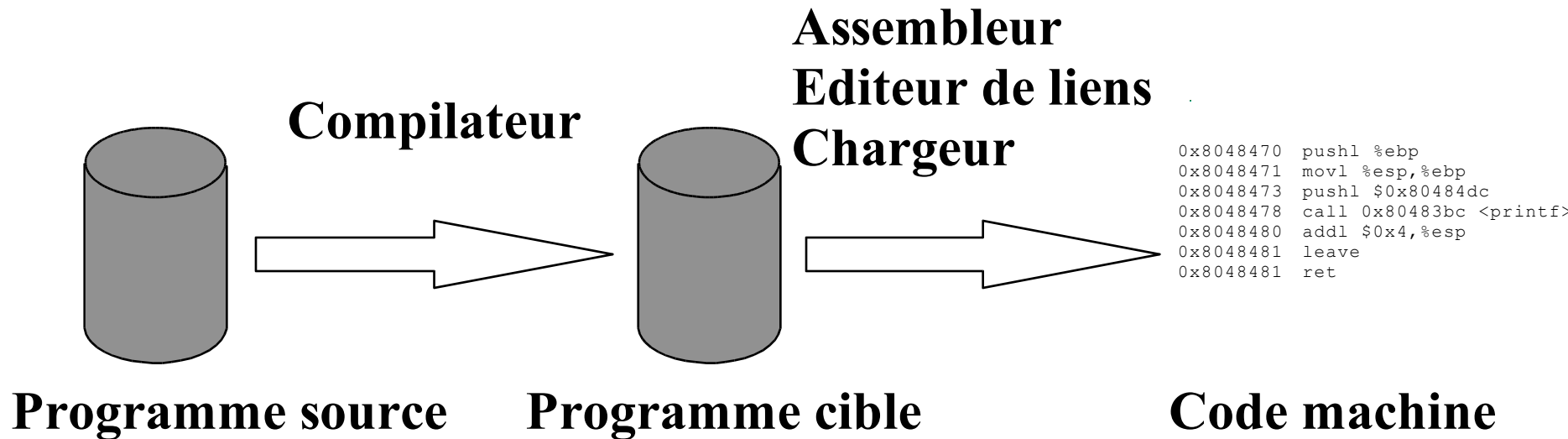
Paradigmes des langages de haut niveau (2)

- **Sérialiser**
 - Construire des séquences d'actions
- **Décomposer par cas**
 - Découper le domaine des données initiales
- **Itérer**
 - Introduire un sous problème intermédiaire paramétré



Réduire la complexité d'un problème

Langage compilé



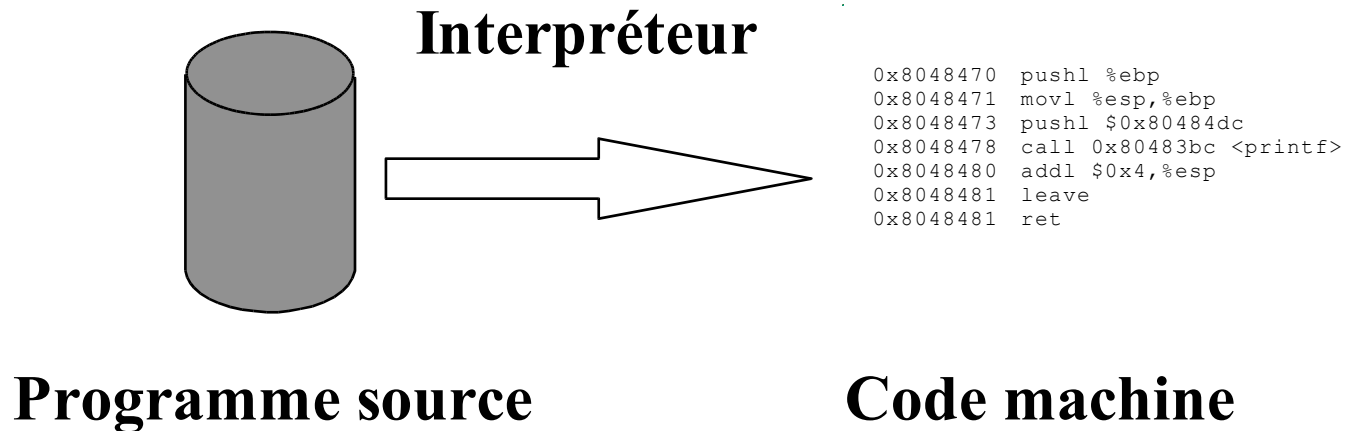
```
$ emacs monProg.c
$ gcc monProg.c -o monProg
$ ./monProg
```

Compilation

- Analyse lexicale
- Analyse syntaxique
- Analyse sémantique
- Génération de code
- Optimisation

- 1) `position := initiale + vitesse * 60`
- 2)
-
- ```
graph TD; A[":="] --- B["position"]; A --- C["+"]; C --- D["initiale"]; C --- E["*"]; E --- F["vitesse"]; E --- G["60"]
```
- 3)
- empiler adresse de position
  - empiler valeur de initiale
  - empiler valeur de vitesse
  - empiler 60
- \*
- +
- :=

# Langage interprété



- emacs monProg.l
- lisp monProg.l