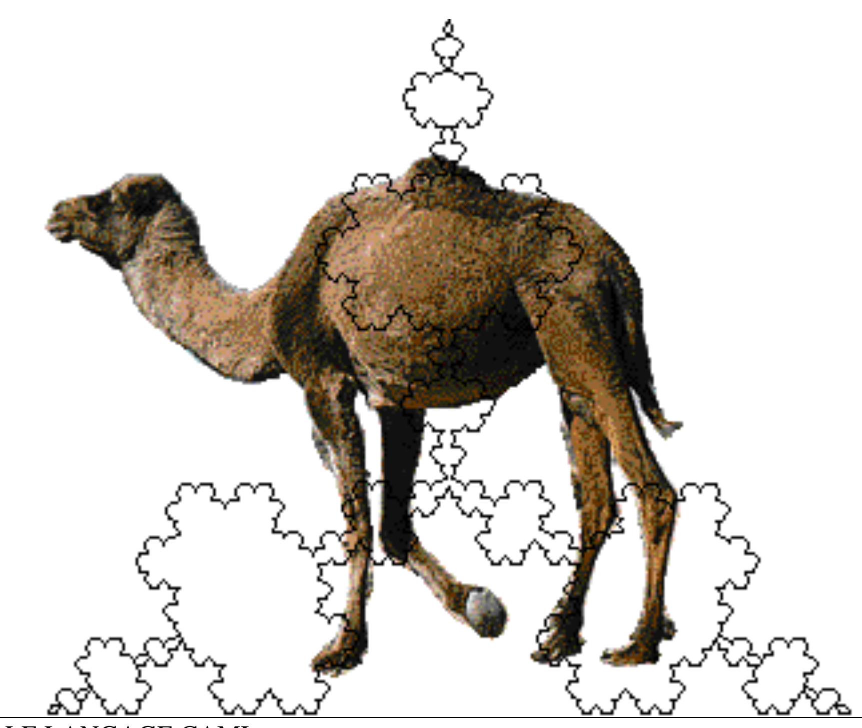
PROGRAMMATION PAR L'EXEMPLE EN CAML

128 problèmes

L.Gacôgne 2003



I	LE LANGAGE CAML	5
II	PREMIERES FONCTIONS, RECURSIVITE TERMINALE	17
III	TRAITEMENT DES LISTES	27
IV	PARCOURS D'ARBRE EN PROFONDEUR	39
V	CHAINES DE CARACTERES	49
VI	AFFECTATION, ITERATION	55
VII	VECTEURS ET TABLEAUX	61
VIII	ARTICLES (RECORDS)	83
IX	TYPES RECURSIFS	91
X	LE PASSAGE PAR NECESSITE ET LES FLUX	109
XI	LE HASARD : ALGORITHMES D'EVOLUTION	113
XII	GRAPHIQUE	139
XIII	MANIPULATION DE LA SOURIS	159
XIV	EXEMPLES DE SYSTEMES MULTI-AGENTS	167

INTRODUCTION

Ce livre est un recueil de plus de 256 exercices et problèmes corrigés, essentiellement dans le langage Caml (version Caml Light). Il n'est pas axé sur l'étude d'un langage de programmation et de son implémentation mais plutôt sur ces exercices et différents styles de programmation. S'il est faux de dire qu'un programme court est un programme efficace, car les contre-exemples sont faciles à trouver, cependant, cela est globalement vrai et il faut toujours chercher la brieveté. Rares seront les fonctions dépassant dix lignes. Ce livre peut faire suite à *"512 problèmes corrigés de programmation en Pascal, Lisp, C++ et Prolog" Ellipses 1995*, car hormis des exemples de base très classiques, ces problèmes sont nouveaux. Ils proviennent, en grande partie, de cours et travaux dirigés donnés à l'Institut d'Informatique d'Entreprise d'Evry.

Le découpage des chapitres est fait suivant les stuctures de données et les fonctions prédéfinies s'y rapportant. Ainsi, après les premiers exemples de fonctions, voit-on la structure de liste, puis les chaînes de caractères, tableaux, articles, et des exemples graphiques...

Depuis les premiers langages de programmation dits évolués, on a coutume de parler de programmation imérative dans le sens où on donne une succession d'ordre (en différé) à la machine. Puis, on parle de langages fonctionnels dans la mesure où on décrit des fonctions f(x) = y, la description de f expliquant comment obtenir la construction du résultat y à partir de la valeur donnée à l'argument désigné par le paramètre x (tous ces mots ont leur importance). Naturellement, une vraie fonction ne modifie rien, c'est une simple définition qu'on utilisera ultérieurement, elle n'a pas "d'effet de bord". La description d'un algorithme qui permet de construire le résultat y pour chaque entrée x, et son utilisation ainsi que les problèmes d'entrée-sortie sont alors entièrement séparés. C'est d'ailleurs ainsi que, le plus possible, on doit procéder. Pascal et C qui sont en grande partie fonctionnels mélangent ces deux types de programmation. Caml, en allant beaucoup plus loin, permet également la programmation impérative, il n'est pas "fonctionnel pur" comme on le verra surtout avec l'affectation et les tableaux. Cependant, Caml a l'immense avantage de calculer automatiquement les types, soulageant ainsi le programmeur, et de permettre de vraies fonctions renvoyant des objets structurés y compris renvoyer des fonctions, contrairement au Pascal, sans que le programmeur n'ait à se soucier du jeu des pointeurs.

La version utilisée est Caml Light. Cette version, sans doute plus lente, permet cependant une meilleure lisibilité.

Nous ferons quelques allusions lors d'exemples à d'autres langages comparables, tels que Lisp, langage fonctionnel non typé, qui autorise une plus grande liberté et Haskell, qui, avec les mêmes possibilités que Caml, est purement fonctionnel, apporte un véritable système de classes et d'héritage, un autre mode d'évaluation permettant un pas de plus dans l'écriture d'objets mathématiques, et une très grande lisibilité.

CHAPITRE I

LE LANGAGE CAML

En 1977, est né en Ecosse le langage fonctionnel typé ML inspiré du λ-calcul de Church. Son fils Caml (Categorical Abstract Machine Language) est né dix ans plus tard en coopération avec l'INRIA (accès sur <caml.inria.fr> pour les versions Mac, Unix ou Windows).
Caml est un langage fonctionnel, comme Lisp, Miranda ou Haskell, ce qui signifie que l'on déclare pour l'essentiel des fonctions au sens mathématique. Et donc, il est est possible de définir une fonction renvoyant des fonctions, des relations comme fonctions booléennes, une fonction de fonctions, des couples, triplets ... ce qui en fait un langage bien plus évolué que les langages classiques, même si ceux-ci s'inspirent de plus en plus de l'aspect fonctionnel, comme C par exemple, où il n'y a normalement que des fonctions.

Caml est donc un langage évolué et puissant qui gagne un certain nombre de compétitions de programmation (voir <icfpcontest.cse.ogi.edu>) mais Haskell est un redoutable concurrent.
La version utilisée dans toute la suite est Caml Light, car la version Objective Caml et son système de classes axé sur la programmation modulaire, n'est pas justifié pour les exercices qui sont l'objet de ce livre. Les différences de syntaxe sont d'ailleurs minimes, par exemple, ce sont des apostrophes au lieu d'anti-apostrophes pour les caractères et on prendra toujours des minuscules pour nommer les paramètres réservant les majuscules aux constructeurs de type.

Contrairement à la famille des Lisp, Scheme et aussi à Prolog, Caml possède des types de données très rigoureux, et non seulement les contrôle comme dans la plupart des langages, mais les calcule automatiquement, (ce qui évite presque toujours au programmeur de les déclarer). Ces types sont nommés de la façon abstraite la plus générale possible (notés a, b, c ...) en accord avec la définition de fonction donnée.

Comme dans tous les langages fonctionnels, un programme n'est qu'une suite de définitions de fonctions et de données qu'il vaut mieux écrire préalablement dans un éditeur. Cette suite de fonctions doit absolument, en Caml, être donnée dans le "sens ascendant". C'est à dire qu'une fonction f qui en appelle une autre g ne doit figurer qu'après g. En cas d'appels mutuels, les définitions seront liées ensemble, par un "and". On peut souvent, si on le désire, écrire une fonction "principale" qui va se contenter d'appeler les autres.

Sous Unix on appelle Camlight, puis on en sort avec quit ();;
Après l'ouverture de Caml, on inclue le programme "nom.ml" par "include" ou bien par un simple copier-coller qui place les définitions dans l'interpréteur. Dans tous les cas, la mise au point, se fera toujours suivant le même cycle : on introduit une expression, l'interpréteur la lit, l'évalue et retourne son type et sa valeur, ou bien un message d'erreur approprié.
Une fois mis au point, "compile" créera un "nom.zi" (i pour interface) et un "nom.zo" (o pour objet), puis "load-object" chargera les fonctions compilées qui s'utiliseront de la même façon mais plus rapidement.

1. Les types simples de Caml

Les types simples de Caml sont :

- Les booléens bool (true et false), n'ayant que les deux valeurs de vérité vrai et faux,
- Les entiers int (les entiers de -2^{30} à $2^{30} - 1$),
- Les caractères char, attention ils sont délimités avec les anti-apostrophes, ainsi par exemple < let a = ‘a‘;; >,
- Les chaînes de caractères "string" (encadrés de guillemets), ainsi < let s = "azerty";; >,
- Les réels float (à ce propos entiers et réels ne sont pas compatibles à moins d'en donner l'ordre par #open "float" dans certaines versions de Caml, sinon + - * / doivent être suivis d'un point).
- Le type unit est le type dont l'unique valeur notée () est vide c'est le "void" du C.

Les types forment une algèbre avec ces constantes (int, unit, char...), des variables 'a, 'b,... des constructeurs unaires (list, stream, vect...) et binaires (*, ->...).

2. Déclarations

C'est le mot réservé "let" qui permet d'indiquer par exemple :
"soit x ayant la valeur 2" par < let x = 2;; >
ou encore "soit f la fonction qui, à l'argument x, fait correspondre x + 2" par < let f x = x + 2;; >.

Il est possible d'utiliser le prime pour des identificateurs de variables ou de fonctions. Concernant les paramètres, on peut écrire par exemple < let x = 2 and x' = 3 in x + x';; > qui renverra 5.
Remarque, en ML on peut même écrire l'affectation multiple "sémantique" sous forme de liste avec < let [x; _; y; 4] = [1; 2; 3; 4];; >

En Caml, par contre, on écrira la déclaration d'un couple comme < let x, y = (2, 3);; > ou bien deux déclarations simultanées comme < let x = 2 and y = 3;; >, ce qui est clair mis à part le fait que = est ici le symbole de définition et non pas la relation d'égalité, et que "and" n'a pas à voir ici avec la conjonction logique notée par le symbole & ou par &&.

Tout sera déclaré en Caml avec "let" ou bien "let rec" pour les définitions récursives, la commande d'évaluation étant le double point-virgule (cet encadrement n'étant même plus nécessaire en Haskell). Ci-dessous un exemple de session où Caml retourne donc à chaque fois le type et la valeur. La variable x est d'abord affectée, puis réaffectée.
La ligne #x;; correspond juste une demande de ce qu'est la valeur de x.

```
#let x = 2;;
x : int = 2

#let x = 1 and y = 3;;
x : int = 1
y : int = 3

#x;;
- : int = 1

#y;;
- : int = 3

#x + y;;
- : int = 4

#x - y;;
- : int = -2
```

Dans l'interpréteur Caml, c'est le double point-virgule qui donne l'ordre d'évaluation, dans tous les exemples qui suivent, l'expression lue par l'interpréteur est encadrée par le "prompt" #, et par le double point-virgule ;; la réponse fournie à la ligne suivante est précédée de -: et de l'appartenance à un type signifiée par le signe :.

Le symbole de renvoi -: pourra être remplacé dans la suite par une flèche, un signe égal ou plutôt une petite main ☞ ce qui est plus clair.

Le prompt # sera également omis, passé les premières pages. Afin de soulager la lecture, on ne mettra plus non plus l'appartenance au type lorsque ce n'est pas vraiment nécessaire.

Voyons maintenant un exemple de session avec des chaînes de caractères. Le symbole de concaténation est la chapeau ^. On y déclare deux variables m et b, dans la réponse de Caml, le":" figure toujours l'appartenance à un type, puis on interroge Caml sur ce que sont m, b, leur concaténation, et la concaténation avec un espace intercalé.

```
#let m = "merci" and b = "beaucoup";;
m : string = "merci"
b : string = "beaucoup"

#m;;
- : string = "merci"

#m^b;;
- : string = "mercibeaucoup"

#m ^ " " ^ b;;
- : string = "merci beaucoup"

#m ^ " " ^ m ^ " " ^ m;;
- : string = "merci merci merci"
```

Exemple d'interrogation de Caml grâce aux fonctions logiques & ou noté && pour "et", "or "ou bien la double barre || pour "ou", if...then...else... (à noter que cette dernière est une fonction à trois arguments de type bool -> a -> a -> a.)
Remarquons encore que dans un programme, les commentaires doivent être encadrés par des parenthèses étoiles.

```
#false or (true & (2 = 2)) or (2 < 1) ;;
☞  bool = true (* le signe = est ici l'égalité, souvent désignée == *)

#let x = 2 in if x = 3 then "oui" else "non";; ☞  string = "non"

#let b = true in if b then "oui" else "non";; ☞  string = "oui"

#let p = false and q = false in not(p & q);;
☞ bool = true (* opérateur logique de Sheffer, "nand" *)

#let p = true and q = true in not(p & q);;  ☞ bool = false

#let p = false and q = false in if p &  q then false else true;;
☞  bool = true (* le  "if" définit ce même opérateur de Sheffer  *)

#let p = false and q = true in if p &  q then false else true;; ☞  bool = true

#let p = true and q = false in if p & q then false else true;; ☞ bool = true

#let p = true and q = true in if p & q then false else true;; ☞  bool = false
```

3. Les types structurés : couples, triplets et n-uplets

Les n-uplets permettent de grouper des valeurs typées différamment. Le séparateur dans les n-uplets est la virgule, et ce sont les parenthèses qui servent de manière facultative à leur encadrement. Ci-dessous, trois exemples :

```
# 7 + 5, 8 < 9, 5 < 0, 1 = 2 ;;
(* remarquons que l'écriture (7+5, 8<9, 5<0, 1=2) est plus familière *)
☞ int * bool * bool * bool = 12, true, false, false

#(1, 2.3);;
☞  int * float = 1, 2.3

#((1, 2), 3);;
☞   (int * int) * int = (1, 2), 3
```

Mais, attention, la virgule n'est pas un simple symbole de séparation, c'est un constructeur de couples, triplets, n-uplets... L'espace, lui, comme en Lisp, est un véritable séparateur. Dans une déclaration comme :

```
#let f(x, y) = (x+y, x-y);;
```

les premières parenthèses sont indispensables, les secondes facultatives car sinon f x, y serait pris comme le couple (f(x), y).
Exemples de déclaration de constantes (= est le signe de définition) :

```
#let (x, y) = (2, 3) ;; (* ou bien  #let x, y = 2, 3 ;; car les parenthèses sont ici facultatives *)
☞   y : int = 3                x : int = 2

#let c = (3, "rien");;
☞ c : int * string = 3, "rien"
```

4. Autres types structurés

En anticipant un peu sur les chapitres suivants, on donne des exemples de calcul de types renvoyés par Caml lorsqu'on lui demande l'évaluation d'expression diverses. Les listes et vecteurs (notions très différentes de par leurs représentations internes, qui sont exposés plus loin se reconnaissent respectivement par les encadrements :

[...; ...; ...] pour les listes, par exemple la succession [1; 2; 3] et :

[|...; ...; ...|] pour les vecteurs, par exemple le vecteur [|1; 2; 3|].

```
#let c = (1, 2, 3) and q = [1; 2; 3] and v = [|1; 2; 3|];;
☞      c : int * int * int = 1, 2, 3
       q : int list = [1; 2; 3]
       v : int vect = [|1; 2; 3|]

#[(1, "azerty"); (2, "edf"); (3, "ratp")];;
☞ (int * string) list = [1, "azerty"; 2, "edf"; 3, "ratp"]

#[|[1; 2]; [3; 4; 5; 6]; [0; 1]; [4]|];;
☞ int list vect = [|[1; 2]; [3; 4; 5; 6]; [0; 1]; [4]|]

#[[1 = 1; 2 > 1]; []; [8 = 4; 1 < 2]];;
☞ bool list list = [[true; true]; []; [false; true]]
```

5. Remarque sur la surcharge des opérateurs

La surcharge signifie qu'un même symbole ou nom de fonctions peut être utilisé pour des types différents, voire des opérations très différentes. Ainsi le signe + servirait aussi bien pour l'addition des entiers que pour celle des réels, en fait ce n'est pas le cas en Caml.
Il n'y a pas de surcharge, y compris dans la version Ocaml, c'est à dire que 2 + 3 utilise l'addition + des entiers et 2.5 +. 3.2 celle +. des réels. De même les affichages print_string, print_int, print_char sont chacune dédiée à un paramètre d'un type bien déterminé. Cet inconvénient oblige naturellement à faire sans arrêt des conversions de type grâce à des fonctions prédéfinies comme int_of_char, float_of_int...

Cependant le symbole d'inégalité < convient pour les "int" et "float" et surtout le signe d'égalité = signifie à la fois la définition d'une constante, le prédicat (heureusement polymorphique) d'égalité, pour tous les types d'objets considérés, mais aussi l'affectation d'une variable dans un : "for ... = ... to ... do ... done".

Exemple de session avec réponses (évaluations ou messages d'erreur) de Caml :

```
#2+3;; ☞ int = 5
#2.5 +. 3.5;; ☞ float = 6.0
#2 + 3.;;
>      ^^
Cette expression est de type float,
mais est utilisée avec le type int.

#2 +. 3.;;
>^
Cette expression est de type int,
mais est utilisée avec le type float.

#2 < 3;; ☞ bool = true
#3 < 2;; ☞ bool = false
#2 <. 3.;;
>^
Cette expression est de type int,
mais est utilisée avec le type float.

#2. <. 3.;; ☞ bool = true
#2. < 3.;; ☞ bool = true
```

6. Premières définitions de fonctions

Répétons l'essentiel qui est, qu'en Caml, il n'y a que des fonctions, on arrive donc, avec le moyen de définir des fonctions, au vif du sujet. Dans toute la suite, tout problème devra s'analyser et se traduire par un découpage en différentes fonctions.
On peut redéfinir, par exemple, la fonction "valeur absolue" (cette fonction est déjà, bien sûr, prédéfinie ainsi que "abs_float" pour les réels) :

```
let abs x = if x < 0 then -x else x ;; ☞   abs : int -> int = <fun>
```

```
Exemple          abs (-5) ;; ☞ int = 5   (* est un exemple d'utilisation *)
```

Plus mathématiquement encore on peut définir :

```
let abs = fun x -> if x < 0 then -x else x;;
```

Dans la syntaxe ML ou en Haskell, ce serait < abs (x) = \ x if x < 0 then -x else x >.

Les parenthèses peuvent être omises s'il n'y a pas ambiguité, "let f x = ..." ou "let f(x) = ..." sont identiques ainsi que les requêtes f 4 ou f(4) ou (f 4), mais pas f -4 qui doit être écrit f (-4) sinon l'analyseur prendra f comme fonction s'appliquant d'abord sur un opérateur, ici la soustraction, ce qui donnera lieu à un message d'erreur.
La construction automatique du type est alors faite sans ambiguité, comme ce type est toujours renvoyé dans une déclaration, on peut voir ainsi deux fonctions ad1 et ad2 (différentes mais donnant le même résultat) typées différemment :

```
let ad1 x y = x + y;; ☞ ad1 : int -> int -> int = <fun>
let ad2 (x, y) = x + y;; ☞ ad2 : int * int -> int = <fun>
```

```
Exemples    ad1 3 5;; p int = 8
            ad2 (3 , 5);; ☞ int = 8
```

Comme il est dit plus haut, il est possible d'écrire une fonction générique (ou polymorphe) d'égalité et une définition polymorphique de premier et second (ces fonctions "fst" et "snd" existent déjà) :

```
let egal (x, y) = (x = y) ;; ☞ egal : 'a * 'a -> bool = <fun>
```

```
Exemple     egal (2, 3) or egal ("a", "a") ;; ☞ bool = true
```

```
let fst (x, y) = x and snd (x, y) = y ;; ☞    fst : 'a * 'b -> 'a = <fun>
                                              snd : 'a * 'b -> 'b = <fun>
```

```
Exemple     fst (2, "a") + snd (true, 3) ;; ☞ int = 5
```

7. Définition par cas

Avec la fonction réservé "match ... with ...", comme avec "fun", une unification avec les formes données sera tentée, (le _ signifie une valeur quelconque comme en Prolog).
Soit par exemple, la définition par cas de l'implication logique dont on sait qu'elle est vraie sauf dans le cas de "vrai" implique "faux". On donne deux définitions équivalentes et il faut noter que dans la seconde, ce paramètre anonyme _ désigne un couple.
On reprend ensuite la définition de l'opérateur de Sheffer déjà donné plus haut par :

$$p \mid q = \text{not } (p \ \& \ q)$$

Mais ici, cette fonction booléenne est définie par tous les résultats possibles.

```
let imp = fun     true false -> false
                  | _ _ -> true;;
        ☞  imp : bool -> bool -> bool = <fun>

let imp x y = match (x, y) with   (true, false) -> false
                                  | _ -> true;;
        ☞  imp : bool -> bool -> bool = <fun>

let sheffer = fun         true true -> false |
                          _ _ -> true;;
        ☞ sheffer : bool -> bool -> bool = <fun>
```

8. Le passage des paramètres en Caml

On appelle "paramètres" les identificateurs que l'on a choisit de nommer dans l'en-tête d'une fonction. Ainsi x est le nom choisi dans la définition de la fonction "abs" plus haut.
On distingue alors comme "argument" les expressions figurant lors de l'appel d'une fonction, ainsi pour la fonction "abs", si on demande à l'interpréteur Caml d'évaluer < abs (4 - 7);; >

l'argument est ici (4 - 7) qui va d'abord être évalué à -3, et le x figurant dans la définition de abs, sera partout remplacé par -3. C'est le "passage par valeur".

Une liaison est cette association temporaire entre un identificateur (ici x) et une valeur et un environnement est un ensemble de liaisons. On distingue la portée statique (ou lexicale) d'une liaison, comme en Caml ou en Scheme, par le fait qu'une fonction détermine cette liaison une fois pour toute, au moment de sa définition.
Ainsi par exemple dans la séquence suivante, la modification de a, ultérieure à la définition de "translation", ne lui change rien, alors que la valeur de a, variable globale, a changé.

let a = 4;;
let **translation** x = x + a;;

| Exemple translation 3;; ☞ 7

let a = 2;;

| Exemple translation 3;; ☞ 7

A l'inverse, la portée dynamique signifie, comme généralement dans les versions de Lisp que chaque appel d'une fonction utilisera les valeurs actuelles, des variables.

9. La currification

Il s'agit de définir des fonctions de plusieurs variables sans utiliser le produit cartésien. L'idée vient de Frege en 1893 en établissant que si F(A, B) désigne l'ensemble des applications de A vers B, alors F(A*B, C) et F(A, F(B, C)) sont isomorphes.
Par exemple une fonction à deux arguments peut être considérée comme une fonction d'une variable (la première) dont le résultat est elle-même une fonction d'une variable (la seconde).

Ainsi, si f est (A*B -> C) alors (Curry f) sera (A -> (B -> C)) et en Caml l'écriture f x y z inférera obligatoirement que f est de type (A -> (B -> (C -> D))) et cette expression sera prise pour une succession d'application de fonctions ((f x) y) z avec (f x) de type A -> (B -> C)) et ((f x) y de type (C -> D).

On adopte en Caml la convention du parenthèsage à gauche, c'est à dire que f g x signifie (f g) x.
Par exemple, avec les notations du Caml, une fonction :
('a -> 'b) -> ('c -> 'd) peut s'écrire en fait :
('a -> 'b) -> 'c -> 'd.
Une fonction currifiée reçoit donc ses arguments un à un. Mais aussi une fonction currifiée est appliquable partiellement, est plus rapide (des constructions de n-uplets en moins) et est privilégiée à la compilation.
A première vue, la currification est une opération ramenant toutes les fonctions à des compositions de fonctions unaires, et si f a été définie comme une fonction à deux arguments, on peut écrire :

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

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

let **comp** (f, g) = fun x-> f(g(x));;
☞ comp : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b = <fun>

let **carre** x = x*x and **cube** x = x*x*x;;
☞ carre : int -> int = <fun> cube : int -> int = <fun>

```
Exemples    curry comp carre cube;; ☞ int -> int = <fun>
            curry comp carre cube 2;; ☞ 64
```

Bien entendu, la composition peut être définie de façon currifiée directement :

```
let comp f g = fun x-> f(g(x));; ☞ comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

```
Exemple    comp carre cube 3;; ☞ 729
```

Notons encore que l'écriture que nous prendrons fun x y -> e est équivalente à function x -> function y -> e. Supposons maintenant qu'une fonction currifiée soit définie par :

let f h x y z = x + y*h z;;

Alors, le type de h est necéssairement (a -> int) et celui de x et de y est int car la présence des symboles + et * apporte cette contrainte.
Ce qui devient vraiment intéressant ensuite, c'est d'interroger Caml sur des "préfixes" de la même expression, ainsi l'avant-dernière (sans le "z") est obligatoirement une fonction d'un type inconnu que Caml nommera 'a, dans int, c'est la fonction dont le paramètre est nommé z dans la toute première expression.

```
let f h x y z = x + y*h z;;
☞ f : ('a -> int) -> int -> int -> 'a -> int = <fun>

f succ;;
☞ int -> int -> int -> int = <fun>

f string_length x y;;     (* string_length  est la longueur d'une chaîne *)
☞ string -> int = <fun>

f list_length x;;         (* list_length est le nombre d'éléments d'une liste *)
☞ int -> '_a list -> int = <fun>

f cube 3;;
☞ int -> int -> int = <fun>

f succ 3 5;;
☞ int -> int = <fun>

f cube 2 3 5;;
☞ int = 377
```

10. Exemple de fonction renvoyant une fonction

On veut définir la translation de paramètre a comme la fonction qui "ajoute a", rien de plus simple :

```
let trans a x = a + x;;  (* dans la syntaxe ML ou en Haskell, ce serait trans(a) = \ x (a + x);; *)
      ☞ trans : int -> int -> int = <fun>
```

```
Exemples    trans 2 ;; ☞ int -> int = <fun>
            trans 2 3 ;; ☞ int = 5
```

Remarque
En cas de manque de parenthèses, un symbole de fonction est toujours prioritaire, par ailleurs -> est prioritaire, il faut donc toujours des parenthèses autour de (a :: q) -> ... sauf à utiliser au lieu de "fun" le mot réservé "function" qui n'attend qu'un seul argument.

De façon anonyme, il est tout à fait possible de demander l'évaluation de l'application d'une expression fonctionnelle sur un argument :

Exemple (fun x -> x + 3) 7 ;; ☞ int = 10

11. Exemples de calculs de types

On donne ici une suite d'expressions dont il n'est pas toujours facile de comprendre les contraintes qui ont amené Caml à renvoyer ses résultats.
Prenons l'exemple suivant, il s'agit d'une définition de f s'appliquant sur 3 arguments, d'abord, celui qui est nommé g, et qui, d'après le corps de la définition est obligatoirement une fonction s'appliquant sur un couple. Ceci signifie que si x est de type a et y de type b, alors g(x, y) est de type c.
Aussi, l'interpréteur Caml attribue-t-il des noms de types a, b, c respectivement pour les arguments x, y et pour le résultat de g. Puis, Caml va indiquer que le type du premier argument de f est donc a*b -> c, à la condition que celui du second x soit a et celui du dernier y soit b, en ce cas, le type retourné sera celui de g, c'est à dire c.
Tous les autres exemples doivent pouvoir s'expliquer de cette façon, ainsi :

```
let f g x y = g (x, y);; ☞  f : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>

let h f g x y = f(x+y) - x*g(x-y);; ☞  h : (int -> int) -> (int -> int) -> int -> int -> int = <fun>

let f g h x y z = (h x) ^(g (y-z));;
☞  f : (int -> string) -> ('a -> string) -> 'a -> int -> int -> string = <fun>
```

Pour ce dernier exemple, il faut rappeler que ^ est la concaténation des chaînes et que le type de x est inconnu.
Pour l'exemple suivant, les symboles unaires f et g étant prioritaires sur ceux, comme +, qui sont binaires, il est clair que les types retournés par f et g sont obligatoirement int.

```
let f g h x y = g x + h y;;
☞ f : ('a -> int) -> ('b -> int) -> 'a -> 'b -> int = <fun>

let f g h x y = (h x)^(g y);;
☞ f : ('a -> string) -> ('b -> string) -> 'b -> 'a -> string = <fun>
```

12. Exercices d'évaluation d'applications de fonctions

Dans les exemples suivants, il faut prendre garde à la disposition des parenthèses qui délimitent véritablement des expressions évaluables, soit comme des entiers, soit comme des fonctions. On pourra, avec profit, refaire à la main ces évaluations.
Pour les faire à la main, il convient de commencer par l'évaluation des sous-expressions les plus profondes.

```
(fun x y -> x + y) (5 - 2) (7 - 5);; ☞ 5

(fun x y -> x * y) 3 7;; ☞ 21

(fun u v -> u + v) ((fun x y -> x + y) (5 - 2) (7 - 5)) ((fun x y -> x * y) 3 7);; ☞ 26

(fun x y z -> x y z) (fun u v -> u + v) ((fun x y -> x + y) (5 - 2) (7 - 5)) ((fun x y -> x * y) 3 7);;
☞ 26

(fun (x, y) -> y - x)  ((fun x y -> (x, y))  ((fun x y -> x +2 * y) 5 2)
                        ((fun x y z -> x + 2 * y + 3 * z) 5 4 3));; ☞ 13
```

13. Expressions avec variables locales

On dispose de deux structures locales équivalentes :

<let x = ... in ... résultat>

ou encore (mais plus en Ocaml) :

<résultat dépendant de x where x = expression>

Exemple de formulations équivalentes (répétons que "and" est ici un simple séparateur n'ayant rien à voir avec la conjonction &)

```
let b = "Bonjour " and c = "Christelle" in b ^ c;; ☞ string = "Bonjour Christelle"

b ^ c where b = "Bonjour " and c = "Christelle";; ☞ string = "Bonjour Christelle"

let x = 3 and y = 5 in let z = x * y and z' = y - x in z + z';; ☞ int = 17

let x = 3 and y = 5 in let z = x * y in z + z' where z' = y - x ;; ☞ int = 17
```

Mais attention, il est difficile et peu lisible de faire une cascade de "where" et cela n'est plus autorisé en Ocaml.

14. Directives utiles

#open "nom de module";;

Permet de charger les fonctions définies dans un fichier à part, en particulier :

#open "float";;

Permet d'utiliser les opérateurs + - * / sans point pour des réels, mais pas de mélanger entier et réel, ainsi 3. +. 2.5 sans cette directive, 3. + 2.5 avec, mais 3 + 2.5 sera encore refusé.

#close "float";; termine cette possibilité.

#open "graphics";; pour accéder aux instructions graphiques.

#include "nom de fichier";; charge les fonctions d'un fichier

Le plus commode, en fait, dans les versions PC et MAC ou UNIX est de copier depuis un traitement de texte quelconque.

#infix "opérateur";;

C'est une directive permettant d'utiliser une fonction binaire de façon infixe. (prefix op) permet d'utiliser un opérateur infixe op comme par exemple +, de façon préfixée. (En Ocaml, suivant la notation Haskell, l'opérateur op devient préfixe dès qu'il est encadré de parenthèses, ainsi (+) 3 4 retournera 7, il est même possible de définir let (op)..., puis d'utiliser op sans les parenthèses de manière infixe). Prenons deux petits exemples :

let **moyenne** a b = (a + b) / 2;; ☞ moyenne : int -> int -> int = <fun>

#infix "moyenne";;

Exemple 4 moyenne 6;; ☞ int = 5

let **double** = (prefix *) 2 ;; ☞ double : int -> int = <fun>

Exemple double 5;; ☞ 10

15. Trace des appels et entrées-sorties et séquence d'instructions

En reprenant la fonction factorielle, on utilise ensuite la fonction "trace" qui renvoie rien () mais dont l'effet de bord est d'afficher la suite des appels.

```
let rec fac x = if x = 0 then 1 else x*fac (x-1);;

trace "fac";; (* On ne met plus de guillemets en Ocaml *)
```

Cette dernière demande répondra :
"La fonction fac est dorénavant tracée. - : unit = ()"
et indiquera tous les appels de cette fonction, mais sans donner la valeur des paramètres avec lesquels elle est appelée.

```
fac 3;;
fac <-- 3
fac <-- 2
fac <-- 1
fac <-- 0
fac --> 1
fac --> 1
fac --> 2
fac --> 6
- : int = 6
```

Inversement, la commande "untrace "fac";;" provoque alors la réponse :
"La fonction fac n'est plus tracée. - : unit = ()"

Cependant, pour "débugger", l'usage est souvent de placer des "print ..." aux endroits cruciaux. C'est le procédé inavoué de tous les programmeurs.

On dispose en effet de procédures, malheureusement non génériques, toutes à un argument :

print_int,
print_float,
print_string,
print_char

Qui renvoient () de type "unit", et des procédures d'attente :

read_int (),
read_key ().

Dans la session qui suit le point-virgule est un séparateur d'instruction comme en Pascal ou en C, mais comme toute expression évaluée renvoie un résultat, le résultat renvoyé par I1; I2;; sera celui de I2. L'instruction préalable I1 ne sera donc utilisée généralement que pour un effet de bord tel un affichage. De toutes façons, bien programmer en fonctionnel, c'est composer des fonctions, plutôt que d'écrire des "instructions" en séquence. Avec la fonction "afficher" ci-dessous on pourrait être tenté de créer un affichage générique, mais le résultat n'est pas à proprement parlé lisible.

```
int_of_char `7`;; ☞ int = 55 (* donne le code ASCII *)

string_of_float 3.14;; ☞  string = "3.14"

float_of_string "3.14 et la suite";; ☞  float = 3.14

print_int 5;; ☞ 5 - : unit = ()
```

let x = 5 and c = "azerty" in print_int x; print_string c;; ☞ 5azerty - : unit = ()

print_char `A`; print_string "ZERT"; print_char `Y`;; ☞ AZERTY- : unit = ()

print_string "Le résultat est "; print_float (2.*.pi) where pi = 3.14;;
☞ Le résultat est 6.28 - : unit = ()

print_string ("Le résultat est " ^ (string_of_float (2.*.pi))) where pi = 3.14;;
☞ Le résultat est 6.28 - : unit = ()

read_int ();; 5 ☞ int = 5

let **afficher** c = print_string " Le résultat est "; c;; ☞ afficher : 'a -> 'a = <fun>

Exemple afficher 23;; ☞ Le résultat est - : int = 23

16. Exceptions

Il existe un type "exception" permettant par exemple :

exception divparzéro;; ☞ L'exception divparzéro est définie.

let **div** a b = if b = 0 then raise divparzéro else a / b;; ☞ div : int -> int -> int = <fun>

Exemples div 7 3;; ☞ int = 2
div 7 0;; ☞ Exception non rattrapée: divparzéro

let div a b = if b = 0 then failwith "Division par zéro !" else a/b;; ☞ div : int -> int -> int = <fun>

Exemple div 7 0;; ☞ Exception non rattrapée: Failure "Division par zéro !"

Mais plus simplement dans la seconde version, failwith "..." lève une exception localement définie et plus généralement la forme "try expr with ex1 -> f1 | ex2 -> f2 | ....." renvoie ce qui est précisé pour chaque exception ex, ou bien la valeur expr si elle est calculable. Dans la fonction "essayer" ci-dessous, il faut remarquer que son type est celui d'une fonction dont le premier paramètre est une fonction (a->b), dont le second a pour type une liste d'éléments de type a, et enfin dont le troisième paramètre est de type "exn" c'est à dire une exception.
Failure est un constructeur d'exception.

exception echec;;

let rec **essayer** f q e = match q with [] -> raise e | x :: r -> try f x with e -> essayer f r e;;
☞ essayer : ('a -> 'b) -> 'a list -> exn -> 'b = <fun>

Exemple essayer (fun x -> if x mod 2 = 0 then x else raise echec)
[1; 3; 5; 6; 8; 7; 5] echec;; ☞ 6

let rec **tenter** f q = match q with [] -> failwith "liste vide" | x :: r -> try f x with Failure _ -> tenter f r;;

Exemple tenter hd [[]; []; [4;2]; [3]; []];; ☞ int = 4

Bien programmer, c'est faire en sorte que les exceptions soient exceptionnelles !

CHAPITRE II

PREMIERES FONCTIONS

ET RECURSIVITE

Nous examinons dans ce chapitre des définitions de fonctions assez simples, c'est-à-dire plutôt des fonctions de nature mathématique, portant sur des entiers ou des réels.

1. Arithmétique, conversion et arrondi d'un réel

Les fonctions arithmétiques prédéfinies indispensables sont :
- Pour les entiers : succ, pred, abs, min, max, en notations préfixées,
  +, - *, / en infixe c'est à dire qu'on écrira "min 2 3" d'une part et "2 + 3" d'autre part.

Le reste de la division de m par n pour des entiers positifs est donné par m mod n.

- Pour les réels : abs_float, sqrt, sin, cos, tan, asin, atan, en radian, exp, log (népériens)
  +., -., *., /. et "power" ou **, pour les réels.

```
5 / 2;; ☞ int = 2
5. /. 2.;; ☞ float = 2.5
2.** 3.;; ☞ float = 8.0
power 2. 3.;; ☞ float = 8.0
```

Notons que "sqrt" désigne la racine carrée, mais la fonction carrée est à redéfinir par sqr x = x*x, soit pour des entiers, soit pour des réels.
Les conversions étant assurées par int_of_float et float_of_int, on peut alors définir :

```
let arrondi x = if x >=. 0. then int_of_float(x +. 0.5) else - (int_of_float (-. x +. 0.5));;
        ☞ arrondi : float -> int = <fun>
```

```
Exemples    arrondi 7.8;; ☞ int = 8
            arrondi (-2.1);; ☞ int = -2
            arrondi (-2.5);; ☞ int = -3
            int_of_float 3.14;; ☞ int = 3
            int_of_float 2.9999;;☞ int = 2
            int_of_float (-2.8);;☞ int = -2 (* oui, ce n'est pas la partie entière ! *)
            float_of_int 7;; ☞ float = 7.0
```

2. Relation de divisibilité

On veut définir le fait que l'entier p divise l'entier n ou non. L'égalité == est cependant aussi autorisée par le signe =. La relation de divisibilité est donc de type int -> int -> bool.

```
let divise p n = ((n/p)*p == n);;
        ☞ divise : int -> int -> bool = <fun>
```

```
Exemples     divise 4 12;; ☞ bool = true
             divise 4 7;; ☞ bool = false
```

Afin de dissocier deux choses qui n'ont rien à voir et qui sont malheureusement confondues dans beaucoup de langage de programmation, il faut rappeler que le signe = désigne ici la définition de l'objet "divise" et == désigne la relation d'égalité utilisée de manière infixe.

A ce propos, la directive "prefix" permet ensuite une utilisation du symbole d'égalité en préfixe.

```
(prefix ==);; ☞ 'a -> 'a -> bool = <fun>
Exemples     (prefix ==) 2 3;; ☞ bool = false
             (prefix ==) 4 4;; ☞ bool = true
```

3. Nombre de chiffres d'un entier

La fonction qui donne le nombre de chiffres d'un entier, par exemple 3 pour 421 et 5 pour 10000 se calcule grâce à int_of_float. Mais aussi récursivement et bien plus rapidement par la seconde écriture.
Toutes les fonctions qui sont définies récursivement, doivent, contrairement à la plupart des langages, le signaler à Caml par < let rec >.

```
let nbchif x = 1 + int_of_float ((log (float_of_int x)) /. log 10.);;
let rec nbchiff x = if x < 10 then 1 else 1 + nbchiff (x /10);;
```

```
nbchif 456789123;; ☞ int = 9
nbchiff 456789123;; ☞ int = 9
nbchiff 0;; ☞ int = 1
```

4. Définition récursive de factorielle

La factorielle de l'entier n définie comme le produit des entiers de 1 à n est donnée ici récursivement de deux façons, la première comme la définition d'un objet mathématique en disant que c'est la fonction qui ... et la seconde, plus familièrement en indiquant ce qu'est le résultat f(n) :

```
let rec fac = fun 0 -> 1 | n -> n*fac (n-1) ;;

let rec fac n = if n = 0 then 1 else n*fac (n - 1);;
```

```
Exemples     fac 7;; ☞ int = 5040
             fac 12;; ☞ int = 479001600
             fac 13;; ☞ int = -215430144
(* ce qui est normal, on tourne dans l'intervalle des entiers *)
```

5. Deux définitions mutuellement récursives sur la parité

Les fonctions pair et impair ("even" et "odd" dans beaucoup de langages, ne sont pas prédéfinies en Caml), mais il est facile de les définir ensemble par une récursivité mutuelle :

```
let rec pair n = if n = 0 then true else impair (n - 1)
and impair n = if n = 0 then false else pair (n - 1);;
```

```
Exemples    pair 5;;☞ bool = false
            impair 12;;☞ bool = false
            impair 7;;☞ bool = true
```

6. Exemple de la dérivation numérique

Il est assez facile de définir la dérivée approximative en un point de f comme un taux d'accroissement. Le point après une valeur numérique ou un opérateur arithmétique indique le type réel !

```
let deriv f dx = fun x -> (f(x +. dx) -. f(x)) /. dx ;;
        ☞ deriv : (float -> float) -> float -> float -> float = <fun>

let cube x = x *. x *. x;; ☞ cube : float -> float = <fun>
```

```
Exemple    deriv (cube, 0.0001) (1.);; ☞ float = 3.00030001
```

7. Exemple de la résolution d'une équation par la méthode de Newton

Pour résoudre une équation f(x) = 0 lorsqu'on connait l'existence d'une racine au voisinage de x_0, la méthode de Newton consiste à partir de cette valeur x_0 et à chaque pas, de définir le suivant x_{n+1} de x_n comme l'intersection avec l'axe des x, de la tangente à f en x_n.

```
let newton f x0 dx eps =
        let f' = deriv  f dx
        in let fin x = abs_float (f x) <. eps and suivant x = x -. (f x) /.(f' x) in loop fin suivant x0
        where rec loop test next x = if test (x) then x else loop test f (f x);;
        ☞ newton : (float -> float) -> float -> float -> float -> float = <fun>
```

Par exemple, la fonction 2cos(x) est nulle pour $\pi / 2$ (parmi d'autres racines) celle-ci sera atteinte avec le point de départ x = 1 radian par exemple.
Il est de toutes manières beaucoup plus simple de définir récursivement "newton" de la façon suivante :

```
let rec newton f x dx eps = if abs_float (f x) <. eps then x else newton f (x -. (f x) /.(f' x)) dx eps
        where f' = deriv f dx ;;
```

```
Exemple    2. *. newton cos 1. 0.001 0.0001;; ☞ float = 3.14159265362
```

8. Dichotomie avec variables locales

Le principe de la dichotomie, dans le but de resoudre f(x) = 0 sur un intervalle [a, b] où on sait que la fonction f s'annule une seule fois, est de diviser en deux cet intervalle au moyen du milieu c, et de reconnaître, grâce à la règle des signes, dans laquelle des deux moitiés, f va s'annuler.
Cette reconnaissance se faisant grâce aux signes contraires que peut prendre f aux extrémités de chaque intervalle ainsi découpé.
On donne une application au nombre d'or qui est la racine positive de $x^2 - x - 1 = 0$.

```
let rec dicho f a b eps = let c = (a +. b)/. 2. in if abs_float (b -. a) <. eps then c
                          else if (f a)*.(f c) < 0. then dicho f a c eps else dicho f c b eps;;
        ☞ dicho : (float -> float) -> float -> float -> float -> float = <fun>
```

```
Exemple    dicho (fun x -> (x*.x -. x -. 1.)) 0. 2. 0.001;; ☞ float = 1.61767578125
```

9. Combinaisons avec le filtrage explicite "match ... with"

La forme let f x y = match x y with ... -> ... | ... -> ... | .... est employée de manière équivalente à let f = fun ... -> ...|. Prenons par exemple le coefficient binômial défini de deux façons différentes. On utilise ici la bonne formule de récurrence $C_n^p = (n / p) C_{n-1}^{p-1}$

```
let rec comb n p = match (n, p) with    | (_, 0) -> 1
                                        | (0, _) -> 1
                                        | (n, p) -> (n * comb (n - 1) (p - 1)) / p;;
    ☞ comb : int -> int -> int = <fun>

let rec comb = fun    | _ 0 -> 1
                      | 0 _ -> 1
                      | n p  -> (n * comb (n - 1) (p - 1)) / p;;
    ☞ comb : int -> int -> int = <fun>
```

| Exemple   comb 7 2;; ☞ int = 21

Dans certaines versions Caml, la structure match c as (n, p) -> ... est autorisée ainsi que celle match ... with ... when ... -> ..., quoique plus compliquées, on pourrait donc écrire "comb" avec un seul paramètre qui sera un couple :

```
let rec comb c = match c with   (_, p) when p = 0 -> 1
                                | (n, _) when n = 0 -> 1
                                | (n, p) -> (n * (comb (n - 1, p - 1))) / p;;
    ☞ comb : int *int -> int = <fun>
```

On peut encore écrire des cas à la suite comme par exemple pour le nombre de jours d'un mois :

```
let nbjours n = match n with 2 -> 28 | 4 | 6 | 9 | 11 -> 30 | _ -> 31;;
```

| Exemples   nbjours 6;; ☞ int = 30
| nbjours 7;; ☞ int = 31

10. Calcul des termes de la suite de Syracuse

Partant d'un entier quelconque, chaque terme de cette suite est calculé comme la moitié du précédent si celui-ci est pair, ou sinon comme le successeur du triple. Pour ce genre de suites, on observe, sans l'avoir démontré, que cette suite parvient toujours à la période 1 - 4 - 2 - 1 ...

```
let rec suite n = print_int n; print_string "  "; if n <> 1 then suite (if pair n then n / 2 else 3 * n + 1)
        where pair p = (p = (p / 2) * 2) ;;
    ☞ suite : int -> unit = <fun>
```

Exemples
suite 7;; ☞ 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 unit = ()
suite 4526;; ☞ 4526 2263 6790 3395 10186 5093 15280 7640 3820 1910 955 2866 1433 4300 2150 1075 3226 1613 4840 2420 1210 605 1816 908 454 227 682 341 1024 512 256 128 64 32 16 8 4 2 1 unit = ()

11. Exemple de la puissance (en terme de composition) d'une fonction

Les fonctions $f^{(0)}(x) = x$,
$f^{(1)}(x) = x$, et pour tout entier n,
$f^{(n+1)}(x) = f(f^{(n)}(x))$
sont appelées itérées de f pour l'opération de composition.

Elles seront produites sans problème par la fonction "iterer" :

```
let rec iterer n f = fun x -> (if n = 0 then x else iterer (n - 1) f (f x));;
        ☞ iterer : int -> ('a -> 'a) -> 'a -> 'a = <fun>

let carre x = x*x;;
        ☞ carre : int -> int = <fun>

let fib n = fst (iterer n (fun (x, y) -> (y, x+y)) (1, 1));; (* est un bon exemple pour Fibonacci *)
        ☞ fib : int -> int = <fun>
```

```
Exemple     iterer 2 carre 3;; ☞ int = 81
Exemple     fib 6;; ☞ int = 13
```

12. Exemple de fonctionnelle sigma

Pour réaliser la fonction $\sum_{x = a..b} f(x)$ avec un pas de h, on définit "somme". Par exemple, si on veut calculer :

$$\sum_{x = 0..10} x^2 = 0^2 + 1^2 + 2^2 + ... +10^2 = 1 + 4 + 9 + ... + 81 + 100 = 385$$

```
let rec somme (f, a, b, h) = if a > b then 0 else f(a) + somme(f, a + h, b, h);;
        ☞ somme : (int -> int) * int * int * int -> int = <fun> (* version non currifiée *)

let carre x = x*x;; ☞ carre : int -> int = <fun>
```

```
Exemple     somme (carre, 0, 10, 1);; ☞ int = 385
```

Ou encore une meilleure variante currifiée (avec un pas de 1) en ce qui concerne la déclaration et donc le calcul du type :

```
let rec sigma f a b = if a > b then 0 else f a + sigma f(a + 1) b;;
        ☞ sigma : (int -> int) -> int -> int -> int = <fun>
```

```
Exemple     sigma ( fun x -> x*x) 0 10;; ☞ int = 385
```

On peut définir pour d'autres usages des boucles, et en ce cas redéfinir "iterer" d'une plus mauvaise manière à vrai dire :

```
let rec loop test c f x = if test c then x else loop test (c + 1) f (f x);;
        ☞ (int -> bool) -> int -> ('a -> 'a) -> 'a -> 'a = <fun>

let iterer n = loop ((prefix =)  n) 0;;
        ☞ int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

13. Intégrale par sommes de Darboux

Rappelons qu'une définition élémentaire de l'intégrale d'une fonction f entre a et b est la limite de la somme des aires algébriques des rectangles de hauteur f(x) déterminés par une subdivision de [a, b] de pas dx. Sans oublier que le point après une constante ou un opérateur arithmétique indique le type réel, prenons l'exemple de $4\int_{[0, 1]} 1/(1+x^2) = 4.\text{Atan } 1 = \pi$.

```
let rec integrale f a b dx = if a >. b then 0. else f(a) *. dx +. integrale f (a +. dx) b dx;;
        ☞ integrale : (float -> float) -> float -> float -> float = <fun>
```

```
Exemple     4.*.integrale (fun x -> 1. /. (1. +. x*.x)) 0. 1. 0.001;;
            ☞ float = 3.14259248692
```

14. Définition récursive terminale de factorielle

La récursivité terminale est établie dans le calcul d'une fonction f si et seulement si dans tout appel de f avec certaines valeurs, le résultat final sera celui d'un autre appel (avec d'autres valeurs). Cela signifie que le second appel ayant achevé son travail, le premier n'a pas à reprendre la main pour un travail quelconque supplémentaire comme par exemple la composition avec une autre fonction, il a "terminé".
L'intérêt de la récursivité terminale pour les langages qui la détectent, c'est que chaque sous-programme (un appel de fonction) n'a pas à conserver son contexte lorsqu'il est dérouté sur l'appel suivant puisqu'il a fini et que les valeurs des arguments qu'on lui a passé n'ont plus d'utilité. L'occupation de la mémoire est alors constante comme dans le cas d'un programme itératif.

Une programmation récursive terminale est visible à l'oeil nu dans :

f(x) = si <condition> alors <valeur> sinon f(...)

Elle est récursive non terminale dans :

f(x) = si ... alors ... sinon g(f(...))

En effet puisque le second appel de f doit être composé avec un traitement supplémentaire g qui ne pourra être fait, qu'une fois revenu au premier appel.
Dans la pratique, pour donner une écriture récursive terminale (ce n'est pas toujours possible), on prend tous les paramètres entrant en ligne de compte en cours de résolution du problème, comme fac' définie pour r (résultat partiellement construit) et n (entier courant descendant de la donnée initiale jusqu'à 1).

```
let fac = fac' 1 where rec fac' r = fun 0 -> r | n -> fac' (n *  r) (n - 1);;
        ☞  fac : int -> int = <fun>
```

| Exemple   fac 7 ;; ☞ 5040

La fonction fac' (à deux arguments) est manifestement récursive terminale, la fonction fac (à un argument) se contentant de l'appeler.

15. Définition récursive terminale de la suite de Fibonacci

Encore une fois, pour la plus simple des suites de Fibonacci définie par fib(0) = fib(1) = 1 et par récurrence fib(n+2) = fib(n+1) + fib(n) le calcul de chaque terme étant fonctions des deux précédents, il faut construire une fonction auxiliaire "fibis" pour ces deux paramètres nommés u et v, alors l'écriture de "fibis" devient un simple report à lui-même.
Le premier paramètre n étant pris en descendant, l'arrêt se fait pour n = 0 ou n = 1.
On aurait eu besoin d'un quatrième paramètre d'arrêt si on l'avait pris en montant.

```
let fib n = let rec fibis n u v = if n < 2 then v else fibis (n - 1) v (u + v) in fibis n 1 1;;
```

| Exemples   fib 6;; ☞ 13   fib 20;; ☞ 10946

16. Définition récursive terminale des combinaisons

La formule de Pascal conduit à de très nombreux calculs redondants, la bonne formule est celle déjà donnée. C'est à dire que si on s'y prend bien on peut conserver des entiers en provoquant dans ce calcul, qu'une alternance de multiplications et de divisions entières.

$$C_n^p = \frac{n}{p} C_{n-1}^{p-1} \text{ (quand c'est possible)} \quad C_n^p = C_{n-1}^p + C_{n-1}^{p-1} \text{ (formule de Pascal)}$$

On reprend l'utilisation d'une fonction auxiliaire "combis" qui aura naturellement pour arguments n et p mais encore le résultat r en cours de construction. C'est toujours ainsi qu'il faut voir les choses pour arriver à une bonne programmation.
Cependant, ici, si on utilisait des réels, on pourrait écrire :

```
let rec combis n p r = if p = 0. then r else combis (n -. 1.) (p -. 1.) (n *. r  /. p)
```

Mais en prenant les divisions, d'abord 2, puis 3 ... en terminant par p, on est sûr d'avoir des entiers, ce qui donne en nombres entiers :

```
let comb n p = let rec combis n p' r = if p' > p then r else combis (n - 1 ) (p' + 1) (n * r / p')
        in combis n 1 1;;
```

```
Exemples    comb 7 3;;☞ 35
            comb 8 2;;☞ 28
            comb 8 4;;☞ 70
            comb 8 5;;☞ 56
```

17. Intégrale par la formule de Simpson

Formule très efficace correspondant à des approximations paraboliques. La fonction "simpson" a pour paramètres la fonction f à intégrer, les bornes d'intégration a, b et l'entier n caractérisant la précision du découpage. Simpson utilise en fait la fonction récursive terminale simpson'.
Les exemples sont les valeurs approchées des intégrales $\int_{[0, 1]} x^2 = 1/3$
et de $4\int_{[0, 1]} 1/(1+x^2) = 4\text{Atan } 1 = \pi$.

```
let simpson f a b n = let h = (b -. a)/.(2.*. n) in (h /. 3.) *. (f(a) -. f(b) +. 2. *. simpson' 0. a)
        where rec simpson' s x = if x >=. b then s
                        else simpson' (s +. 2. *. f (x +. h) +. f (x +. 2.*.h)) (x +. 2.*.h);;
        ☞ simpson : (float -> float) -> float -> float -> float -> float = <fun>
```

```
Exemples    simpson (fun x -> (x*.x)) 0. 1. 2.;; ☞ float = 0.333333333333
            simpson (fun x -> 4./.(1.+.x*.x)) 0. 1. 20.;; ☞   float = 3.14159265358
```

18. Multiplication russe

On calcule le produit de deux entiers a*b de la manière suivante : on divise a par 2 tant que c'est possible, en doublant b, sinon on décrémente a et on ajoute b au résultat.
Il s'agit en fait de toujours prendre le quotient entier de a par 2 dans une colonne en doublant b dans une seconde colonne, puis le résultat sera la somme des nombres de la seconde colonne situés en regard d'un nombre impair dans la première.
La fonction "mult'" qui est récursive terminale, réalise cette suite d'opérations.

Exemple (en ligne) 26*54 = 13*108 = ... :

26	13	6	3	1	0	
54	108	216	432	864		sont additionnés 108 + 432 + 864 = 1404

```
let mult a b = mult' a b 0
        where rec mult' a b r = (* a*b + r est dit un invariant de boucle *)
                if a = 0 then r else if impair a then mult' (a - 1) b (r + b) else mult'(a / 2) (2 * b) r
        and impair x = x mod 2 = 1;;
```

```
Exemples    mult 26 54;; ☞ int = 1404
            mult 785 6235;; ☞ int = 4894475
```

19. Anniversaire

Quelle est la probabilité p(n) qu'au moins deux personnes dans un groupe de n personnes, aient leur anniversaire le même jour ? Sans tenir compte des années bissextiles, et sans utiliser la formule de Strirling $n! \sim n^n e^{-n}\sqrt{(2\pi n)}$, on considére l'événement contraire "chaque personne du groupe a une date d'anniversaire qui lui est spécifique". La probabilité de cet événement est le nombre d'injections de n personnes vers 365 jours, alors que le nombre de cas possibles est le nombre d'applications quelconques de n personnes vers 365 jours.
Pour n > 365, la probabilité est 1, sinon :

$$p_n = 1 - \frac{A_{365}^n}{365^n} = 1 - \frac{365*364* \dots (365 - n + 1)}{365^n}$$

```
let proba n = 1. -. (proba' (float_of_int (n -1 )) 1.)
        where rec proba' n r = if n = -1. then r else proba' (n -. 1.) (r *. (1. -.  (n /. 365.)));;
```

```
Exemples        proba 12;; ☞ float = 0.167024788838
                proba 58;; ☞ float = 0.991664979389
                proba 145;; ☞ float = 1.0 (* on a vite l'approximation 1 *)
                proba 368;; ☞ float = 1.0
```

20. Les tours de Hanoï

Fameux jeu où n plateaux de tailles échelonnées (le plus grand en dessous, le plus petit au dessus) sont placés à gauche de trois emplacements nommés "gauche", "milieu", "droite". En ne déplaçant toujours que le plateau du dessus d'une des pile pour le mettre sur un de taille plus grande dans une autre pile, on doit transporter l'ensemble sur la pile de droite.

```
let hanoi n = hanoi' n "gauche" "milieu" "droite"
        where rec hanoi' n d i a          (* d, i, a désignent départ, intermédiaire et arrivée *)
        = if n > 0 then (hanoi' (n-1) d a i;
                        print_string("transfert de " ^ d ^ " à " ^ a^ " \n");
                        hanoi' (n-1) i d a);;
```

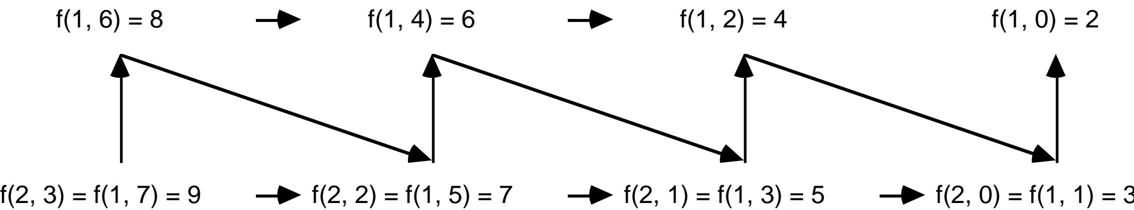
```
hanoi 3;; ☞
transfert de gauche à droite
transfert de gauche à milieu
transfert de droite à milieu
transfert de gauche à droite
transfert de gauche à droite
transfert de milieu à gauche
transfert de milieu à droite
transfert de gauche à droite
hanoi 4;; ☞
transfert de gauche à milieu
transfert de gauche à droite
transfert de milieu à droite
transfert de gauche à milieu
transfert de droite à gauche
transfert de droite à milieu
transfert de gauche à milieu
transfert de gauche à droite
transfert de milieu à droite
transfert de milieu à gauche
transfert de droite à gauche
transfert de milieu à droite
transfert de gauche à milieu
transfert de gauche à droite
transfert de milieu à droite
```

21. La fonction d'Ackerman

C'est un exemple de fonction récursive non primitive récursive (mais ceci n'est pas du tout à cause de son double appel car par exemple $f(n, 0) = 10^n$ et $f(n, m+1) = f(f(n, m), m)$ a un double appel et est primitive récursive).

```
let rec ak x y = if x = 0 then y + 1 else ak (x - 1) (if y = 0 then 1 else ak x (y - 1));;
```

| Exemple ak 2 3;; ☞ 9 (* dont l'arbre simplifié des appels est : *)



On déduit de la définition d'Ackerman certaines restrictions :

$f(0, n) = n + 1$ $f(1, n) = n + 2$ $f(2, n) = 2n + 3$ $f(3, n) = 2^{n+3} - 3$

Cette fonction d'Ackerman f peut se définir pas à pas sur l'ordre total lexicographique de N^2 car $f(n, 0) = f(n - 1, 1)$ et que $(n-1, 1) < (n, 0)$ pour cet ordre, de plus $f(n, m) = f(n - 1, f(n, m - 1))$ qui est déjà défini car $(n-1, f(n, m - 1)) < (n, m)$.
Si on connait $f(n, m)$ pour tous les (n, m) tels que $f(n, m) \leq v$ alors tous les (i, j) vérifiant $f(i, j) = v+1$ sont obtenus, ce qui permet de déduire une procédure itérative de calcul.

Remarque : la fonction d'Ackerman généralisée est :

$A(0, a, b) = a + 1$, $A(1, a, 0) = a$, $A(2, a, 0) = 0$, $A(3, a, 0) = 1$, et enfin :
si $n > 3$, $A(n, a, 0) = 2$ et $A(n, a, b) = A(n - 1, A(n, a, b - 1), a)$,

on peut vérifier de plus :
$A(1, a, b) = a + b$, $A(2, a, b) = ab$, $A(3, a, b) = a^b$, $A(4, a, b) = 2^{(a \text{ puiss } b)}$.

22. Point fixe et fonction de Mac Carthy

Si F est l'ensemble des fonctions partiellement définies sur D à valeurs dans D et si f est une fonction croissante (pour l'inclusion des graphes) de F dans F, le théorème de Knaster-Tarski indique que f admet un plus petit point fixe f c'est à dire tel que $f(f) = f$. Si de plus f est continue (vérifiant $f(\cup f_n) = \cup f(f_n)$ lorsque f_n est une suite croissante au sens de l'inclusion des graphes, alors ce point fixe est $\cup_{k \in N} f^k(\emptyset)$.

Par exemple "fac" est le plus petit point fixe de $f(f)(x)$ = si x = 0 alors 1 sinon $x.f(x-1)$ et la fonction de McCarthy ci-dessous est celui de $f(f)(x)$ = si x > 100 alors x-10 sinon $f^2(x+11)$, on a alors une suite de fonction partielles :

$f(\emptyset)(x)$ = si x > 100 alors x - 10,
d'où $f^2(\emptyset)(x)$ = si x > 100 alors x - 10 sinon si x > 99 alors x - 9,
et : $f^k(\emptyset)(x)$ = si x > 100 alors x - 10 sinon si x > 101 - k alors 91 pour $k \leq 11$ et cette suite devient stationnaire à $f^{20}(\emptyset)(x)$ = si x > 100 alors x - 10 sinon 91.

La fonction f_{91} de Mac Carthy est définie récursivement par :

```
let rec m x = if x > 100 then x - 10 else m (m (x + 11));;
```

La fonction généralisée de McCarthy est définie par :

mc(x, y) = si x > 100 alors si y = 1 alors x - 10 sinon mc (x - 10, y - 1)
sinon mc (x + 11, y + 1)
dans laquelle y compte les appels et f(x, y) = f_{91}(x)

```
let rec mc x y = if x > 100 then if y = 1 then x - 10 else mc (x-10) (y-1) else mc (x + 11) (y + 1);;
```

Exemple
mc 100 2;; ☞ 91
mc 123 5;; ☞ 91
mc 45 23;; ☞ 91

CHAPITRE III

TRAITEMENT DES LISTES

La liste chaînée est la structure de données fourre-tout la mieux adaptée à la récursivité car elle est représentée par un doublet d'adresses. La première pointant sur une valeur (la tête de la liste "hd") et la seconde sur la sous-liste qui suit (la queue "tl" c'est à dire également un doublet d'adresses. La liste vide [] joue donc toujours le rôle de dernier doublet. L'opérateur infixe double deux points :: ("cons" en Lisp, : en Haskell, . (le point) en ML et dans les anciens Lisp, enfin la barre | en Prolog) construit une liste en ajoutant le premier argument en tête du second qui doit être une liste.

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

Les autres fonctions commodes sur les listes sont :

- hd (le premier élément, List.hd en Ocaml),
- tl (la queue de la liste privée de son premier élément, List.tl en Ocaml),
- @ est la concaténation des listes (rappel ^ est celle des chaînes)
- enfin [] est la liste vide et le point virgule est le séparateur au sein d'une liste.
- rev est l'opération miroir.

```
hd [1; 2; 3 ];;
☞ int = 1

tl ["z"; "d"; "f"] ;;
☞ string list = ["d"; "f"]

[1; 2; 3; 4; 5; 6] @ [7; 8; 9];;
☞ int list = [1; 2; 3; 4; 5; 6; 7; 8; 9]

rev [1;2;3;4;5;6;7];;
☞  int list = [7; 6; 5; 4; 3; 2; 1]
```

Nous redéfinissons ci-dessous quelques fonctions de base à titre d'exercice, il faut bien entendu utiliser celles qui existent dans Caml.

```
let rec long = fun [] -> 0 | (a :: q) -> 1 + long q;; (* déjà prédéfinie list_length *)
        ☞   long : 'a list -> int = <fun>
let rec app x = fun  [] -> false | (y :: q) -> (x = y) or (app x q);;
        (* appartenance  déjà prédéfinie avec  mem ou List.mem en Ocaml*)
        ☞   app : 'a -> 'a list -> bool = <fun>
```

```
Exemples     app 1 [ 4; 2; 5; 1; 8];;
             ☞ bool = true
             app 1 [2; 5];;
             ☞ bool = false
```

```
let rec concat = fun [] q -> q | (a :: q) r -> a :: concat q r;;
    (* concaténation de deux listes prédéfinie de manière infixe avec le symbole @ *)
```

La fonction qui donne l'élément de rang n (à partir de 0) d'une liste est souvent commode, elle peut être définie avec un argument structuré, bien que l'interpréteur signalera que tous les cas ne sont pas considéré.

```
let rec nth (a::q) = fun 0 -> a | n -> nth q (n - 1);; (* n-ième élément d'une liste à partir de 0*)

let rec tete q n = if q = [] or n = 0 then [] else (hd q):: (tete (tl q) (n - 1));;
    (* les n premiers éléments de q*)

let rec queue q n = if q = [] or n = 0 then q else queue (tl q) (n - 1);;
    (* tout sauf les n premiers éléments de q *)
```

Deux prédicats quantificateurs "exists" et "for_all" commodes pour les listes :

```
exists (fun x -> (x=2)) [1;8;2;6];;      ☞ bool = true
exists (fun x -> (x=2)) [1;6;9];;        ☞ bool = false
for_all (fun x -> (x=2)) [2;9];;         ☞ bool = false
for_all (fun x -> (x=2)) [2;2];;         ☞ bool = true
```

Signalons encore la fonction "union" éliminant les répétitions dans le premier argument :

```
union [1;2;3;3;4] [2;3;4;5;5];;
☞ int list = [1; 2; 3; 4; 5; 5]
```

L'association "assoc" donne la première ordonnée possible pour une abscisse et une liste de couple en second argument, cette fonction, facile à redéfinir existe en Ocaml sous le nom de "List.assoc" :

```
assoc 2 [(1, 5) ;(5, 6); (8, 2); (2, 7); (0, 4); (2, 5)];;
☞ int = 7
```

1. Aplatissement d'une liste

L'aplatissement est en quelque sorte le retrait de toutes les parenthèses intérieures au sein d'une liste ("List.flatten" en Ocaml) .

```
let rec aplatir = fun [] -> [] | ([] :: q) -> aplatir q | ((a :: q) :: r) -> a :: aplatir (q :: r);;
    ☞ aplatir : 'a list list -> 'a list = <fun>

aplatir [[1; 2]; [3; 4; 5]; [6; 7]; [8]];;
☞ int list = [1; 2; 3; 4; 5; 6; 7; 8]
```

2. Retrait d'un élément dans une liste

```
let rec ret x = fun [] -> [] | (a :: q) -> (if x = a then q else a :: (ret x q));;
    ☞ ret : 'a -> 'a list -> 'a list = <fun>

ret 4 [2; 3; 5; 4; 7; 5; 4; 8; 4];;
☞ [2; 3; 5; 7; 5; 4; 8; 4]
```

Pour le retrait de toutes les occurrences :

```
let rec retout x = fun [] -> [] | (a :: q) -> (if x = a then (retout x q) else a :: (retout x q));;
        ☞ retout : 'a -> 'a list -> 'a list = <fun>

retout 4 [2; 3; 5; 4; 7; 5; 4; 8; 4];; ☞ [2; 3; 5; 7; 5; 8]
```

3. Somme termes à termes

Etant donné deux listes, on souhaite obtenir la liste des sommes des éléments de même rang sur la plus courte longueur.

```
let rec som = fun (a::q) (b::r) -> (a + b) :: (som q r) | [] _ -> [] | _ [] -> [];;

        ☞ som : int list -> int list -> int list = <fun>

som [1; 2; 3] [3; 4; 5; 6];;
☞ int list = [4; 6; 8]
```

4. Transformation d'une liste de couple en couple de listes

Soient deux listes de couples, on forme de façon récursive terminale, le couple formé par la liste m de toutes les abscisses et celle n des ordonnées.

```
let split lc = split' [] [] lc (* prédéfinie mais renvoie les éléments à l'endroit *)
        where rec split' m n = fun [] -> m, n | ((x, y)::r) -> split' (x::m) (y::n) r;;

        ☞  split : ('a * 'b) list -> 'a list * 'b list = <fun>

split [(1, "a"); (2, "z"); (3, "e"); (4, "r"); (5, "t"); (6, "y")];;
☞  int list * string list = [6; 5; 4; 3; 2; 1], ["y"; "t"; "r"; "e"; "z"; "a"]
```

5. Elément de rang donné et listes infinies

Dans la définition suivante, on a deux listes infinies x et y (qu'on ne peut donc afficher) mais que l'on peut consulter grâce à la fonction "nth" qui renvoie l'élément de rang donné (à partir de 0) d'une liste.

```
let rec nth = fun 0 (x :: q) -> x | n (x :: q) -> nth (n - 1) q;;

let rec x = 1::y and y = 2::x;;
x : int list = [1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1;
2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1;
2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; ...

y : int list = [2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2;
1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2;
1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; ...

nth 3 [5; 9; 4; 7; 2];; ☞ 7
nth 35 x;; ☞ 2
nth 148 x;; ☞ 1
nth 149 x;; ☞ 2
nth 148 y;; ☞ 2
nth 149 y;; ☞ 1
```

6. Crible d'Eratosthène

On construit d'abord une fonction "intervalle" délivrant la liste des entiers consécutifs de a à b.

```
let rec interval a b = if a > b then [] else a :: interval (succ a) b;;
```

```
interval 2 17;; ☞ int list = [2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17]
```

Puis une fonction "filtre" qui servira en bien des occasions, une fonction très générale délivrant les éléments d'une liste satisfaisant une propriété "test", et enfin la fonction "retire-multiples".

```
let rec filtre test = fun [] -> [] | (a::r) -> if test a then a :: (filtre test r) else filtre test r;;
```

```
filtre (fun x-> (x = 4)) [4; 1; 5; 2; 4; 6];;                ☞ int list = [4; 4]
filtre (fun x -> (x  = 5 * (x / 5))) [0; 2; 5; 7; 10; 15; 17];;   ☞ int list = [0; 5; 10; 15]
```

```
let retmultiples n = filtre (fun m -> m mod n <> 0);;
```

```
retmultiples 4 [1; 2; 4; 5; 8; 7; 12; 13; 16; 18];;
☞ int list = [1; 2; 5; 7; 13; 18]
```

```
let crible m = crible' (interval 2 m) where rec crible' = fun [] -> []
                    |(n::R) -> if n*n > m then (n::R) else n :: crible' (retmultiples n R) ;;
```

```
crible 100;;
☞ int list = [2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71; 73;
79; 83; 89; 97]
```

Remarque Haskell permet des définitions construites à partir d'une propriété ainsi :
[x*x | x <- [1; 2; 3]] pour [1; 4; 9] ou encore [sqrt x | x <- [1 ..]] pour la liste "infinie" des racines carrées d'entiers, ou bien [(x, y) | x <- [1, 2, 3], y <- ['a', 'b']] pour 6 couples int*char. On a alors, dans ce langage, les définitions équivalentes :

```
depuis n = n : depuis (n + 1)
retirmultiples d (n:q) = if mod n d == 0 then lasuite else n : lasuite
                    where lasuite = retirmultiples d q
crible (x:q) = x:crible (retirmultiples x q)
nbpremiers = crible (depuis 2)
```

```
take 20 nbpremiers ☞ [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
```

Attention la définition "premier n = elem n nbpremiers" est possible mais ne répondra jamais False, il faut jouer sur l'ordre en définissant plutôt :

```
premier n = elem n (jusqua n nbpremiers)
          where jusqua n (p : q) = if p > n then [] else p : (jusqua n q)
```

```
premier 53 ☞ True
premier 54 ☞ False
```

7. Tri par insertion

Les trois méthodes de tri qui suivent considèrent des listes d'entiers à trier dans l'ordre croissant. L'extension à un ordre quelconque sera vu après. La méthode de tri par insertion consiste à prendre le premier élément de la liste, et à l'insérer à sa place dans le reste de la liste préalablement trié.

```
let rec insere (x, q) = if q = [] then [x] else if x < hd q then x :: q else hd q:: insere(x, tl q) ;;
```

```
insere (4, [-3; -1; 0; 3; 6; 8; 9]) ;;
☞ int list = [-3; -1; 0; 3; 4; 6; 8; 9]
```

```
let rec ins x = fun [] -> [x] | (a :: q) -> (if x < a then x :: a :: q else a:: (ins x q));;
        (* autre typage pour l'insertion *)
let rec trins q = if tl q = [] then q else insere ((hd q),  trins (tl q));;
```

```
trins([4; 8; -5; 2; -3; 0; 4; -9; 7]);;
☞ int list = [-9; -5; -3; 0; 2; 4; 4; 7; 8]
```

8. Tri par fusion

Ce tri (le meilleur de tous) consiste à décomposer la liste à trier en deux parties (ce peut être deux moitiés comme le début et la fin, ou mieux comme ci-dessous les éléments de rangs pairs ou impairs lp et li), à trier séparément ces deux parties puis à les fusionner. On peut démontrer que le temps d'exécution de ce tri de n données, est toujours de l'ordre de n.ln(n), alors qu'il est du même ordre pour le tri par segmentation, mais seulement en moyenne, et de l'ordre de n^2 pour les tris par insertion ou extraction.

```
let rec fusion  =  fun | [] q -> q
                | q  [] -> q
                | (a :: q) (b :: m) -> if a < b then a :: fusion q (b :: m) else b :: fusion (a :: q) m;;
```

```
fusion [1; 3; 5; 6] [0; 2; 4; 6; 8; 9];;
☞ int list = [0; 1; 2; 3; 4; 5; 6; 6; 8; 9]
```

```
let rec decomp lp li pair = fun
                | [] -> (lp, li)
                | (a :: r) -> if pair then decomp (a :: lp) li false r else decomp lp (a :: li) true r;;
    ☞ decomp : 'a list -> 'a list -> bool -> 'a list -> 'a list * 'a list = <fun>
```

```
decomp [] [] true [0; 1; 2; 3; 4; 5; 6; 7] ;; ☞ int list * int list = [6; 4; 2; 0], [7; 5; 3; 1]
```

```
let rec tri q = if list_length q < 2 then q else fusion (tri lp, tri li) where (lp, li) = decomp [] [] true q;;
    ☞   tri : int list -> int list = <fun>
```

```
tri [4; 8; -5; 2; -3; 0; 4; -9; 7; 1] ;; ☞ int list = [-9; -5; -3; 0; 1; 2; 4; 4; 7; 8]
```

9. Tri par segmentation

Le principe de ce tri consiste à prendre un élément dit "pivot", et à parcourir la liste en mettant de côté dans deux listes g et d les éléments respectivement plus petits et plus grands que le pivot. La valeur du pivot choisi est simplement la tête hd(q), sa position est déterminée par le résultat de "partition" . lorsque les deux listes sont ainsi constituées, on les trie suivant le même algorithme et on concatène le tout.

```
let partition q = partition' (hd q) [] [] (tl q)
(* range dans g et d ceux qui sont plus petits ou plus grands que p *)
        where rec partition' p g d = fun [] -> g, d
                        | (a :: r) -> if a < p then partition' p (a::g) d r else partition' p g (a :: d) r;;
```

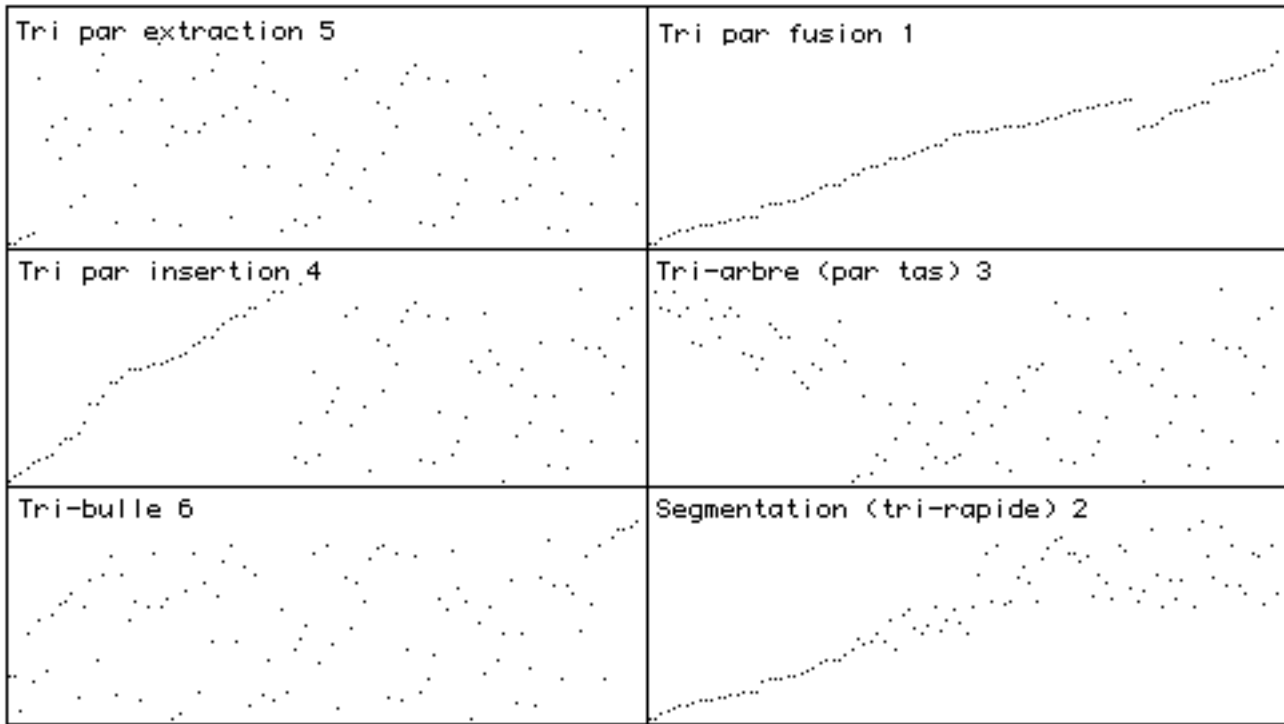
```
let rec triseg q = if list_length q< 2
                        then q
                        else let l1, l2 = partition q in (triseg l1) @ [hd  q] @ (triseg l2) ;;
```

```
triseg [4; 8; -5; 2; -3; 0; 4; -9; 7; 1];; ☞   int list = [-9; -5; -3; 0; 1; 2; 4; 4; 7; 8]
triseg ["marie"; "christelle"; "verra"; "anne"; "tauty"; "sophie"];;
☞   string list = ["anne"; "christelle"; "marie"; "sophie"; "tauty"; "verra"]
```

Remarque
Il est intéressant de donner le même programme en Haskell (où ++ est la concaténation des listes, : le "cons" et <- est l'appartenance), il faut noter la définition par cas où l'argument est structuré (le "match" est donc sous-entendu) analogue aux clauses du Prolog :

```
triseg [] = []
triseg (x : q) = triseg g ++ [x] ++ triseg d          where   g = [y | y <- q, y < x]
                                                              d = [y | y <- q, y >= x]
```

L'illustration suivante sur 100 points choisis aléatoirement entre 0 et 50, un programme de démonstration fourni avec Caml, passe la main à six procédures de tri pour les mêmes temps. Le tri par fusion a presque fini alors que le tri-bulle n'a repoussé que quelques-uns des plus grands éléments et le tri par extraction n'a trouvé que quelques uns des plus petits.



Si maintenant, on peut introduire un "ordre" comme paramètre dans le tri et la partition, en ce cas l'appel serait triseg (prefix <), ce qui donne, avec un peu de variantes :

```
let partition p q rel = partition' p [] [] q
        (* range dans g et d ceux qui sont plus petits ou plus grands que p *)
        where rec partition' p g d = fun  [] -> g, d
                        | (a :: r) -> if rel a p then partition' p (a :: g) d r else partition' p g (a :: d) r;;

let rec triseg rel = fun   [] -> []
                | [x] -> [x]
                | (x :: r) -> (triseg rel l1) @ (x :: (triseg rel l2)) where l1, l2 = partition x r rel;;
```

```
triseg (prefix <) [4; 8; -5; 2; -3; 0; 4; -9; 7; 1] ;;
☞ [-9; -5; -3; 0; 1; 2; 4; 4; 7; 8]

triseg (fun x y -> y mod x = 0) [48; 24; 6; 3; 12; 1] ;;
☞ [1; 3; 6; 12; 24; 48] (* la divisibilité est ici une relation totale *)

triseg (prefix <) ["eva"; "julie"; "adea"; "eve"; "anna"; "paola"; "anne"; "claire"; "aida"];;
☞ ["adea"; "aida"; "anna"; "anne"; "claire"; "eva"; "eve"; "julie"; "paola"]
```

10. Définition récursive terminale de l'inversion

C'est l'opération mirroir qui est d'ailleurs prédéfinie sous le nom de "rev". Invbis est à deux arguments le résultat r, et les données d.

```
let inv q = invbis [] q where rec invbis r = fun [] -> r | (a :: d) -> invbis (a :: r) d;;
```

```
inv [0; 1; 2; 3; 4; 5; 6; 7; 8; 9];;
☞ int list = [9; 8; 7; 6; 5; 4; 3; 2; 1; 0]
```

11. Petite fonction de Ramanujan

Soit dec(n) le nombre de décompositions en entiers d'un entier n, par exemple 1 + 1 + 1 + 1 + 1 = 1 + 1 + 1 + 2 = 1 + 2 + 2 = 1 + 1+ 3 = 1 + 4 = 2 + 3 = 5, d'où dec(5) = 7.
Le célèbre Ramanujan (1887-1920) qui a reconnu 1729 comme premier entier décomposable de 2 façons en 2 cubes $1^3 + 12^3 = 9^3 + 10^3 = 1729$, a donné en 1917 la formule asymptotique dec(n) $\sim (1/4\pi\sqrt{3})\exp(\pi\sqrt{(2\pi/3)})$ déjà bien vérifiée pour dec(200) = 3 972 999 029 388.
Il peut être calculé par récurrence en écrivant que les décompositions de n avec les valeurs x et celles d'une liste q sont celles avec q augmentées de celles de n-x, mais avec les valeurs de q auxquelles on ajoute x à chaque fois x :

```
let rec compte k = if k = 0 then [] else [k] @ (compte (k - 1));;

let dec n = decbis n (compte n)
        where rec decbis n = fun
        | [] -> 0
        | (x :: q) -> (decbis n q) + (if x < n then decbis (n - x) (x :: q) else if x = n then 1 else 0);;
```

```
compte 15;;☞ [15; 14; 13; 12; 11; 10; 9; 8; 7; 6; 5; 4; 3; 2; 1]
dec 5;; ☞ 7
dec 10;; ☞ 42
dec 12;; ☞ 77
```

Maintenant, pour avoir ces décompositions sous les yeux et non plus seulement leur nombre, on peut suivre le même modèle :

```
let voirdec n = decbis n (compte n)
        where rec decbis n = fun [] -> []
        | (x :: q) -> (decbis n q) @ (if x < n then map (fun s -> x :: s) (decbis (n - x) (x :: q))
                                else if x = n then [[n]] else []);;
```

```
voirdec 5;;
☞ [[1; 1; 1; 1; 1]; [2; 1; 1; 1]; [2; 2; 1]; [3; 1; 1]; [3; 2]; [4; 1]; [5]]
```

Les décompositions sans répétitions peuvent se faire indépendamment par ce programme de L.Chéno, mais on pourra le faire également grâce au problème du "compte est bon" plus loin.

```
let partition n = let cons a b = a::b in
        let rec aux n k =
                if k > n then aux n n
                else if n = 0 & k = 0 then [[]]
                else if k = 0 then []
                else it_list (fun l j -> map (cons j) (aux (n - ) (j - 1))@ l) [] (compte k)
        in aux n n;;
```

```
partition 7;;
☞ [[4; 2; 1]; [4; 3]; [5; 2]; [6; 1]; [7]]
```

12. Suite des fourmis

Soit, d'après B.Werber, la suite débutant par 1 et telle que chaque terme soit l'énumération du terme précédent, énumération constituée simplement par le nombre de fois qu'un chiffre se trouve plusieurs fois écrit consécutivement.
Ainsi à l'étape 111221, la lecture de gauche à droite de ce "mot" sera 3 fois le chiffre 1 puis 2 fois le chiffre 2 et enfin 1 fois le chiffre 1. L'étape suivante sera donc constituée par 312211. On constate qu'il est posible d'obtenir 33 ou 222 dans la chaîne, mais 4 ?
Nous programmons cette suite au moyen de deux fonctions mutuellement récursives "suite" et "suitbis". Malheureusement le typage introduit beaucoup de difficultés pour le moindre affichage.

```
let rec suite = fun [] -> [] | (x :: q) -> suitbis 1 x q
and suitbis n x = fun [] -> [n; x]
                    | (y :: q) -> if x = y then suitbis (n + 1) x q else n :: x :: (suite (y :: q));;
let rec aff = fun [] -> print_newline() | (x :: q) -> print_int x; aff q;;
let rec essai e = fun 0 -> aff e | n -> aff e; essai (suite e) (n - 1);;
```

```
essai [1] 10;; ☞
1
11
21
1211
111221
312211
13112221
1113213211
31131211131221
13211311123113112211
11131221133112132113212221 - : unit = ()
```

Remarque A titre de comparaison, le même programme en Lisp :

```
(defun suite (L) (if (null L) L (suitbis1 (car L) (cdr L))))
(defun suitbis (N X L) (cond
        ((null L) (list N X))
        ((eq (car L) X) (suitbis (1+ N) X (cdr L)))
        (t (mcons N X (suite L)))))
(defun essai (k) (print (set 'e '(1))) (repeat k (print (set 'e (suite e)) )))
```

```
(essai 5) ☞
(1)
(1 1)
(2 1)
(1 2 1 1)
(1 1 1 2 2 1)
(3 1 2 2 1 1)
```

Exemple

Si on a 18 occurrences consécutives du chiffres 5, les valeurs 18 et 5 ne disparaitront plus. L'option qui consiste à fractionner les entiers 18 en [1; 8] est facile, il suffit de modifier "suitbis".

```
let rec decomp n q = if n < 10 then n :: q else decomp (n / 10) ((n mod 10) :: q);;
```

```
decomp 231 [4];; ☞ [2; 3; 1; 4]
```

```
let rec suite = fun [] -> [] | (x :: q) -> suitbis 1 x q
and suitbis n x = fun [] -> decomp n [x]
                    (y :: q) -> if x=y then suitbis (n + 1) x q else n :: x  :: (suite (y :: q));;
```

```
essai [2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 4; 4; 2; 3; 3; 2; 2; 1; 1; 1; 4; 4; 4; 4; 4; 4;
4; 4; 4; 4; 4; 4; 4; 4; 2; 2; 0; 0; 0; 0] 5;; ☞

22222222222222222442332211144444444444444220000
17224122322311442240
117221411221322131111414221410
1111722111421221113221113311141114221114111O
31117223114121122311322312321114311422311431110
13211172213211411122122132113221311121312211141321142213211413211O - : unit
= ()
```

13. Fonctionnelles sur les listes

Outre "filtre" déjà vu à l'occasion du crible d'Eratosthène, le fameux "mapcar" du Lisp, map (qui est prédéfini en "List.map" en Ocaml) est une fonction qui applique son premier argument f (devant être une fonction à un argument) sur son second argument devant être une liste d'éléments du domaine de f. Par exemple map succ [1; 2; 3] donne [2; 3; 4]. Il peut être redefini de trois façons (attention parenthèses indispensables pour la fonction anonyme) :

```
let rec map f q = if q = [] then [] else f (hd q) :: (map f (tl q)) ;;
        ☞  map : ('a -> 'b) -> 'a list -> 'b list = <fun>

let rec map f = fun [] -> [] | (a :: q) -> f(a) :: (map f q);;
```

Un autre typage possible pour map

```
let rec map (f, q) = if q = [] then [] else f(hd q) :: map (f, tl q);;
        ☞  map : ('a -> 'b) * 'a list -> 'b list = <fun>

map (hd, [[4; 5]; [2; 7; 3]; [1; 2; 3; 4]]);;      ☞ int list = [4; 2; 1]
map ((fun x -> x*x), [1; 2; 3; 4; 5]);;             ☞ int list = [1; 4; 9; 16; 25]
```

14. Une application à l'ensemble des parties d'un ensemble

On désire, en représentant les ensembles par des listes, obtenir l'ensemble des parties, c'est à dire ici, la liste des sous-listes, il y aura donc la liste vide et la liste complète comme cas particuliers, et les éventuelles répétitions seront prises en compte.

```
let rec ttparties = fun [] -> [[]] | (a :: q) -> let p = ttparties q and f = fun x -> a :: x  in p @ (map f p);;

ttparties [1; 2; 3];; ☞ int list list = [[]; [3]; [2]; [2; 3]; [1]; [1; 3]; [1; 2]; [1; 2; 3]]
```

On peut définir récursivement à l'aide de "map" la fonction "parties p m" renvoyant les parties constituées de p éléments de m :

```
let rec parties p = fun [] -> if p = 0 then [[]] else []
                        | (a :: q) -> (parties p q) @ (map (fun x-> a :: x) (parties (p - 1) q));;

parties 2 [1; 2; 3] -> [[1; 2]; [1; 3]; [2; 3]] (* les paires *)
```

Avec la fonction déjà vue "filtre" à deux arguments test et m renvoyant la liste des éléments de m vérifiant le test. Par exemple si "pair" a été défini, filtre pair [2;7;5;3;2;4;2] -> [2;2;4;2], ou encore filtre (fun x->(x=0)) [1;2] -> []

```
let rec filtre test = fun   | [] -> []
                            | (a :: q) -> if test a then a :: (filtre test q) else filtre test q;;
```

On peut redéfinir "parties" à l'aide de "ttparties", de "filtre", et de "list_length" prédéfinie. Mais cela consiste à d'abord construire toutes les parties, pour filtrer celles qui ont exactement p éléments, ce qui est bien entendu assez maladroit.

```
let rec parties p m = filtre (fun x -> (list_length x = p)) (ttparties m);;
```

15. Application aux permutations

La fonction "permut" pour une liste q doit fournir la liste des permutations de q, perm q x fournit la liste des permutations de la liste q privée de x, auxquelles x est rajouté en tête, ce qui permet une programmation élégante mutuellement récursive :

```
let rec ret x = fun [] -> [] | (a :: q) -> (if x = a then q else a::(ret x q));;

let rec aplatir = fun [] -> [] | ([] :: q) -> aplatir q | ((a :: q)::q') -> a :: aplatir (q :: q');; (* déjà vue *)

let rec permut = fun [] -> [[]] | q -> aplatir (map (perm q) q)
and perm q x = map (fun z -> x :: z) (permut (ret x q));;
```

Mais cela peut aussi s'écrire très curieusement :

```
let rec permut = let perm q x = map (fun z -> x :: z) (permut (ret x q))
                 in fun   | [] -> [[]]
                          | q -> aplatir (map (perm q) q);;
```

```
permut [1; 2; 3];; ☞ int list list = [[1; 2; 3]; [1; 3; 2]; [2; 1; 3]; [2; 3; 1]; [3; 1; 2]; [3; 2; 1]]
```

Pour aller au delà, on va réaliser une visualisation les 120 permutations (pour des entiers) :

```
let vue x = map print_int x; print_string " ";;
```

```
map vue (permut [1;2;3;4;5]);; ☞ 12345 21345 13245 31245 23145 32145 12435 21435
14235 41235 24135 42135 13425 31425 14325 41325 34125 43125 23415 32415 24315
42315 34215 43215 12354 21354 13254 31254 23154 32154 12534 21534 15234 51234
25134 52134 13524 31524 15324 51324 35124 53124 23514 32514 25314 52314 35214
53214 12453 21453 14253 41253 24153 42153 12543 21543 15243 51243 25143 52143
14523 41523 15423 51423 45123 54123 24513 42513 25413 52413 45213 54213 13452
31452 14352 41352 34152 43152 13542 31542 15342 51342 35142 53142 14532 41532
15432 51432 45132 54132 34512 43512 35412 53412 45312 54312 23451 32451 24351
42351 34251 43251 23541 32541 25341 52341 35241 53241 24531 42531 25431 52431
45231 54231 34521 43521 35421 53421 45321 54321
```

Une façon assez analogue, mais plus intuitive :

```
let rec permut = fun []-> [[]] |(a :: q) -> let p = permut q in aplatir  (map (fun x -> distrib a [] [] x) p)
         where rec distrib a r g = fun      [] -> (g @ [a])::r
                                            | (x :: d) -> distrib a ((g @ [a; x] @ d) :: r) (g @ [x]) d;;
```

Remarque

Les possibilités d'Haskell d'écrire [x | x <- e \\ f] pour " l'ensemble des x tels que x appartienne à la différence ensembliste e - f " permettent de définir les permutations de l'ensemble e comme toutes celles de e - {x}, précédées de x, et ceci pour chaque x de e :

```
perm [] = [[]]
perm e  = [x : p | x <- e, p <- perm (e \\ [x]) ]
```

16. Polynômes représentés par la liste ascendante des coefficients

On représente ainsi $3 + x - x^2$ par [3; 1; -1] et $2 + 5x^2 + x^3$ par [2; 0; 5; 1]. On peut alors définir très simplement l'addition de polynômes, la multiplication par un scalaire, le produit par le polynôme particulier X, et enfin le produit des polynômes.

```
let rec add = fun [] p -> p | p [] -> p | (a :: p) (b :: q) -> (a + b) :: add p q;; (* Addition *)

let rec muls c = fun [] -> [] | (a :: q) -> c * a :: (muls c q);; (* Multiplication par un scalaire*)

let mulv p = 0 :: p;; (* Multiplication par X *)

let rec mul p = fun [] -> [] | (a :: q) -> add (muls a p) (mul (mulv p) q);; (* multipl. polynômiale *)

let deg p = let rec deg' p ac = match p with [] -> 0
                                        | (0::q) -> deg' q (ac + 1)
                                        | (x :: q) -> ac + (deg' q 1)
        in deg' p 0;; (* Degré d'un polynôme *)
```

```
add [3; 1; -1] [2; 0; 5; 1];;     ☞ [5; 1; 4; 1]
mul [3; 1; -1] [2; 0; 5; 1];;     ☞ [6; 2; 13; 8; -4; -1]
deg [0;1;2;3;4;0;0;0];;           ☞ 4
deg [0;1;2;0;0;0;6];;             ☞ 6
```

17. Application d'une fonction et applatissement, exemple du langage engendré par un alphabet

En anticipant sur le chapitre des chaînes de caractères, la fonctionnelle flat_map réalise en plus de "map", la concaténation des sous-listes produites, par exemple :

```
map (fun m -> [m]) ["a"; "b"];; ☞ [["a"]; ["b"]]
flat_map (fun m -> [m]) ["a"; "b"];; ☞ ["a"; "b"]
```

On veut produire tous les mots de longueur n construits à partir d'un alphabet, la liste a :

```
let rec mots n a = if n = 0 then [""]
        else let res = mots (n - 1) a in flat_map (fun m -> (map (fun s -> s^m) a)) res;;
```

```
mots 3 ["a"; "b"];;
☞ ["aaa"; "baa"; "aba"; "bba"; "aab"; "bab"; "abb"; "bbb"]
mots 3 ["a"; "b"; "c"];;
☞ ["aaa"; "baa"; "caa"; "aba"; "bba"; "cba"; "aca"; "bca"; "cca"; "aab"; "bab"; "cab";
"abb"; "bbb"; "cbb"; "acb"; "bcb"; "ccb"; "aac"; "bac"; "cac"; "abc"; "bbc"; "cbc"; "acc"; "bcc";
"ccc"]
```

18. Les fonctionnelles list_it et it_list

Le "apply" du Lisp existe en Caml sous le nom de it_list, il consiste à appliquer une fonction binaire f (associative ou non) d'élément neutre (ou non) e sur tous les éléments d'une liste. Elle pourrait être redéfinie par :

```
let rec apply f e = fun [] -> e | (a :: q) -> f a (apply f e q);;
        ☞ apply : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

```
apply max 0 [4; 8; 5; 2; 6; 4];;     ☞ int = 8
apply (prefix +) 0 [1; 2; 3; 4];;    ☞ int = 10
```

La fonction prédéfinie list_it prend les éléments de la liste du dernier au premier (par la droite), alors que it_list les prend du premier au dernier (par la gauche). "list_it" peut être redéfini par :

```
it_list f e ll = match ll with [] -> e | (x :: q) -> f x (it_list f e q);;
```

```
list_it (fun x y -> x*y) [1;2;3;4;5] 1;;        ☞ 120
it_list (fun x y -> x*y) 1 [1;2;3;4;5];;        ☞ 120
```

Ces fonctions prédéfinie existent en Haskell sous le nom de <foldl opérateur base liste> qui renvoie l'itération de l'opérateur depuis la base avec les éléments de gauche à droite de la liste et <foldr opérateur base liste> qui renvoie l'itération de gauche à droite sur la liste, en terminant par la base. Noms qui sont repris en List.fold_left et List.fold_right en Ocaml.
Ainsi fold_left f e [a; b; c] -> f (f (f e a) b) c)

CHAPITRE IV

PARCOURS D'ARBRES EN PROFONDEUR

1. Décomposition d'un nombre grâce à une liste donnée (le compte est bon)

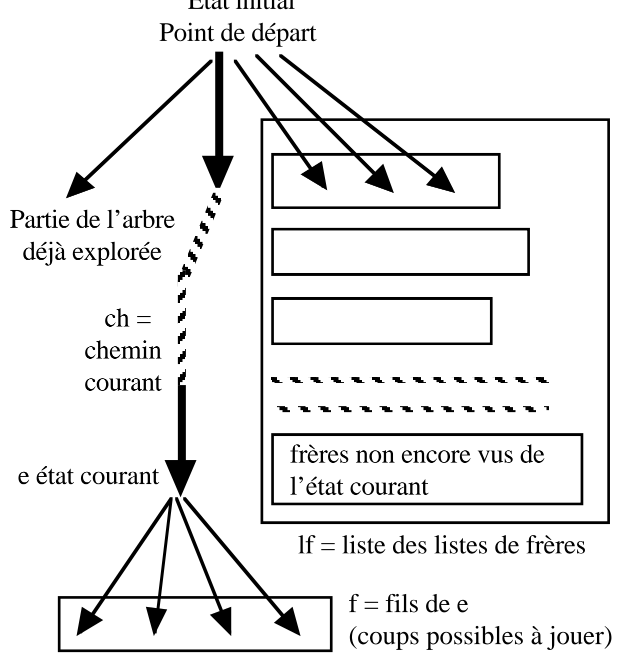
Chercher à résoudre un problème en construisant pas à pas l'éventuelle solution, quitte à revenir en arrière dans la suite des choix à faire, est connu sous le nom d'algorithme à retour ou "backtracking". Il s'agit en fait d'explorer une arborescence en suivant l'ordre racine-gauche-droite encore appelé "parcours en profondeur d'abord".

L'exemple le mieux connu de problème est celui des reines de Gauss, exposé plus loin, mais celui de la décomposition d'un nombre grâce à une liste fournie est plus facile à programmer. Dans la version ci-dessous on construit pas à pas une liste ch de nombres choisis en retirant de la somme s chacun des choix. C'est pourquoi le succès est assuré dès lors que s devient nul au second cas.

Le cas où la liste des nombres restants f est vide, se décompose en deux, si rien n'a pu être choisi car ch est vide, c'est un échec, sinon on retire le dernier choix de ch, on le rajoute donc à s, c'est le retour en arrière, c'est à dire une remontée dans l'arbre.

Le troisième cas indique que si le premier des choix possibles est trop grand (il dépasse s) alors on va voir les frères, c'est un déplacement "horizontal" à droite dans l'arbre.

Enfin le dernier cas exprime une descente en déduisant le premier nombre de f de s et en l'ajoutant à la liste ch.




```
let rec parcours s f ch lf = (* s = somme à annuler, f = liste des entiers fils d'un même noeud *)
                (* ch = liste des entiers déjà choisis, lf = liste des listes de leurs freres *)
    if f = [] then      if ch = [] then (* échec *) []
                              else parcours (s + (hd ch)) (hd lf) (tl ch) (tl lf)     (* remontée *)
    else if hd f = s then      (* fini avec succès *)     ((hd f) :: lc)
    else if hd f < s then (*descente*) parcours (s - (hd f)) (tl f) ((hd f) :: ch) ((tl f) :: lf)
    else parcours s (tl f) ch lf;;        (* on va chercher le frère *)
```

```
let decomp s ln = parcours s ln [] [];;
        (* c'est la fonction principale, qui se contente d'appeler le début du parcours *)
```

```
decomp 7 [3; 2; 1; 4; 3];; ☞ [3; 1; 3]
decomp 7 [3; 2; 1];; ☞ []
decomp 17 [3; 4; 2; 3; 1; 5; 2; 1];; ☞ [5; 3; 2; 4; 3]
```

Nous donnons à présent une seconde solution plus courte mais peut être moins lisible utilisant une exception, où cette exception indique comment repartir :

```
exception impossible;;

let rec decomp  s ln = match ln with [] -> if s = 0 then [] else raise impossible
                | x::q -> try let sol = decomp (s - x) q in x :: sol  with impossible -> decomp s q;;
```

Remarque

La solution plus haut est directement inspirée d'une écriture des fonctions en langage Lisp. Il faut savoir que Lisp n'est pas typé et que les parenthèses délimitent de façon cohérente toute expression évaluable écrite en notation préfixée. C'est à dire que la demande (f a b c) retourne, si c'est possible, la valeur de la fonction f appliquée sur les arguments a, b, c. La puissante fonction "cond", qui ne peut avoir d'équivalent en langage typé, s'arrête sur l'évaluation du résultat correspondant au premier test non faux. Le paramètre ch désigne toujours la liste des choix, exprimée à l'envers, le dernier choix est en premier.

```
(defun som (s ch lf f) (cond ; f = liste des nombres restants (non choisis)
        ((eq s 0) ch)
        ((null f) (if (null ch) 'impossible  (som (+ s (car ch)) (cdr ch) (cdr lf) (car lf) )))
        ((< s (car f)) (som s ch lf (cdr f))) ; le premier nombre possible est trop grand
        (t (som (-  s (car f)) (cons (car f) ch) (cons (cdr f) lf) (cdr f))  ))) ; descente

(defun decomp (s ln) (som s nil nil ln ))
```

```
(decomp 7 '(3 2 1 4 3)) ☞ (3 1 3)
```

2. Obtention de toutes les solutions et application aux décompositions entières d'un entier

Dans ce qui précède on s'arètait à la première solution rencontrée. Une solution pour avoir toutes les décompositions (en tenant compte de l'ordre) peut s'envisager avec une "itération" map :

```
let rec tdec s = fun    | [] -> []
                        | (x :: q) -> if x > s then tdec s q else (tdec s q) @ (if s = x then [[s]]
                                  else map (fun t -> x :: t) (tdec (s - x) q));;
```

```
tdec 7 [3; 2; 1; 4; 3];; ☞ [[4; 3]; [2; 1; 4]; [3; 4]; [3; 1; 3]]
```

Ou encore avec deux fonctions mutuellement récursives :

```
let rec tdec s = fun    | [] -> []
                        | (x :: q) -> if x > s then tdec s q else ttdec (tdec s q) [x] (s - x) q
and ttdec sol lc = fun  | 0 _ -> (lc :: sol)
                        | _ [] -> sol
                        | s (x :: q) -> ttdec (sol @ (ttdec [] lc s q)) (x :: lc) (s - x) q;;
```

```
tdec 7 [3; 1; 4;2; 3];;
☞ [[2; 4; 1]; [3; 4]; [4; 3]; [3; 1; 3]]
tdec 17 [3; 6; 5; 1; 4; 2; 3];;
☞ [[2; 4; 5; 6]; [3; 2; 1; 5; 6]; [3; 2; 4; 5; 3]; [3; 4; 1; 6; 3]; [3; 5; 6; 3]; [2; 1; 5; 6; 3]]
```

Une autre solution utilisant un tableau et des exceptions est donnée plus loin. Concernant la décomposition d'un entier en partition d'entiers distincts il suffit d'appliquer tdec à :

```
let rec compte k = if k = 0 then [] else k :: (compte (k - 1));;
let partition n = tdec n (compte n);;
```

```
compte 5;; ☞ [5; 4; 3; 2; 1]
partition 7;; ☞ [[4; 2; 1]; [4; 3]; [5; 2]; [6; 1]; [7]]
```

3. Problème général et application au "compte est bon"

Dans la fonction de parcours nommée "bak", "fils" doit être une fonction qui à un noeud associe la liste de ses fils (les coups suivants possibles dans un jeu), "init" est l'état initial ou racine, et "but" est la fonction booléenne indiquant si un état (un noeud de l'arbre) est terminal. En général, pour tous ces problèmes, c'est la fonction "fils" qui est la plus difficile à écrire. Dans ce qui suit, ch est le chemin complet exprimé à l'envers, c'est à dire la liste des noeuds parcourus de la racine à l'état courant e, enfin r désigne la liste des états restant à explorer.

```
let bak fils but init = bak' [init] []
        where rec bak' = fun [] ch -> false, rev ch
                | (e :: r) ch -> if but e then true, rev (e :: ch)      (* ces parenthèses obligatoires*)
                                else if mem e ch then bak' r ch  (* cas où e a déjà été vu *)
                                else bak' ((fils e) @ r) (e :: ch) ;;
        ☞   bak : ('a -> 'a list) -> ('a -> bool) -> 'a -> bool * 'a list = <fun>
```

Remarque

Pour un parcours en profondeur limitée, il suffit de rajouter un paramètre entier n à bak et à bak', le dernier appel de bak' étant avec n-1 et il faut un test supplémentaire à la manière de :
"if n < 1 then (false, inv C)".

Retour sur le "compte est bon" avec l'expression générale du backtracking :

Un état e est une liste d'entiers pris dans la liste initiale de nombres ln, "ret" ne retire que la première occurrence de x dans a, "dif" n'est pas la différence ensembliste, mais renvoie la liste a privée des éléments de b en tenant compte des répétitions.

```
let rec ret x a = match a with [] -> [] | (y :: q) -> if x=y then q else y :: (ret x q);;

let rec dif a b = match b with [] -> a | (x :: q) -> dif (ret x a) q;;

let suivants ln e = map (fun x -> x::e) (dif ln e);;

let decomp s ln = bak (suivants ln) (fun e -> (s = it_list (prefix +) 0 e)) [] ;;
```

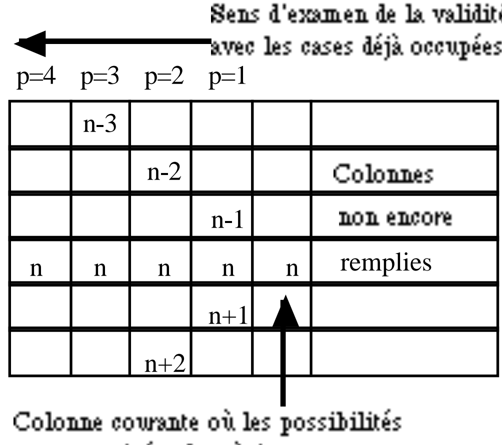
```
suivants  [1;2;3;4;5;3;8;1] [2;3;1];;
☞ [[4; 2; 3; 1]; [5; 2; 3; 1]; [3; 2; 3; 1]; [8; 2; 3; 1]; [1; 2; 3; 1]]
decomp 7 [3; 2; 1; 4; 3];; ☞   tout le chemin aboutissant à [3; 1; 3]
```

4. Exemple des reines de Gauss

Ce célèbre problème consiste à placer n reines sur un damier de n lignes et n colonnes de façon à ce qu'aucune ne soit en position d'être prise par une autre. On s'arrête à la première solution rencontrée. Pour 3 reines placées aux 3 premières colonnes sur les lignes 1, 3, 5, on représentera cet état provisoire par [5; 3; 1] la position suivante i sera donc empilée en premier. La validité indique que la rangée i est compatible avec l'état antérieur e, à savoir que j colonnes avant celle où on se propose de mettre une reine à la ligne i, si la reine se trouve à la ligne k, on doit avoir $i \neq k$ et sans prise possible en diagonale c'est à dire $k \neq i + j$ et $k \neq i - j$:

```
let valide i e = test i 1 e
where rec test = fun _ _ [] -> true
| i j (k :: q) -> (i <> k) & (abs(k - i) <> j)
                    & (test i (j + 1) q);;
```

```
valide 5 [3;1];; ☞ bool = true
valide 2 [3;1];; ☞ bool = false
```



Les états qui suivent e forment la liste fille f des états i :: e pour les entiers i de 1 à n, pourvu qu'ils soient valides avec e.

```
let suiv n e = suiv' 1 []
where rec suiv' i f = if i = n + 1 then f
else suiv' (i + 1)
        (if valide i e then (i :: e) :: f else f) ;;
```

```
suiv 8 [5;3;1];;          ☞ : int list list = [[8; 5; 3; 1]; [7; 5; 3; 1]; [2; 5; 3; 1]]
```

En Haskell, on aurait très simplement suiv e = [i::e | i <- [1..n], valide i e]. Par ailleurs, pour voir le résultat, on utilise une instruction :

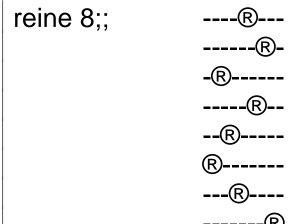
```
make_string 7 `s`;;       ☞ string = "sssssss"  (* fabrication de chaînes de caractères *)
```

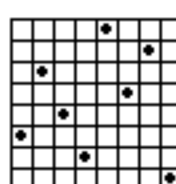
```
let rec vue e n = if e <> [] then (vue' (hd e); vue (tl e) n)
        where vue' i = (print_string (make_string (i - 1) `-`); print_string "®";
                        print_string (make_string (n - i) `-`); print_newline() ) ;;
```

La composition des trois fonctions "hd", "rev", "snd" n'étant là pour n'avoir que le dernier état qui nous intéresse. Et finalement, la fonction principale "reine" ne dépend que de la dimension du damier.

```
let reine n = vue (hd (rev (snd (bak (suiv n) (fun x -> n = list_length x) [])))) n;;
```

reine 8;;





Remarque sur le même programme en Lisp

Dans la fonction "fils", on veut renvoyer les positions possibles suivantes.
La fonction prédéfinie "append" réalise la concaténation entre nil qui signifie le faux, mais est en même temps la liste vide, d'où un raccourci permis par ce langage non typé.

```
(defun reines (n)
        (reso n (list n) (list (cdr (fils () n))) (fils (list n) n)))

(defun reso (n ch lf f) (cond
        ((null ch) 'echec) ; cas où l'arbre est entièrement exploré
        ((eq (length ch) n) ch) ; cas final du succès
        ((null f) (reso n (cdr ch) (cdr lf) (car lf))) ; cas d'une remontée
        (t (reso n (cons (car f) ch) (cons (cdr f) lf) (fils (cons (car f) ch) n))))) ; descente

(defun fils (ch n) (if (zerop n) nil (append (fbis n 1 ch) (fils ch (1- n)))))
```

```
(defun fbis (n p ch) (cond
        ((null ch) (list n))
        ((eq (car ch) n) nil)
        ((eq (+ (car ch) p) n) nil)
        ((eq (- (car ch) p) n) nil)
        (t (fbis n (1+ p) (cdr ch)))))

        (reines 8) ☞ (5 7 2 6 3 1 4 8)
```

Remarque sur une écriture Caml concentrée

On peut tout grouper dans un "programme" (ce qui n'est pas recommandé, car si la lecture de la programmation est descendante, sa mise au point est très difficile, la programmation fonctionnelle est plutôt ascendante, ce qui est déjà vrai en Lisp mais encore plus en Caml. Il est prudent que chaque petite fonction soit testée au fur et à mesure, et pour son type, et à son éxecution) la "reine" suivante est bien plus un exercice de style qu'un programme lisible :

```
let reine n = vue (hd (inv (snd (bak suiv (fun x -> n = list_length x) []))))
        where rec vue e = (if e <> [] then (vue' (hd e); vue (tl e) )
                where vue' i = (print_string (make_string (i - 1) `-`); print_string "®";
                                print_string (make_string (n -  i) `-`); print_newline() ) )
        and inv q = (inv' [] q where rec inv' r = fun [] -> r | (a :: d) -> inv' (a :: r) d)
        and suiv e = (suiv' 1 []
        where rec suiv' i f = if i = n+1 then f else suiv' (i + 1) (if (valide i e) then (i :: e):: f else f)
        where valide i e = test i 1 e
        where rec test = fun _ _ [] -> true| i j (k:: r) -> (i <> k) & (abs(k - i) <> j) & (test i (j + 1) r))
        and bak s but d = bak' [d] []
                where rec bak' = fun [] c -> false, inv c
                                | (e :: r) c -> if but e then true, inv (e :: c)
                                        else if mem e c then bak' r c
                                        else bak' ((s e) @ r) (e :: c) ;;
```

5. Exemple du taquin 3*3

Dans ce jeu, un carré de côté n, une case est laissée vide, on peut y faire glisser l'une des voisines. Après un certain nombre de tels mouvements, il est possible en partant d'un état initial donné d'arriver à la moitié des configurations possibles. On note I et B les deux états fixés : initial et but, par exemple :

I =

5	2	8
3	4	0
6	7	1

B =

1	2	3
8	0	4
7	6	5

On adopte une représentation sous forme de liste où 0 figure la case vide et où le n-ième élément (compté à partir de 0) est le couple (i, j) repérant sa position du numéro de ce rang. La fonction précédente "bak" est modifiée de façon à renvoyer un booléen uniquement, mais en visualisant les mouvements du chemin ch qui est la liste des états retenus au cas où le problème est faisable.

```
let round x = if x >=. 0. then int_of_float(x +. 0.5) else -(int_of_float (-. x +. 0.5));;  (* arrondi *)

let voir ll = let n = round (sqrt (float_of_int (list_length ll))) in (* sert à visualiser la grille carrée *)
                for i = 1 to n do (for j = 1 to n do print_int (rang i j ll) done; print_newline()) done
        where rec rang i j = fun (x :: q) -> if x = (i, j) then 0 else 1 + (rang i j q);;

let bak fi but d = bak' [d] []
        where rec bak' = fun [] ch -> false| (e :: r) ch-> if but e then (map voir (rev (e :: ch)); true)
                                else if mem e ch then bak' r ch  (* cas où e a déjà été vu *)
                                else  bak' ((fi e) @ r) (e :: ch);;  (* cas où on explore les fils de e *)
```

```
let ham (a, b) (c, d) = abs (a - c) + abs (b - d);; (* distance de Hamming*)

let fils q = fils' [] (tl q)
        where rec fils' f = fun [] -> f | (p:: r) -> fils' (if ham p (hd q) = 1 then nouv:: f else f) r
                where nouv = (p :: (avant [] (tl q))) @ (hd (q) :: r)
                where rec avant la = fun [] -> rev la
                                        | (a :: x) -> if a = p then rev la else avant (a :: la) x ;;
        ☞ fils : (int * int) list -> (int * int) list list = <fun>
```

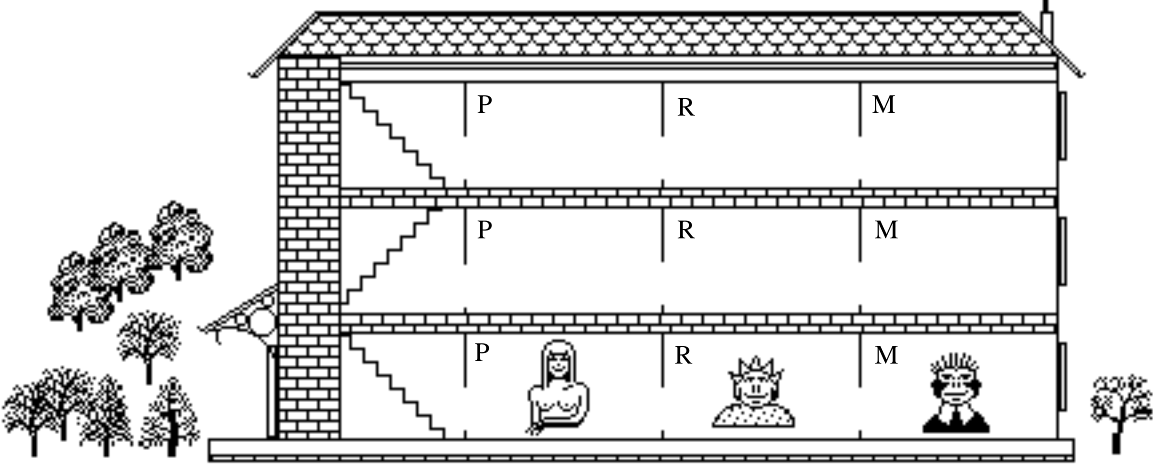
"Avant la x" est une fonction locale donnant le résultat "la" formé des éléments de la liste q situés strictement entre le premier et p, sachant que r est la queue de q qui suit p. Le paramètre x symbolise le début de la liste q qui doit être parcourue jusqu'à trouver p. Le carré "init" (initial) est formé en ligne par les numéros 5 2 8, puis 3 4 vide, et 6 7 1.

```
let init = [(2,3); (3,3); (1,2); (2,1); (2,2); (1,1); (3,1); (3,2); (1,3)];; (* l'état initial *)

fils init;; ☞ (int * int) list list = [[1, 3; 3, 3; 1, 2; 2, 1; 2, 2; 1, 1; 3, 1; 3, 2; 2, 3]; [2, 2; 3, 3;
1, 2; 2, 1; 2, 3; 1, 1; 3, 1; 3, 2; 1, 3]; [3, 3; 2, 3; 1, 2; 2, 1; 2, 2; 1, 1; 3, 1; 3, 2; 1, 3]]
(*On vérifiequ'il s'agit des 3 états possibles à partir de I *)

let fin = [(2,2); (1,1); (1,2); (1,3); (2,3); (3,3); (3,2); (3,1); (2,1)];;
        (* "fin" est le but à atteindre *)

let taquin = bak fils (fun x -> (x = fin));;
```

Ce problème, demandé par "taquin init;;" est très long, il ne mène guère à la solution même pour de petites valeurs de n, on utilise alors des heuristiques cherchant par exemple à minimiser la distance de Hamming d'avec le but.

6. La princesse, le roi et le vilain baron

Au premier niveau d'un château se trouvent trois chambres occupées par une princesse, le roi et un méchant. Les deux étages au dessus contiennent chacun trois chambres réservées dans le même ordre à ces trois personnages. Il faut les faire passer à un autre étage sans que jamais la princesse ne se trouve au même étage que le méchant sans qu'il y ait le roi pour s'interposer. De plus, aucune personne ne peut pénetrer dans une chambre occupée et une personne au plus doit se trouver à un moment donné dans les escaliers.



On peut montrer par récurrence que pour n personnes (n > 2) sur n étages, la plus courte solution se fait en 2n + 1 mouvements. On va résoudre le problème pour une hiérarchie de n personnes. Le problème est donc de conserver à chaque étage une sous-liste consécutive "croissante" c'est à dire respectant l'ordre initial. Le problème des tours de Hanoi impose simplement une croissance

(comme P < M sans R comme séparateur), par contre, on peut envisager un troisième problème où on impose des sous-listes consécutives (croissantes ou décroissantes comme M, R ou R, P) en s'arrêtant sur la même disposition à un étage quelconque différent du rez de chaussée, en ce cas 3 étages et 2n-1 mouvements suffisent.

On fait chercher, en incluant les contraintes dans la fonction donnant les fils (au plus deux pour chaque tête d'étage). La fonction "place" parcourt les étages pour retirer x et le placer d'une certaine façon, lv = partie de la liste déjà vue, le dernier argument est la liste restant à voir. La fonction renvoie l'unique état où x est placé dans un étage vide si i = 0, de façon croissante si i = 1 et décroissant si i = -1. Pour l'entier x il y a en effet au plus 3 états suivants.

```
let rec place x fait lv i = fun [] -> if fait then rev lv else []
            (* unique état où x  placé avant son suivant si i = 1 *)
      | ([] :: q) -> if  i = 0 & not fait then place x true ([x] :: lv) i q else place x fait ([] :: lv) i q
            (* x déjà placé sur étage vide *)
      | ((y :: t) :: q) -> if x = y  then place x  fait (t :: lv) i q  (* l'étage débutant par x tronqué *)
            else if x + i  = y & not fait then  place x  true ((x :: y :: t) :: lv) i q
            else place x fait ((y :: t) :: lv) i q;; (* cas où x ne peut être placé devant y *)

let rajout m q = if m=[] then q else m::q;;

let fils e = let rec filsbis e f =
      fun [] -> f | ([] :: r) -> filsbis e f r  (* si x est déjà seul, on ne le met pas ds un ét. vide *)
      | ((x :: etag) :: r) -> filsbis e  (rajout (place x  false [] 1 e)
                        (rajout (place x  false [] (-1) e) (rajout (place x  false  []  0 e) f))) r
      in filsbis e [] e;;
```

```
(* exemple avec croissant et décroissant *)
fils [[2];[3];[1]];;
☞ [[[1; 2]; [3]; []]; [[3; 2]; []; [1]]; [[]; [2; 3]; [1]]; [[]; [3]; [2; 1]]]

fils [[4;3;2;1]; []; [5;6]; []; [7]];;
☞ [[[4; 3; 2; 1]; [7]; [5; 6]; []; []];  [[5; 4; 3; 2; 1]; []; [6]; []; [7]];  [[4; 3; 2; 1]; [5]; [6]; []; [7]];
[[3; 2; 1]; []; [4; 5; 6]; []; [7]];   [[3; 2; 1]; [4]; [5; 6]; []; [7]]]
```

```
let rec construit ind et = let rec compte n = if n = 0 then [] else n :: (compte (n - 1))
      in if et = 1 then [(compte ind) ] else [] :: (construit ind (et - 1));;
```

```
construit 7 5 ☞ [[]; []; []; []; [7; 6; 5; 4; 3; 2; 1]] *)
```

```
let rec diff a b = match a with [] -> [] | (x::q) -> if mem x b then diff q b else x :: (diff q b);;
      (* différence ensembliste *)

let rec rech ch lf but  p = (* le départ sera avec un chemin ch contenant l'état initial, et lf ; les fils *)
      if lf=[] then [] (* échec *)
      else if mem but (tl (hd ch)) then rev ch
            (* le but est ici la présence de l'étage initial ds un étage supérieur *)
      else if hd lf = [] then rech (tl ch) (tl lf) but (p + 1) (* remontée *)
      else if p < 1 then rech ch ((tl (hd lf)) :: (tl lf)) but p
            (* profondeur limitée, sinon solutions trop longues *)
      else rech ((hd (hd lf)) :: ch) ((diff (fils (hd (hd lf))) ch) :: (tl (hd lf)) :: (tl lf)) but (p - 1);;
            (* descente *)

let jeu n t = let init = rev (construit n t) in rech [init] [fils init] (hd init) (2*n + 1);;
      (* exemple avec respect de l'ordre *)
```

```
jeu 5 5;; ☞ [[[5; 4; 3; 2; 1]; []; []; []; []]; [[4; 3; 2; 1]; [5]; []; []; []];  [[3; 2; 1]; [5]; [4]; []; []]; [[3;
2; 1]; []; [5; 4]; []; []];  [[2; 1]; [3]; [5; 4]; []; []]; [[1]; [3]; [5; 4]; [2]; []];  [[]; [3]; [5; 4]; [2]; [1]]; [[];
[3]; [5; 4]; []; [2; 1]];  [[5]; [3]; [4]; []; [2; 1]]; [[5]; []; [4]; []; [3; 2; 1]];  [[5]; []; []; []; [4; 3; 2; 1]];
[[]; []; []; []; [5; 4; 3; 2; 1]]]
```

Exemple pour croissant et décroissant :

jeu 8 3 ;;
☞ [[[8; 7; 6; 5; 4; 3; 2; 1]; []; []]; [[7; 6; 5; 4; 3; 2; 1]; [8]; []]; [[6; 5; 4; 3; 2; 1]; [7; 8]; []]; [[5; 4; 3; 2; 1]; [6; 7; 8]; []]; [[4; 3; 2; 1]; [5; 6; 7; 8]; []]; [[3; 2; 1]; [4; 5; 6; 7; 8]; []]; [[2; 1]; [3; 4; 5; 6; 7; 8]; []]; [[1]; [2; 3; 4; 5; 6; 7; 8]; []]; [[]; [2; 3; 4; 5; 6; 7; 8]; [1]]; [[]; [3; 4; 5; 6; 7; 8]; [2; 1]]; [[]; [4; 5; 6; 7; 8]; [3; 2; 1]]; [[]; [5; 6; 7; 8]; [4; 3; 2; 1]]; [[]; [6; 7; 8]; [5; 4; 3; 2; 1]]; [[]; [7; 8]; [6; 5; 4; 3; 2; 1]]; [[]; [8]; [7; 6; 5; 4; 3; 2; 1]]; [[]; []; [8; 7; 6; 5; 4; 3; 2; 1]]]

7. L'alpiniste et les camps de base

Un alpiniste solitaire doit escalader un sommet situé à N jours de marche, ne pouvant porter qu'au maximum une quantité P exprimée en nombre de jours de vivres, il doit faire un certain nombre d'aller-retours pour laisser des vivres à différents "camps de base". On considère que ces camps sont numérotés de 0 (le départ où il dipose d'une quantité Q) à N (le sommet), espacés d'une journée de marche, qu'il ne prend pas de jour de repos et qu'il doit revenir à son point de départ. Le test de réussite peut être simplement : être à la position N du sommet avec une répartition des charges quelconque en 0 et 1 partout ailleurs car toute solution autre signifie qu'il y a eu trop de vivres montés.

Il faut donc construire un programme ayant pour données N, P, Q indiquant si l'aventure est possible, et, si oui, donnant le nombre de jours de marche et le détail des trajets (se restreindre à de petites valeurs de N et P).
Exemples pour :
N = 1, P = 2, Q = 2, Nombre de jours 2 → position 0 vivres [2,0] → position 1 vivres [0,1] → position 0 vivres [0,0]

N = 3, P = 4, Q = 7, pas de solution

N = 3, P = 4, Q = 8, Nombre de jours 8 → position 0 vivres [8,0,0,0] → position 1 vivres [4,3,0,0] → position 0 vivres [4,2,0,0] → position 1 vivres [0,5,0,0] → position 2 vivres [0,1,3,0] → position 3 vivres [0,1,0,2] (remarque, monter au sommet avec une charge de 2 suffisait) position 2 vivres [0,1,1,0] → position 1 vivres [0,1,0,0] → position 0 vivres [0,0,0,0]

N = 3, P = 3; Q = 17, ce n'est pas possible mais si Q = 18, Nombre de jours 18 → position 0 vivres [18,0,0,0] → position 1 vivres [15,2,0,0] → position 0 vivres [15,1,0,0] → position 1 vivres [12,3,0,0] → position 0 vivres [12,2,0,0] → position 1 vivres [9,4,0,0] → position 2 vivres [9,1,2,0] → position 1 vivres [9,1,1,0] → position 0 vivres [9,0,1,0] → position 1 vivres [6,2,1,0] → position 0 vivres [6,1,1,0] → position 1 vivres [3,3,1,0] → position 0 vivres [3,2,1,0] → position 1 vivres [0,4,1,0] → position 2 vivres [0,1,3,0] → position 3 vivres [0,1,0,2] → position 2 vivres [0,1,1,0] → position 1 vivres [0,1,0,0] → position 0 vivres [0,0,0,0]

Un état du problème est représenté par la liste (i q0 q1 q2 q3 ... qn) où i désigne la position de l'alpiniste et q0 la quantité de vivres au départ, q1 celle à la première étape, ... qn celle au sommet, ainsi l'état initial est (0 q 0 0 ... 0), tout état débutant par 0 peut être considéré comme un retour, et toute suite d'état comprenant un état débutant par n vérifiera le prédicat "sommet".
On construit deux fonctions "monte" et "descend" retournant la liste des vivres obtenue à partir de la position i courante, de la charge k à transporter et de la suite s des vivres (de 0 à n). La fonction fils pour un état e appelle la fonction filsbis récursive terminale qui délivre son résultat r de tous les états possibles avec une charge k en décroissant k >1.

Avec au choix 2 fonctions "fils" :

```
let rec nth i s = if i = 0 then hd s else nth (i -1) (tl s)
        (* "and" n'est qu'une liaison entre 3 définitions *)

and tete s n = if s = [] or n = 0 then [] else (hd s):: (tete (tl s) (n - 1)) (* les n premiers de s *)
```

```
and queue s n = if s = [] or n = 0 then s else queue (tl s) (n - 1);;  (* sauf les n premiers *)

let monte i k s = (tete s i) @ (((nth i s) -  k) :: ((nth (i + 1) s) + k - 1) :: (queue s (i + 2)));;

let descend i k s = (tete s (i - 1)) @ (((nth (i -1) s) + k - 1) :: ((nth i s) - k) :: (queue s (i + 1)));;
```

```
monte 2 3 [1;2;3;4;5];;
☞ [1; 2; 0; 6; 5]
descend 2 3 [1;1;5;2];;
☞ [1; 3; 2; 2]
```

```
let fils n p = fun (i :: le) -> let q = nth i le in filsbis le (min q p) []
        where rec
        filsbis le k r =
                if k = 0 then r (* donne la liste r des fils de l'état le pour la position courante i *)
                else filsbis le (k - 1)
                        (if i = 0 then (1 :: (monte 0 k le)) :: r
                                else if i = n then ((n - 1) :: (descend n k le)) :: r
                        else ((i + 1) :: (monte i k le)) :: ((i - 1) :: (descend i k le)) :: r);;

let fils n p = fun (i::le) -> (* donne la liste r des fils de l'état le pour la position courante i *)
        let f = ref [] in (* f sera une variable locale *)
        for k = 1 to (min (nth i le) p) do (* utilisation d'une boucle *)
                if 0 < i then f := ((i - 1) :: (descend i k le)) :: !f;
                if i < n then f :=  ((i + 1) :: (monte i k le)) :: !f
        done; !f;; (* version anticipant sue le chapitre 6 *)
```

```
fils 3 2 [2;2;3;4;5];;
☞ [[3; 2; 3; 3; 5]; [1; 2; 3; 3; 5]; [3; 2; 3; 2; 6]; [1; 2; 4; 2; 5]]
```

```
let retour e = match e with 0 :: _ -> true | _ -> false;;

let rec sommet n s = match s with [] -> false | ((x :: _) :: q) -> if x = n then true else sommet n q;;
```

Dans la fonction de parcours nommée "bak" de type int -> (int list -> int list list) -> int list -> bool * int list list, "suivants" doit être une fonction qui à un noeud associe la liste de ses fils (les coups suivants possibles dans un jeu) ce sera la fonction "fils n p" (à qui il ne manque plus qu'un argument, d'où l'avantage de la currification).
L'argument "init" est l'état initial ou racine, il n'y aurait qu'à mettre un argument supplémentaire "but" (fonction booléenne indiquant si un noeud de l'arbre est terminal) pour rendre la fonction "bak" utilisable pour tout problème de backtracking.

```
let bak n suivants init = bak' [init] (suivants init) [[]]
        (* init est l'état initial, ch est le chemin, e est l'état courant *)
        where rec bak' = fun
                | [] _ _  -> false, [] (* cas de l'échec *)
                | (e :: ch) [] (f :: lf) -> bak' ch f lf  (* cas où il n'y a plus de fils, on remonte *)
                | ch (e :: r) lf -> if (retour e) && (sommet n ch) then true, rev (e :: ch)
                                        (* cas du succès *)
                                else if mem e ch then bak' r ch lf  (* cas où e a déjà été vu *)
                                else bak' (e :: ch) (suivants e) (r :: lf) ;;
                                        (* cas de l'examen du premier fils disponible *)

let rec zeros k = if k=0 then [] else 0 :: (zeros (k - 1));;

let init n q = 0 :: q :: (zeros n);;
```

```
init 4 12;;
☞ [0; 12; 0; 0; 0; 0]
```

```
let reso n p q = bak n (fils n p) (init n q);;
```

reso 3 4 7;;
☞ false, []
reso 3 4 8;;
☞ true, [[0; 8; 0; 0; 0]; [1; 4; 3; 0; 0]; [0; 4; 2; 0; 0]; [1; 0; 5; 0; 0]; [2; 0; 1; 3; 0]; [3; 0; 1; 1; 1]; [2; 0; 1; 1; 0]; [1; 0; 1; 0; 0]; [0; 0; 0; 0; 0]] ou bien true, [[0; 8; 0; 0; 0]; [1; 4; 3; 0; 0]; [0; 4; 2; 0; 0]; [1; 0; 5; 0; 0]; [2; 0; 1; 3; 0]; [3; 0; 1; 0; 2]; [2; 0; 1; 1; 0]; [1; 0; 1; 0; 0]; [0; 0; 0; 0; 0]]

reso 3 3 18;;
☞ [[0; 18; 0; 0; 0]; [1; 15; 2; 0; 0]; [0; 15; 1; 0; 0]; [1; 12; 3; 0; 0]; [0; 12; 2; 0; 0]; [1; 9; 4; 0; 0]; [0; 9; 3; 0; 0]; [1; 6; 5; 0; 0]; [0; 6; 4; 0; 0]; [1; 3; 6; 0; 0]; [0; 3; 5; 0; 0]; [1; 0; 7; 0; 0]; [2; 0; 4; 2; 0]; [1; 0; 4; 1; 0]; [2; 0; 1; 3; 0]; [3; 0; 1; 1; 1]; [2; 0; 1; 1; 0]; [1; 0; 1; 0; 0]; [0; 0; 0; 0; 0]] (* solution en 18 trajets, différente de celle de l'énoncé. *)

CHAPITRE V

CHAINES DE CARACTERES

Les chaînes de caractères de Caml doivent être délimitées par des guillemets. Les quelques fonctions prédéfinies suivantes sont illustrées sur des exemples.

```
let a="sic transit gloria mundi";;  ☞ a : string = "sic transit gloria mundi"
set_nth_char c 3 `x`;;              ☞ unit = ()
        (* cette fonction est un effet de bord, c'est une affectation *)
c;;                                 ☞ string = "azexty"
string_length c;;                   ☞ int = 6          (* String.length en Ocaml *)
sub_string c 3 2;;                  ☞ string = "xt"    (* String.sub en ocaml *)
        (* les caractères sont comptés à partir de 0 *)
nth_char "abcd" 2;;                 ☞ char = `c`       (* String.get en Ocaml *)
string_of_int 2003;;                ☞ string = "2003"
int_of_string "2003";;              ☞ int = 2003
int_of_char `a`;;                   ☞ int = 97    (* donne le code ASCII *)
char_of_int 65;;                    ☞ char = `A`
string_of_float 3.14;;              ☞ string = "3.14"
float_of_string "1.5e3";;           ☞ float = 1500
```

1. Manipuler des caractères avec des chaînes

Il n'y a pas de fonction construisant une chaîne à partir de caractères, il faut donc la définir :

```
let string_of_char x = let c = " " in set_nth_char c 0 x; c;;
        ☞ string_of_char : char -> string = <fun>
```

```
string_of_char `a`;;                ☞ string = "a"
```

2. Transformation d'une liste de chaîne en une chaîne

Mais pas d'une liste quelconque, problème : faire une fonction abstraite.

```
let rec implose = fun [] -> "" | (a :: q) -> a ^ (implose q);;
        ☞ implose : string list -> string = <fun>
```

```
implose ["a"; "b"; "cd"; "efg"];;   ☞ string = "abcdefg"
```

3. Liste des caractères d'une chaîne

```
let rec explose = fun "" -> [] | x -> (nth_char x 0) :: (explose (lasuite x))
        where lasuite c = sub_string c 1 ((string_length c) - 1);;
☞ explose : string -> char list = <fun>
```

```
explose "CAML";;                    ☞ char list = [`C`; `A`; `M`; `L`]
explose     "A
            la
            ligne";;
☞ char list = [`A`; `\n`; `\t`; `l`; `a`; `\n`; `\t`; `l`; `i`; `g`; `n`; `e`]
        (* \n et \t sont les caractères "return" et "tabulation" de codes 13 et 9 *)
```

4. Application à la liste des chiffres d'un nombre réel

On souhaite obtenir la liste des chiffres d'un nombre réel, en ayant retiré la virgule. On peut, cependant, constater, que "string_of_float" limite sa lecture.

```
let rec retvirgule l = match l with [] -> []
        | (x::q) -> if x = `.`then q else x :: (retvirgule q);;
        (* pour retirer la virgule *)

let listchif r = map (fun x -> int_of_char x  -  48) (retvirgule (explose (string_of_float r)));;
```

```
listchif 3.14159;; ☞ [3; 1; 4; 1; 5; 9]
listchif 12.;;  ☞ [1; 2; 0]
string_of_float 123456789.987654321;; ☞ "123456789.988"
listchif 123456789.987654321;; ☞ [1; 2; 3; 4; 5; 6; 7; 8; 9; 9; 8; 8]
```

5. Montage électrique

Un montage formé de résistances va être codé par une chaîne de "s" pour série et "p" pour parallèle, suivis chaque fois de deux sous-circuits, les résistances étant des chiffres de 1 à 9. Les "s" et "p" sont donc en fait deux opérateurs binaires dans une expression non parenthèsée.
Le but, ici, est de programmer une fonction calculant la résitance du circuit à partir de la chaîne de caractère le décrivant.

La solution proposée consiste à rechercher dans la chaîne, la première sous-chaîne correspondant à une résistance calculable et renvoyant le couple formé par cette résistance numérique et l'indice de la position suivante à lire dans la chaîne. Ainsi, pour "s34p37" et la position 0, la fonction renverra le calcul de la première résistance, soit 3 + 4 = 7 et l'indice 3 où devra commencer la lecture suivante à savoir "p37", où la fonction renverra 2.1

```
let chiffre c  = 47 < c && c < 58;;

let rec resistance s i = (* si est le code du caractère à la position i de la chaîne s *)
        let si = int_of_char (nth_char s i) in
        if chiffre si then (float_of_int (si - 48), i + 1)
        else (let (a, j) = resistance s (i + 1) in
                let (b, k) = resistance s j in
                if si = int_of_char  `s` then (a  +. b, k) else (a *. b  /. (a  +. b), k) );;
```

```
resistance "p37" 0;;☞  float * int = 2.1, 3
resistance "s34p37" 0;;☞  float * int = 7.0, 3
resistance "ss5ps23s4p236" 0;;☞  float * int = 13.5490196078, 13
```

6. Nombres de Champernowne et d'Erdös

0,123456789101112...est construit avec les entiers successifs, imaginé en 1933, il a été démontré que ce nombre est transcendant, normal et universel ainsi que le nombre de Erdös formé de la même manière, mais avec les nombre premiers 0.23571113171923... On va en écrire le début, en se limitant à n caractères, d'une façon récursive très simple.

```
let rec nbchiff x = if x < 10 then 1 else 1 + nbchiff (x /10);;

let champerno nc = print_string "0."; champ' 1 nc
where rec champ' n nc = if nc > 0 then (print_int n; champ' (n + 1) (nc - nbchiff n));;
```

```
champerno 128;;
☞
0.12345678910111213141516171819202122232425262728293031323334353637383940
414243444546474849505152535455565758596061626364656667686 9
- : unit = ()
```

Pour le nombre d'Erdös, on reprend le prédicat de divisibilité, puis on crée récursivement celui de primalité en testant la divisibilité de p par 2, puis tous les entiers impairs de deux en deux jusqu'à la racine carrée de p, attendu que si un diviseur est supérieur, il aurait déjà été rencontré comme quotient accompagnant un diviseur plus petit. La modification de "champerno" en "erdos" est alors évidente.

```
let divise d n = (n = (n / d) * d);;

let premier p = if p = 2 then true else if divise 2 p then false else premier' 3 p
        where rec premier' d p = if d*d > p then true
        else if divise d p then false else premier' (d + 2) p;;
```

```
premier 51;; ☞ false
premier 53;; ☞ true
```

```
let suivant n = (* renvoie le premier nombre premier qui suit n *) suivant' (n + 1)
        where rec suivant' p = if premier p then p else suivant' (p + 1);;
```

```
suivant 53;; ☞ 59
suivant 59;; ☞ 61
suivant 61;; ☞ 67
suivant 67;; ☞ 71
```

```
let erdos nc = print_string "0."; erdos' 1 nc
        where rec erdos' n nc = if nc > 0 then (print_int n; erdos' (suivant n) (nc - nbchiff n));;
```

```
erdos 128;; ☞
0.1235711131719232931374143475359616771737983899710110310710911312713113
7139149151157163167173179181191193197199211223227229233239- : unit = ()
```

7. Numération japonaise

Le japonais étant extrèmement régulier dans sa syntaxe et en particulier pour la numération, l'exemple de l'écriture littérale des nombres est assez facile.
On définit avec "japunit", la liste des mots de un à dix en japonais (On utilise aussi 4 = yo, 7 = nana, et 9 = kokono). La fonction "nth" permet d'y puiser à partir de l'index 1 contrairement aux habitudes, ainsi le troisième élément est la valeur littérale de 3.

La liste "japdiz contient les mots 10, 100, 1000, 10000 (le programme ne prévoit pas au delà).

On reprend la liste des chiffres "listchif", mais pour un entier :Une fonction "specons" permet de ne rien dire pour 0 et de ne pas mentionner "un" dans "dix" ou cent", enfin la fonction "num" à un argument entier (qui se le repasse en liste).

```
let japunit = ["itchi "; "ni "; "san "; "shi "; "go "; "roku "; "shitchi "; "hatchi "; "ku "; "giu "]
and japdiz = ["giu "; "hyaku "; "sen "; "man "];;

let rec explose = fun "" -> [] | x -> (nth_char x 0)::(explose (lasuite x))
                        where lasuite c = sub_string c 1 ((string_length c) - 1);;

let listchif r = map (fun x -> int_of_char x - 48) (explose (string_of_int r));;

let rec nth n = fun (x :: q) -> if n = 1 then x else nth (n - 1) q;;

let specons c mot suite = (* est un "cons" spécifique tenant compte de c = 0 ou 1 *)
        if c = 0 then suite else if c = 1 then mot :: suite else (nth c japunit) :: (mot :: suite);;

let num n =  list_it (fun x y -> x ^ y) (num' (listchif n)) ""
        where rec num' = fun [0] -> []
                        | [c] -> [nth c japunit]
                        | (c :: s) -> specons c (nth (list_length s) japdiz) (num' s);;
```

```
listchif 123456789;; ☞ [1; 2; 3; 4; 5; 6; 7; 8; 9]
num 289;; ☞ "ni hyaku hatchi giu ku "
num 23456;; ☞ "ni man san sen shi hyaku go giu roku "
num 10517;; ☞ "man go hyaku giu shitchi "
num 40602;; ☞ "shi man roku hyaku ni "
```

8. Ecriture littérale des nombres en bobo

L'intéret des langues régulières (ou presque) comme le bobo qui est, comme le cambodgien, en numération quinaire, ou des langues décimale (japonais...) ou vigésimale (breton..., ici 40 se dit aussi deux-vingt) rend la transcription des nombres assez simple. Le style est différent de l'exercice précédent. (Exercice spécialement prévu pour un cours donné à l'ecole Supérieure d'Informatique de Bobo-Dioulasso).

```
let rec litt = fun 0 -> ""
        | 1 -> "tele"
        | 2 -> "pela"
        | 3 ->"saa"
        | 4 -> "naa"
        | 5 -> "koo"
        | 6 -> "konala"
        | 7 -> "kopera"
        | 8 -> "korosoon"
        | 9 -> "koronoon"
        | 10 -> "fun"
        | 20 -> "kioro"
        | 100 -> "zelo"
        | x -> if x < 20 then espace 10 0 (x-10)
        else if x < 100 then espace (x / 20) 20 (x mod 20) (* faut-il dire un vingt ou vingt ? *)
        else if x < 1001 then espace (x / 100) 100 (x mod 100)  (* fun-zelo au lieu de zelo-fun *)
        else espace (x / 1000) 1000 (x mod 1000)

and espace a b c = (litt a) ^" " ^ (litt b) ^ " " ^ (litt c);;
```

```
litt 50;; ☞ "pela kioro fun"
litt 2536;; ☞ "pela fun zelo  koo zelo tele kioro fun  konala"
litt 12345;; ☞ "fun  pela fun zelo  saa zelo pela kioro koo"
```

9. Ecriture littérale des nombres en français avec versions exotiques

C'est le problème classique de transcrire un nombre en toutes lettres, mais en français cette fois, ce qui présente bien plus de particularités que dans la plupart des langues. Nous allons supposer que septante et nonante sont attestés partout en Suisse et Belgique et huitante en Suisse, quoique non réellement vrai.

La fonction n'est prévue que jusqu'à 9999, mais ce n'est pas difficile de continuer.

```
let rec litt lg = fun 0 -> "" | 1 -> "un" | 2 -> "deux" | 3 -> "trois" | 4 -> "quatre" | 5 -> "cinq" | 6 -> "six"
        | 7 -> "sept"| 8 -> "huit" | 9 -> "neuf" | 10 -> "dix" | 11 -> "onze" | 12 -> "douze"
        | 13 -> "treize" | 14 -> "quatorze" |15 -> "quinze" | 16 -> "seize"| 17 -> "dix-sept"
        | 18 -> "dix-huit" | 19 -> "dix-neuf" | 20 -> "vingt" | 30 -> "trente" | 40 -> "quarante"
        | 50 -> "cinquante"| 60 -> "soixante"| 70 -> "septante"
        | 80 -> if lg = `s`then "huitante" else "quatre-vingt"
        |90 -> "nonante" |100 -> "cent" | 1000 -> "mille"
        | n -> let u = n mod 10  and d = n / 10 mod 10
                and c = n / 100 mod 10 and m = n / 1000 mod 10 in
        (littbis m (litt lg 1000)) ^ (littbis c (litt lg 100)) ^
        (if (d = 7 or d = 9) && lg = `f`
                then (litt lg (10*(d-1))) ^ (if d = 7 & u = 1 then " et " else " ") ^ (litt lg (u+10))
        else (litt lg (10*d)) ^ (if u = 1 & d <> 8 then " et " else  (" ")) ^ (litt lg u))

where littbis x s = if x = 0 then "" else if x = 1 then s ^ " " else (litt lg x) ^ " " ^ s ^ " ";;
```

```
litt `b` 274;; ☞ "deux cent septante quatre"
litt `b` 7495;; ☞ "sept mille quatre cent nonante cinq"
litt `b` 495;; ☞ "quatre cent nonante cinq"
litt `f` 9481;; ☞ "neuf mille quatre cent quatre-vingt un"
litt `s` 782;; ☞ "sept cent huitante deux"
litt `f` 400;; ☞ "quatre cent  "
litt `f` 41;; ☞ "quarante et un"
```

10. Masse des mots

En attribuant à chaque lettre de l'alphabet son rang (a = 1, ..., z = 26) et sans tenir compte des accents, on va nommer masse d'un mot, la somme des rangs de ses lettres (minuscules).
Ainsi m(chef) = 3 + 8 + 5 + 6 = 22 et m(otorhinolaryngologistes) = 312
Cherchons alors les nombres dont l'écriture littérale a la même masse :
En rappelant que "int_of_char " délivre le code Ascii, la lettre a ayant le code 97 et l'espace, de code 32, ne comptant pas, nous pouvons définir le rang par :

```
let rang x = if x = ` ` or x = `-` then 0 else int_of_char x - 96;;

let rec explose = fun "" -> [] | x -> (nth_char x 0)::(explose (lasuite x))
                        where lasuite c = sub_string c 1 ((string_length c) - 1);;
let masse m = list_it (fun x y -> (rang x) + y) (explose m) 0;;
```

Nous construisons alors une fonction "recherche" à un argument n qui nous renvoie le premier entier à partir de n satisfaisant au but anoncé. Il semblerait qu'après 258, de tels nombres soient rares, par ailleurs, la fonction "litt" n'est valable que jusqu'à 9999, il faut donc l'étendre.

Voir : *Mots en forme,* E.Angelini D.Lehman, éd. Quintette 2001

```
let rec recherche n = if n = masse (litt `f` n) then n else recherche (n + 1);;
```

```
masse "otorhinolaryngologistes";; ☞ 312
```

```
recherche 4;; ☞ 222
recherche 223;; ☞ 232
recherche 233;; ☞ 258 (* Et cette dédicace qui fait 2^9 : *)
masse "a cette belle anne-christelle marie sophie exceptionnelle";; ☞ 512
```

11. Horloge

On souhaite réaliser la transcription littérale de l'heure en adoptant le parler usuel (cinq heures moins vingt ; midi et quart ; etc ...).

```
let arrondi x = if x >=. 0. then int_of_float(x +. 0.5) else -(int_of_float (-. x +. 0.5));;
let rec nth n q = if n = 1 then hd q else nth (n-1) (tl q);; (* définition à partir du rang 1 *)

let tm = ["cinq"; "dix"; "quart"; "vingt"; "vingt-cinq"; "et demie"]
and th = ["une"; "deux"; "trois"; "quatre"; "cinq"; "six"; "sept"; "huit"; "neuf"; "dix";"onze"; "midi "];;

let minute m = (* produit les chaînes "moins dix" , "quart", etc ... suivant la valeur m des minutes,
en arrondissant de cinq en cinq minutes. Lorsque la demie est passée, on fait bien sûr la
soustraction avec 60 minutes. *)
	let n = arrondi ((float_of_int m) /. 5.) in
	if n = 0 or n = 12 then ""
	else let cop = if n = 3 then "et " else if n = 9 then "le " else ""
		(* cas de "et quart" ou "moins le quart" *) in
	if n < 7 then cop ^ (nth n tm) else "moins " ^ cop ^ (nth (12 - n) tm);;

let heure h = (* produit les chaînes "midi",  "dix heures", etc ... en prenant garde au pluriel, et au
fait que "midi" et "minuit" ne sont pas suivis de "heures" *)
	if h = 0 or h = 24 then "minuit "
	else if h > 12    then nth (h - 12) th
	else nth h th;;
let heurebis h s =
	s ^ (if (h mod 12) <> 0 then if (h <> 1) && (h <> 13) then " heures " else " heure " else "");;

let ampm h = (* produit le complément "du matin", "du soir",  etc *)
	if 1 < h && h < 12 then " du matin"
	else if 12 < h && h < 17 then " de l'après-midi"
	else if 16 < h && h < 23 then " du soir"
	else "";;

let horloge h m =
	if m > 33  then (heurebis (h + 1) (heure (h+1))) ^ (minute m) ^ (ampm (h + 1) )
	else (heurebis h (heure h)) ^ (minute m) ^ (ampm h);;
```

```
horloge 0 12;; ☞ "minuit dix"
horloge 0 27;; ☞ "minuit vingt-cinq"
horloge 4 52;; ☞"cinq heures moins dix du matin"
horloge 8 17;; ☞ "huit heures  et quart du matin"
horloge 10 47;; ☞ "onze heures moins le quart du matin"
horloge 11 55;; ☞ "midi moins cinq"
horloge 12 32;; ☞ "midi et demie"
horloge 12 49;; ☞ "une heure moins dix de l'après-midi"
horloge 15 7;; ☞ "trois heures cinq de l'après-midi"
horloge 17 36;; ☞ "six heures moins vingt-cinq du soir"
horloge 23 46;; ☞"minuit moins le quart"
```

CHAPITRE VI

AFFECTATION, ITERATION

Affectation

Nous abordons maintenant une programmation "altérante", c'est à dire moins purement fonctionnelle. Il s'agit des "variables" ou plutôt des "références" que l'on va modifier, puis des vecteurs et tableaux dans le chapitre suivant. Il n'y a pas à proprement parlé de variables en Caml, car "let" définit des constantes. Pour avoir de vraies variables comme dans les langages impératifs classiques, il faut utiliser la fonction "ref" qui a pour "effet de bord" d'aller attribuer une place précise à cette variable dont la valeur sera accessible par la fonction ! et modifiée par l'opérateur := (ou <- pour les tableaux ou les éléments "mutables" d'un article).
On décrit ici une session où x est une référence, sa valeur est donc accessible par !x. La fonction "decr" pour décrémentation, ne renvoie rien, mais modifie son argument.

```
let x = ref 4;;
☞  x : int ref = ref 4

x;;
☞ int ref = ref 4

!x;;
☞ int = 4       (* le ! est l'opérateur valeur équivaut au y^ du pascal *)

decr x;;
☞ unit = ()
(* incr et decr sont l'incrémentation et la décrémentation pour un argument ref d'entier *)

x;;
☞ int ref = ref 3

let y = ref x;;
☞  y : int ref ref = ref (ref 3)  (* définit y comme un pointeur sur x *)
```

Itération

Il n'y a que deux primitives pour décrire des "boucles", ce sont "for...=... to (ou bien downto) ... do... done", et "while... do... done" (noter l'affectation = dans le "for")

```
for i = 1 to 10 do print_int i; print_string " " done ;;
☞ 1 2 3 4 5 6 7 8 9 10  unit = ()
```

1. Exemple de la factorielle

La définition itérative normale dans l'ancienne version de ML, suivie de deux versions en Caml (références obligatoires) est :

```
let fac n = let c = n and r = 1 in while c > 0 do r, c = c * r, c - 1 done; r;; (* version ML *)
let fac n = let c = ref 1 in for i = 1 to n do c := i * !c done; !c;;
let fac n = let c = ref n and r = ref 1 in while !c > 0 do r := !c * !r ; decr c done; !r;;
```

```
fac 7;; ☞ int = 5040
```

2. Recherche des solutions entières de l'équation $x^2 - y^2 = a$

En cherchant les facteurs (x + y) et (x - y) de même parité, a étant une constante fixée. Exemple pour $x^2 - y^2 = 45$; si x - y vaut successivement 1, 3, 5, et x + y, respectivement 45, 15, 9, alors x admet pour valeurs 23, 9, 7, avec respectivement y, 22, 6, 2.

Si $x - y = p$, $x + y = q$, pour que $x, y \in N$, il faut absolument que p et q aient la même parité, c'est à dire que p + q soit pair, ce qui est équivalent. D'autre part, pour éviter les répétitions on se limite aux couples (p, q) de diviseurs de a tels que $1 \leq p \leq \sqrt{a} \leq q \leq a$

```
let arrondi x = if x >=. 0. then int_of_float(x +. 0.5) else -(int_of_float (-. x +. 0.5));;

let reso p q = (* Résoud le système x - y = p , x + y = q *)
        print_string " /  "; print_int ((p + q) / 2); print_string ", "; print_int ( (q - p) / 2);;

let divise m n = (* Indique si m divise n ou non *)
        (n mod m) = 0;;

let impair n = not (divise 2 n);;

let dioph a =
        for p = 1 to arrondi (sqrt (float_of_int a)) do
                if divise p a then  let q = a /p in if impair (p) = impair (q) then reso p q
        done;;
```

```
dioph 45;; ☞ /  23, 22 /  9, 6 /  7, 2
dioph 145;; ☞ /  73, 72 /  17, 12
dioph 512;; ☞ /  129, 127 /  66, 62 /  36, 28 /  24, 8
```

3. Suite de Fibonacci

C'est encore la suite telle que chaque terme est égal à la somme des deux termes précédents, en partant de 1. Les premiers termes sont donc 1 1 2 3 5 8 13 21 ...
Cette suite est déterminée par : $u_0 = u_1 = 1$ puis $u_n = u_{n-1} + u_{n-2}$ elle est presque géométrique pour les termes de rang élevés. Pour ne pas employer trop de variables, (il serait évidemment maladroit de conserver toutes les valeurs successives dans un tableau), on peut se servir simplement de trois variables p, q, r représentant toujours trois termes consécutifs).
C'est évidemment bien moins joli que des appels récursifs.

```
let fibo n =
        let p = ref 1 and q = ref 1 and r = ref 1 in
        for i = 2 to n do r := !p + !q; p := !q; q := !r done; !r ;;
```

```
fibo 5;; ☞ 8
fibo 8;; ☞ 34
fibo 20;; ☞ 10946
```

4. Couple parfait

On recherche les couples de carrés parfaits non nuls dont la juxtaposition constitue un carré parfait (inférieur à 5000), ainsi par exemple 324 et 9 sont les carrés de 18 et 3 tels que 3249 en est également un, celui de 57.

```
let m = 100;;
let round x = if x >=. 0. then int_of_float(x +. 0.5) else -(int_of_float (-. x +. 0.5)) and sqr x = x*x;;
let nbchif x = 1 + int_of_float ((log (float_of_int x)) /. log 10.);;
        (* donne le nombre de chiffres de l'entier x *)
```

On peut alors donner l'ordre suivant :

```
for i = 1 to m do for j = 1 to m do
        let i2 = sqr i and j2 = sqr j in let c = ref i2  in
                for q = 1 to (nbchif j2) do c := 10*(!c) done;
        let r  =  sqrt (float_of_int (!c + j2)) in
        if abs_float(r  -. float_of_int (round r) )< 0.001 & !c + j2 < 5000
        then (print_string "("; print_int i; print_string ", "; print_int j;
        print_string ") de carrés ("; print_int i2; print_string ", "; print_int j2;
        print_string ") carré de ";  print_float r; print_newline())
done done;;

☞
(1, 15) de carrés (1, 225) carré de 35.0
(2, 3) de carrés (4, 9) carré de 7.0
(2, 15) de carrés (4, 225) carré de 65.0
(2, 30) de carrés (4, 900) carré de 70.0
(4, 3) de carrés (16, 9) carré de 13.0
(4, 9) de carrés (16, 81) carré de 41.0
(6, 1) de carrés (36, 1) carré de 19.0
(12, 2) de carrés (144, 4) carré de 38.0
(18, 3) de carrés (324, 9) carré de 57.0
```

5. Problème des 3 briques

Il s'agit de de placer 3 briques adjacentes à 3 côtés d'une quatrième de façon à ce que les dimensions des faces qui se correspondent soit les mêmes, mais chacune des 4 sont des parallélépipèdes de dimensions distinctes entières dont la somme des carrés est égale au triple de leur produit. On cherche le plus petit assemblage vérifiant ces propriétés par un bel emboîtement de boucles.
Soient x, y, z les dimensions entières de la première, les autres ont donc respectivement pour dimensions (u, y, z), (x, v, z), (x, y, w).

```
let m = 1000;;
let rec maxi = fun [] -> 0 | (x :: q) -> max x (maxi q) ;;
let pr x = print_int x; print_string " ";;

for x = 1 to m do for y = x+1 to m do for z = y+1 to m do
        if (sqr x) + (sqr y) + (sqr z) = 3*x*y*z then for u = 1 to m do
        if not (mem u [x; y; z]) & (sqr u) + (sqr y) + (sqr z) = 3*u*y*z then for v = 1 to m do
        if not (mem v [u; x; y; z]) & (sqr x) + (sqr v) + (sqr z) = 3*x*v*z then for w = 1 to m do
        if not (mem w [u; v; x; y; z]) & (sqr x) + (sqr y) + (sqr w) = 3*x*y*w then
        (pr x; pr y; pr z; pr u; pr v; pr w; print_string " volumes ";
        pr (x*y*z); pr (u*y*z); pr (x*v*z); pr (x*y*w);
        print_string " max "; pr (maxi [x*y*z; u*y*z; x*v*z; x*y*w]); print_newline())
done done done done done done;;
```

```
☞       1 5 13 194 34 2  volumes 65 12610 442 10 max 12610
        2 5 29 433 169 1  volumes 290 62785 9802 10 max 62785
```

6. Exclusion réciproque de deux suites

Soient a et b deux suites strictement croissantes d'entiers, le problème est d'énumérer de façon croissante les éléments de a non dans b ou de b non dans a. On le fait en tenant les indices courants i et j respectivement pour les deux suites et k pour celle que l'on construit.

```
let pr n = print_int n; print_string " ";;
let diff a b n = let i = ref 0 and j = ref 0 and u = ref 0 in
        for k = 0 to n do while a !i = b !j do incr i; incr j done;
                if a !i < b !j then (pr (a !i); incr i) else (pr (b !j); incr j) done;;

let pair k = 2 * k and mul3 k = 3 * k;;
```

Alors la demande "diff pair mul3 12;;" ou bien sans les définir, va donner les n + 1 premiers éléments dans l'ordre, de la première suite (les entiers pairs) qui ne sont pas multiples de 3 et des multiples de 3 non pairs.

```
diff (fun n -> 2*n) (fun n -> 3*n) 12;; ☞ 2 3 4 8 9 10 14 15 16 20 21 22 26
```

7. Suite partitionnant les entiers avec ses différences

On considère la suite croissante d'entiers telle qu'avec la suite formée par toutes les différences de termes consécutifs de cette suite, les deux suites ainsi formées constituent une partition des entiers.

Le début est nécéssairement 1 (+ 2 =) 3 (+ 4 =) 7 (+ 5 =) 12 (+ 6 =) 18 (+ 8 =) etc ...
Ce n'est évidemment pas la seule solution, mais on illustre ici les références en nommant n le terme courant, d la différence avec le terme suivant, et s l'ensemble des termes de la première suite jusqu'à n. Les éléments de la suite sont stockés dans une liste "mutable" s. Pour analyser le programme, le mieux est de le faire tourner à la main sur le papier pour m = 64 par exemple. Si n+ d n'est pas déjà dans l'ensemble s, on l'y place, on affiche une indication et on cherche la plus petite différence d suivante qui n'est pas dans s.

```
let suite m = (* affiche les termes jusqu'à m *)
    let n = ref 1 and d = ref 2 and s = ref [1] (* initialisation des premiers termes *)
    in print_int !n;
    while !n < m do
        if mem (!n + !d) !s then incr d
        else    (n := !n + !d; s := !n :: !s;
                print_string " (+ "; print_int !d; print_string " ) --> "; print_int !n; incr d;
                        while mem !d !s do incr d done)
    done;;
```

```
suite 64;; ☞
1 (+ 2 ) --> 3 (+ 4 ) --> 7 (+ 5 ) -->12 (+ 6 ) --> 18 (+ 8 ) --> 26 (+ 9 ) --> 35 (+ 10 ) --> 45 (+
11 ) --> 56 (+ 13 ) --> 69- : unit = ()
```

8. Exemple de conversion de chaîne en valeur numérique

Considérons une suite d'entiers dont chaque terme est la somme des carrés des chiffres qui composent le terme précédent dans la suite. On démontre qu'une telle suite est toujours périodique et qu'il n'y a que deux périodes possibles.

Traîter les termes de la suite comme des chaînes de caractères, est particulièrement adapté à ce problème. Chaque terme n est converti en la chaîne de ses chiffres ch grâce à la fonction prédéfinie "string_of_int", puis chacun des chiffres, ayant une valeur x, on en tire le carré qui s'ajoute au compteur m, lequel va constituer le terme suivant, d'où la fonction "suite". Bien sûr la boucle "for" peut avantageusement être remplacée par une définition récursive.

```
let sqr n = n*n;;

let rec suite n = print_int n; print_string "  "; (* Le nombre est écrit suivi d'un séparateur *)
        if n = 1 or n = 4 then print_string " fini "
        else (let m = ref 0 and ch = string_of_int n in
                for i = 0 to string_length ch - 1 do
                                m := !m + sqr (int_of_string (sub_string ch i 1)) done;
                suite !m);;
```

```
string_of_int 4568 ☞ string = "4568"
int_of_string "421";; ☞  int = 421
suite 7;; ☞ 7  49  97  130  10  1   fini - : unit = ()
suite 452317;; ☞ 452317  104  17  50  25  29  85  89  145  42  20  4   fini - : unit = ()
```

9. Calcul du poids

Il s'agit d'appliquer deux formules donnant le poids en kg, en fonction de la taille t en cm et de l'âge a en années par (3*t - 250) * (a + 270) / 1200 pour les hommes et (t / 2 - 30) * (180 + a) / 200 (femmes). La fonction calcule le poids idéal (la variable "pi"), et mesure la différence avec le poids réel, le paramètre sf du sexe est "false" pour les hommes et "true" pour les femmes.

```
let round x = if x >=. 0. then int_of_float(x +. 0.5) else -(int_of_float (-. x +. 0.5));;

let poids a t pr sf =
        let pi = ref (if  sf then (t /. 2. -. 30.)  *. (180. +. a) /. 200.
                else (3.*.t -. 250.) *. (a +. 270.) /. 1200.)
        in
        pi := float_of_int (round (10. *. !pi) /10) ;  (* astuce pour ne garder qu'une décimale *)
        let dif = pr  -. !pi in
        print_string (if abs_float dif < 3. then " Ca va, la différence est de "
                                else if dif < 0. then " Vous pouvez grossir car la différence est  "
                                else " Vous devez maigrir de ");
        dif;;
```

```
poids 15. 150. 80. true;; ☞ Vous devez maigrir de - : float = 37.0
poids 40. 181. 56. false;; ☞ Vous pouvez grossir car la différence est  - : float = -19.0
```

10. Combinaisons C_n^p

Afin de limiter les erreurs de calculs et d'aller au maximum des possibilités en capacité, on peut veiller à alterner les opérations de façon à toujours maintenir un résultat partiel entier au sein de la boucle.

$$C_n^p = \frac{n*(n-1)*(n-2)*(n-3)* \dots *(n-p+1)}{2*3*4* \dots *p}$$

L'astuce consiste à alterner les multiplications et les divisions, en commençant par la multiplication de deux entiers consécutifs n et n-1. En effet, parmi eux il y en a necéssairement un qui est pair, donc on peut immédiatement après diviser par 2, puis en multipliant par l'entier suivant n-3, des trois facteurs, là aussi, un des trois est multiple de 3, on peut donc aussitôt après diviser par 3.
On peut aussi débuter par un premier facteur égal à n-p+1, et un premier diviseur d égal à 1, et en continuant ainsi, le dernier facteur sera n et le dernier diviseur sera p. Cette solution permet de repousser le plus loin possible l'instant où la capacité en nombre entier sera atteinte.

```
let comb n p = let c = ref 1 in for d = 1 to p do c :=  !c * (n - p + d) / d done;
                !c;; (* le contenu de c est renvoyé *)
```

```
comb 7 3;; ☞ 35
comb 6 4;; ☞ 15
comb 8 5;; ☞ 56
```

11. Fonction logarithme néperien

On veut la réaliser la fonction "ln", sans avoir recours à celle qui est déjà implantée. Pour cela,on peut montrer que si x > 0, alors x s'écrit de façon unique sous la forme $2^k y\sqrt{2}$ avec k ∈ Z et un réel y ∈ [1/√2, √2[, pour cet y, en posant u = (y - 1) / (y + 1), on a ln(y) = $\Sigma\ 2u^{2i+1} / (2i+1)$, série, pour i = 0 jusqu'à l'infini.
Pour tout x positif, x= $2^k z$ avec k unique dans Z, il suffit de poser z = y√2 d'où y ∈ [1/√2, √2[, on montre alors y = (1 + u) / (1 - u) et on vérifie que -1 < u < 1 donc :
ln(y) = ln(1 + u) - ln(1 - u) = $\sum 2u^{2i+1} / (2i+1)$ pour i entier.

La fonction "decompose" va valculer ce couple de réels k et y, pour un x > 0, puis la fonction "serie" va donner la somme partielle pour i < 7 et y ∈ [1/√2, √2[, et enfin la fonction "lnapprox" qui pour x > 0, donne une valeur approchée de ln(x).

```
let decompose x =
	let k = ref 0. and p = ref 1. in (* p représente 2 puissance k *)
	if x >= 2. then while x >= 2. *. !p do p := 2. *. !p; k := !k +. 1. done
	else if x < 1. then while !p > x do p := !p /. 2.; k := !k -. 1. done;
	(!k, x  /. (!p *. sqrt(2.)) );;

let serie y = (* s est la somme partielle, v est u^2i *)
	let u = (y -. 1.) /. (y  +.1.) and  s = ref 1. in let v  = ref (u*.u) in
	for i = 1 to 6 do  s := !s +. !v /. (float_of_int (2*i +1)); v := !v *. !v done;
	2. *. u *. !s ;;

let lnapprox x =
	let k, y = decompose x in (k +. 0.5) *. 0.69 +. serie y;;
```

```
lnapprox 0.36;; ☞ -1.01693047669
lnapprox 2.718;; ☞ 0.995175544889
lnapprox 512.;; ☞ 6.20842765271
```

CHAPITRE VII

TABLEAUX

Vecteurs

Les fonctions prédéfinies et la syntaxe des vecteurs est donnée par l'exemple :

```
let v = [|1; 2; 3; 4|];;            ☞ v : int vect = [|1; 2; 3; 4|]
v.(3);;                             ☞ int = 4       (* équivalent à vect_item v 3;; *)
v.(1) <- 5;;                        ☞ unit = ()     (* équivalent à vect_assign v 1 5;; *)
v;;                                 ☞ int vect = [|1; 5; 3; 4|]
        (* les vecteurs repérés à partir de 0 *)
vect_length v;;                     ☞ int = 4       (* dimension d'un vecteur *)
map_vect succ v;;                   ☞ int vect = [|2; 6; 4; 5|] (* équivalent du map _list *)
list_of_vect v;;            (* transformation de type vecteur en type liste *)
        ☞ int list = [1; 5; 3; 4]
vect_of_list [`x`; `y`; `z`];;      (* transformation inverse *)
        ☞ char vect = [|`x`; `y`; `z`|]
sub_vect [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|] 3 2;;
        ☞ int vect = [|3; 4|]
make_vect 4  "azerty";;
        ☞ string vect = [|"azerty"; "azerty"; "azerty"; "azerty"|]
make_matrix 2 3 `a`;;
        ☞ char vect vect = [|[|`a`; `a`; `a`|]; [|`a`; `a`; `a`|]|]
```

1. Problème de Dijkstra

Un tableau ne contient que trois sortes d'éléments (on simplifiera avec 1, 2, 3) et il s'agit de le ranger dans l'ordre sans faire appel à un algorithme de tri, mais en faisant un seul balayage du tableau à l'aide de trois indices. On considère pour cela, qu'en cours d'exécution, les éléments du tableau sont tous des 1 entre les indices 0 et p - 1, qu'ils sont inconnus entre les indices p et q. Ce sont des 2 entre q + 1 et r - 1 et des 3 entre r et n - 1, si n est la taille du tableau.

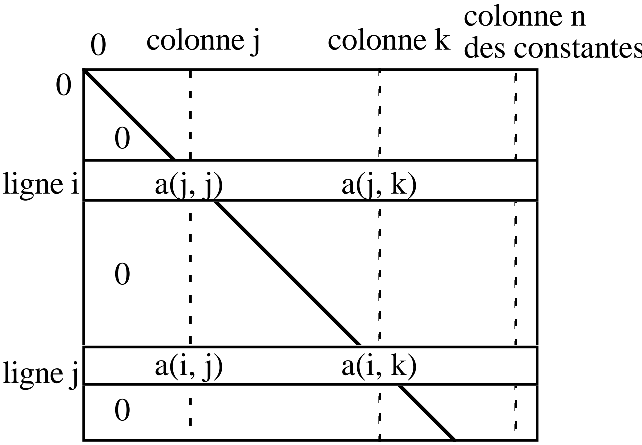
```
let range t = let n = vect_length t and p = ref 0 in let q = ref (n -1 ) and r = ref n in
        while !p <= !q do match t.(!p) with
                | 1 -> incr p
                | 2 -> t.(!p) <- t.(!q); t.(!q) <- 2; decr q
                | 3 -> t.(!p) <- t.(!q); decr r; t.(!r) <- 3; decr q; if !q + 1 < !r then t.(!q + 1) <- 2 done;
t;;
```

```
range [|3;2;2;1;3;3;2;2;2;1;1;1;2;2;2;3;1;2;1;1;2;3;3;3;2;1;2;2;1;1;1|];;
☞ [|1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 3; 3; 3; 3; 3; 3; 3|]
```

2. Exemple de la triangulation des matrices par la méthode de Gauss

L'algorithme classique de triangulation pour une matrice a de n lignes et p colonnes (ici p = n+1 pour un système ayant autant d'équations que d'inconnues) consiste pour chaque colonne j < n à (lignes numérotées de 0 à n-1) :

a) Chercher à partir de la diagonale pour j ≤ i < n la première ligne i où le cefficient a[i, j] est non nul (pivot, on peut aussi chercher la plus grande valeur absolue)
b) S'ils sont tous nuls, on passe à la colonne suivante.
c) Si i est différent de j, on échange les lignes i et j.
d) Diviser toute la ligne j par ce pivot (on a donc 1 sur la diagonale).
e) Remplacer chaque ligne au-dessous par une combinaison linéaire de cette ligne et de la ligne de référence j de façon à obtenir 0 dans la colonne j.



Prenons par exemple le système : $\begin{cases} 2x - 3y + z - t = -5 \\ -x - 2y + z + t = -3 \\ 3x + 2y - 5z + 3t = 4 \\ -7x + 3y + 2t = 7 \end{cases}$ soit à trianguler $A = \begin{bmatrix} 2 & -3 & 1 & -1 & -5 \\ -1 & 0 & -2 & 1 & -3 \\ 3 & 2 & -5 & 3 & 4 \\ -7 & 3 & 3 & 2 & 7 \end{bmatrix}$

```
let echange m i j =
        (* échange les lignes i et j dans la matrice m *)
        for k = 0 to vect_length m.(0) - 1 do
                let x = m.(i).(k) in m.(i).(k) <- m.(j).(k); m.(j).(k) <- x
        done;;

let normalise m i p =
        (* modifie la ligne i de la matrice m en divisant tout par le réeel p *)
        m.(i) <-  map_vect (fun x -> x /. p) m.(i);;

let combine m i j p =
        (* remplace dans la matrice m, la ligne i, par cette ligne moins p fois la ligne j *)
        for k = 0 to vect_length m.(0) - 1 do m.(i).(k) <-  m.(i).(k) -. p *. m.(j).(k) done;;

let rangpivot m j =
(* donne le n° de la ligne où se trouve le premier élément non nul de la colonne j à partir de la
ligne j, dans la matrice m. Par convention, ce rang sera -1 au cas où tous sont nuls. *)

        let i = ref j in while !i < vect_length m & m.(!i).(j) = 0. do incr i done;
        if m.(!i).(j) = 0. then -1 else !i;;
```

```
let a = [|[|2.; -3.; 1.; -1.; -5.|]; [|-1.; 0.; -2.; 1.; -3.|]; [|3.; 2.; -5.; 3.; 4.|]; [|-7.; 3.; 0.; 2.; 7.|] |];;

rangpivot a 1;;
☞ 2
```

```
let trig m = (* construit la matrice triangulaire mt à partir de la matrice m de n lignes et n+1
colonnes, et calcule le déterminant det du premier carré n*n de m. Attention un simple let mt = m
vmodifierai m *)
let dim = vect_length m - 1 and det = ref 1. in let mt = make_matrix (dim + 1) (dim + 2) 0. in
        for i = 0 to dim do for j = 0 to dim+1 do mt.(i).(j) <- m.(i).(j) done done;
        for j = 0 to dim - 1 do let i = rangpivot mt j in
                if i = -1 then det := 0.
                else (  if i <> j then (echange mt i j; det := -.(!det));
                        normalise mt j mt.(j).(j);
                        det := !det*.mt.(j).(j);
                        for i = j + 1 to dim do combine mt i j mt.(i).(j) done )
        done;
        det := !det*.mt.(dim).(dim); (!det, mt);;
```

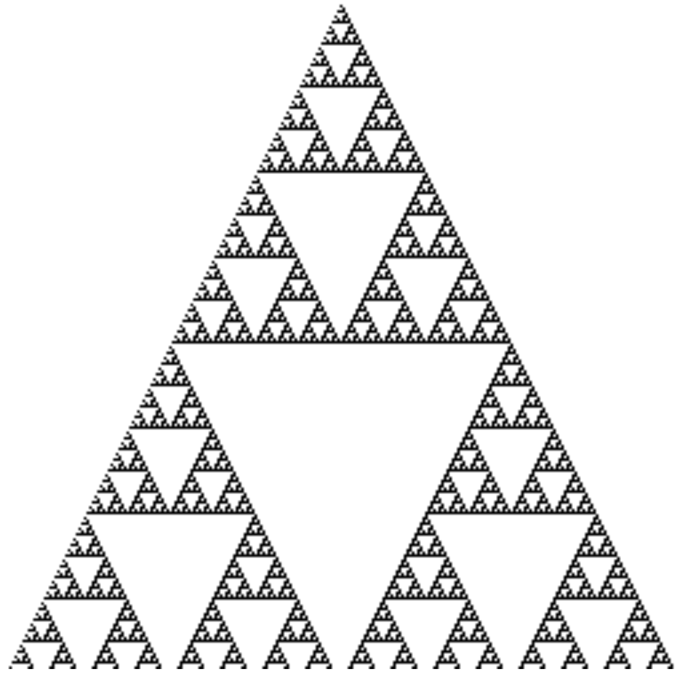
```
trig a;; ☞ 1.64102564103, [|  [|1.0; -1.5; 0.5; -0.5; -2.5|];
        (* déterminant de a *)  [|-0.0; 1.0; 1.0; -0.333333333333; 3.66666666667|];
                                [|-0.0; -0.0; 1.0; -0.512820512821; 0.948717948718|];
                                [|0.0; 0.0; 0.0; 1.64102564103; 6.5641025641|]|]
```

```
let reso m = (* renvoie si possible le vecteur solution du système triangulaire n*n *)
        let n = vect_length m in let x = make_vect n 0. in x.(n-1) <- m.(n-1).(n) /. m.(n-1).(n-1);
        for i = n - 2  downto 0 do x.(i) <- m.(i).(n);
                        for k = i + 1 to n - 1 do x.(i) <- x.(i) -. m.(i).(k)*.x.(k) done done; x;;
```

```
reso (snd (trig a));;
☞ [|1.0; 2.0; 3.0; 4.0|]
```

3. Le triangle de Pascal modulo 2

On sait que le triangle de Pascal énumère ligne par ligne les coefficients binômiaux et que sa disposition en pyramide est pratique en illustrant le fait que chaque coefficient est la somme des deux qui lui sont immédiatement supérieurs.
Une astuce de programmation est qu'à cause de la symétrie de ce triangle, on peut calculer les coefficients et les ranger dans un vecteur en effectuant ce calcul de droite à gauche. Plus précisémént, la ligne n du triangle doit contenir $C_n^0, C_n^1, ..., C_n^n$. Si le vecteur contient $C_{n-1}^0, ..., C_{n-1}^{n-1}$ alors, on rajoute à la fin $C_n^n = 1$ puis de $p = n - 1$ à $p = 1$ en descendant le coefficient $C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$ peut être calculé et immédiatement affecté dans le vecteur, cette nouvelle valeur ne venant pas perturber le calcul suivant.
On fait la même chose avec les restes dans la division par 2 de ces coefficients, et ils dessinent alors une étrange figure fractale (de dimension $\log_2 3$) que l'on fait point par point, en anticipant un peu (chapitre graphique).



```
let impair n = n mod 2 =1;;

#open "graphics";;

let point x y =  set_color black; moveto x y; lineto x y;;
```

```
let triangle dim = let v = make_vect (dim + 1) 0 in v.(0) <- 1; open_graph"";
        for n = 1 to dim do for p = n downto 1 do v.(p) <- (v.(p) + v.(p - 1)) mod 2;
                if impair v.(p) then point (p + (dim - n) / 2) (dim - n) done
        done;
        read_key(); close_graph();;
```

4. Jour de la semaine

La fonction "jour" donne le jour de la semaine grâce à la formule de Zeller : si le mois m est supérieur ou égal à 3 on le change en m-2 et sinon en m+10 ainsi que l'année a en a-1.
On pose alors na le numéro de l'année dans le siècle et s le numéro du siècle, et enfin :

$$f = j + na - 2*s + na \text{ div } 4 + s \text{ div } 4 + (26*m - 2) \text{ div } 10.$$

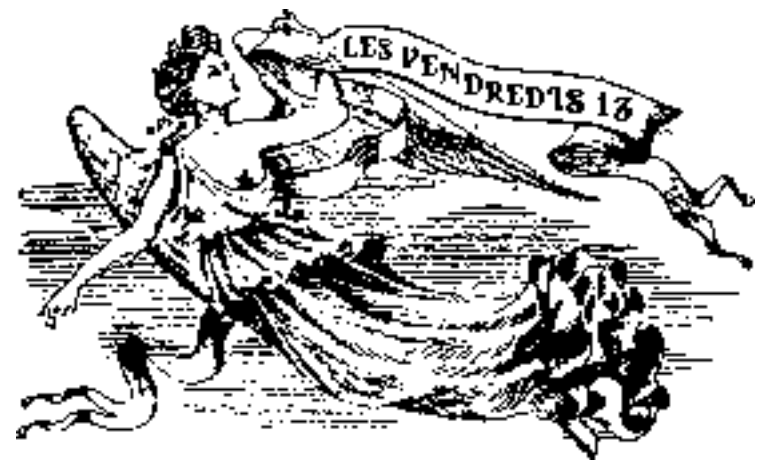
Le code est donné par f mod 7 (0 si dimanche).
Grégoire XIII a imposé sa réforme du calendrier le jeudi 4 octobre 1582 qui fut alors suivi du vendredi 15 octobre 1582. Avant ce changement, le programme doit également fonctionner en ajoutant 3 à f.
Dans beaucoup de langages de programmation, les fonctions partie entière, modulo et arctangente ne sont pas mathématiquement correctes, c'est pourquoi on redéfinit "reste". La fonction "jour" utilise le petit tableau immuable des jours.

```
let rec reste x d = if x < 0 then reste (x+d) d else if d <= x then reste (x-d) d else x;;

let jour j m a  = let mm, aa = if m > 3 then m - 2, a else m + 10, a - 1 in (* ici que des entiers *)
let s = aa / 100 and na = aa mod 100 in let f = j + na - 2*s + (na / 4) + (s / 4) + (26 * mm - 2) / 10
                + if a < 1582 or (a = 1582 & (m < 10 or m = 10 & j<5)) then 3 else 0
in [|"dimanche" ; "lundi"; "mardi"; "mercredi"; "jeudi"; "vendredi"; "samedi"|].(reste f 7);;
```

```
jour 4 10 1582;; ☞ "jeudi"
jour 1 1 2003;; ☞ "mercredi"
jour 14 7 1789;; ☞ "mardi"
jour 5 7 1968;; ☞ "vendredi"
jour 2 11 2006;; ☞ "jeudi"
```

5. Produire les vendredis 13

On utilise un réel tel que 2003.0613 pour désigner une date, ainsi la partie entière sera l'année 2003, les deux chiffres suivants donneront le mois, puis les deux suivants le jour.
On utilise ensuite simplement la fonction "jour" précédente pour énumérer les vendredis 13, grâce à une fonction "suivant" qui donne l'année et le mois où il y a un vendredi 13 après le mois m. Le typage impose un peu d'acrobaties pour manipuler à la fois des entiers et des réels.



```
let biss an = let a = round an in
        if a mod 4 <> 0 then false
        else if a mod 400 = 0 then  true
        else if a mod 100 = 0 then false
        else true ;;

let suivant m a  = match m with
        | 1 -> if biss a then a +. 0.045 else a +. 0.105
        | 2 -> if biss(a) then  a +. 0.085 else a +. 0.035
        | 3 -> a +. 0.115
        | 4 -> a +. 0.075
        | 5 -> a +. 1.015
        | 6 -> a +. 1.025
        | 7 -> if biss(a +. 1.) then a +.  1.065 else a +.  1.095
        | 8 -> if biss(a +. 1.) then a +.  1.105 else a +. 1.055
        | 9 ->  a +.  0.125
        | 10 -> if biss(a +. 1.) then   a  +. 1.095 else  a +.  1.045
        | 11 -> if biss(a +. 1.) then a  +. 1.055 else  a +. 1.085
        | 12 -> if biss(a +. 1.) then a  +. 1.035 else  a +. 1.065 ;;

let mois d = int_of_float (100.*.(d -. (float_of_int (int_of_float d))));;

let vendredis an = (* donne la liste entre 1900 et an *)
        let tm = [|" "; "janvier"; "février"; "mars"; "avril"; "mai"; "juin"; "juillet"; "août"; "septembre";
                "octobre"; "novembre"; "décembre"|] in let m = ref 10 and a = ref 1899. in
while !a <= an do let r = suivant !m !a in m := mois r; print_string ("     "^tm.(!m)^"  ");
                print_int (int_of_float r); a := float_of_int (int_of_float r); done;;
```

février	1970	mars	1970	novembre	1970	août	1971	octobre	1972
avril	1973	juillet	1973	septembre	1974	décembre	1974	juin	1975
février	1976	août	1976	mai	1977	janvier	1978	octobre	1978
avril	1979	juillet	1979	juin	1980	février	1981	mars	1981
novembre	1981	août	1982	mai	1983	janvier	1984	avril	1984
juillet	1984	septembre	1985	décembre	1985	juin	1986	février	1987
mars	1987	novembre	1987	mai	1988	janvier	1989	octobre	1989
avril	1990	juillet	1990	septembre	1991	décembre	1991	mars	1992
novembre	1992	août	1993	mai	1994	janvier	1995	octobre	1995
septembre	1996	décembre	1996	juin	1997	février	1998	mars	1998
novembre	1998	août	1999	octobre	2000	avril	2001	juillet	2001
septembre	2002	décembre	2002	juin	2003	février	2004	août	2004
mai	2005	janvier	2006	octobre	2006	avril	2007	juillet	2007
juin	2008	février	2009	mars	2009	novembre	2009	août	2010
mai	2011	janvier	2012	avril	2012	juillet	2012	septembre	2013
décembre	2013	juin	2014	février	2015	mars	2015	novembre	2015
mai	2016	janvier	2017	octobre	2017	avril	2018	juillet	2018
septembre	2019	décembre	2019	mars	2020	novembre	2020	août	2021

6. Jeu du démineur

Sur un quadrillage, chaque case est éventuellement minée, la seule donnée que l'on ait est le nombre de mines qui entoure chacune d'elles (une case possède 8 voisines hormis les cases du bord). Le but est de connaître avec précision le terrain, éventuellement de trouver des parcours pour faire un jeu. On va programmer au moyen de deux fonctions mutuellement récursives "poss" indiquant la possibilité d'arriver à une solution, et "avance".

Attention, $\begin{bmatrix} 3 & 4 & 3 \\ 4 & 6 & 4 \\ 3 & 4 & 3 \end{bmatrix}$ possède deux solutions $\begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$ et $\begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$

Remarque : la fonction (or A B) n'évalue pas B au cas où A est vrai. L'intérêt de cette programmation est qu'on peut faire du backtracking sans le savoir en disant qu'il est possible de

résoudre le problème en avancant de haut en bas et de gauche à droite dans le tableau avec un 0 ou avec un 1 (dernière ligne de "poss").

1	1	1	1	1	3	2	1		0	0	0	0	1	0	0	0
2	1	2	1	3	3	2	1		0	1	0	0	0	1	1	0
2	2	3	1	2	2	3	1		0	1	0	0	0	1	0	0
1	2	2	2	3	2	2	0		0	0	1	0	0	0	0	0
1	2	4	2	3	1	2	1		0	0	0	1	0	1	0	0
1	1	4	3	4	3	2	2		0	1	0	1	0	0	1	0
1	2	3	4	2	3	3	2		0	0	1	0	1	0	0	1
0	1	2	2	2	2	1	2		0	0	0	1	0	0	1	0

Le tableau des données, et le plan des mines à découvrir.

Avec deux fonctions mutuellement récursives mais, toutefois, un tableau m, ce qui simplifie les écritures, (définir m et s auparavant) on va écrire :

```
let valide i j = 0 <=  i & 0 <= j & i <  vect_length m &  j<  vect_length m;;
let v i j = if valide i j then s.(i).(j) else 0;;
        (* donne le nombre de mine (0 ou 1) en s (i, j) dans tous les cas *)

let compte i j = (*vérifie que le nombre de mines dans "s" autour de (i, j) est m(i, j) *) m.(i).(j)
= v (i-1) (j-1) + v (i-1) j + v (i-1) (j+1) + v i (j-1) + v i (j+1) + v (i+1) (j-1) + v (i+1) j + v (i+1) (j+1);;

let bonvoisin i j = (* test s(i, j) cohérent avec les 3 voisins antérieurs suivant l'ordre de parcours *)
        ((not (valide (i - 1) (j - 1))) or (compte (i - 1) (j - 1))) &
        ((j <> vect_length m) or (not (valide (i - 1) j)) or (compte (i - 1) j)) &
        ((i <> vect_length m) or (not (valide i (j - 1))) or (compte i (j - 1)));;

let rec poss i j = (* teste s'il est possible d'arriver à une solution en affectant s(i, j) *)
        if j = vect_length m then poss (1+ i) 0 (* passage à la ligne suivante *)
        else if i = vect_length m then true (* fini *)
        else (avance 0 i j) or (avance 1 i j)
and avance b i j = (* place b en s(i, j) et teste si cela peut conduire à une solution *)
        s.(i).(j) <- b; if bonvoisin i j then poss i (1 + j) else false;;
```

```
let m = [|       [|1; 1; 1; 1; 1; 3; 2; 1|];
                 [|2; 1; 2; 1; 3; 3; 2; 1|];
                 [|2; 2; 3; 1; 2; 2; 3; 1|];
                 [|1; 2; 2; 2; 3; 2; 2; 0|];
                 [|1; 2; 4; 2; 3; 1; 2; 1|];
                 [|1; 1; 4; 3; 4; 3; 2; 2|];
                 [|1; 2; 3; 4; 2; 3; 3; 2|];
                 [|0; 1; 2; 2; 2; 2; 1; 2|]  |] ;;       (* toujours le même exemple *)
let s = make_matrix  (vect_length m ) (vect_length m ) 0;; (* s sera la solution *)
poss 0 0;; -> true
s;;  ☞  [|       [|0; 0; 0; 0; 1; 0; 0; 0|];
                 [|0; 1; 0; 0; 0; 1; 1; 0|];
                 [|0; 1; 0; 0; 0; 1; 0; 0|];
                 [|0; 0; 1; 0; 0; 0; 0; 0|];
                 [|0; 0; 0; 1; 0; 1; 0; 0|];
                 [|0; 1; 0; 1; 0; 0; 1; 0|];
                 [|0; 0; 1; 0; 1; 0; 0; 1|];
                 [|0; 0; 0; 1; 0; 0; 1; 0|] |]
```

```
let m =  [|
          [|1;3;2;3;2;3;4;5;4;3;2;3;4;5;3|]
          [|1;4;2;5;4;6;6;7;5;5;4;6;6;8;5|];
          [|3;6;4;5;2;5;5;6;5;3;1;4;5;8;5|];
          [|4;5;3;3;2;5;5;5;5;2;4;5;6;8;5|];
          [|5;6;6;5;5;5;4;3;5;2;5;3;5;5;3|];
          [|5;5;4;1;2;2;4;4;6;3;6;3;6;4;3|];
          [|5;6;6;5;6;5;5;4;6;4;6;2;5;1;2|];
          [|3;4;3;2;2;2;3;4;5;3;5;2;6;2;3|];
          [|3;5;6;6;6;6;6;6;6;5;6;3;6;2;3|];
          [|1;1;3;4;5;5;5;5;4;3;3;2;5;2;3|];
          [|3;4;5;4;5;6;7;8;6;6;5;5;6;3;3|];
          [|3;4;5;5;5;6;6;8;5;5;3;4;4;2;2|];
          [|5;7;6;5;3;4;3;5;3;5;5;6;6;4;3|];
          [|5;8;8;8;6;6;5;6;5;6;6;5;4;1;1|];
          [|3;5;5;5;4;3;2;2;2;3;4;3;3;2;2|] |];;
```

```
poss 0 0;; ->  true
s;; -> [|  [|0; 0; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1|];
           [|1; 0; 1; 0; 0; 0; 1; 1; 1; 0; 0; 0; 1; 1; 1|];
           [|1; 0; 0; 0; 1; 0; 1; 1; 0; 0; 1; 0; 1; 1; 1|];
           [|1; 1; 1; 1; 1; 0; 1; 1; 0; 1; 0; 0; 1; 1; 1|];
           [|1; 1; 0; 0; 0; 0; 1; 1; 0; 1; 0; 1; 1; 1; 1|];
           [|1; 1; 0; 1; 1; 1; 0; 0; 0; 1; 0; 1; 0; 0; 0|];
           [|1; 1; 0; 0; 0; 0; 0; 1; 1; 1; 0; 1; 0; 1; 0|];
           [|1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 0; 1; 0; 1; 0|];
           [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 1; 0; 1; 0|];
           [|0; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 0; 1; 0|];
           [|0; 0; 0; 1; 1; 1; 1; 1; 1; 0; 0; 0; 0; 1; 0|];
           [|1; 1; 0; 0; 0; 0; 1; 1; 1; 0; 1; 1; 1; 1; 0|];
           [|1; 1; 1; 1; 1; 0; 1; 1; 1; 0; 1; 1; 0; 0; 0|];
           [|1; 1; 1; 1; 1; 0; 0; 0; 0; 0; 1; 1; 0; 1; 1|];
           [|1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 0; 0; 0|]|]
```

Remarque, une solution analogue mais avec des listes serait :
L'affectation s.(i).(j) <- b est modifiée en une écriture fonctionnelle où place b i j s renvoie une liste analogue à s. Il faut, de plus, quelques fonctions d'affichage, mais tout peut se faire de façon fonctionnelle, cependant, la vitesse d'éxécution est deux fois plus lente (affichage). De plus, les 14 lignes de programme passent à 23 lignes !

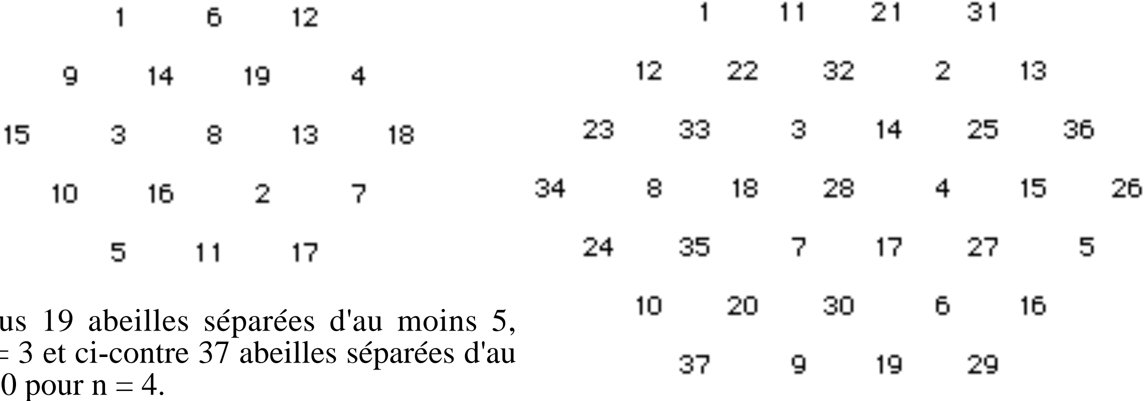
```
let valide i j m = 0 <=  i & 0 <= j & i <  list_length m & j<  list_length (hd m);;

let rec v (*  équivalent d'un m (i, j)  en donnant pour valeur 0 si coordonnées non valides *)
= fun  0 0 ((x :: rl) :: rt) -> x  | 0 j  ((_ :: rl) :: rt) -> v 0 (j-1) (rl :: rt)
        | i j (_ :: rt) -> v (i - 1) j rt | _ _  _ -> 0;;

let compte i j m s = (*vérifie que le nombre de mines dans s autour de (i, j) est m(i, j) *) v i j m =
v (i-1) (j-1) s + v (i-1) j s + v (i-1) (j+1) s +v i (j-1) s + v i (j+1) s +v (i+1) (j-1) s + v (i+1) j s +v (i+1)
(j+1) s ;;

let bonvoisin i j m s =
        (* teste si s(i, j) cohérent avec les 3 voisins antérieurs suivant l'ordre de parcours de m *)
        ((not (valide (i - 1) (j - 1) m)) or (compte (i - 1) (j - 1) m s)) &
        ((j <> list_length (hd m)) or (not (valide (i - 1) j m)) or (compte (i - 1) j m s)) &
        ((i <> list_length  m) or (not (valide i (j - 1) m)) or (compte i (j - 1) m s));;

let rajout = fun x [] -> [[x]] | x (pl :: rt) -> (x :: pl) :: rt ;;
        (* rajoute x en tête de la première sous-liste *)

let rec place b = fun 0 0 ((_ :: rpl) :: rt) -> ((b :: rpl) :: rt)
        | 0 j ((x :: rpl) :: rt) -> (rajout x  (place b 0 (j - 1) (rpl :: rt)) )
        | i j (pl :: rt) -> pl :: (place b (i - 1) j rt) ;; (* c'est l'inconvénient par rapport aux tableaux *)

let rec aff = fun [] -> print_string " voila " | ([] :: q) -> print_newline(); aff q
        | ((x :: rpl) :: rt) -> print_int x; print_string "  "; aff (rpl :: rt);;

let rec poss i j m s = (* teste s'il est possible d'arriver à une solution en placant s(i, j) *)
        if j = list_length (hd m) then poss (1+ i) 0 m s (* passage à la ligne suivante *)
        else if i = list_length m then (aff s; true) (* fini, une construction complète a pu être faite *)
        else (avance 0 i j m s) or (avance 1 i j m s) (* essai avec 0 ou avec 1 *)
and avance b i j m s = (* "place" b en s(i, j) et teste si cela peut conduire à une solution *)
        avancaveclebon b i j m (place b i j s)

and avancaveclebon b i j m s' = if bonvoisin i j m s' then poss i (1 + j) m s' else false;;

let demine m = poss 0 0 m m;; (* le second m sert de trame pour la construction du résultat s *)
```

```
demine [[2;3;3;2]; [1;2;4;2]; [2;2;2;1]];; ☞
0 0 1 1
1 1 0 1
0 0 0 0
 voila - : bool = true

demine [ [1; 1; 1; 1; 1; 3; 2; 1]; [2; 1; 2; 1; 3; 3; 2; 1]; [2; 2; 3; 1; 2; 2; 3; 1]; [1; 2; 2; 2; 3; 2;
2; 0]; [1; 2; 4; 2; 3; 1; 2; 1]; [1; 1; 4; 3; 4; 3; 2; 2]; [1; 2; 3; 4; 2; 3; 3; 2]; [0; 1; 2; 2; 2; 2; 1; 2]
] ;; ☞
0 0 0 0 1 0 0 0
0 1 0 0 0 1 1 0
0 1 0 0 0 1 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 1 0 0
0 1 0 1 0 0 1 0
0 0 1 0 1 0 0 1
0 0 0 1 0 0 1 0              voila - : bool = true

demine [[3;5;5;5;5;5;5;5;5;5;5;3]; [5;8;8;8;8;8;8;8;8;8;8;5]; [5;8;8;8;8;8;8;8;8;8;8;5];
[5;8;8;8;8;8;8;8;8;8;8;5]; [5;8;8;8;8;8;8;8;8;8;8;5]; [3;5;5;5;5;5;5;5;5;5;5;3]];; ☞

1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1     voila - : bool = true
```

Attention, $\begin{bmatrix} 3 & 4 & 3 \\ 4 & 6 & 4 \\ 3 & 4 & 3 \end{bmatrix}$ possède deux solutions $\begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$ et $\begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$

7. La ruche

On considère une ruche hexagonale dont les côtés sont tous formés de n alvéoles hexagonales. Il s'agit de placer un nombre na d'abeilles numérotées de telle sorte que deux abeilles voisines aient des numéros différant d'au moins une valeur dif. La difficulté, ici, est de trouver une représentation. Les tableaux carrés représentent des matrices, mais la nature a trouvé un optimum dans le pavage hexagonal, on va prendre un tableau rectangulaire, certes mal adapté, (hauteur h, largeur l) où 0 représentera les cases interdites dans les coins et -1, les cases non encore occupées.



Ci-dessus 19 abeilles séparées d'au moins 5, pour n = 3 et ci-contre 37 abeilles séparées d'au moins 10 pour n = 4.

Si n est toujours le côté de l'hexagone, na = 3n(n-1) + 1 est le nombre d'abeilles qu'on y loge. La traduction, presque exacte, montre encore certaines facilités comme l'initialisation des tableaux, et certains inconvénients comme l'indexation obligatoire à partir de 0 ce qui oblige à revoir

toutes les bornes. En particulier, la k-ième abeille porte le numéro k-1, et lorsqu'elle est placée, il faut affecter dans le tableau t , la valeur k+1.

```
let init n = (* renvoie l, h, na, le tableau t des numéros d'abeille, et "pris" *)
	let h = 2*n - 1 and l = 4*n - 3 and na = 3*n*(n - 1) + 1 in
	let t = make_matrix h l 0  and pris = make_vect na false in
	for i = 0 to n-1 do let j = ref (n- 1 - i) in
		while !j <= 3*n-3+i do t.(i).(!j) <- -1; j := !j + 2 done done;
	for i = n  to h-1 do let j = ref (i  + 1 - n) in
		while !j <= 5 * n - 5-i do  t.(i).(!j) <- -1; j := !j + 2 done
	done;
	(l, h, na, t, pris);;

let valide i j t l h = (* indique si la case t[i, j] existe et permise *)
	(0 <=  j) & (j < l) & (0 <= i) & (i < h) & (0 <> t.(i).(j));;

let bonvoisin k i j t l h dif =
	(* indique si k placé en t(i, j) respecte une différence > dif avec ses voisins *)
	(not (valide i (j - 2) t l h) or (dif < abs (k -  t.(i).(j - 2))))
	& (not (valide (i - 1) (j - 1) t l h) or (dif < abs (k -  t.(i - 1).(j - 1))))
	& (not (valide (i - 1) (j + 1) t l h) or (dif < abs (k - t.(i - 1).(j + 1))));;

let rec poss i j t l h pris na dif =
	(* vrai s'il est possible d'arriver à une solution en plaçant une abeille en t[i, j] *)
	if j >= l then poss (i + 1) 0 t l h pris na dif (* passage à la ligne suivante du tableau *)
	else if i >= h then true (* cas où on a remplit avec succès *)
	else if t.(i).(j) = 0 then poss i (j + 1) t l h pris na dif (* case interdite, on continue *)
	else	(let k = ref 0 and b = ref false in
		while (!k < na) & not(!b) do
		if not (pris.(!k)) & bonvoisin !k i j t l h dif
		then (t.(i).(j) <- !k+1; pris.(!k) <- true;
			b := poss i (j + 2) t l h pris na dif; if not(!b) then pris.(!k) <- false);
		incr k done;
		!b);;

let ruche n dif = let l, h, na, t, pris = init n in
	if not (poss 0 0 t l h pris na dif) then print_string ("impossible"); t;;
```

On donne ci-dessous quatre exemples, bien sûr la vue de la ruche hexagonale n'est pas excellente, l'achêvement du programme pour avoir une bonne présentation avec des alvéoles, n'est pas compliqué mais demanderait une sérieuse mise au point au sein d'une fenêtre graphique.

```
ruche 2 1;;☞[|  [|0; 1; 0; 5; 0|]; c'et à dire   1   5
                [|6; 0; 4; 0; 2|];              6   4   2
                [|0; 3; 0; 7; 0|] |]              3   7
```

```
ruche 3 5;;☞ [|[|0; 0; 1; 0; 13; 0; 7; 0; 0|];
              [|0; 14; 0; 8; 0; 2; 0; 15; 0|];
              [|9; 0; 3; 0; 16; 0; 10; 0; 4|];
              [|0; 17; 0; 11; 0; 5; 0;18; 0|];
              [|0; 0; 6; 0; 19; 0; 12; 0; 0|] |]
```

```
ruche 4 10;; ☞
[|	[|0; 0; 0; 1; 0; 13; 0; 2; 0; 24; 0; 0; 0|];
	[|0; 0; 15; 0; 37; 0; 26; 0; 14; 0; 3; 0; 0|];
	[|0; 5; 0; 27; 0; 16; 0; 4; 0; 29; 0; 18; 0|];
	[|28; 0; 17; 0; 6; 0; 30; 0; 19; 0; 8; 0; 31|];
	[|0; 7; 0; 33; 0; 20; 0; 9; 0; 32; 0; 21; 0|];
	[|0; 0; 23; 0; 10; 0; 35; 0; 22; 0; 11; 0; 0|];
	[|0; 0; 0; 36; 0; 25; 0; 12; 0; 34; 0; 0; 0|] |]
```

```
ruche 5 6;; ☞   [|
[|0; 0; 0; 0; 1; 0; 9; 0; 2; 0; 10; 0; 3; 0; 0; 0; 0|];
[|0; 0; 0; 11; 0; 19; 0; 27; 0; 18; 0; 26; 0; 12; 0; 0; 0|];
[|0; 0; 4; 0; 28; 0; 5; 0; 35; 0; 6; 0; 20; 0; 29; 0; 0|];
[|0; 13; 0; 21; 0; 14; 0; 22; 0; 15; 0; 30; 0; 7; 0; 16; 0|];
[|23; 0; 31; 0; 8; 0;32; 0;40; 0; 24; 0; 17; 0;25; 0; 33|];
[|0; 39; 0; 47; 0; 41; 0; 49; 0;57; 0;36; 0;52; 0; 42; 0|];
[|0; 0; 56; 0; 34; 0; 58; 0; 43; 0; 51; 0; 45; 0; 60; 0; 0|];
[|0; 0; 0; 50; 0; 44; 0; 37; 0; 61; 0; 38; 0; 54; 0; 0; 0|];
[|0; 0; 0; 0; 59; 0; 53; 0; 46; 0; 55; 0; 48; 0; 0; 0; 0|]|]
```

8. Réseau de neurones, algorithme de rétropropagation

Les réseaux de neurones artificiels datent de 1943 mais ont été très utilisés à partir de 1986 en reconnaissance des formes. Suite à la publication de l'algorithme de rétropropagation, l'un des modèles les plus séduisants et les plus utilisés, est celui du réseau à couches. Dans chacune des couches, chaque neurone est relié à ceux de la couche précédente dont il reçoit les informations (les entrées), et à chaque neurone de la couche suivante à qui il transmet une information : sa sortie, mais il n'est pas relié aux autres neurones de sa propre couche.
L'entrée e d'un neurone est la somme des sorties des neurones de la couche précédente, pondérées par les poids w. Par suite, le neurone produit s = f(e) qui sera la mesure de sa sortie portée par son "axone" (f est la fonction de transfert qui ramene à [-1, 1] ou à [0, 1] suivant les besoins de l'application. L'axone transmet ensuite cette valeur par l'intermédiaire de "synapses" aux dendrites d'autres neurones. (Les poids mesurent en fait l'efficacité des synapses.)
L'apprentissage du réseau, consiste en une modification de ces poids au fur et à mesure des expériences, c'est à dire de la confrontation entre le vecteur sortant de la dernière couche et celui qui est attendu en fonction d'un vecteur d'entrée.

Voir : Rumelhart D.E. Hinton G.E. Williams R.J. *Learning internal representations by error propagation.* Parallel distributed processing MIT Press p. 318-362, 1986

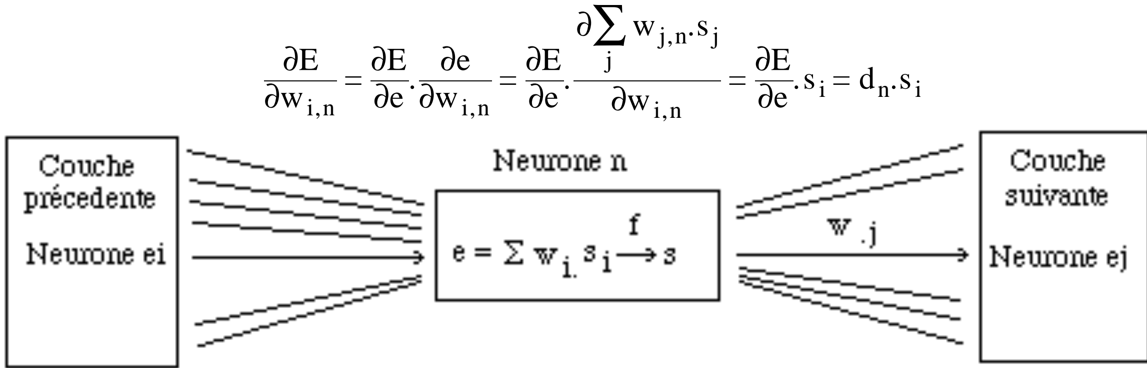
Dans un réseau multicouche, chaque neurone, à l'intérieur, ayant $e = \sum w_{i,n} s_i$ pour entrée, où i parcourt les neurones de la couche précédente, aura une sortie s = f(e). Lorsque l'on propose le vecteur $X = (x_1, x_2, ..., x_n)$, à la première couche, la dernière restitue le vecteur $S = (s_1, ..., s_p)$ alors qu'on attend $Y = (y_1, ..., y_p)$ comme réponse.
Le but de cet algorithme est d'exprimer l'erreur quadratique $E = \sum_{1 \leq i \leq p} (y_i - s_i)^2$ et de chercher à la minimiser en modifiant chaque poids w, suivant l'influence qu'il a sur E :

$$w(t+dt) - w(t) = -\mu . \partial E / \partial w$$

Dans cette équation de mise à jour, μ est un coefficient positif, le "pas" du gradient. En effet, si cette dérivée est nulle ou faible, cela veut dire que l'erreur E dépend peu de ce poids là w et donc qu'il n'y a pas lieu de le modifier. Si, elle est positive et importante, cela signifie au contraire que E croît avec w, donc on diminue w d'autant.

On cherche donc à exprimer ce gradient qui est le vecteur formé par tous les $\partial E / \partial w$
Plaçons nous entre le neurone i d'une couche et un neurone n fixé :
Pour la commodité de la lecture, on notera i l'indice parcourant la couche précédente, et j celui de la couche suivante, si d_n est la dérivée partielle de E par rapport à e, on a :

$$\frac{\partial E}{\partial w_{i,n}} = \frac{\partial E}{\partial e} . \frac{\partial e}{\partial w_{i,n}} = \frac{\partial E}{\partial e} . \frac{\partial \sum_j w_{j,n} . s_j}{\partial w_{i,n}} = \frac{\partial E}{\partial e} . s_i = d_n . s_i$$



Car tous les termes de la sommation pour $j \neq i$ ne dépendent pas de $w_{i,n}$.
Si le neurone n'est pas en sortie, alors on calcule :

$$d_n = \frac{\partial E}{\partial e} = \sum_{j \text{ suivant}} \frac{\partial E}{\partial e_j} . \frac{\partial e_j}{\partial e} = \sum d_j \frac{\partial e_j}{\partial s} . \frac{\partial s}{\partial e} = \sum d_j . w_{n,j} . f'(e)$$

Et si n est un neurone en sortie : $$d_n = \frac{\partial E}{\partial e} = \frac{\partial \sum_{j=1}^{p} (Y_j - S_j)^2}{\partial e} = 2(S_n - Y_n).f'(e)$$

La règle d'apprentissage est donc à chaque présentation d'exemple (X, Y), de mesurer la sortie S, l'erreur E, et de modifier chaque poids $w_{i,j}$ en le remplaçant par $w_{i,j} - \mu d_j s_i$ avec :
$d_n = (\sum d_j w_{n,j}) f'(e_n)$ pour les indices j en aval de n, sauf si n est en sortie auquel cas on a plutôt : $d_n = 2(s_n - Y_n) f'(e_n)$. Il y a rétro-propagation de l'erreur commise dans la mesure où les modifications vont devoir être effectuées de la dernière couche vers la première.
Ce processus est répété sur plusieurs exemples jusqu'à ce qu'il y ait convergence suivant un seuil fixé.

Le réseau a tendance à se conformer au dernier exemple présenté. C'est pourquoi on peut modifier la procédure d'apprentissage en présentant successivement tous les exemples avec une seule rétropropagation (par lot) à chaque fois et si l'erreur est supérieure au seuil au moins une fois au cours de ce balayage, ou bien en cumulant les erreurs, on refait un balayage complet. Le pas d'apprentissage μ permet de régler la vitesse de modification des poids mais il est très difficile à régler, la convergence et sa vitesse sont assez chaotiques.

Nous allons, dans cet exemple, modèliser une fonction de R^n vers R^p avec une seule couche cachée contenant k neurones. La fonction de transfert est une sigmoïde dans [0, 1] de pente assez faible de façon à ce que sa dérivée ne soit pas trop souvent approchée par 0, ce qui ne modifierait pas les poids, elle ne s'applique pas sur la couche de sortie. Ces poids sont aléatoirement initialisés dans [0, 1].

Nous allons désigner un tel réseau par des constantes et des vecteurs. La programmation est ici très impérative et assez facile. La fonction "apprend" permet de modifier les poids jusqu'à ce que l'erreur sur un exemple (x, y) soit inférieure à la valeur eps, par contre, la fonction "apprendlot" modifie les poids en rétropropageant sur chaque exemple, l'erreur cumulée sur un vecteur d'exemples xx avec leurs sorties attendues yy.

```
let a = 0.1 and n = 9 and p = 1 and k = 36;; (* pente a/4 en 0 pour f, ce 36 est assez arbitraire *)

let cachee = make_vect k 0. and sortie = make_vect p 0.
        and poids1 = make_matrix n k 0. and poids2 = make_matrix k p 0.;;

let init () = for i = 0 to n-1 do for j = 0 to k-1 do poids1.(i).(j) <- random__float 1. done done;
        for i = 0 to k-1 do for j = 0 to p-1 do poids2.(i).(j) <- random__float 1. done done;;

let rec transfert t = 1./.(exp(-.a * .t) +.1.)
and sqr t = t*.t
and  transfert' t = let e = exp(-.a * .t) in a*.e/.(sqr (1.+.e));; (* dérivée de "transfert *)

let propag x =
        (* x est un vecteur de dim n, le résultat est la sortie vecteur de dim p, modifie les états *)
        let sortiecachee = make_vect k 0. in for q = 0 to k-1 do cachee.(q) <- 0. done;
        for j = 0 to p-1 do sortie.(j) <- 0. done;
        for q = 0 to k-1 do for i = 0 to n-1 do
                cachee.(q) <- cachee.(q) +. poids1.(i).(q) *. (transfert x.(i)) done done;
        for q = 0 to k-1 do sortiecachee.(q) <- transfert cachee.(q) done;
        for j = 0 to p-1 do for q = 0 to k-1 do sortie.(j) <- sortie.(j) +. poids2.(q).(j) *.
sortiecachee.(q)
        done done;
        sortie;;

let erreur x y = let s = propag x and e = ref 0. in
        for j = 0 to p - 1 do e := !e +. (sqr (s.(j) -. y.(j))) done; (sqrt !e) /. 2.;;
```

```
let retro mu x y = (* ne renvoie rien, mais modifie les poids *)
	let d = make_vect p 0. and dw = make_vect k 0. in
	for j = 0 to p - 1 do d.(j) <-  2.*.(sortie.(j) -. y.(j))*.(transfert' sortie.(j))
	done;
	for q = 0 to k - 1 do for j = 0 to p - 1 do
		poids2.(q).(j) <- poids2.(q).(j) -. mu*.d.(j)*.(transfert cachee.(q));
		dw.(q) <- dw.(q) +. d.(j)*.poids2.(q).(j)
	done done;
	for i = 0 to n - 1 do for q = 0 to k - 1 do
		poids1.(i).(q)
			<- poids1.(i).(q) -. mu *.dw.(q) *.(transfert' cachee.(q)) *. (transfert x.(i))
	done done;;

let rec apprend x y mu eps = let e = erreur x y in print_float e; print_newline();
	if eps < e then (retro mu x y; apprend x y mu eps);;

let rec apprendlot xx yy mu eps = let ne = vect_length xx and e = ref 0. in
	for h = 0 to ne-1 do e := !e +. erreur xx.(h) yy.(h) done; print_float !e; print_newline();
	if eps < !e then (for h = 0 to ne - 1 do retro mu xx.(h) yy.(h) done;
	apprendlot xx yy mu eps);;
```

```
apprend [|1.; 0.; 1.; 0.; 1.; 0.; 1.; 0.; 1.|] [|1.; 0.; 0.; 0.|] 0.1 0.01;; (* exemple si p = 4 *)
```

Application à la reconnaissance de figures

On représente de petits carrés 3*3 par des vecteurs de 9 réels, et on choisit pour expérimenter trois petits dessins binaires : une barre horizontale en bas, la diagonale et une barre verticale à droite.

$$\begin{bmatrix}0&0&0\\0&0&0\\1&1&1\end{bmatrix} \to 0 \quad \begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix} \to 0{,}5 \quad \begin{bmatrix}0&0&1\\0&0&1\\0&0&1\end{bmatrix} \to 1$$

L'apprentissage est cependant extrèmement lent, après quoi, on teste le réseau de neurones sur des figures voisines.
On demandera :

```
apprendlot [| [|1.; 0.; 1.; 0.; 1.; 0.; 1.; 0.; 1.|]; [|1.; 1.; 1.; 1.; 0.; 1.; 1.; 1.; 1.|]; [|0.; 1.; 0.; 0.; 1.; 0.; 0.; 1.; 0.|]; [|0.; 0.; 0.; 1.; 1.; 1.; 0.; 0.; 1.|] |]  [| [|1.; 0.; 0.; 0.|]; [|0.; 1.; 0.; 0.|]; [|0.; 0.; 1.; 0.|]; [|0.; 0.; 0.; 1.|] |] 0.1 0.1;;

apprendlot [| [|0.; 0.; 0.; 0.; 0.; 0.; 1.; 1.; 1.|]; [|1.; 0.; 0.; 0.; 1.; 0.; 0.; 0.; 1.|]; [|0.; 0.; 1.; 0.; 0.; 1.; 0.; 0.; 1.|] |]  [| [|0.|]; [|0.5|]; [|1.|] |] 0.1 0.1;;
```

Une fois l'apprentissage terminé, par lot, sur ces quatre exemples, on vérifie grâce à "propag", que ceux-ci ont bien été appris, puis, on va tester sur des images telles qu'une barre en bas ou à gauche.

```
propag [|1.; 0.; 1.; 0.; 1.; 0.; 1.; 0.; 1.|] ;;        à comparer avec    [|1.; 0.; 0.; 0.|]
propag [|1.; 1.; 1.; 1.; 0.; 1.; 1.; 1.; 1.|] ;;                           [|0.; 1.; 0.; 0.|]
propag [|0.; 1.; 0.; 0.; 1.; 0.; 0.; 1.; 0.|] ;;                           [|0.; 0.; 1.; 0.|]
propag [|0.; 0.; 0.; 1.; 1.; 1.; 0.; 0.; 1.|] ;;                           [|0.; 0.; 0.; 1.|]
```

9. Décomposition d'un entier avec un choix de nombres (le compte est bon)

Cet exemple de programme de backtracking a déjà été vu, mais utilisant ici une exception. Les nombres donnés figurent dans un tableau, ceux que l'on choisit sont également mis dans un tableau, ce qui constitue donc une petite variante de programmation.

```
exception trouvé;;

let decomp s e = let n = vect_length(e) in let choisis = make_vect n false in
        let rec essai s = if s = 0 then raise (trouvé)
        else for i = 0 to n - 1 do
                        if not choisis.(i) & e.(i) <= s
                        then  (choisis.(i) <- true; essai (s - e.(i)); choisis.(i) <- false)
        done
        in try essai s; [||] with trouvé -> choisis;;
```

```
decomp 7 [|1; 5; 3; 2; 4|];;        ☞ bool vect = [|true; false; false; true; true|]
```

10. Problème général de l'exploration d'une arborescence

L'exemple précédent peut être rédigé d'une manière plus systématique et avec un esprit encore plus "impératif", en utilisant encore un vecteur "choix" contenant les choix possibles.
A chaque noeud de l'arbre, l'état e représente la branche y arrivant, une itération sur les éléments non "choisis" de "choix" permet successivement de construire les "fils" possibles de e. Ceux-ci sont ou non "valides".

Ainsi avec le même exemple :

```
let choix = [|1; 5; 3; 2; 4|] and s = 7;;

let rec vue e = rev (map (fun x -> print_string " "; print_int x) e); print_newline();;

let  valide e = (list_it (prefix +) e 0) <= s;;
        (* vrai si la somme des éléments de l'état e ne dépasse pas s *)

let fils e i = choix.(i)::e ;;
        (* premier fils à étudier après l'état e  au moyen du i-ième élément de "choix" *)

let final e = (it_list (prefix +) 0 e) = s;;

let exploarb init = let choisis = make_vect (vect_length choix ) false in essai init
        where rec essai e = if final e then vue e  else  for i = 0 to (vect_length choix  -1) do
        if not choisis.(i) then
                let f = fils e i in if valide f then (choisis.(i) <- true; essai f; choisis.(i) <- false) done;;
```

```
exploarb [] ;; ☞ 1 2 4 -  1 4 2 -  5 2 -  3 4 -  2 1 4 -  2 5 -  2 4 1 -  4 1 2 -  4 3 -  4 2 1
```

11. Application au problème du téléphone

Dans un esprit analogue, on veut composer un numéro de dix chiffres tous distincts tel que chaque groupe de deux chiffres à partir du premier soit supérieur à la somme de ceux qui le précede. On reprend le schéma général précédent du backtrack, en représentant un état par une liste d'entiers à l'envers (dernier choisi en premier) et ses fils par les entiers construits en lui adjoignant à gauche un chiffre non encore choisi.
La fonction "doublets" renvoie la liste inversée des nombres à deux chiffres composant l'argument comme liste d'entiers, "valide" vérifie que le premier groupe de deux chiffres est supérieur à la somme des autres doublets.

```
let pair n = (n = 2 * (n / 2));;
let rec doublets = fun [] -> [] | [x] -> [x] | (x :: y :: q) -> (x + 10*y) :: (doublets q);;
```

```
doublets [9;8;7;6;5;4;3;2;1;0];;☞ [89; 67; 45; 23; 1]
```

```
let rec vue e = rev (map (fun x -> print_string (if x < 10 then " 0" else " "); print_int x)
                        (doublets e)); print_newline();;
```

```
vue [9;8;7;6;5;4;3;2;1;0];; ☞ 01 23 45 67 89
```

```
let valide lc = let n = list_length lc in if not(pair n) or  n < 3 then true
        else match lc with  (x :: y :: q) -> (x + 10*y  > (it_list (prefix +) 0  (doublets q)));;
```

```
valide [9;8;5;6;2;3];; ☞ true
valide [2;0;5;3];; ☞ false
```

```
let choix = [|0; 1; 2; 3 ; 4; 5; 6; 7 ; 8; 9|] ;;

let fils e i = choix.(i) :: e;;
        (* premier fils à étudier après l'état e  au moyen du i-ième élément de "choix" *)

let final e = (list_length e = 10);;
```

```
exploarb [] ;; (* exploarb n'a pas changé, il donne tous les résultats : *)
 03 15 26 47 98 -  03 15 26 48 97 -  03 15 27 46 98 -  03 15 27 48 96 -  03 15 28 47 96 -
03 16 25 47 98 -  03 16 25 48 97 -  03 16 27 48 95 -  03 17 25 46 98 -  03 17 25 48 96 -
03 17 26 48 95 -  03 18 25 47 96 -  05 13 26 47 98 -  05 13 26 48 97 -  05 13 27 46 98 -
05 13 27 48 96 -  05 13 28 47 96 -  05 16 23 47 98 -  05 16 23 48 97 -  05 17 23 46 98 -
05 17 23 48 96 -  06 13 25 47 98 -  06 13 25 48 97 -  06 13 27 48 95 -  06 15 23 47 98 -
06 15 23 48 97 -  07 13 25 46 98 -  07 13 25 48 96 -  07 13 26 48 95 -  07 15 23 46 98 -
07 15 23 48 96 -  08 13 25 47 96
```

12. Application au problème des dominos

Pour n entier fixé, un jeu de domino est constitué de toutes les paires de numéros $\{x, y\}$ avec la condition $x, y \leq n$. Il y a donc $1 + 2 + 3 + ... + (n+1) = (n+1)(n+2) / 2$ pièces c'est-à-dire 28 pour le jeu normal avec $n = 6$.
On souhaite connaître toutes les boucles possibles formées avec des dominos, en particulier les plus longues.
Avec le même algorithme général, il suffit encore de réfléchir à la représentation des données : un domino sera évidemment un couple d'entiers, on prévoie une procédure "construc" pour engendrer les "paires".
La fonction "valide" ne fait ici réellement que vérifier que le fils qui lui est passé est acceptable, sa validité est vérifiée dans "fils", "final" renvoie vrai si on peut boucler la suite de dominos. Comme il y a énormément de solutions, on peut sélectionner les plus longues en filtrant les "vues" pour des longueurs supérieures à (par exemple 19 si $n = 5$, 20 si $n = 6$). Afin de limiter, on peut également imposer [(0, 0)] comme état initial en forçant "choisis.(0)" à "faux".

```
let n = 5;;

let choix = make_vect ((n+1)*(n+2)/2 )(0, 0);;

let construc m = let k = ref 0 in
        for i = 0 to m do for j = i to m do  choix.(!k) <- (i, j); incr k done done;;
        (* les dominos sont donc des couples rangés avec i ≤ j *)

let rec vue ld = rev (map (fun (x, y) -> print_int x; print_string "+"; print_int y; print_string "  ") ld);
        (* ld est une liste de dominos déjà rangés comme 1+4  4+2  2+3  3+0 ... *)
        print_newline();;
```

```
vue [4,0; 4,4; 0,4; 5,0; 3,5; 1,3; 0,1];; ☞ 0+1   1+3   3+5   5+0   0+4   4+4   4+0
```

```
let  valide ld = not( ld = []) and fils e i = let (x, y) = choix.(i) in if e = [] then [(x, y)]
        else let (u, v) = hd e in
                if x = v then (x, y) :: e else if y = v then (y, x) :: e else [];;
(* donne le premier fils à étudier après l'état e au moyen du i-ième élément de "choix" si celui-ci
ne convient pas, [] est renvoyé *)
```

```
let final ld = (list_length ld > 17) & fst (last ld) = snd (hd ld)
        where rec last = fun    | [x] -> x
                                | (_ :: q) -> last q;;
```

final [4,0; 4,4; 0,4; 5,0; 3,5; 1,3; 0,1];; ☞ true

Exemple (une des premières solutions de longueur 18) :

exploarb [] ;;
☞ 0+0 0+1 1+1 1+2 2+0 0+3 3+1 1+4 4+2 2+2 2+5 5+3 3+3 3+4 4+4 4+5 5+5 5+0

et avec les 28 dominos pour n = 6
☞ 0+0 0+1 1+1 1+2 2+0 0+3 3+1 1+4 4+0 0+5 5+1 1+6 6+2 2+2 2+3 3+3 3+4 4+2 2+5 5+3 3+6 6+4 4+4 4+5 5+5 5+6 6+6 6+0

13. Problème d'Euler du parcours du cavalier sur un échiquier

C'est le problème consistant, sur un échiquier de n cases sur n cases, partant de la case dx, dy (comptées à partir de 0), à parcourir tout l'échiquier (une seule fois chaque case) en diagonales de rectangles de 2 sur 3 à chaque fois comme le fait le cavalier sur un jeu d'échec.

```
let rec filtre p = fun [] -> [] | (x :: q) -> if p x then x :: (filtre p q) else filtre p q;;
        (* fonctionnelle déjà vue *)

let voisins i j n = filtre (fun (x, y) -> 0 <= x & x < n & 0 <= y & y < n)
        (* liste des au plus 8 coups possibles *)
                [(i-2, j-1); (i-2, j+1); (i-1, j-2); (i-1, j+2); (i+1, j-2); (i+1, j+2); (i+2, j-1); (i+2, j+1)];;


let affiche tab n = for x = 0 to (n -  1) do for y = 0 to (n - 1) do
        let k = tab.(x).(y) in
        if k < 10 then print_char ` `; print_int k; print_string " " done;
        print_newline () done; print_newline();;


let cheval n dx dy = let ech = make_matrix n n 0 in ech.(dx).(dy) <- 1; parcours dx dy 1

where rec parcours i j k = if k = n*n then (affiche ech n; true) else essai (k+1) (voisins i j n)

and essai e = fun [] -> false (* aucun saut possible après *)
                | ((x, y)::f) -> if ech.(x).(y) <> 0 then essai e f (* case déjà occupée *)
                        else (ech.(x).(y) <- e; (parcours x y e) or (ech.(x).(y) <- 0; essai e f));;
```

cheval 5 0 0;; ☞

```
 1 18  5 10  3
 6 11  2 19 14
17 22 13  4  9
12  7 24 15 20
23 16 21  8 25
```

cheval 6 0 0;; ☞

```
 1  8  5 20  3 10
 6 19  2  9 34 21
15 28  7  4 11 32
18 25 16 33 22 35
29 14 27 24 31 12
26 17 30 13 36 23
```

cheval 8 0 0;; ☞

```
 1 12  9  6  3 14 17 20
10  7  2 13 18 21  4 15
31 28 11  8  5 16 19 22
64 25 32 29 36 23 48 45
33 30 27 24 49 46 37 58
26 63 52 35 40 57 44 47
53 34 61 50 55 42 59 38
62 51 54 41 60 39 56 43
```

14. Un autre parcours du cavalier sur un échiquier

C'est le problème voisin consistant, sur un échiquier de n cases sur n cases, partant de la case dx, dy (comptées à partir de 0), à parcourir tout l'échiquier (une seule fois chaque case) en suivant une ligne ou colonne et se déplaçant 3 cases plus loin ou alors en diagonale d'un carré de 3 sur 3 à chaque fois.
Par rapport au problème précédent, seule la fonction "voisins" est à modifier :

```
let voisins i j n = filtre (fun (x, y) -> 0 <= x & x < n & 0 <= y & y < n)
        [(i, j - 3); (i, j + 3); (i - 3, j); (i + 3, j); (i + 2, j - 2); (i + 2, j + 2); (i - 2, j - 2); (i - 2, j + 2)];;
```

Exemples   cheval 5 0 0;;   cheval 6 0 0;;

1	25	18	2	24
20	10	5	21	11
7	16	13	8	17
4	22	19	3	23
14	9	6	15	12

1	21	26	2	20	31
28	36	18	29	35	17
25	11	8	32	14	7
4	22	27	3	19	30
9	33	15	10	34	16
24	12	5	23	13	6

15. Retour sur les reines de Gauss

On refait la recherche de la première solution (et non de toutes) dans le problème de placer n dames non en prises mutuelles sur un damier n*n, grâce à l'entier n et un vecteur "choix" en dur (global) représentant les numéros de ligne choisis pour chaque colonne de 0 à n-1. Exemple d'une programmation très classique :

```
let n = 8;;

let choix = make_vect n 0;;

let valide l c = let j = ref 0 and drap = ref true in
        while !j < c & !drap do
                let l' = choix.(!j) in drap := (l' <> l) & abs(l-l') <> c - !j; incr j
        done; !drap;;

let rec reine c =
        if c = n then true
        else let l = ref 0 and poss = ref false in
        (while !l < n & not (!poss) do
                if valide !l c then (choix.(c) <- !l; poss := reine (c + 1)); incr l
        done; !poss);;

        (* indique s'il est possible de continuer, pour trouver une solution, en placant une dame à
        une certaine ligne l de la colonne c *)

let main() = if reine 0 then for c = 0 to n-1 do let l = choix.(c) in print_string (make_string l`-`);
                print_string "®"; print_string (make_string (n - l - 1) `-`); print_newline() done;;
```

```
main();;        ®-------
                ----®---
                -------®
                -----®--
                --®-----
                ------®-
                -®------
                ---®----
```

16. Problème des phrases réflexives

"On compte cinq a, un b, huit c, huit d, dix-neuf e, dix-sept i, en ce curieux titre de projet caml qui contient d'autre part vraiment deux j, sept p, et pour finir dix r, cinq s, seize t, douze u, trois z".
On souhaite produire de telles phrases comptant leurs lettres, et qui soient vraies, on se limite pour cela à un certain nombre de corps de phrases type comme :

(| cher lecteur | ami lecteur | chers lecteurs), vous avez (| bien | vraiment | parfaitement) raison de compter (| et de recompter) (| soigneusement | avec soin) (| toutes) les lettres de cette (| longue | curieuse | bizarre) phrase avant de la croire quand elle (| vous) affirme qu'elle contient (| exactement) ... a ... b ... ... et (finalement | pour finir | enfin | pour terminer) ... z.

On peut naturellement prévoir d'autres gabarits et beaucoup plus de variantes, mais on négligera les espaces, accents et symboles de ponctuations, de plus l'énumération des 26 lettres n'est pas obligatoire, il est plus sage de commencer avec une dizaine de lettres. On se limite également à 100 occurrences d'une même lettre.

Pour trouver une solution, à partir d'une phrase, on va utiliser la méthode de Hofstadter consistant à affecter au hasard des valeurs littérales aux lettres (par exemple quatre a, cinq b ..). A chaque itération on compare le nombre présent dans la phrase au nombre réel (par exemple, si on compte 7 a et 2 b, on remplacera quatre par sept et cinq par deux).
La recherche consiste à remplacer pour chaque lettre, son nombre affiché par son nombre réel (ou du moins sa traduction en français littéral) et on recommence.

Si pour toutes les lettres il y a accord, alors naturellement on stoppe la procédure en délivrant la solution.
Il est très curieux de constater que chaque étape bouleversant complètement le compte, on arrive néanmoins parfois à une solution (un point fixe de la fonction "modif").
Mais il est prudent de stopper également après un nombre d'itérations de l'ordre 100.
On représente une phrase comme la liste de ses mots (une traduction en liste de tous les caractères serait plus simple et plus adaptée à la programmation récursive, mais elle obligerait à faire un traitement spécial pour les nombre littéraux, ceux-ci seront formés d'un seul mot grâce à des traits d'union).

```
let rec ret x = fun [] -> [] | (a::q) -> (if x = a then (ret x q) else a::(ret x q));;

let tradliste ph = rev (ret "" (trad' [] ph))
        where rec trad' r  ph = if ph = "" then r else let lg = (string_length ph) and i = ref 0 in
                while !i < lg & (nth_char ph !i) <> ` ` do incr i done;
                trad' ((sub_string ph 0 !i) :: r)
                        (if !i > lg-2 then "" else (sub_string ph (!i + 1) (lg - !i  - 1)));;

let rec aff = fun [] -> print_newline() | (x :: q) -> (print_string (x ^" "); aff q);;
        (* affiche une liste de mots *)

let rec tradmot = fun [] -> "" | (x :: q) -> x  ^" " ^ (tradmot q);;

let compte ph = let c = make_vect 26 0 in compte' ph
where rec compte' = fun [] -> c | (m :: q) -> (comptemot m; compte' q)

and comptemot m = for i = 0 to (string_length m) - 1 do
        let a = int_of_char (nth_char  m i) in
        if 64 < a & a < 91 then c.(a - 65) <- c.(a - 65) + 1
        else if 96 < a & a < 123 then c.(a - 97) <- c.(a  -  97) +1
        done;;
```

```
tradliste "Sic transit gloria mundi.";; ☞  string list = ["Sic"; "transit"; "gloria"; "mundi."]
```

```
tradmot ["Défense "; "d'afficher."];;
☞ string = "Défense d'afficher."

compte (tradliste "Vous avez bien raison de compter les lettres de cette phrase bizarre avant de la croire quand elle vous affirme qu'elle contient onze a, trois b, sept c, huit d, trente-cinq e, cinq f, quatre g, sept h, vingt-sept i, un j, un k, huit l, trois m, vingt-et-un n, douze o, huit p, six q, seize r, dix-sept s, vingt-huit t, dix-neuf u, huit v, un w, six x, un y, et pour finir six z.");;
☞ [|11; 3; 7; 8; 35; 5; 4; 7; 27; 1; 1; 8; 3; 21; 12; 8; 6; 16; 17; 28; 19; 8;1; 6; 1; 6|]
```

```
let rec litt = fun 0 -> "" | 1 -> "un" | 2 -> "deux" | 3 -> "trois" | 4 -> "quatre"
        | 5 -> "cinq"|  6 -> "six"
        |7 -> "sept" | 8 -> "huit" | 9 -> "neuf" | 10 -> "dix"
        | 11 -> "onze" | 12 -> "douze"
        | 13 -> "treize" | 14 -> "quatorze" |15 -> "quinze"
        | 16 -> "seize"| 17 -> "dix-sept"
        | 18 -> "dix-huit" | 19 -> "dix-neuf" | 20 -> "vingt"
        | 30 -> "trente" | 40 -> "quarante"
        | 50 -> "cinquante" | 60 -> "soixante" | 80 -> "quatre-vingt"
        | 100 -> "cent"
        | n -> let u = n mod 10 and d = n / 10 in
                if d = 7 or d = 9
                then (litt (10*(d-1))) ^ (if d = 7 & u = 1 then "-et-" else "-") ^ (litt (u+10))
                else (litt (10*d)) ^ (if u = 1 & d <> 8 then "-et-" else  ("-")) ^ (litt u);;
```

```
litt 22;;☞ "vingt-deux"
litt 71;;☞  "soixante-et-onze"
litt 81;;☞ "quatre-vingt-un"
```

```
let modif phl v = modif' v  phl
where rec modif' v = fun [] -> []
        | [m] -> [m] | (x :: y :: q) -> if string_length y = 1or mem (nth_char y 1) [`,`;`.`]
                                then (litt v.(int_of_char (nth_char y 0) - 97)) :: y :: (modif' v q)
                                else x :: (modif' v (y::q));;
```

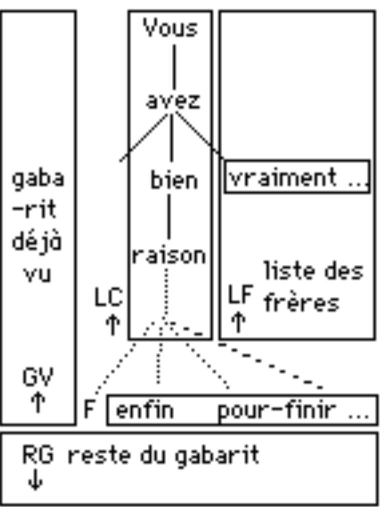
```
modif ["Ya";"trois";"a,";"deux";"b,";"et";"un";"c."] [|7;8;9|];;
☞ ["Ya";"sept";"a,";"huit";"b,";"et";"neuf";"c."]
```

```
let ni = 100;;     (* nombre d'itérations que l'on se fixe *)

let cherche ph = print_string ph; let phl = tradliste ph in aff phl;
                        cherche' phl (modif phl (compte phl)) 0

where rec cherche' p0 p1 n =
        if p0 = p1 then ((string_of_int n) ^ " étapes : " ^ (it_list (fun x y ->x ^ " " ^ y) " " p0))
        else if n > ni then "Pas trouvé."
        else  cherche' p1 (modif p1 (compte p1)) (n + 1);;

let verif pf = let ll = tradliste pf in if modif ll (compte ll) = ll then "phrase réflexive" else "faux";;
```

```
cherche "Ya six a, trois b et deux c.";;
☞ string = " Ya deux a, un b et un c." (* qui est tout de suite stable *)

cherche "Vous avez bien raison de compter les lettres de cette phrase bizarre avant de la croire quand elle vous affirme qu'elle contient un a, deux b, trois c, quatre d, cinq e, six f, sept g, huit h, neuf i, dix j, onze k, douze l, treize m, quatorze n, quinze o, seize p, dix-sept q, dix-huit r, dix-neuf s, vingt t, vingt-et-un u, vingt-deux v, vingt-trois w, vingt-quatre x, vingt-cinq y et pour finir zéro z.";;
```

☞ Vous avez bien raison de compter toutes les lettres de cette phrase avant de la croire quand elle vous affirme qu'elle contient seize a, deux b, huit c, huit d, quarante-et-un e, quatre f, cinq g, cinq h, vingt-trois i, un j, un k, huit l, quatre m, vingt-trois n, quatorze o, quatre p, douze q, vingt-deux r, treize s, trente-trois t, vingt-cinq u, neuf v, un w, quatre x, un y, et pour terminer six z. (* Très long, beaucoup plus que 1000000 *)

Par ailleurs, en considérant un "gabarit" comme une liste de listes de mots, un backtrack permettrait de chercher toutes les solutions accessibles en un nombre limité d'itérations, à partir des phrases engendrées par ce gabarit.




```
let recherche g = parcours (hd g) [] [] (tl g) [hd g]
where rec parcours f lc lf rg gv =
                    (* gv + rg constituent la référence *)
if rg = [] then let res = cherche (tradmot (rev ((hd f) :: lc))) in
          if res = "Pas trouvé." then
          (parcours (hd lf) (tl lc) (tl lf) ((hd gv) :: rg) (tl gv))
                    (* remontée *)
          else res (* cas du succès *)
else if f = [] then if lc = [] then "échec"
          else (parcours (hd lf) (tl lc) (tl lf) ((hd gv) :: rg) (tl gv))
else (parcours (hd rg) ((hd f) :: lc) ((tl f) :: lf) (tl rg) ((hd rg) :: gv))
                    (* dernier cas : la descente *);;
```

```
recherche [["On-trouve"; "Il-y-a"; "On-compte"; ]; ["vingt"]; ["a,"]; ["trois"]; ["b,"]; ["dix"]; ["c,"]; ["six"]; ["d,"]; ["trente"]; ["e,"]; ["dans"; "en"]; ["cette"]; ["curieuse"; "";"bizarre"; "etrange"; "bizarroide"]; ["phrase"; "sentence"]; ["qui"]; ["possede"; "contient"]; ["aussi"; "de plus"; "d'autre part"; "encore"]; ["vraiment"; ""; "exactement"]; ["trois"]; ["f,"; "g,"; "h,"; "i,"; "j,"; "k,"]; ["cinq"]; ["m,"]; ["six"]; ["n,"]; ["et"]; [""; "enfin"; "pour finir"; "pour terminer"]; ["sept"]; ["r,"; "s,"; "t,"]; ["un"]; ["u."]];;
```

☞ On trouve cinq a, un b, six c, cinq d, quatorze e, dans cette curieuse phrase qui possede aussi deux h, un m, onze n, et enfin cinq r, dix u."

Amélioration de Pitrat : un seul des nombres pour lequel le compte réel est différent du nombre littéral affiché, est modifié. Ce nombre concerne la première lettre à partir de celle qui a été modifiée l'étape précédente afin d'éviter de boucler.

Voir : Rapport interne Laforia 96/26 Université Paris VI, 1996.

17. Distance linguistique

On définit la "distance d'édition" entre deux mots x y comme le nombre minimum de transformations pour passer de x à y, nombre pondéré par des poids. Les transformations sont suppression ou insertion d'une lettre, substitution d'une lettre par une autre. Ainsi (sans poids) edit ("chat", "cat") = 1 (une suppression), edit ("chat", "chien") = 3 (2 substitutions, 1 suppression) et edit ("chien", "chenil") = 3 (1 suppression, 2 ajouts).
On définit la similarité de deux mots comme :

$$sim(x, y) = 1 / (1 + [edit(x, y) / \min (long\ x, long\ y)]^4)$$

De cette façon on obtient un coefficient entre 0 et 1. Si $X = \{x_1, x_2, ... x_n\}$ et $Y = \{y_1, y_2, ... y_n\}$ sont deux familles de mots on définit alors $dl (X, Y) = (\sum_{1 \le i \le n} edit (x_i, y_i)) / n$
En notant $x_{1,p}$ le préfixe du mot x formé de la première à la p-ième lettre, et par x_i la i-ième lettre de x, trouver une formule de récurrence entre $edit (x_{1,i}, y_{1,j})$ et $edit (x_{1,i'}, y_{1,j'})$ avec $i' \le i$, $j' \le j$. Développer l'arbre des appels par exemple pour "chat" et "cat" ou "katz".

Voir : Ganascia J.G. *Extraction of recurrent patterns from stratified ordered trees,* Machine Learning Conf. Springer LNAI 2167 p167-178, 2001

On définit des poids pour subst dans un même groupe de lettre (1), autre subst (2), déletion ou ajout (4), on prend les groupes très simplifiés voyelles, labiales (p, b, v, f), dentales (t, d), apicales (l, r), sifflantes (s, z, x, ch), occlusives (c, k, g, q), aspirées (h), palatales (j, w), nasales (m, n). Ci-dessous sim x y le, indique si x et y appartiennent à un même élément de la liste de listes où on groupé les voyelles, les consomnes dentales, occlusives...

```
let rec sim x y = fun [] -> false | (e :: q) -> if mem x e & mem y e then true else sim x y q;;

let ed a b = edit a b (string_length a) (string_length b)

    where rec edit a b la lb = if la = 0 then 4 * lb else if lb = 0 then 4 * la else min (min
    ((edit (sub_string a 0 (la - 1)) b (la - 1) lb)
            + (if mem (nth_char a (la - 1)) [`a`;`e`;`i`;`o`;`u`;`y`] then 2 else 4))
    ((edit a (sub_string b 0 (lb - 1)) la (lb - 1))
            + (if mem (nth_char b (lb - 1)) [`a`;`e`;`i`;`o`;`u`;`y`] then 2 else 4)))
    (cout (nth_char a (la - 1)) (nth_char b (lb - 1)) + edit (sub_string a 0 (la - 1))
            (sub_string b 0 (lb - 1)) (la - 1) (lb - 1))
    where cout x y = if x = y then 0 else if sim x y [[`a`;`e`;`i`;`o`;`u`;`y`]] then 1
    else if sim x y [[`p`;`b`;`v`;`f`]; [`t`;`d`]; [`l`;`r`];
                        [`s`;`z`;`x`]; [`c`;`k`;`h`;`q`;`g`]; [`j`;`y`;`w`;`v`]; [`m`;`n`]]
    then 2 else 3;;
```

Avec une première définition :
ed "chat" "cat" ☞ 4
ed "chat" "chien" ☞ 7
ed "chat" "gato" ☞ 7
ed "katz" "cat" ☞ 5
ed "katz" "chat" ☞ 7
ed "dix" "ten" ☞ 4
ed "zehn" "deset" ☞ 10
ed "dix" "deset" ☞ 10
ed "ten" "zehn" ☞ 6
ed "taxi" "taksi" ☞ 5
ed "restaurant" "resto" ☞ 21
ed "restau" "resto" ☞ 5
ed "restaurant" "restor" ☞ 17
ed "resto" "restoran" ☞ 12
ed "restaurant" "restora" ☞ 13
ed "restaurant" "restoran" = 9,

Avec seconde définition:
ed "restau" "resto" ☞ 3   ed "chat" "cat" ☞ 4   ed "xat" "cat" ☞ 3

Application au calcul de toutes les distances deux à deux entre les langues sur un échantillon donné. Ces distances sont enmagasinées dans la matrice dd, et on calcule les distances minimale "mi" et maximale "ma", de façon à s'en servir au mieux dans la représentation graphique.
Un individu est un vecteur de dimension 2.ml où ind(i) et ind(i + ml) sont des listes de chiffres dont l'effet de "contract lc 0 275" donne les coordonnées à l'écran.

```
let dist a b = let d = ref 0 in for i = 1 to mm do d := !d + ed lg.(a).(i) lg.(b).(i) done; !d;;

let dd = make_matrix ml ml 0;;

let calcul () = let lmax = ref 0 and lmin = ref 1000 in for i= 0 to ml  - 1 do for j = i + 1 to ml  -  1 do
    dd.(i).(j) <- dist i j;
    lmax := max !lmax dd.(i).(j);
    lmin := min !lmin dd.(i).(j) done done; (!lmin, !lmax);;
```

```
let mi , ma = calcul ();;   (* si on calcule min et max dans dd, on accentue les distances : *)
let m = 270./.(float_of_int (ma - mi));;
let ml = 34 and mm = 10;;                (* nb de langues et de mots dans chacune *)
let lg = make_matrix 60 11 "";;          (* par exemple lg.(0).(5);; -> "cinq" *)

lg.(0) <- [|"Français";"un";"deux";"trois";"quatre";"cinq";"soleil";"eau";"maison";"tete";"main"|];;
lg.(1) <- [|"Albanais";"nje";"dy";"tre";"kater";"pese";"dieli";"uje";"xtepi";"krye";"dore"|];;
lg.(2) <- [|"Gallois";"un";"dau";"tri";"pedwar";"pump";"haul";"dwr";"ty";"pen";"llaw"|];;
lg.(3) <- [|"Grec";"ene";"dio";"tris";"teseris";"pente";"ilios";"nero";"spiti";"kefalo";"keri"|];;
lg.(4) <- [|"Hindi";"ek";"do";"tin";"txar";"panx";"surya";"pani";"ghar";"sir";"hath"|];;
lg.(5) <- [|"Lituanien";"vienas";"du";"trys";"keturi";"penki";"saule";"vanduo";"namas";"galva";"ranka"|];;
lg.(6) <- [|"Letton";"viens";"divi";"tris";"xetri";"pieci";"saule";"udens";"nams";"galva";"roka"|];;
lg.(7) <- [|"Cinghalais";"eka";"deka";"tuna";"hatara";"paha";"ira";"watura";"geya";"hisa";"ata"|];;
lg.(8) <- [|"Allemand";"ein";"zwei";"drei";"vier";"funf";"sonne";"wasser";"haus";"kopf";"hand"|];;
lg.(9) <- [|"Danois";"en";"to";"tre";"fire";"fem";"solen";"vandet";"huset";"hovedet";"haanden"|];;
lg.(11) <- [|"Anglais";"one";"two";"thri";"four";"five";"sun";"water";"house";"head";"hand"|];;
lg.(12) <- [|"Espagnol";"uno";"dos";"tres";"quatro";"cinco";"sol";"agua";"casa";"cabeza";"mano"|];;
lg.(13) <- [|"Roumain";"un";"doi";"trei";"patru";"cinci";"soarele";"apa";"casa";"capul";"mina"|];;
lg.(14) <- [|"Polonais";"jeden";"dwa";"trzy";"cztery";"piech";"solnce";"woda";"dom";"glowa";"reka"|];;
lg.(15) <- [|"Russe";"odin";"dva";"tri";"xetyre";"pyat";"solntse";"voda";"dom";"golova";"ruka"|];;
lg.(16) <- [|"Bulgare";"edno";"dve";"tri";"xetiri";"pet";"sluntse";"voda";"kuxta";"glava";"ruka"|];;
lg.(17) <- [|"Finnois";"yksi";"kaksi";"kolme";"nelje";"viisi";"aurinko";"vesi";"talo";"paa";"kasi"|];;
lg.(18) <- [|"Estonien";"uks";"kaks";"kolm";"neli";"viis";"paike";"vesi";"maja";"pea";"kasi"|];;
lg.(10) <- [|"Hongrois";"egy";"keto";"harom";"negy";"ot";"nap";"viz";"haz";"fej";"kez"|];;
lg.(19) <- [|"Turc";"bir";"iki";"uc";"dort";"pex";"gunex";"su";"ev";"bax";"el"|];;
lg.(20) <- [|"Ouigour";"bir";"iki";"utch";"tot";"bax";"kun";"su";"bolma";"bax";"qol"|];;
lg.(21) <- [|"Mongol";"neg";"xoyor";"gurav";"dorov";"tav";"nar";"us";"ger";"tolgoy";"gar"|];;
lg.(22) <- [|"Kabyle";"yiwen";"sin";"tlata";"arba";"xemsa";"ittij";"aman";"axxam";"aqeru";"afus"|];;
lg.(23) <- [|"Hébreu";"akhat";"xtayim";"xalox";"arba";"xamax";"xemex";"mayim";"bet";"rox";"yad"|];;
lg.(24) <- [|"Arabe";"wahid";"tnin";"talata";"arba";"khamsa";"shams";"ma";"dar";"ras";"yed"|];;
lg.(25) <- [|"Araméen";"gha";"tray";"tlata";"arba";"hamxa";"yoma";"maye";"beta";"rexa";"ida"|];;
lg.(26) <- [|"Ethiopien";"and";"hulet";"sost";"arat";"amest";"tsehay";"weha";"bet";"ras";"edj"|];;
lg.(27) <- [|"Maltais";"wiehed";"tnejn";"tlieta";"erbgha";"hamsa";"xemx";"ilma";"dar";"ras";"id"|];;
lg.(28) <- [|"Tibétain";"chig";"nyi";"sum";"shi";"nga";"nyima";"tchu";"khangpa";"go";"lakwa"|];;
lg.(29) <- [|"Coréen";"hana";"tui";"set";"net";"tasot";"teyang";"mul";"txip";"mori";"son"|];;
lg.(30) <- [|"Japonais";"itchi";"ni";"san";"shi";"go";"taiyo";"mizu";"utxi";"atama";"te"|];;
lg.(31) <- [|"Malais";"satu";"dua";"tiga";"empat";"lima";"matahari";"air";"rumah";"kepala";"tangan"|];;
lg.(32) <- [|"Malgache";"isa";"roa";"telo";"efatra";"dimy";"masandro";"rano";"trano";"loha";"tanana"|];;
lg.(33) <- [|"Tagalog";"isa";"dalawa";"talo";"apat";"lima";"araw";"tubig";"bahay";"ulo";"kamay"|];;
lg.(34) <- [|"Lapon";"okta";"guokte";"golma";"njelje";"vihta";"baivax";"txasi";"visti";"caivi";"giehta"|];;
lg.(35) <- [|"Arménien";"meg";"yergu";"yereg";"txors";"hing";"arev";"dxur";"dun";"kelur";"tzerk"|];;
lg.(36) <- [|"Tamoul";"onru";"irendu";"munru";"mangu";"aindu";"suriyan";"nir";"vidu";"tala";"kai"|];;
lg.(37) <- [|"Somali";"kow";"laba";"sedex";"afar";"xan";"qoraxda";"biyoha";"aqalka";"madaxa";"gacanta"|];;
lg.(38) <- [|"Géorgien";"erti";"ori";"sami";"otkhi";"khuti";"mze";"tskali";"sakhli";"tavi";"kheli"|];;
lg.(39) <- [|"Tchétchène";"ca";"chi";"qo";"di";"phi";"malh";"hi";"ca";"kort";"kilg"|];;
lg.(40) <- [|"Basque";"bat";"ni";"hiru";"lau";"bortz";"iguzki";"ur";"etche";"buru";"esku"|];;
lg.(41) <- [|"Swahili";"moja";"mbili";"tatu";"ne";"tano";"jua";"maji";"nyumba";"kixwa";"mkono"|];;
lg.(42) <- [|"Burushaski";"hin";"altan";"isken";"walto";"tsundo";"sa";"xil";"ha";"kapal";"riri"|];;
lg.(43) <- [|"Maya";"jun";"cai";"oxi";"caji";"voho";"tonatiuh";"ya";"calli";"jolon";"ka"|];;
lg.(44) <- [|"Quichua";"juk";"ishcai";"quinsa";"tahua";"pixica";"inti";"yacu";"huasi";"uma";"maki"|];;
lg.(45) <- [|"Guarani";"petel";"mokol";"mbohapi";"irundy";"sinko";"kuarahy";"y";"oga";"aka";"po"|];;
lg.(46) <- [|"Mapuche";"kine";"epu";"kula";"meli";"kechu";"antu";"co";"ruca";"lonco";"cuq"|];;
lg.(47) <- [|"Cantonais";"yat";"yih";"saam";"sel";"ngh";"yath";"seui";"nguk";"tauh";"sau"|];;
lg.(48) <- [|"Vietnamien";"mot";"hai";"ba";"bon";"nam";"mat";"muoc";"nha";"dau";"ban"|];;
lg.(49) <- [|"Khmer";"muoy";"pi";"bei";"buon";"praim";"preahatit";"tuk";"phteas";"kbal";"day"|];;
lg.(50) <- [|"Zoulou";"nye";"bili";"tatu";"ne";"hlanu";"langa";"amanzi";"indhlu";"ikhanda";"isandhla"|];;
lg.(51) <- [|"Afar";"enek";"namay";"sidoh";"ferey";"konoy";"aran";"le";"bura";"amyota";"gaba"|];;
lg.(52) <- [|"Yoruba";"okan";"meji";"meta";"merin";"marun";"orun";"omi";"ilé";"ori";"owo"|];;
lg.(53) <- [|"Moré";"ié";"yi";"ta";"naase";"nu";"uende";"kom";"yiri";"zugu";"nugu"|];;
lg.(54) <- [|"Kikuyu";"mwe";"iri";"tatu";"na";"tano";"riua";"mai";"nyumba";"mutwe";"guoko"|];;
lg.(55) <- [|"Masai";"obo";"are";"okuni";"onguan";"imiet";"enkolong";"enkare";"aji";"dukuya";"aina"|];;
lg.(56) <- [|"Dogon";"tiru";"izi";"tanu";"nal";"nuru";"nal";"di";"giru";"kuu";"nuu"|];;
lg.(57) <- [|"Wolof";"benn";"naar";"natt";"nenent";"juroom";"jant";"ndox";"ker";"bopp";"loxo"|];;
lg.(58) <- [|"Peul";"gooto";"didi";"tati";"nayi";"jowi";"naange";"ndiyam";"suudu";"hoore";"junngo"|];;
lg.(59) <- [|"Shona";"mwe";"viri";"tatu";"na";"shanu";"zuva";"mvura";"imba";"musoro";"ruoko"|];;
```

Après le calcul, une fois pour toute du tableau des distances dd, en utilisant l'algorithme SSGA défini plus loin au chapitre 11, on trace dans une fenêtre une représentation des langues par des points minimisant l'erreur.
Un individu-chromosome est un vecteur de dim 2ml, sa "fitness" est l'erreur quadratique, (problème de facteur d'échelle), à fixer un ou deux gènes du chromosome c'est à dire deux langues particulières...)

Naturellement un tel programme ne serait intéressant que sur un échantillon plus grand de mots supposés fondamentaux, mais surtout avec l'alphabet phonétique, par ailleurs l'intérêt n'est pas de retrouver une classification en familles bien connues de langues (sémitiques, malayo-polynésienne, indo-européennes et toutes leurs subdivisions ...) mais sur les langues isolées et dont l'histoire n'est pas connue.

```
let fitness x (* x est un vecteur réel de dimension 2*ml *) = let d = ref 0. in
        for i = 0 to ml - 1 do for j = i +1 to ml - 1 do d := !d +.
                sqr(m*.(float_of_int (dd.(i).(j) - mi)) -. sqrt (sqr (x.(i) -. x.(j)) +. sqr(x.(i+ml) -. x.(j+ml))))
        done done; !d;;
let vue t a b = print_string " Meilleur="; print_int (round t.(0).score); print_string " pour (";
        for i = 0 to vect_length t.(0).genes - 2 do print_int (round (contract (t.(0).genes).(i) a b));
        print_string ", " done;
        print_int (round(contract (t.(0).genes).(vect_length t.(0).genes - 1) a b));  print_char `)`;;
let voir ind m a b = efface (2*(round m)); cadre (round m); for i = 0 to ml -1 do
        let x, y = dilat m (contract ind.genes.(i) a b) a b, dilat m (contract ind.genes.(i+ml) a b) a b
        in moveto x y; set_color red; draw_string lg.(i).(0) (*sub_string  0 4*) done;;
```

```
On peut lancer
phi 12 100000 0. fitness 0. 275. (2*ml) 33 33 true;;
```

Russe Bulgare Géorgien Somali
Lituanien Polonais Swahili
Allemand
Danois Letton Tibétain
Gallois
Albanais Tchétchène
Espagnol
Roumain Japonais
Français
Coréen
Grec Anglais
Hindi Basque
Cinghalais Ouigour
Malais Hongrois Turc
Malgache
Estonien
Tagalog
Arménien Araméen
Mongol Arabe
Hébreu
Finnois Ethiopien
Lapon Tamoul Kabyle Maltais

Ce n'est pas idéal, on voit clairement les incohérences, mais pour le petit nombre de mots et le peu de travail effectué sur leur différences de morphologie, ce n'est pas si mal. On a clairement pour les 42 premières langues du tableau, les langues sémitiques dans la région en bas à droite, les langues ouraliennes à droite, slaves, baltes, latines et indo-européennes en général en haut à gauche à l'exception de l'arménien.

CHAPITRE VIII

ARTICLES (RECORDS)

Le mixage des types à l'intérieur d'une même fiche se fait par des articles. Ceux-ci sont définis entre des accolades qui renferment plusieurs "champs". Les champs, qui peuvent être variables, doivent être signalés par "mutable".
Par exemple on écrira :

```
type article = {nom : string; mutable age : int; sexe : bool; diplomes : string list};;
```

L'accès se fait à l'intérieur d'une fiche A (avec la même syntaxe que pour les vecteurs qui sont des articles à un seul champ entier) par le point, ainsi A.nom, et la modification A.age <- 12.
Dans une définition de type :

```
type t = {mutable  champ : int ref};;
avec par exemple : let x = {champ = ref 0};;
```

On aura par exemple l'affectation x.champ := 1;; qui signifie que le contenu !x.champ sera 1. L'affectation x.champ <- ref 1;; aurait fait de même.

```
let a = {nom = "Christelle"; age = 35; sexe = true; diplomes = ["bac"; "licence d'histoire"]};;
a.age;; ☞ int = 35
a.age <- 36;; ☞ unit = ()
a.age;; ☞ int = 36
```

1. Définition de type par article, exemple des fractions

En ML un type de donnée et les opérations s'y rapportant :

```
let rec pgcd a b = if a = b then a else if a < b then pgcd b a else pgcd (a-b) b;;
type fraction = int*int
        with irr x = a/p, b/p where a, b = x and p = pgcd a b
        and plus x y = irr a*b + c*d, b*d where a, b = x and c, d = y
        and mul x y = irr a*c, b*d where a, b = x and c, d = y;;
```

Mais ce n'est plus possible en Caml où il faut revenir à une syntaxe plus lourde, les articles sont définis avec des accolades et les noms choisis pour les champs :

```
type fraction = {num : int; den :int};;                ☞  Type fraction defined.
let irr x = let a = x.num and b = x.den in let p = pgcd a b in {num = a / p; den = b / p} ;;
        ☞  irr : fraction -> fraction = <fun>

irr {num = 12; den = 16};;                             ☞ fraction = {num = 3; den = 4}
```

```
let plus x y = irr {num = a * d + b * c; den = b * d}
        where a = x.num and b = x.den and c = y.num and d = y.den;;
        ☞ plus : fraction -> fraction -> fraction = <fun>

    plus {num = 1; den = 2} {num = -1;den = 6};;    ☞ fraction = {num = 1; den = 3}

let mul x y = irr {num = a*c; den = b*d}
        where a = x.num and b = x.den and c = y.num and d = y.den;;
        ☞ mul : fraction -> fraction -> fraction = <fun>

    mul {num = 2; den = 3} {num = 6;den = 5};;    ☞ fraction = {num = 4; den = 5}
```

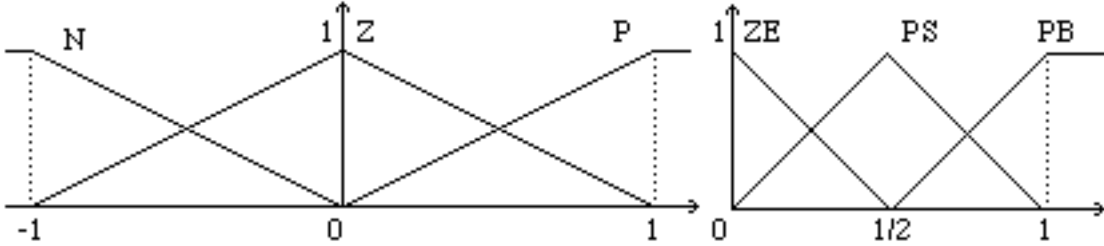
2. Exemple des complexes

Les complexes peuvent être définis tout à fait de la même façon, comme des couples de réels où la somme est la somme des couples, et le produit de telle sorte que i = (0, 1) soit de carré -1.

```
type complex = {re : float; im : float};;    ☞ Type complex defined.

    let c1 = {re = 0.; im = 1.};;    ☞ c1 : complex = {re = 0; im =1 }

let prod c1 c2 = {re = c1.re *. c2.re -. c1.im*.c2.im; im = c1.re*.c2.im +. c1.im*.c2.re};;
        ☞ prod : complex -> complex -> complex = <fun>

    prod c1 {re = 2.; im = -1.};;    ☞ complex = {re = 1; im = 2}
```

3. Exemple du contrôleur flou de Sugeno

Le principe de cet algorithme est celui d'une fonction de deux variables à valeur réelle déterminée comme la moyenne des valeurs renvoyées par un jeu de règles floues plus ou moins vérifiées par les deux entrées : étant données deux réels x, y (des grandeurs physiques), chaque règle est vérifiée par le min des valeurs d'appartenance de x et y aux deux prédicats flous composant les hypothèses de la règle.
Le résultat est la moyenne pondérée des conclusions des règles, pondérées par les différents niveaux de satisfaction de ces règles. Soient 8 prédicats définis par des fonctions d'appartenance triangulaires (1 en un certain "sommet" et 0 à l'extérieur du support [s - pe, s + pe] où pe est la "pente". ANY est le prédicat universellement vérifié, Z est zéro, NB est "negative big" etc.
Une règle possède ici trois hypothèses et deux conclusions, mu est la fonction d'appartenance à un prédicat p d'un réel x, "sat" est le niveau de satisfaction d'une règle pour une entrée (x, y, z) suivant le min de Zadeh.
Voir : Gacôgne L. *Eléments de logique floue,* Hermès 1997



On définit le type "pred" par énumération :

```
#open "float";;
type pred = ANY | N | Z | P | NB | NS | ZE | PS | PB;;

type regle = {c1 : float; c2 : float; h1 : pred; h2 : pred; h3 : pred};;
```

```
let opp = fun NB -> PB | NS -> PS | N -> P;; (* mem est le prédicat d'appartenance à une liste *)
let pente p = if mem p [NB; PB; NS; PS; ZE] then 0.5 else 1.;;
let rec sommet p = if p = PS then 0.5 else if p = PB or p = P then 1.
                        else if p = ZE or p = Z then 0. else - (sommet (opp p));;
let max x y = if x < y then y else x and min x y = if x < y then x else y ;;
```

La fonction d'appartenance : (mu p x) renvoie la valeur (entre 0 et 1) du prédicat p pour un x dans l'intervalle [-1, 1].

```
let rec mu p x = if mem p [N; NB; NS] then mu (opp p) (- x)
            else if (mem p [P; PB] & 1. < x) or (p = ANY) then 1.
            else max 0. ((pe - abs_float (x - (sommet p))) / pe) where pe = pente p ;;

let sat r x y z = min (mu r.h1 x) (min (mu r.h2 y) (mu r.h3 z));;
      (* satisfaction de la règle r par les entrées x, y, z *)

let reg p q r u v = { h1 = p; h2 = q; h3 = r; c1 = u ; c2 = v };;

let r = [reg P P P 0. 1.; reg Z ANY P 1. (-0.5); reg P ANY Z (-1.) (-0.5); reg ANY Z ANY 0. (-1.)];;
      (* r n'est qu'un petit exemple *)
```

La fonction principale "fuzzy" (dans cet exemple est une fonction de R^3 dans R^2) calcule le couple de conséquences que donne la liste de règle lr pour une entrée exacte (x, y, z) suivant l'algorithme de Sugeno consistant à faire la moyenne pondérée des sorties.
Cet algorithme modèlise donc, par une sorte d'interpolation, une fonction, dans le cas où, seuls des experts du domaine peuvent donner une conclusion dans certains cas seulement.

```
let fuzzy lr x y z = fuzzy' 0. 0. 0. lr (* sc somme des coefficients de satisfaction des règles *)
      where rec fuzzy' s1 s2 sc = fun [] -> if sc = 0. then (0., 0.) else (s1 / sc, s2 / sc)
            | (r :: q) -> let m = sat r x y z in fuzzy' (s1 +  m * r.c1) (s2 +  m * r.c2) (sc +  m) q;;

fuzzy r 1. 0.5 0.5;; ☞ float * float = -0.333333333333, -0.166666666667
```

4. Exemple de la fonction de Smarandache

Soit la fonction s(n) = min{m / n divise m!}, ainsi s(5228)=1307, s(9504)=11.
On calculera les valeurs de s pour n de 1 à 100.
On peut remarquer que si p est premier s(p) = p et $s(p^2) = 2p$, $s(p^n) \leq np$, en général, et si p < q premiers s(pq) = q.
On peut montrer que dans tous les cas $s(n) \leq n$ et déterminer s(n) en fonction des valeurs $s(p^k)$ où p est premier intervenant dans la décomposition en facteurs premiers de n.
On trouve ici une méthode et une structuration des données pour calculer les valeurs de s de 1 à 10000 ou plus en conservant quelques résultats au fur et à mesure.

Solution naïve ne convenant déjà plus à partir de 11 :

```
let smar n = if n=1 then 0 else smar' 1 1
      where rec smar' k f = if f mod n = 0 then k else smar' (k + 1) (f * (k  + 1));;
```

Solution obtenue en stockant les nombres premiers et la valeur de s pour leurs puissances :

```
let divise p n = (n mod p = 0);;
      (* prédicat indiquant si p divise n c.a.d. n multiple de p *)

let rec exposant p n = (* donne l'exposant du facteur premier p dans la décomposition de n *)
      if divise p n then 1 + exposant p (n / p) else 0;;

let fautvoir p n = p <> 0 & (p <= n);; (* p est un diviseur potentiel de n *)
```

```
let sm p n = (* donne le résultat pour l'entier premier p à la puissance n *)
	let m = ref p and c = ref 1 in
		while !c < n do m := !m + p; c := 1 + !c + exposant p (!m / p) done;
	!m;;
type prem = {mutable pr : int; mutable ex : (int*int) list};;
let t = make_vect 1500 {pr = 0; ex = []};;
for i = 0 to vect_length t - 1 do t.(i) <- {pr = 0; ex = []} done;;
	(* initialisation du tableau "prem" *)

let smar n = let i = ref 0 and v = ref 0 in
while fautvoir (t.(!i).pr) n & !i < vect_length t do let p = t.(!i).pr in let e = exposant p n in
		if e <> 0 then (if e < 3 then v := max !v (e*p)
		else if exists (fun x -> (fst x = e)) t.(!i).ex then v := max !v (assoc e t.(!i).ex)
		else let r = sm p e in (v := max !v r; t.(!i).ex <- (e, r) :: t.(!i).ex));
	incr i
done;
if !v = 0 & n <> 1 then (while t.(!i).pr <> 0 & !i < vect_length t do incr i done; t.(!i).pr <- n; v := n);
!v;;

let main m = for n = 1 to m do
		print_string " s("; print_int n; print_string ")="; print_int (smar n) done;
	print_newline(); print_string "Les nombres premiers calculés sont ";
	let i = ref 0 in
	while t.(!i).pr <> 0 & !i < vect_length t do print_int t.(!i).pr; print_char ` `; incr i done;
	print_string " ("; print_int !i; print_string " nombres premiers)";;
```

```
main 100;; ☞ s(1)=0 s(2)=2 s(3)=3 s(4)=4 s(5)=5 s(6)=3 s(7)=7 s(8)=4 s(9)=6 s(10)=5
s(11)=11 s(12)=4 s(13)=13 s(14)=7 s(15)=5 s(16)=6 s(17)=17 s(18)=6 s(19)=19 s(20)=5
s(21)=7 s(22)=11 s(23)=23 s(24)=4 s(25)=10 s(26)=13 s(27)=9 s(28)=7 s(29)=29 s(30)=5
s(31)=31 s(32)=8 s(33)=11 s(34)=17 s(35)=7 s(36)=6 s(37)=37 s(38)=19 s(39)=13
s(40)=5 s(41)=41 s(42)=7 s(43)=43 s(44)=11 s(45)=6 s(46)=23 s(47)=47 s(48)=6
s(49)=14 s(50)=10 s(51)=17 s(52)=13 s(53)=53 s(54)=9 s(55)=11 s(56)=7 s(57)=19
s(58)=29 s(59)=59 s(60)=5 s(61)=61 s(62)=31 s(63)=7 s(64)=8 s(65)=13 s(66)=11
s(67)=67 s(68)=17 s(69)=23 s(70)=7 s(71)=71 s(72)=6 s(73)=73 s(74)=37 s(75)=10
s(76)=19 s(77)=11 s(78)=13 s(79)=79 s(80)=6 s(81)=9 s(82)=41 s(83)=83 s(84)=7
s(85)=17 s(86)=43 s(87)=29 s(88)=11 s(89)=89 s(90)=6 s(91)=13 s(92)=23 s(93)=31
s(94)=47 s(95)=19 s(96)=8 s(97)=97 s(98)=14 s(99)=11 s(100)=10

Les nombres premiers calculés sont 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61
67 71 73 79 83 89 97  (25 nombres premiers) - : unit = ()
```

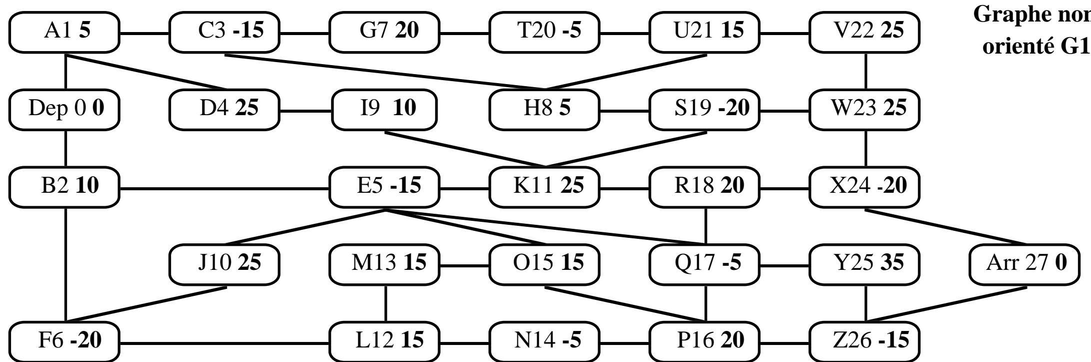
Autre solution plus concise et moins lisible mais avec une estimation empirique de la taille du vecteur :

```
let main m = let p = make_vect (m/4 + 10) 0 in
for n = 1 to m do print_string (" "^(string_of_int n)^"->"); print_int
	(let i = ref 0 and v = ref 0 in while 0 < p.(!i) & p.(!i) < n & !i < vect_length p
		do let e = exposant p.(!i) n in if e <> 0 then v := max !v (sm p.(!i) e); incr i done;
	if !v = 0 & n <> 1 then
		(while p.(!i) <> 0 & !i < vect_length p do incr i done; p.(!i) <- n; v := n); !v)
where rec  exposant p n = if n mod p = 0 then 1 + exposant p (n/p) else 0
and sm p n =
		let m = ref p and c = ref 1 in
		while !c<n do m:=!m+p; c:=1+ !c+exposant p (!m/p) done; !m done;
		print_newline(); let i=ref 0 in
		while p.(!i)<>0 & !i<vect_length p do print_int p.(!i); print_char ` `; incr i done;
		print_string " ("; print_int !i; print_string " nombres premiers calculés) ";;
```

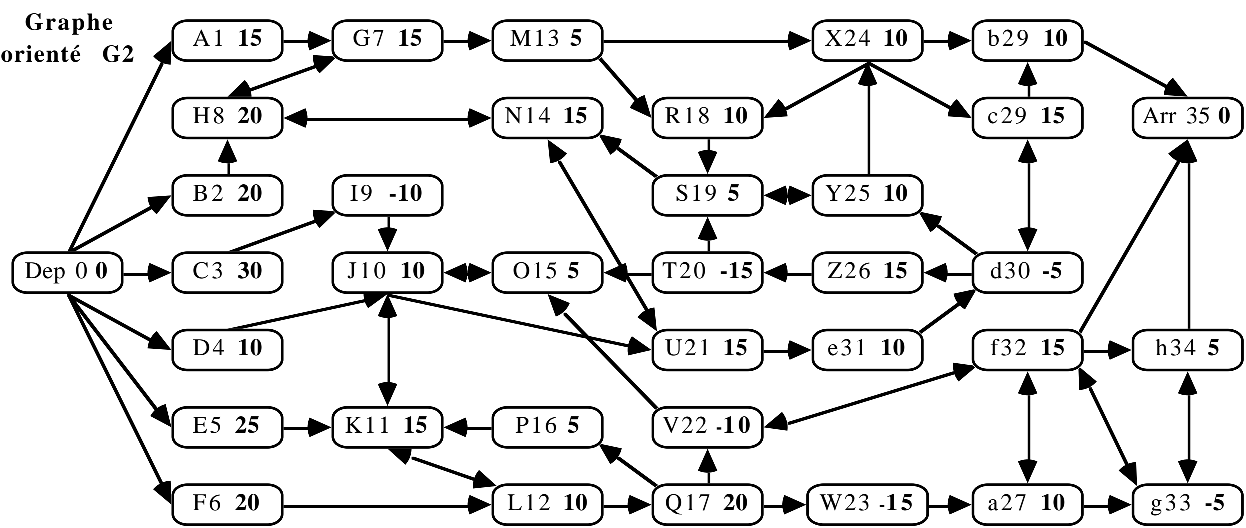
```
main 10;;
☞ 1->0 2->2 3->3 4->4 5->5 6->3 7->7 8->4 9->6 10->5
2 3 5 7  (4 nombres premiers calculés) - : unit = ()
```

5. Meilleur chemin dans un graphe

Le problème consiste à trouver le chemin de "Dep" à "Arr" totalisant le plus grand total de points (les points sont en gras). Les sommets peuvent être repérés par une lettre ou un numéro.



Il faut prévoir une représentation du graphe qui puisse servir aussi bien à un graphe non orienté (une relation symétrique) comme G1 (on doit trouver 200) qu'à un graphe orienté tel que G2 (où le maximum est 205).



On choisit de représenter le graphe par un vecteur de noeuds, chaque noeud ayant un nom, une valeur et une liste de noeuds accessibles. Il sera plus facile d'accéder aux noeuds par leur numéro.

```
type noeud = {Nom : string; Val : int; Acc : int list};;
```

```
let g1 = [|{Nom = "Dep"; Val = 0; Acc =[1;2]}; {Nom = "A"; Val = 5; Acc =[0;3;4]};
{Nom = "B"; Val = 10; Acc =[0;5;6]}; {Nom = "C"; Val = -15; Acc =[1;7;8]};
{Nom = "D"; Val = 25; Acc =[1;9]}; {Nom = "E"; Val = -15; Acc =[2;10;11;15;17]};
{Nom = "F"; Val = -20; Acc =[2;10;12]}; {Nom = "G"; Val = 20; Acc =[3;20]};
{Nom = "H"; Val = 5; Acc =[3;19;21]}; {Nom = "I"; Val = 10; Acc =[4;11]};
{Nom = "J"; Val = 25; Acc =[5;6]}; {Nom = "K"; Val = 25; Acc =[5;9;18;19]};
{Nom = "L"; Val = 15; Acc =[6;13;14]}; {Nom = "M"; Val = 5; Acc =[12;15]};
{Nom = "N"; Val = -5; Acc =[12;16]}; {Nom = "O"; Val = 15; Acc =[5;13;16]};
{Nom = "P"; Val = 20; Acc =[14;15;17;26]}; {Nom = "Q"; Val = -5; Acc =[5;16;18;25]};
{Nom = "R"; Val = 20; Acc =[11;17;24]}; {Nom = "S"; Val = -20; Acc =[8;11;23]};
{Nom = "T"; Val = -5; Acc =[7;21]}; {Nom = "U"; Val = 15; Acc =[8;20;22]};
{Nom = "V"; Val = 25; Acc =[21;23]}; {Nom = "W"; Val = 25; Acc =[19;22;24]};
{Nom = "X"; Val = -20; Acc =[18;23;27]}; {Nom = "Y"; Val = 35; Acc =[17;26]};
{Nom = "Z"; Val = -15; Acc =[16;25;27]}; {Nom = "Arr"; Val = 0; Acc =[]} |];;
```

```
let g2 = [|
{Nom = "Dep"; Val = 0; Acc =[1;2;3;4;5;6]}; {Nom = "A"; Val = 15; Acc =[7]};
{Nom = "B"; Val = 20; Acc =[8]}; {Nom = "C"; Val = 30; Acc =[9]};
{Nom = "D"; Val = 10; Acc =[10]}; {Nom = "E"; Val = 25; Acc =[11]};
{Nom = "F"; Val = 20; Acc =[12]}; {Nom = "G"; Val = 15; Acc =[8;13]};
{Nom = "H"; Val = 20; Acc =[7;14]}; {Nom = "I"; Val = -10; Acc =[10;15]};
{Nom = "J"; Val = 10; Acc =[11;21]}; {Nom = "K"; Val = 15; Acc =[10;12]};
{Nom = "L"; Val = 10; Acc =[11;17]}; {Nom = "M"; Val = 5; Acc =[18;24]};
{Nom = "N"; Val = 15; Acc =[8;21]}; {Nom = "O"; Val = 5; Acc =[9]};
{Nom = "P"; Val = 5; Acc =[11]}; {Nom = "Q"; Val = 20; Acc =[16;22;23]};
{Nom = "R"; Val = 10; Acc =[19]}; {Nom = "S"; Val = 5; Acc =[14;25]};
{Nom = "T"; Val = -15; Acc =[15;19]}; {Nom = "U"; Val = 15; Acc =[31]};
{Nom = "V"; Val = -10; Acc =[15;32]}; {Nom = "W"; Val = -15; Acc =[27]};
{Nom = "X"; Val = 10; Acc =[18;28;29]}; {Nom = "Y"; Val = 10; Acc =[19;24]};
{Nom = "Z"; Val = 15; Acc =[20]}; {Nom = "a"; Val = 10; Acc =[32;33]};
{Nom = "b"; Val = 10; Acc =[35]}; {Nom = "c"; Val = 15; Acc =[28;30]};
{Nom = "d"; Val = -5; Acc =[25;26;29]}; {Nom = "e"; Val = 10; Acc =[30]};
{Nom = "f"; Val = 15; Acc =[22;27;33;34;35]}; {Nom = "g"; Val = -5; Acc =[27;32]};
{Nom = "h"; Val = 5; Acc =[35]}; {Nom = "Arr"; Val = 0; Acc =[]}
        |];;
```

```
let rec explo ch sp i g n =
(* cherche à continuer un chemin ch (de somme sp) arrivant au sommet de numéro i et renvoie
un couple chemin, somme maximale (il peut y avoir plusieurs solution équivalente) *)
        if i = n then (rev ch, sp)
                (* le tableau est indexé de 0 (sommet départ) à n (sommet arrivée) *)
        else (let ss = ref 0 and chs = ref [] in
                for j = 1 to n do
                        if valide j then
                                let (sol, s) = explo (j :: ch) (sp + g.(j).Val) j g n
                                in if s > !ss then (chs := sol; ss := s)
                done; (!chs, !ss));

where valide j = mem j g.(i).Acc & not (mem j ch);;

let rec vue (ch, s) g = if ch = [] then (print_string " somme "; print_int s)
                else (print_string (g.(hd ch).Nom ^ " "); vue ((tl ch), s) g);;

let jeu g = vue (explo [0] 0 0 g ((vect_length g)-1)) g;;
```

```
jeu g1;;
☞ Dep B E J F L M O P Z Y Q R K I D A C G T U V W X Arr  somme 200

jeu g2;;
☞ Dep F L Q P K J U e d Y S N H G M X c b Arr  somme 205
```

Autre solution sans définition de type, ni tableau, avec une fonction un peu plus pure, récursive terminale sans boucle, ni variables locales.

On représente le graphe comme une liste de listes, chaque sous-liste" étant un numéro de sommet (0 pour le départ, 99 pour l'arrivée) et sa valeur suivis des numéros de sommets accessibles :

```
let g0 = [[0; 0; 1; 2; 3]; [1; 1; 0; 2; 99]; [2; 3; 0; 1; 3; 99]; [3; 2; 0; 2]; [99; 0; 1; 2]];;

let g1 = [[0;0;1;2]; [1;5;0;3;4]; [2;10;0;5;6]; [3;-15;1;7;8]; [4;25;1;9];
[5;-15;2;10;11;15;17]; [6;-20;2;10;12]; [7; 20;3;20]; [8;5;3;19;21];
[9;10;4;11]; [10;25;5;6]; [11;25;5;9;18;19]; [12;15;6;13;14]; [13;5;12;15];
[14;-5;12;16]; [15;15;5;13;16]; [16;20;14;15;17;26];
[17;-5;5;16;18;25]; [18;20;11;17;24]; [19;-20;8;11;23]; [20;-5;7;21];
[21;15; 8;20;22]; [22; 25; 21;23]; [23;25;19;22;24]; [24;-20;18;23;99];
[25;35;17;26]; [26;-15;16;25;99]; [99;0]];;
```

```
let g2 = [[0; 0; 1;2;3;4;5;6]; [1; 15; 7]; [2; 20; 8]; [3; 30; 9]; [4; 10; 10]; [5; 25; 11];
[6; 20; 12]; [7; 15; 8;13]; [8; 20; 7;14]; [9; -10; 10;15]; [10; 10; 11;21];
[11; 15; 10;12]; [12; 10; 11;17]; [13; 5; 18;24]; [14; 15; 8;21]; [15; 5; 9]; [16; 5; 11];
[17; 20; 16;22;23]; [18; 10; 19]; [19; 5; 14;25]; [20; -15; 15;19]; [21; 15; 31];
[22; -10; 15;32]; [23; -15; 27]; [24; 10; 18;28;29]; [25; 10; 19;24]; [26; 15; 20];
[27; 10; 32;33]; [28; 10; 99]; [29; 15; 28;30]; [30; -5; 25;26;29]; [31; 10; 30];
[32; 15; 22;27;33;34;99]; [33; -5; 27;32]; [34; 5; 99]; [99; 0]];;
```

```
let rec fils s ll = if s = hd (hd ll) then tl (tl (hd ll)) else fils s (tl ll);;
        (* renvoie la liste des sommets accessibles depuis s *)

let rec vue = fun [] -> print_string " somme = "; ()
        | (x::q) -> (print_int x; print_string " "; vue q);;

let rec val s = fun [] -> 0
                | (x::q) -> if s = hd x then hd (tl x) else val s q;; (* valeur de s dans g *)

let rec explo ch fr sp fi sol sm g =
        (* ch chemin, fr liste des listes des frères restant à explorer pour chq noeud de ch *)
        if ch = [] then (vue (rev sol); sm)
        else if hd ch = 99 then
        explo (tl ch) (tl fr) (sp - val (hd ch) g) (hd fr) (if sm < sp then ch else sol) (max sp sm) g
        else if fi = [] then explo (tl ch) (tl fr) (sp - val (hd ch) g) (hd fr) sol sm g (* retour *)
        else if mem (hd fi) ch then (explo ch fr sp (tl fi) sol sm g)         (* boucles interdites *)
        else (explo ((hd fi) :: ch) ((tl fi) :: fr) (sp + (val (hd fi) g)) (fils (hd fi) g) sol sm g);;
                (* examen du noeud suivant *)

let jeu g = explo [0] [[]] 0 (fils 0 g) [] 0 g;;
```

```
jeu g0;;
☞ 0 3 2 1 99  somme = 6
jeu g1;;
☞ 0 2 5 10 6 12 13 15 16 26 25 17 18 11 9 4 1 3 7 20 21 22 23 24 99  somme = 200
jeu g2;;
☞ 0 6 12 17 16 11 10 21 31 30 25 19 14 8 7 13 24 29 28 99  somme = 205
```

La seconde version testée sur G1 et G2, 20 fois de suite permet une mesure du temps d'éxecution et fait apparaître un gain de 25% par rapport à la première. Naturellement, ce que l'on gagne en évitant la lourdeur des données typées, on le perd en lisibilité (surtout avec l'artifice du 99) sauf à redonner les noms (ici des lettres grâce aux codes ASCII).

Remarque

Ce dernier point ne présente aucune gêne en Lisp où rien n'est typé, la structure de liste devient un fourre-tout et la puissante et universelle fonction "cond" qui n'a pas d'équivalent dans les autres langages entre dans presque toutes les définitions :
On représente le graphe G comme liste d'états, chaque état étant caractérisé par la liste formée de (nom, valeur, liste des noeuds acc).

Ci-dessous une simple traduction en Lisp où (Ch = chemin, liste des sommets (le dernier choisi en tête), Fr = liste correspondante des listes de sommets accessibles, Fi = liste des fils).

```
(defun explo (Ch Fr Sp Fi Sol Sm G) (cond ; Sm=meilleure somme, Sp = somme partielle,
        ((null Ch) (prin1 (reverse Sol)) (prin1 'somme-maximale=) Sm) ; fini, on renvoie Sm
        ((eq (car Ch) 'arr) ; un chemin est achevé
                (explo (cdr Ch) (cdr Fr) Sp (car Fr) (if (< Sm Sp) Ch Sol) (max Sm Sp) G))
        ((null Fi) (explo (cdr Ch) (cdr Fr) (- Sp (val (car Ch) G)) (car Fr) Sol Sm G)) ; retour
        ((member (car Fi) Ch) (explo Ch Fr Sp (cdr Fi) Sol Sm G)) ; boucle interdite
        (t (explo (cons (car Fi) Ch) (cons (cdr Fi) Fr)
                        (+ Sp (val (car Fi) G)) (fils (car Fi) G) Sol Sm G))))
```

```
(defun fils (E L) (if (eq E (caar L)) (cddar L) (fils E (cdr L))))
(defun val (E L) (if (eq E (caar L)) (cadar L) (val E (cdr L))))
(defun jeu (G) (explo (list 'dep) nil 0 (fils 'dep G) nil 0 G))
```

```
(set 'G0 '((dep 0 a b c) (a 1 dep arr b) (b 3 dep arr a c) (c 2 dep b) (arr 0)))

(set 'G1 '((dep 0 a b) (a 5 c d dep) (b 10 dep e f) (c -15 a d g h) (d 25 a c i) (e -15 b j k o q)
(f -20 b j l) ( g 20 c t) (h 5 c s u) (i 10 d k) (j 25 e f) (k 25 e i r s) (l 15 f m n) (m 5 l o) (n -5 l
p) (o 15 e m p) (p 20 n o q z) (q -5 e p r y) (r 20 k q x) (s -20 h k w) (t -5 g u) (u 15 t h v) (v
25 u w) (w 25 s v x) (x -20 r w arr) (y 35 q z) (z -15 p arr) (arr 0)))
```

CHAPITRE IX

TYPES RECURSIFS

Exemple de type défini par énumération

Commençons par dire ce qu'est un type défini par énumération, car passé ce premier exemple, ils vont prendre tout leur intérêt avec des définitions récursives.

```
type couleur = trefle | carreau | coeur | pique ;;
        ☞  Type couleur defined.
function trefle -> 1 | carreau -> 2 | coeur -> 3 | pique -> 4 ;;
        ☞ couleur -> int = <fun>
type mut = A of mutable int | B of mutable int*int;;
        ☞ Type mut defined.
```

1. Exemple de type récursif polymorphe, les arbres binaires

On définit les arbres binaires dont seules les feuilles sont étiquetées par des objets d'un certain type 'a, un arbre est donc soit une feuille, soit un noeud dont les deux fils sont eux-même des arbres. On définit alors récursivement avec une grande simplicité des fonctions simples comptant les noeuds, les feuilles, la longueur de la plus grande branche, et le prédicat "isom" indiquant que deux arbres ont même forme.

```
type 'a arbre = feuille of 'a | noeud of 'a arbre * 'a arbre ;;

let rec nbnoeuds = fun (feuille _) -> 1 | (noeud(a, b)) -> 1 + nbnoeuds(a)+nbnoeuds(b);;
let rec nbfeuilles = fun (feuille _) -> 1 | (noeud(a, b)) -> nbfeuilles(a)+nbfeuilles(b);;
let rec prof = fun (feuille _) -> 0 | (noeud (a, b)) -> 1 + max (prof a) (prof b);;

    let a = noeud (feuille 3, noeud (noeud (feuille 4, feuille 6), feuille 5));;
            (* est un exemple construit *)
    nbfeuilles a;;     ☞ int = 4
    nbnoeuds a ;;      ☞ int = 7
    prof a ;;          ☞ int = 3
```

L'exemple de l'arbre a est donc : et b est :

3 5 4 6

sic mundi transit gloria

```
let rec isom  =  fun       | (feuille _) (feuille _) -> true
                           | (noeud (ag, ad)) (noeud (bg, bd)) -> isom ag bg && isom ad bd
                           | _ -> false;;
    ☞ isom : 'a arbre  -> 'b arbre -> bool = <fun>
```

```
let b = noeud (feuille "sic", noeud (noeud (feuille "transit", feuille "gloria"), feuille "mundi"));;
isom a a;; ☞ true

isom a (noeud(feuille 4, feuille 7)));; ☞ false
```

2. Arbres binaires ordonnés et équilibrés

Pour un arbre binaire étiqueté, on définit le type :

```
type 'a arb = vide | noeud of ('a * 'a arb * 'a arb);;
```

Pour un type 'a muni d'une relation d'ordre totale (nous prendrons les entiers dans tous les exemples), on dit qu'un arbre est ordonné si en chaque noeud, l'étiquette est supérieure à toutes celles de son sous-arbre gauche, et inférieure à toutes celles de son sous-arbre droit, par exemple :

```
let a = noeud(4, noeud(2, noeud(1, vide, vide), noeud(3, vide, vide)), noeud(5, vide, noeud(6, vide, vide)));;

let b = noeud(7, noeud(3, noeud(1, vide, vide), noeud(2, vide, vide)),
                    noeud(6, noeud(4, vide, vide), noeud(5, vide, vide)));;
```

La fonction "verif", teste si un prédicat p est vérifié par tous les noeuds d'un arbre, on s'en sert alors pour définir la fonction booléenne "orderd" = "être un arbre ordonné" :

```
let rec verif p = fun       | vide -> true  | noeud(e, g, d) -> p e & verif p g & verif p d;;
```

```
verif (fun x -> x mod 2 = 0) a;; ☞ false
verif (fun x -> x < 10) a;; ☞ true
```

```
let rec ordered = fun vide -> true
      | noeud(e, g, d) -> verif (prefix > e) g & verif (prefix < e) d & ordered g & ordered d;;
```

```
ordered a;; ☞ true
ordered b;; ☞ false
```

L'insertion d'un entier x dans un arbre étiqueté ordonné, consiste à rajouter un noeud bien placé avec cette valeur x, au sein de l'arbre :

```
let rec ins x  = fun vide -> noeud(x, vide, vide)
                 | noeud(e, g, d) -> if e < x then noeud(e, g, ins x d) else noeud(e, ins x g, d);;
```

```
ins 0 a;; ☞ noeud(4, noeud (2, noeud (1, noeud (0, vide, vide), vide), noeud (3, vide, vide)), noeud (5, vide, noeud (6, vide, vide)))
```

```
let inslist ll = it_list (fun a x -> ins x a) ll;;
      (* permet d'insérer successivement tous les éléments de ll dans un arbre *)
```

```
inslist a [7; 8; 5; 0; -2; 4; -1];;
☞
noeud(4, noeud (2, noeud (1, noeud (0, noeud (-2, vide, noeud (-1, vide, vide)), vide), vide), noeud (3, vide, noeud (4, vide, vide))), noeud (5, noeud (5, vide, vide), noeud (6, vide, noeud (7, vide, noeud (8, vide, vide)))))
```

On construit ci-dessous une fonction de création d'un arbre binaire dont tous les fils gauches sont vides avec des entiers de 1 à k à droite :

```
let rec creer k = if k > 0 then ins k (creer (k-1)) else vide;;
```

Verification d'un ordre et retour des valeurs min et max en ce cas. On définit un type spécial qui permet, soit de renvoyer un couple d'entiers, soit de lever une exception :

```
type 'a result = rien | res of 'a * 'a;;
exception nonord;;

let rec minmax = fun vide -> rien
        | (noeud(e, g, d)) -> let r1 = minmax g and r2 = minmax d in
        match r1, r2 with
                | rien, rien -> res(e, e)
                | rien, res(min2, max2) -> if e < min2 then res(e, max2) else raise nonord
                | res(min1, max1), rien -> if max1 < e then res(min1, e) else raise nonord
                | res(min1, max1), res(min2, max2) ->
                        if max1 < e & e < min2 then res(min1, max2) else raise nonord;;
```

```
minmax a;; ☞ int result = res (1, 6)
minmax b;; ☞   Exception non rattrapée: nonord
```

Remarque

Pour la même fonction en Lisp, naturellement sans type ni exception, on peut simplifier en représentant toute feuille par sa valeur et tout arbre par le triplet (e g d) où g et d sont les représentations des sous-arbres gauche et droit de la racine e.

```
(defun minmax (A) (cond        ((null A) nil )
                               ((atom A) (list A A))
                               (t (aux (car A) (minmax (cadr A)) (minmax (caddr A)))))
(defun aux (e g d) (cond       ((null g) (cond   ((null d) (list e e))
                                                  ((< e (car d)) (list e (cadr d)))
                                                  (t nil)))
                               ((null d) (if (< (cadr g) e) (list (car g) e) nil))
                               ((< e (cadr g)) nil)
                               ((< (car d) e) nil)
                               (t (list (car g) (cadr d)))))
```

```
(minmax '(4 (2 1 3) (5 () 6))) ☞ (1 6)
(minmax '(7 (3 1 2) (6 4 5))) ☞ nil
```

Parcours d'un arbre binaire quelconque en préfixé (racine-gauche-droite), en postfixé ou ordre polonais (gauche-droite-racine) et en infixé (gauche-racine-droite) :

```
let rec prefixe = fun vide -> [] | (noeud(e, g, d)) -> e :: (prefixe g) @ (prefixe d);;

let rec postfixe = fun vide -> [] | (noeud(e, g, d)) -> (postfixe g) @ (postfixe d) @ [e];;

let rec infixe = fun vide -> [] | (noeud(e, g, d)) -> (infixe g) @ e :: (infixe d);;
```

```
prefixe b;; ☞ [7; 3; 1; 2; 6; 4; 5]        postfixe b;; ☞ [1; 2; 3; 4; 5; 6; 7]
infixe b;; ☞ [1; 3; 2; 7; 4; 6; 5]         infixe a;; ☞ [1; 2; 3; 4; 5; 6]
```

On peut alors redéfinir "ordered" en vérifiant que la liste résultant du parcours infixe est ordonnée croissante ce qui permet de ne parcourir l'arbre qu'une fois :

```
let ordered a = croissante (infixe a)
        where rec croissante = fun [] -> true | [_] -> true | (x :: y :: q) -> x < y & croissante (y :: q);;
```

Application d'une fonction définie sur le type 'a pour tous les noeuds d'un arbre :

let rec **maparb** f = fun vide -> vide | (noeud(e, g, d)) -> noeud(f e, maparb f g, maparb f d);;

```
maparb (fun x -> char_of_int (64+x)) a;;
☞ noeud (`D`, noeud (`B`, noeud (`A`, vide, vide), noeud (`C`, vide, vide)), noeud (`E`,
vide, noeud (`F`, vide, vide)))
```

Itération d'une fonction f sur tous les noeuds d'un arbre, suivant un ordre de parcours :

let rec **itarbin** f x = fun vide -> x | (noeud(e, g, d)) -> itarbin f (f (itarbin f x g) e) d;;

let rec **itarbpre** f x = fun vide -> x | (noeud(e, g, d)) -> itarbpre f (itarbpre f (f x e) g) d;;

let rec **itarbpost** f x = fun vide -> x | (noeud(e, g, d)) -> f (itarbpost f (itarbpost f x g) d) e;;

```
itarbin (prefix +) 0 a;; ☞ 21
```

let **enfin** ll x = ll @ [x];; (* rajoute l'élément x à la fin de la liste ll *)

let **infixe** = itarbin enfin [];; (* est une autre définition du parcours infixe *)

```
iinfixe a;; ☞ [1; 2; 3; 4; 5; 6]
```

let prefixe = itarbpre enfin [] and postfixe = itarbpost enfin [];;

Construction d'un arbre équilibré à partir des éléments d'une liste :

```
let rec bal ll = match ll with [] -> vide
        | (x :: q) -> let k = (list_length ll) / 2 in noeud(x, bal(tete q k), bal (queue q k))
    where rec tete ll n = if n < 1 or ll = [] then [] else (hd ll) :: (tete (tl ll) (n - 1))
    and queue ll n = if ll = [] then [] else if n = 0 then ll else queue (tl ll) (n - 1);;
```

```
bal [2;3;1;4;5;6;9;8;7;2;4;5;6;8;3];; ☞ noeud (2,
        noeud (3, noeud (1, noeud (4, vide, vide), noeud (5, vide, vide)),
                noeud (6, noeud (9, vide, vide), noeud (8, vide, vide))),
        noeud (7, noeud (2, noeud (4, vide, vide), noeud (5, vide, vide)),
                noeud (6, noeud (8, vide, vide), noeud (3, vide, vide))))
```

3. Exemple de la dérivation formelle

On définit de manière simplifiée des expressions arithmétiques comme entiers, variables et additions ou multiplications d'expressions. Si une assignation de valeurs est donnée pour les variables, une expression peut alors être calculée :

type exp = Constant of int | Variable of string | Add of exp*exp | Mul of exp*exp;;

```
let rec eval ass = fun    (* ass est une "assignation" pour les variables*)
    | (Constant n) -> n
    | (Variable x) -> ass x
    | (Add (e, f)) -> eval ass e + eval ass f
    | (Mul (e, f)) -> eval ass e * eval ass f;;
```

```
let rec par = fun         (* est censé délivrer une chaîne parenthèsée *)
    | (Constant n) -> string_of_int n
    | (Variable x) -> x
    | (Add (e, f)) -> "(" ^ (par e) ^ "+" ^ (par f) ^ ")"    (* ^ est la concaténation des chaînes *)
    | (Mul (e, f)) -> "(" ^ (par e) ^ "*" ^ (par f) ^ ")";;
```

```
let rec par' = fun (* tient compte de la priorité de la multiplication sur la somme *)
        | (Constant n) -> string_of_int n
        | (Variable x) -> x
        | (Add (e, f)) -> "(" ^ (par' e) ^ "+" ^ (par' f) ^ ")"
        | (Mul (e, f)) -> (par' e) ^ "*" ^ (par' f);;
```

```
let e = Mul (Mul (Constant 2, Variable "x"), Add (Variable "x", Constant 1));;
☞ e: exp = Mul (Mul (Constant 2, Variable "x"), Add (Variable "x", Constant 1))

par e;; ☞ string = "((2*x)*(x+1))"
par' e;; ☞ string = "2*x*(x+1)"
```

```
let rec deriv x = fun           (* dérivation formelle *)
        | (Constant n) -> Constant 0
        | (Variable v) -> Constant (if v = x then 1 else 0)
        | (Add (u, v)) -> Add (deriv x u, deriv x v)
        | (Mul (u, v)) -> Add (Mul (deriv x u, v), Mul(u, deriv x v) ) ;;
        ☞ deriv : string -> exp -> exp = <fun>
```

```
deriv "x" e;; (* naturellement il faudrait une simplification après *)
☞
Add (Mul
        (Add (Mul (Constant 0, Variable "x"), Mul (Constant 2, Constant 1)),
        Add (Variable "x", Constant 1)),
        Mul (Mul (Constant 2, Variable "x"), Add (Constant 1, Constant 0)))
```

4. Dérivation un peu plus complète

En prenant des constantes réelles, les opérateurs opp, sqr, sin, cos, exo, ln, +, -, *, / on construit un type d'expressions avec les constructeurs Op1, Op2 pour ces opérateurs (de façon à pouvoir en rajouter). On écrit ensuite la fonction de dérivation d'une expression suivant une variable, puis une fonction de simplification (au moyen du point fixe) suffisamment étoffée pour vérifier le théorème de Schwarz sur des exemples. La simplification fait appel à des fonctions "d'assignation" qui associe par exemple la fonction sinus pour le mot "sin".

```
type expr = Cst of float | Var of string | Op2 of string*expr*expr | Op1 of string*expr;;
let rec deriv x = fun           (* dérivation formelle *)
        | (Cst _) -> Cst 0.
        | (Var v) -> Cst (if v = x then 1. else 0.)
        | (Op2 (f, u, v)) -> let u', v' = (deriv x u, deriv x v) in
                        if mem f ["+"; "-"] then Op2(f, u', v')
                        else if f = "*" then Op2("+", Op2(f, u', v), Op2(f, u, v'))
                        else Op2(f, Op2("-", Op2("*", u', v), Op2("*", u, v')), Op1("sqr", v))
        | (Op1 (f, u)) -> let u' = deriv x u in match f with "opp" -> Op1("opp", u') |
                "ln" -> Op2("/", u', u) | "exp" -> Op2("*", u', Op1("exp", u)) |
                "sin" -> Op2("*", u', Op1("cos", u)) |
                "cos" -> Op1("opp", Op2("*", u', Op1("sin", u))) |
                "sqr" -> Op2("*", Cst 2., Op2("*", u', u));;
```

```
let rec fixe f x = let x' = f x in if x = x' then x else fixe f x';;
```

Par exemple, prenons la fonction $f(x) = \sqrt{(2 - x)}$, son point fixe vérifie $x = f(x)$, c'est 1.

```
fixe (fun x -> sqrt (2. -. x)) 0.;; ☞ 1.0
```

```
let ass1 = fun "sin" -> sin | "cos" -> cos | "ln" -> log | "exp" -> exp
        | "opp" -> (fun x -> -.x) | "sqr" -> (fun x -> x *. x) | _ -> (fun x -> x)
and ass2 f x y = match f with "+" -> x +. y
        | "-" -> x -. y | "*" -> x *. y | "/" -> x /. y;;
```

```
let rec etape = fun
        | (Op1(f, u)) -> (match u with    | (Cst n) -> Cst(ass1 f n)
                                          | _ -> Op1 (f, etape u))
        | (Op2("*", Cst 0., _)) -> Cst 0.
        | (Op2("*", _, Cst 0.)) -> Cst 0.
        | (Op2("*", Cst 1., v)) -> v
        | (Op2("*", u, Cst 1.)) -> u
        | (Op2("+", Cst 0., v)) -> v
        | (Op2("+", u, Cst 0.)) -> u
        | (Op2("-", Cst 0., v)) -> Op1("opp", v)
        | (Op2("-", u, Cst 0.)) -> u
        | (Op2("/", Cst 0., _)) -> Cst 0.
        | (Op2("/", u, Cst 1.)) -> u
        | (Op2(f, Cst x, Op2(g, Cst y, v))) when f=g -> Op2((match f with "/" -> "*"
                                               | "-" -> "+" | _ -> f), Cst (ass2 f x y), v)
        | (Op2(f, u, v)) -> (match u, v with (Cst n), (Cst m) -> Cst(ass2 f n m)
                                               | _, _ -> Op2(f, etape u, etape v))
        | u -> u;;

let simplif = fixe etape;;
```

```
simplif (Op2 ("*", Op2("/", Op2("/", Cst 2., Cst 3.), Cst 5.0), Op1 ("sqr", Op2 ("*", Cst 2.0,
Cst 3.0))));; ☞ Cst 4.8

let u = Op2("-", Op2("*", Cst 2., Op1("sin", Op2("*", Cst 3., Var "x"))),
                    Op2("*", Cst 5., Op2("*", Op1("sqr", Var "y"), Op1("ln", Var "y"))));;

simplif (deriv "x" u);;
☞ Op2 ("*", Cst 6.0, Op1 ("cos", Op2 ("*", Cst 3.0, Var "x")))

let v = Op2("*", Op1("sqr", Var "x"), Op1("ln", Var "y"));;

simplif (deriv "y" (deriv "x" v));;
☞Op2 ("*", Op2 ("*", Cst 2.0, Var "x"), Op2 ("/", Cst 1.0, Var "y"))

simplif (deriv "x" (deriv "y" v)) = simplif (deriv "y" (deriv "x" v));; ☞ true
```

5. Calcul propositionnel

On définit les propositions du calcul propositionnel de manière récursive à partir du vrai et du faux, et de variables propositionnelles qui sont représentées par des chaînes de caractères et qui recevront une éventuelle assignation à vrai ou à faux. Evaluation par "val", d'une expression booléenne, p, q, r sont des variables propositionnelles qui pourront être assignées à vrai ou à faux. Puis les connecteurs de négation, conjonction, disjonction, implication permettent de construire des formules complexes. Cette complexité est mesurée par la fonction "haut".

```
type prop = Vrai | Faux | Var of string
            | Non of prop
            | Et of prop*prop
            | Ou of prop*prop
            | Imp of prop*prop;;

let rec val ass = fun     (* ass est une "assignation" pour les variables propositionnelles *)
        | Vrai -> true
        | Faux -> false
        | (Var x) -> (ass x)
        | (Non p) -> (not (val ass p))
        | (Ou (p1, p2)) -> ((val ass p1) or (val ass p2))
        | (Et (p1, p2)) -> ((val ass p1) & (val ass p2))
        | (Imp (p1, p2)) -> (not (val ass p1) or (val ass p2));;
```

```
let ass1 = fun "p"-> true | "q"-> false | "r"-> true
        and ass2 = fun "p"-> false | "q"-> false | "r"-> true | _ -> true ;;
(* ass1 et ass2 typés "char -> bool" mais ass1 n'est pas définit exhaustivement *)

val ass1 (Ou (Var "q", Et (Var "p", Var "r")));;     ☞ bool = true
val ass1 (Imp (Et (Imp (Var "p", Var "q"), Imp (Var "q", Var "r")), Imp (Var "p", Var "r")));;
        ☞ bool = true (* vrai aussi avec toute autre assignation *)
```

Profondeur ou "hauteur" d'une expression

```
let rec haut = fun
    | Vrai -> 0
    | Faux -> 0
    | (Var _) -> 0
    | (Non p) -> 1+(haut p)
    | (Et (p, q)) -> 1+(max (haut p) (haut q)) | (Ou(p, q)) -> (max (haut p) (haut q)) + 1
    | (Imp(p, q)) -> (max (haut p) (haut q)) + 1;;
```

Simplification d'une expression booléenne en utilisant quelques règles :

```
let rec simp = fun Vrai -> vrai | Faux -> faux | (Var v) -> Var v
    | (Et (p, q)) -> (match simp p, simp q with
        | (Vrai, p) -> p | (p, Vrai) -> p | (Faux, p) -> Faux | (p, Faux) -> Faux
        | (p, q) -> if p = q then p else if p=non(q) or q = non(p) then Faux else Et(p,q))
    | (Ou (p, q)) -> (match simp p, simp q with
        | (vrai, p) -> Vrai | (p, Vrai) -> Vrai | (Faux, p) -> p | (p, Faux) -> p
        | (p,q) ->if p=q then p else if p = Non(q) or q = Non(p) then Vrai else Ou(p,q))
    | (Imp (p, q)) -> simp (Ou (simp (Non p), q))
    | (Non p) -> match simp p with Vrai -> Faux | Faux -> Vrai | Non(q) -> q | q -> Non(q);;

simp (Et(Non(Non(vrai)), Ou(Imp(vrai, Var "p"), et(Var "p", Var "p"))));;
☞ prop = Var "p"

let rec vue = fun Vrai -> "1" | Faux -> "0" | (Var p) -> p
    | (Non p) -> "¬(" ^ (vue p) ^ ")"
    | (Et(p, q)) -> (vue p) ^ "." ^ (vue q) | (Ou(p, q)) -> "(" ^ (vue p) ^ " ou " ^ (vue q) ^ ")"
    | (Imp(p, q)) -> "(" ^ (vue p)^" -> " ^ (vue q) ^ ")";;

vue ( simp (Et (Non (Non (Vrai)), Ou (Imp (Vrai, Var  "p"), Et (Var  "p", Var  "p")))));;
☞ string = "p"
```

6. Décomposition sous formes normales disjonctives dans le calcul propositionnel

Les littéraux sont les formules atomiques ou leurs négations, les décompositions sous formes normales disjonctives ou conjonctives, sont les formules logiques s'expriment repectivement disjonction de "monômes", c'est à dire de conjonction de littéraux, ou comme conjonction de disjonctions.
On visualisera les formes normales disjonctives par des listes de monômes écrits sous la forme habituelle, notamment avec le symbole ¬ pour la négation.

```
let litt = fun    | Vrai -> true
                  | Faux -> true
                  | (Var _) -> true
                  | (Non (Var _)) -> true
                  | _ -> false;;

let rec monome = fun (Et(p, q)) -> (monome p) & (monome q) | x -> litt x ;;

monome (Et( Vrai, Et( Faux, Non ( Var "r"))));;   ☞ bool = true
```

```
| monome (Ou( Et( Var "p", Var"q"), vrai));;          ☞ bool = false

let rec somme = fun (Ou(p, q)) -> (somme p) & (somme q) | x -> litt x;;

let rec plate = fun [] -> []
            | ([] :: q) -> plate q
            | ((a :: q) :: m) -> a :: plate (q :: m);;
(* Décomposition sous forme normales disjonctive ou conjonctive : *)

let rec disj p = if monome p then [p] else (fun
            | (Ou(p, q)) -> (disj p) @ (disj q)
            | (Non q) -> map (fun x -> Non x) (conj q)
            | (Et(p, q)) -> let p' = disj p in plate (map disj (map (fun x -> Et(q, x)) p'))
            | (Imp(q, r)) -> (disj (Non q)) @ (disj r) )   p

    and conj p =  if somme p then [p]
            else (fun (Et(q, r)) -> (conj q)@(conj r)
            | (Non q) -> map (fun x -> Non x) (disj q)
            | (Ou(p, q)) -> let p' = conj p in plate (map conj (map (fun x -> Ou(q, x)) p'))
            | (Imp(q, r)) -> (conj (Non (Et(q, Non r)))) )   p ;;

| map vue (disj (Ou (Et (Var "p", Ou (Et (Var "q", Non (Var "r")), Et (Non(Var "q"), Var  "r"))),
|         (Et (Et (Var "r", Non (Var "p")), Ou (Var "q", Non (Var "q")))))));; (* petit exemple *)
| ☞ string list = ["p.q.¬(r)"; "p.¬(q).r"; "r.¬(p).q"; "r.¬(p).¬(q)"]
```

Ou bien, on définit une première fonction qui élimine les implications et rentre les négations jusqu'aux variables, puis grâce à la fonction "point fixe", on se contente de dire avec "étape" qu'il faut distribuer le "et" sur le "ou", et enfin "fixe étape" va chercher le "pont fixe", c'est à dire lorsque plus aucune modification n'est possible.

On réalisera ainsi la forme normale disjonctive. Enfin "listou" réalise la liste des listes constituées par les monômes.

```
let rec  rentreneg = function
    | Non(Et(p, q)) -> Ou(rentreneg (Non p), rentreneg (Non  q))
    | Non(Ou(p, q)) -> Et(rentreneg (Non p),  rentreneg (Non q))
    | Non(Non p) -> rentreneg p
    | Non(imp(p, q)) -> Et(rentreneg p, rentreneg(Non q))
    | Imp(p, q) -> Ou(rentreneg(Non p), rentreneg q)
    | Ou(p, q)  -> Ou(rentreneg p, rentreneg q)
    | Et(p, q) -> Et(rentreneg p, rentreneg q)
    | p -> p;;

| rentreneg (Imp (Et (Var "p", Ou (Imp (Var "q", Non (Var "r")), Var "p")), Var "q"));;
| ☞ Ou (Ou (Non (Var "p"), Et (Et (Var "q", Var "r"), Non (Var "p"))), Var "q")

let rec fixe f x = let x' = f x in if x = x' then x else fixe f x';;

| fixe (fun x -> sqrt (2. -. x)) 0.;; ☞ 1.0

let rec etape = function
    | Et(Ou(p, q), r) -> Ou(Et(p, r), Et(q, r))
    | Et(p, Ou(q, r)) -> Ou(Et(p, q), Et(p, r))
    | Et(p, q) -> Et(etape p, etape q)
    | Ou(p, q) -> Ou(etape p, etape q)
    | p -> p;;

| fixe etape (Ou (Ou (Non (Var "p"), Et (Et (Var "q", Var "r"), Non (Var "p"))), Var "q"));;
| ☞ Ou (Ou (Non (Var "p"), Et (Et (Var "q", Var "r"), Non (Var "p"))), Var "q")
```

```
let rec listet = function Et(p, q) -> (listet p)@ (listet q)
                        | Non(var p) -> ["¬"^p]
                        | Var p -> [p];;
let rec listou = function Ou(p, q) -> (listou p) @ (listou q)
                        | p -> [listet p];;
```

```
listou (Ou (Ou (Non (Var "p"), Et (Et (Var "q", Var "r"), Non (Var "p"))), Var "q"));;
☞ [["¬p"]; ["q"; "r"; "¬p"]; ["q"]]
```

7. Refutabilité d'une théorie

Un littéral est une variable propositionnelle ou sa négation, une clause est une disjonction de littéraux représentée par une liste, et enfin une théorie est une conjonction de clauses représentée par une liste de listes.
Une clause sera une tautologie si elle contient a ainsi que ¬a, par contre, la clause vide représentera le "faux".

```
type litteral = Var of string | Neg of string;;

let contraire = fun (Var x) -> Neg x | (Neg x) -> Var x
and variable = fun ((Var x)::_) -> x | ((Neg x)::_) -> x;;
        (* renvoie la première variable d'une clause *)

let rec tauto = fun
        | [] -> false
        | (x :: q) -> (mem (contraire x) q) || tauto q;; (* || est, comme "or", la disjonction *)

let rec remplace p val c = match c with
        | [] -> [] (* renvoie la clause c où p est remplacée par val = vrai, faux *)
        | (l :: q) -> if variable c <> p then l :: (remplace p val q)
                else if (val & l = Var p) || (not val & l = Neg p) then [l; contraire l]
                else remplace p val q;;

let rec assigne p val = map (fun c -> remplace p val c);;
        (* remplace p par val dans une liste de clauses *)
```

Une variable propositionnelle p est dite pure, si elle apparait toujours de la même façon, soit p, soit ¬p. En ce cas, si p est pure (positive) , de première occurrence dans la clause c, c et S sera réfutable si et eulement si S[p <- vrai] l'est (algorithme de Loveland).

```
let rec mem2 x = fun [] -> false | (l::q) -> mem x l || mem2 x q;;
        (* appartenance au second niveau *)

let rec pure q = fun [] -> [] | (a :: r) -> if mem2 (contraire a) q then pure q r
                        else match a with Var p -> assigne p true q | Neg p -> assigne p false q;;

let rec refutable lcl = match lcl with
        | [] -> false (* développe l'arbre en remplaçant chq variable par vrai ou faux *)
        | ([Var p] :: q) -> refutable (assigne p true q)
        | ([Neg p] :: q) -> refutable (assigne p false q)
        | [] :: _ -> true
        | (c :: q) -> if tauto c then refutable q
                else let s = pure q c in if s <> [] then refutable s
                else let v = variable c in
                        (refutable (assigne v true q)) & (refutable (assigne v false q));;
```

```
refutable [[Var "a"]; [Neg "a"]];; ☞ true
refutable [[Var "a"; Var "b"]; [Var "b"]];; ☞ false
refutable [[Var "a"; Neg "a"]];; ☞ false
refutable [[Var "a"]; [Var "b"]; [Neg "a"; Neg "b"]];; ☞ true
```

8. Lambda-calcul

Le λ-calcul est le plus petit ensemble de termes contenant un ensemble dénombrable de variables et stable par «l'application» (uv) si u et v sont des termes, et par "l'abstraction" λx t où x est une variable et t un terme. On définit une α–équivalence des termes λx u et λx'u' dans la mesure où u' est équivalent à u dans lequel x' est substitué à toutes les occurences de x (ils ne diffèrent que par des changements de nom des variables). Λ est l'ensemble quotient.
La β-réduction consiste à déduire d'un terme, un terme sans "rédex" c'est à dire sans sous-terme de la forme (λx u)t, terme que l'on remplace par u[où t prend la place de x]. Celà signifie concrètement que l'on applique la fonction u de variable x au terme t. La forme normale (sans rédex) est unique dans Λ.
On pose :

I = λx x (l'identité) et :
F = λx λy y (le faux ou zéro),
K = V = λx λy x (le vrai), ensuite peuvent être définis la condition :
IF = λb λx λy b x y, puis la négation :
$\neg$ = λx (IF x F V), les connecteurs :
ET = λx λy x y F, OU = λx λy x V y, enfin les entiers de Church :
$\underline{0}$ = F, $\underline{1}$ = λf λx (f x), $\underline{2}$ = λf λx (f (f x)) et $\underline{n}$ = λf λx (f^k x).

On dit alors que la fonction f de N^n dans N est représentable en λ-calcul par F si et seulement si pour tous entiers $k_1, ..., k_n$ si $f(k_1, ..., k_n)$ n'est pas défini alors l'expression F$\underline{k_1}, ..., \underline{k_n}$ n'est pas réductible sinon si $f(k_1, ..., k_n) = k$ alors l'expression F $\underline{k_1}, ..., \underline{k_n}$ se réduit en $\underline{k}$. On peut en effet construire toutes les fonctions récursives telles que :

SUC = λn λf λx ((n f) (f x)), la somme :
+ = λn λm λf λx ((n f) ((m f) x)) ou encore :
* = λn λm λf λx ((n (m f)) x).

On construit à l'intérieur de Caml un petit langage fonctionnel, c'est à dire un évaluateur d'expressions, lesquelles expressions seront des constantes entières (pour se limiter) ou des fonctions (donc de "int" vers "int"). On ne fait pas de renommage de variables (α–conversion) pour l'instant.

```
type expr = Var of string | Cst of int | Plus of expr*expr | Fonc of string*expr | App of expr*expr;;

    let e1 = Fonc("x", Var "x");; (* est la fonction identité *)
    let e2 = App(Fonc("x", Fonc("y", Plus( Var "x", Var "y"))), Cst 3);; (* = (x -> (y -> x+y)) 3 *)
    let e3 = Fonc("x", Plus(Var "x", Var "y"));;

let rec ret x = fun [] -> [] | (y :: q) -> if x = y then ret x q else y:: (ret x q);;
let rec libres = fun (Var x) -> [x] | (Cst x) -> [] | (Plus(e1, e2)) -> (libres e1) @ (libres e2)
              | (App(e1, e2)) -> (libres e1) @ (libres e2) | (Fonc(x, e)) -> ret x (libres e);;

    libres e1;; ☞ []
    libres (Plus(Var "x", Cst 7));; ☞ ["x"]
    libres e3;;☞ ["y"]

let rec subst x t = fun
    | (Var y) -> if x=y then t else Var y
    | (Cst y) -> Cst y | (Plus(e, f))-> Plus(subst x t e, subst x t f)
    | (App(e, f)) -> App(subst x t e, subst x t f)
    | (Fonc(y, e)) -> if x=y then Fonc(y, e)
                else if mem y (libres t) then failwith ("capture de la variable "^y)
                else Fonc(y, subst x t e);;

    subst "x" (Var "y") e3;; ☞ Fonc ("x", Plus (Var "x", Var "y"))
    subst "y" (Plus(Var "x", Cst 1)) e3;;
    ☞ Exception non rattrapée: Failure "capture de la variable x"
```

Il ne faut pas que l'on subtitue une variable liée dans une expression par un terme t qui contiendrait cette variable libre (capture de variable).
Pour évaluer une expression dans ce langage fonctionnel, il faut la réduire au maximum, notamment l'application d'une fonction (x -> e) sur un terme t est e[x <- t] c'est à dire subst x t e. On réécrit donc de gauche à droite une fois puis on fait converger un enchaînement de réécritures.

```
let rec fixe f x = let x' = f x in if x = x' then x else fixe f x';;

let valeur = fun (Cst _) -> true
                | (Fonc(_, _)) -> true
                | _ -> false;;

let rec reecrire = fun
        | (Plus(Cst n, Cst m)) -> Cst (n+m)
        | (Plus(e, f)) -> if valeur e then Plus(e, reecrire f) else Plus(reecrire e, f)
        | (App(Fonc(x, e), f)) -> if valeur f then subst x f e else App(Fonc(x, e), reecrire f)
        | (App(e, f)) -> App(reecrire e, f)
        | e -> e;;
```

```
reecrire e2;; ☞ Fonc ("y", Plus (Cst 3, Var "y"))
```

```
let evaluer e = if libres e = [] then fixe reecrire e else failwith "impossible";;
```

```
evaluer e1;; ☞ Fonc ("x", Var "x")
evaluer e3;; ☞ "impossible"
evaluer (App(e1, Cst 5));; ☞ Cst 5
```

On définit à présent un "script" comme une suite de déclarations permettant d'affecter un nom à une valeur (numérique ou fonctionnelle) ou d'évaluations. Son évaluation sera la liste des évaluations. Par exemple la "fiche" :

```
let suc = fun x -> x + 1;;
suc 4;;
let y = 7;;
let phi = fun z -> z + y;;
phi(y);;
```

```
type ligne = Eval of expr | Decl of string*expr;;
```

```
let fiche = [Decl("suc", Fonc("x", Plus(Var "x", Cst 1))); Eval (App(Var "suc", Cst 4));
Decl("y", Cst 7); Decl("phi", Fonc("z", Plus(Var "z", Var "y"))); Eval (App(Var "phi", Var
"y"))];;
```

```
let rec reecfich = fun
        | [] -> []
        | ((Eval e) :: q) -> if valeur e then (Eval e) :: (reecfich q) else (Eval (evaluer e)) :: q
        | ((Decl(n, e)) :: q) -> map (fun (Eval ex) -> Eval (subst n e ex)
        | (Decl(f, ex)) -> Decl(f, subst n e ex)) q;;

let evalfich = fixe reecfich;;
```

```
evalfich fiche;; ☞ ligne list = [Eval (Cst 5); Eval (Cst 14)]
```

9. Algorithme d'unification de Robinson - Pitrat

Il s'agit de reconnaître un motif dans une expression, par exemple le motif x + (y + z) filtrera l'expression u + (3x + 2), à condition qu'on réalise des substitutions x <- u, y <- 3x, z <- 2. Le but évident est de faire du calcul symbolique en appliquant des règles de réécriture.

On va se contenter de termes écrits avec une addition, rajouter d'autres opérations ne pose pas de difficulté, par contre, il faut éventuellement renommer des variables. Une substitution va être représentée par une liste d'association (Assoc prédéfini renvoie éventuellement l'exception "Not_found").

```
type expr = Var of string | Cst of int | Plus of expr*expr  ;;


let rec applique s = fun
        | (Var x ) -> (try assoc  x s with Not_found -> Var x )
        | (Cst n )-> Cst n
        | (Plus(e1, e2)) -> Plus (applique s e1, applique s e2);;

        applique [("x", Var "y"); ("y", Cst 3)] (Plus(Var "x", Plus(Var "y", Cst 2)));;
        ☞ [ Plus (Var "y", Plus (Cst 3, Cst 2))


let rec unif m  e = match (m, e ) with
        | (Cst n) , (Cst p) -> if n = p then [] else failwith "echec"
        | (Var x), e -> [(x, e)]
        | (Plus(m1, m2)), (Plus(e1, e2)) -> let s' = unif m1 e1 in s' @ (unif (applique s' m2) e2)
        | _, _ -> failwith "echec";;

        unif (Plus(Var "x", Plus(Var "y", Cst 2))) (Plus(Var "y", Plus(Var "x", Cst 2)));;
        ☞ ["x", Var "y"; "y", Var "x"]

        unif (Plus(Var "x",Var "y")) (Plus(Cst 3, Cst 2));;
        ☞ ["x", Cst 3; "y", Cst 2]
        unif (Plus(Var "x",Var "x")) (Plus(Cst 3, Cst 2));;
        ☞   Failure "echec"
```

10. Traitement de la récursivité

Cet exemple est un des plus intéressant, car il permet d'entrevoir comment un langage fonctionnel peut gérer la récursivité.

On rajoute, en vue d'un exemple, la soustraction et la multiplication ainsi que le test "négatif", puis grâce à "Letrec", des définitions éventuelles de fonctions récursives. On modifie également "libres" et "subst".

```
type expr = Var of string
        | Cst of int
        | Sub of expr*expr
        | Mul of expr*expr
        | Fonc of string*expr
        | App of expr*expr
        | Letrec of string*expr
        | Neg of expr*expr*expr;;

let rec ret x = fun [] -> []
                (y :: q) -> if x = y then ret x q else y :: (ret x q);; (* bien sûr inchangé *)
```

```
let rec libres = fun (Var x) -> [x]
        | (Cst x) -> []
        | (Sub(e1, e2)) -> (libres e1) @ (libres e2)
        | (Mul(e1, e2)) -> (libres e1) @ (libres e2)
        | (Neg(e1, e2, e3)) -> (libres e1) @ (libres e2)@(libres e3)
        | (Fonc(x, e)) -> ret x (libres e)
        | (App(e1, e2)) -> (libres e1) @ (libres e2)
        | (Letrec(f, e)) -> libres e;;

        libres (Fonc ("x", Mul(Var "x", Sub(Var "x", Cst 1))));; ☞ []
        libres (Letrec ("f", Fonc("x", App(Var "f", Sub(Var "x", Cst 1)))));; ☞   ["f"]

let rec subst x t = fun (Var y) -> if x=y then t else Var y
        | (Cst y) -> Cst y
        | (Sub(e, f))-> Sub(subst x t e, subst x t f)
        | (Mul(e, f))-> Mul(subst x t e, subst x t f)
        | (App(f, e)) -> App(subst x t f, subst x t e)
        | (Fonc(y, e)) -> if x=y then Fonc(y, e)
                    else if mem y (libres t) then failwith ("capture de la variable "^y)
                    else Fonc(y, subst x t e)
        | (Letrec(f, e))-> if x = f then Letrec(f, e)
                    else if mem f (libres t) then failwith ("capture de "^f)
                    else Letrec(f, subst x t e)
        | (Neg(e1, e2, e3))-> Neg(subst x t e1, subst x t  e2, subst x t e3);;

let rec  chaine = fun (Var x )-> x
        | (Cst n) -> string_of_int n
        | (Sub(a, b)) -> (chaine a) ^ "-" ^ (chaine b)
        | (Mul(a, b)) -> (chaine a) ^ "*" ^ (chaine b)
        | (App(f, e)) -> "("^(chaine f) ^ " (" ^ (chaine e) ^ "))"
        | (Fonc(x, e)) -> "(" ^ x^" -> " ^ (chaine e) ^ ")"
        | (Neg(t, a, b)) -> "si " ^ (chaine t) ^ " < 0 alors " ^ (chaine a) ^ " sinon " ^ (chaine b)
        | (Letrec(f, e)) -> f ^ " = " ^ (chaine e)
and voir e= print_string (chaine e);;

        let ex = Letrec("f", Fonc("x", Neg(Var "x", Cst 1, Mul(Var "x",
                                        App(Var "f", Sub(Var "x", Cst 1))))));;
        voir ex;; ☞ f = (x -> si x < 0 alors 1 sinon x*(f (x-1)))
```

Letrec n'est autre qu'une définition particulière de fonction qui permettra, lors de l'évaluation, de les distinguer des expressions débutant par Fonc (beaucoup de langages peuvent aller voir en avant si la fonction déclarée est récursive).
Dans ce qui suit tout n'est pas prévu, bien sûr, en matière de collision de nom de variables, mais le plus important y est : lors d'une définition avec Letrec(f, e), une expression ff = Fonc(f, e) est créée, ff est donc à deux variables mais curryfiée, et non récursive. Letrec f e se transforme en fix ff, qui se transforme lui-même en ff (fix ff).

```
let rec fixe f x = let x' = f x in if x = x' then x else fixe f x';;

let rec reecrire = fun (Sub(Cst n, Cst m)) -> Cst (n-m)
        | (Mul(Cst n, Cst m)) -> Cst (n*m)
        | (Neg(Cst n, a, b)) -> if n <= 0 then a else  b
        | (Neg(t, a, b)) -> Neg(reecrire t, reecrire a, reecrire b)
        | (Sub(e, f)) -> Sub(reecrire e, reecrire f)
        | (Mul(e, f)) ->  Mul(reecrire e, reecrire f)
        | (App(Var  "fix",  ff)) -> App(ff, App(Var "fix",  ff ))
        | (App(Fonc(x, e),  v)) -> subst x  v e
        | (App(f, e)) -> App(reecrire f,  reecrire e)
        | (Letrec(f, e)) -> App(Var "fix",  Fonc(f, e))| e -> e;;
```

```
let evaluer = fixe  (fun e -> print_newline(); voir e; reecrire  e);;
        (* donne en même temps une trace *)
```

```
evaluer (App(ex, Cst 3));;
(f = (x -> si x < 0 alors 1 sinon x*(f (x-1))) (3))
((fix ((f -> (x -> si x < 0 alors 1 sinon x*(f (x-1)))))) (3))
(((f -> (x -> si x < 0 alors 1 sinon x*(f (x-1)))) ((fix ((f -> (x -> si x < 0 alors 1 sinon x*(f (x-
1)))))))) (3))
((x -> si x < 0 alors 1 sinon x*((fix ((f -> (x -> si x < 0 alors 1 sinon x*(f (x-1)))))) (x-1))) (3))
si 3 < 0 alors 1 sinon 3*((fix ((f -> (x -> si x < 0 alors 1 sinon x*(f (x-1)))))) (3-1))
3*((fix ((f -> (x -> si x < 0 alors 1 sinon x*(f (x-1)))))) (3-1))
3*(((f -> (x -> si x < 0 alors 1 sinon x*(f (x-1)))) ((fix ((f -> (x -> si x < 0 alors 1 sinon x*(f (x-
1)))))))) (2))
3*((x -> si x < 0 alors 1 sinon x*((fix ((f -> (x -> si x < 0 alors 1 sinon x*(f (x-1)))))) (x-1)))
(2))
3*si 2 < 0 alors 1 sinon 2*((fix ((f -> (x -> si x < 0 alors 1 sinon x*(f (x-1)))))) (2-1))
3*2*((fix ((f -> (x -> si x < 0 alors 1 sinon x*(f (x-1)))))) (2-1))
3*2*(((f -> (x -> si x < 0 alors 1 sinon x*(f (x-1)))) ((fix ((f -> (x -> si x < 0 alors 1 sinon x*(f
(x-1)))))))) (1))
3*2*((x -> si x < 0 alors 1 sinon x*((fix ((f -> (x -> si x < 0 alors 1 sinon x*(f (x-1)))))) (x-1)))
(1))
3*2*si 1 < 0 alors 1 sinon 1*((fix ((f -> (x -> si x < 0 alors 1 sinon x*(f (x-1)))))) (1-1))
3*2*1*((fix ((f -> (x -> si x < 0 alors 1 sinon x*(f (x-1)))))) (1-1))
3*2*1*(((f -> (x -> si x < 0 alors 1 sinon x*(f (x-1)))) ((fix ((f -> (x -> si x < 0 alors 1 sinon
x*(f (x-1)))))))) (0))
3*2*1*((x -> si x < 0 alors 1 sinon x*((fix ((f -> (x -> si x < 0 alors 1 sinon x*(f (x-1)))))) (x-
1))) (0))
3*2*1*si 0 < 0 alors 1 sinon 0*((fix ((f -> (x -> si x < 0 alors 1 sinon x*(f (x-1)))))) (0-1))
3*2*1*1
3*2*1
3*2
6-
☞ Cst 6
```

11. Arithmétique de Peano

Le système de Peano est l'un des premiers systèmes formels (1899) construit au dessus du calcul des prédicats. L'alphabet des symboles utilisés est un ensemble dénombrable de variables V et les symboles $\{\neg, \vee, \exists, 0, S, +, *, =\}$ où S va se réaliser comme la fonction successeur. Les connecteurs $\wedge$ et $\rightarrow$ n'étant que des abréviations.
Les axiomes et règles sont ceux de la logique des prédicats, plus les axiomes de l'égalité et les 6 axiomes dits de Peano :

$\forall x \forall y$

$$Sx = Sy \rightarrow x = y$$
$$x + 0 = x$$
$$\neg(Sx = 0)$$
$$x + Sy = S(x + y)$$
$$x*0 = 0$$
$$x*Sy = (x*y) + x$$

Ainsi que le schéma d'axiome de récurrence (ce n'est pas un axiome, mais une famille infinie d'axiomes), pour toute formule F à une variable libre :

$$[F(0) \wedge \forall x\, F(x) \rightarrow F(Sx)\,] \rightarrow \forall y\, F(y)$$

On montre que l'arithmétique (du premier ordre) n'est pas finiment axiomatisable. Le résultat logique le plus important est que la consistance (ou non-contradiction) de la théorie de

l'arithmétique est exprimable dans cette théorie par une formule, mais celle-ci ne peut être ni démontrée, ni réfutée.

Cet énoncé de non-contradiction est donc indécidable et montre "l'incomplétude" de l'arithmétique.
C'est le célèbre théorème de Gödel (1940) établi grâce à une numérotation de toutes les formules et démonstrations. Le théorème de Tarski (1944) montre pour sa part qu'il n'existe pas de formule qui puisse assurer de la validité d'un énoncé. On va redéfinir les entiers et leur addition et multiplication en suivant les axiomes :

```
type entier = zero | S of entier;;

let un = S zero and deux = S (S zero) and trois = S (S (S zero));;

let rec add n m = match n with zero -> m | S k -> S (add k m);;

let rec mul n m = match n with zero -> zero | S k -> add m (mul k m);;

let rec puis n m = match m with zero -> S zero | S k -> mul n (puis n k);;

    mul deux trois;;
    ☞ S (S (S (S (S (S zero)))))
```

Comme pour tout type récursif on peut définir une fonctionnelle d'itération d'une fonction f avec un point de départ e, l'argument de la fonction:

```
let rec itentier f e = fun zero -> e | (S k) -> f (itentier f e k);;
    ☞ itentier : ('a -> 'a) -> 'a -> entier -> 'a
```

Cette fonctionnelle permet alors de redéfinir :
Pour additionner x à un certain y, il faut itérer y fois l'opération successeur à partir de x, pour multiplier par x un certain argument y, il faut répéter l'addition de x à partir de zéro, y fois, enfin pour définir les puissanes :

```
let add x = itentier S x;;

let mul x = itentier (add x) zero;;

let puis x = itentier (mul x) un;;

    puis deux trois;;
    ☞ S (S (S (S (S (S (S (S zero)))))))
```

Conversion de ces entiers "artificiels" vers les "int" du Caml :

```
let int_of_entier = itentier (fun x -> x+1) 0;;

    int_of_entier (S (S (S (S (S (S trois))))));;
    ☞ 9

let pair = itentier (fun x -> not x) true;;

    pair trois;; ☞ false
    pair (S (S (S (S (S (S zero))))));; ☞ true

let nul = itentier (fun x -> false) true;;

    nul deux;; ☞ false
    nul zero;; ☞ true
```

12. Ordinaux et cardinaux

Dans la théorie des ensembles de Zermelo, les objets de l'univers sont appelés "ensembles" et la relation binaire ∈ est dite "appartenance".
L'inclusion ⊂ est une simple abréviation définie par $x \subset y \Leftrightarrow \forall z\ z \in x \Rightarrow z \in y$, et un ordinal a est défini comme un ensemble où ∈ est un bon ordre et $\forall x\ x \in a \Rightarrow x \subset a$.
On vérifie que zéro = ∅, un = {∅}, deux = {∅, {∅}} sont des ordinaux, tous les éléments d'un ordinal sont des ordinaux; que la collection de tous les ordinaux est totalement et bien ordonnée pour ∈, et que le successeur d'un ordinal a (le plus petit majorant strict) est l'ensemble $a \cup \{a\}$, noté a + 1 (a est dit prédécesseur de a+1).
Définitions :
Ordinal fini a ⇔ [$\forall b \leq a$ et $b \neq 0 \Rightarrow$ b a un prédécesseur]
Ordinal limite : ordinal sans predécesseur

L'axiome de l'infini est simplement l'existence d'un ordinal non fini.

Le premier ordinal limite est l'ensemble ω des ordinaux finis (en même temps le cardinal "dénombrable"). L'ordinal suivant est donc ω + 1 et le second ordinal limite est ω + ω.
On définit le type ordinal comme :

```
type ord = Zero | Suc of ord | Lim of (int -> ord);;

let rec nat n = if n=0 then Zero else Suc (nat (n-1));; (* conversion entier -> ordinal *)

        nat 3;;
        ☞ Suc (Suc (Suc Zero))

let omega = Lim nat;;
let rec nom no =
        let ln = ["zero"; "un"; "deux"; "trois"; "quatre"; "cinq"; "six"; "sept"; "huit"; "neuf"; "dix"] in
        match no with
                | Zero -> "zero"
                | Lim nat -> "omega" | Lim _ -> "lim "^"..." | _ -> aux no ln
        and aux = fun
                | Zero (nom :: _) -> nom
                | (Suc x) [n] -> "suc " ^ (aux x [n])
                | (Suc x) (n :: q) -> aux x q
                | (Lim f) (n :: q) -> (nom (Lim f)) ^ " + " ^ n ;;

        nom (Suc (Suc (Suc (Suc Zero))));; ☞ "quatre"
        nom (Suc (Suc (Lim nat)));; ☞ "omega + deux"
        nom (Suc (Suc (Suc omega)));; ☞ "omega + trois"
        nom (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc Zero))))))))))));;
        ☞ "suc suc dix"
        nom (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc omega))))))))))));;
        ☞  "suc suc omega + dix"

let rec orden n = if n=0 then omega else Suc (orden (n-1));;
                (* conversion entier -> ordinal dénombrable *)

        nom (orden 5);;
        ☞ "omega + cinq"
```

Comme pour Peano, on peut définir une addition et même une fonctionnelle d'itération, mais on ne pourra pas en faire grand chose.

```
let rec add x y = match y with Zero -> x | Suc z -> Suc (add x z)
                        | Lim r -> Lim (fun z -> (add x) (r z));;
```

```
let orditer e s q = f where rec f = fun Zero -> e | (Suc x) -> s(f x) | (Lim r) -> q (comp f r)
and comp f g x = f(g x);;
    ☞ orditer : 'a -> ('a -> 'a) -> ((int -> 'a) -> 'a) -> ord -> 'a = <fun>
```

ou bien :

```
let rec orditer  f e = fun Zero -> e | (Suc x) -> f (orditer f e x) | (Lim r) -> Lim (comp (orditer f e) r);;
    ☞ orditer : (ord -> ord) -> ord -> ord -> ord = <fun>
```

Remarque

Pour la culture générale, un cardinal est un ordinal non équipotent à un de ses élément, les ordinaux finis sont tous des cardinaux, par contre, $\omega + 1$, $\omega + \omega$, $\omega*\omega$ sont tous dénombrables, la collection des cardinaux infinis diffère beaucoup des ordinaux infinis, comme ils peuvent être indexés par les ordinaux, ils sont notés $\omega = \aleph_0$ pour le premier, $\aleph_{\alpha+1}$ pour le suivant de $\aleph_\alpha$, et si a est limite, $\aleph_\alpha = U_{\beta<\alpha} \aleph_\beta$.
L'axiome du choix est le fait que tout ensemble est équipotent à un ordinal (et donc possède un cardinal).
L'axiome généralisé du continu est que $\aleph_{\alpha+1} = card(P(\aleph_\alpha)) = 2^{\aleph_\alpha}$.
Cardinal inaccessible : $\pi < \omega$ et $\forall$ $a < \pi$ alors $2^a < \pi$ (On note généralement π le premier cardinal inaccessible).
Théorème d'Easton : Si F est une fonctionnelle croissante H telle que $H(\aleph_\alpha)$ ne puisse jamais être limite d'une famille de cardinaux inférieurs indexée par un cardinal inférieur, il existe un modèle de ZF cvérifiant $2^{\aleph_\alpha} = \aleph_\alpha$.

13. Structure de liste circulaire

On définit la structure de cycle ou liste circulaire (doublement chaînée) qui servira, entre autre, à modèliser les polygones. Un cycle est définit par une "cellule", le "même cycle" peut être abordé par la cellule voisine, c'est le rôle des fonctions avance et recule.

```
type 'a cycle = Vide | Cel of 'a cellule
    and 'a cellule = {val : 'a; mutable avant : 'a cycle; mutable apres : 'a cycle};;

let valeur = fun Vide -> failwith "cycle vide" | (Cel c) -> c.val;;

let avance = fun Vide -> Vide | (Cel c) -> c.avant;;

let recule = fun Vide -> Vide | (Cel c) -> c.apres;;

let changeavant nouv = fun Vide -> failwith "cycle vide" | (Cel c) -> (c.avant <- nouv);;

let changeapres nouv = fun Vide -> failwith "cycle vide" | (Cel c) -> (c.apres <- nouv);;
```

Fonctions de suppression de l'élément courant et d'insertion d'un objet x dans un cycle

```
let seul = fun Vide -> false | c -> (c= avance c);;

let supprime = fun Vide -> failwith "cycle vide " | c -> if seul c then Vide
    else  (changeavant (avance c) (recule c); changeapres (recule c) (avance c); avance c);;

let insavant x cycle = let c = Cel {val = x; avant = avance cycle; apres = cycle} in
    if cycle = Vide then (changeavant c c; changeapres c c; c)
    else (changeavant c cycle; changeapres c (avance cycle); c) ;;

let insapres x cycle = let c = Cel {val = x; avant = cycle; apres = recule cycle} in
    if cycle = Vide then (changeavant c c; changeapres c c; c)
    else (changeavant c (recule cycle); changeapres c cycle; c) ;;
```

Fonctions de conversion d'un cycle en liste et réciproque

```
let taille = fun Vide -> 0 | cycle -> let rec compte n c = if c = cycle then n
          else compte (n + 1) (avance c) in compte 1 (avance cycle);;

let cycletolist = fun Vide -> [] | c -> let rec aux cycle res = if cycle = c then (valeur c)::res
                                          else aux (recule cycle) ((valeur cycle)::res) in aux (recule c) [];;

let rec listocycle = fun [] -> Vide | (a :: q) -> insapres a  (listocycle q);;
```

```
taille (listocycle [1;2;3;4;5]);; ☞ 5
cycletolist (listocycle [1;2;3;4;5]);; ☞ [1; 2; 3; 4; 5]
```

il faut repasser par des listes si on veut afficher quoique ce soit car dans sa démarche d'affichage, caml ne peut que tourner en rond par exemple avec :

```
listocycle [1;2;3];;
☞ int cycle = Cel {val = 1; avant = Cel {val = 3; avant = Cel {val = 2; avant = Cel
        {val = 1; avant = Cel {val = 3; avant = Cel {val = 2; avant = Cel {val = 1; avant = Cel
                {val = 3; avant = Cel {val = 2; avant = Cel {val = 1; avant = Cel {.; .; .}; apres
= ...}; ...
```

```
let rec creer n = if n = 0 then Vide else insapres n (creer (n - 1));;
```

```
cycletolist (creer 7);; ☞ [7; 6; 5; 4; 3; 2; 1]
cycletolist(listocycle (cycletolist (creer 7)));; ☞ [7; 6; 5; 4; 3; 2; 1]
cycletolist(avance (creer 7));; ☞ [6; 5; 4; 3; 2; 1; 7]
cycletolist(recule (creer 7));; ☞ [1; 7; 6; 5; 4; 3; 2]
cycletolist (supprime(creer 7));; ☞ [6; 5; 4; 3; 2; 1]
```