Cours de programmation fonctionnelle et logique

11 décembre 2002

Table des matières

1 Présentation du cours **3**

2 Présentation de `ocaml` **4**
2.1 Le langage . . . 4
2.2 Exécuter des programmes `ocaml` . . . 4
2.3 Les définitions . . . 4
2.3.1 Définitions globales . . . 4
2.3.2 Définitions locales . . . 5
2.4 Les fonctions . . . 5
2.4.1 Définition globale de fonctions . . . 5
2.4.2 Application de fonctions . . . 6
2.4.3 Définition locale de fonctions . . . 6
2.4.4 Fonctions à plusieurs arguments . . . 6
2.4.5 Fonctions anonymes . . . 6
2.4.6 Conventions syntaxiques . . . 7
2.5 Les tests et l'alternative . . . 7
2.6 Les programmes . . . 7
2.7 Programmation impérative . . . 7
2.7.1 Séquencement . . . 8
2.7.2 Les références . . . 8
2.7.3 Les boucles . . . 8
2.7.4 Les tableaux . . . 9
2.7.5 Caractères et chaînes de caractères . . . 9

3 Listes et filtrage **10**
3.1 Définition des listes . . . 10
3.1.1 Différences entre les listes et les tableaux . . . 10
3.1.2 Syntaxe . . . 10
3.1.3 Deux constructeurs . . . 10
3.1.4 Représentation graphique des listes . . . 10
3.2 Filtrage . . . 10
3.2.1 Filtrage par fonction anonyme . . . 11
3.2.2 Filtrage explicite . . . 12
3.2.3 Filtrage sur plusieurs arguments . . . 12
3.3 Filtrage sur les listes . . . 13
3.4 Synonyme dans les filtres . . . 13
3.5 Retour sur les listes . . . 13

4 Programmation fonctionnelle *versus* impérative **14**
4.1 Principes des deux paradigmes . . . 14
4.1.1 Principe de la programmation impérative . . . 14
4.1.2 Principe de la programmation fonctionnelle (pure) . . . 14
4.1.3 Exemple : la factorielle . . . 14
4.2 Propriétés des langages fonctionnels . . . 15
4.3 Notion de variable et de définition . . . 16
4.3.1 En mathématiques . . . 16
4.3.2 En informatique . . . 16

5 Récursivité **17**
5.1 Fonctions récursives simples . . . 17
5.1.1 Notion de récursivité . . . 17
5.1.2 Portée statique et définitions récursives . . . 17
5.1.3 Ordre des appels récursifs . . . 18
5.1.4 Traçage des appels . . . 19
5.2 Exemple : les palindromes . . . 19
5.3 Les tours de Hanoï . . . 20
5.3.1 Le problème . . . 20
5.3.2 Programme . . . 20
5.3.3 Complexité . . . 22
5.4 Récursivité et boucles . . . 22
5.5 Validité d'une définition récursive . . . 23

6 Typage et polymorphisme **24**
6.1 Notion de polymorphisme . . . 24
6.1.1 Polymorphisme dans les expressions de type . . . 24
6.1.2 Exemples simples . . . 25
6.2 Synthèse du type le plus général . . . 25
6.3 Algèbre des types de `ocaml` . . . 26
6.4 Curryfication . . . 27
6.5 Algorithme de typage . . . 28
6.5.1 Unicité du type . . . 28
6.5.2 Premier exemple . . . 29
6.5.3 Deuxième exemple . . . 29
6.5.4 Typage avec contraintes . . . 30
6.5.5 Algorithme général . . . 30

7 Listes et fonctionnelles **31**
7.1 Fonctionnelles simples sur les listes . . . 31
7.1.1 Application successive d'une action à tous les éléments d'une liste . . . 31
7.1.2 Application en parallèle d'une action à tous les éléments d'une liste . . . 32
7.2 Fonctionnelles complexes sur les listes . . . 32
7.2.1 Notion d'abstraction de schéma de programme . . . 32
7.2.2 Autre schéma récursif : l'accumulation . . . 33
7.2.3 Accumuler avec les éléments d'une liste . . . 33
7.2.4 Accumuler encore... . . . 34

8 Types somme et types produit **35**
8.1 Type énuméré . . . 35
8.2 Type somme . . . 35
8.3 Type produit . . . 35

9 Les exceptions **36**
9.1 Erreurs et rattrapages d'erreurs . . . 36
9.2 Valeurs exceptionnelles . . . 37
9.3 Définition d'exceptions . . . 38
9.4 Les exceptions comme moyen de calcul . . . 38

10 Éléments de logique **41**
10.1 Calcul propositionnel . . . 41
10.1.1 Définition . . . 41
10.1.2 Règles de dominance entre connecteurs . . . 41
10.1.3 Tables de valeurs de vérité . . . 41
10.1.4 Analyse de vérité d'une formule . . . 41
10.2 Méthode de résolution . . . 42
10.2.1 Vocabulaire . . . 42
10.2.2 Règle de résolution . . . 42
10.2.3 Preuve par résolution . . . 43
10.2.4 Exemple : arbre de résolution/réfutation . . . 43
10.2.5 Application à la programmation logique . . . 43
10.2.6 Complétude . . . 44

11 Pro(grammation) log(ique) **44**
11.1 Langage utilisé . . . 44
11.2 Expression de faits . . . 44
11.3 Questions . . . 45
11.4 Conjonction de questions . . . 46
11.5 Introduction de règles de déduction . . . 47
11.6 Principe d'effacement . . . 48
11.7 Problème classique : représentation et manipulation des listes . . . 48

12 L'unification ... ou pourquoi 1+1 n'est pas égal à 2 **50**
12.1 Retour sur le principe d'effacement . . . 50
12.1.1 Calcul de la longueur d'une liste . . . 50
12.1.2 $1+1 \neq 2$ . . . 50
12.2 Représentation des prédicats par des arbres . . . 51
12.3 Unification . . . 52

13 Contrôle d'exécution et coupure **54**
13.1 Motivation de la coupure . . . 54
13.1.1 Premier exemple : appartenance à une liste . . . 54
13.1.2 Deuxième exemple : fonctions booléennes . . . 54
13.2 La coupure . . . 55
13.3 Vue théorique . . . 56
13.4 Les dangers de la coupure . . . 56
13.5 La négation . . . 57
13.6 Exemple : résolution des trinômes du second degré . . . 58

1 Présentation du cours

Ce cours se décompose en deux parties, chacune correspondant à un paradigme de programmation. La première s'intérresse à la programmation fonctionnelle et est illustrée avec le langage `caml`. La deuxième partie est une initiation à la programmation logique et est illustrée au moyen de Prolog. Ce cours a pour but de vous montrer que l'on peut programmer autrement, qu'il n'existe pas que la programmation impérative...

Les documents relatifs au cours peuvent normalement être retrouvés à l'url : `http://icps.u-strasbg.fr/~vivien/Enseignement` (sujets de TD, TP, corrigés, etc.).

2 Présentation de `ocaml`

2.1 Le langage

Le langage qui servira de base à la partie fonctionnelle de ce cours est `ocaml` (Objective Caml).
Documentation et distribution : `http://caml.inria.fr/`.

2.2 Exécuter des programmes `ocaml`

Deux méthodes :
- compilation : classique.
- système interactif : l'interpréteur
  - pour lancer l'interpréteur, taper `ocaml` ; le signe d'invite « # » apparaît ;

```
[vivien@gauvain Caml]$ ocaml
        Objective Caml version 3.00

#
```

  - entrer l'expression à évaluer ;
  - clore avec la marque de fin de phrase « ;; ».
  - exemple :

```
# 2+2;;
- : int = 4
```

Lecture : le résultat de l'évaluation de l'expression est une valeur ; cette valeur est de type entier ; elle est égale à « 4 ».
L'expression `ocaml` est juste « `2+2 ; ;` ». Je n'indique le « # » que parce que je veux aussi présenter la réponse de l'interpréteur.
À noter : il y a déduction (inférence) automatique du type.
Pour sortir de l'interpréteur :

```
# #quit;;
[vivien@gauvain Caml]$
```

2.3 Les définitions

2.3.1 Définitions globales

```
# let s = 1+2+3;;
val s : int = 6
```

Lecture : le résultat de l'évaluation de l'expression est la définition d'un nouvel objet, appelé « s » ; cet objet est de type entier ; sa valeur est égale à « 6 ».
La définition est **permanente** :
- Version compilée : l'objet « s » est défini pour tout le reste du programme.
- Version interprétée : l'objet « s » est défini tant que l'interpréteur n'est pas tué.

Par conséquent, une fois l'objet défini de manière globale, il peut être évalué n'importe quand, et à tout moment on peut demander sa valeur à l'interpréteur.

```
# s;;
- : int = 6

# let s2 = s * s;;
val s2 : int = 36
```

On ne spécifie pas forcément de type, mais on peut le faire :

```
# let (t : int) = 1+2+3;;
val t : int = 6
```

Les **types de base** sont :

- int ;
- float ;
- bool (« true » ou « false ») ;
- char ;
- string.

Dans la pratique on ne spécifiera jamais de type.

2.3.2 Définitions locales

Il s'agit de définitions **temporaires** et non plus **permanentes**.

```
# let u = 20 in u*4;;
- : int = 80
```

Ici, l'objet « u » n'est défini qu'au sein de l'expression « u*4 ;; », et n'est pas défini en dehors. Par conséquent si on interroge l'interpréteur, il répondra qu'il ne connaît pas l'objet en question.

```
# u;;
# u;;
Unbound value u
```

Dans une définition locale, on peut réutiliser un identificateur d'un objet défini globalement.

```
# s;;
- : int = 6

# let s = 20 in s*4;;
- : int = 80
```

La valeur de l'objet global n'est bien évidemment pas modifiée :

```
# s;;
- : int = 6
```

La définition locale du nom est indépendante du type actuel du nom (si celui-ci est déjà défini globalement).

```
# let s = "Bon" and s2 = "jour" in s^s2;;
- : string = "Bonjour"
```

Le mot clé « `and` » permet des définitions multiples et simultanées.

2.4 Les fonctions

2.4.1 Définition globale de fonctions

On veut définir une fonction `successeur` qui à l'entier `x` associe l'entier `x+1`.

```
# let successeur (x) = x + 1;;
val successeur : int -> int = <fun>

# let successeur x = x + 1;;
val successeur : int -> int = <fun>
```

Lecture :

- Le résultat de l'évaluation de l'expression est la définition d'un nouvel objet, appelé `successeur`.
- Cet objet est de type `int -> int`. Autrement dit, c'est une fonction qui prend en entrée un entier et qui renvoie un entier. Cette écriture est à rapprocher de l'écriture mathématique : `successeur` : $\mathbb{N} \to \mathbb{N}$
- C'est une fonction et l'interpréteur, incapable d'afficher sa valeur, indique juste `<fun>`.

Le nom `successeur` a une valeur :

```
# successeur;;
- : int -> int = <fun>
```

La définition d'une fonction n'est pas fondamentalement différente de la définition d'un entier. De même, la valeur d'une fonction n'est pas fondamentalement différente de celle d'un entier... Les fonctions ont le même statut que les entiers, ce sont des « citoyens de première classe ».

2.4.2 Application de fonctions

```
# successeur(2);;
- : int = 3
```

```
# successeur 2;;
- : int = 3
```

```
# (successeur 2);;
- : int = 3
```

Les parenthèses sont optionnelles. En général on ne les met que si elles sont nécessaires.

2.4.3 Définition locale de fonctions

```
# let prédécesseur x = x-1 in
    (prédécesseur 3)*(prédécesseur 4);;
- : int = 6
```

```
# prédécesseur 5;;
# prédécesseur 5;;
Unbound value prédécesseur
# let prédécesseur_carré x =
    let pred x = x-1
    in (pred x)*(pred x);;
val prédécesseur_carré : int -> int = <fun>
```

```
# prédécesseur_carré 8;;
- : int = 49
```

2.4.4 Fonctions à plusieurs arguments

```
# let moyenne a b = (a + b)/2;;
val moyenne : int -> int -> int = <fun>
```

```
# moyenne 5 3;;
- : int = 4
```

```
# (moyenne 5 3);;
- : int = 4
```

```
# moyenne (5 3);;
# moyenne (5 3);;
This expression is not a function, it cannot be applied
```

Quand une parenthèse contient plusieurs éléments, le premier d'entre eux doit obligatoirement être une fonction.

```
# (moyenne (5))(3);;
- : int = 4
```

Fonction sans argument :

```
# let fonction_deux () = 2;;
val fonction_deux : unit -> int = <fun>
```

```
# let valeur_deux      = 2;;
val valeur_deux : int = 2
```

2.4.5 Fonctions anonymes

Ce sont des valeurs fonctionnelles qui sont définies mais pas nommées.

```
# function x -> 2*x+1;;
- : int -> int = <fun>
```

Application :

```
# (function x -> 2*x+1) 2;;
- : int = 5
```

Définition mathématique des fonctions

$$\texttt{successeur} : \mathbb{Z} \to \mathbb{Z}$$
$$x \mapsto x+1$$

Expression `ocaml` totalement équivalente :

```
# let (successeur : int -> int) = function x -> x+1;;
val successeur : int -> int = <fun>
```

Écriture effective :

```
# let successeur = function x -> x+1;;
val successeur : int -> int = <fun>
```

Fonctions à plusieurs arguments

Une fonction anonyme ne prend en entrée qu'**un seul** argument.

```
# let moyenne = function a -> function b-> (a+b)/2;;
val moyenne : int -> int -> int = <fun>
```

2.4.6 Conventions syntaxiques

`let f = function x -> ...`	$\Leftrightarrow$	`let f x = ...`
`f (x)`	$\Leftrightarrow$	`f x`
`(f x) y`	$\Leftrightarrow$	`f x y`
`f (g y)`	$\not\Leftrightarrow$	`f g y`
`function x -> f x`	$\Leftrightarrow$	`f`

2.5 Les tests et l'alternative

« `if ... then ... else ...` »

```
# let valeur_absolue x = if x >= 0 then x else -x;;
val valeur_absolue : int -> int = <fun>
```

Les tests calculent un résultat qui est de type booléen :

```
# 2 < 1;;
- : bool = false
```

```
# (valeur_absolue 3) = (valeur_absolue (-3));;
- : bool = true
```

2.6 Les programmes

Ce sont des fonctions !

Un programme est une fonction qui calcule le résultat souhaité. Une fonction manipule des valeurs typées, une fonction est une valeur typée, les programmes `ocaml` sont donc des valeurs typées !

2.7 Programmation impérative

`ocaml` est loin d'être un langage fonctionnel pur. Il contient en effet les principales structures impératives. Cette partie de `ocaml` est décrite à des fins d'exhaustivité **mais ne doit en aucun cas être utilisée**.

2.7.1 Séquencement

```
# print_string "Bon"; print_string "jour";;
Bonjour- : unit = ()
```

- Opérateur de séquencement : « ; ».
- `Bonjour` : impression.
- Le résultat est de type `unit`.
- L'évaluation de l'expression renvoie la valeur `()`, c'est-à-dire « rien ». C'est le seul objet de type `unit`.

Toutes les fonctions `ocaml` rendent un résultat.

```
# begin
   1+2; 0;
 end;;
# begin
1+2; 0;
end;;
Warning: this expression should have type unit.
- : int = 0
# 1+2; 0;;
# 1+2; 0;;
Warning: this expression should have type unit.
- : int = 0
```

Le résultat du premier calcul a été jeté. Le résultat de l'évaluation d'une séquence est le résultat de l'évaluation du dernier terme de cette séquence. Le couple `begin-end` sert à délimiter la séquence. Ici il est superflu.

2.7.2 Les références

Il s'agit de cases mémoires **modifiables** : on peut les lire et les réécrire.

Création :

```
# let compteur = ref 0;;
val compteur : int ref = contents = 0
```

Lecture :

```
# !compteur;;
- : int = 0
```

Écriture :

```
# compteur := 2;;
- : unit = ()

# !compteur;;
- : int = 2
```

Les références se manipulent comme les autres objets :

```
# let incrémente c = c:= !c + 1;;
val incrémente : int ref -> unit = <fun>

# incrémente compteur; !compteur;;
- : int = 3
```

2.7.3 Les boucles

Boucle *tant que* :

```
while expression-condition
do
  expression-actions
done;;
```

Boucle *pour* :

```
for identificateur = expression-début to/downto expression-fin
do
  expression-actions
done;;
```

```
# let imprime_chiffres () =
    for i=0 to 9 do
       print_int i
    done;
    print_newline ();;
val imprime_chiffres : unit -> unit = <fun>
```

```
# imprime_chiffres();;
0123456789
- : unit = ()
```

2.7.4 Les tableaux

Création (définition) :

```
# let p = [|3;2;1|];;
val p : int array = [|3; 2; 1|]
```

```
# let q = Array.make 4 2;;
val q : int array = [|2; 2; 2; 2|]
```

Taille :

```
# Array.length q;;
- : int = 4
```

Lecture (le premier élément d'un tableau est toujours l'élément d'indice « 0 ») :

```
# p.(0);;
- : int = 3
```

Écriture :

```
# q.(0) <- 1;;
- : unit = ()
```

```
# q;;
- : int array = [|1; 2; 2; 2|]
```

2.7.5 Caractères et chaînes de caractères

```
# let mot = "Ocaml";;
val mot : string = "Ocaml"
```

```
# mot.[0];;
- : char = 'O'
```

```
# mot.[0] <- 'o';;
- : unit = ()
```

```
# mot;;
- : string = "ocaml"
```

```
# String.length mot;;
- : int = 5
```

3 Listes et filtrage

3.1 Définition des listes

3.1.1 Différences entre les listes et les tableaux

- on n'accède pas directement à un élément d'une liste, il faut **obligatoirement** la parcourir séquentiellement, de sa tête vers sa queue ;
- les listes peuvent grossir dynamiquement quand la taille d'un tableau est fixée une fois pour toutes ;
- un élément ajouté à une liste est ajouté en tête de cette liste ;
- on ne peut pas changer la valeur d'un élément d'une liste.

Remarque : tous les éléments d'une liste doivent être du même type.

3.1.2 Syntaxe

- la liste est délimitée par des crochets « [ ] » ;
- ses éléments sont séparés par des « ; ».

Liste composée des éléments 1, 2 et 3, dans l'ordre :

```
# [1;2;3];;
- : int list = [1; 2; 3]
```

3.1.3 Deux constructeurs

- « `[]` » appelé `nil` (néant) et qui désigne la liste vide.
- « `::` » appelé `cons`, pour constructeur de liste ; c'est l'opérateur qui rajoute un élément en tête d'une liste.

```
# [];;
- : 'a list = []
```

(la liste vide est *polymorphe*.)

```
# 3::[];;
- : int list = [3]
```

```
# 1::2::3::[];;
- : int list = [1; 2; 3]
```

```
# 0::[1;2;3;];;
- : int list = [0; 1; 2; 3]
```

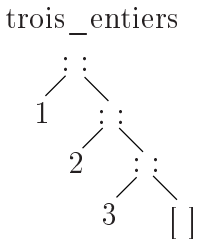
3.1.4 Représentation graphique des listes

```
# [e1;e2;e3;e4];;
```

est une abréviation pour

```
# e1::e2::e3::e4::[];;
```

```
# let trois_entiers = [1;2;3];;
val trois_entiers : int list = [1; 2; 3]
```



3.2 Filtrage

Le filtrage est une **définition par cas**.

3.2.1 Filtrage par fonction anonyme

Suite de Fibonacci : `Fib`$(0) = 1$; `Fib`$(1) = 1$; `Fib`$(n+2) =$ `Fib`$(n) +$ `Fib`$(n+1)$.

```
# let rec fib n =
   if n=0
   then 1
   else if n=1
        then 1
        else (fib (n-2)) + (fib (n-1));;
val fib : int -> int = <fun>
# let rec fib = function
     0 -> 1
   | 1 -> 1
   | n -> (fib (n-2)) + (fib (n-1));;
val fib : int -> int = <fun>
```

Barre verticale « | » signifie « *ou* ». La flèche « `->` » indique l'expression à évaluer.

Argument anonyme :

```
# let égal_un = function
    1 -> true
  | n -> false;;
val égal_un : int -> bool = <fun>

# let égal_un = function
    1 -> true
  | _ -> false;;
val égal_un : int -> bool = <fun>
```

Le symbole « _ » désigne *tous les autres cas*.

Les filtrages doivent toujours être exhaustifs (et non redondants...).

```
# let égal_un = function
    1 -> true;;
# let égal_un = function
      1 -> true;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
0
val égal_un : int -> bool = <fun>

# égal_un 2;;
Exception: Match_failure ("", 14, 35).

# let égal_un = function
    _ -> false
  | 1 -> true;;
# let égal_un = function
_ -> false
| 1 -> true;;
Warning: this match case is unused.
val égal_un : int -> bool = <fun>
# let rec fib = function
     0 -> 1
   | 1 -> 1
   | n+2 -> (fib n) + (fib (n+1));;
# let rec fib = function
0 -> 1
| 1 -> 1
```

```
| n+2 -> (fib n) + (fib (n+1));;
Syntax error
```

L'argument du filtrage doit **obligatoirement** être soit une valeur, soit une variable.

```
# let égaux a = function
    a -> true
  | _ -> false;;
# let égaux a = function
a -> true
| _ -> false;;
Warning: this match case is unused.
val égaux : 'a -> 'b -> bool = <fun>

# égaux 1 2;;
- : bool = true
```

3.2.2 Filtrage explicite

Calcul $(a/b) + c$: on veut vérifier que le deuxième argument (le diviseur) est non nul.

```
# let diventiere a = function
    0 -> (function c -> failwith "Division par zéro")
  | b -> (function c -> (a / b) + c);;
val diventiere : int -> int -> int -> int = <fun>
# let diventiere a b c =
   match b with
    0 -> failwith "Division par zéro"
  | _ -> (a / b) + c;;
val diventiere : int -> int -> int -> int = <fun>
```

`failwith` signale un cas d'erreur. Une fonction `ocaml` doit toujours renvoyer une valeur, comme ici le calcul ne peut pas être effectué, on renvoie une **valeur exceptionnelle** signalant le problème.

3.2.3 Filtrage sur plusieurs arguments

Le filtrage ne peut se faire que sur **un seul argument** à la fois. Solution : l'utilisation de n-uplets.

```
# let pseudo_carré a b =
   match a with
     0 -> 0
   | _ -> match b with
            0 -> 0
          | _ -> a*a+b*b;;
val pseudo_carré : int -> int -> int = <fun>
# let pseudo_carré a b =
   match (a,b) with
     (0,_) -> 0
   | (_,0) -> 0
   |   _   -> a*a+b*b;;
val pseudo_carré : int -> int -> int = <fun>
```

$$f(a,b) = \begin{cases} 0 & \text{si } a = 0 \\ 0 & \text{si } b = 0 \\ a^2 + b^2 & \text{sinon} \end{cases}$$

Les parenthèses sont ici superflues :

```
# let pseudo_carré a b =
   match a,b with
     0,_ -> 0
   | _,0 -> 0
   |  _  -> a*a+b*b;;
val pseudo_carré : int -> int -> int = <fun>
```

3.3 Filtrage sur les listes

Fonction qui teste si une liste est vide.

```
# let nulle = function
    [] -> true
  | _ -> false;;
val nulle : 'a list -> bool = <fun>
```

```
# (nulle [], nulle [1]);;
- : bool * bool = true, false
```

Nommage de la tête et du reste de la liste.

```
# let tete = function
   t::r -> t
 | _   -> failwith "tete";;
val tete : 'a list -> 'a = <fun>
```

Se lit : si la liste argument est non vide, on appelle `t` sa tête et `r` son reste.

```
# tete [1;2;3;4];;
- : int = 1
# let reste = function
   _::r -> r
 | _   -> failwith "reste";;
val reste : 'a list -> 'a list = <fun>
```

```
# reste [1;2;3;4];;
- : int list = [2; 3; 4]
```

Pour décomposer un type structuré, comme les listes, on filtre en utilisant les constructeurs du type, ici « [] » et « :: ».

```
# let bizarre = function
   []   -> 0
 | t::_ -> t;;
val bizarre : int list -> int = <fun>
```

3.4 Synonyme dans les filtres

Ici, la paire (a, b) représentera le nombre $a * 10^b$.

```
# let valeur_absolue = function
   (a,b) -> if a < 0 then (-a,b) else (a,b);;
val valeur_absolue : int * 'a -> int * 'a = <fun>
```

Dans le cas où a est positif, on aimerait pouvoir renvoyer directement la paire reçue en argument, ce que permet l'introduction d'un synonyme par « `as` » :

```
# let valeur_absolue = function
   (a,b) as paire -> if a > 0 then (-a,b) else paire;;
val valeur_absolue : int * 'a -> int * 'a = <fun>
```

3.5 Retour sur les listes

Les manipulations de listes ne modifient pas les listes existantes...

```
# let trois_entiers = [1;2;3];;
val trois_entiers : int list = [1; 2; 3]
```

```
# let reste = function []->failwith "liste vide !" | _::r ->r;;
val reste : 'a list -> 'a list = <fun>
```

```
# 4::(reste(reste (0::trois_entiers)));;
- : int list = [4; 2; 3]
```

```
# trois_entiers;;
- : int list = [1; 2; 3]
```

4 Programmation fonctionnelle *versus* impérative

4.1 Principes des deux paradigmes

4.1.1 Principe de la programmation impérative

Modifications successives de l'état de la machine par des commandes, des effets.
À chaque étape de l'écriture d'un algorithme « impératif », il faut :
- déterminer quelles valeurs sont nécessaires au calcul ;
- associer des emplacements mémoires à ces valeurs ;
- décrire les transformations à appliquer à la mémoire.

In fine
- l'état final **contient** le résultat de l'algorithme.

4.1.2 Principe de la programmation fonctionnelle (pure)

- Évaluation d'expressions (par application de fonctions).
- Le résultat d'une évaluation n'est pas (explicitement) écrit en mémoire : il est argument d'une autre expression ou fonction.
- Un programme fonctionnel spécifie **ce qui** doit être calculé, et non pas **comment** le calculer.
- La mémoire est exclusivement géré par l'interpréteur ou le compilateur, et non par le programmeur.

4.1.3 Exemple : la factorielle

Définition mathématique

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * f(n-1) & \text{sinon} \end{cases}$$

Version impérative

```
# let fact_imp n =
   let a =ref(1) in
     for i=2 to n
       do a := i * !a done;
     !a;;
val fact_imp : int -> int = <fun>
```

1. On réserve un emplacement mémoire, que l'on appelle `a`.
2. On stocke la valeur « 1 » dans la mémoire `a`.
3. À l'étape `i`, on lit la valeur stockée dans `a`, on la multiplie par `i`, et on stocke le résultat dans `a`.
4. On lit la valeur stcokée dans `a`.

Version fonctionnelle

```
# let rec fact_rec = function 0 -> 1 | n -> n * (fact_rec (n-1));;
val fact_rec : int -> int = <fun>
```

Version impérative récursive (en caricaturant...)

```
# let rec fact_imp n =
   let a =ref(0)
   in match n with
        0 -> a := 1
      | _ -> let b = ref(fact_imp (n-1))
             in a := n * !b;
   print_int !a;;
# let rec fact_imp n =
```

```
let a =ref(0)
in match n with
         0 -> a := 1
       | _ -> let b = ref(fact_imp (n-1))
              in a := n * !b;
    print_int !a;;
This expression has type unit but is here used with type int
```

4.2 Propriétés des langages fonctionnels

1. **Inférence de type automatique** :
   - garantie la correction du programme (typage fort) ;
   - pas de typage obligatoire par le programmeur mais inférence du type le plus général possible et **polymorphisme**. Le polymorphisme est la faculté d'accepter des arguments de types différents.

     ```
     # let fonction_un x = 1;;
     val fonction_un : 'a -> int = <fun>
     ```

     Maximum sur une liste :

     ```
     # let rec max_sur_liste = function
         [] -> failwith "liste vide"
       | [a] -> a
       | t::r -> max t (max_sur_liste r);;
     val max_sur_liste : 'a list -> 'a = <fun>

     # max_sur_liste [4;7;3];;
     - : int = 7

     # max_sur_liste ['a'; 'c'; 'z'];;
     - : char = 'z'
     ```

2. Les **fonctions** sont des **individus de première classe** : pas de différence fondamentale entre un entier et une fonction, les deux objets se manipulent de la même manière. Les fonctions peuvent être : membre de structures de données ; passées comme argument à une fonction ; retournées comme résultat d'une fonction.

   Dans les deux derniers cas, on parle de **fonctions d'ordre supérieur** ou de **fonctionnelles**.

   ```
   # let identite x = x;;
   val identite : 'a -> 'a = <fun>

   # (identite identite) 3;;
   - : int = 3
   ```

   Minimum sur une liste :

   ```
   # let rec min_sur_liste = function
       [] -> failwith "liste vide"
     | [a] -> a
     | t::r -> min t (min_sur_liste r);;
   val min_sur_liste : 'a list -> 'a = <fun>

   # min_sur_liste [4;7;3];;
   - : int = 3
   ```

   Fonctionnelle générique :

   ```
   # let rec opt_sur_liste f = function
       [] -> failwith "liste vide"
     | [a] -> a
     | t::r -> f t (opt_sur_liste f r);;
   val opt_sur_liste : ('a -> 'a -> 'a) -> 'a list -> 'a = <fun>

   # opt_sur_liste max [4;7;3];;
   ```

```
- : int = 7
```

```
#opt_sur_liste min [4;7;3];;
- : int = 3
```

```
#opt_sur_liste (function x -> function y -> x+y) [4;7;3];;
- : int = 14
```

3. **Récursivité**.
4. Définition par **filtrage**. *switch* élaboré.
5. **Évaluation paresseuse** (parfois) : un argument n'est évalué que quand c'est nécessaire. Un argument peut donc n'être jamais évalué.
6. **Transparence référentielle** si pas d'effets de bords, si une variable n'est définie qu'une seule fois (pas d'affectations). Autrement dit, « des égaux peuvent être remplacés pas des égaux ».

   `(f a) + (f a)` $\Leftrightarrow$ `let x = f a in x + x`

   Avec la fonction suivante, il n' y a pas équivalence :

   ```
   #let f x = print_int x; x;;
   val f : int -> int = <fun>
   ```

   ```
   #(f 2) + (f 2);;
   22- : int = 4
   ```

   ```
   #let x = f 2 in x + x;;
   2- : int = 4
   ```

   Il est alors possible de raisonner sur les programmes, de les analyser, de prouver des propriétés et de les garantir.

4.3 Notion de variable et de définition

4.3.1 En mathématiques

Variable = inconnue. On obtient la valeur de la variable par la résolution d'une équation. La définition est donc *implicite*.

$$\left\{\begin{array}{rcl} x+4 & = & 4x-2 \\ 2x-3 & = & 1 \end{array}\right. \quad \Rightarrow \quad x=2$$

$$\left\{\begin{array}{rcl} x+2 & = & 4x-1 \\ 2x-3 & = & 1 \end{array}\right. \quad \Leftrightarrow \quad \left\{\begin{array}{rcl} x & = & (2+1)/3 \\ x & = & (1+3)/2 \end{array}\right. \quad \Rightarrow \quad \text{pas de solution.}$$

En informatique, les « équations » ci-dessous ont un sens.

$$\begin{array}{rcl} x & = & (2+1)/3; \\ x & = & (1+3)/2 \end{array}$$

4.3.2 En informatique

Variable = emplacement mémoire.
Définition = affectation = écriture d'une valeur en mémoire.

Dans un langage fonctionnel, une variable n'est « affectée » qu'une seule fois. Une nouvelle définition ne change pas la valeur de la variable, mais crée une nouvelle variable de même nom.

Premier exemple :

```
#let f x = x + 1;;
```

```
#let g = f;;
```

```
#let f x = x + 2;;
```

```
# g 0;;
- : int = 1
```

Deuxième exemple :

```
# type entier = Integer of int;;
type entier = Integer of int

# let a = Integer 3;;
val a : entier = Integer 3

# type entier = Integer of int;;
type entier = Integer of int

# let f = function Integer x -> x;;
val f : entier -> int = <fun>

# f a;;
# f a;;
This expression has type entier but is here used with type entier
```

On a donc bien cohabitation de deux définitions *différentes* du type `entier`, et de deux définitions *différentes* du constructeur `Integer`...

5 Récursivité

De l'art d'écrire des programmes qui résolvent des problèmes que l'on ne sait pas résoudre soi-même !

5.1 Fonctions récursives simples

5.1.1 Notion de récursivité

Définition : une définition récursive est une définition dans laquelle intervient ce que l'on veut définir. C'est donc *une définition qui se mord la queue.*

```
# let rec z = 1::z;;
```

Risque que la définition soit dénuée de sens :

```
# let rec f x = f x;;
val f : 'a -> 'b = <fun>
```

Principe et intérêt : on réduit la résolution d'un problème complexe à la résolution d'un ou plusieurs problèmes, de même nature, mais de taille inférieure (ou plus simples). Prérequis : on sait résoudre un certain nombre de cas simples (ou cas d'arrêt de la récursion.).

Difficultés :

- détection des cas où la récursivité entraîne le non-sens ;
- vérification de la validité de la définition.

5.1.2 Portée statique et définitions récursives

En mathématiques, comme en `caml`, on ne peut parler que de ce qui a été préalablement défini. Donc, *pour pouvoir utiliser un identificateur, il faut qu'il ait été préalablement défini.*

Portée statique : on peut trouver la définition indépendamment de l'exécution. (*portée dynamique* : la valeur dépend de la façon dont le calcul se déroule.)

Pour connaître la valeur associée à un identificateur `x`, il suffit d'examiner le texte du programme qui précède `x` pour trouver la liaison qui définit `x`.

Exemples

L'expression :

```
# let x = 1 in
    let x = x+1 in
      x+3;;
- : int = 5
```

est équivalente à :

```
# let y = 1 in
    let x = y+1 in
      x+3;;
- : int = 5
```

L'expression :

```
# let x = 1 in
    let x = x+1 and y = x + 2 in
      x+y;;
- : int = 5
```

est équivalente à :

```
# let z = 1 in
    let x = z+1 and y = z + 2 in
      x+y;;
- : int = 5
```

Nécessité d'une définition préalable

Un nom fait toujours référence à une définition préalable.

```
let f = ... f ...
```

Le nom `f` en membre droit ne fait pas référence au nom `f` en membre gauche puisque celui-ci n'est pas encore défini (nous sommes à l'intérieur de sa définition).

```
# let fact = function
    0 -> 1
  | n -> n*(fact (n-1));;
# let fact = function
0 -> 1
| n -> n*(fact (n-1));;
Unbound value fact
```

D'où la nécessité du mot-clef `rec` :

```
# let rec fact = function
    0 -> 1
  | n -> n*(fact (n-1));;
val fact : int -> int = <fun>
```

5.1.3 Ordre des appels récursifs

Fonction qui énumère les entiers par ordre décroissant, de `n` jusqu'à 1 :

- si `n` vaut 0, ne rien faire ;
- sinon, on imprime `n` puis on énumère les nombres par ordre décroissant de `n` jusqu'à 1.

```
# let rec compte_à_rebours = function
    0 -> ()
  | n -> begin print_int n; print_string " "; compte_à_rebours (n-1) end;;
val compte_à_rebours : int -> unit = <fun>
```

```
# compte_à_rebours 10;;
10 9 8 7 6 5 4 3 2 1 - : unit = ()
```

Interversion de l'affichage et de l'appel récursif :

```
# let rec compte = function
    0 -> ()
  | n -> begin compte (n-1); print_int n; print_string " " end;;
val compte : int -> unit = <fun>
```

```
# compte 10;;
1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

L'impression a maintenant lieu au retour des appels récursifs.

5.1.4 Traçage des appels

```
# let successeur x = x+1;;
val successeur : int -> int = <fun>

# #trace successeur;;
successeur is now traced.
   # successeur 2;;
   successeur <-- 2       Appel de fonction et argument.
   successeur --> 3       Retour de fonction et valeur retournée.
   - : int = 3
```

Retour sur les fonctions d'énumérations

```
# #trace compte;;
compte is now traced.

# #trace compte_à_rebours;;
compte_à_rebours is now traced.

# compte_à_rebours 3;;
compte_à_rebours <-- 3
3 compte_à_rebours <-- 2
2 compte_à_rebours <-- 1
1 compte_à_rebours <-- 0
compte_à_rebours --> ()
compte_à_rebours --> ()
compte_à_rebours --> ()
compte_à_rebours --> ()
- : unit = ()

# compte 3;;
compte <-- 3
compte <-- 2
compte <-- 1
compte <-- 0
compte --> ()
1 compte --> ()
2 compte --> ()
3 compte --> ()
- : unit = ()
```

Remarquez quand les impressions ont lieu : avant l'appel récursif pour `compte_à_rebours`, et après pour `compte`.

5.2 Exemple : les palindromes

Un palindrome est un mot qui se lit aussi bien à l'endroit qu'à l'envers (cf. figure 1).

- La chaîne vide est un palindrome.
- Un mot réduit à un seul caractère est un palindrome.
- Un mot d'au moins deux caractères est un palindrome si :
  - ses premier et dernier caractères sont identiques ;
  - le mot situé entre ces caractères est un palindrome.

```
# let rec palindrome s =
   match String.length s with
     0 -> true
   | 1 -> true
   | n -> if s.[0]=s.[n-1]
```

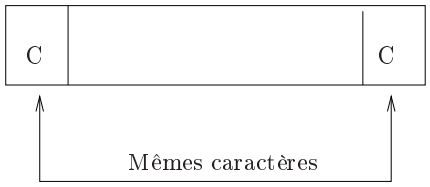


FIG. 1 – Structure d'un palindrome

```
          then palindrome (String.sub s 1 (n-2))
          else false;;
val palindrome : string -> bool = <fun>
```

Opérateurs booléens : `et` = &, `ou` = `or`.

```
# let rec palindrome s =
   match String.length s with
     0 -> true
   | 1 -> true
   | n -> (s.[0]=s.[n-1]) & palindrome(String.sub s 1 (n-2));;
val palindrome : string -> bool = <fun>
```

Version sans construction de sous-chaînes

On utilise deux indices, `i` et `j`, désignant respectivement le début et la fin de la sous-chaîne traitée. Initialement, `i` vaut 0 et `j` la longueur de la chaîne moins 1. À chaque étape on compare les caractères d'indice `i` et `j` :

- s'ils sont égaux on continue ;
- s'ils sont différents on arrête ;
- si `i`≥`j`, on a atteint ou dépassé le milieu et on arrête.

```
# let rec palin s i j =
   (i >= j) or ((s.[i]=s.[j])&(palin s (i+1) (j-1)));;
val palin : string -> int -> int -> bool = <fun>

# let palindrome s = palin s 0 (String.length s - 1);;
val palindrome : string -> bool = <fun>
```

Et avec une fonction locale

```
# let palindrome s =
   let rec palin i j =
     (i >= j) or ((s.[i]=s.[j])&(palin (i+1) (j-1)))
   in palin 0 (String.length s - 1);;
val palindrome : string -> bool = <fun>
```

5.3 Les tours de Hanoï

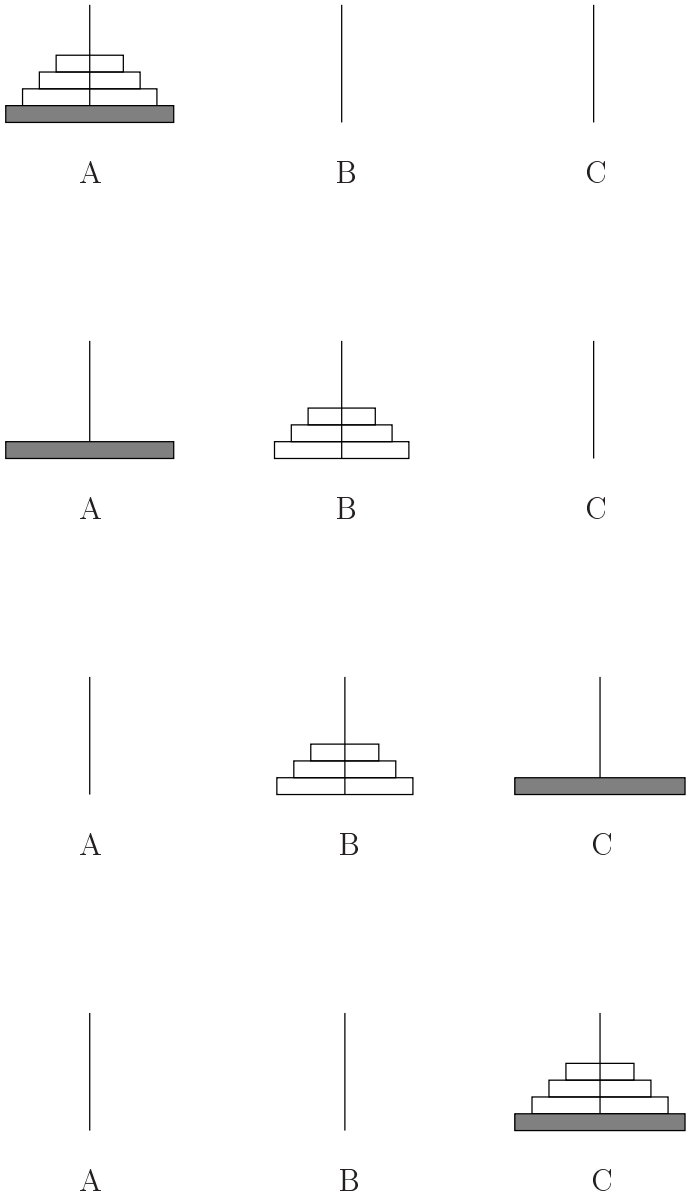
5.3.1 Le problème

Le jeu est constitué d'une plaquette de bois où sont plantées trois tiges. Sur ces tiges sont enfilés des disques de diamètres tous différents. Les seules règles du jeu, sont que l'on ne peut déplacer qu'un seul disque à la fois, et qu'il est interdit de poser un disque sur un disque plus petit.

Au début tous les disques sont sur la tige de gauche, et à la fin sur celle de droite.

5.3.2 Programme

```
# let mouvement de vers =
```



Principe : nous supposons que nous savons résoudre le problème pour (n-1) disques.
Algorithme : on déplace les (n-1) plus petits disques de `A` vers `B`, puis on déplace le plus gros disque de `A` vers `C`, puis on déplace les (n-1) plus petits disques de `B` vers `C`.
Validité : il n'y a pas de viols possibles puisque le plus gros disque est toujours en « bas » d'une tige et que l'hypothèse nous assure que nous savons déplacer le « bloc » de (n-1) disques en respectant les règles.

```
   print_string("Déplace un disque de la tige "^de^" à la tige "^vers);
   print_newline();;
val mouvement : string -> string -> unit = <fun>

# let rec hanoi départ milieu arrivée = function
   0 -> ()
 | n -> hanoi départ arrivée milieu (n-1);
        mouvement départ arrivée;
        hanoi milieu départ arrivée (n-1);;
val hanoi : string -> string -> string -> int -> unit = <fun>

# print_newline(); hanoi "A" "B" "C" 3;;

Déplace un disque de la tige A à la tige C
Déplace un disque de la tige A à la tige B
Déplace un disque de la tige C à la tige B
Déplace un disque de la tige A à la tige C
Déplace un disque de la tige B à la tige A
Déplace un disque de la tige B à la tige C
Déplace un disque de la tige A à la tige C
- : unit = ()
```

Il ne faut pas chercher à comprendre *comment* ça marche, mais *pourquoi* ça marche...

5.3.3 Complexité

```
# let rec compte_hanoi départ milieu arrivée = function
   0 -> 0
 | n -> compte_hanoi départ arrivée milieu (n-1) +
        1 +
        compte_hanoi milieu départ arrivée (n-1);;
val compte_hanoi : 'a -> 'a -> 'a -> int -> int = <fun>

# let rec compte_hanoi = function
   0 -> 0
 | n -> compte_hanoi (n-1) + 1 + compte_hanoi (n-1);;
val compte_hanoi : int -> int = <fun>

# let rec compte =
  function   0 -> 0
           | n -> 2*(compte (n-1))+1;;
val compte : int -> int = <fun>

# let rec compte_rapide n = 2.0 ** n -. 1.0;;
val compte_rapide : float -> float = <fun>

# compte_rapide 64.0;;
- : float = 1.84467440737e+19
```

Pour arriver au même résultat, `compte_hanoi` aurait dû faire $2^{64} - 1$ additions, ce qui lui aurait pris plus de 5845 ans à raison de 100 10^6 additions par secondes.

Moralité : il ne suffit pas que les programmes soient corrects...

5.4 Récursivité et boucles

Fonction qui épelle un mot :

```
# let rec épelle_aux s i =
   if i < String.length s then
     begin
```

```
     print_char(s.[i]);
     print_char ' ';
     épelle_aux s (i+1)
     end;;
val épelle_aux : string -> int -> unit = <fun>
```

```
# let épelle s = épelle_aux s 0;;
val épelle : string -> unit = <fun>
```

```
# let épelle s =
  let rec épelle_aux i =
   if i < String.length s then
     begin
     print_char(s.[i]);
     print_char ' ';
     épelle_aux (i+1)
     end
 in épelle_aux 0;;
val épelle : string -> unit = <fun>
```

Il s'agit en fait d'une boucle déguisée :

```
# let épelle s =
   for i=0 to String.length s - 1 do
     print_char(s.[i]); print_char ' ';
   done;;
val épelle : string -> unit = <fun>
```

Mais tous les programmes ne peuvent pas être *facilement* dérécursivés (tours de Hanoï...).

5.5 Validité d'une définition récursive

Il faut s'assurer que la fonction est complètement définie, c'est-à-dire, qu'elle est définie sur tout son domaine d'applications.

Définition par décomposition : il faut vérifier que toutes les formes d'objets du type décomposé sont décrits par la fonction (à vérifier même si la définition n'est pas récursive).

```
# let f = function true -> 1;;
# let f = function true -> 1;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
false
val f : bool -> int = <fun>
```

Le filtrage est exhaustif mais la fonction peut ne pas être définie pour toute valeur.

```
# let rec f p =
   if p >= 0
     then match p with
       0 -> 1
     | n -> (f (n-2)) + 1
     else failwith "probleme";;
val f : int -> int = <fun>
```

Il faut s'assurer qu'il n'y a pas d'applications indéfinies (fonction définie uniquement si `a=1` ou si `b=1`).

```
# let rec produit (a,b) =
   if a = 1 then b else produit (b,a);;
val produit : int * int -> int = <fun>
```

Critères d'exhaustivité

- Existence d'un ordre strict tel que les valeurs successives des arguments invoqués par la définition soient strictement décroissantes à partir d'un certain rang.
```
#let rec nombre n =
   if n < 10 then 1 else 1 + nombre (n/10);;
val nombre : int -> int = <fun>
```
(nombre de chiffres de la représentation décimale d'un entier.)
- La suite des valeurs successives des arguments atteint toujours une valeur pour laquelle la fonction est explicitement définie. La fonction ci-dessous teste si `a` est un diviseur de **b**.
```
#let rec diviseur (a,b) =
 if a < 0
   then failwith "diviseur"
   else
     if a >=b
       then a = b
       else diviseur (a,b-a);;
val diviseur : int * int -> bool = <fun>
```
La suite des valeurs `b`, `b-a`, `b-2*a`, etc. est strictement décroissante, car `a` est strictement positif, et on fini toujours pas aboutir à un couple d'arguments `(a,b)` tel que `b-a` est négatif, cas défini explicitement.

Il reste des cas incertains, tel la suite de Syracuse :
```
#let rec syracuse = function
   0 -> 1
 | 1 -> 1
 | n -> if (n mod 2=0) then syracuse (n/2)
                       else syracuse (3*n+1);;
val syracuse : int -> int = <fun>
```
Non démontré : elle vaut 1 sur $\mathbb{N}$.

6 Typage et polymorphisme

6.1 Notion de polymorphisme

polymorphe : étymologie = qui peut prendre plusieurs formes. Il s'agit d'objets ou de programmes qui peuvent servir *sans modifications* dans des contextes très différents.

Une fonction de tri qui s'applique à un seul type d'objets (e.g. les entiers) est *monomorphe*.

Une fonction de tri qui s'applique à tous les types d'objets est *polymorphe*. Cette fonction de tri pourra être employée sur des objets de types différents : pour trier un ensemble d'entiers ou un ensemble de chaînes de caractères. Mais, *a priori*, pas pour trier un ensemble contenant simultanément des entiers et des chaînes de caractères.

Des *objets* peuvent être polymorphes, comme la liste vide qui peut être vue comme une liste d'entiers, de chaînes de caractères, etc. :
```
#[];;
- : 'a list = []
```

6.1.1 Polymorphisme dans les expressions de type

L'introduction du polymorphisme nécessite une notion de type qui permette de remplacer plusieurs types différents. `ocaml` utilise des *paramètres de type* : `'a` (paramètre de type nommé « a »).

Le polymorphisme de `caml` est *paramétrique*. Il fonctionne en « tout ou rien » : un paramètre remplace n'importe quel type et non pas un certain nombre de types. Il n'y a pas de possibilités en `caml` de définir un type qui s'appliquerait uniquement à des entiers et à des chaînes de caractères. Il n'existe pas de type du style `(int or string) -> ...`

Un programme `caml` s'applique donc :

– soit à tous les types possibles (`'a -> ...`) ;
– soit à un seul et unique type (`int -> ...`).

6.1.2 Exemples simples

Fonction monomorphe (à cause de l'addition).

```
# let successeur x = x+1;;
val successeur : int -> int = <fun>
```

Fonction polymorphe (comme elle n'utilise pas son argument on peut l'appliquer à n'importe quoi).

```
# let fonction_un x = 1;;
val fonction_un : 'a -> int = <fun>

# fonction_un 2;;
- : int = 1

# fonction_un "oui";;
- : int = 1
```

Une fonction polymorphe peut utiliser son argument.

```
# let identite x = x;;
val identite : 'a -> 'a = <fun>
```

Ici, *le type du résultat est exactement celui de l'argument.* Le paramètre de type `'a` remplace n'importe quel type, mais il remplace le même type partout dans l'expression de type. On peut, par exemple, utiliser la fonction `identite`, avec le type `string -> string`.

```
# identite "truc";;
- : string = "truc"
```

Spécialisation : remplacement d'un paramètre de type par un type quelconque. Ici, `'a` est spécialisé en `string`. On n'est pas obligé de spécialiser un paramètre de type par un type de base. On peut, par exemple, spécialiser `'a` avec `int -> int`.

```
# successeur;;
- : int -> int = <fun>

# identite successeur;;
- : int -> int = <fun>

# identite successeur 3;;
- : int = 4

# let reste = function
   t::r -> r
 |  []  -> failwith "liste vide";;
val reste : 'a list -> 'a list = <fun>

# [];;
- : 'a list = []

# 1::[];;
- : int list = [1]

# reste (1::[]);;
- : int list = []
```

Ici, la spécialisation a eu lieu avec le type `int list`.

6.2 Synthèse du type le plus général

Inférence de type : le compilateur `caml` donne un type à chaque expression saisie, de manière totalement automatique. Pour ce faire, il utilise les types des *valeurs de base* et les *règles de typage* pour les constructions du langage.

Le type inféré contient le plus petit ensemble de contraintes nécessaires au bon déroulement de l'évaluation des expressions du programme. Plus petit ensemble de contraintes implique **type le plus général possible** pour chaque expression.

```
# let successeur x = x+1;;
val successeur : int -> int = <fun>
```

Son argument doit être entier car on lui ajoute l'entier 1 au moyen de l'addition des entiers de symbole « + ».

```
# let identite x = x;;
val identite : 'a -> 'a = <fun>
```

Il n'y a pas ici de contraintes sur l'argument de la fonction (par contre la valeur retournée est de même type que l'argument).

Le polymorphisme est une *conséquence de l'absence de contraintes* sur un argument ou une valeur.

```
# let double (f : int -> int) x = 2 * (f x);;
val double : (int -> int) -> int -> int = <fun>
```

`x` doit être de type entier car on a imposé la contrainte que l'argument de `f` devait être de type entier : `f : int -> int`.

```
# let double f x = 2 * (f x);;
val double : ('a -> int) -> 'a -> int = <fun>
```

La fonction est maintenant polymorphe. Le contrôleur de type a découvert que `f` avait pour seule contrainte de rendre un entier en résultat (car son résultat est argument de la multiplication des entiers), mais qu'il n'était nullement nécessaire que `f` prenne un entier en argument.

Comme le polymorphisme découle de l'absence de contraintes, un paramètre de type peut être remplacé sans risque d'erreurs par n'importe quel type, y compris par un type lui-même polymorphe.

```
# identite identite;;
- : '_a -> '_a = <fun>
```

Le résultat de cette évaluation est la fonction `identite`.

6.3 Algèbre des types de `ocaml`

L'algèbre des types est l'ensemble des types utilisés par le système `ocaml`. Il existe trois catégories de types :

1. les types de bases : `int`, `string`, etc.
2. les types composites : `int -> int`, `int list`, etc.
3. les paramètres de type : `'a`.

Les types composites sont obtenus au moyen de constructeurs de types tels que « `->` ». Soient deux types $t1$ et $t2$. $t1$ « `->` » $t2$ est le type des fonctions ayant un argument de type $t1$ et rendant un résultat de type $t2$.

`->` est un constructeur de type binaire infixe. `list` et `array` sont des constructeurs de types unaires et postfixes. En `ocaml`, tous les constructeurs de types unaires sont postfixes. Les constructeurs de types sans arguments (`int`) sont dits *constructeurs de types constants*.

Les paires

Un nouveau constructeur de type binaire infixe : « `*` ». Soient deux types $t1$ et $t2$. $t1$ `*` $t2$ est le type produit cartésien des types $t1$ et $t2$, c'est-à-dire le type des couples dont le premier élément est de type $t1$ et le deuxième de type $t2$. La notation est identique à celle des couples en mathématiques.

```
# (1,"e");;
- : int * string = 1, "e"
```

Les paires peuvent être utilisées comme argument ou résultat d'une fonction.

```
# let addition_soustraction (x,y) = (x+y, x-y);;
val addition_soustraction : int * int -> int * int = <fun>

# addition_soustraction (4,3);;
- : int * int = 7, 1
```

Les paires permettent d'écrire des fonctions qui rendent plusieurs résultats. Les paires se généralisent aux n-uplets.

```
# (1, 2.3, "t",[5;6]);;
- : int * float * string * int list = 1, 2.3, "t", [5; 6]
```

6.4 Curryfication

Une fonction qui prend une paire comme argument ne prend qu'un seul argument et non pas deux.

```
# let addition (x,y) = x+y;;
val addition : int * int -> int = <fun>
```

La fonction addition ne prend donc qu'un seul argument.

```
# let add x y = x+y;;
val add : int -> int -> int = <fun>
```

La fonction `add` prend deux arguments et est donc différente de la fonction `addition`.

Une fonction qui reçoit ses argument un par un, comme `add`, est dite **curryfiée**. Une fonction qui reçoit tous ses arguments à la fois sous la forme d'un n-uplet est dite **non curryfiée**. Le terme *curryfié* vient du nom du logicien Haskell Curry.

Application partielle

La différence entre les fonctions `add` et `addition` provient de la manière de les appliquer : il est légal d'appliquer `add` à un seul argument.

```
# add 1;;
- : int -> int = <fun>

# let suivant = add 1;;
val suivant : int -> int = <fun>

# suivant 3;;
- : int = 4
```

Au contraire, la fonction `addition` doit recevoir tous ses arguments en une seule fois.

Application partielle : c'est la capacité qu'a une fonction curryfiée de ne recevoir qu'un certain nombre de ses arguments.

```
# let sum = function x -> function y -> x+y;;
val sum : int -> int -> int = <fun>
```

Les fonctions `add` et `sum` sont équivalentes.

Curryfication et « type flèche »

Une fonction curryfiée est un cas particulier de fonctionnelle puisqu'elle permet de créer d'autres fonctions, en fixant un certain nombre de ses arguments. Cette propriété est inscrite dans le type de la fonction curryfiée :

```
# add;;
- : int -> int -> int = <fun>
```

Le constructeur `->` associe à droite. Donc `add` est de type : `int -> (int -> int)`. Autrement dit, `add` est une fonction qui prend comme argument un entier et qui renvoie une objet de type `int -> int`, c'est-à-dire une fonction qui prend en entrée un entier et qui renvoie un entier.

```
# let ajoute_deux = add 2;;
val ajoute_deux : int -> int = <fun>

# add 2 3;;
- : int = 5

# (add 2) 3;;
- : int = 5

# ajoute_deux 3;;
- : int = 5
```

Autre approche

Pour tout entier `n` on peut écrire une fonction d'ajout :

```
# let n = 732;;
val n : int = 732

# let ajoute_n x = n + x;;
val ajoute_n : int -> int = <fun>
```

`add` est une fonction générique qui généralise toutes ces fonctions : c'est une fonctionnelle.

Pour une fonction curryfiée très générale, spécialiser un de ses arguments peut permettre d'obtenir une fonction intéressante. Une fonctionnelle de recherche d'extremum acceptant la fonction de comparaison en premier paramètre peut être spécialisée par application partielle en fonction de recherche de maximum ou de minimum.

```
# let rec extr comp = function
   t::u::r -> if (comp t u) then extr comp (t::r)
                            else extr comp (u::r)
  |  [t]   -> t
  |   _    -> failwith "Liste vide !";;
val extr : ('a -> 'a -> bool) -> 'a list -> 'a = <fun>

# let maximum = extr (function x -> function y -> x > y);;
val maximum : '_a list -> '_a = <fun>

# maximum [3;5;2];;
- : int = 5

# let minimum = extr (function x -> function y -> x < y);;
val minimum : '_a list -> '_a = <fun>

# minimum [3;5;2];;
- : int = 2
```

Fonctionnelles de curryfication et de dé-curryfication

Fonctionnelle qui associe à une fonction non curryfiée acceptant une paire en argument, la version curryfiée correspondante.

```
# let curry f = function x -> function y -> f (x,y);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

Fonctionnelle inverse :

```
# let uncurry f = function (x,y) -> f x y;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

6.5 Algorithme de typage

6.5.1 Unicité du type

Dans une fonction, toutes les occurences d'un même argument sont du même type.

```
# let f x = 3;;
val f : 'a -> int = <fun>

# f 4;;
- : int = 3

# f "a";;
- : int = 3

# (f 4) + (f "a");;
- : int = 6

# let g = f in (g 4) + (g "a");;
```

```
- : int = 6
```

```
# let g = (function x -> (f x) + 1 ) in (g 4) + (g "a");;
- : int = 8
```

```
# let g h = (h 4) + (h "a");;
# let g h = (h 4) + (h "a");;
This expression has type string but is here used with type int
```

6.5.2 Premier exemple

```
# let double f x = 2*(f x);;
val double : ('a -> int) -> 'a -> int = <fun>
```

```
f : a
x : b
2*(f x) : c
double : a -> b -> c
```

`*` : opération primitive, donc de type connu `int -> int -> int`

```
2 : int = vrai
(f x) : int
c : int
(f x) : int
f : d -> e, a = d -> e
x : d, b = d
e = int
a -> b -> c
(d -> e) -> b -> c
(d -> e) -> d -> c
(d -> int) -> d -> int
```

d'où le type.

6.5.3 Deuxième exemple

```
let machin n f x = f (n f x);;
n : a
f : b
x : c
f (n f x) : d
machin : a -> b -> c -> d
```

`f` est appliquée à un argument, donc c'est une fonction.

```
f : e -> d, b = e -> d
(n f x) : e
```

`n` est appliquée (premier élément d'un groupe entre parenthèses), donc c'est une fonction.

```
n : b -> c -> e = a
machin : a -> b -> c -> d
machin : (b -> c -> e) -> b -> c -> d
machin : ((e -> d) -> c -> e) -> (e -> d) -> c -> d
```

```
# let machin n f x = f (n f x);;
val machin : (('a -> 'b) -> 'c -> 'a) -> ('a -> 'b) -> 'c -> 'b = <fun>
```

6.5.4 Typage avec contraintes

```
# let machin (n: ('a->'a)->('a->'a)) f x = f (n f x);;
val machin : (('a -> 'a) -> 'a -> 'a) -> ('a -> 'a) -> 'a -> 'a = <fun>

n : (a->a)->(a->a)
f : b
x : c
f (n f x) : d
machin : ((a->a)->(a->a)) -> b -> c -> d
f : e -> d = b
n f x : e
n : b -> c -> e = (a->a)->(a->a)
b -> c -> e = b -> (c -> e)
b = a -> a or b = e -> d, e = a, d = a
c -> e = a -> a, c = a, e = a
machin : ((a->a)->(a->a)) -> (a->a) -> a -> a
```

6.5.5 Algorithme général

Algorithme

Initialisation

- $\Gamma = \emptyset$: ensemble des déclarations de types de variables ;
- $E = \emptyset$: système d'équations entre les types ;
- $T =$ `t : a` : ensemble des expressions à typer.

Étape courante

On prend une expression `u : a` de T et on l'enlève de T. Cas possibles :

- `u = function x -> v`.
  - $\Gamma = \Gamma \cup \{$`x : b`$\}$, où `b` est une nouvelle lettre.
  - On rajoute `v : c` dans T, où `c` est une nouvelle lettre.
  - $E = E \cup \{$`a = b-> c`$\}$.
- `u = x` où `x` est une variable. Alors il existe un type `b` tel que `x : b` $\in \Gamma$. $E = E \cup \{$`a = b`$\}$.
- `u = (v w)`. Soit `b` une nouvelle lettre. $T = T \cup \{$`v : b -> a`$\} \cup \{$`w : b`$\}$.
- `u = if v then w else z`. $T = T \cup \{$`v : bool`$\} \cup \{$`w : a`$\} \cup \{$`z : a`$\}$.

[1]

Exemple

```
# let double = function f -> function x -> 2*(f x);;
val double : ('a -> int) -> 'a -> int = <fun>
```

Initialisation

- $\Gamma = \{$`2 : int`, `* : int -> int -> int`$\}$;
- $E = \emptyset$;
- $T =$ `function f -> function x -> 2*(f x) : a`.

Étapes successives

1. `u = function f -> function x -> 2*(f x) : a`, $T = \emptyset$.
   - $\Gamma = \Gamma \cup \{$`f : b`$\}$.

[1] `match ... with` sur une liste ?

- $T = \{$`function x -> 2*(f x) : c`$\}$.
- $E = E \cup \{$`a = b -> c`$\}$.

2. `u = function x -> 2*(f x) : c`, $T = \emptyset$.
   - $\Gamma = \Gamma \cup \{$`x : d`$\}$.
   - $T = \{$`2*(f x) : e`$\}$.
   - $E = E \cup \{$`c = d -> e`$\}$.
3. `u = 2*(f x) : e`, $T = \emptyset$.
   - $T = \{$`2 : g, (f x) : h, * : g -> h -> e`$\}$.
4. `u = 2 : g`, $T = \{$`(f x) : h, * : g -> h -> e`$\}$.
   - `2 : int` $\in \Gamma \Rightarrow E = E \cup \{$`g = int`$\}$.
5. `u = * : g -> h -> e`, $T = \{$`(f x) : h`$\}$.
   - `* : int -> int -> int` $\in \Gamma \Rightarrow E = E \cup \{$`g = int, h = int, e = int`$\}$.
6. `u = (f x) : h`, $T = \emptyset$.
   - $T = \{$`f : i -> h, x : i`$\}$.
7. `u = f : i -> h`, $T = \{$`x : i`$\}$.
   - $E = E \cup \{$`b = i -> h`$\}$.
8. `u = x : i`, $T = \emptyset$.
   - $E = E \cup \{$`d = i`$\}$.

Résultat : $E = \{$`a = b -> c, c = d -> e, g = int, g = int, h = int, e = int, b = i -> h, d = i`$\}$

```
a = b -> c
a = (i -> h) -> c
a = (i -> int) -> (d -> e)
a = (i -> int) -> (i -> int)
double : ('a -> int) -> 'a -> int
```

7 Listes et fonctionnelles

7.1 Fonctionnelles simples sur les listes

7.1.1 Application successive d'une action à tous les éléments d'une liste

Soit une fonction `f` et une liste `l`. La fonctionnelle `List.iter` applique `f` tour à tour à tous les éléments de `l`. Autrement dit, évaluer `List.iter f [e1 ;e2 ;e3 ; ... ;en]` est équivalent à exécuter la séquence `begin f e1 ; f e2 ; f e3 ; ... ; f en ; () end`.

```
# List.iter print_int [1;2;3];;
123- : unit = ()
```

Nous pouvons écrire nous-mêmes une fonction équivalente.

```
# List.iter;;
- : ('a -> unit) -> 'a list -> unit = <fun>

# let rec iter f = function
      []  -> ()
   | t::r -> f t; iter f r;;
val iter : ('a -> 'b) -> 'a list -> unit = <fun>

# let rec iter f = function
      []  -> ()
   |  [a] -> f a
   | t::r -> f t; iter f r;;
val iter : ('a -> unit) -> 'a list -> unit = <fun>
```

7.1.2 Application en parallèle d'une action à tous les éléments d'une liste

La fonctionnelle `List.map` retourne la liste obtenue par application d'une fonction `f` à chaque élément d'une liste `l`. Autrement dit, évaluer `List.map f [e1;e2;e3;...;en]` équivaut à évaluer `[f e1;f e2;f e3;...;f en]`. `List.map` distribue la fonction `f` sur les éléments d'une liste.

```
# List.map (function x -> x+1) [1;2;3];;
- : int list = [2; 3; 4]
```

Nous pouvons écrire nous-mêmes une fonction équivalente.

```
# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>

# let rec map f = function
      []  -> []
  | t::r -> f t :: map f r;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

7.2 Fonctionnelles complexes sur les listes

7.2.1 Notion d'abstraction de schéma de programme

Somme des éléments d'une liste

```
# let rec somme = function
      []  -> 0
  | t::r -> t + (somme r);;
val somme : int list -> int = <fun>
```

Produit des éléments d'une liste

```
# let rec produit = function
      []  -> 1
  | t::r -> t * (produit r);;
val produit : int list -> int = <fun>
```

Concaténation de toutes les chaînes d'une liste

```
# let rec implose = function
      [] -> ""
  | t::r -> t^(implose r);;
val implose : string list -> string = <fun>
```

Concaténation des éléments d'une liste de listes

```
# let rec concatene_listes = function
     []   -> []
  | t::r -> t @ concatene_listes r;;
val concatene_listes : 'a list list -> 'a list = <fun>
```

On voit se dessiner un schéma de programme

```
let rec f = function
    []  -> élément_neutre
 | t::r -> t opération_infixe (f r);;
```

Pour obtenir une fonctionnelle générale qui implémente ce schéma de programme, il nous suffit « d'abstraire » l'opération infixe et l'élément neutre.

Ici, toutes les opérations sont infixes : il faut les remplacer par des opérations préfixes équivalentes. Par exemple on remplace « + » par `add`

```
# let add = function x -> function y -> x + y;;
val add : int -> int -> int = <fun>
# let rec itérateur f b = function
      []  -> b
  | t::r -> f t (itérateur f b r);;
val itérateur : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

```
# let somme_i = itérateur add 0;;
val somme_i : int list -> int = <fun>
```
Longueur d'une liste
```
# let rec taille = function
      [] -> 0
   | t::r -> 1 + (taille r);;
val taille : 'a list -> int = <fun>

# let longueur = itérateur (function x -> function y -> 1+y) 0;;
val longueur : '_a list -> int = <fun>
```
Concaténation de deux listes (éléments pris dans l'ordre inverse)
```
# let rec concat l2 = function
      []  -> l2
   | t::r -> t :: concat r l2;;
val concat : 'a list -> 'a list -> 'a list = <fun>

# let concat_i l2 = itérateur (function x -> function y-> x :: y) l2;;
val concat_i : 'a list -> 'a list -> 'a list = <fun>
```

7.2.2 Autre schéma récursif : l'accumulation

Somme avec accumulateur
```
# let rec somme_accu accu = function
      []  -> accu
   | t::r -> somme_accu (accu + t) r;;
val somme_accu : int -> int list -> int = <fun>

# let somme l = somme_accu 0 l;;
val somme : int list -> int = <fun>

# let somme = somme_accu 0;;
val somme : int list -> int = <fun>
```
Fonctionnelle générale
```
# let rec accumulateur_sur_listes f accu = function
      []  -> accu
   | t::r -> accumulateur_sur_listes f (f t accu) r;;
val accumulateur_sur_listes : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fu
n>

# let somme_accu_i = accumulateur_sur_listes (function x-> function y-> x+y) 0;;
val somme_accu_i : int list -> int = <fun>
```

7.2.3 Accumuler avec les éléments d'une liste

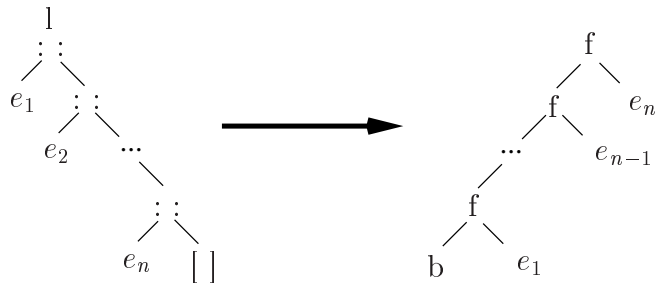
La fonctionnelle `List.fold_left f b l` effectue de multiples compositions de la fonction à deux arguments `f`, en utilisant les éléments de la liste `l` comme seconds arguments de `f`. La valeur de base `b` est utilisée pour le premier argument du premier appel. Le résultat de chaque appel est passé en premier argument de l'appel suivant.

Le schéma d'application de cette fonction est donc le suivant :

`List.fold_left f b [e1 ;e2 ; ... ; en] = f(...(f(f b e1) e2)...)en`

Comme précédemment, nous pouvons écrire notre propre version de cette fonctionnelle.
```
# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

# let rec fold_left f b = function
      []  -> b
   | t::r -> fold_left f (f b t) r;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```



Si f est l'addition on obtient : `List.fold_left f b [e1 ; e2 ; .... en] = b + e1 + e2 ... + en`.

```
# let somme =  List.fold_left (function x -> function y -> x + y) 0;;
val somme : int list -> int = <fun>
```

Concaténation des éléments d'une liste de chaînes de caractères.

```
# let concat_left = List.fold_left (function x -> function y-> x ^ y) "";;
val concat_left : string list -> string = <fun>

# concat_left ["a";"b";"c"];;
- : string = "abc"
```

7.2.4 Accumuler encore...

`List.fold_right f l b` effectue de multiples compositions de la fonction `f` à deux arguments, en utilisant les éléments de `l` comme premier argument. L'élément de base `b` est utilisé comme second argument lors du dernier appel, puis le résultat de chaque appel est utilisé comme second argument de l'appel suivant.

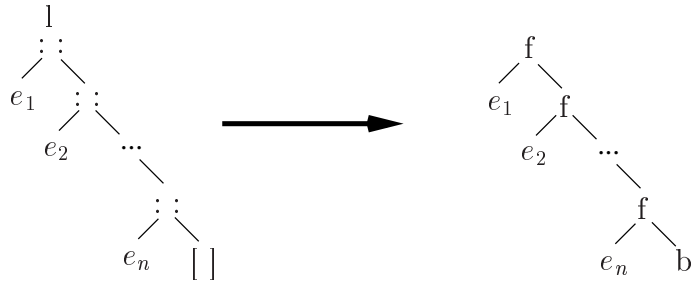
Le schéma d'application de cette fonction est donc le suivant :

`List.fold_right f [e1 ;e2 ; ... ; en] b = f e1 (f e2 (...(f en b) ...))`

Comme précédemment, nous pouvons écrire notre propre version de cette fonctionnelle.

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>

# let rec fold_right f l b = match l with
     []  -> b
  | t::r -> f t (fold_right f r b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```



Concaténation des éléments d'une liste de chaînes de caractères.

```
# let concat_right l = List.fold_right (function x -> function y-> x ^ y) l "";;
val concat_right : string list -> string = <fun>

# concat_right ["a";"b";"c"];;
- : string = "abc"
```

Copie d'une liste (remplaçant `::` par `::` et `[]` par `[]`)

```
# let copie_liste l = List.fold_right (function x-> function y -> x::y) l [];;
val copie_liste : 'a list -> 'a list = <fun>
```

Fonction de concaténation de deux listes

```
# let concatene l1 l2 = List.fold_right (function x-> function y -> x::y) l1 l2;;
val concatene : 'a list -> 'a list -> 'a list = <fun>
```

ou

```
# let concatene = List.fold_right (function x-> function y -> x::y);;
val concatene : '_a list -> '_a list -> '_a list = <fun>
```

8 Types somme et types produit

8.1 Type énuméré

```
# type couleur = Trèfle | Carreau | Coeur | Pique;;
type couleur = Trèfle | Carreau | Coeur | Pique
```

Le type `couleur` est une énumération de quatre valeurs, chacune étant définie par un constructeur de type constant.

```
# let atout = Trèfle;;
val atout : couleur = Trèfle
```

8.2 Type somme

```
# type carte =   As of couleur
               | Roi of couleur
               | Dame of couleur
               | Valet of couleur
               | Petite_carte of int * couleur;;
type carte =
As of couleur
| Roi of couleur
| Dame of couleur
| Valet of couleur
| Petite_carte of int * couleur
```

```
# let carte1 = As atout;;
val carte1 : carte = As Trèfle
```

```
# let carte2 = Petite_carte(7, Pique);;
val carte2 : carte = Petite_carte (7, Pique)
```

Filtrage

Comme pour les listes, le filtrage sur un type somme se fait par décomposition de l'objet suivant les constructeurs possibles.

```
# let valeur_d_une_carte latout = function
     As _               -> 11
   | Roi _              -> 4
   | Dame _             -> 3
   | Valet c            -> if c = latout
                           then 20
                           else 2
   | Petite_carte (10,_) -> 10
   | Petite_carte (9,c)  -> if c = latout
                           then 14
                           else 0
   | Petite_carte _ -> 0;;
val valeur_d_une_carte : couleur -> carte -> int = <fun>
```

8.3 Type produit

```
# type monôme = {coefficient : int; degré : int};;
```

```
type monôme =  coefficient : int; degré : int;

# let m1 = {coefficient = 1; degré = 0};;
val m1 : monôme = coefficient = 1; degré = 0

# m1.coefficient;;
- : int = 1

# m1.degré;;
- : int = 0

# let de_degré_zéro = function
     {degré = 0; coefficient = _ } -> true
   |           _                   -> false;;
val de_degré_zéro : monôme -> bool = <fun>

# let de_degré_zéro = function
     {degré = 0} -> true
   |        _    -> false;;
val de_degré_zéro : monôme -> bool = <fun>

# let abs_monôme = function
   {coefficient = c; degré = d} as m ->
     if c < 0 then {coefficient = -c; degré = d} else m;;
val abs_monôme : monôme -> monôme = <fun>
```

9 Les exceptions

9.1 Erreurs et rattrapages d'erreurs

Toute fonction doit renvoyer une valeur (à moins qu'elle ne boucle indéfiniment).

Problème : il existe des fonctions qui ne peuvent retourner de valeurs sensées pour toutes les valeurs possibles de leurs arguments :

- division par 0 ;
- fonction qui renvoie la tête d'une liste appliquée à la liste vide.

Dans ce cas la fonction doit échouer. Il y a arrêt des calculs et renvoi d'un « signal d'erreur ».

failwith

```
# failwith;;
- : string -> 'a = <fun>

# let tete = function
    x::_ -> x
  |  _   -> failwith "tete";;
val tete : 'a list -> 'a = <fun>

# tete [];;
Exception: Failure "tete".
```

« Uncaught exception » signifie « exception non rattrapée ». On a déclenché une *valeur exceptionnelle*, ce qui a interrompu l'exécution du programme, mais on n'a pas essayé de récupérer cette valeur.

On a donc totalement interrompu l'exécution mais il se pouvait très bien que l'on était capable de continuer le calcul, même en cas d'erreur. Exemple : fonction qui calcule la somme des têtes d'une liste de listes.

```
# let rec somme_têtes_de_listes = function
    [] -> 0
 | t::r -> (somme_têtes_de_listes r) + (tete t);;
val somme_têtes_de_listes : int list list -> int = <fun>
```

```
# somme_têtes_de_listes [[2;3];[];[1;4]];;
Exception: Failure "tete".
```

Ici le calcul s'arrête car une des listes est vide. On pourrait simplement décider que si une des listes est vide elle ne contribue pas à la somme. La solution est de « rattraper » l'exception.

`try ... with`

`try` ... `with` : signifie « essayer ... avec ». `try` expression `with` filtrage sur les valeurs exceptionnelles.

```
# let rec somme_têtes_de_listes = function
    [] -> 0
  | t::r -> (somme_têtes_de_listes r) +
            try (tete t) with Failure "tete" -> 0;;
val somme_têtes_de_listes : int list list -> int = <fun>

# somme_têtes_de_listes [[2;3];[];[1;4]];;
- : int = 3
```

Sémantique : on essaye de calculer l'expression

- Si tout se passe bien, on renvoie le résultat obtenu.
- Sinon, une valeur exceptionnelle est générée.
  - Si cette valeur exceptionnelle rentre dans un des cas du filtrage, on évalue l'expression qui lui est associée par le filtrage.
  - Sinon, on renvoie la valeur exceptionnelle, l'exception se propage. Tout se passe comme si le filtrage se terminait toujours par le filtre « `x -> x` ».

```
# let test l = try tete l with Failure "tete" -> 0;;
val test : int list -> int = <fun>

# test [1];;
- : int = 1

# test [];;
- : int = 0

# let test l = try tete l with Failure "reste" -> 0;;
val test : int list -> int = <fun>

# test [1];;
- : int = 1

# test [];;
Exception: Failure "tete".
```

9.2 Valeurs exceptionnelles

Les « erreurs » sont des valeurs à part entière du langage. On les appelle « valeurs exceptionnelles ». Elles appartiennent au type prédéfini `exn`.

L'échec signalé par la fonction tête est la valeur exceptionnelle `Failure "tete"`.

```
# let échec_de_tête = Failure "tete";;
val échec_de_tête : exn = Failure "tete"

# échec_de_tête;;
- : exn = Failure "tete"
```

`Failure` est en fait un constructeur du type `exn`.

Pour *lever* une valeur exceptionnelle, on utilise la fonction `raise`.

```
# raise;;
- : exn -> 'a = <fun>

# raise échec_de_tête;;
Exception: Failure "tete".
```

La fonction `raise` interrompt immédiatement les calculs en cours pour déclencher la valeur exceptionnelle qu'elle a reçu en argument. `raise` peut intervenir dans n'importe quel contexte et avec n'importe quel type. Comme les calculs ne seront de toutes façons jamais effectués, le contexte peut faire toutes les hypothèses qu'il veut sur la valeur renvoyée par « raise ». Il faut quand même que l'expression soit bien formée.

```
# let f x = x + (raise échec_de_tête);;
val f : int -> int = <fun>

# let g x = (raise échec_de_tête) ^ x;;
val g : string -> string = <fun>

# let h x y = x + (raise échec_de_tête) ^ y;;
# let h x y = x + (raise échec_de_tête) ^ y;;
This expression has type int but is here used with type string
```

9.3 Définition d'exceptions

De nombreuses fonctions `ocaml` déclenchent, quand elles échouent, l'exception construite au moyen de `Failure` avec leur nom en argument, grâce au déclencheur prédéfini `failwith`.

```
# let failwith s = raise (Failure s);;
val failwith : string -> 'a = <fun>
```

Le type `exn` est un type somme dont la définition n'est jamais achevée : on peut toujours lui rajouter des constructeurs de types constants ou rationnels au moyen du déclarateur `exception;;`

```
# exception Stop;;
exception Stop

# exception Erreur_fatale of int;;
exception Erreur_fatale of int

# raise (Erreur_fatale 4);;
Exception: Erreur_fatale 4.
```

9.4 Les exceptions comme moyen de calcul

Les exceptions permettent de transporter un résultat et permettent de signaler un événement attendu.

```
# exception Trouvé;;
exception Trouvé

# let recherche_char_in_liste l c =
  let rec recherche_aux = function
       []  -> false
    | t::r -> if (t=c)
              then raise Trouvé
              else recherche_aux r
 in try recherche_aux l
    with Trouvé -> true;;
val recherche_char_in_liste : 'a list -> 'a -> bool = <fun>

# recherche_char_in_liste ['q';'w';'e';'r';'t';'y'] 'q';;
- : bool = true

# recherche_char_in_liste ['q';'w';'e';'r';'t';'y'] 'a';;
- : bool = false
```

Cette fonction prend en entrée une liste de chaînes de caractères (représentées par des listes de caractères) et renvoie, s'il en existe une, une liste qui contient le caractère recherché.

```
# exception Liste of char list;;
exception Liste of char list

# let recherche_liste n l =
```

```
 try
   List.fold_right
     (function x -> function y -> if recherche_char_in_liste x n
                                  then raise (Liste x)
                                  else y)
     l
     []
 with Liste s -> s;;
val recherche_liste : char -> char list list -> char list = <fu
n>

# recherche_liste 'e' [['a';'b';'c'];['d';'e';'f'];['g';'h';'i']];;
- : char list = ['d'; 'e'; 'f']

# recherche_liste 'z' [['a';'b';'c'];['d';'e';'f'];['g';'h';'i']];;
- : char list = []
```

De l'art de bien rattraper les exceptions

Les lieux de déclenchement des exceptions et de leurs rattrapages doivent être judicieusement choisis afin d'optimiser le comportement du programme. Considérons la fonction suivante calculant le produit des éléments d'une liste :

```
# let rec multiply = function
      []  -> 1
   | t::r -> t * (multiply r);;
val multiply : int list -> int = <fun>
```

L'appel

```
# multiply [1;2;3;4;0;6;7];;
```

effectuera le calcul : $1 \times (2 \times (3 \times (4 \times (0 \times (6 \times (7))))))$. On peut déjà optimiser ce calcul sans recourir aux exceptions :

```
# let rec multiply = function
      []  -> 1
   | 0::_ -> 0
   | t::r -> t * (multiply r);;
val multiply : int list -> int = <fun>
```

Le calcul effectué sera alors : $1 \times (2 \times (3 \times (4 \times (0))))$. Introduisons maintenant une exception :

```
# exception Zéro;;
exception Zéro

# let rec multiply = function
      []  -> 1
   | 0::_ -> raise Zéro
   | t::r -> t * (multiply r);;
val multiply : int list -> int = <fun>
```

Ici il n'y a pas de rattrapage de l'exception et la fonction ne renverra pas de valeur entière si la liste contient un zéro :

```
# multiply [1;2;3;4;0;6;7];;
Exception: Zéro.
```

Rattraper l'exception est donc ici nécessaire, encore faut-il rattraper les exceptions au bon endroit.

```
# let rec multiply = function
      []  -> 1
   | 0::_ -> raise Zéro
   | t::r -> try t * (multiply r) with Zéro -> 0;;
val multiply : int list -> int = <fun>

# multiply [1;2;3;4;0;6;7];;
```

```
- : int = 0
```

Ici la fonction a bien le comportement voulu, par contre, l'exception étant rattrapée dès son lancement, il n'y a pas plus d'optimisation des calculs que dans la version optimisée sans exceptions le calcul effectué étant : $1 \times (2 \times (3 \times (4 \times (0))))$. La fonction suivante améliore ce comportement puisqu'elle n'effectue aucune multiplication quand la liste contient un zéro :

```
#  let rec f = function
      []  -> 1
   | 0::_ -> raise Zéro
   | t::r -> t * (f r);;
val f : int list -> int = <fun>

# let multiply_opt l = try (f l) with Zéro -> 0;;
val multiply_opt : int list -> int = <fun>
```

Pour s'en convaincre, il suffit de comparer la trace de l'exécution de chacune des deux fonctions (raison pour laquelle `f` n'a pas été définie comme une fonction locale à `multiply_opt`) :

```
# #trace  multiply;;
multiply is now traced.

# #trace  multiply_opt;;
multiply_opt is now traced.

# #trace  f;;
f is now traced.

# multiply [1;2;3;4;0;6;7];;
multiply <-- [1; 2; 3; 4; 0; 6; 7]
multiply <-- [2; 3; 4; 0; 6; 7]
multiply <-- [3; 4; 0; 6; 7]
multiply <-- [4; 0; 6; 7]
multiply <-- [0; 6; 7]
multiply raises Zéro
multiply --> 0
multiply --> 0
multiply --> 0
multiply --> 0
- : int = 0

# multiply_opt [1;2;3;4;0;6;7];;
multiply_opt <-- [1; 2; 3; 4; 0; 6; 7]
f <-- [1; 2; 3; 4; 0; 6; 7]
f <-- [2; 3; 4; 0; 6; 7]
f <-- [3; 4; 0; 6; 7]
f <-- [4; 0; 6; 7]
f <-- [0; 6; 7]
f raises Zéro
f raises Zéro
f raises Zéro
f raises Zéro
f raises Zéro
multiply_opt --> 0
- : int = 0
```

Ceux qui ne seraient pas encore convaincus peuvent remplacer l'appel à la multiplication par l'appel à une fonction :

```
# let mul x y = x * y;;
```

et tracer également les appels à `mul`...

10 Éléments de logique

10.1 Calcul propositionnel

10.1.1 Définition

Ici, nous onsidérons des langages logiques ne contenant que :
- les *variables propositionnelles* (notées p, q, r, etc.) : ce sont des variables à valeurs booléennes, c'est-à-dire qui valent soit « vrai », soit « faux » ;
- les *connecteurs* : $\neg$ (non), $\wedge$ (et), $\vee$ (ou), $\rightarrow$ (implique) et $\leftrightarrow$ (équivalent à) ;
- les *parenthèses* : pour éliminer les ambiguïtés.

Exemple : $\neg p \vee \neg(\neg q \rightarrow r)$.

10.1.2 Règles de dominance entre connecteurs

Les connecteurs sont classés du plus prioritaire au moins prioritaire :
- $\neg$
- $\wedge$, $\vee$
- $\rightarrow$, $\leftrightarrow$
- $\exists$, $\forall$

10.1.3 Tables de valeurs de vérité

Nous codons « vrai » par 1 et « faux » par 0. Nous calculons, dans une table de vérité, les valeurs de vérité d'une formule, en fonction des valeurs des variables propositionnelles (cf. figure 2).

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
0	0	1	0	0	1	1
1	0	0	0	1	0	0
0	1	1	0	1	1	0
1	1	0	1	1	1	1

FIG. 2 – Table de vérité des connecteurs.

$p \wedge q$ est vrai si et seulement si p est vrai et q est vrai.
« p vrai implique q vrai » est équivalent à $q \vee (\neg p)$.
Valuation : fonction qui assigne à chaque variable p une valeur $v(p)$ égale à 0 ou 1.

10.1.4 Analyse de vérité d'une formule

Il s'agit simplement de calculer la valeur de vérité de la formule.

$$\neg(p \vee q \rightarrow p) \wedge q$$

$p = 0$		$p = 1$
$\neg(0 \vee q \rightarrow 0) \wedge q$		$\neg(1 \vee q \rightarrow 1) \wedge q$
$\neg(q \rightarrow 0) \wedge q$		$\neg(1 \rightarrow 1) \wedge q$
$q = 0$	$q = 1$	$\neg(1) \wedge q$
$\neg(0 \rightarrow 0) \wedge 0$	$\neg(1 \rightarrow 0) \wedge 1$	$\neg(1) \wedge q$
$\neg(1) \wedge 0$	$\neg(0) \wedge 1$	$0 \wedge q$
$0 \wedge 0$	$1 \wedge 1$	0
0	1	0

Formule toujours fausse : *antilogie*. Exemple : $q \wedge \neg q$.
Formule toujours vraie : *tautologie*. Exemple : $q \vee \neg q$.

10.2 Méthode de résolution

Étant donnée une formule, on recherche une valuation pour laquelle la formule soit vraie.

10.2.1 Vocabulaire

- *Littéral* : variable propositionnelle ou sa négation. Si u est un littéral, $\bar{u} = \neg u$, u et $\bar{u}$ sont dits complémentaires.
- *Clause* : disjonction de littéraux : $p \vee q \vee \neg r$.
- *Forme clausale* : Forme Normale Conjonctive, ou conjonction de clauses (conjonction de disjonctions) : $(p \vee q \vee \neg r) \wedge (p \vee \neg q)$.

10.2.2 Règle de résolution

Soient deux clauses C_1 et C_2. Soit u un littéral tel que $u \in C_1$ et $\bar{u} \in C_2$ Alors :

$$\frac{C_1, C_2}{C_1 \vee C_2 \setminus \{u, \bar{u}\}}$$

où C_1, C_2 signifie $C_1 \wedge C_2$. Autrement dit, nous remplaçons l'expression $C_1 \wedge C_2$ par l'expression $C_1 \vee C_2 \setminus \{u, \bar{u}\}$.

Explications

$C_1 = C_1' \vee u$ et $C_2 = C_2' \vee \bar{u}$.

Alors $C_1 \wedge C_2 \mapsto C_1' \vee C_2'$. $C_1' \vee C_2'$ est la *résolvante* de C_1 et C_2. Toute valuation, tout modèle, qui valide $C_1 \wedge C_2$ valide $C_1' \vee C_2'$. Réciproquement, s'il existe une valuation satisfaisant $C_1 \vee C_2 \setminus \{u, \bar{u}\}$, il en existe une validant $C_1 \wedge C_2$.

Démonstration

1. Montrons tout d'abord l'implication : $v(C_1 \wedge C_2) = 1 \Rightarrow v(C_1' \vee C_2') = 1$.
   Soit v une valuation qui valide $C_1 \wedge C_2$. Si $v(C_1 \wedge C_2) = 1$, alors $v(C_1) = 1$ et $v(C_2) = 1$. u peut prendre deux valeurs, et nous avons donc deux cas à considérer.
   (a) $v(u) = 1$ et $v(\bar{u}) = 0$. Or $v(C_2) = 1$ et $C_2 = C_2' \vee \bar{u}$. Donc, $v(\bar{u}) = 0$ implique $v(C_2') = 1$. Donc $v(C_1' \vee C_2') = 1$.
   (b) $v(u) = 0$ et $v(\bar{u}) = 1$. Or $v(C_1) = 1$ et $C_1 = C_1' \vee u$. Donc, $v(u) = 0$ implique $v(C_1') = 1$. Donc $v(C_1' \vee C_2') = 1$.
2. Nous montrons maintenant l'implication : $v(C_1' \vee C_2') = 1 \Rightarrow \exists v', v'(C_1 \wedge C_2) = 1$.
   Soit v une valuation qui valide $C_1' \vee C_2'$. On veut en déduire une valuation v' qui valide $C_1 \wedge C_2$. Nous avons encore deux cas à considérer, suivant que $v(C_1')$ vaut 1 ou 0.
   (a) $v(C_1') = 1$. On pose alors $v' = v$ avec $v'(\bar{u}) = 1$.
   $v'(C_1) = v'(C_1' \vee u) = v'(C_1') \vee v'(u) = 1 \vee 0 = 1$.
   $v'(C_2) = v'(C_2' \vee \bar{u}) = v'(C_2') \vee 1 = 1$.
   (b) $v(C_1') = 0$, alors $v(C_2') = 1$. On pose alors $v' = v$ avec $v'(u) = 1$.
   $v'(C_1) = v'(C_1' \vee u) = v'(C_1') \vee v'(u) = 0 \vee 1 = 1$.
   $v'(C_2) = v'(C_2' \vee \bar{u}) = v'(C_2') \vee v'(\bar{u}) = 1 \vee 0 = 1$.

10.2.3 Preuve par résolution

Soit Σ un ensemble de clauses. Nous appelons preuve par résolution à partir de Σ une suite de clauses $C_1, ..., C_n$ telle que :

- soit $C_i \in \Sigma$;
- soit C_i est obtenue à partir de deux clauses antérieures et de la règle de résolution.

Le dernier élément de la suite de clauses s'appelle le *but* ou la conclusion. Si le but est la clause vide, l'ensemble de clauses est inconsistant.

10.2.4 Exemple : arbre de résolution/réfutation

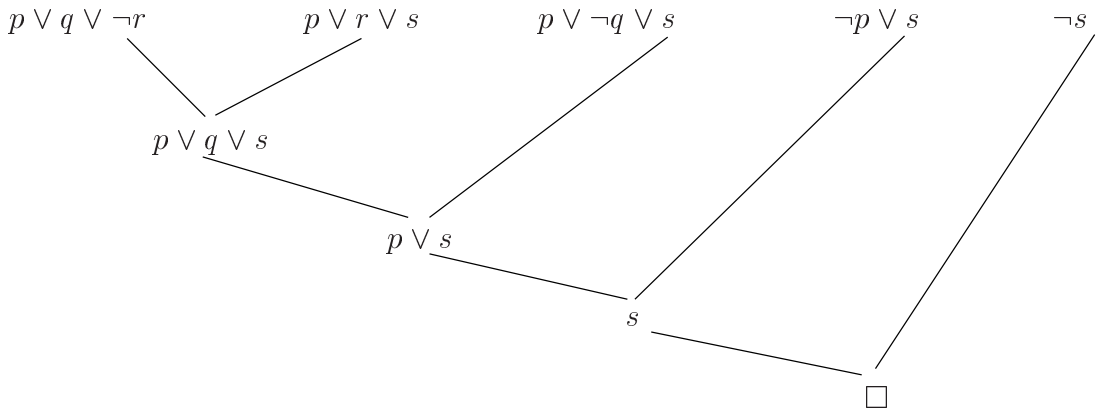


FIG. 3 – Exemple d'arbre de réfutation

$\square$: clause vide ou réfutation.

10.2.5 Application à la programmation logique

Clause de Horn

Une clause de Horn est une clause qui a au plus un littéral positif. Exemple : $\neg p \vee q \vee \neg r$.

Une clause de Horn avec un littéral positif est dite *positive*. Une clause de Horn dont les littéraux sont négatifs est dite *négative*.

Lien entre clauses de Horn et programmation logique

$$\begin{aligned} \neg p \vee \neg q \vee \neg r \vee s \quad &\Leftrightarrow \\ \neg(p \wedge q \wedge r) \vee s \quad &\Leftrightarrow \\ (p \wedge q \wedge r) \rightarrow s \quad & \end{aligned}$$

La dernière équation est une assertion logique typique : si les conditions p, q et r sont vérifiées, alors la propriété s est vraie.

Soit un ensemble de propriétés logiques $C_1, ..., C_n$, C_i étant une clause de Horn. A-t-on le résultat B ?

Question : la formule $C_1 \wedge ... \wedge C_n \rightarrow B$ est-elle vraie ? Autrement dit, la formule $B \vee \neg(C_1 \wedge ... \wedge C_n)$ est-elle vraie, c'est-à-dire, est-elle une tautologie ?

Preuve par réfutation :

nous supposons la formule $\neg(B \vee \neg(C_1 \wedge ... \wedge C_n))$ vraie et nous montrons l'inconsistance. Or $\neg(B \vee \neg(C_1 \wedge ... \wedge C_n)) = \neg B \wedge \neg\neg(C_1 \wedge ... \wedge C_n)) = \neg B \wedge C_1 \wedge ... \wedge C_n$. Notre problème revient donc à montrer l'inconsistance de l'ensemble des clause $\neg B, C_1, ..., C_n$, c'est-à-dire, à montrer qu'il n'existe pas de valuation satisfaisant l'ensemble des clause $\neg B, C_1, ..., C_n$.

Exemple.

On suppose que l'on a les propriétés suivantes : $C_1 : p$, $C_2 : p \to q$ et $C_3 : q \to r$. La question est : « a-t-on r ? » (la formule r est-elle vraie).

Commençons par transformer nos propriétés en clauses de Horn.

$$C_1 \quad p \tag{1}$$
$$C_2 \quad q \vee \neg p \tag{2}$$
$$C_3 \quad r \vee \neg q \tag{3}$$
$$\text{But} \quad r \tag{4}$$

Nous raisonnons par réfutation, c'est-à-dire par l'absurde. Soit $E = p \wedge (q \vee \neg p) \wedge (r \vee \neg q) \wedge \neg r$. Nous sommes donc à la recherche d'une instanciation de p, q et r telle que la formule E soit vraie. Nous voulons en fait montrer qu'une telle instanciation n'existe pas.

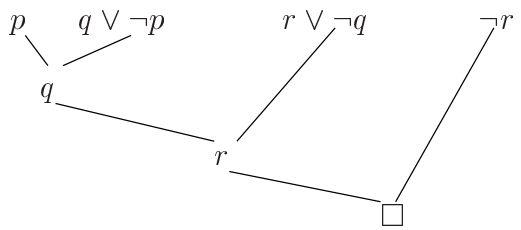


FIG. 4 – Preuve de la formule $(p \wedge (p \to q) \wedge (q \to r)) \to r$.

Nous aboutissons sur la clause vide, il n'existe donc pas d'instanciation de p, q et r qui permette de valider E. Donc la formule E est toujours fausse et la formule $\neg E$ est toujours vraie. Donc r est vraie à chaque fois que les clauses C_1, C_2 et C_3 le sont. Nous avons donc prouvé la formule : $(p \wedge (p \to q) \wedge (q \to r)) \to r$.

10.2.6 Complétude

Si Σ est un ensemble de clauses inconsistant, alors il existe un arbre de réfutation de Σ. On peut donc prouver une formule qui est vraie !

11 Pro(grammation) log(ique)

11.1 Langage utilisé

SWI-Prolog. Documentation : `http://swi.psy.uva.nl/projects/SWI-Prolog/`.

11.2 Expression de faits

Un fait est une « vérité première » :
- Loïc est le père de Gérard.
- Gérard est le père de Maxime.
- Gérard est le père de Lise.

Expression de ces faits en Prolog. On définit un prédicat `père`, c'est-à-dire une fonction à valeurs booléennes. Par défaut, *ce prédicat est faux là où il n'est pas défini.*

```
père(loïc,gérard).
père(gérard,maxime).
père(gérard,lise).
```

Le prédicat `père` ainsi défini n'est donc vrai que pour les couples d'argument `(loïc,gérard)`, `(gérard, maxime)` et `(gérard,lise)`.

Attention : on peut utiliser des lettres accentuées dans les noms d'identificateurs, mais *les noms de constantes doivent toujours commencer par une minuscule.*

Les faits sont écrits dans un programme, appelé par exemple `exemples` (extension « .pl » facultative). On lance l'interpréteur, puis on charge le programme dans l'interpréteur :

```
[vivien@gauvain Sources]$ pl
Welcome to SWI-Prolog (Version 3.4.4)
Copyright (c) 1990-2000 University of Amsterdam.
Copy policy: GPL-2 (see www.gnu.org)

For help, use ?- help(Topic). or ?- apropos(Word).

?- consult(exemples).
% exemples compiled 0.00 sec, 804 bytes

Yes
?- [exemples].
% exemples compiled 0.00 sec, 0 bytes

Yes
?- halt.
[vivien@gauvain Sources]$
```

Après avoir chargé le programme, l'interpréteur nous répond « Yes » pour indiquer que la compilation s'est passée sans erreurs.

11.3 Questions

Les faits étant déclarés, on peut poser des questions dans l'interpréteur. L'interpréteur nous renvoie alors la valeur de vérité de la formule.

```
?- père(loïc,gérard).

Yes

?- père(loïc,maxime).

No
```

La réponse à la deuxième question est négative, car le prédicat `père` n'ayant pas été défini sur le couple `(loïc,maxime)` prend la valeur *faux* sur ce couple.

On peut aussi poser des questions existencielles : « de qui Loïc est-il le père ? »

```
?- père(loïc,X).

X = gérard

Yes
```

Ici le deuxième argument n'est plus une constante mais une variable. *Un nom de variable commence toujours par une majuscule.* L'interpréteur ne renvoie pas *une* mais *toutes* les valeurs possibles.

```
?- père(gérard,X).

X = maxime ;

X = lise ;

No
```

Pour ce faire il faut taper un « ; » après chaque réponse pour que l'interpréteur affiche la réponse suivante. Quand l'interpréteur a affiché toutes les réponses possibles, il renvoie « No ».

Une question peut bien sûr porter sur plusieurs variables.

```
?- père(X,Y).

X = loïc
Y = gérard ;

X = gérard
Y = maxime ;

X = gérard
Y = lise ;

No
```

Les règles sont consultées suivant l'ordre de leur écriture, ce qui n'est pas sans conséquences sur le comportement et le résultat des programmes.

11.4 Conjonction de questions

On cherche des valeurs `Y` et `Z` telles ques `père(loïc,Y)` et `père(Y,Z)` soient vrais.

```
?- père(loïc,Y),père(Y,Z).

Y = gérard
Z = maxime ;

Y = gérard
Z = lise ;

No
```

L'ordre dans lequel la question est rédigée est important : la question est élaborée de la gauche vers la droite et est évaluée de la gauche vers la droite. Nous avons trois règles :

1. père(loïc,gérard).
2. père(gérard,maxime).
3. père(gérard,lise).

La figure 5 nous montre l'arbre de recherche des solutions pour cette question. Si nous réécrivons la question comme ci-dessous :

```
?- père(Y,Z),père(loïc,Y).

Y = gérard
Z = maxime ;

Y = gérard
Z = lise ;

No
```

nous obtenons la même réponse, mais l'arbre de recherche, présenté figure 6, devient tout de suite plus important. Il vaut toujours mieux mettre en tête les prédicats les plus contraignants.

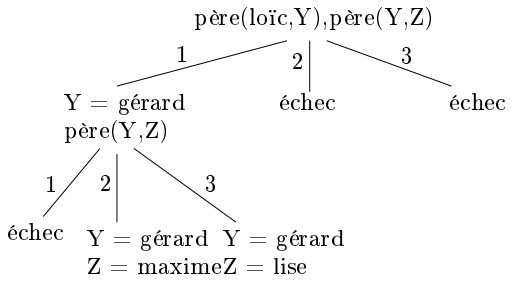


FIG. 5 – Importance de l'ordre d'écriture des prédicats : première partie.

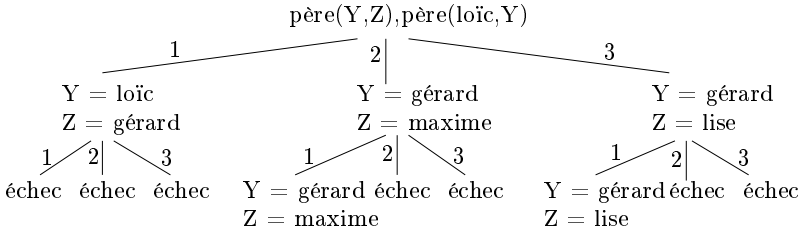


FIG. 6 – Importance de l'ordre d'écriture des prédicats : deuxième partie.

11.5 Introduction de règles de déduction

```
grandpère(X,Y) :- père(X,Z),père(Z,Y).
```

Cette formule signifie que `grandpère(X,Y)` est vrai si `père(X,Z)` est vrai et si `père(Z,Y)` est vrai. Autrement dit, cette formule est équivalente à la formule logique :

$$\texttt{père(X,Z)} \wedge \texttt{père(Z,Y)} \rightarrow \texttt{grandpère(X,Y)}$$

Exemple d'utilisation :

```
?- grandpère(X,lise).

X = loïc ;

No
```

Règles récursives

```
ancêtre(X,Y) :- père(X,Y).
ancêtre(X,Y) :- père(X,Z),ancêtre(Z,Y).
```

Le prédicat `ancêtre` est ici défini par deux règles :

$$\begin{array}{rcl} \texttt{père(X,Y)} & \rightarrow & \texttt{ancêtre(X,Y)} \\ \texttt{père(X,Z)} \wedge \texttt{ancêtre(Z,Y)} & \rightarrow & \texttt{ancêtre(X,Y)} \end{array}$$

Donc, le prédicat `ancêtre` est défini par la formule :

$$\texttt{père(X,Y)} \vee (\texttt{père(X,Z)} \wedge \texttt{ancêtre(Z,Y)}) \rightarrow \texttt{ancêtre(X,Y)}$$

La disjonction est obtenue pas l'introduction de plusieurs règles.

11.6 Principe d'effacement

Pour répondre à une question $q_1, ..., q_n$, c'est-à-dire pour satisfaire une conjonction de relations, Prolog va tenter d'effacer ces « buts » un à un, dans l'ordre où ils se présentent.

1. Soit B la suite initiale des buts. Soit C l'ensemble initial des contraintes. $C = \emptyset$.
2. Si le but est $B = b_1, b_2, ..., b_n$, on efface b_1.
3. Régle à appliquer. Soit p :- $q_1, ..., q_m$ la première des règles du programme telles que la contrainte $C \cup \{p = b_1\}$ ait une solution. Alors $B = q_1, ..., q_m, b_2, ..., b_n$ et $C = C \cup \{p = b_1\}$.
4. Itération : on recommence en 1 sauf si :
   – $B = \emptyset$, auquel cas on a obtenu une solution, la réponse étant C.
   – On bloque sur un but b_1 que l'on ne peut pas effacer, il y a alors échec.
   Dans tous les cas, il y a retour en arrière sur le dernier choix effectué, et une nouvelle tentative pour effacer, d'une autre manière, le dernier but effacé, afin de trouver une *autre* réponse possible.
   Lorsque l'on effectue un nouveau choix pour tenter d'effacer différemment un but `b`, on « défait » toutes les affectations de variables qui ont eu lieu depuis le précédent choix fait pour effacer `b` (*backtracking*).

11.7 Problème classique : représentation et manipulation des listes

– Liste vide : `[]`.
– Liste non vide : `[T|R]`, où `T` est un élément et `R` une liste.
– Liste à trois éléments : `[1, 2, 3]`.

On veut écrire une fonction `concatène(X,Y,Z)` qui est vraie si la liste `Z` est égale au résultat de la concaténation des listes `X` et `Y`. On veut donc pouvoir obtenir les réponses suivantes :

```
?- concatène([1,2],[3],[1,2,3]).

Yes
?- concatène([1,2],[3],Z).

Z = [1, 2, 3]

Yes
?- concatène(X,[3],[1,2,3]).

X = [1, 2]

Yes
?- concatène(X,Y,[1,2,3]).

X = []
Y = [1, 2, 3] ;

X = [1]
Y = [2, 3] ;

X = [1, 2]
Y = [3] ;

X = [1, 2, 3]
Y = [] ;

No
```

Construction de la solution :

```
concatène([],Y,Z) :- vrai si Y = Z
concatène([T|R],Y,Z) :- vrai si Z = [U|S], T = U et  concatène(R,Y,S)
```

Solution

```
concatène([],Y,Y).
concatène([T|R],Y,[T|Z]) :- concatène(R,Y,Z).
```

Fonction `inv`

On veut une fonction qui nous renverse une liste.

```
?- inv([1,2],Y).

Y = [2, 1]

Yes
?- inv(X,[2,1]).

X = [1, 2]

Yes
```

On définit cette fonction au moyen de deux règles :

1. `inv([],[]).`
2. `inv([X|Y],Z) :- inv(Y,T),concatène(T,[X],Z).`

Le comportement de ce prédicat est illustré par l'exemple de la figure 7.

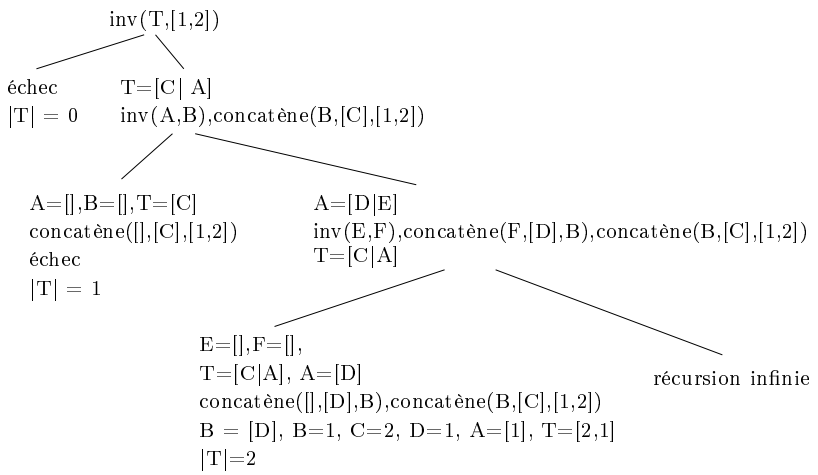


FIG. 7 – Importance de l'ordre d'écriture des prédicats : récursion infinie.

On redéfinit cette fonction au moyen des deux règles :

1. `inv([],[]).`
2. `inv([X|Y],Z) :- concatène(T,[X],Z),inv(Y,T).`

Le comportement de ce prédicat est illustré par l'exemple de la figure 8.

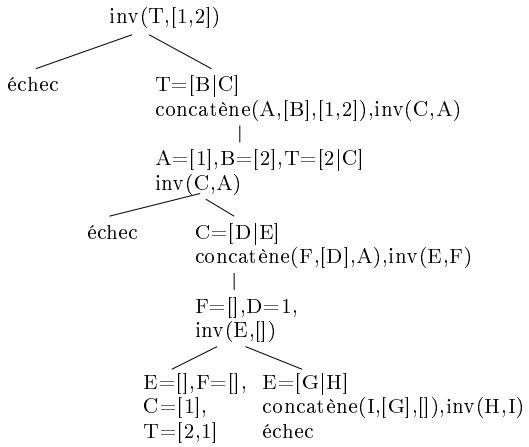


FIG. 8 – Importance de l'ordre d'écriture des prédicats : récursion finie.

12 L'unification ... ou pourquoi 1+1 n'est pas égal à 2

12.1 Retour sur le principe d'effacement

12.1.1 Calcul de la longueur d'une liste

Le prédicat qui calcule la longueur d'une liste est défini au moyen de deux règles :

1. `longueur([],0).`
2. `longueur([T|R], N) :- longueur(R, M), N is M+1.`

Question :

```
?- longueur([3],1).
```

Effacement :

1. On teste la règle 1. Les contraintes sont `[] = [3]` et `0 = 1`. Générer de telles contraintes semble assez facile. Ici, on n'a pas de solution, l'interpréteur renvoie « `No` ».
2. On teste la règle 2. Les contraintes sont ici : `T=3`, `R = []` et `N = 1`. Comment obtient-on la décomposition de la liste argument ?
   Le but devient `longueur(R,M), N is M+1`.

Unifier c'est

1. Tester les égalités `[] = [3]` et `0 = 1`.
2. Tester les égalités `[T|R] = [3]` et `N = 1`. Ici on essaye d'unifier `[T|R]` et `[3]`. Autrement dit, on essaye de rendre « un », identique, les deux expressions.

12.1.2 1+1 ≠ 2

On définit le prédicat suivant :

```
égal_à_deux(X) :- X = 1+1.
```

où « = » est le test général d'égalité.

```
?- égal_à_deux(2).

No
?- égal_à_deux(1+1).

Yes
```

```
?- égal_à_deux(X).

X = 1+1 ;

No
```

Pourquoi ?

12.2 Représentation des prédicats par des arbres

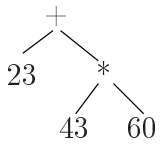


FIG. 9 – Représentation d'une expression arithmétique par un arbre.

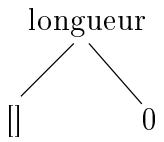


FIG. 10 – Représentation d'un prédicat.

- Expression arithmétique : $23 + 43 * 60$ (figure 9).
- Relation (prédicat) : `longueur([],0)` (figure 10).
- Cas général : prédicat R d'arité n (figure 11).

Avec une telle représentation, les deux expressions $23 + (43 * 60)$ et $23 + (60 * 43)$ sont *a priori* différentes (cf. figure 12).

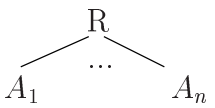


FIG. 11 – Représentation par un arbre : cas général.

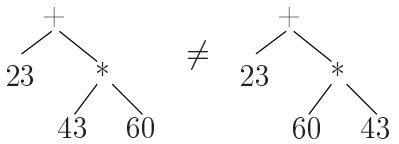


FIG. 12 – Deux expressions *a priori* différentes.

Les structures-arbres présentées ici ne sont pas forcément complètement définies. Prenons l'exemple de l'expression arithmétique : $8 + X * 3$. L'arbre présenté figure 13 représente l'infinité des arbres obtenus en remplaçant la variable X par un arbre quelconque. Ici, X peut être remplacé par une valeur ou, plus généralement, par un arbre.

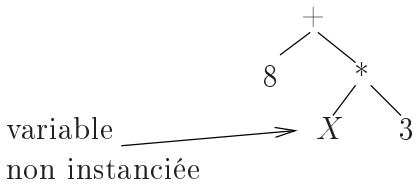


FIG. 13 – Arbre non complètement défini.

Les arbres élémentaires sont des constantes :
- entier
- chaîne de caractères ;
- identificateur ;
- ...

12.3 Unification

Prolog est un langage qui manipule des arbres qui peuvent contenir des parties variables. Un mécanisme de base consiste à décider si deux arbres *peuvent être rendus égaux*, ou non. Le cas échéant, l'égalité est réalisée en affectant des variables aux valeurs figurant dans l'un ou l'autre des arbres considérés.

Algorithme classique d'unification

Soient deux arbres t_1 et t_2.
Unification(t_1, t_2)

1. t_1 est constant ou est une variable de valeur connue, t_2 est constant ou est variable de valeur connue. Alors $t_1 = t_2$.
2. t_1 est une variable de valeur inconnue et t_2 est constant ou est variable de valeur connue. Alors `valeur(`t_1`)` $\rightarrow$ t_2`, vrai`.
3. t_1 et t_2 sont des variables de valeur inconnue. Alors `valeur(`t_1`)` $\rightarrow$ t_2`, vrai`.
4. $t_1 = f(u_1, ..., u_n)$ et $t_2 = f(u'_1, ..., u'_n)$. Alors Unification(u_1, u'_1), ...,Unification(u_n, u'_n).

Exemples

1. `nul(0).`
   ```
   ?- nul(1).

   No
   ?- nul(X).

   X = 0 ;

   No
   ?
   ```
   Unification(0,1) $\Leftrightarrow$ $0 = 1$: faux.
   Unification(X,0) $\Leftrightarrow$ `valeur(X)` $\leftarrow$ `0`, vrai.
2. `un(1).`
   ```
   ?- nul(X), un(X).

   No
   ```
   Unification(X,0) $\Leftrightarrow$ `valeur(X)` $\leftarrow$ `0`, vrai. Unification(X,1) $\Leftrightarrow$ $0 = 1$: faux.
3. `egal(X,X).`
   (a) ```
   ?- egal(Y,Z).

   Y = _G153
   Z = _G153

   Yes
   ```
   `_G153` est un nom de variable.
   (b) ```
   ?- egal(f(0,X),f(Y,1)).

   X = 1
   Y = 0 ;

   No
   ```
   Première unification : `f(0,X)` avec `f(Y,1)`.
   Deuxième unification : 0 avec `Y`.
   Troisième unification : `X` avec 1.

(c) `?- egal(f(X,2,g(Z)),f(g(1),Y,g(4))).`

```
X = g(1)
Z = 4
Y = 2 ;

No
```

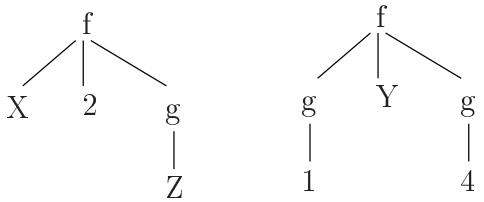


FIG. 14 – Deux arbres à unifier.

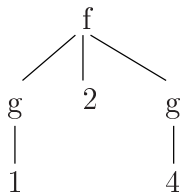


FIG. 15 – Résultat de l'unification.

(cf. figures 14 et 15.) Première unification : `f(X,2,g(Z))` avec `f(g(1),Y,g(4))`.
Deuxième unification : `X = g(1)`.
Troisième unification : `Y = 2`.
Quatrième unification : `g(Z)` avec `g(4)`.
Cinquième unification : `Z=4`.

(d) `egal(f(Y,1,g(X)),f(X,Y,g(2))).`

```
No
```

Première unification : `f(Y,1,g(X))` avec `f(X,Y,g(2))`.
Deuxième unification : `X = Y`.
Troisième unification : `Y = 1` (donc `X=1`).
Quatrième unification : `g(X)` avec `g(2)`.
Cinquième unification : `X=2`, or `X=1`, donc `1=2` : faux.

4. `inf(X,f(aa,X)).`

```
?- inf(Y,Z).

Y = _G147
Z = f(aa, _G147) ;

No
?- inf(Y,Y).
```

Première unification : `X = Y` (`X` et `Y` non connus).
Deuxième unification : `Y = f(aa,X)` et `Y` non connu. Donc `Y` de la forme `Y = f(Z,T)`. `Z = aa`, `T = X`, avec ici `X = Y` et `Y = f (Z,T)`. D'où `T = f(aa,T)`. Ça part à l'infini : objet à définition récursive.

5. `mult(X,Y,Z) :- 2*(X+5) = Y * Z.`

```
?- mult(X,Y,Z).

X = _G165
Y = 2
Z = _G165+5

Yes
?
```

6. `égal_à_deux(X) :- X = 1+1.`

```
?- égal_à_deux(2).

No
```

On tente l'unification de 2 et de 1+1, qui sont deux expressions de racines différentes : 2 et +, d'où l'échec.

13 Contrôle d'exécution et coupure

13.1 Motivation de la coupure

13.1.1 Premier exemple : appartenance à une liste

Soit la fonction qui vérifie qu'un élément est bien dans une liste.

- Première solution :
```
membre1(X,[Y|L]) :- X = Y.
membre1(X,[Y|L]) :- X \= Y, membre1(X,L).
```
- Deuxième solution :
```
membre2(X,[X|_]).
membre2(X,[_|L]) :- membre2(X,L).
```

L'idée ici, c'est que le cas d'égalité a été traité avec la première règle et que l'on n'a pas à en tenir compte lors de l'écriture de la deuxième règle.

Évaluation du travail effectué

Cf. les figures 16 et 17.

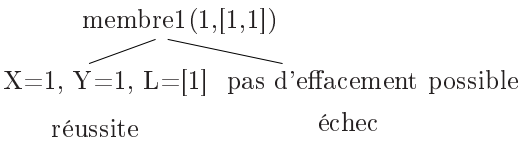


FIG. 16 – Évaluation avec le prédicat `membre1`.

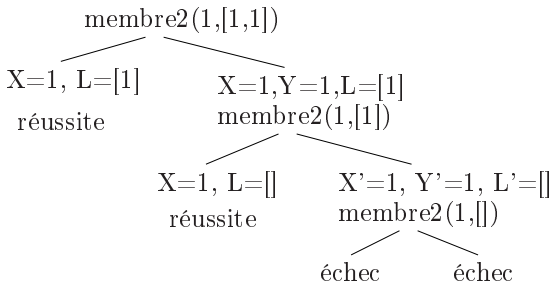


FIG. 17 – Évaluation avec le prédicat `membre2`.

Conclusions :

- L'écriture du programme a de grosses conséquences sur son exécution.
- Seule la première occurence de `X` dans `L` nous intéressait ici. Nous aurions aimé pouvoir spécifier que le programme devait s'arrêter dès la première occurence trouvée.

13.1.2 Deuxième exemple : fonctions booléennes

On s'intéresse ici au codage des fonctions booléennes. On veut associer une valeur booléenne, 0 ou 1, à chaque formule.

Soit une formule `p`. Si `p` s'efface, `p` est vraie, d'où :`valeur_booléenne(p,1) :- p.`

Comment faire pour associer la valeur 0 à `p` quand `p` ne s'efface pas ? Première idée :

```
valeur_booléenne(P,1) :- P.
valeur_booléenne(P,0).

?- valeur_booléenne(membre1(2,[1]),Y).

Y = 0 ;
```

```
No
?- valeur_booléenne(membre1(1,[1]),Y).

Y = 1 ;

Y = 0 ;

No
```

Si p ne s'efface pas, la valeur 0 lui est associée, et tout va pour le mieux. Pas contre, si p s'efface, les valeurs 0 et 1 lui sont associées ! On voudrait que si p s'efface, la seule valeur associée à p soit 1. Solution : la coupure !

13.2 La coupure

Notation : « ! » (point d'exclamation). La coupure permet d'oublier des points de choix lors de l'effacement.

Solution de notre problème de valeurs booléennes :

```
valeur_booléenne(P,1) :- P, !.
valeur_booléenne(P,0).

?- valeur_booléenne(membre1(2,[1]),Y).

Y = 0 ;

No
?- valeur_booléenne(membre1(1,[1]),Y).

Y = 1 ;

No
```

Interprétation

Nous avons deux cas à considérer :

1. p s'efface (p peut être évalué à vrai, cf. figure 18). Une fois que p a été évalué à vrai, on « évalue » la coupure (« ! »). Celle-ci supprime le *backtracking* sur les choix mis en attente entre le moment où elle a été introduite et celui où elle a été effacée.

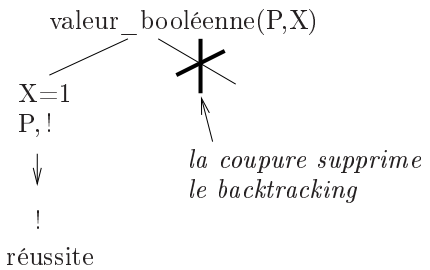


FIG. 18 – Coupure : cas où p s'efface.

2. p ne s'efface pas (p ne peut pas être évalué à vrai, cf. figure 19). Comme p ne s'efface pas, la coupure n'est pas atteinte et il y a *backtracking*.

Dans la recherche d'appartenance d'un élément à une liste, la coupure permet d'arrêter le programme lors de la découverte de la première solution possible :

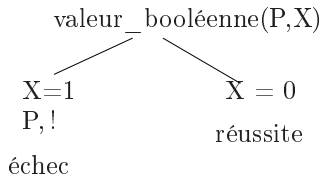


FIG. 19 – Coupure : cas où p ne s'efface pas.

```
membre3(X,[X|L]) :- !.
membre3(X,[Y|L]) :- membre3(X,L).
```

13.3 Vue théorique

Soit $b_1, ..., b_n$ la suite de buts à effacer à un instant donné, et soit $c_1, ..., c_m$ la suite des choix en attente à cet instant, du plus ancien au plus récent. $c_1, ..., c_m$ sont donc des buts déjà effacés. *Notation* : $[c_1, ..., c_m]b_1, b_2, ..., b_n$.

Supposons que pour effacer b_1 on utilise la règle p :- $q_1, ..., q_i, !, q_{i+1}, ..., q_l$, avec la contrainte $b_1 = p$. On obtient donc le nouvel état :

$$[c_1, ..., c_m, b_1]q_1, ..., q_i, !, q_{i+1}, ..., q_l, b_2, ..., b_n$$

Les effacement successifs donnent :

$$[c_1, ..., c_m, b_1, q_1]q_2, ..., q_i, !, q_{i+1}, ..., q_l, b_2, ..., b_n$$

$$[c_1, ..., c_m, b_1, q_1, ..., q_i]!, q_{i+1}, ..., q_l, b_2, ..., b_n$$

$$[c_1, ..., c_m]q_{i+1}, ..., q_l, b_2, ..., b_n$$

L'effacement de la coupure supprime donc tous les choix (quelque soit la règle concernée) mis en attente, depuis le plus récent et jusqu'à celui dont l'effacement a provoqué l'apparition de la coupure. Ici, la suite des choix en attente est rétablie à ce qu'elle était juste avant l'effacement de b_1.

Autre exemple : la conditionnelle

On veut réaliser ici une conditionnelle *if C then A else B*.

```
ifthenelse(C,A,B) :- C, !, A.
ifthenelse(C,A,B) :- B.
```

Si `C` s'évalue à vrai on s'interdit d'avoir recours à la deuxième règle et on évalue `A`. Sinon, on évalue `B`.

Attention : les règles doivent être écrites dans cet ordre !

Retour sur les fonctions booléennes

```
et(P,Q,1) :- P, Q, !.
et(P,Q,0).
ou(P,Q,1) :- P, !.
ou(P,Q,1) :- Q, !.
ou(P,Q,0).
```

13.4 Les dangers de la coupure

Il faut toujours prouver qu'introduire la coupure ne va pas nous faire croire, à tort, qu'un problème n'a pas de solutions.

```
homme(pierre).
homme(jacques).
grand(jacques).
?- homme(X),grand(X).

X = jacques ;

No

?- homme(X),!,grand(X).

No
```

13.5 La négation

`non(P) :-` si `P` est vrai alors renvoyer faux... Comment coder une telle propriété ? On utilise le prédicat `fail` qui est toujours faux.

```
non(P) :- P, fail.
non(P).
```

Si `p` est vrai alors faux, sinon vrai. Le problème ici, c'est que si `p` est vrai, `non(P)` vaut simultanément vrai et faux. On doit donc utiliser la coupure :

```
non(P) :- P, !, fail.
non(P).
```

On effectue ce que l'on appelle une « négation par l'échec ». La coupure oblige la première régle à échouer et interdit le *backtracking*.

Autre version :

```
non(P) :- P, !, 0 = 1.
non(P).
```

Signification théorique de la négation

`p` est vrai, ou `p` s'efface, s'il existe **une** utilisation qui permet d'effacer `p`. `p` est vrai s'il existe au moins une instanciation des variables qui le composent qui lui permette de s'évaluer à vrai. Donc `non(P)` est vrai si et seulement si `p` ne s'évalue **jamais** à vrai.

On suppose que l'on dispose d'un prédicat `chiffre` qui s'évalue à vrai sur les éléments de l'ensemble $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

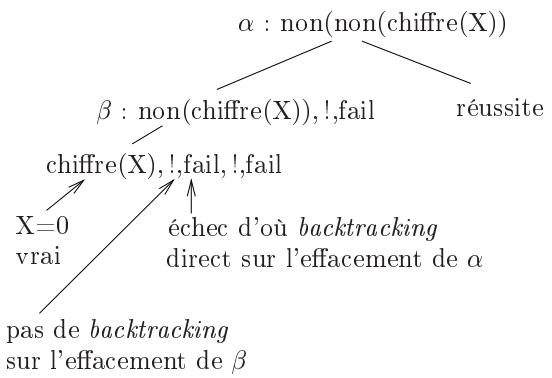


FIG. 20 – Signification de la double négation.

Dans l'exemple de la figure 20, on a vérifié que `chiffre(X)` pouvait s'effacer, mais on n'a pas créé de conditions sur `X`, contrairement à un simple appel à `chiffre(X)`. Donc ici : `non(non(chiffre(X)))` signifie « existe-t-il une solution ? », alors que `chiffre(X)` signifie « qu'est-ce qu'il existe comme solution ? ».

```
?- chiffre(X).

X = 0 ;

X = 1 ;

X = 2 ;

X = 3 ;

X = 4 ;

X = 5 ;

X = 6 ;

X = 7 ;

X = 8 ;

X = 9 ;

No
?- non(non(chiffre(X))).

X = _G219 ;

No
```

13.6 Exemple : résolution des trinômes du second degré

On s'intéresse donc aux équations de la forme : $a.x^2 + b.x + c$.

```
résoudre(0,0,0,X) :-
   !, string_to_list(X,"indéfini").
résoudre(0,0,_,X) :-
   !, string_to_list(X,"aucune solution").
résoudre(0,B,C,X) :-
   !, X is -C/B.
résoudre(A,B,C,X) :-
   D is B*B - 4*A*C, solutions(A,B,D,X).
solutions(A,B,0,S) :-
   X is - B/(2*A), !,
   string_to_list(T,"Solution double : "),
   string_to_atom(SX,X),
   string_concat(T,SX,S).
solutions(A,B,D,S) :-
   D < 0, !, X is -B/(2*A),
   E is sqrt(-D), Y is E/(2*A),
   string_to_list(P," + i*("),
   string_to_list(N, " - i*("),
   string_to_list(PAR1,"), "),
   string_to_list(PAR2,")"),
   string_to_atom(SX,X),
   string_to_atom(SY,Y),
   string_concat(SX,P,PP),
   string_concat(PP,SX,PPP),
   string_concat(PPP,PAR1,PPPP),
   string_concat(SY,N,NN),
```

```
   string_concat(NN,SY,NNN),
   string_concat(NNN,PAR2,NNNN),
   string_concat(PPPP,NNNN,S).
solutions(A,B,D,S) :-
   E is sqrt(D), X is (-B+E)/(2*A),
   Y is (-B-E)/(2*A),
   string_to_atom(SX,X),
   string_to_atom(SY,Y),
   string_to_list(ESP,", "),
   string_concat(SX,ESP,SS),
   string_concat(SS,SY,S).
```