

# Optimalisation d'un programme exemplaire (1)

- Calculer la valeur d'un dépôt à terme de 5 ans, à la fin de chaque année, en considérant une capitalisation mensuelle et une trimestrielle, pour un taux d'intérêt annuel donné
- Données d'entrée :
  - ♦ `depotInit`  $d_0$  montant initial du dépôt nombre
  - ♦ `tauxInteret`  $t_a$  taux d'intérêt annuel en % nombre
- Données de sortie :
  - ♦ `depotMens`  $d_a$  valeurs du dépôt, capit. mens. tableau de nombres
  - ♦ `depotTrim`  $d_a$  valeurs du dépôt, capit. trim. tableau de nombres
- Formule de calcul :
$$d_a = d_0 \cdot \left[ \left( 1 + \frac{t_a / 100}{12/m} \right)^{12/m} \right]^a$$
  - ♦ où :  $a$  – numéro d'année,  $m$  – période de capitalisation en mois

# Optimalisation d'un programme exemplaire (2)

- Premier essai : long, difficile de saisir l'idée générale

depot.m

```
depotInit = input("Montant initial : ");
tauxInteret = input("Taux d'interet : ");
depotMens(1)=depotInit*(1+(tauxInteret/100)/12)^12;
depotMens(2)=depotInit*(1+(tauxInteret/100)/12)^12^2;
depotMens(3)=depotInit*(1+(tauxInteret/100)/12)^12^3;
depotMens(4)=depotInit*(1+(tauxInteret/100)/12)^12^4;
depotMens(5)=depotInit*(1+(tauxInteret/100)/12)^12^5;
depotTrim(1)=depotInit*(1+(tauxInteret/100)/4)^4;
depotTrim(2)=depotInit*(1+(tauxInteret/100)/4)^4^2;
depotTrim(3)=depotInit*(1+(tauxInteret/100)/4)^4^3;
depotTrim(4)=depotInit*(1+(tauxInteret/100)/4)^4^4;
depotTrim(5)=depotInit*(1+(tauxInteret/100)/4)^4^5;
disp("Avec capitalisation mensuelle : "); disp(depotMens);
disp("Avec capitalisation trimestrielle : "); disp(depotTrim);
```

+ correction

# Optimalisation d'un programme exemplaire (3)

- Découpage : l'idée générale claire dans *depot.m*

## depot.m

```
depotInit = input("Montant initial : ");
```

```
tauxInteret = input("Taux d'interet : ");
```

```
calculerdepotmens;
```

```
calculerdepottrim;
```

```
disp("Avec capitalisation mensuelle : "); disp(depotsMens);
```

```
disp("Avec capitalisation trimestrielle : "); disp(depotsTrim);
```

### ► calculerdepotmens.m

```
depotsMens(1)=depotInit*(1+(tauxInteret/100)/12)^12;
```

```
depotsMens(2)=depotInit*(1+(tauxInteret/100)/12)^12^2;
```

```
depotsMens(3)=depotInit*(1+(tauxInteret/100)/12)^12^3;
```

```
depotsMens(4)=depotInit*(1+(tauxInteret/100)/12)^12^4;
```

```
depotsMens(5)=depotInit*(1+(tauxInteret/100)/12)^12^5;
```

### ► calculerdepottrim.m

```
depotsTrim(1)=depotInit*(1+(tauxInteret/100)/4)^4;
```

```
...
```

```
depotsTrim(5)=depotInit*(1+(tauxInteret/100)/4)^4^5;
```

+ compréhensibilité  
+ réutilisabilité : on peut copier les procédures de calcul dans un autre programme

# Optimalisation d'un programme exemplaire (4)

- Fonctions : éliminent les risques liés à la portée globale des variables désignées avec les mêmes noms dans les 3 fichiers

## depot.m

```
depotInit = input("Montant initial : ");
tauxInteret = input("Taux d'interet : ");
depotMens=calculerdepotmens(depoteInit,tauxInteret);
depotTrim=calculerdepottrim(depoteInit,tauxInteret);
disp("Avec capitalisation mensuelle : "); disp(depoteMens);
disp("Avec capitalisation trimestrielle : "); disp(depoteTrim);
```

## calculerdepotmens.m

```
function dep=calculerdepotmens(depoteInit,tauxInteret)
dep(1)=depoteInit*(1+(tauxInteret/100)/12)^12;
...
dep(5)=depoteInit*(1+(tauxInteret/100)/12)^12^5;
endfunction
```

## calculerdepottrim.m

```
function dep=calculerdepottrim(depoteInit,tauxInteret)
dep(1)=depoteInit*(1+(tauxInteret/100)/4)^4;
...
```

+ fiabilité

← Un programmeur formé qui comprend la portée des variables, peut bien utiliser les mêmes noms de variables dans de différents espaces (c'est l'intérêt même des portées différentes)

# Optimalisation d'un programme exemplaire (5)

- Combinaison des deux fonctions en une, la période de capitalisation devenue son troisième argument

## depot.m

```
depotInit = input("Montant initial : ");
tauxInteret = input("Taux d'interet : ");
depotMens=calculerdepot(depoteInit,tauxInteret,1);
depotTrim=calculerdepot(depoteInit,tauxInteret,3);
disp("Avec capitalisation mensuelle : "); disp(depoteMens);
disp("Avec capitalisation trimestrielle : "); disp(depoteTrim);
```

## ► calculerdepot.m

```
function dep=calculerdepot(depoteInit,tauxInteret,periodeCapital)
dep(1)=depoteInit*(1+(tauxInteret/100)/(12/periodeCapital))...
^(12/periodeCapital);
...
dep(5)=depoteInit*(1+(tauxInteret/100)/(12/periodeCapital))...
^(12/periodeCapital)^5;
endfunction
```

- + lisibilité : code plus court
- + compréhensibilité : on ne se demande plus pourquoi le codeur a créé 2 fonctions qui font le même
- + réutilisabilité : une fonction universelle
- + extensibilité : autres ou plus de périodes
- lisibilité : une formule répétée encore plus longue

# Optimalisation d'un programme exemplaire (6)

- Calculer une fois, sauvegarder dans une variable, l'utiliser plusieurs fois

## depot.m

```
depotInit = input("Montant initial : ");  
tauxInteret = input("Taux d'interet : ");  
depotMens=calculerdepot(depoteInit,tauxInteret,1);  
depotTrim=calculerdepot(depoteInit,tauxInteret,3);  
disp("Avec capitalisation mensuelle : "); disp(depoteMens);  
disp("Avec capitalisation trimestrielle : "); disp(depoteTrim);
```

## calculerdepot.m

```
function dep=calculerdepot(depoteInit,tauxInteret,periodeCapital)  
tauxFrac=tauxInteret/100;  
nbreCap=12/periodeCapital;  
dep(1)=depoteInit*(1+tauxFrac/nbreCap)^nbreCap;  
...  
dep(5)=depoteInit*(1+tauxFrac/nbreCap)^nbreCap^5;  
endfunction
```

- + lisibilité
- compréhensibilité : on peut facilement oublier ce que représente *nbreCap*
- + efficacité économique : plus de variables mais moins d'opérations

# Optimalisation d'un programme exemplaire (7)

- On décrit la signification des variables à l'aide de commentaires

## depot.m

```
depotInit = input("Montant initial : ");  
tauxInteret = input("Taux d'interet : ");  
depotMens=calculerdepot(depoteInit,tauxInteret,1);  
depotTrim=calculerdepot(depoteInit,tauxInteret,3);  
disp("Avec capitalisation mensuelle : "); disp(depoteMens);  
disp("Avec capitalisation trimestrielle : "); disp(depoteTrim);
```

## calculerdepot.m

```
function dep=calculerdepot(depoteInit,tauxInteret,periodeCapital)  
tauxFrac=tauxInteret/100; # taux d'interet comme fraction  
nbreCap=12/periode_capital; # nombre de capitalisations par an  
dep(1)=depoteInit*(1+tauxFrac/nbreCap)^nbreCap;  
...  
dep(5)=depoteInit*(1+tauxFrac/nbreCap)^nbreCap^5;  
endfunction
```

+ compréhensibilité

# Optimalisation d'un programme exemplaire (8)

- Au lieu de répéter les lignes du code, on introduit une structure de répétition

## depot.m

```
depotInit = input("Montant initial : ");  
tauxInteret = input("Taux d'interet : ");  
depotMens=calculerdepot(depoteInit,tauxInteret,1);  
depotTrim=calculerdepot(depoteInit,tauxInteret,3);  
disp("Avec capitalisation mensuelle : "); disp(depoteMens);  
disp("Avec capitalisation trimestrielle : "); disp(depoteTrim);
```

## calculerdepot.m

```
function dep=calculerdepot(depoteInit,tauxInteret,periodeCapital)  
tauxFrac=tauxInteret/100; # taux d'interet comme fraction  
nbreCap=12/periodeCapital; # nombre de capitalisations par an  
for an=[1:5]  
dep(an)=depoteInit*(1+tauxFrac/nbreCap)^nbreCap^an;  
endfor  
endfunction
```

- + lisibilité : code plus court
- + compréhensibilité : il est devenu évident qu'une même formule est utilisée pour dans chaque année
- lisibilité : il n'est pas évident que `dep(an)...` est répété 5 fois



# Optimalisation d'un programme exemplaire (9)

- Indentation du contenu de la fonction et de la structure de contrôle

## depot.m

```
depotInit = input("Montant initial : ");  
tauxInteret = input("Taux d'interet : ");  
depotMens=calculerdepot(depoteInit,tauxInteret,1);  
depotTrim=calculerdepot(depoteInit,tauxInteret,3);  
disp("Avec capitalisation mensuelle : "); disp(depoteMens);  
disp("Avec capitalisation trimestrielle : "); disp(depoteTrim);
```

## calculerdepot.m

```
function dep=calculerdepot(depoteInit,tauxInteret,periodeCapital)  
→ tauxFrac=tauxInteret/100; # taux d'interet comme fraction  
  nbreCap=12/periodeCapital; # nombre de capitalisations par an  
  for an=[1:5]  
    → dep(an)=depoteInit*(1+tauxFrac/nbreCap)^nbreCap^an;  
  endfor  
endfunction
```

|                     |
|---------------------|
| + lisibilité        |
| + compréhensibilité |

# Optimalisation d'un programme exemplaire (10)

- Et si, dans le futur, on voudrait changer le nombre d'années considérées ? ou ajouter plus de périodes de capitalisation ?

depot.m

```
depotInit = input("Montant initial : ");  
tauxInteret = input("Taux d'interet : ");  
for periodeCapital=[1,3]  
    dep=calculerdepot(depotInit,tauxInteret,periodeCapital,5);  
    printf("Avec capitalisation tous les %d mois : %.2f\n", periodeCapital, dep);  
endfor
```

calculerdepot.m

```
function dep=calculerdepot(depotInit,tauxInteret,...  
periodeCapital,nbreAnnees)  
    tauxFrac=tauxInteret/100; # taux d'interet comme fraction  
    nbreCap=12/periodeCapital; # nombre de capitalisations par an  
    for an=[1:nbreAnnees]  
        dep(an)=depotInit*(1+tauxFrac/nbreCap)^nbreCap^an;  
    endfor  
endfunction
```

- + extensibilité : nombre d'années facilement modifiée (c'est par ailleurs grâce à l'introduction de la boucle *for*)
- + extensibilité : un seul nombre à ajouter pour chaque nouvelle période de capitalisation

# Optimalisation d'un programme exemplaire (11)

- La multiplication prend moins du temps que l'élevation à une puissance

depot.m

```
depotInit = input("Montant initial : ");
tauxInteret = input("Taux d'interet : ");
for periodeCapital=[1,3]
    dep=calculerdepot(depotInit,tauxInteret,periodeCapital,5);
    printf("Avec capitalisation tous les %d mois : %.2f\n", periodeCapital, dep);
endfor
```

calculerdepot.m

```
function dep=calculerdepot(depotInit,tauxInteret,periodeCapital,nbreAnnees)
    tauxFrac=tauxInteret/100; # taux d'interet comme fraction
    nbreCap=12/periodeCapital; # nombre de capitalisations par an
    for an=[1:nbreAnnees]
        dep(an)=depotInit*(1+tauxFrac/nbreCap)^(nbreCap*an);
    endfor
endfunction
```

+ efficacité  
économique :  
temps de calcul  
plus court

# Optimalisation d'un programme exemplaire (12)

- Encore moins d'opérations arithmétiques qui sont répétées pour chaque année : une multiplication seulement

depot.m

```
depotInit = input("Montant initial : ");
tauxInteret = input("Taux d'interet : ");
for periodeCapital=[1,3]
    dep=calculerdepot(depotInit,tauxInteret,periodeCapital,5);
    printf("Avec capitalisation tous les %d mois : %.2f\n", periodeCapita
endfor
```

calculerdepot.m

```
function dep=calculerdepot(depotInit,tauxInteret,periodeCapital,nbreAnnees)
    tauxFrac=tauxInteret/100; # taux d'interet comme fraction
    nbreCap=12/periodeCapital; # nombre de capitalisations par an
    mult=(1+tauxFrac/nbreCap)^nbreCap; # multiplicateur chaque annee
    dep(1)=depotInit*mult;
    for an=[2:nbreAnnees]
        dep(an)=dep(an-1)*mult;
    endfor
endfunction
```

+ efficacité économique : temps de calcul plus court  
+ correction : pour nbre\_annees=1 ça marche car pour [2:1] la boucle n'est pas exécutée (il a fallu s'en assurer)

# Optimalisation d'un programme exemplaire (13)

- **Comparaison du temps de calcul**

- ◆ On peut l'évaluer seulement sur un grand nombre d'opérations
- ◆ Offres pour 1000 clients de la banque, 20 dépôts initiaux, durée de 40 ans
- ◆ On élimine l'affichage pour qu'il n'affecte pas le temps (c'est assez lent)
- ◆ Les fonctions d'Octave **tic** et **toc** permettent de mesurer le temps d'exécution : *toc* renvoie le temps qui est passé depuis *tic*

testdepot.m

```
tic();
```

```
for client=[1:1000]
```

```
    for depotInit=[200:200:4000]
```

```
        for periodeCapital=[1,3,6,12]
```

```
            dep=calculerdepot(depotInit,tauxInteret,periodeCapital,40);
```

```
        endfor
```

```
    endfor
```

```
endfor
```

```
toc()
```

**version finale : 40,5 secondes**

$dep(an)=dep(an-1)*mult$

**version de départ : 46,7 secondes soit +15 %**

$dep(an)=depotInit*(1+(tauxInteret/100)/(12/periodeCapital))...^{(12/periodeCapital)^an}$

# Paradigmes de programmation

- Un **paradigme de programmation** est une manière de programmer et en même temps d'exécuter les programmes
  - ♦ Il fournit et détermine
    - ▶ la **vue** qu'a le programmeur de l'exécution de son programme
    - ▶ la **manière** dont l'algorithme est codé dans un langage de programmation
- Les paradigmes sont liés aux :
  - ♦ différentes **théories** mathématiques du calcul
    - ▶ p. ex. calcul arithmétique – programmation impérative, calcul logique – programmation logique...
  - ♦ différents **langages** de programmation
    - ▶ il y a plus de langages que de paradigmes
    - ▶ un paradigme peut être utilisé dans plusieurs langages
    - ▶ un langage peut être fondé sur plusieurs paradigmes
  - ♦ différents **concepts** de programmation
    - ▶ ce sont des techniques, mécanismes ou idées élémentaires (p. ex. boucle, récurrence, objet...)
    - ▶ un concept peut être présent dans plusieurs paradigmes



# Principaux paradigmes (1)

- **Programmation impérative**
  - ◆ Voit le programme comme une séquence d'instructions exécutées par l'ordinateur pour changer l'état (les valeurs des données) de ce programme
  - ◆ Correspond au mode **impératif** dans les langages humains donc assez naturel
  - ◆ Reflète le mode de fonctionnement de la plupart des processeurs
  - ◆ Le plus répandu et concerne le plus grand nombre des langages
  - ◆ N'est pas toujours le meilleur choix, surtout en termes d'efficacité
  - ◆ Langages : assembleurs ; descendants de Fortran et Algol
- **Programmation structurée**
  - ◆ Recommande une organisation **structurée** hiérarchique du code à l'aide de laquelle de différentes étapes peuvent être discernées
  - ◆ Facilite l'achèvement de la correction, la fiabilité, la lisibilité, la compréhensibilité...
  - ◆ Langages : la plupart des descendants de Fortran et Algol sauf p. ex. Basic original

# Principaux paradigmes (2)

- **Programmation procédurale**
  - ◆ Repose sur l'appel de **procédures** (aussi appelées fonctions – Matlab, routines...) qui contiennent des séries d'actions à réaliser
    - ▶ Une extension et un support de la programmation structurée
  - ◆ N'importe quelle procédure peut être appelée à n'importe quelle étape de l'exécution du programme
    - ▶ Parfois aussi depuis la procédure elle-même (concept de la **récurtivité**)
  - ◆ Il devient possible de réutiliser le même code à différents moments du programme sans avoir à le copier
    - ▶ Modifications plus faciles, y compris élimination d'erreurs
    - ▶ Code plus lisible, compréhensible, réutilisable, extensible
  - ◆ Langages : C, C++, Fortran, Pascal mais p. ex. pas Java ni Basic original



## Principaux paradigmes (3)

- Exemple comparatif : calcul de la somme des éléments d'un vecteur *vec* ; pour simplifier, *taille* contient déjà la taille déterminée

programmation  
impérative  
non-structurée

```
som ← 0
n ← taille
debut_somme:
si non (n > 0) alors aller à fin_somme
som ← som+vec[n]
n ← n-1
aller à debut_somme
fin_somme:
```

programmation  
impérative  
structurée

```
som ← 0
n ← taille
tant que n > 0 faire
    som ← som+vec[n]
    n ← n-1
fin tant que
```

programmation  
impérative  
procédurale

```
fonction somme(v,t)
    som ← 0
    n ← t
    tant que n > 0 faire
        som ← som+v[n]
        n ← n-1
    fin tant que
    retourner som
fin fonction

...
somme(vec,taille)
```

# Principaux paradigmes (4)

- **Programmation orientée objet**
  - ◆ Le programme est construit autour d'**objets** qui combinent les données (attributs) et les actions (méthodes)
  - ◆ C'est une programmation structurée
  - ◆ mais pas procédurale : les actions ne sont pas séparées des données
    - ▶ Mêmes actions adaptées aux types particuliers (vecteur.moyenne, matrice.moyenne...)
  - ◆ Les méthodes servent aussi à lire et à modifier les valeurs des données qui restent cachées du monde extérieur
    - ▶ Protection de données – fiabilité
  - ◆ Langages : SmallTalk, C++, C#, JavaScript, Java, VB.NET (impératifs) ; autres impératifs – comme supplément ; Python, Ruby, OCaml (fonctionnels et impératifs)

programmation orientée objet

```
classe Vec
  attributs : elems, taille
  méthode somme()
    som ← 0
    n ← taille
    tant que n > 0 faire
      som ← som+elems[n]
      n ← n-1
    fin tant que
  retourner som
fin méthode
créateur Vec(v,t)
  elems=v
  taille=t
fin créateur
fin classe
...
oVec = nouveau Vec(vec,taille)
oVec.somme()
```

# Principaux paradigmes (5)

- **Programmation déclarative**
  - ◆ Dans le programme il est **déclaré** ce qui doit être achevé mais il n'est pas dit comment
    - ▶ Le même résultat est produit quel que soit le moment et le contexte de l'appel
  - ◆ Le « comment » est laissé à l'interpréteur ou au compilateur
    - ▶ Le contraire de la programmation impérative
  - ◆ Forme primitive :
    - ▶ **programmation descriptive** – XML, HTML
  - ◆ Autres formes :
    - ▶ fonctionnelle
    - ▶ logique
    - ▶ par contraintes
- **Programmation fonctionnelle**
  - ◆ Le programme est composé de **fonctions** qui définissent les relations entre les données
  - ◆ Ici, *fonction* n'est pas un synonyme mais contraste avec *procédure* ; c'est plus proche de la fonction en mathématiques
    - ▶ C'est le contraire de la programmation procédurale
  - ◆ L'ordre d'actions n'est pas défini ; il peut être quelconque pourvu que les relations soient gardées
  - ◆ Il n'existe pas un état du programme ou des variables
  - ◆ Langages : Lisp, OCaml, Scheme (programmation procédurale y est également possible) ; Haskell



# Principaux paradigmes (6)

- **Programmation logique**

- ◆ Le programme définit le problème à résoudre à l'aide de faits et de règles de **logique**

- ▶ Les interpréteurs y sont des démonstrateurs de théorèmes
- ▶ Règle :  $p(X, Y)$  si  $q(X)$  et  $r(Y)$
- ▶ Fait :  $s(X, Y)$  vrai
- ▶ Si l'interpréteur reçoit  $p(4, z)$  comme problème, il essaye de prouver que c'est vrai ; ainsi il va trouver (à rebours) une valeur de  $z$  adéquate

- ◆ Langages : Prolog

- Idée de sommation avec récurrence

- ◆ la somme de  $n$  éléments est définie comme le 1<sup>er</sup> élément (la tête) plus la somme des  $n-1$  éléments qui suivent (la queue)
- ◆ la somme de 0 éléments (vecteur vide) est définie explicitement comme 0

programmation fonctionnelle

somme : nombre  $\rightarrow$  nombre

somme([ ]) := 0

somme([tete queue]) := ...  
tete + somme(queue)

...

somme(vec)

programmation logique

somme([ ], 0)  $\wedge$

$\forall(H, T, N, M)$

[somme( $T, M$ )  $\wedge$  eq( $N, M+H$ ) ...  
 $\supset$  somme([ $H|T$ ],  $N$ )]

...

somme(vec, som)

# Paradigmes et langages

- Un paradigme donné peut être mieux ou pire adapté à un langage particulier
  - ♦ La plupart des langages sont fondés sur un ou plusieurs paradigmes délibérément choisis
- Avec la plupart des langages il est possible d'appliquer de différents paradigmes
  - ♦ Même distants des fondations du langage ou contradictoires entre eux
  - ♦ C est un langage structurel, cependant il contient la commande *goto* qui permet de sauter vers n'importe quel emplacement du code
  - ♦ Lisp est un langage fonctionnel mais on peut y appliquer la programmation procédurale (qui est le contraire de la programmation fonctionnelle)
  - ♦ Ce n'est pas toujours bénéfique du point de vue de correction et fiabilité
    - ▶ En général : **plus de liberté**  $\Leftrightarrow$  **plus facile de commettre des erreurs** même s'il paraît plus facile d'apprendre le langage et d'écrire un code !
- Suivre les paradigmes associés avec le langage utilisé, c'est :
  - ♦ améliorer la qualité du programme
  - ♦ minimiser le risque d'erreurs de programmation



# Sommation avec de différents langages et paradigmes (1)

- ◆ Pour simplifier, on ignore ce qu'on doit faire pour placer les variables *vec* et *taille* dans la mémoire et définir leurs valeurs, afficher le résultat...
- ◆ Ce ne sont pas des solutions uniques ; elles tentent de démontrer clairement les paradigmes fondateurs d'un langage donné
- ◆ Elles ne sont pas optimales en tout aspect ; elle sont en revanche comparables

| Lang. Basic original<br>Prog. impérative structurée | Assembleur du dsPIC30<br>Prog. impérative structurée | Code machine correspondant<br>du même processeur |
|---|--|--|
| 200 LET som = 0                                     | clr w2   | EB0100 111010110000000010000000                  |
| 210 LET n = taille                                  | mov #Vec,w0  | 208500 001000001000010100000000                  |
| 220 IF NOT n > 0 THEN GOTO 260                      | mov Taille,w1  | 8048C1 100000000100100011000001                  |
| 230 LET som = som + vec(n)                          | sl w1,w1   | D00081 110100000000000010000001                  |
| 240 LET n = n - 1                                   | add w1,w0,w1   | 408080 010000001000000010000000                  |
| 250 GOTO 220  | boucle: cpsgt w1,w0                                  | E60800 111001100000100000000000                  |
| 260   | bra finbcl   | 370003 001101110000000000000011                  |
|   | mov [--w1],w3  | 7801C1 011110000000000011100001                  |
|   | add w2,w3,w2   | 410103 010000010000000010000011                  |
|   | bra boucle   | 37FFFB 0011011111111111111111011                 |
|   | finbcl: mov w2,Som                                   | 8848D2 100010000100100011010010                  |

Langage 3G

Langage 2G

Langage 1G

w1 ≈ n (précisément ce n'est pas l'indice d'un élément mais son adresse, c.-à-d. numéro de « boîte » élémentaire dans la mémoire)

# Sommation avec de différents langages et paradigmes (2)

Lang. Visual Basic  
Prog. impérative procédurale

```
Function Somme(v, t)
  Somme = 0
  n = t - 1
  Do While n >= 0
    Somme = Somme + v(n)
    n = n - 1
  Loop
End Function
...
Som=Somme(Vec,Taille)
```

Lang. C  
Prog. impérative procédurale

```
double somme(double* v, int t)
{
  double s=0;
  while (t > 0)
    s+=v[--t];
  return s;
}
...
som=somme(vec,taille);
```

Lang. Matlab  
Prog. impérative procédurale

```
function s=somme(v, t)
  s=0;
  while t > 0
    s+=v(t);
    t--;
  endwhile
endfunction
...
som=somme(vec,taille);
```

Langage 3G

Langage 4G

En Visual Basic, C et C++  
les éléments d'un vecteur  
sont numérotés  
à commencer de 0

L'astérisque ici marque le **passage indirect (par référence) de l'argument** : en fait, « v » ne garde pas le vecteur mais juste l'adresse de l'emplacement (« numéro de boîte ») de son premier élément dans la mémoire

# Sommation avec de différents langages et paradigmes

## (3)

|   |  |   |  |
|---|--|---|--|
| <p>Lang. Pascal<br/>Prog. impérative procédurale</p> <pre>Type   Vecteur = Array[1..maxtaille] of Real; Function Somme(Var v : Vecteur;   t : Integer) : Real; Var   s : Real; Begin   s := 0;   While t &gt; 0 Do   Begin     s := s+v[t];     t := t-1;   End;   Somme := s; End; ... som := Somme(vec,taille);</pre> | <p>Pascal, C et C++<br/>exigent une déclaration<br/>du type pour chaque<br/>variable</p> | <p>Lang. C++<br/>Prog. orientée objet impérative</p> <pre>class Vec { private:   double* elems;   int taille; public:   Vec(double* v, int t)   {     elems=v;     taille=t;   }   double somme()   {     int s=0;     int n=taille;     while (n &gt; 0)       s+=elems[--n];     return s;   } };</pre> | <p>Le code en langage C présenté<br/>avant fonctionnerait aussi<br/>comme un code C++ mais<br/>il ne suivrait pas le paradigme<br/>fondateur du C++ :<br/>l'orientation objet</p> <p style="text-align: center;">→ suite<br/>↓</p> <pre>Vec* oVec=new Vec(vec,taille); ... som=oVec-&gt;somme();</pre> |
|---|--|---|--|



# Sommation avec de différents langages et paradigmes (4)

Lang. Lisp  
Prog. fonctionnelle

```
(defun somme (v)
  (if (null v)
      0
      (+ (first v) (somme (rest v)))))
...
(somme vec)
```

Lang. Haskell  
Prog. fonctionnelle

```
somme :: [Float] -> Float
somme [] = 0
somme (tete:queue) = tete + somme queue
...
somme vec
```

Lang. Lisp  
Prog. procédurale

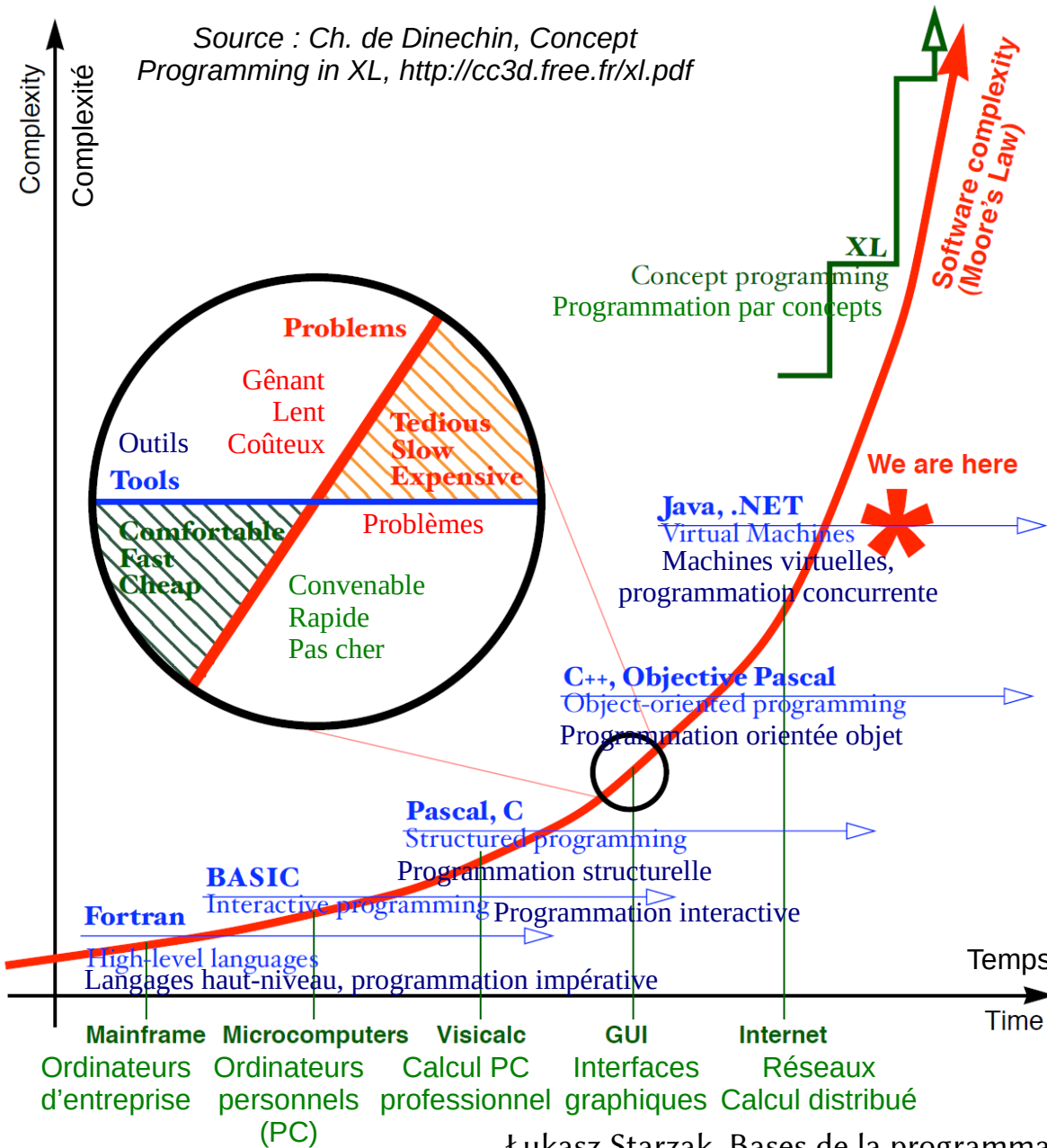
```
(defun somme (v)
  (let ((s 0))
    (do ((v v (rest v))
        ((endp v) s)
        (incf s (first v)))))
...
(somme vec)
```

Lang. Prolog  
Prog. logique

```
somme(0, []).
somme(Somme, [Tete|Queue]) :-
  somme(SomTemp, Queue), Somme is Tete + SomTemp.
...
somme(Som,Vec).
```

Les deux codes en Lisp produisent le même résultat, pourtant le premier est visiblement plus court et lisible (pour quelqu'un qui connaît le langage, bien sûr) ; c'est parce qu'il repose sur le paradigme fondateur de ce langage, contrairement au deuxième code.

# Progression de langages, paradigmes et outils de programmation



- Chaque nouveau paradigme et langage inventé augmente :
  - ♦ la complexité maximale de problèmes abordables
  - ♦ le niveau d'interactivité qui est (encore) facile à réaliser
- Problème : la complexité accroît de façon exponentielle
  - ♦ la lois de Moore appliquée au génie logiciel

# 7. Erreurs d'exécution et de programmation

Gestion d'erreurs d'exécution

Erreurs d'exécution définies par l'utilisateur en Octave

Erreurs de programmation et techniques du débogage

Le débogueur d'Octave

# Gestion d'erreurs d'exécution

- Une **erreur d'exécution** (en plus général – une **exception**) est une condition ou situation exceptionnelle qui apparaît lors de l'exécution d'un programme et qui gêne sa continuation
- La **gestion d'erreurs** (ou **d'exceptions**) permet d'y réagir et essayer de les remédier tout pendant l'exécution du programme
- Structure **try-catch** (essaye-attrape)
  - ♦ **try**  
*actions1*
  - ♦ **catch**  
*actions2*
  - ♦ **end**
  - ♦ L'ordinateur essaye d'exécuter les commandes du bloc *actions1* jusqu'à ce qu'une erreur se produise
  - ♦ À ce moment, il omet le reste du bloc *actions1* et passe à l'exécution du bloc *actions2*
  - ♦ En Octave, le message d'erreur est accessible avec la commande **lasterr**

# Gestion d'erreurs – exemple

- Sans gestion d'erreurs
  - ♦ `chom=[10.7 10.1 10.1 10.6]`  
`coeff=[0.974 1.044 1.022 0.964]`  
`moyDesaison=(chom*coeff)/4`  
`moy=mean(chom)`
  - ♦ `error: operator *: nonconformant arguments (op1 is 1x4, op2 is 1x4)`
  - ♦ L'exécution du code est interrompue
  - ♦ La valeur de *moy* reste indéfinie
- Avec gestion d'erreurs
  - ♦ `try`  
`moyDesaison=(chom*coeff)/4`  
`catch`  
`disp('Impossible de calculer la`  
`moyenne desaisonnalisée !')`  
`disp(lasterr)`  
`end`  
`moy=mean(chom)`
- ♦ Impossible de calculer la moyenne desaisonnalisée !  
operator \*: nonconformant arguments (op1 is 1x4, op2 is 1x4)  
moy = 10.375
- ♦ Au moins on connaît la moyenne non désaisonnalisée
- ♦ Le programme peut continuer
- On peut même remédier
  - ♦ On suppose que l'utilisateur s'est trompé et on transpose *coeff*
  - ♦ `catch`  
`if (columns(coeff)==columns(chom)`  
`& rows(coeff)==rows(chom))`  
`moyDesaison=(chom*coeff)/4`  
`else`  
`disp('Impossible de calculer')`  
`endif`  
`end`
  - ♦ `moyDesaison = 10.377`

# Erreurs d'exécution définies par l'utilisateur

- **error**(*message*)
  - ◆ Affiche le message d'**erreur** et arrête l'exécution du code
  - ◆ Pour les problèmes qui rendent la continuation impossible ou vain
  - ◆ `if (nbreMois != 0)`
    - `moyenne = total / nbreMois;`
    - `else`
      - `error("Division par 0 !");`
    - `endif`
    - `prevision3ans = present + moyenne*36;`
      - ▶ Calcul n'est pas possible et le résultat est indispensable
- **warning**(*message*)
  - ◆ Affiche le message d'**avertissement** mais continue l'exécution du code
  - ◆ Pour les cas où la continuation est possible mais où l'utilisateur doit être conscient de l'apparition d'une exception
  - ◆ `if (marge < 0)`
    - `warning("Vous avez entre une marge negative.");`
    - `endif`
    - `prixVente = prixAchat * (1+marge);`
      - ▶ Inhabituel mais possible et ne gêne pas le calcul

# Erreurs de programmation

- **Il n'est pas possible d'écrire un code correct dès le premier moment**
  - ◆ Une **erreur de programmation** est une formulation du code d'un programme incorrecte du point de vue de la *correction* de ce programme
  - ◆ On estime de 0,5 à 3 erreurs de programmation par 1000 lignes du code
- Une erreur de programmation peut être découverte :
  - ◆ pendant la vérification du programme **avant qu'il soit exécuté**
  - ◆ mais souvent seulement lorsqu'elle **se manifeste pendant l'exécution** par :
    - ▶ **arrêt** (même de l'ordinateur) avant que le résultat complet soit fourni
    - ▶ **résultat incorrect** (parfois difficile de repérer, p. ex. quand il s'agit de calculs trop complexes pour être effectués à main)
  - ◆ Mieux vaut que ce soit pendant **l'exécution par le programmeur**
    - ▶ L'utilisateur fait confiance au logiciel ; probable qu'il ne repère pas l'erreur
    - ▶ L'utilisateur est plutôt incapable de corriger le code (même s'il est informaticien lui-même) alors il abandonne le logiciel
- **Le programmeur est donc obligé de :**
  - ◆ **tester** son programme
  - ◆ **éliminer** les erreurs de programmation qu'il a commises



# Techniques du débogage

- Par **débogage** on comprend recherche et élimination des erreurs de programmation pour assurer la correction et la fiabilité du code
  - ♦ de l'anglais « debug » = éliminer des punaises
- Lorsqu'un programme ne produit pas les résultats attendus, en général on essaye de trouver la ligne qui provoque l'erreur
  1. **Réanalyser** le programme ligne par ligne et essayer de prévoir les résultats
  2. Insérer des commandes d'**affichage** (en Matlab : en effaçant les points-virgules ou bien en utilisant *disp* ou *printf*) pour vérifier l'état des variables au moment donné de l'exécution du programme
  3. À l'aide d'un **débogueur**, exécuter le programme ligne par ligne et observer son avancement et les changements en valeurs des variables
    - ♦ Technique intermédiaire entre 2 et 3, disponible seulement en Octave : Insérer la commande **keyboard** dans le code du programme
      - ▶ Arrête l'exécution du code là où elle est insérée et permet d'entrer des instructions d'Octave quelconques en ligne
      - ▶ On peut alors examiner la valeur en cours de n'importe quelle variable, vérifier le résultat d'une fonction avec d'autres valeurs des arguments...



# Le déboguer d'Octave

- Un **débogueur** est un outil conçu pour faciliter le débogage
  - ♦ Un débogueur étant intégré dans Octave, on l'utilise depuis sa ligne de commandes standard
  - ♦ Avec d'autres environnements de programmation, le débogueur peut être un logiciel séparé
- Les commandes du débogueur d'Octave
  - ♦ Mettre un point d'arrêt (ang. breakpoint)  
**dbstop** *nom\_fonction*  
*numéro\_ligne*
  - ♦ Retirer un point d'arrêt  
**dbclear** *nom\_fonction*  
*numéro\_ligne*
  - ♦ Arrêter lorsqu'un avertissement se produit  
**dbstop if warning**
  - ♦ Arrêter lorsqu'une erreur se produit  
**dbstop if error**
  - ♦ Arrêter lorsque une valeur *NaN* (pas-un-nombre) ou *Inf* (infini) apparaît  
**dbstop if naninf**
  - ♦ Reprendre l'exécution  
**dbcont**
  - ♦ Afficher la liste de points d'arrêt  
**dbstatus** *nom\_fonction*
  - ♦ Exécuter une ou plusieurs lignes  
**dbstep** *nombre\_lignes*
  - ♦ Quitter le débogueur  
**dbquit**

# Exemple du débogage (1)

- Calculer la marge brute des produits vendus par chaque vendeur d'un magasin exprimée comme un pourcentage de la marge du meilleur vendeur

- ♦ Formule arithmétique

$$r_i = \frac{v_i - c_i}{\max\{v_i - c_i\}} \cdot 100$$

où :

$v_i$  – total des prix de vente

(de l' $i$ -ème employé)

$c_i$  – total des coûts de revient

$r_i$  – marge relative

en pourcentage

- Programme

- ♦ 

```
function mrel=margerel(vente,cout)
    mabs = vente-cout;
    mmax = maxi(mabs);
    mrel = mabs/mmax;
endfunction
```
- ♦ 

```
function xm=maxi(x)
    n = numel(x);
    xm = x(1);
    for m=2:n
        if x(m) > xm
            xm = x(m);
        endif
    endfor
endfunction
```

- Vérification

- ♦ 

```
> v=[20,16,18,12];
> c=[18,12,17,9];
> margerel(v,c)
ans = [0.667 1.333 0.333 1.000]
```

## Exemple du débogage (2)

- On constate que le résultat n'est pas correct alors on essaye de trouver la ligne où l'erreur s'est produite

- ◆ On met un point d'arrêt dans la ligne 3 de la fonction *margerel*  
> `dbstop margerel 3`

- ◆ On appelle la fonction de nouveau  
> `margerel(v,c)`  
`stopped in margerel.m at line 3`

- ◆ On vérifie si les arguments ont été passés correctement et s'ils n'ont pas été changés par hasard

```
debug> vente  
vente = 20 16 18 12
```

```
debug> cout  
cout = 18 12 17 9
```

C'est correct

- ◆ On vérifie alors si les marges absolues ont été calculées correctement

```
debug> mabs  
mabs = 2 4 1 3
```

C'est correct aussi

```
1 function mrel=margerel(vente,cout)  
2   mabs = vente-cout;  
3   mmax = maxi(mabs);  
4   mrel = mabs/mmax;  
5 endfunction
```

## Exemple du débogage (3)

- ♦ On avance d'une ligne et on vérifie si *mmax* (marge maximale) a pris la valeur correcte de 4

```
debug> dbstep
stopped in margerel.m at line 4
debug> mmax
mmax = 3
```

```
1 function mrel=margerel(vente,cout)
2   mabs = vente-cout;
3   mmax = maxi(mabs);
4   mrel = mabs/mmax;
5 endfunction
```

On constate que la source du problème, c'est la marge maximale déterminée incorrectement

- ♦ Néanmoins, le reste du code ne doit forcément pas être correct non plus ; on impose alors la valeur correcte de *mmax* et on vérifie si on obtient ainsi le résultat final correct

```
debug> mmax=4
debug> dbcont
ans = [0.500 1.000 0.250 0.750]
```

Le résultat est correct, alors on constate que le seul problème apparaît lors de la détermination de la marge maximale

- ♦ Puisque cette détermination est effectuée dans la fonction *maxi*, l'erreur doit se trouver dans le code de cette fonction

## Exemple du débogage (4)

- On cherche alors dans *maxi*

- ◆ On enlève le point d'arrêt dans la fonction *margerel*

```
debug> dbclear margerel
```

- ◆ On met un point d'arrêt dans la fonction *maxi*

```
debug> dbstop maxi 4
```

- ◆ On exécute *margerel* encore une fois

```
debug> margerel(v,c)
```

```
stopped in maxi.m at line 4
```

- ◆ On avance pour voir comment la boucle « pour » est exécutée

```
debug> dbstep
```

```
stopped in maxi.m at line 5
```

```
debug> dbstep
```

```
stopped in maxi.m at line 6
```

Si on est en ligne 6, alors le résultat de la comparaison est correct

- ◆ On avance encore pour voir si la valeur maximale (*xm*) est mise à jour

```
debug> dbstep
```

```
stopped in maxi.m at line 5
```

```
debug> xm
```

```
xm = 3
```

Tandis qu'on attendait 4... Pourquoi est-ce que c'est arrivé ?

```
1 function xm=maxi(x)
2   n = numel(x);
3   xm = x(1);
4   for m=2:n
5     if x(m) > xm
6       xm = x(m);
7     endif
8   endfor
9 endfunction
```

## Exemple du débogage (5)

- On analyse l'instruction à l'intérieur de la structure *si*
  - ♦ Le  $n$ , c'est le nombre des éléments de  $x$
  - ♦ Alors à chaque fois on affectera  $xm$  avec la valeur du dernier élément de  $x$
  - ♦ ...au lieu de l'affecter avec la valeur de l'élément en cours, donc numéro  $m$
- On corrige le code du programme
  - ♦ On termine la session de débogage avec `dbquit`
  - ♦ On modifie le code de la fonction *maxi* pour qu'il contienne en ligne 6 :  
`xm = x(m);`
  - ♦ On vérifie  
`> margerel(v,c)`  
`ans = 0.500 1.000 0.250 0.750`
  - ♦ C'est correct maintenant
  - ♦ On a réussi à localiser et éliminer l'erreur

```
1 function xm=maxi(x)
2   n = numel(x);
3   xm = x(1);
4   for m=2:n
5       if x(m) > xm
6           xm = x(n);
7       endif
8   endfor
9 endfunction
```

# Interrogation

- Composition
  - ◆ 5 problèmes composés de sous-problèmes
  - ◆ 1 sous-problème concernant la **synthèse** (développement) d'un programme plus 1 ou 2 sous-problèmes concernant l'**analyse** d'autres programmes
    - ▶ la longueur (sans compter l'en-tête et la ligne de fin) sera de 10 lignes maximum (7 en moyenne) à analyser ; 6 lignes maximum (2 en moyenne) à écrire
    - ▶ le degré de difficulté ne sera plus haut à celui des programmes analysés en conférence et développés en TP
    - ▶ environ 1/3 du nombre de points total
  - ◆ le reste aura un caractère **théorique** (problèmes abordés en conférence)
  - ◆ des **exemples** des programmes à synthétiser et à analyser ainsi que la **gamme** des problèmes théoriques seront donnés dans un document qui sera publié sur la page web après la dernière conférence
- Conditions
  - ◆ Comme précisé sur la page web, section *Cours magistral (CM)* ▶ *Méthode et conditions d'évaluation*