Les langages

Les paradigmes

Modèle impératif abstrait

Boehm et Jacopini

Programmation impérative

Programmation fonctionnelle

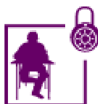
Programmation en logique

Langages de l'informatique

Prof. Patrick Bellot

Télécom ParisTech

Licence des droits d'usage



Droits d'usage autorisé

Par le téléchargement ou la consultation de ce document, l'utilisateur accepte la licence d'utilisation qui y est attachée, telle que détaillée dans les dispositions suivantes, et s'engage à la respecter intégralement.

La licence des droits d'usage de ce document confère à l'utilisateur un droit d'usage sur le document consulté ou téléchargé, totalement ou en partie, dans les conditions définies ci-après, et à l'exclusion de toute utilisation commerciale.

Le droit d'usage défini par la licence autorise un usage dans un cadre académique, par un utilisateur donnant des cours dans un établissement d'enseignement secondaire ou supérieur et à l'exclusion expresse des formations commerciales et notamment de formation continue. Ce droit comprend :

- le droit de reproduire tout ou partie du document sur support informatique ou papier,
- le droit de diffuser tout ou partie du document à destination des élèves ou étudiants.

Aucune modification du document dans son contenu, sa forme ou sa présentation n'est autorisé.

Les mentions relatives à la source du document et/ou à son auteur doivent être conservées dans leur intégralité.

Le droit d'usage défini par la licence est personnel, non exclusif et non transmissible.

Tout autre usage que ceux prévus par la licence est soumis à autorisation préalable et expresse de l'auteur : bellot@telecom-paristech.fr.

Plan

1 Les langages

2 Les paradigmes
- Modèle impératif abstrait
- Boehm et Jacopini
- Programmation impérative
- Programmation fonctionnelle
- Programmation en logique

Les Langages

Dans les années 50, on programmait les ordinateurs en écrivant directement des suites d'octets, parfois en binaire !

```
0x12 0x23 0xA7 0x56 0xff 0x00 0x56 0x12 0x34
0xf3 0x45 0x23 0x12 0xf5 0xaf 0xee 0x23 0x56
0x12 0x56 0x58 0xba 0xb2 0x78 0x83 0x87 0xa1
0x72 0xab 0xbe 0x55 0x00 0xfe 0x76 0xaa 0x12
0x76 0x23 0xff 0xbe 0xaf 0xcc 0xc0 0xc4 0x33
0x56 0x56 0x23 0x32 .... .... .... ....
```

L'ère des pionniers est terminée et nous disposons de langages de programmation.

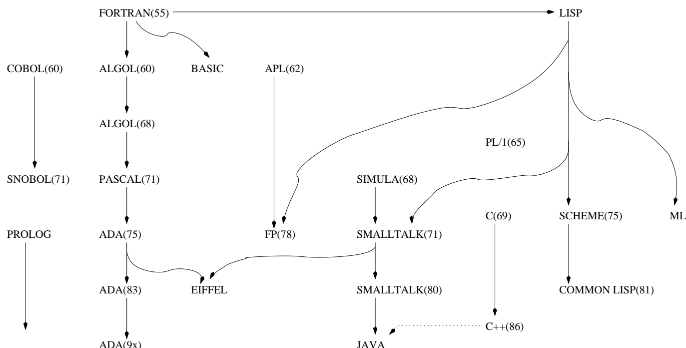
FORTRAN

En 1958 (grande année !), John W. BACKUS crée le premier langage de programmation : FORTRAN.

```
00010 SUBROUTINE MP(A,B,C,N1,N2,N3)
00020 REAL A(N1,N3), B(N1,N2), C(N2,N3)
00030
00040 DO 70 I=1,N3
00050    A(I,1) = A(I,1)+1
00060    CALL SMXPY(N2,A(I,1),N1,C(1,I),B)
00070 CONTINUE
00080
00090 RETURN
00100 END
```

Un programme appelé *compilateur* traduit ce texte en une suite de codes numériques que l'UAL de l'ordinateur sait exécuter.

Les Langages



Les Fonctions d'un Langage

Langages de l'informatique

Patrick Bellot

Les langages

Les paradigmes

Modèle impératif abstrait

Boehm et Jacopini

Programmation impérative

Programmation fonctionnelle

Programmation en logique

Donner des noms symboliques aux mots de la mémoire

Lorsque nous écrivons en C :

```
int i ;
...
i = i+2 ;
```

le compilateur se charge de trouver un mot dans la mémoire pour ranger l'entier `i`, disons à l'ADRESSE `0x00001000`, et génère le code suivant :

```
movi  r1,0x00001000 -> 0x0a 0x01 0x00 0x00 0x10 0x00
load  r2,*r1        -> 0x13 0x21
addi  r2,0x00000002 -> 0x24 0x02 0x00 0x00 0x00 0x02
store r2,*r1        -> 0x15 0x21
```

Le programmeur ne se préoccupe pas de l'adresse en mémoire de la variable `i`. Cette adresse n'est pas obligatoirement fixée comme dans l'exemple ci-dessus. C'est même assez rare. En général, elle varie d'une exécution du programme à une autre.

Les Fonctions d'un Langage

Gérer la représentation binaire des données

Lorsque nous écrivons en C :

```
float f ;
```

le compilateur se charge de trouver deux mots dans la mémoire pour ranger le nombre flottant `f`. La REPRESENTATION BINAIRE du nombre flottant peut dépendre du système d'exploitation.
De même, lorsque nous écrivons en Pascal :

```
type
    etudiant = record
                 nom    : array[1..25] of character ;
                 prenom : array[1..25] of character ;
               end ;
var
    e : etudiant ;
```

le compilateur se charge de calculer la TAILLE des données et de réserver la zone mémoire correspondante.

Les Fonctions d'un Langage

Allocations et désallocations de mémoire

Lorsque nous écrivons en Ada :

```
function MIRROR(A : STRING) return STRING is
   RESULT : STRING(A'RANGE) ;
begin
   for N in A'RANGE loop
      RESULT(N) := A(A'LAST-(N-A'FIRST)) ;
   end loop ;
   return RESULT ;
end MIRROR ;
...
MESSAGE : constant STRING := "hello" ;
...
S := MIRROR(MESSAGE) ;
```

Lors de l'appel de la fonction `MIRROR`, une variable de type `STRING` est créée. Le langage a donc dû allouer une zone de mémoire.

Les Fonctions d'un Langage

Allocations et désallocations de mémoire

De même, lorsque nous écrivons en C :

```
int f() {
  etudiant e ;
  ...
} ;
```

une variable de type `etudiant` est créée (allouée) à l'entrée de la fonction `f`, elle sera détruite (désallouée) à la sortie de la fonction et la mémoire utilisée sera rendue au système.

Les langages

Les paradigmes

Modèle impératif abstrait

Boehm et Jacopini

Programmation impérative

Programmation fonctionnelle

Programmation en logique

Les Fonctions d'un Langage

Réaliser des construtions classiques

- Appels de procédures et de fonctions :

```
A := produit_matrice(B,C) ;
```

- Structures de contrôle de l'exécution :

```
while (i < 100) { ... } ;
for (int i=0;i<100;i++) { ... } ;
repeat { ... } until (i > 100) ;
if (i > 100) then { ... } else { ... } ;
```

- Gérer les déclarations de types et les accès aux composantes des données.
- Gérer la *visibilité* des données et des programmes.

Les Fonctions d'un Langage

Libérer des contraintes *hard* et *soft*

Lorsque nous écrivons en C :

```
etudiant *ptr = (etudiant *)malloc(sizeof(etudiant)) ;
```

nous faisons appel à une fonction `malloc` qui alloue une zone de mémoire et vous rend l'ADRESSE, un POINTEUR, sur le début de cette zone. Cette fonction est fournie dans une LIBRAIRIE de fonctions attachée au compilateur. Elle est DEPENDANTE du système d'exploitation.
Lorsque nous écrivons en C :

```
int i ;
```

nous écrivons quelque chose de TRES DEPENDANT de l'ordinateur sur lequel on est.
Le langage de programmation assure la PORTABILITE des programmes : un programme écrit en langage C sous Unix pourra théoriquement être recompilé et exécuté sur un autre ordinateur ayant éventuellement un autre système d'exploitation.

Plan

1 Les langages

2 Les paradigmes
- Modèle impératif abstrait
- Boehm et Jacopini
- Programmation impérative
- Programmation fonctionnelle
- Programmation en logique

Les paradigmes

Il n'existe pas qu'un seul type de langage de programmation. Ces langages sont regroupés en classes. Les principales classes sont :

- la programmation *impérative* : C, Ada, Pascal, Fortran, Cobol, etc.
- la programmation *à objets* et *à acteurs* : C++ , Java, Eiffel, SmallTalk, ActTalk, etc.
- la programmation *fonctionnelle* : Lisp, Scheme, Caml, etc.
- la programmation *en logique* : Prolog et ses différentes variantes.

Les paradigmes

D'autres classes de langages sont moins importantes mais néanmoins très utiles :

- les langages pour le *parallélisme* : Ada, Concurrent xxxx, etc.
- les langages de *script* impératifs ;
- les langages spécialisés (en général impératifs) qui proposent des fonctionalités spécifiques : Perl, Python, etc.
- les langages de *simulation* ;
- les langages *à flots de données* : Id, SiSal, etc.
- les langages de programmation *sans variable* : FP, Graal, APL, etc.
- les langages *synchrones*, les langages *réactifs*, les langages de *programmation événementielle*, etc.

Le Paradigme Impératif

Le paradigme impératif tient son nom du fait que les programmes sont des suites d'instructions, des ordres que l'ordinateur doit exécuter.

```
int vec[100] ;
...
int somme = 0 ;
for (int i=0;i<100;i++)
   somme += vec[i] ;
```

Un programme est composé de suites d'*instructions* encadrées par des *structures de contrôle de l'exécution* :

- boucles `for`, `while`, `until`, etc.
- `if - then -`, `if - then - else -`, etc.
- instruction `goto` ;
- appels de procédures et de fonctions ;
- échappements, exceptions, etc.

Plan

1 Les langages

2 Les paradigmes

- Modèle impératif abstrait
- Boehm et Jacopini
- Programmation impérative
- Programmation fonctionnelle
- Programmation en logique

Le Modèle Impératif Abstrait

Langages de l'informatique

Patrick Bellot

Les langages

Les paradigmes

Modèle impératif abstrait

Boehm et Jacopini

Programmation impérative

Programmation fonctionnelle

Programmation en logique

C'est celui de la *Machine de Von Neumann*. Des instructions sont écrites consécutivement et numérotées comme dans les vieux langages Basic. Deux types d'instructions :

- l'instruction d'affectation :
  `<variable> := <expression> ;`
- l'instruction de *saut conditionnel* :
  `if <condition> goto <instruction> ;`
  et *inconditionnel* :
  `goto <instruction> ;`
  équivalente à :
  `if 0==0 goto <instruction> ;`

Le Modèle Impératif Abstrait

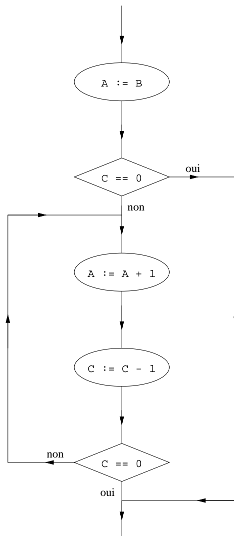
Exemple d'un programme d'addition

Pré-condition : `B` et `C` sont des variables contenant des entiers positifs et... on ne sait faire que les opérations *successeur* (i.e. +1) et *prédécesseur* (i.e. -1) sur les entiers.

```
00 : A := B ;
01 : if (C == 0) goto 05 ;
02 : A := A + 1 ;
03 : C := C - 1 ;
04 : if (C <> 0) goto 02 ;
05 : end ;
```

Post-condition : `A` contient `B+C`, `C` vaut zéro et `B` est inchangé.

Représentation graphique des programmes : l'organigramme



Le Modèle Impératif Abstrait

Il fut un temps pas si lointain où l'on programmait ainsi avec quelques améliorations.

Ce temps a pris fin avec avec Algol et surtout le méritant Pascal.

La grande avancée théorique fut le difficile Théorème de Boehm et Jacopini

Le Modèle Impératif Abstrait

Langages de l'informatique

Patrick Bellot

Les langages

Les paradigmes

Modèle impératif abstrait

Boehm et Jacopini

Programmation impérative

Programmation fonctionnelle

Programmation en logique

Les structures de contrôle de l'exécution peuvent être traduites en instructions élementaires du modèle impératif abstrait.

Le Modèle Impératif Abstrait

Structure **while**

```
while <condition> do {
  <instruction 1>
  ...
  <instruction k>
} ;
```

devient :

```
n:      if (not <condition>) goto n+k+2 ;
n+1:    <instruction 1>
        ...
n+k:    <instruction k>
n+k+1:  goto n ;
n+k+2:
```

Example

Structure **repeat**

```
repeat {
  <instruction 1>
  ...
  <instruction k>
} until <condition> ;
```

devient :

```
n:        <instruction 1>
          ...
n+k-1:    <instruction k>
n+k:      if (not <condition>) goto n ;
```

Example

Structure **for**

```
for (i=0;i<n;i++) {
  <instruction 1>
  ...
  <instruction k>
}  ;
```

devient :

```
i = 0 ;
while (i < n) {
  <instruction 1>
  ...
  <instruction k>
  i++ ;
} ;
```

Example

Structure **if-then**

```
if <condition> {
  <instruction 1>
  ...
  <instruction k>
} ;
```

devient :

```
n:         if (not <condition>) goto n+k+1 ;
n+1:       <instruction 1>
           ...
n+k:       <instruction k>
n+k+1:
```

Example

Structure **if-then-else**

```
if <condition> {
  <instruction 1>
  ...
  <instruction i>
} else {
  <instruction' 1>
  ...
  <instruction' j>
} ;
```

devient :

```
n:        if (not <condition>)
                goto n+i+2 ;
n+1:      <instruction 1>
          ...
n+i:      <instruction i>
n+i+1:    goto n+i+j+2 ;
n+i+2:    <instruction' 1>
          ...
n+i+j+1: <instruction' j>
n+i+j+2:
```

Plan

1 Les langages

2 Les paradigmes
- Modèle impératif abstrait
- Boehm et Jacopini
- Programmation impérative
- Programmation fonctionnelle
- Programmation en logique

Le Modèle Impératif Abstrait

L'idée n'est, bien sûr, pas de programmer avec des structures de contrôle de l'exécution évoluées et de les traduire à la main dans le modèle impératif abstrait : cela est le travail du COMPILATEUR du langage comme nous allons le voir plus loin.

L'idée est, bien entendu, de programmer tout simplement avec ces structures de contrôle :

- le programme est *structuré* ;
- la pensée du programmeur est *structurée* ;
- le programme est plus *compréhensible* ;
- il est possible de *raisonner* sur le programme.

Le Modèle Impératif Abstrait

Chacune de ces structures a une action bien précise et connue de tous. Les structures sont *hierarchiquement* emboîtées. C'est le premier pas vers la *programmation structurée*.

Mais est-ce possible de tout programmer avec ces structures de contrôle ? N'existe-t-il pas des algorithmes que l'on peut exprimer avec des `goto`(s) et que l'on ne peut pas exprimer sans ces `goto`(s) ?

Le Modèle Impératif Abstrait

Le théorème de Boehm et Jacopini (1967) énonce que :

Tout programme comportant éventuellement des sauts conditionnels peut être réécrit en un programme équivalent n'utilisant que les structures de contrôle `while` et `if-then`.

La voie est ouverte vers la programmation moderne.

Le Modèle Impératif Abstrait

Exemple :

```
000 : A := B ;
001 : if (C == 0) goto 005 ;
002 : A := A + 1 ;
003 : C := C - 1 ;
004 : if (C <> 0) goto 002 ;
005 : end ;
```

peut devenir :

```
A := B ;
while (C <> 0) {
  A := A + 1 ;
  C := C - 1 ;
} ;
```

Le Modèle Impératif Abstrait

Faut-il bannir les `goto`(s) ?

La réponse académique est OUI !

La réponse pratique n'est pas toujours OUI !

Cependant on réservera l'usage du `goto` au traitement d'exceptions très simples.

Exemple : le traitement d'une requête SQL.

Plan

1 Les langages

2 Les paradigmes
- Modèle impératif abstrait
- Boehm et Jacopini
- Programmation impérative
- Programmation fonctionnelle
- Programmation en logique

Les caractéristiques de l'impératif

C'est au programmeur qu'incombe la gestion de l'exécution du programme.

Le programme impératif est composé d'instructions à l'intérieur de structures de contrôle emboîtées.

Ce sont ces structures, utilisées par le programmeur, qui décident ce qui doit être exécuter et dans quel ordre cela doit l'être.

Les caractéristiques de l'impératif

C'est au programmeur qu'incombe la gestion des données.

C'est à lui que revient le devoir de déclarer les types et les variables.

C'est à lui de décider de la durée de vie des variables.

C'est à lui de décider ce qu'elles contiendront et quand.

C'est à lui qu'incombe la gestion de la mémoire utilisée par son programme pour y ranger les données.

Les caractéristiques de l'impératif

La programmation impérative est une programmation pas à pas et mot à mot.

Exemple :

```
ps := 0 ;
for i:=1 to N do
   ps := ps + A[i]*B[i] ;
```

Les langages impératifs sont rigides. On ne peut y définir des extensions telles que de nouvelles structures de contrôle. C'est pourquoi la documentation de ces langages est souvent très lourde.

Les caractéristiques de l'impératif

Une autre caractéristique des langages impératifs est qu'ils se ressemblent tous :

The differences between Fortan *and* Algol 68, *although considerable are less significant than the fact that both are based on the programming style of the Von Neumann computer. In fact, some may say that I bear some responsability in this problem.*

John W. Backus

De fait, les différences entre les langages impératifs sont avant tout dans la syntaxe et les outils spécifiques. L'idéal est d'apprendre un langage de très haut niveau comme Ada et un autre de très bas niveau comme C.

Une proposition de John W. Backus

En sa basant sur le langage APL et sur la programmation fonctionnelle, John W. Backus a proposé les Systèmes FP.

Dans ces sytèmes, 8 opérateurs de haut niveau libèrent le programmeur de la programmation pas-à-pas et mot-à-mot.

L'aspect fonctionnel libère le programmeur de la gestion de la mémoire, exactement comme pour Lisp que nous verrons plus loin.

Le problème ? Illisible !

Une proposition de John W. Backus

Dans les systèmes FP, les arguments sont des séquences, les éléments des séquences peuvent être des atomes ou d'autres séquence.

La fonction de transposition (cas particulier) :

$$trans : [\,[a_1, \dots, a_n], [b_1, \dots, b_n]] = [[a_1, b_1], \dots, [a_n, b_n]]$$

La fonction de distribution à droite :

$$distr : [[a_1, \dots, a_n], b] = [[a_1, b], \dots, [a_n, b]]$$

Une proposition de John W. Backus

L'opérateur de réduction $/$:

$$/(f,e) : [\,] = e$$

$$/(f,e) : [x_1, x_2 \ldots, x_n] = f : [a_1, /(f,e) : [x_2, \ldots, x_n]]$$

L'opérateur de composition o :

$$f \;o\; g : x = f : (g : x)$$

L'opérateur de distribution α:

$$\alpha(f) : [x_1, \ldots, x_n] = [f : x_1, \ldots, f : x_n]$$

Une proposition de John W. Backus

La fonction produit scalaire:

$$ps = /(+,0)\ o\ \alpha(*)\ o\ trans$$

La fonction produit de matrice-vecteur:

$$pmv = \alpha(ps)\ o\ distr$$

La fonction produit de matrice-matrice:

$$pmm = \alpha(pmv)\ o\ distr\ o\ [\#1, trans\ o\ \#2]$$

Plan

1 Les langages

2 Les paradigmes
- Modèle impératif abstrait
- Boehm et Jacopini
- Programmation impérative
- Programmation fonctionnelle
- Programmation en logique

Le Paradigme Fonctionnel

Né en même temps que le paradigme impératif, le paradigme fonctionnel est issu de la théorie mathématique du λ-calcul d'A. Church (1930). Il a été développé à la fin des années 50 par J. McCarthy avec le langage Lisp puis repris dans de nombreux langages dont Scheme et ML.

Dans ces principes, on a l'utilisation de *fonctions* pour programmer :

```
(defun f(x) (+ x 1))
```

Thèse de Church : avec des fonctions, on peut calculer tout ce qui est calculable.

Le Paradigme Fonctionnel

Le principe de base de la programmation est la récursivité. Il ne viendrait jamais à l'idée d'un programmeur fonctionnel "compétent" de programmer avec une boucle...

```
(defun f91(x)
  (if (> x 100)
    (- x 10)
    (f91 (f91 (+ x 11)))))
```

```
int f91(int x) {
  if (x > 100)
    return x-10 ;
  return f91(f91(x+11)) ;
}
```

```
(def tak(x y z)
  (if (<= x y)
    y
    (tak (tak (- x 1) y z)
         (tak (- y 1) z x)
         (tak (- z 1) x y))))
```

```
int tak(int x, int y, int z) {
  if (x <= y)
    return y ;
  return tak(tak(x-1,y,z),
             tak(y-1,z,x),
             tak(z-1,x,y)) ;
}
```

```
(def fact(n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))
```

```
int fact(int n) {
  return
     (n == 0)
     ?1
     :(n * fact(n-1)) ;}
```

Programme = Donnée

```
? (defun twice(f x) (apply f (apply f x)))
= twice
? (defun z(x) (+ x 1))
= z
? (twice 'z 1)
= 3
```

Un programme peut être donné de manière anonyme :

```
? (twice '(lambda (x) (+ x 1)) 1)
= 3
```

`(lambda (x) (+ x 1))`, soit $(\lambda x.x+1)$ dans la notation mathématique du λ-calcul, est une fonction anonyme.

La structure de liste

La structure de liste est une structure de listes chaînées qui sont construites à l'aide de doublets de listes appelés *cons* :

- `nil` ou `()` est la liste de vide ;
- `(cons x y)` permet de construire un doublet dont le premier éléments est `x` et le deuxième est `y` ;
- `(car d)` permet d'extraire la première composante d'un doublet ;
- `(cdr d)` permet d'extraire la seconde composante d'un doublet ;
- `(null x)` permet de distinguer un doublet de la liste vide.

Le Paradigme Fonctionnel

Un doublet est noté `(a . b)`

Une liste est notée `(a b c d)`

C'est en fait `(a . (b . (c . (d .()))))`

On parle de :

- tête ou `car` de liste pour le premier élément ;
- et de queue ou `cdr` de liste pour la liste privée de son premier éléement.

```
? (defun last(x)
    (if (null (cdr x))
      (car list)
      (last (cdr x))))
= last
? (last '(a b c d))
= d
```

NB. Les programmes sont des listes.

Le Paradigme Fonctionnel

Les listes sont gérées dynamiquement

Elles sont allouées par `cons`.

Elles sont récupérées automatiquement.
Garbage Collector

NB. Filiation : Lisp $\Rightarrow$ SmallTalk $\Rightarrow$ Java.

La Réflexivité

- Lisp est écrit en Lisp.
- Lisp est réflexif.

```
(defun toplevel()
  (forever
     (print (eval (read)))))
```

Tout y est redéfinissable.

- Redéfinir `read` et `print` permet de changer la *syntaxe* du langage.
- Redéfinir `eval` permet de redéfinir la *sémantique* du langage.

La Réflexivité

Gag de première année de Lisp:

```
? (defun eval(x) 'IMBECILE)
= eval
? (+ x 1)
= IMBECILE
? (quit)
= IMBECILE
```

NB. SmallTalk est un langage réflexif.
NB. Java est un langage partiellement réflexif.

Extension du langage lui-même

Toujours la Réflexivité

Exempe ne fonctionnant pas car c'est légèrement plus compliqué à réaliser mais on n'est pas là pour apprendre Lisp !!!

```
(defun ifn(p f g) (if p g f))
```

Redéfinition dynamique

Toujours la Réflexivité

L'imprimeur `print` est normalement écrit en Lisp et utilise la fonction primitive `print-char` chaque fois qu'il veut imprimer un caractère...
L'imprimeur `pretty-print` a la même fonction de `print` mais il effectue une impression plus jolie avec des indentations et tient compte de la longueur de ligne à l'affichage. Pour cela, il doit calculer la longueur d'impression, i.e. le nombre de caractères, des expressions qu'il imprime.

```
(defun print-length (expr)
   (let ((compteur 0))
      (flet ((print-char (c) (incr 'compteur)))
         (print expr)
         compteur)))
```

Les inventions de Lisp

Lisp a été un formidable laboratoire !

La réflexivité.

La récursivité.

Les listes.

L'interprétation.

Les fonctions, objets de 1ère classe.

Les opérateurs (un peu comme dans les systèmes FP).

Le Garbage Collector.

Les exceptions.

Les machines virtuelles.

SmallTalk et la POO.

L'évaluation partielle.

Mais la normalisation de Lisp a tué Lisp !

Les caractéristiques du fonctionnel

Le programmeur ne gère pas l'exécution de son programme.
— FAUX DANS LA PRATIQUE —
Structure de contrôle impérative. Ordre d'évaluation.

Le programmeur ne gère pas l'utilisation de la mémoire.
— VRAI —
Mais le GC n'est pas gratuit.

La programmation est plus rigoureuse, naturelle et facile.
— VRAI SI ON LE VEUT —
Mais la même discipline peut être utilisée dans d'autres langages. Même en C...

Le Paradigme Fonctionnel

Applications

- `xemacs/emacs` est écrit en Lisp.
- `SmallTalk` a été écrit en Lisp.
- De nombreux systèmes experts sont écrits en Lisp.
- Des programmes d'apprentissage sont écrits en Lisp.
- En règle générale, l'I.A. (à qui Lisp doit beaucoup), lorsqu'elle n'a pas pas particulièrement besoin de performances, aime bien utiliser Lisp.

Plan

1 Les langages

2 Les paradigmes
- Modèle impératif abstrait
- Boehm et Jacopini
- Programmation impérative
- Programmation fonctionnelle
- Programmation en logique

Le Paradigme Logique

Née à Marseille dans les années 70, la programmation en logique permet d'utiliser une catégorie restreinte de formules logiques pour programmer : les *clauses de* Horn.

Clause de Horn

$$\text{Si } q_1(x,\ldots) \text{ et } \cdots \text{ et } q_n(x,\ldots) \text{ alors } p(x,\ldots)$$
$$q_1(x,\ldots) \wedge \cdots \wedge q_n(x,\ldots) \Rightarrow p(x,\ldots)$$
$$p(x,\ldots) \text{ si } q_1(x,\ldots) \text{ et } \cdots \text{ et } q_n(x,\ldots)$$
$$p(x,\ldots) \Leftarrow q_1(x,\ldots) \wedge \cdots \wedge q_n(x,\ldots)$$

Clause Prolog

```
p(x,...)  :- q1(x,...), ...  , qn(x,...).
```

N.B. Colmerauer 1973 et Kowalski 1974.

Langages de l'informatique

Patrick Bellot

Les langages

Les paradigmes

Modèle impératif abstrait

Boehm et Jacopini

Programmation impérative

Programmation fonctionnelle

Programmation en logique

On programme des Relations entre des données représentées par des atomes, symboles et nombres, et des termes (fonctionnels), exemple : `voiture(jean)`

Des Faits :

```
pere(patrick,jerome).
pere(patrick,helene).
pere(patrick,camille).
pere(patrick,daniel).
mere(marianne,jerome).
mere(marianne,helene).
mere(fadila,camille).
mere(lam,daniel).
```

Des Relations :

```
parent(X,Y) :- pere(X,Y).
parent(X,Y) :- mere(X,Y).

grand-pere(X,Y) :- pere(X,Z), parent(Z,Y).
grand-mere(X,Y) :- mere(X,Z), parent(Z,Y).
```

Le Paradigme Logique

Langages de l'informatique

Patrick Bellot

Les langages

Les paradigmes

Modèle impératif abstrait

Boehm et Jacopini

Programmation impérative

Programmation fonctionnelle

Programmation en logique

Symbole

Un symbole commence par une lettre minuscule.

Variable

Une variable commence par une lettre majuscule.

Termes Fonctionnels

Un terme fonctionnel est symbole de fonction appliqué à d'autres termes. Il n'a pas de valeur. Il sert à désigner. **Exemple :** on décide que le terme `voiture(X)` désigne la voiture de `X`.

```
couleur(voiture(patrick), bleu).
```

Les buts (ou queries ou question)

Tester :

```
?- pere(patrick,camille).
OK
?- pere(patrick,julie).
NO
```

Chercher :

```
?- pere(patrick,X).
X=jerome, OK
X=helene, OK
X=camille, OK

?- couleur(voiture(X),bleu).
X=patrick, OK

?- couleur(voiture(X),rose).
NO
```

Le Paradigme Logique

La méthode de résolution (démonstration) des buts de Prolog s'appelle la SLDNF-résolution.
Selection, Linear, Definite, Negation as failure

- Basée sur l'*Unification*. Unifier deux termes, c'est touver les valeurs aux variables qu'ils contiennent pour que les deux termes soient syntaxiquement identiques.

  Exemple: `g(f(X,Y),Z,2,U)` et `g(f(1,3),Y,T,V)`
  L'unification donne : `X=1`, `Y=Z=3`, `T=2`, `U=V`

  C'est un algorithme très efficace !

- Résolution d'un but `p(a,b)` avec une clause `p(x,y) :- q1(...), ..., qn(...).`:
  - Unification du but `p(a,b)` avec la tête de clause `p(x,y)`.
  - Si succès, démonstration de `q1(...), ..., qn(...)` dans l'environnement résultant de l'unification.

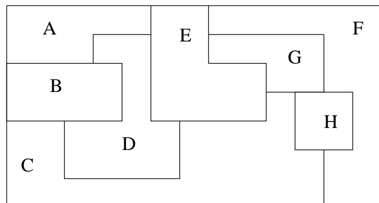
Le Paradigme Logique

La méthode de résolution (démonstration) des buts de Prolog s'appelle la SLDNF-résolution.
Selection, Linear, Definite, Negation as failure

- Négation par l'échec : ce qui n'est pas vrai est faux (hypothèse du monde clos). Ce n'est pas la négation logique.
- Résolution utilisant l'ordre d'enregistrement des clauses et des faits : rien de logique. Le programmeur doit faire très attention à l'ordre des clauses.
- Nécessité d'outils de contrôle impératif :
  - le Cut qui détruit la branche de recherche en cours ;
  - le *gel* qui permet de retarder la résolution d'un but intermédiaire jusqu'à ce qu'une variable soit liée à une valeur
- Pas de calcul : pas d'arithmétique.

  D'où la nécessité d'ajouts impératifs comme `Z is X+Y`.

Le coloriage de carte



Comment colorier une carte avec 4 couleurs (r,j,v,b) de telle manière que 2 zones contiguës aient des couleurs différentes ?

Le coloriage de carte

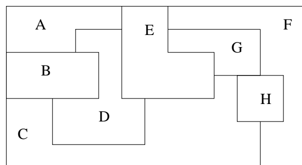
Définir les couleurs

`color(C)` est un prédicat signifiant que `C` est une couleur pour la carte. Il est défini par 4 faits :

```
color(r).
color(j).
color(v).
color(b).
```

Un but `color(X)` a quatre solutions qui seront trouvée dans 4 branches de démonstration différentes correspondant aux 4 faits ci-dessus.

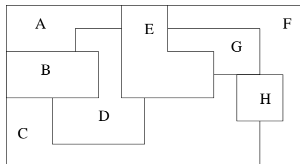
Le coloriage de carte



Poser le problème

`carte(A,B,C,D,E,F,G,H)` est un prédicat signifiant que `A,B,C,D,E,F,G,H` sont des couleurs contituant un coloriage admissible pour la carte: la couleur `A` coloriant la zone marquée "A", la couleur `B` coloriant la zone marquée "B", etc.

Le coloriage de carte



Algotithmique Generate-and-test

- **Phase Generate.** Le programme génère toutes les solutions envisageables, c'est-à-dire tous les coloriages possibles. Pour cela, on utilise le prédicat `color(X)` qui génère 4 branches de démonstration pour chacune des valeurs possibles de `X`.
- **Phase Test.** Le programme contrôle si la solution générée est conforme. Pour cela, il suffit de tester si les couleurs de deux zones contiguës sont bien différentes.

Langages de l'informatique

Patrick Bellot

Les langages

Les paradigmes

Modèle impératif abstrait

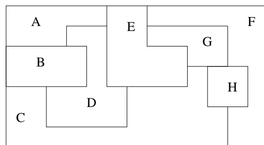
Boehm et Jacopini

Programmation impérative

Programmation fonctionnelle

Programmation en logique

Le coloriage de carte




```
carte(A,B,C,D,E,F,G,H) :-

   color(A), color(B), color(C), color(D),
   color(E), color(F), color(G), color(H),

   A /= B, A /= D, A /= E,
   B /= C, B /= D,
   C /= D, C /= E, C /= F, C /= G, C /= H,
   D /= E,
   E /= F, E /= G,
   F /= G, F /= H,
   G /= H.
```

L'algorithme est juste mais il y a un problème. Lequel ?

Le coloriage de carte

Solution: Mixer Generate and Test pour couper plus tôt les branches de la démonstration.

```
carte(A,B,C,D,E,F,G,H) :-
   color(A),
   color(B), A /= B,
   color(C), B /= C,
   color(D), A /= D, B /= D, C /= D,
   color(E), A /= E, C /= E, D /= E,
   color(F), C /= F, E /= F,
   color(G), C /= G, E /= G, F /= G,
   color(H), C /= H, F /= H, G /= H.
```

L'algorithme est juste et le problème algorithmique est résolu.

Cependant, la programmation n'est plus totalement logique.

Le Paradigme Logique

Les contraintes
Dernier grand avatar de Prolog

Objectif : être le plus logique possible en gardant la résolution SLDNF et fournir un outil puissant.

Domaines Finis

On déclare que les variables concernées appartiennent à des domaines finis, ex : `1..34`. On pose des contraintes sur ces variables, ex : `X >= 0`, `X+Y=0`...

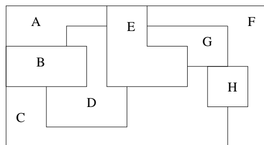
Domaines Continus

Essentiellement des contraintes sur les nombres réels.

Résolution

Un moteur extérieur.

Le coloriage de carte



Avec des contraintes:

```
carte(A,B,C,D,E,F,G,H) :-

   A /= B, A /= D, A /= E,
   B /= C, B /= D,
   C /= D, C /= E, C /= F, C /= G, C /= H,
   D /= E,
   E /= F, E /= G,
   F /= G, F /= H,
   G /= H

   color(A), color(B), color(C), color(D),
   color(E), color(F), color(G), color(H),.
```

L'algorithme redevient logique.

Le Paradigme Logique

Langages de l'informatique

Patrick Bellot

Les langages

Les paradigmes

Modèle impératif abstrait

Boehm et Jacopini

Programmation impérative

Programmation fonctionnelle

Programmation en logique

Les remarques qui s'appliquent à Lisp s'appliquent également à Prolog.

On programme rarement en purement logique.

Prolog est particulièrement adaptés aux stratégies d'essai-erreur, aux problèmes de recherche arborescente, aux problèmes de planification, aux systèmes experts, etc. Toute une algorithmique spécifique.

Prolog est aussi un langage réflexif.