UTC503 : Paradigmes de programmation

—

Paradigme déclaratif

—

Illustration avec Prolog (programmation logique) et la commande make et ses Makefile

Xavier Crégut
<Prénom.Nom@enseeiht.fr>

ENSEEIHT
Sciences du Numérique

Objectifs

- Comprendre les principes du paradigme déclaratif
- Quelques exemples...
- Illustration 1 : La programmation logique (Prolog)
  - 1972 : A. Colmerauer et P. Roussel développent le langage Prolog qui permet :
    - de modéliser un monde
    - de poser des questions sur ce monde
  - GNU Prolog, SWI-Prolog (version en ligne : `https://swish.swi-prolog.org/`), etc.
  - lancement de l'interpréteur SWI-Prolog : swipl
  - Extension des fichiers Prolog : .pl
- Illustration 2 : La commande make et ses fichiers Makefile
  - 1977 : Stuart Feldman (Bell Labs) crée make pour accélérer la production des exécutables.
  - optimiser la création de documents (exécutables souvent) en ne refaisant que ce qui est nécessaires pour prendre en compte les modifications réalisées depuis la dernière mise à jour.

Paradigme déclaratif : principes et exemples

Définition (wikipedia) : La programmation déclarative est un paradigme de programmation qui exprime la logique d'un calcul sans en décrire le flot de contrôle.

Exemples :

- Le **langage HTML** décrit la structure d'un document (titre, sections, paragraphes, listes...) mais pas la manière de le mettre en forme (gras, italique, couleur, taille, etc).
- Le **langage SQL** permet d'exprimer des requêtes sur une base de données sans dire comment les données seront extraites et combinées pour une BD particulière.
- Le **fichier Makefile** de la commande `make` donne les dépendances entre fichiers et les actions pour les mettre à jour sans dire dans quel ordre ces actions seront exécutées.
- Les **menus textuels** proposés par le module menus.py :

```
from menus import menu, ajouter_entree, gerer
def exemple_menu():
    def cmd_a(): print('Je fais A')
    def cmd_b(): print('Je fais B')
    principal = Menu()          # Constuire le menu
    ajouter_entree(principal, "opération A", cmd_a)
    ajouter_entree(principal, "opération B", cmd_b)
    gerer(principal)            # Le lancer
```

```
1) Opération A
2) Opération B
0) Quitter
Votre choix : 1
Je fais A

1) Opération A
2) Opération B
0) Quitter
Votre choix :
```

Le flot de contrôle est décrit dans la fonction `gerer` du module menus.py.

Sommaire

Programmation logique : principe

La programmation logique est :

- un paradigme de programmation fondé sur la logique mathématique
- un programme contient des faits et des règles
- exécuter un programme consiste à répondre à des questions
- l'exécution est un raisonnement automatique sur ces faits et ces règles (une preuve)

Vite ! Un exemple !

Un exemple en Prolog

Des faits et quelques questions informelles

Voici des **faits** :

```
parent(zoe, lea).
parent(zoe, tom).
parent(luc, lea).
parent(luc, tom).
parent(luc, leo).
parent(isa, bob).
parent(isa, leo).

parent(eric, luc).
parent(eric, sam).
parent(lisa, luc).
parent(lisa, sam).
```

- `parent` est un **predicat** d'arité 2
- Il faut lui donner du sens. On lit :
  `parent(Le_parent, L_enfant).`
  `Le_parent` est un parent de `L_enfant`.
  (au sens père et mère !)

Et des **questions** :

1. Est-ce que zoe est un parent de sam ?
2. Est-ce que lisa est un parent de sam ?
3. Est-ce que luc est un parent de tom ?
4. Quels sont les enfants de luc ?
5. Qui sont les parents de leo ?
6. Qui sont les parents de lisa ?

Un exemple en Prolog

Les questions en Prolog

Les faits (rappel) :

```
parent(zoe, lea).
parent(zoe, tom).
parent(luc, lea).
parent(luc, tom).
parent(luc, leo).
parent(isa, bob).
parent(isa, leo).

parent(eric, luc).
parent(eric, sam).
parent(lisa, luc).
parent(lisa, sam).
```

Tout ce qui ne pas être prouvé vrai est faux !

Les questions en Prolog et les réponses...

```
?- parent(zoe, sam).
false.

?- parent(lisa, sam).
true.

?- parent(luc, tom).
true ;
false.

?- parent(luc, Enfant).
Enfant = lea ;
Enfant = tom ;
Enfant = leo.

?- parent(Parent, leo).
Parent = luc ;
Parent = isa.

?- parent(Parent, lisa).
false.
```

Un exemple en Prolog

Des règles

Voici des exemples de **règles** : (forme : `tête :- corps.`)

```
grandparent(GP, PE) :- parent(GP, Personne), parent(Personne, PE).
ancetre(Ancetre, Personne) :- parent(Ancetre, Personne).
ancetre(Ancetre, Descendant) :- parent(Ancetre, P), ancetre(P, Descendant).
```

Ils utilisent des **variables** (**inconnues**). Elles commencent par une majuscule.

```
?- grandparent(G, tom).
G = eric ;
G = lisa ;
false.
```

```
?- grandparent(lisa, E).
E = lea ;
E = tom ;
E = leo ;
false.
```

```
?- ancetre(A, tom).
A = zoe ;
A = luc ;
A = eric ;
A = lisa ;
false.
```

```
?- ancetre(eric, D).
D = luc ;
D = sam ;
D = lea ;
D = tom ;
D = leo ;
false.
```

Interprétation d'une règle

```
ancetre(A, D) :- parent(A, P), ancetre(P, D).
```

- Pour tout A, pour tout D, il existe un P tels que *quantificateurs*
  si A est parent de P et P est ancêtre de D alors A est ancêtre de D.
- $\forall A \forall D \exists P.(\text{parent}(A, P) \wedge \text{ancetre}(P, D) \implies \text{ancetre}(A, D))$
  Rq : Quantification universelle des variables de la tête et existentielle de celles du corps.

Utilisation de SWI Prolog

- Pour le lancer : swipl fichier.pl
- Pour quitter l'interpréteur : halt. ou CTRL-D
- Pour charger un fichier prolog : `["fichier.pl"].`
- Pour définir des faits ou des règles sous l'interpréteur : `[user].` puis CTRL-D pour terminer.
- Pour avoir de l'aide : help.
- Quand une question est posée, « espace » permet de passer à la solution suivante, « entrée » arrête sur cette solution.
- Pour tout tracer : `trace.` (faire espace pour avancer). `notrace.` pour arrêter.
- `listing.` : affiche les règles d'un prédicat (e.g. `listing(parent).`) ou de tous (`listing.`). Voir `apropos`.

Exercice 1 Essayer les questions suivantes avec le programme famille.pl

```
?- fille(lea).          ?- fille(F).              % qui sont les ancêtres de lisa ?
?- garcon(lea).         ?- parent(luc, Enfant).   % qui sont les descendants de lisa ?
?- parent(isa, bob).    ?- parent(Parent, tom).   % qui sont grands parents de sam ?
?- parent(bob, isa).    ?- parent(Parent, eric).  % qui sont les petis-enfants luc ?
?- parent(bob, sam).    ?- parent(P, E).          % les petits-enfants de moins de 8 ans ?
```

Rq : Le prédicat `age(Personne, Age)` donne l'age (en années) d'une personne.

Les termes

Le « programme » est composé de termes.

Un terme peut être :

- Une **variable** (var/1), une **inconnue** : le nom commence par une majuscule ou un souligné
  - Exemples : X, Y, P, Personne, Parent, Enfant, Age, _, _N, X1, etc.
- un **élément atomique** (atomic) :
  - un **atome** (atom/1) : un nom symbobolique (commence par une minuscule)
    - exemples : luc, lea, etc.
  - un **nombre** (number/1) : **entier** (integer/1) ou **réel** (float/1)
  - les chaînes de caractères :
    - Exemple : "F305"
- un **élément composé** (compound/1) : un nom symbolique avec une arité
  - lie plusieurs termes par une relation sémantique
  - l'arité indique le nombre de paramètres
  - convention : verbe(sujet, complement...)
  - exemples : parent(luc, tom) age(tom, 10) f(X, g(h(a))) avec parent/2, age/2, f/2, g/1, h/1

Remarque : nonvar/1 correspond à atomic/1 et compound/1.

Unification

Dans un terme on peut substituer une variable par un terme (ne contenant pas cette variable). Prolog applique ce principe pour répondre aux questions : **unification**.

Unification : Deux termes sont unifiables ssi il existe une ou plusieurs substitutions qui, appliquées dans le même ordre sur les deux termes donnent un même terme.

Exemple 1 : peut-on unifier les termes suivants ?

1. $f(X, g(Y))$ et $f(a, g(f(b)))$? oui avec $X = a$ et $Y = f(b)$.
2. $f(a, g(X))$ et $f(X, g(b))$? non il faudrait $X = a$ et $X = b$.

Principe intuitif : Comparer les deux termes vues comme des arbres :

- si un nœud est une variable, cette variable est substituée par le terme de l'autre arbre
- sinon les deux nœuds doivent être identiques : même arité et fils unifiables

Remarque : L'unification se note $=$ en Prolog.

Exercice 2 Les termes suivants sont-ils unifiables ?

```
?- 3 = 1 + 2.
?- f(a, f(b), g(X)) = f(Y, X, g(f(b))).
?- g(g(b), g(Y)) = g(X, g(c)).
?- g(a, g(b), g(Y)) = g(Y, X, g(c)).
?- f(X, f(Y)) = f(a, Y).
?- g(a, f(b)) = f(a, f(Y)).
```

Règle, fait, question

Règle (rule) : Une règle est de la forme « `tête :- corps.` ».

- tête (head) est un prédicat.
- corps (body) est composé de plusieurs prédicats séparés par des virgules.
- Ne pas oublier le point à la fin !

Forme générale : `T :- C1, C2, ..., Cn.`

Intuitivement :

- (déduction) si C1 et C2 et ... Cn sont vrais alors T est vrai.
- (preuve) pour prouver T, il faut prouver C1 et prouver C2... et prouver Cn.

Vision mathématique : si X_i variables de T et Y_j variables de C_k :

- $\forall X_i \exists Y_j.(C_1 \wedge C_2 ... \wedge C_n \implies T)$
- ou $\forall X_i \exists Y_j.(\neg C_1 \vee \neg C_2 ... \vee \neg C_n \vee T)$ C'est une clause de Horn

Fait (fact) : Une règle sans corps.

Question (query) : Une règle dans tête.

Exercices

Exercice 3 : Famille

3.1. Définir un prédicat qui dit si une personne est mineure. On pourra utiliser N1 < N2.
3.2. Définir un prédicat qui dit si une personne est majeure.
3.3. Définir un prédicat qui permet de lier une mère et son enfant.
3.4. Définir un prédicat qui permet de connaitre les oncles.

Exercice 4 : Films de cinéma
Répondre aux questions qui sont dans movies.pl.

Fonctionnement de prolog

Prolog est un langage déclaratif :

1. les faits et les règles décrivent le monde
2. les questions posent des questions sur ce monde

Principales popriétés :

1. En prolog on décrit le QUOI et pas le COMMENT.
2. Pas de notion de changement d'état (comme en fonctionnel).
   - pas d'affectation mais la possibilité d'introduire de nouvelles variables
   - le symbole = correspond à l'unification.
   - `is` permet de lier une variable à un entier : `N1 is N + 1`.
3. Forme d'écriture des règles qui se rapproche du fonctionnel (récursivité / induction) Attention, ici des relations plutôt que des fonctions.

Exécution : évaluation d'une question :

1. Prolog définit une procédure (déduction ou inférence) qui indique comment sont exploitées les règles et les faits pour évaluer une question.
2. Trois réponses possibles à une question : vrai (true), échec (fail), ne termine pas !
3. Pour une réponse « vrai », une substitution (unification) pour les variables est indiquée.
4. Cherche toutes les solutions via mécanisme de backtracking.

Évaluation sur un exemple

```
1   f(a) :- f(e), f(f).
2   f(a) :- f(g), f(h).
3   f(a) :- f(b).
4   f(a) :- f(d).
5   f(b) :- f(c).
6   f(c) :- f(d).
7   f(c) :- f(e).
8   f(c).
9   f(g).
```

1. les règles sont considérées dans l'ordre
   - 1, puis 2, puis 3, puis 4 pour f(a).
2. termes d'un corps évalués de gauche à droite
   - pour 1, f(e), puis f(f).
   - pour 2, f(g), puis f(h).
   - ...
3. si un terme échoue (Fail), la règle échoue
   on essaie avec la suivante (Redo) : bactracking
4. si tous les termes sont vérifiés, une solution trouvée
5. s'il reste des règles, on continue

```
[trace]  ?- f(a).
   Call: (8) f(a) ? creep
   Call: (9) f(e) ? creep
   Fail: (9) f(e) ? creep
   Redo: (8) f(a) ? creep
   Call: (9) f(g) ? creep
   Exit: (9) f(g) ? creep
   Call: (9) f(h) ? creep
   Fail: (9) f(h) ? creep
   Redo: (8) f(a) ? creep
   Call: (9) f(b) ? creep
   Call: (10) f(c) ? creep
   Call: (11) f(d) ? creep
   Fail: (11) f(d) ? creep
   Redo: (10) f(c) ? creep
   Call: (11) f(e) ? creep
   Fail: (11) f(e) ? creep
   Redo: (10) f(c) ? creep
   Exit: (10) f(c) ? creep
   Exit: (9) f(b) ? creep
   Exit: (8) f(a) ? creep
true ;
   Redo: (8) f(a) ? creep
   Call: (9) f(d) ? creep
   Fail: (9) f(d) ? creep
   Fail: (8) f(a) ? creep
false.
```

Prédicats prédéfinis

prédicats sur les atomes :

```
T1 ==  T2
T1 \== T2
T1 @<  T2
T1 @=< T2
T1 @>= T2
T1 @>  T2
```

prédicats sur les entiers :

```
N1 =:=  N2
N1 =\= N2
N1  <  N2
N1  =< N2
N1  >= N2
N1  >  N2
```

Remarque : @< (ou ses variantes) est utile quand on a une relation symétrique, par exemple `conjoint` ou `frere_ou_soeur` et que l'on ne veut pas avoir les deux couples (a, b) et (b, a).

Prédicat dif : Il existe aussi le prédicat `dif` pour la différence qui est plus général que \==. Les deux ont même comportement s'il n'y a pas de variables dans les termes comparés.
Voir `https://stackoverflow.com/questions/13757261/using-2-or-dif-2`

```
?- dif(X, Y), X = Y.          false.
?- X = Y, dif(X, Y).          false.

?- X = Y, X \== Y.            false
?- X \== Y, X = Y.            X = Y.
```

Min, max et factorielle

Exercice 5 Écrire un prédicat min(A, B, Min), vrai si Min est le plus petit de A et B.
Définir aussi un prédicat max.

Exercice 6 Écrire un prédicat qui calcule un factorielle pour un entier naturel donné.

Exercice 7 : Expressions arithmétiques
7.1. Comment représenter les expressions arithmétiques de la forme constante, somme de deux expressions ou différence de deux expressions ?
7.2. Représenter e0 = 1, e1 = 4 - 5 et e2 = (1 + 2) - (3 + (4 - 5)).
7.3. Écrire un prédicat qui évalue une expression arithmétique. La valeur de e0 est 1, celle de e1 est -1 et celle de e2 est 1.

Coupe-choix (cut)

Première solution :

```
min1(A, B, M) :- A < B, A = M.
min1(A, B, M) :- A >= B, B = M.
```

```
?- min1(8, 5, R).
R = 5.
```

```
?- min1(2, 4, R).
R = 2 ;
false.
```

```
?- min1(X, 4, 2).
ERROR: Arguments are
sufficiently instant
```

Pour `min1(2, 4, R).`, la recherche continue pour essayer de trouver d'autres solutions.
Il n'y en a pas. On pourrait donc stopper la recherche dès la (première) solution trouvée.

Coupe-choix (cut) : Le coupe-choix (*cut* en anglais), noté ! en Prolog, est un prédicat qui s'évalue à vrai et interdit tout backtracking pour la tête de cette règle.

Deuxième version :

```
min2(A, B, M) :- A < B, !, A = M.
min2(_, B, M) :- B = M.
```

```
?- min2(8, 5, R).
R = 5.
```

```
?- min2(2, 4, R).
R = 2.
```

```
?- min2(X, 4, 2).
ERROR: ...
```

Plus le test `A >= B` en raison du coupe-choix. Pourquoi unifier dans le corps de la règle ?

Troisième version :

```
min(A, B, A) :- A < B, !.
min(_, B, B).
```

```
?- min(8, 5, R).
R = 5.
```

```
?- min(2, 4, R).
R = 2.
```

```
?- min(X, 4, 2).
X = 2.
```

Définition de Max : On s'appuie sur min !

```
max(A, B, R) :- min(A, B, Min),
                R is A + B - Min.
```

```
?- max(8, 5, R).
R = 8.
```

```
?- max(2, 4, R).
R = 4.
```

```
?- max(X, 4, 8).
ERROR: ...
```

Les listes

Principe : Une liste est définie comme une tête et une queue (suite) :

- La liste vide : `[]`
- Opérateur d'ajout en tête `[tête | queue]`

```
?- X = [5 | []].                % X = [5].
?- X = [1 | [5 | []]].          % X = [1, 5].
?- X = [1, 2 | []].             % X = [1, 2].
?- X = [[1 |[]]  | [2| []]].    % X = [[1], 2].
```

- Notation plus pratique : [1, 2, a, "xxx"] (liste de 4 éléments)
- Unification : `?: [E1, E2 | Q] = [1, 2, 3, 4]` donne `E1 = 1, E2 = 2, Q = [3, 4].`

Fréquence

```
% frequence(X, L, F) : F est la fréquence de X dans L, une liste.
frequence(_, [], 0).
frequence(X, [X|Q], F) :- frequence(X, Q, F1), F is F1 + 1.
frequence(X, [Y|Q], F) :- dif(X, Y), frequence(X, Q, F).
```

Quelles réponses aux questions suivantes ?

```
?- frequence(1, [0, 1, 2, 0], F).
?- frequence(4, [0, 1, 2, 0], F).
?- frequence(0, [0, 1, 2, 0], F).
?- frequence(X, [0, 1, 2, 3, 0, 3], 2).
?- frequence(X, [0, 1, 2, Y, 0, 3], 3).
```

Les listes (2)

Appartenance

```
% dans(X, L) : X est dans L, une liste.
dans(X, [X|_]).
dans(X, [_|Q]) :- dans(X, Q).
```

Quelles réponses aux questions suivantes ?

```
?- dans(1 ,[0, 1, 2, 0]).
?- dans(4 ,[0, 1, 2, 0]).
?- dans(0 ,[0, 1, 2, 0]).
?- dans(X ,[0, 1, 2, 0]).
?- L = [0, 1, 2, 0], dans(X, L), frequence(X, L, F).
```

Quelques prédicats sur les listes

```
member(?Elem, ?List)          % True if Elem is a member of List
append(?List1, ?List2, ?List1AndList2)
                              % List1AndList2 is the concatenation of List1 and List2
select(?Elem, ?List1, ?List2) % Is true when List1, with Elem removed, results in List2.
nextto(?X, ?Y, ?List)         % True if Y directly follows X in List.
nth0(?Index, ?List, ?Elem)    % True when Elem is the Index'th element of List. Counting starts at 0.
nth1(?Index, ?List, ?Elem)    % True when Elem is the Index'th element of List. Counting starts at 1.
last(?List, ?Last)            % Succeeds when Last is the last element of List.
reverse(?List1, ?List2)       % Is true when the elements of List2 are in reverse order compared to List1.
permutation(?Xs, ?Ys)         % True when Xs is a permutation of Ys.
max_member(-Max, +List)       % True when Max is the largest member in the standard order of terms.
sum_list(+List, -Sum)         % Sum is the result of adding all numbers in List.
max_list(+List:list(number), -Max:number) % True if Max is the largest number in List. Fails if List is empty.
min_list(+List:list(number), -Min:number) % True if Min is the smallest number in List. Fails if List is empty
intersection(+Set1, +Set2, -Set3) % True if Set3 unifies with the intersection of Set1 and Set2.
...
```

https://www.swi-prolog.org/pldoc/man?section=lists

Conclusion

- déclaratif : on décrit le problème à résoudre (pas la manière de le résoudre)
  - Attention, l'ordre des règles a une importance (ordre de prise en compte, coupe-choix)
  - L'ordre des termes a une importance (évaluation de gauche à droite)
    Remarque : Dans une approche purement déclarative, l'ordre ne devrait pas avoir d'importance.
- procédure d'« exécution » : elle explique comment Prolog répond aux questions.
- Pas d'état, pas d'affectation.
- Mécanisme d'induction proche de la récursivité / induction des langages fonctionnels
- Dans une approche purement déclarative, tout paramètre d'un prédicat pourrait être une inconnue (un résultat à calculer dans une approche fonctionnelle).

Sommaire

Exemple de programme C

carre.h
```c
/* Module définissant la fonction carre. */
#ifndef CARRE__H
#define CARRE__H
extern int carre(int nombre);
  /* Le carré de nombre. */
#endif
```

carre.c
```c
#include "carre.h"
int carre(int nombre) {
  return nombre * nombre;
}
```

cube.h
```c
/* Module définissant la fonction cube. */
#ifndef CUBE__H
#define CUBE__H
extern int cube(int nombre);
  /* Le cube de nombre. */
#endif
```

cube.c
```c
#include "cube.h"
#include "carre.h"
int cube(int nombre) {
  return nombre * carre(nombre);
}
```

test_carre.c
```c
#include <assert.h>
#include <stdlib.h>
#include "carre.h"
void tester_carre() {
    assert(4 == carre(2));
    assert(0 == carre(0));
    assert(9 == carre(-3));
}
int main() {
    tester_carre();
    return EXIT_SUCCESS;
}
```

test_cube.c
```c
#include <assert.h>
#include <stdlib.h>
#include "cube.h"
void tester_cube() {
    assert(8 == cube(2));
    assert(0 == cube(0));
    assert(-27 == cube(-3));
}
int main() {
    tester_cube();
    return EXIT_SUCCESS;
}
```

ihm.c
```c
#include <stdlib.h>
#include <stdio.h>
#include "carre.h"
#include "cube.h"
int main() {
  int n;
  printf("Sasir un nombre : ");
  int nb = scanf("%i", &n);
  if (nb != 1) {
    printf("Je n'ai pas compris. Désolé.\n");
  } else {
    printf("Le carré de %i est %i.\n", n, carre(n));
    printf("Le cube  de %i est %i.\n", n, cube(n));
  }
  return EXIT_SUCCESS;
}
```

Comment compiler ces programmes ?

En compilant tous les fichiers .c nécessaires ensemble :

```
c99 -Wall -pedantic carre.c test_carre.c -o test_carre
c99 -Wall -pedantic carre.c cube.c test_cube.c -o test_cube
c99 -Wall -pedantic carre.c cube.c ihm.c -o ihm
```

Est-ce raisonnable pour une application avec des milliers de fichiers ?

Avec la compilation séparée.

```
-- compilation des fichiers .c individuellement (produit un .o)
c99 -c -Wall -pedantic carre.c
c99 -c -Wall -pedantic cube.c
c99 -c -Wall -pedantic test_carre.c
c99 -c -Wall -pedantic test_cube.c
c99 -c -Wall -pedantic ihm.c

-- production des exécutables (éditeur de lien)
c99 carre.o test_carre.o -o test_carre
c99 carre.o cube.o test_cube.o -o test_cube
c99 carre.o cube.o ihm.o -o ihm
```

Et maintenant ?

1. Que faire si on change ihm.c (pour améliorer le dialogue avec l'utilisateur) ?
2. Que faire si on change carre.c (pour améliorer la fonction) ?
3. Que faire si on change cube.h (pour améliorer la fonction) ?
4. Que faire si on change carre.h (pour améliorer la fonction) ?

Analyse du problèmes

Démarche :

1. Quelles sont les dépendances (utilisation, dépend de) entre les différents fichiers ?
2. Que faire pour mettre à jour un fichier ?
3. Comment sait-on qu'une compilation ou une édition des liens échoue ?
4. Et si on veut changer les options de compilation (ajouter -g, etc.)
5. Peut-on éviter d'écrire des choses redondantes ?
6. Et pour nettoyer le dossier de travail de tous les fichiers engendrés ?

Premier Makefile

```
CC=c99
CFLAGS=-Wall -pedantic
LD=${CC}
LDFLAGS=
EXE=test_carre test_cube ihm
all: ${EXE}
test_carre: test_carre.o carre.o
        ${LD} ${LDFLAGS} $^ -o $@
test_cube: test_cube.o carre.o cube.o
        ${LD} ${LDFLAGS} $^ -o $@
ihm: ihm.o carre.o cube.o
        ${LD} ${LDFLAGS} $^ -o $@
carre.o: carre.c carre.h
        ${CC} ${CFLAGS} -c $<
cube.o: cube.c carre.h cube.h
        ${CC} ${CFLAGS} -c $<
ihm.o: ihm.c carre.h cube.h
        ${CC} ${CFLAGS} -c $<
test_carre.o: test_carre.c carre.h
        ${CC} ${CFLAGS} -c $<
test_cube.o: test_cube.c carre.h cube.h
        ${CC} ${CFLAGS} -c $<
clean:
        ${RM} *.o ${EXE} a.out
```

1. Utilisation :
   - make : réalise le premier but
   - make xxx : réalise le but xxx
2. plusieurs règles de la forme :

   ```
   but: dependances
           actions
   ```

   - but : le fichier à mettre à jour
   - dépendances : les fichiers dont il dépend
   - actions : commandes qui mettent à jour le but
   - Attention : une tabubation avant les actions !
3. CC, CFLAGS, etc sont des macros (variables)
   - ${CC} permet d'accéder à la valeur d'une variable
   - $@ : le but
   - $^ : toutes les dépendances
   - $< : la première dépendance
4. all : premier but (convention)
5. clean : pour nettoyer

Makefile avec règles implicites

```
CC=c99
CFLAGS=-Wall -pedantic
LD=${CC}
LDFLAGS=

EXE=test_carre test_cube ihm

all: ${EXE}

test_carre: carre.o
test_cube: carre.o cube.o
ihm: carre.o cube.o

carre.o: carre.h
cube.o: carre.h cube.h
ihm.o: carre.h cube.h
test_carre.o: carre.h
test_cube.o: carre.h cube.h

%.o: %.c
	${CC} ${CFLAGS} -c $<

%: %.o
	${LD} ${LDFLAGS} $^ -o $@

clean:
	${RM} *.o ${EXE} a.out
```

1. Beaucoup de redondances dans le Makefile précédent :
   - pour produire un .o à partir d'un .c
   - pour produire l'exécutable
2. Solution : les **règles implicites**

   ```
   %.o: %.c
       actions
   ```
3. Conséquences :
   - on supprime l'action pour les règles sur .o
   - on supprime des dépendance le fichier en .c (inutile)
   - idem pour les exécutables
4. Avantage : suppression des actions redondantes
5. Remarque : make définit des règles implicites pour la plupart des langages...

Makefile final

```
CC=c99
CFLAGS=-Wall -pedantic
LD=${CC}
LDFLAGS=

EXE=test_carre test_cube ihm

all: ${EXE}

test_carre: carre.o
test_cube: carre.o cube.o
ihm: carre.o cube.o

depend:
	makedepend -Y. *.c

clean:
	${RM} *.o ${EXE} a.out

.PHONY: clean all depend
# DO NOT DELETE

carre.o: carre.h
cube.o: cube.h carre.h
ihm.o: carre.h cube.h
test_carre.o: carre.h
test_cube.o: cube.h
```

1. Danger : Il faut avoir les bonnes dépendances !
   - Mauvaises dépendances $\Longrightarrow$ mauvaise mises à jour
2. Remarque : on peut les déduire des `#include`
3. Conséquence :
   - écrire un outil qui les calculent
   - c'est makedepend.
4. Pour se souvenir comment l'utiliser, ajouter un but « depend » qui met à jour les dépendances.
5. `.PHONY` : pseudo but pour lister les buts qui ne correspondent pas à des fichiers.