Cours d'informatique de MPSI (2011-2012)

Stéphane Flon

Table des matières

Avant de commencer

Introduction à l'option informatique

1. Objectifs de formation

L'informatique, telle qu'enseignée en option en MPSI, est envisagée comme une science, et non comme un amas de techniques ou de compétences : comme le dit le programme officiel, « l'informatique est une science opérant sur des représentations rigoureuses de concepts bien définis ».

Cela signifie notamment que l'enseignement de cette option ne consistera pas seulement à vous faire programmer : une grande part du travail s'effectuera loin de tout ordinateur. Bien entendu, et heureusement, l'enseignement ne sera pas purement théorique, et vous pourrez souvent confronter votre apprentissage à un principe de réalité, en l'implémentant en salle info.

Pour cela, nous utiliserons le langage Caml (light), tout à fait approprié à l'esprit de l'option info (beaucoup plus que Maple par exemple ...).

Nous aurons souvent recours aux mathématiques, afin de poser les bases d'une programmation rigoureuse. Elles nous permettront notamment :

(1) de *prouver* un programme (comme on prouve un théorème mathématique).

(2) d'évaluer l'efficacité d'un programme, en particulier le temps qu'il met à s'exécuter (étude de la *complexité* d'un programme).

(3) de confronter des parti-pris de programmation, de choisir, parmi eux, le plus adapté à la situation rencontrée (récursif *vs* impératif, étude des *structures de données*).

Il s'agira donc, grâce à cette démarche rigoureuse, de réfléchir –en décortiquant ce que l'on veut faire, ce dont on dispose, et comment on va s'y prendre–, avant de se lancer dans la programmation proprement dite.

Dans le cadre de cette option, ce sont les mathématiques qui sont au service de l'informatique, et non l'inverse.

L'idée est, entre autres, d'éviter la technique de programmation suivante, si souvent rencontrée : « écrire un programme au hasard qui, avec un peu de chance, va donner le bon résultat, puis on se rend compte qu'en fait non, puis on passe des heures à tenter de le corriger ».

En cela, cette option est très utile à qui veut bien programmer plus tard, même si elle va sans doute trop loin pour un ingénieur dont la programmation serait une activité secondaire. Cependant, cette discipline, par la rigueur et la logique qu'elle requiert, reste toujours bénéfique, au même titre que les mathématiques : en tant qu'ingénieur, vous ne vérifierez pas la continuité d'une fonction avant d'appliquer le théorème des valeurs intermédiaires, mais votre formation vous permettra de prendre conscience d'un cas retors, voire de le résoudre (cf. le phénomène de Gibbs).

2. Choisir concrètement l'option informatique

2.1. Conséquences sur l'orientation

En choisissant l'option informatique, vous fermez la voie de la PSI : vous ne pourrez aller qu'en MP (ou MP*). Bien entendu, cela ne vous limitera pas plus tard à une carrière dans l'informatique, toutes les écoles restent accessibles avec cette option, et, au sein de ces écoles, toutes les voies sont possibles.

L'option informatique permet de passer tous les concours MP, indifféremment de l'option SI, à l'exception notable de Polytechnique, où les listes d'admis sont séparées selon l'option.

2.2. Organisation du travail

Vous aurez informatique le vendredi matin de 8h à 10h : une heure de cours, puis une heure de TD. L'heure de TD se fera parfois devant l'ordinateur mais pas systématiquement, loin de là.

Certaines de ces séances seront consacrées aux devoirs surveillés (deux par trimestre).

Vous aurez également quelques colles Caml au long de l'année (horaires à définir).

3. Bref aperçu du programme de SUP

- *Initiation à Caml*
  On présente le langage Caml, ses qualités, sa philosophie, sa syntaxe.
- *Méthodes de programmation*
  Programmation impérative (boucles for, while, conditionnelle if, etc.). Comment prouver un programme impératif.
  Programmation récursive : une nouvelle philosophie de programmation (la fonction « s'appelle elle-même » dans sa définition). Compléments mathématiques (sur les relations d'ordre), ensembles inductifs. Comment prouver un programmer récursif.
- *Analyse des algorithmes*
  Complexités spatiale et temporelle (performance théorique des algorithmes conçus).
  Méthode diviser pour régner.
  Algorithmes classiques de tri (comment ranger des valeurs selon un ordre donné).
- *Structures de données et algorithmes*
  Notion de structure de données. Listes, piles. Comment choisir la structure de données adaptée à un problème, quelles conséquences cela a-t-il sur la programmation ?
- *Logique*
  Calcul propositionnel. Distinction entre syntaxe et sémantique. Circuits logiques.
- *Et en Spé ?*
  Bien sûr, vous reprendrez en partie ce qui a été fait en Sup, mais vous étudierez également de nouveaux sujets, parfois très abstraits, comme les automates finis.

4. FAQ

4.1. Quelles sont les bonnes raisons de suivre l'option informatique ?

On peut suivre l'option informatique
- par goût de l'informatique, par exemple parce qu'on
  - veut devenir informaticien, ou intégrer une école d'informatique (désolé pour l'évidence ...)
  - l'a déjà pratiqué (C/C++, php, etc.)
  - a apprécié la partie programmation de Maple.
  - n'a pas apprécié la partie programmation de Maple, parce qu'on l'a trouvée trop peu rigoureuse.
  - voudrait commencer la programmation sur de bonnes bases.
- par goût des (et facilités en) mathématiques, surtout algébriques : relations d'ordre, structures algébriques, récurrence, dénombrement.
- par goût de l'abstraction, de la rigueur.

4.2. Quelles sont les bonnes raisons de ne pas suivre l'option informatique ?

On a bien raison de ne pas suivre l'option informatique si :
- on surkiffe la SI.
- on veut aller en PSI (ou s'en garder la possibilité).
- veut intégrer une école spécialisée dans un domaine précis, relevant de la SI (cela dit, SI comme informatique sont certainements utiles, au moins comme bagage culturel, à tout ingénieur qui se respecte).
- on a toutes les difficultés du monde à programmer une boucle for en Maple, ou si on est exaspéré car Maple nous en veut de ne pas avoir mis « ; ».

Pour les écoles généralistes, les deux options se valent.

4.3. Quelles sont les mauvaises raisons de suivre, ou de ne pas suivre, l'option informatique

Il ne faut pas se déterminer à cause :
- des horaires.
- de la « stratégie » : choisissez votre option selon vos goûts, et non parce qu'untel est une méga-torche et prend telle option ...
- de l'élitisme : l'option info est ouverte à tous ceux qui sont motivés, même s'ils ne sont pas géniaux en maths (mais ils s'engagent en connaissance de cause).

Première partie

Programmation en Caml

CHAPITRE I

Introduction

1. Le langage Caml

Bien que l'enseignement de l'option informatique ne se résume pas à la programmation, il nous faut disposer d'un langage dans lequel travailler, afin de mettre en pratique la théorie et de se confronter à un certain principe de réalité. Cette année, nous utiliserons le langage Caml, et plus précisément Caml light, que vous pourrez trouver facilement (et gratuitement) sur le net.

Sans rentrer dans une analyse poussée de ce langage, on peut lui reconnaître de nombreuses qualités pédagogiques, que nous passons brièvement en revue, et sur lesquelles nous reviendrons plus en détail au chapitre suivant :

1.1. Rigueur

En Caml, tous les objets (y compris les fonctions) ont un *type*, c'est-à-dire que chacun appartient à une catégorie bien précise. On dit que ce langage est *fortement typé*.

Par exemple, comme en mathématiques, toute fonction a une source (un ensemble de départ) et un but (un ensemble d'arrivée) déterminés.

Après chaque définition, Caml répond sous la forme

```
nom : type = valeur
```

Après chaque expression, Caml répond sous la forme

```
- : type = valeur
```

(la réponse étant parfois précédée d'effets de bord, voir plus loin).

Dans l'exemple

```
# 2 + 2;;

- : int = 4
```

Caml nous informe que le résultat 4 de l'expression 2 + 2 est de type entier (int).

Ces contraintes de type confèrent à Caml une certaine rigidité : Caml n'aime pas mélanger les choux et les carottes [1]. Par exemple, Caml n'accepte pas que nous additionnions un entier (type int) avec un nombre à virgule flottante (type float) :

```
# 2 + 2.4;;

Toplevel input:
>2 + 2.4;;
>     ^^^
This expression has type float,
but is used with type int.
```

Cette rigidité doit toutefois être mise au crédit de Caml, car elle permet d'éviter ou détecter de nombreuses erreurs de programmation.

1.2. Simplicité

Les bases de ce langage – dont nous nous contenterons – sont assez réduites, et la plupart des programmes que nous écrirons ne prendront que quelques lignes. Pourtant, nous pourrons répondre à de nombreux problèmes relativement compliqués.

1. sauf si on choisit de les mélanger en définissant un type plus général

Nous avons vu que chaque objet avait un type : faut-il pour autant déclarer le type de chaque objet en Caml ? Non, car le typage est automatique : Caml devine, autant que possible, de quel type d'objet vous parlez.

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

f : int -> int = <fun>
```

J'ai ici défini la fonction f, dont Caml a automatiquement déterminé le type, à savoir une fonction qui à un entier associe un entier : il a deviné que l'argument x de f était entier, puisque je lui ai ajouté 2, et sait que le résultat obtenu est de type entier.

1.3. Fonctionnel

Nous venons de voir un exemple de fonction en Caml. En fait, un programme Caml n'est pour ainsi dire qu'une succession de fonctions, que nous nous contentons d'appliquer en fin de programme. Dans un programme Caml, il s'agit bien plus de décrire les fonctions que nous souhaitons employer (programmation fonctionnelle) que de donner des instructions de gestion des données ou de la mémoire de l'ordinateur (programmation impérative). Il impose donc une vision assez mathématique de la programmation.

Il faut bien comprendre que les fonctions en Caml sont des objets comme les autres, ce qui fait qu'on peut les prendre en argument, comme en mathématiques : ainsi,

```
# let translate_de_2_depart_arrivee f x = f (x + 2) + 2;;

translate_de_2_depart_arrivee : (int -> int) -> int -> int = <fun>
```

est la fonction qui, à une fonction f de type int -> int, associe la fonction de même type, qui à x (entier) associe $f(x+2)+2$. C'est donc, avec les notations usuelles, un élément de $(\mathbb{N}^{\mathbb{N}})^{(\mathbb{N}^{\mathbb{N}})}$.

1.4. Polyvalent

Caml n'est pas un langage purement fonctionnel, dans le sens où il a également des traits impératifs, comme les boucles for et while par exemple, que vous connaissez sans doute déjà :

```
# for i = 1 to 10 do print_int i done;;

12345678910- : unit = ()
```

```
# let i = ref 2;;
i : int ref = ref 2

# while !i < 1000 do i := !i * !i done;;
- : unit = ()

# i;;
- : int ref = ref 65536
```

Il admet aussi des définitions par filtrage très puissantes et élégantes :

```
# let est_voyelle_minuscule = function
        | 'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true
        | _ -> false;;
est_voyelle_minuscule : char -> bool = <fun>

# est_voyelle_minuscule 'e';;
- : bool = true

# est_voyelle_minuscule 's';;
- : bool = false

# est_voyelle_minuscule 'Y';;
- : bool = false
```

1.5. Sûr

Comme on l'a déjà observé, le typage automatique en Caml permet d'éviter de nombreuses erreurs. De plus, Caml n'hésite pas à produire des avertissements en cas de programmation tendancieuse. Par exemple, il prévient lorsqu'un cas de filtrage est oublié :

```
let f (x, y)= match (x, y) with
        | (0, y) -> y
        | (x, 0) -> x;;

Toplevel input:
.............match (x, y) with
        | (0, y) -> y
        | (x, 0) -> x..
Warning: this matching is not exhaustive.
f : int * int -> int = <fun>
```

Sans rentrer dans les détails, cette fonction associe à un couple d'entiers un entier. Cependant, sa définition ne donne la valeur que lorsque l'une des coordonnées est nulle.

2. Quelques conseils

Donnons ici quelques recommandations de bon sens, afin de faciliter la programmation, et la compréhension de vos programmes par quelqu'un d'autre (moi, par exemple).

2.1. Réfléchir avant de programmer

Il est recommandé, avant de se lancer dans l'écriture d'un programme, de bien réfléchir à la structure que l'on veut lui donner, éventuellement par écrit, au brouillon. Plutôt que d'écrire un premier jet incorrect, puis de lui appliquer des rustines jusqu'à ce qu'il fonctionne (ou semble fonctionner), au risque de le rendre incompréhensible, essayez de savoir où vous allez.

Pour bien structurer votre programme, répondant à un problème précis, vous pouvez essayer de décomposer ce dernier en sous-problèmes, se décomposant éventuellement à leur tour en (sous-)sous-problèmes, jusqu'à tomber sur des briques élémentaires faciles à programmer et à assembler : il s'agit donc de programmer par raffinements successifs, ou encore par *analyse descendante*.

Un autre avantage d'avoir écrit des sous-programmes élémentaires consiste en leur possible réutilisation et partage.

2.2. Pratiquer

Un langage de programmation s'apprend en le pratiquant : il ne faut pas vous contenter de comprendre les programmes trouvés ici ou là, il vous faudra tester votre bonne assimilation en programmant, en les testant, en devinant ce qu'un programme va produire, etc.

Comme en mathématiques, il faudra travailler activement en TD, et non se contenter de lire un corrigé.

2.3. Rendre votre programme lisible

N'oubliez pas que le programme sera sans doute lu par quelqu'un d'autre, et que sa compréhension n'est pas forcément immédiate : il vous faut donc faire des efforts de lisibilité. D'ailleurs, même si votre programme est à usage personnel, il vaut mieux le clarifier, car vous pouvez avoir du mal à le comprendre une fois que vous l'avez laissé de côté pendant quelques temps.

N'hésitez pas à insérer des commentaires dans vos programmes (utiliser *(* ... *)* en Caml), expliquant votre démarche, pourquoi votre algorithme fonctionne, les notations, etc.

On simplifie grandement la compréhension d'un programme en donnant des noms parlants aux objets introduits : de même qu'en mathématiques, il est possible mais malvenu de noter un certain nombre complexe x et z sa partie réelle (ou un entier naturel ε), il vaut mieux éviter en informatique de noter « addition » ou même « k » la loi de multiplication des entiers.

Pensez également à aérer et hiérarchiser le texte, notamment en utilisant les indentations.

Vous pouvez aussi rendre votre programmation plus abstraite, ce qui rend parfois le texte plus lisible. Attention cependant, dans certains cas, l'abstraction ne simplifie pas vraiment la compréhension.

2.4. Laisser l'ordinateur travailler

La récursivité et l'aspect fonctionnel de Caml permettent d'écrire des programmes qui ne semblent être que des reformulations du problème posé. Ces programmes sont le plus souvent clairs, efficaces, élégants, et simples à écrire. Il est parfois possible, au prix d'un effort mathématique, d'enlaidir et compliquer lesdits programmes, mais on n'a pas grand chose à y gagner. Les intoxiqués de la programmation impérative doivent apprendre à laisser l'ordinateur faire le job !

CHAPITRE II

Programmation en Caml

Nous ne présentons ici qu'une petite partie des fonctionnalités de Caml light. Vous pouvez consulter l'aide en ligne pour approfondir certains aspects de ce langage.

1. Brève présentation de Caml light

L'implémentation Caml light que nous utiliserons se présente comme une boucle interactive :
- Caml invite l'utilisateur à écrire une phrase par le « prompt » # ;
- l'utilisateur tape une phrase, c'est-à-dire, le plus souvent, une *expression* ou une *déclaration*, qu'il termine par « ; ; », et valide en appuyant sur entrée ;
- Caml analyse alors la syntaxe de cette phrase ;
- lorsque cette syntaxe est correcte, il détermine le type de l'expression ;
- si cette expression est bien typée, il l'évalue, puis renvoie une réponse sous la forme − : **type** = valeur.

Remarque : à chacune des étapes évoquées, Caml peut renvoyer un message d'erreur.
Remarque : cette manière de programmer, au sein de cette boucle interactive, est conviviale. Elle permet d'utiliser Caml comme une calculatrice, ou un peu comme Maple (en moins souple). Elle a cependant des limites évidentes, dès que le programme écrit est élaboré. On pourra donc, le cas échéant, écrire un fichier en .ml, que l'on ouvrira dans l'implémentation Caml light.

2. Notion d'effet de bord

2.1. Définition

En informatique, on appelle *effet (de bord)* d'une fonction une modification de l'environnement : il peut s'agir par exemple de la modification d'une variable ou de l'argument de la fonction appliquée, de l'écriture sur un fichier, d'un affichage (« print »). Une fonction qui n'agit que par effet de bord (qui ne renvoie aucune valeur) est appelée *procédure*.

Caml dispose de plusieurs fonctions prédéfinies, qui n'agissent que par effet de bord, comme la fonction print_int :

```
# print_int 3;;

3- : unit = ()
```

qui a pour effet d'afficher l'entier pris en argument.

2.2. Nécessité d'un type unit en Caml

En Caml, toutes les fonctions ont un argument, et renvoient un résultat : il doit en être ainsi des procédures, qui ne renvoient pourtant pas de résultat. Pour lever cette difficulté, on a défini le type unit, dont l'unique élément est (), et que l'on peut comprendre comme « rien ». En Caml, une procédure n'est rien d'autre qu'une fonction dont le résultat est (). C'est le cas par exemple de print_int, utilisée ci-dessus.
Remarque : il faudra bien distinguer par exemple les deux fonctions suivantes :

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

# let affiche_successeur n = print_int (n + 1);;
affiche_successeur : int -> unit = <fun>
```

En effet, affiche_successeur est une procédure, elle renvoie (). En particulier, on ne peut pas la composer avec elle-même, contrairement à successeur :

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

# successeur (successeur 3);;
- : int = 5
```

```
# affiche_successeur 3;;
4- : unit = ()

# affiche_successeur (affiche_successeur 3);;
Toplevel input:
>affiche_successeur (affiche_successeur 3);;
>                    ^^^^^^^^^^^^^^^^^^^^
This expression has type unit,
but is used with type int.
```

Remarque : de même qu'une fonction peut renvoyer (), une fonction peut prendre ce terme en argument : c'est par exemple le cas de la fonction suivante

```
# let Hello_world () = print_string "Hello world";;
Hello_world : unit -> unit = <fun>

# Hello_world ();;
Hello world- : unit = ()

# Hello_world (Hello_world ());;
  (* on peut composer cette procédure par elle-même,
  puisque son argument est de type unit *)
Hello worldHello world- : unit = ()
```

3. Identificateurs

Il est souvent utile de donner un nom au résultat d'un calcul effectué par Caml. Comme en mathématiques, on peut avoir besoin de « variables » *globales* et *locales*.

3.1. Identificateurs globaux

On parle aussi, en se conformant à l'usage, de variables globales, même si nous verrons que cette expression peut induire en erreur. Les identificateurs globaux sont conservés en mémoire, et donc utilisables dans l'intégralité du programme. Ils sont à réserver aux objets dont nous avons besoin tout au long de notre travail.

Pour introduire (ou définir, ou déclarer) un identificateur global, la syntaxe est

```
let identificateur = valeur
```

Une telle définition est une *liaison* d'un identificateur (autrement dit d'un nom, d'une abréviation) à une valeur (qui devient ainsi identifiée), que l'on peut utiliser dans la suite de notre programme. C'est l'équivalent du « soit » en mathématiques, tel qu'employé dans l'expression « Soit M la borne supérieure de f sur $[0, 1]$. »

```
# let y = 7;; (* 7 a pour nom y *)
y : int = 7

# let successeur n = n + 1;;
successeur : int -> int = <fun>

# successeur y;;(* Caml sait qui est désigné par le symbole y *)
- : int = 8
```

```
# let y = 7;;
y : int = 7
```

```
# y + 2;;
- : int = 9
```

Remarque : voici un point relativement compliqué. Il ne faut pas confondre cette *définition* de l'*affectation*. **Une fois défini, un nom conserve la valeur donnée, à moins de le redéfinir**, c'est pourquoi le terme de « variable » est plutôt à éviter dans ce cadre, et il vaut mieux parler d'identificateur.

```
# let y = 7;; (* 7 a pour nom y *)
y : int = 7

# let x = y + 2;; (* 7 + 2 a pour nom x *)
x : int = 9

# let y = 2;; (* y n'identifie plus 7 mais 2 *)
y : int = 2

# x;; (* x est toujours le nom de 9 *)
- : int = 9
```

Pour une raison obscure, nous avons redéfini y. Nous pouvons même le redéfinir à partir de sa définition actuelle :

```
# let y = 7;; (* 7 a pour nom y *)
y : int = 7

# let y = y + 2;; (* 7+2 a pour nom y *)
y : int = 9
```

Cependant, cette programmation horrible est à proscrire, tant elle contrevient à l'esprit Caml. D'une manière générale, il ne semble pas opportun de redéfinir un identificateur [1].

3.2. Identificateurs locaux

Ce sont les identificateurs auxiliaires, permettant de réaliser et de donner un sens à un calcul secondaire : ce sont des définitions « jetables » (et « jetées »). Ainsi en est-il des indices muets que nous utilisons dans les sommes : quand on écrit $\sum_{k=0}^{n} k^3$ en mathématiques, n est un entier naturel qu'il faut avoir introduit dans le corps du texte, tandis que k est une variable muette ne servant qu'à proprement définir cette somme. Il faut savoir de quel (type de) n on parle, mais, inversement, il faut que k n'ait pas de valeur affectée (la phrase « Soit $k = 3$ et $n = 5$. Calculons $\sum_{k=0}^{n} k^3$. » est rejetée en mathématiques).

La syntaxe est la même que pour les identificateurs globaux, mais on lie la déclaration de l'identificateur local à l'expression dans laquelle il est utilisé par **in** :

```
let identificateur_local = valeur in expression
```

```
# let y = 7 in y + 2;;
- : int = 9

# y;;
Toplevel input:
>y;;
>^
The value identifier y is unbound.
```

On peut aussi observer que cette phrase est une expression et non une déclaration.

On peut par ailleurs utiliser un identificateur local pour définir un identificateur global :

```
# let x = (* x est global *)
        let y = 7 in (* y est local, et sert à définir x *)
        y + 2;;
x : int = 9

# x;;
```

1. mais on peut le faire par mégarde si le programme est long

```
- : int = 9

# y;;
Toplevel input:
>y;;
>^
The value identifier y is unbound.
```

En reprenant l'analogie avec l'exemple de la somme en mathématiques, que se passe-t-il si nous avons la mauvaise idée d'utiliser un identificateur global pour identificateur local ?

```
# let y = 3;;
y : int = 3

# let x =
        let y = 7 in
        y + 2;;
x : int = 9

# x;;
- : int = 9

# y;;
- : int = 3
```

L'identificateur local l'a emporté au sein de l'expression à laquelle il est associé, ce qui est normal (à quoi servirait-il sinon ?). Bien sûr, il vaut mieux éviter cette réutilisation pour le moins maladroite.

Pour mieux insister sur le caractère secondaire d'un identificateur local, on peut inverser les ordres de la définition et de l'expression, en utilisant **where** au lieu de **in**, c'est-à-dire en écrivant

```
expression where identificateur_local = valeur
```

On peut ainsi écrire

```
# let x = y + 2 where y = 7;;
x : int = 9

# x;;
- : int = 9
```

Remarque : dans la mesure du possible, on privilégiera les identificateurs locaux. Cela limitera les erreurs.

Afin d'éviter de trop nombreuses définitions emboîtées, on pourra utiliser **and** pour introduire plusieurs identificateurs (qu'ils soient locaux ou globaux, et que ce soit avec **in** ou **where**) :

```
# z * z - x * x - y * y where z = 13 and x = 5 and y = 12;;
- : int = 0
```

4. Types élémentaires

On recense ici les types élémentaires en Caml, *i.e.* ceux qui ne sont pas construits à partir d'autres types : par exemple, le type int (celui des entiers) est élémentaire, pas le type int -> int (celui des fonctions qui à un entier associent un entier). Nous avons déjà parlé du type unit. Il ne s'agit pas ici de décrire de manière exhaustive ces types et les fonctions associées, mais de présenter les plus utiles d'entre eux.

4.1. Le type booléen

Le type booléen ne prend que deux valeurs : true et false. Un *prédicat* est une fonction à valeurs booléennes (c'est surtout à cela que servent les booléens). On dispose du connecteur logique « et », s'écrivant & ou &&, du connecteur logique « ou », s'écrivant **or** ou ||, et de l'opérateur unaire de négation **not**.

Remarque : ces opérateurs binaires « et » et « ou » sont dits *infixes* car ils se placent entre les arguments.

Remarque : faites attention, **and** ne désigne pas le connecteur logique « et ».

Remarque : il faut savoir que Caml effectue une évalutation dite *paresseuse*, ce qui signifie que dès qu'il dispose d'informations suffisantes pour déterminer la valeur du booléen considéré, il arrête le parcours. Concrètement, Caml répond true à 0 = 0 || (1 / 0 = 0);;, alors que (1 / 0 = 0) produirait une erreur.

De même, on peut définir la fonction factorielle ainsi (peu importe si vous ne comprenez pas encore cette définition)

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

ce qui permet d'effectuer quelques calculs :

```
# fact 10;;
- : int = 3628800
```

mais pas tous :

```
# fact 5345346;;
Uncaught exception: Out_of_memory
```

Pourtant, Caml évalue sans problème, grâce à sa « paresse », le booléen suivant :

```
# 1 = 0 && fact 5345346 = 123;;
- : bool = false
```

alors qu'il n'évaluerait pas fact 5345346 = 123 && 1=0.

Les connecteurs logiques ne sont donc pas vraiment commutatifs : on pensera à mettre en premier lieu ce qui se calcule vite.

4.2. Le type int

C'est, comme nous l'avons dit, le type des entiers. Il faut cependant prendre garde, car Caml gère seulement les entiers compris entre -2^{30} et $2^{30}-1$, et se ramène à un tel entier par congruence modulo 2^{31} (si vous préférez, il travaille dans $\mathbb{Z}/2^{31}\mathbb{Z}$). On obtient donc des résultats étonnants si on ne fait pas attention :

```
# 10000 * 10000 * 10000;;
- : int = -727379968
```

On dispose des opérateurs binaires (à deux arguments) infixes + − ∗ / mod, dont les définitions sont claires. Précisons simplement que a / b et a mod b désignent respectivement le quotient et le reste de la division euclidienne de a par b lorsque a et b sont des entiers naturels (b non nul), mais que (-7) mod 5 par exemple vaut -2.

Remarque : on peut, et c'est un procédé général, donner une version *préfixe* de ces fonctions en utilisant la primitive **prefix** :

```
# let plus = prefix +;;
plus : int -> int -> int = <fun>

# plus 3 5;;
- : int = 8
```

On a également l'opérateur unaire (à un argument) −, de passage à l'opposé.

Caml intègre les opérateurs binaires préfixes min et max (min a b et max a b donnent respectivement le minimum et le maximum de a et b).

On dispose en outre d'opérateurs de comparaison = < > <= >= <> entre entiers, qui sont des prédicats (ils renvoient des booléens).

Remarque : nous verrons que ces opérateurs sont en fait *polymorphes*, et permettent donc d'autres comparaisons que celles entre entiers.

4.3. Le type float

Le type float est celui des nombres à virgule flottante (en bref : des flottants), qui sont des approximations des nombres réels.

Remarque : la virgule dont il est question s'écrit avec un point (3.5, 4. sont acceptés, mais pas .6) :

```
# 2.6;;
- : float = 2.6
```

Les opérateurs sur ces nombres sont pour l'essentiel ceux définis pour les entiers int, suivis d'un point, *i.e.* +. −. ∗. /.. On dispose aussi de l'exponentiation ∗∗ (par exemple, 2. ∗∗ 10.;; renvoie − : float = 1024.0).
Remarque : pour de grands nombres, Caml passe en notation scientifique mantisse-exposant. Il répond par exemple à 3. ∗∗ 45.;; par − : float = 2.95431270655e+21.

On peut aussi utiliser les opérateurs de comparaison =. <. <=. >=. <>.
Remarque : en réalité, il n'est pas nécessaire de mettre de point après ces symboles lorsqu'on traite de float. Nous développerons cette remarque en 6.5.

Enfin, les fonctions usuelles sqrt exp log sin cos tan acos asin atan, dont il est inutile de donner la définition, sont des primitives Caml.
Remarque : retenez bien que ∗∗ s'emploie pour les nombres à virgule flottante, et non les entiers.

4.4. Le type char

C'est le type des caractères. On les écrit entre accents graves (pour ne pas les confondre avec un identificateur).

4.5. Conversion de types élémentaires

Caml dispose de nombreuses fonctions de conversion de type, que l'on considère comme inélégantes. Leurs noms sont explicites : int_of_float , float_of_int , string_of_float , float_of_string , string_of_int , int_of_string , etc. Précisons que type1_of_type2 convertit un objet de type type2 en un objet de type type1.

4.6. Affichage de types élémentaires

On utilise les procédures suivantes, aux noms transparents : print_int , print_float , print_char, print_string . À ce propos, on peut produire un retour à la ligne (comme effet de bord) grâce à print_newline (), print_char '\n' ou print_string "\n" au choix.

5. Deux exemples de types non élémentaires

5.1. Les n-uplets

Un couple de première composante a et de seconde composante b s'écrit (a, b). Son type est type_de_a ∗ type_de_b.
Remarque : les objets a et b n'ont pas nécessairement le même type.
Remarque : on accède à la première (resp. seconde) composante par les fonctions fst et snd, mais il est facile de redéfinir ces fonctions.
Remarque : on peut dans certaines circonstances se passer des parenthèses.

De manière analogue, on définit plus généralement un n-uplet (où n est un entier supérieur ou égal à deux) avec la syntaxe évidente (a_1, ..., a_n).

```
# snd (3,(5,7));;
- : int * int = 5, 7
```

5.2. Les fonctions

Nous allons voir à la section suivante comment définir des fonctions. Mentionnons simplement ici que le type type1 −> type2 est celui des fonctions dont l'argument est de type type1, dont les valeurs sont de type type2.

6. Programmation fonctionnelle

Un programme Caml est souvent une suite de fonctions, que l'on applique à la fin. Il est donc très important de savoir définir et utiliser les fonctions en Caml. La plupart des fonctions que nous définirons seront récursives, mais on s'intéresse seulement dans cette section aux fonctions non récursives (on écrira simplement **let rec** f au lieu de **let** f pour une fonction récursive).

6.1. Les fonctions : des objets comme les autres

Les fonctions sont des objets Caml au même titre que les nombres à virgule flottante ou les listes d'entiers par exemple. Nous en déduisons les remarques suivantes :
Remarque : nous pouvons définir des fonctions prenant comme argument une fonction, ou ayant pour valeurs des fonctions. Une telle fonction est appelée *fonctionnelle*.

```
# let successeur x = x + 1;; (* définition d'une fonction simple *)
successeur : int -> int = <fun>

# let ajoute_deux_a_l_argument f = function x -> f (x + 2);;
(* définition d'une fonctionnelle *)
ajoute_deux_a_l_argument : (int -> 'a) -> int -> 'a = <fun>

# let ajoute_trois = ajoute_deux_a_l_argument successeur;;
(* application de la fonctionnelle *)
ajoute_trois : int -> int = <fun>

# ajoute_trois 5;;
- : int = 8
```

Remarque : il n'est pas nécessaire qu'une fonction ait un identificateur (vous n'imposez pas que l'entier 3 ait un identificateur pour travailler avec) :

```
# (function x -> x + 1)(3);; (* application d'une fonction sans la déclarer *)
- : int = 4
```

6.2. Fonctions d'une variable

Supposons que nous voulions définir une fonction f à un argument x. Caml accepte les syntaxes suivantes :

```
let f = function x -> valeur_de_f_en_x

let f = fun x -> valeur_de_f_en_x

let f x = valeur_de_f_en_x
```

où valeur_de_f_en_x est la formule donnant $f(x)$.
Pour définir par exemple la fonction successeur, de type int -> int, nous pouvons donc écrire

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

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

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

Pour appliquer la fonction f en x, on tape simplement f(x), ou même f x dans certains cas (lorsqu'il n'y a pas d'ambiguïté) : par exemple, avec la fonction précédente, successeur (5);; , successeur 8;; et successeur (−3);; renvoient respectivement, 6, 9 et −2, mais Caml répond à successeur −2;; par le message d'erreur

```
Toplevel input:
>successeur -2;;
>^^^^^^^^^^
This expression has type int -> int,
but is used with type int.
```

Caml a constaté un problème en essayant de retrancher 2 à successeur, qui n'est pas un entier.

6.3. Fonctions de plusieurs variables

Nous avons donné une syntaxe pour une fonction d'un argument. Notons que cet argument peut très bien être un couple ou un triplet par exemple. Une loi de composition interne prend un unique argument, qui est un couple. Pour l'addition, on obtient ainsi :

```
# let addition (x, y) = x + y;; (* (x,y) est un couple d'entiers *)
addition : int * int -> int = <fun>

# addition (3, 5);;
- : int = 8
```

Cependant, les informaticiens préfèrent souvent voir l'addition comme une application qui à un entier x associe l'application qui à un entier y associe x+y. On obtient ainsi une fonction à deux arguments x et y. Caml autorise des syntaxes analogues à celles vues dans le cas d'une fonction d'une variable, *à l'exception de la première* (celle avec **function**) :

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

# addition 3 6;;
- : int = 9
```

Nous aurions tout aussi bien pu écrire **let** addition x y = x + y;;
En revanche, **let** addition = **function** x y -> x + y;; renvoie le message d'erreur

```
Toplevel input:
>let addition = function x y -> x + y;;
>                           ^^^
The constructor x is unbound.
```

Tout cela est bien joli, mais quel est l'intérêt de considérer l'addition ainsi ? Le principal avantage est que nous pouvons appliquer addition en un entier x fixé, obtenant ainsi l'application de translation (ou d'ajout) de x, *i.e.* la fonction $y \mapsto x + y$:

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

# let ajout_de_quatre = addition 4;;
ajout_de_quatre : int -> int = <fun>

# ajout_de_quatre 9;;
- : int = 13
```

Remarque : on tente[2] le plus souvent de se débarrasser du parenthésage. Pour ce faire, on peut retenir que

```
f x + g y équivaut à f(x)+g(y)

f x y équivaut à (f x) y
```

On dit que l'application associe à gauche.
(addition 2) 9 donne 11, mais addition (2 9) produit un message d'erreur. En effet, dans ce second cas, on demande à Caml d'appliquer 2 à 9, ce qui n'a pas grand sens :

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

# add (2 9);;
Toplevel input:
>add (2 9);;
>     ^
This expression is not a function, it cannot be applied.
```

2. on ne vous impose pas toutefois de chasser toutes les parenthèses inutiles

6.4. Curryfication

Caml permet de définir l'addition sous la forme **let** addition x y = x + y;;. Cette forme de l'addition est dite *curryfiée*. Formalisons un peu le procédé ayant permis de passer de l'addition usuelle à sa version curryfiée (ce procédé s'appelant *curryfication*).

Considérons trois ensembles A, B, C, et soit $\mathcal{F}(A \times B, C)$ l'ensemble des applications de $A \times B$ dans C (c'est aussi $C^{A \times B}$). Cet ensemble est en bijection canonique avec $\mathcal{F}(A, \mathcal{F}(B, C))$, (*i.e.* avec $(C^B)^A$), par :

$$\begin{array}{rcl} \nabla : \mathcal{F}(A \times B, C) & \to & \mathcal{F}(A, \mathcal{F}(B, C)) \\ f & \mapsto & \left(\begin{array}{rcl} \nabla(f) : A & \to & \mathcal{F}(B, C) \\ a & \mapsto & \left(\begin{array}{rcl} \nabla(f)(a) : B & \to & C \\ b & \mapsto & f(a, b) \end{array} \right) \end{array} \right) \end{array}$$

6.5. Le polymorphisme

En 4.2, nous remarquons que les opérateurs de comparaison sont polymorphes. Qu'entend-on vraiment par cela ? Vous vous rappelez que tout objet Caml a un type, et que toute fonction est un objet : toute fonction a donc un type. Cependant, le type de certaines fonctions (ou d'autres objets) n'est pas toujours figé. Si on veut écrire la version préfixe de < par exemple, Caml répond :

```
# let inferieur_strict a b = a < b;;
inferieur_strict : 'a -> 'a -> bool = <fun>
```

Il signale ainsi que l'important est que a et b aient le même type, qu'il note 'a (et qui est un *paramètre de type*), mais sans préciser ce type.

C'est pourquoi Caml répond sans broncher aux requêtes suivantes : 4 < 2;;, 5.3 < 2.1;;, " petit " < "grand ";; (et il répond false à chaque fois).

Remarque : le polymorphisme en Caml fonctionne par « tout ou rien » : si un objet n'a pas un type précis, il a potentiellement tous les types.

6.6. Filtrage de motifs

C'est l'un des traits les plus agréables de Caml. Le filtrage permet souvent des définitions très claires et simples de fonctions. Supposons que nous voulions définir une fonction f. Les syntaxes suivantes sont possibles

```
let f n = match n with
        | n_0 -> valeur_de_f_en_n_0
        | n_1 -> valeur_de_f_en_n_1
        ...
        | _ -> valeur_de_f_dans_les_autres_cas
```

```
let f = fun (* ou function pour une fonction à un unique argument *)
        | n_0 -> valeur_de_f_en_n_0
        | n_1 -> valeur_de_f_en_n_1
        ...
        | _ -> valeur_de_f_dans_les_autres_cas
```

Remarque : les valeurs filtrées (comme n_0) sont bien des valeurs, ou des motifs, mais ils ne peuvent pas être des identificateurs déjà définis (tout identificateur dans un filtre est un nouvel identificateur).

Remarque : la dernière ligne règle le sort de tous les cas non encore traités. Elle n'est pas toujours utile, mais permet de s'assurer que le filtrage est exhaustif (tous les cas possibles sont étudiés).

Remarque : le filtrage s'effectue dans l'ordre de lecture, et peut très bien être redondant (c'est le filtrage antérieur qui l'emporte alors).

Remarque : Caml prévient si le filtrage n'est pas exhaustif, et il prévient également si un cas de filtrage est inutile.

Remarque : si plusieurs éléments de la source ont la même valeur, on peut les regrouper dans le filtrage avec |. Voir l'exemple de est_voyelle_minuscule page 14.

```
# let f = fun
        | 0 y -> 1
        | _ _ -> 0;;
f : int -> 'a -> int = <fun>
```

```
# f 5 2;;
- : int = 0
```

On peut filtrer en changeant le type

```
# let f x y = match (x,y) with
          | (0, _) -> 1
          | (_, 0) -> 1
          | _ -> 0;;
f : int -> int -> int = <fun>

# f 5 2;;
- : int = 0
```

On peut filtrer selon une seule variable

```
# let ou_logique a b = match a with
          | true -> true
          | false -> b;;
ou_logique : bool -> bool -> bool = <fun>

# ou_logique false true;;
- : bool = true
```

7. Introduction à la récursivité

Il ne s'agit pas ici d'étudier la récursivité du point de vue théorique, mais d'en comprendre le principe sur des exemples simples.

Une fonction est dite *récursive* si elle s'appelle dans sa propre définition. Cela n'empêche pas la fonction d'être parfaitement définie. En fait, ce procédé est souvent utilisé en mathématiques, lorsqu'on définit une fonction (par exemple une suite récurrente), une notation (par exemple a^n où a est un élément d'un anneau, n un entier naturel) ou un symbole (par exemple celui de somme) par récurrence.

En Caml, pour déclarer une fonction récursive f, on utilise la syntaxe :

```
let rec f = définition_récursive_de_f
```

On peut ainsi définir la fonction factorielle de la manière suivante (à la fois récursive et par filtrage) :

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

# fact 5;;
- : int = 120
```

Remarque : si l'on omet le mot **rec**, Caml prévient que l'identificateur de la fonction n'est pas lié :

```
# let fact = function
        | 0 -> 1
        | n -> n * fact (n - 1);;
Toplevel input:
>       | n -> n * fact (n - 1);;
>                  ^^^^
The value identifier fact is unbound.
```

Remarque : si on met le mot **rec** pour déclarer une fonction non récursive, Caml ne dit rien de particulier :

```
let rec carre = function n -> n * n;;
carre : int -> int = <fun>

carre 5;;
- : int = 25
```

Cependant, il serait bien sûr malvenu d'employer cette construction **let rec** pour définir une fonction non récursive ...

8. Programmation impérative

La plupart des programmes que nous écrirons en Caml seront récursifs, et nous exploiterons surtout les aspects fonctionnels de ce langage.

Cependant, la programmation impérative (ou itérative) est possible, et parfois souhaitable (dans beaucoup de problèmes sur les vecteurs par exemple).

8.1. La conditionnelle

La conditionnelle est la classique construction « Si condition alors expression1 sinon expression2 ». Elle s'écrit

```
if condition then expression1 else expression2
```

Remarque : condition est de type booléen, expression1 et expression2 *sont nécessairement de même type. N'oubliez pas de vérifier la cohérence du typage.*

En fait, la partie else est facultative, et

```
if condition then expression1
```

se lit comme

```
if condition then expression1 else ()
```

Comme les expressions de la conditionnelle ont nécessairement le même type, cette syntaxe sans « else » n'est possible que si expression1 est de type unit.

```
# let vaut_un x = if x = 1 then true;;
Toplevel input:
>let vaut_un x = if x = 1 then true;;
>                ^^^^^^^^^^^^^^^^^^
This expression has type unit,
but is used with type bool.
```

Pour pallier cette incohérence de typage, on peut écrire (mais pas indifféremment) :

```
# let vaut_un_effet x = if x = 1 then print_string "true";;
vaut_un_effet : int -> unit = <fun>

# let vaut_un_predicat x = if x = 1 then true else false;;
vaut_un_predicat : int -> bool = <fun>

# vaut_un_effet 1;;
true- : unit = ()

# vaut_un_predicat 1;;
- : bool = true
```

Remarque : la conditionnelle n'est pas tant employée que cela, puisque le filtrage la remplace avantageusement le plus souvent. En particulier, il n'y a pas en Caml de « elif ».

8.2. Séquencement

Pour regrouper plusieurs expressions, on peut effectuer un *séquencement* : la syntaxe Caml est

```
begin
        expression1;
        expression2;
        ...
        expressionN;
end
```

Le résultat de cette séquence est celui de la dernière expression. Un tel séquencement n'a de sens que si les expressions précédentes agissent par effet de bord (Caml produit un avertissement dans le cas contraire).
Remarque : ce séquencement est intéressant dans le cas d'une conditionnelle, car la conditionnelle risque de s'arrêter trop tôt sinon.

Comparer par exemple les phrases suivantes :

```
# if 0 = 1 then begin
        print_int 2;
        print_int 3;
        end;;
- : unit = ()

# if 0 = 1 then
        print_int 2;
        print_int 3;
        ;;
3- : unit = ()
```

Remarque : on peut remplacer **begin** instructions **end** par (instructions).

8.3. Boucles

8.3.1. *Boucle inconditionnelle.* C'est la classique boucle for, dont la syntaxe est la suivante

```
for indice = valeur_initiale to valeur_finale do expression_pour_l_indice_i done
```

Remarque : on peut aussi décrémenter l'indice en mettant **downto** au lieu de **to**. L'indice, de type int, joue le rôle d'un identificateur local, et il n'apparaît pas forcément dans le corps de la boucle.
Remarque : la boucle for joue un peu le rôle d'un séquencement dont on aurait ordonné (et décrit) les instructions à l'aide d'un indice. En particulier, elles ne tolèrent pas les déclarations globales, et produisent un avertissement dans le cas où les expressions ne sont pas de type unit.
Remarque : la boucle for se terminant par **done**, il n'est pas utile d'utiliser le séquencement pour regrouper les éventuelles sous-instructions.

```
# for i = 1 to 5 do print_int i; print_newline () done;;
1
2
3
4
5
- : unit = ()
```

8.3.2. *Boucle conditionnelle.* C'est la non moins classique boucle while, dont voici la syntaxe :

```
while condition do expression done
```

Remarque : condition est un booléen, testé en entrée de boucle (contrairement à la boucle de type « repeat ... until ... », non implémentée en Caml).

Il est difficile de donner un exemple d'utilisation de boucle while sans utiliser la notion de référence, à laquelle nous consacrons la sous-section suivante.

```
# let i = ref 1 in
while !i < 10 do print_int !i; i :=  !i + 1 done;;
123456789- : unit = ()
```

8.4. Références

La notion de référence, similaire à celle de pointeur en C, sert à définir un contenant, que l'on donne avec son contenu initial. Il faut bien comprendre que le contenu de ce contenant peut être changé (mais que l'on doit conserver le même type).

La syntaxe est

```
let contenant = ref contenu
```

Pour accéder à contenu à partir de contenant, on écrit !contenant.
Remarque : contenant est de type type_du_contenu ref.

Pour modifier (ou réaffecter) le contenu de contenant, la syntaxe est

```
contenant := nouveau_contenu
```

Remarque : nouveau_contenu doit être du même type que contenu. Cette expression est de type unit (on agit par effet de bord).

```
# let y = ref 5;;
y : int ref = ref 5

# y := 7;;
- : unit = ()

# y := 5.6;;
Toplevel input:
>y := 5.6;;
>     ^^^
This expression has type float,
```

```
but is used with type int.
```

Il est essentiel de comprendre qu'une référence représente un contenu, et qu'il faut donc se méfier des déclarations, qui sont cette fois-ci dynamiques :

```
# let x = ref 113;;
x : int ref = ref 113

# let y = x;;
y : int ref = ref 113

# y := 597;;
- : unit = ()

# x;;
- : int ref = ref 597
```

Les noms x et y désignent ici le même contenant, tout changement du contenu de l'un s'appliquera à l'autre.

9. Types prédéfinis en Caml

Nous avons déjà recensé les types simples unit, bool, int, float, ainsi que les types composés obtenus à partir de la flèche −> ou du produit cartésien ∗. Nous ne traitons ici que de types prédéfinis par Caml, mais nous verrons en 11 que l'on peut en créer.

9.1. Le type string

C'est celui des chaînes de caractères, que l'on place entre guillemets.

Les termes d'une telle chaîne de caractères s sont indicés, le premier indice étant 0 (et non 1). Pour accéder au terme d'indice i de s, on écrit simplement s.[i]. Pour le modifier (un peu comme une référence), la syntaxe est

```
s.[i] <- nouvelle_valeur
```

où nouvelle_valeur est un caractère. On observe d'ailleurs que les chaînes de caractères se comportent un peu comme des références indexées (comme des vecteurs, en fait).

```
# let a = "test";;
a : string = "test"

# let b = a;;
b : string = "test"

# b.[0] <- 'b';;
- : unit = ()

# a;;
- : string = "best"
```

On accède à la longueur d'une chaîne par la primitive string_length. Pour créer une chaîne de caractères, on dispose de la fonction curryfiée make_string dont le premier argument est la longueur et le second le caractère de remplissage de la chaîne initiale :

```
# make_string 45 'z';;
- : string = "zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz"
```

On dispose également d'une fonction d'extraction de chaîne sub_string selon la syntaxe

```
sub_string chaîne début longueur
```

On peut concaténer deux chaînes avec le constructeur ^ :

```
# let rec je_sais = function
        | 0 -> "Je␣sais";
        | n ->  je_sais (n-1) ^ "␣que␣je␣sais";;
je_sais : int -> string = <fun>

# je_sais 3;;
- : string =
 "Je␣sais␣que␣je␣sais␣que␣je␣sais␣que␣je␣sais"
```

9.2. Les vecteurs ou tableaux

Les vecteurs (ou tableaux) sont des emplacements consécutifs en mémoire, où l'on peut stocker des éléments de même type. Si un tableau stocke des objets de type type_1, son type est type_1 vect.

Un tableau s'écrit

```
[|element_0; element_1; ... ; element_{N-1}|]
```

Remarque : un tableau a une taille fixée, à laquelle on peut accéder grâce à la fonction vect_length (de type 'a vect -> int).

On accède à l'élément d'indice $i \in [\![0, N-1]\!]$ d'un tableau T de taille N par l'instruction T.(i).

Remarque : pour les chaînes de caractères, on utilise les crochets, et on utilise les parenthèses pour les tableaux.

Remarque : faites attention aux indices (le premier indice est 0 et non 1).

Remarque : les tableaux étant de taille fixe, il est fréquent qu'on traite un problème les concernant par la programmation impérative.

Pour construire un tableau initial de taille N dont tous les termes valent a, on utilise la fonction make_vect en écrivant make_vect N a :

```
# make_vect 3 2.5;;
- : float vect = [|2.5; 2.5; 2.5|]
```

On peut aussi réaffecter l'élément d'indice i d'un tableau T en lui donnant la valeur b (de même type que la valeur actuelle) selon la syntaxe :

```
T.(i) <- b
```

La liaison entre vecteurs est la même que pour les chaînes de caractères :

```
# let T = [|3; 10; 3|];;
T : int vect = [|3; 10; 3|]

# let T' = T;;
T' : int vect = [|3; 10; 3|]

# T'.(1) <- 3;;
- : unit = ()

# T;;
- : int vect = [|3; 3; 3|]
```

9.3. Les listes

La définition d'une liste est récursive : la liste vide est une liste de type 'a list , et si L est une liste dont les termes sont de type type_1, et a un objet de type type_1, alors on obtient une nouvelle liste, de même type, en plaçant a devant L, grâce au constructeur infixe « : : ». On dit que a (resp. L) est la *tête* (resp. la *queue*, ou la *fin*) de la liste obtenue a :: L.

```
# let l = [];;
l : 'a list = []

# let l' = 3 :: l;;
l' : int list = [3]
```

```
# let l'' = 5 :: 8 :: l';;
l'' : int list = [5; 8; 3]
```

Remarque : à l'inverse des tableaux, il est facile d'ajouter un terme en tête d'une liste. En contrepartie, l'accès à un terme d'une liste ne s'effectue pas en temps constant (contrairement au cas des tableaux). C'est le terme en tête de liste que l'on obtient le plus rapidement, puis le suivant, etc.
Remarque : par nature, les listes privilégient une programmation récursive.

On accède à la taille (ou longueur) d'une liste par la fonction list_length , que l'on peut reprogrammer ainsi :

```
# let rec taille_liste = function
        | [] -> 0
        | a :: q -> 1 + taille_liste q;;
taille_liste : 'a list -> int = <fun>

# taille_liste [2; 5; 8; 9];;
- : int = 4
```

Remarque : il est inutile d'identifier la tête de la liste dans le second cas de filtrage, de sorte que a :: q peut être remplacé par _ :: q.

On accède à la tête (resp. la queue) d'une liste grâce à la fonction hd (resp. tl).

On dispose aussi d'un opérateur de concaténation entre listes (de même type), noté @.

Enfin, les fonctions mem et index de types respectifs ('a -> 'a list -> bool) et ('a -> 'a list -> int) teste l'appartenance (resp. donne le premier indice) d'un objet dans une liste.

9.4. Exceptions

Il existe en Caml un type tout à fait particulier, celui des exceptions exn. Dans une première approche, on peut voir une exception comme une valeur que l'on renvoie lorsque l'expression n'est pas évaluable.

```
# 1 / 0;;
Uncaught exception: Division_by_zero

# index 3 [1; 1; 1];;
Uncaught exception: Not_found
```

Pour *déclencher* (ou *lever*) une exception, on utilise la primitive raise, qui, toutes affaires cessantes (cette primitive est prioritaire), renvoie l'exception détectée :

```
# 1 / 0 = 0 || 0 = 0;;
Uncaught exception: Division_by_zero
```

Il est possible de définir soi-même une exception, en l'introduisant selon l'une des syntaxes suivantes

```
exception ma_nouvelle_exception

exception ma_nouvelle_exception of type_de_cette_exception
```

Dans le premier cas, un déclenchement de l'exception produira Uncaught **exception**: ma_nouvelle_exception, alors que Caml répondra Uncaught **exception**: ma_nouvelle_exception valeur_déclenchant_l_exception dans l'autre cas :

```
# exception egalite;;
Exception egalite defined.

# let taux f x y = if x = y then raise egalite else (f y -. f x) /. (y -. x);;
taux : (float -> float) -> float -> float -> float = <fun>

# taux cos 0. 0.01;;
- : float = -0.00499995833347

# taux sin 3. 3.;;
Uncaught exception: egalite
```

```
# exception egalite' of float;;
exception egalite' defined.

# let taux' f x y = if x = y then raise (egalite' x) else (f y -. f x) /. (y -. x);;
taux' : (float -> float) -> float -> float -> float = <fun>

# taux' cos 0. 0.01;;
- : float = -0.00499995833347

# taux' sin 3. 3.;;
Uncaught exception: egalite' 3.0
```

Les deux exceptions les plus courantes sont Failure et Invalid_argument. On a donc implémenté des fonctions failwith et invalid_arg, de type string -> 'a, qui lèvent directement ces exceptions :

```
# let racines_de_ax_plus_b = fun
        | 0. 0. -> failwith "tout␣reel␣est␣solution"
        | 0. _ -> raise (Failure "aucune␣solution")
        | a b -> -. b /. a;;
racines_de_ax_plus_b : float -> float -> float = <fun>

# racines_de_ax_plus_b 0. 0.;;
Uncaught exception: Failure "tout␣reel␣est␣solution"

# racines_de_ax_plus_b 0. 1.;;
Uncaught exception: Failure "aucune␣solution"

# racines_de_ax_plus_b 2. 1.;;
- : float = -0.5
```

uncaught **exception**: une_certaine_exception signifie que l'exception une_certaine_exception n'a pas été *rattrapée* : comment rattraper une exception, et qu'entend-on au juste par là ? On peut dire à Caml quelle attitude adopter dans le cas où une exception serait levée, selon la syntaxe

```
try expression with attitude_à_adopter_en_cas_d_exception
```

où attitude_à_adopter_en_cas_d_exception est un filtrage dont les filtres sont des exceptions, que Caml utilise si expression a déclenché une exception :

```
# let f x y = try x / y with Division_by_zero -> 42;;
f : int -> int -> int = <fun>

# f 3 2;;
- : int = 1

# f 4 0;;
- : int = 42
```

Ce filtrage sur les exceptions revêt un grand intérêt dans le cas d'exceptions paramétrées. Par exemple, on peut prolonger la fonction taux d'accroissement de la fonction sinus de la manière suivante :

```
# exception egalite of float;;
Exception egalite defined.

# let taux_de_sinus x y = try if x = y then
                                raise (egalite x)
                                else (sin y -. sin x) /. (y -. x)
                        with egalite x -> cos x;;
taux_de_sinus : float -> float -> float = <fun>

# taux_de_sinus 0. 0.01;;
```

```
- : float = 0.999983333417

# taux_de_sinus 1. 1.;;
- : float = 0.540302305868
```

10. Bref retour sur le filtrage

Il ne vous a pas échappé, dans l'exemple précédent, que la fonction taux_de_sinus pouvait être écrite, de manière bien plus simple, de la manière suivante :

```
# let taux_de_sinus x y = if x = y then
                            cos x
                          else (sin y -. sin x) /. (y -. x);;
taux_de_sinus : float -> float -> float = <fun>
```

Le seul avantage que l'on pourrait concéder à la première programmation est la meilleure mise en valeur du fait que l'on soit tombé sur un cas « interdit ».

Il est naturel de vouloir remplacer dans la seconde écriture la conditionnelle par un filtrage :

```
# let taux_de_sinus = fun
        | x x -> cos x
        | x y -> (sin y -. sin x) /. (y -. x);;
Toplevel input:
>       | x x -> cos x
>              ^
The variable x is bound several times in this pattern.
```

Pourquoi Caml n'a-t-il pas accepté ce filtrage ? Chaque identificateur ne peut apparaître qu'une seule fois dans un filtre. Les seuls tests d'égalité possibles dans un filtrage simple sont ceux avec une constante.

Pour effectuer malgré tout un test comme dans la fonction ci-dessus, on peut utiliser une *garde*, avec le mot-clé when :

```
# let taux_de_sinus = fun
        | x y when x = y -> cos x
        | x y -> (sin y -. sin x) /. (y -. x);;
taux_de_sinus : float -> float -> float = <fun>
```

Cherchons maintenant à écrire une fonction f de type 'a list -> 'a list, qui, à une liste d'au moins deux termes, enlève les deux premiers termes s'ils sont distincts, ajoute le premier terme s'ils sont égaux, et laisse toute autre liste invariante.

Parmi les nombreuses programmations possibles, on propose :

```
# let f = function
        | a :: a' :: q when a = a' -> a :: a :: a' :: q
        | a :: a' :: q -> q
        | l -> l;;
f : 'a list -> 'a list = <fun>

# f [1; 1; 3];;
- : int list = [1; 1; 1; 3]

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

Le premier filtre est lisible, mais l'écriture du résultat obtenu ne rend pas justice à la simplicité de l'opération effectuée, à savoir ajouter le premier terme en tête de liste. Pour mieux mettre en valeur cette simplicité, et rendre le programme plus clair, on peut utiliser le mot-clé **as**, qui permet d'identifier le cas de filtrage. Ainsi peut-on réécrire la fonction f de la manière suivante :

```
# let f = function
        | a :: a' :: q as l when a = a' -> a :: l
        | a :: a' :: q -> q
        | l -> l;;
f : 'a list -> 'a list = <fun>
```

L'opération effectuée dans le premier cas de filtrage est maintenant beaucoup plus lisible.

11. Types définis par l'utilisateur

En Caml, on peut définir ses propres types.

11.1. Type constant, type paramétré

On peut définir un type en lui donnant un nom, en écrivant

```
type nom_du_type = valeur_constante_du_type
```

```
# type Isole = unique;;
Type Isole defined.

# let f = function unique -> 1;;
f : Isole -> int = <fun>

# let a = unique;;
a : Isole = unique

# f a;;
- : int = 1
```

Remarque : le type unit est un exemple de type constant (déjà défini).

On peut créer également un type paramétré, en écrivant

```
type nom_du_type of type_du_type
```

```
# type Param = p of int;;
Type Param defined.

# let g = function p m -> p (m * m);;
g : Param -> Param = <fun>

# let a = p 3;;
a : Param = p 3

# g a;;
- : Param = p 9
```

Notons que l'on peut définir des types polymorphes, en utilisant les habituelles variables de type comme 'a et 'b, en les déclarant comme ceci :

```
type 'a nom_du_type
```

```
# type 'a Polym = q of 'a;;
Type Polym defined.

# let x = q 3;;
x : int Polym = q 3

# let h (q a) = [q a];;
h : 'a Polym -> 'a Polym list = <fun>

# h (q 3);;
- : int Polym list = [q 3]
```

Remarque : on peut également effectuer une abréviation de type selon l'instruction :

```
type nom_de_l_abreviation == type_deja_defini
```

```
# type couple = int * float;;
Type couple defined.

# let phi = function (a : couple) -> fst a;;
phi : couple -> int = <fun>

# phi (4, 5.5);;
- : int = 4
```

Remarque : cet exemple vous montre également comment forcer le type d'un argument.

11.2. Type somme et type produit

Un type somme correspond à une union disjointe de types déjà connus. La syntaxe est

```
type nom_du_type = | type_1 | type_2 | ... | type_n
```

Remarque : la première barre verticale est facultative.

```
# type nombre = | Entier of int | Reel of float;;
Type nombre defined.

# let somme = fun
        | (Entier a) (Entier b) -> Entier (a + b)
        | (Reel a) (Entier b) -> Reel (a +. float_of_int b)
        | (Entier a) (Reel b) -> Reel (float_of_int a +. b)
        | (Reel a) (Reel b) -> Reel (a +. b);;
somme : nombre -> nombre -> nombre = <fun>

# somme (Entier 3) (Reel 5.7);;
- : nombre = Reel 8.7
```

Un type produit (ou enregistrement) correspond à un produit cartésien de types déjà connus. La syntaxe est

```
type nom_du_type = {Etiquette_1 : type_1; ... ; Etiquette_n : type_n}
```

La déclaration d'un objet ayant ce type s'écrit

```
let identificateur = {Etiquette_1 = valeur_de_type_1; ... ;
        Etiquette_n = valeur_de_type_n}
```

On peut préciser qu'un type enregistrement est *mutable* (on peut en modifier le contenu, à la manière d'un tableau), en insérant le mot clef **mutable** avant l'étiquette concernée :

Pour modifier le contenu d'une étiquette mutable, on écrira

```
nom_du_type.etiquette <- nouvelle_valeur
```

```
# type bulletin = {Maths : float; Physique : float; Informatique : float;
mutable Appreciation : string; LeReste : int};;
Type bulletin defined.

# let eleve = {Maths = 18.8; Physique = 19.2; Informatique = 16.7;
Appreciation = "nul"; LeReste = 14};;
eleve : bulletin = {Maths = 18.8; Physique = 19.2; Informatique = 16.7;
Appreciation = "nul"; LeReste = 14}

# eleve.LeReste;;
- : int = 14

# eleve.Appreciation <- "convenable";;
- : unit = ()
```

```
# eleve;;
- : bulletin =
 {Maths = 18.8; Physique = 19.2; Informatique = 16.7;
  Appreciation = "convenable"; LeReste = 14}
```

11.3. Type récursif

On peut définir un type de manière récursive, exactement comme on le ferait pour un ensemble inductif :

```
# type entier = | Zero | succ of entier;;
Type entier defined.

# let rec addition a b = match a with
        | Zero -> b
        | succ c -> succ (addition c b);;
addition : entier -> entier -> entier = <fun>

# addition (succ (succ Zero)) (succ (succ Zero));;
- : entier = succ (succ (succ (succ Zero)))

# type liste = | Liste_Vide | Cons of int * liste;;
Type liste defined.

# let queue = function
        | Liste_Vide -> failwith "Pas de queue pour la liste vide"
        | Cons (a, li) -> li;;
queue : liste -> liste = <fun>

# queue (Cons(3, Cons(5, Cons(7, Liste_Vide))));;
- : liste = Cons (5, Cons (7, Liste_Vide))

# type arbre_binaire = | Feuille of int | Noeud of int * arbre_binaire * arbre_binaire;;
Type arbre_binaire defined.

# let bonsai = Noeud (1, Noeud (2, Feuille 3, Feuille 4), Feuille 5);;
bonsai : arbre_binaire =
 Noeud (1, Noeud (2, Feuille 3, Feuille 4), Feuille 5)
```

Remarque : pour de tels types, la programmation récursive s'impose naturellement, et la structure des programmes découle simplement de la structure inductive (*i.e.* des cas de base et des constructeurs).
Remarque : on peut également définir des types mutuellement récursifs, de la même manière que l'on définit des fonctions récursives croisées.

Deuxième partie

Méthodes de programmation

CHAPITRE III

Programmation impérative

1. Correction et terminaison

Étant donné un algorithme, il est important de vérifier sa *correction* (ou sa validité), *i.e.* qu'il renvoie bien le résultat souhaité, et sa *terminaison*, *i.e.* qu'il ne boucle pas indéfiniment (pour certains algorithmes récursifs ou pour des boucles conditionnelles).

Remarque : on peut également se poser la question de son efficacité (*i.e.* de sa rapidité et de l'espace qu'il prend), ce que nous ferons lors du cours sur la complexité.

En réalité, on ne peut pas prouver, en toute généralité, la terminaison d'un algorithme donné. Penser à la suite de Syracuse. Pire, on peut prouver qu'on ne peut pas le prouver.

Nous ne rentrerons pas dans les détails de la preuve d'un programme impératif : c'est un sujet délicat, qui peut vite devenir très compliqué. Donnons le principe général de preuve d'une boucle.

2. Notion d'invariant de boucle

Une boucle, qu'elle soit conditionnelle ou inconditionnelle (on dit aussi indexée), peut se schématiser de la manière suivante :

On part d'une précondition (ou condition d'entrée) dans le module, qui décrit l'état des variables avant d'entrer dans la boucle.

On effectue un certain nombre de passages dans la boucle, non nécessairement connu à l'avance.

En sortie de boucle, la condition de sortie (ou postcondition), qui doit correspondre au résultat souhaité.

On appelle *invariant de boucle* un prédicat sur les arguments de la boucle, renvoyant vrai avant passage dans la boucle, restant vrai après passage dans la boucle : en sortie de boucle, l'invariant de boucle est vrai, ce qui doit nous permettre de prouver qu'on a obtenu le résultat souhaité.

Remarque : il s'agit donc de montrer un résultat par récurrence finie.

Prenons l'exemple de la fonction factorielle, de manière impérative avec boucle indexée :

```
# let fact n =
        let temp = ref 1 in
                for i = 1 to n do temp := !temp * i done;
                !temp;;
fact : int -> int = <fun>

# fact 5;;
- : int = 120
```

Pourquoi cette fonction renvoie-t-elle bien la bonne valeur ? La précondition est : « temp a pour valeur 1 ». Montrons que pour tout $k \in [\![0, n]\!]$, la valeur de temp après la k-ème itération est $k!$.

C'est vrai pour $k = 0$ (avant l'entrée dans la boucle), et, supposant la propriété vraie à un rang $k \in [\![0, n-1]\!]$, alors après un nouveau passage dans la boucle (pour $i = k + 1$), la valeur de temp est $(k + 1)!$.

En sortie de boucle, *i.e.* après la n-ième itération, on obtient $n!$ comme valeur pour temp.

En fait, il n'est pas nécessaire de tant détailler la correction de cette boucle (une preuve n'est pas systématique). On se contentera de mettre en commentaire l'invariant de boucle :

```
# let fact n =
        let temp = ref 1 in
                for i = 1 to n do temp := !temp * i done;
                (* invariant : !temp = i! *)
                !temp;;
fact : int -> int = <fun>
```

Remarque : bien sûr, cette fonction produit un résultat étrange si on prend un argument négatif (on aurait pu lever une exception).
Remarque : que se passe-t-il si on omet la dernière ligne ? Et si on la remplace par temp ?
Remarque : en réalité, il est souvent plus pratique de déterminer le corps de la boucle, d'en trouver un invariant, et d'en déduire la précondition.
Remarque : l'invariant permet de faire le pont entre la précondition et la postcondition.

Pour donner un exemple avec une boucle conditionnelle, reprenons la fonction factorielle :

```
# let factw n =
        let i = ref n in
                let temp = ref 1 in
                        while !i > 1 do temp := !temp * !i; i := !i - 1  done;
                        (* invariant : (!i)! * !temp = n! *)
                        !temp;;
fact : int -> int = <fun>

# factw 5;;
- : int = 120
```

La fonction termine (même si $n < 1$) car !i est décrémenté à chaque passage dans la boucle, et ne peut donc rester indéfiniment strictement supérieur à 1. En sortie, !i vaut 1 : grâce à l'invariant de boucle, on constate que !temp vaut bien la valeur souhaitée.

Fonction puissance de manière naïve :

```
# let puissance a b = let temp = ref 1 in
        for i = 1 to b do temp := !temp * a done;
        (* invariant : !temp = a ^ i *)
        !temp;;
puissance : int -> int -> int = <fun>

# puissance 3 4;;
- : int = 81
```

Remarque : on aurait également pu introduire l'*itératrice* $f : x \mapsto ax$, que l'on effectue à chaque passage dans la boucle.

Fonction somme

```
# let somme t = let temp = ref 0 in
        for i = 0 to vect_length t - 1 do temp :=  !temp + t.(i) done;
        (* invariant : !temp = somme des termes de t d'indices 0 à i *)
        !temp;;
somme : int vect -> int = <fun>

# somme [|3; 5; 11|];;
- : int = 19
```

```
# let fact n =
        let temp = ref 1 in
                for i = 1 to n do temp := !temp * i done;
                !temp;;
fact : int -> int = <fun>

# let factw n =
        let i = ref n in
                let temp = ref 1 in
                        while !i > 1 do temp := !temp * !i; i := !i - 1  done;
                        !temp;;
fact : int -> int = <fun>

# factw 5;;
- : int = 120

# let puissance a b = let temp = ref 1 in
        for i = 1 to b do temp := !temp * a done;
        !temp;;
puissance : int -> int -> int = <fun>

# puissance 3 4;;
- : int = 81

# let somme t = let temp = ref 0 in
        for i = 0 to vect_length t - 1 do temp :=  !temp + t.(i) done;
        !temp;;
somme : int vect -> int = <fun>

# somme [|3; 5; 11|];;
- : int = 19
```

CHAPITRE IV

Récursivité

1. Introduction

Informellement, une fonction est dite *récursive* lorsqu'elle s'appelle elle-même dans sa définition. Le problème de terminaison, rencontré pour les boucles conditionnelles, se rencontre évidemment dans ce cadre. Nous ne pourrons d'ailleurs pas déterminer si une fonction récursive quelconque termine (voir l'exemple de la suite de Syracuse plus loin). En réalité, il n'est même pas envisageable de pouvoir répondre à cette question un jour

Pour préciser à Caml que l'on définit une fonction récursive, on utilise la déclaration :

```
let rec fonction_recursive =
```

Pour prouver la terminaison et la correction d'une fonction récursive, nous allons introduire un cadre mathématique adapté, qui s'appuie sur la notion de relation d'ordre.

Pour comprendre le fonctionnement ou débuguer les fonctions récursives, on peut utiliser la fonction trace de type string -> unit.

2. Compléments sur les relations d'ordre

Dans la suite, nous considèrerons un ensemble E muni d'un ordre total ou partiel $\preceq$, dont nous noterons $\prec$ l'ordre strict associé.

La récursivité est liée à la notion de preuve par récurrence. On a vu en mathématiques des exemples de preuves par récurrence simple, cette méthode permettant de montrer la terminaison et la correction des fonctions suivantes :

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

# trace "fact";;
The function fact is now traced.
- : unit = ()

# fact 5;;
fact <-- 5
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
fact --> 6
fact --> 24
fact --> 120
- : int = 120
```

```
# let rec f = function
        | 0 -> 5.
        | n -> (f (n - 1) +. 5. /. f (n - 1)) /. 2.;;
f : int -> float = <fun>
```

```
# f 4;;
- : float = 2.23606889564

# sqrt 5.;;
- : float = 2.2360679775
```

La preuve de ces fonctions utilise de manière naturelle les notions de prédécesseur d'un entier naturel non nul et de successeur d'un entier naturel.

On a également vu la notion de preuve par récurrence forte (ou avec prédécesseurs), qui permet de prouver des fonctions récursives avec plusieurs appels :

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

# fibo 5;;
- : int = 8
```

En réalité, la démonstration par récurrence simple est difficile à généraliser, puisque fondée sur ces notions de prédécesseur et de successeur. Par exemple, pour l'ordre lexicographique sur $\mathbb{N}^2$, pourtant total, certains éléments non nuls n'admettent pas de prédécesseur.

De plus, il peut être utile de travailler sur un ensemble partiellement ordonné, tel que $\mathbb{N}\setminus\{0,1\}$ pour la divisibilité (dans la recherche de la décomposition d'un entier en produit de facteurs premiers par exemple).

C'est donc la récurrence forte que nous allons généraliser.

Définition (Élément minimal)

Un élément m d'une partie Ω de E est dit *minimal (dans Ω)* s'il n'en existe pas de strictement plus petit, *i.e.*

$$\forall x \in \Omega, \quad (x \preceq m) \Rightarrow (x = m)$$

2.a

Remarque : ne pas confondre élément minimal et élément minimum (sauf si l'ordre est total).
Remarque : on peut bien sûr définir la notion d'élément maximal, mais elle ne nous sera pas très utile ici.

Exemple (Éléments minimaux)

(1) Le seul élément minimal de $\mathbb{N}$ usuel est 0.

(2) Dans $\mathbb{Z}$ usuel, il n'y a pas d'élément minimal (il n'y a pas nécessairement existence d'un élément minimal).

(3) Les éléments minimaux de $\mathbb{N}\setminus\{0,1\}$ pour la divisibilité sont les nombres premiers. On observe en particulier qu'il n'y a pas nécessairement unicité d'un élément minimal.

i

Définition (Ensemble bien fondé)

On dit que l'ensemble ordonné $(E, \preceq)$ est *bien fondé* (et que l'ordre $\preceq$ est bien fondé) si toute partie non vide de E admet un élément minimal.

2.b

Remarque : bien sûr, on ne demande pas l'existence d'un élément minimal pour E seulement.

Exemple (Ensemble bien fondé)

(1) $\mathbb{N}$ usuel est bien fondé. Mieux, toute partie non vide admet un plus petit élément (on le dit *bien ordonné*). En fait $(E, \preceq)$ est bien ordonné si et seulement si $(E, \preceq)$ est bien fondé et totalement ordonné.

(2) $\mathbb{Z}$ n'est pas bien fondé.

(3) $\mathbb{N}$ pour la divisibilité est bien fondé.

(4) Toute partie d'un ensemble bien fondé $(E, \preceq)$ est bien fondée pour l'ordre induit.

(5) L'ordre lexicographique est bien fondé (et bien ordonné).

ii

Théorème (Caractérisation des ensembles bien fondés)

$(E, \preceq)$ est bien fondé si et seulement si il n'existe pas de suite infinie strictement décroissante d'éléments de E.

2.a

Démonstration

S'il existe une suite infinie strictement décroissante (x_n) d'éléments de E, alors $\{x_n, n \in \mathbb{N}\}$ est une partie de E n'admettant pas d'élément minimal.
S'il n'existe pas de telle suite, alors soit Ω une partie non vide de E : soit $x_0 \in \Omega$. Supposons pour $n \in \mathbb{N}$ fixé, avoir construit $x_0, \dots, x_n$. Si x_n n'est pas minimal dans Ω, on prend $x_{n+1} \in \Omega$ tel que $x_{n+1} \prec x_n$. La suite ainsi construite est nécessairement finie, s'arrêtant à un indice N, et x_N est minimal dans Ω.

□

Remarque : si E est fini, alors il est bien fondé. Plus généralement, si, pour tout $e \in E$, E n'a qu'un nombre fini d'éléments inférieurs à e, alors E est bien fondé.

Théorème d'induction

On suppose $(E, \preceq)$ bien fondé. Soit $\mathcal{M}$ l'ensemble des éléments minimaux de E. Soit p un prédicat sur E tel que :

(1) $\forall\, x \in \mathcal{M}, \quad p(x)$.

(2) $\forall\, x \in E, \quad (\forall\, y \prec x,\ p(y)) \Rightarrow p(x)$.

p est alors vérifié sur tout E.

2.b

Démonstration

Soit Ω l'ensemble des éléments de E où p est faux : supposons Ω non vide. Ω admet un élément minimal x. x n'est pas minimal dans E (car $\Omega \cap \mathcal{M} = \emptyset$). De plus, pour tout élément y de E tel que $y \prec x$, on a $p(y)$: d'après (2), $p(x)$ est vérifié, c'est absurde.

□

Remarque : en toute logique, la seconde condition entraîne la première, qui est donc superflue. Cependant, on préfère cette version plus lisible.
Remarque : on pouvait aussi raisonner en termes de parties de E : E est la seule partie de E contenant $\mathcal{M}$ et qui comprend tout élément dont elle comprend les éléments strictements inférieurs dans E.
Remarque : on parle d'hypothèse d'induction.

3. Terminaison et correction des fonctions récursives

On considère une fonction récursive f de A dans B. On suppose disposer d'une fonction Φ de A dans un ensemble bien fondé E, que l'on appelle parfois *graduation*.

Définition (Cas de base)

On appelle *cas de base* de f tout élément a de A tel que $\Phi(a)$ soit minimal dans $\Phi(A)$.

3.a

Notons $\mathcal{B}$ l'ensemble des cas de base, et $\mathcal{M} = \Phi(\mathcal{B})$.

Remarque : on n'impose pas $\Phi(a)$ minimal dans E mais dans $\Phi(A)$.

Théorème (Terminaison d'une fonction récursive)

On suppose que f termine pour les cas de base, et que pour tout $x \in A$, l'exécution de $f(x)$ ne produit [a] qu'un nombre fini d'appels à $y_1, \dots, y_k \in A$ vérifiant $\Phi(y_j) \prec \Phi(x)$ pour tout $j \in [\![1, k]\!]$.
La fonction f termine alors pour toute valeur $x \in A$ de son argument.

3.a

a. outre un nombre fini d'opérations élémentaires

Démonstration

Notons p le prédicat sur $\Phi(A)$: $p(e)$ (est vrai) si et seulement si f termine pour tout élément de $\Phi^{-1}(\{e\})$. D'après le théorème d'induction, p est vérifié sur $\Phi(A)$: le théorème est prouvé.

□

De la même manière, on montre :

Théorème (Correction d'une fonction récursive)

Soit p_f le prédicat défini, pour tout $x \in A$, par : « $f(x)$ retourne (en temps fini) le résultat souhaité ».
On suppose :

(1) $\forall x \in \mathcal{B}, p_f(x)$

(2) pour tout $x \in A$, l'exécution de $f(x)$ ne produit qu'un nombre fini d'appels, à $y_1, \dots, y_k \in A$ vérifiant $\Phi(y_j) \prec \Phi(x)$ pour tout $j \in [\![1, k]\!]$, et : $(\forall j \in [\![1, k]\!],\ p_f(y_j)) \Rightarrow p_f(x)$

La fonction f est alors correcte (et termine) pour toute valeur $x \in A$ de son argument.

3.b

Remarque : en pratique, il peut être avantageux de chercher d'abord un ordre bien fondé adapté à notre problème, puis de programmer, que l'inverse.

D'ailleurs, la difficulté réside parfois dans le choix de E et de la graduation Φ.

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

# trace "fact";;
The function fact is now traced.
- : unit = ()

# fact 5;;
fact <-- 5
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
fact --> 6
fact --> 24
fact --> 120
- : int = 120

# let rec f = function
        | 0 -> 5.
        | n -> (f (n - 1) +. 5. /. f (n - 1)) /. 2.;;
f : int -> float = <fun>

# f 4;;
- : float = 2.23606889564

# sqrt 5.;;
- : float = 2.2360679775

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

# fibo 5;;
- : int = 8
```

```
# let rec syracuse = function
        | 1 -> print_string "true"; print_newline ()
        | n when n mod 2 = 0 -> syracuse (n / 2)
        | n -> syracuse (3 * n + 1);;
syracuse : int -> unit = <fun>

# for i = 100 to 110 do syracuse i done;;
true
true
true
true
true
true
true
true
true
true
true
- : unit = ()

# trace "syracuse";;
The function syracuse is now traced.
- : unit = ()

# syracuse 10;;
syracuse <-- 10
syracuse <-- 5
syracuse <-- 16
syracuse <-- 8
syracuse <-- 4
syracuse <-- 2
syracuse <-- 1
true
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
- : unit = ()

# syracuse 36;;
syracuse <-- 36
syracuse <-- 18
syracuse <-- 9
syracuse <-- 28
syracuse <-- 14
syracuse <-- 7
syracuse <-- 22
syracuse <-- 11
syracuse <-- 34
syracuse <-- 17
syracuse <-- 52
syracuse <-- 26
syracuse <-- 13
syracuse <-- 40
syracuse <-- 20
```

```
syracuse <-- 10
syracuse <-- 5
syracuse <-- 16
syracuse <-- 8
syracuse <-- 4
syracuse <-- 2
syracuse <-- 1
true
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
syracuse --> ()
- : unit = ()
```

4. Récursivité croisée

Il arrive que l'on définisse deux ou plusieurs fonctions de manière récursive, qui s'appellent mutuellement : on parle de récursivité *croisée*. La syntaxe Caml est

```
let rec fonction 1 = définition de fonction1
        and
fonction2 = définition de fonction2
```

Par exemple, on peut définir les fonctions pair et impair de type int -> bool de la manière suivante :

```
# let rec pair = function | 0 -> true | n -> impair (n - 1)
        and
impair = function | 0 -> false | n -> pair (n - 1);;
pair : int -> bool = <fun>
impair : int -> bool = <fun>

# trace "pair";;
The function pair is now traced.
- : unit = ()

# trace "impair";;
The function impair is now traced.
- : unit = ()

# pair 5;;
pair <-- 5
impair <-- 4
pair <-- 3
impair <-- 2
pair <-- 1
impair <-- 0
impair --> false
pair --> false
impair --> false
pair --> false
impair --> false
pair --> false
- : bool = false
```

5. Gestion d'une fonction récursive

5.1. Notion de pile d'exécution

Pour gérer les appels récursifs, Caml effectue un empilement des valeurs à calculer (on parle de pile d'exécution), qu'il finit par dépiler pour arriver au calcul final. Par exemple, pour la fonction factorielle, nous obtenons le comportement suivant :

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

# trace "fact";;
The function fact is now traced.
- : unit = ()

# fact 12;;
fact <-- 12
fact <-- 11
fact <-- 10
```

```
fact <-- 9
fact <-- 8
fact <-- 7
fact <-- 6
fact <-- 5
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
fact --> 6
fact --> 24
fact --> 120
fact --> 720
fact --> 5040
fact --> 40320
fact --> 362880
fact --> 3628800
fact --> 39916800
fact --> 479001600
- : int = 479001600
```

En cas de plusieurs appels récursifs à chaque passage, cela se complique, comme le montre l'exemple de la suite de Fibonacci :

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

# trace "fibo";;
The function fibo is now traced.
- : unit = ()

# fibo 4;;
fibo <-- 4
fibo <-- 2
fibo <-- 0
fibo --> 1
fibo <-- 1
fibo --> 1
fibo --> 2
fibo <-- 3
fibo <-- 1
fibo --> 1
fibo <-- 2
fibo <-- 0
fibo --> 1
fibo <-- 1
fibo --> 1
fibo --> 2
fibo --> 3
fibo --> 5
- : int = 5
```

Pour mieux comprendre le calcul de fibo 4, on peut dessiner l'*arbre des appels* de fibo 4.

La fonction trace est donc très utile pour comprendre le fonctionnement d'une fonction récursive. Cependant, il faut savoir qu'elle ne gère pas (ou mal en tout cas) les fonctions polymorphes, et donc beaucoup de fonctions sur les listes chaînées :

```
# let rec pasderepet = function
        | [] -> []
        | [a] -> [a]
        | a :: q -> if a = hd q then pasderepet q else a :: pasderepet q;;
pasderepet : 'a list -> 'a list = <fun>

# trace "pasderepet";;
The function pasderepet is now traced.
- : unit = ()

# pasderepet [1; 3; 3; 3; 5; 5; 10; 10; 10];;
pasderepet <-- [<poly>; <poly>; <poly>; <poly>; <poly>; <poly>; <poly>;
                <poly>; <poly>]
pasderepet <-- [<poly>; <poly>; <poly>; <poly>; <poly>; <poly>; <poly>;
                <poly>]
pasderepet <-- [<poly>; <poly>; <poly>; <poly>; <poly>; <poly>; <poly>]
pasderepet <-- [<poly>; <poly>; <poly>; <poly>; <poly>; <poly>]
pasderepet <-- [<poly>; <poly>; <poly>; <poly>; <poly>]
pasderepet <-- [<poly>; <poly>; <poly>; <poly>]
pasderepet <-- [<poly>; <poly>; <poly>]
pasderepet <-- [<poly>; <poly>]
pasderepet <-- [<poly>]
pasderepet --> [<poly>]
pasderepet --> [<poly>]
pasderepet --> [<poly>]
pasderepet --> [<poly>; <poly>]
pasderepet --> [<poly>; <poly>]
pasderepet --> [<poly>; <poly>; <poly>]
pasderepet --> [<poly>; <poly>; <poly>]
pasderepet --> [<poly>; <poly>; <poly>]
pasderepet --> [<poly>; <poly>; <poly>; <poly>]
- : int list = [1; 3; 5; 10]
```

Pour remédier à ce problème, on peut forcer le typage de la fonction pasderepet, en précisant le type de son argument

```
# let rec pasderepet (l : int list) = match l with
        | [] -> []
        | [a] -> [a]
        | a :: q -> if a = hd q then pasderepet q else a :: pasderepet q;;
pasderepet : int list -> int list = <fun>

# trace "pasderepet";;
The function pasderepet is now traced.
- : unit = ()

# pasderepet [1; 3; 3; 3; 5; 5; 10; 10; 10];;
pasderepet <-- [1; 3; 3; 3; 5; 5; 10; 10; 10]
pasderepet <-- [3; 3; 3; 5; 5; 10; 10; 10]
pasderepet <-- [3; 3; 5; 5; 10; 10; 10]
pasderepet <-- [3; 5; 5; 10; 10; 10]
pasderepet <-- [5; 5; 10; 10; 10]
pasderepet <-- [5; 10; 10; 10]
pasderepet <-- [10; 10; 10]
pasderepet <-- [10; 10]
pasderepet <-- [10]
```

```
pasderepet --> [10]
pasderepet --> [10]
pasderepet --> [10]
pasderepet --> [5; 10]
pasderepet --> [5; 10]
pasderepet --> [3; 5; 10]
pasderepet --> [3; 5; 10]
pasderepet --> [3; 5; 10]
pasderepet --> [1; 3; 5; 10]
- : int list = [1; 3; 5; 10]
```

Remarque : il est intéressant de retenir cette manière de forcer le typage. On peut d'ailleurs observer que l'on peut aussi forcer le typage à l'arrivée :

```
# let evalue x f = (f x : int);;
evalue : 'a -> ('a -> int) -> int = <fun>
```

5.2. Récursivité terminale

Parmi les fonctions récursives, il existe une sous-classe, particulièrement facile à gérer : celle des fonctions à récursivité dite *terminale*. Une fonction est dite récursive terminale si, dans le corps de sa définition, *l'unique appel* récursif se fait en toute *fin de parcours*.

Remarque : en particulier, on n'effectue aucune opération sur l'appel récursif (par exemple, la fonction fact n'est pas récursive terminale).

Dans le cas d'une fonction récursive terminale, les calculs ou effets de bord précédant cet unique appel récursif deviennent jetables, et il n'y a plus pour Caml qu'à calculer la valeur pour cet appel. Il n'y a donc pas d'empilement, et la complexité spatiale d'une telle fonction est réduite (comme dans le cas d'une programmation impérative). On ne risque pas dans ce cas de dépasser la taille limite de la pile d'exécution (« stack overflow »).

```
# let rec term = function
        | 0 -> print_int 0
        | n -> print_int n; term (n - 1);;
term : int -> unit = <fun>

# term 10;;
109876543210- : unit = ()

# trace "term";;
The function term is now traced.
- : unit = ()

# term 5;;
term <-- 5
5term <-- 4
4term <-- 3
3term <-- 2
2term <-- 1
1term <-- 0
0term --> ()
term --> ()
term --> ()
term --> ()
term --> ()
term --> ()
- : unit = ()
```

En revanche, la fonction suivante n'est pas récursive terminale :

```
# let rec nonterm = function
        | 0 -> print_int 0
        | n -> nonterm (n - 1); print_int n;;
```

```
nonterm : int -> unit = <fun>

# trace "nonterm";;
The function nonterm is now traced.
- : unit = ()

# nonterm 5;;
nonterm <-- 5
nonterm <-- 4
nonterm <-- 3
nonterm <-- 2
nonterm <-- 1
nonterm <-- 0
0nonterm --> ()
1nonterm --> ()
2nonterm --> ()
3nonterm --> ()
4nonterm --> ()
5nonterm --> ()
- : unit = ()
```

Bien sûr, les fonctions fact et fibo définies ci-dessus ne sont pas récursives terminales.

5.3. De la récursivité non terminale à la récursivité terminale

Il y a toute légitimité à vouloir privilégier les fonctions récursives terminales, qui constituent souvent le juste milieu entre la clarté des fonctions récursives, et la rapidité supposée de la programmation impérative. Une idée générale pour tenter de programmer de manière récursive terminale est d'utiliser un *accumulateur*, qui joue le rôle de « mémoire » de notre programme.

Par exemple, pour la fonction factorielle, nous pouvons proposer :

```
# let rec fact_term = fun
        | (1, x) -> x
        | (n, x) -> fact_term (n - 1, x * n);;
fact_term : int * int -> int = <fun>

# trace "fact_term";;
The function fact_term is now traced.
- : unit = ()

# fact_term (5, 1);;
fact_term <-- 5, 1
fact_term <-- 4, 5
fact_term <-- 3, 20
fact_term <-- 2, 60
fact_term <-- 1, 120
fact_term --> 120
fact_term --> 120
fact_term --> 120
fact_term --> 120
fact_term --> 120
- : int = 120
```

Remarque : il ne faut pas être impressionné par les nombreux renvois, qui sont en réalité tous identiques.
Remarque : pour retrouver une fonction factorielle de type int -> int, il suffit de définir fact n comme fact (n, 1), par exemple localement :

```
# let fact =
      let rec fact_term = fun
              | (1, x) -> x
              | (n, x) -> fact_term (n - 1, x * n)
```

```
        in function n -> fact_term (n, 1);;
```

Remarque : on peut se demander pourquoi je n'ai pas employé une fonction curryfiée.

```
# let rec curry_fact_term = fun
        | 1 x -> x
        | n x -> curry_fact_term (n - 1) (x * n);;
curry_fact_term : int -> int -> int = <fun>

# trace "curry_fact_term";;
The function curry_fact_term is now traced.
- : unit = ()

# curry_fact_term 5 1;;
curry_fact_term <-- 5
curry_fact_term --> <fun>
curry_fact_term* <-- 1
curry_fact_term <-- 4
curry_fact_term --> <fun>
curry_fact_term* <-- 5
curry_fact_term <-- 3
curry_fact_term --> <fun>
curry_fact_term* <-- 20
curry_fact_term <-- 2
curry_fact_term --> <fun>
curry_fact_term* <-- 60
curry_fact_term <-- 1
curry_fact_term --> <fun>
curry_fact_term* <-- 120
curry_fact_term* --> 120
curry_fact_term* --> 120
curry_fact_term* --> 120
curry_fact_term* --> 120
curry_fact_term* --> 120
- : int = 120
```

Voici un exemple de programmation récursive terminale de la suite de Fibonacci :

```
# let rec fibo_term = fun
        | (0, x, y) -> x
        | (n, x, y) -> fibo_term ((n - 1), y, x + y);;
fibo_term : int * int * int -> int = <fun>

# trace "fibo_term";;
The function fibo_term is now traced.
- : unit = ()

# fibo_term (5, 1, 1);;
fibo_term <-- 5, 1, 1
fibo_term <-- 4, 1, 2
fibo_term <-- 3, 2, 3
fibo_term <-- 2, 3, 5
fibo_term <-- 1, 5, 8
fibo_term <-- 0, 8, 13
fibo_term --> 8
fibo_term --> 8
fibo_term --> 8
fibo_term --> 8
fibo_term --> 8
fibo_term --> 8
- : int = 8
```

Remarque : on donnera plus tard une méthode encore bien plus efficace grâce à la stratégie « diviser pour régner ».

6. Ensembles inductifs, induction structurelle

Définition (Ensemble inductif)

Soit E un ensemble, $\mathcal{B}$ une partie de E, $\mathcal{C}$ un ensemble d'applications $\Phi : E^{a(\Phi)} \to E$, où $a(\Phi)$ est l'arité de Φ.
On appelle *ensemble défini inductivement* par $\mathcal{B}$ et $\mathcal{C}$ la plus petite partie $\mathcal{T}$ de E (pour l'inclusion) contenant $\mathcal{B}$, et stable par tous les éléments de $\mathcal{C}$, dans le sens où pour tout $\Phi \in \mathcal{C}$, tout $(t_1, \dots, t_{a(\Phi)}) \in \mathcal{T}^{a(\Phi)}$, $\Phi(t_1, \dots, t_{a(\Phi)}) \in \mathcal{T}$.
De plus, les éléments de $\mathcal{B}$ sont les *éléments de base* de $\mathcal{T}$, et les éléments de $\mathcal{C}$ sont appelés *constructeurs* de $\mathcal{T}$.

6.a

On fixe pour la suite les notations de cette définition

Démonstration

Justification de l'existence de $\mathcal{T}$ (l'unicité étant claire).

□

Exemple (Ensembles définis inductivement)

Ensemble des entiers naturels, $2\mathbb{N}$.
Listes

i

Définition (Éléments de complexité au plus n)

On pose $\mathcal{T}_0 = \mathcal{B}$, et, pour tout $n \in \mathbb{N}$, $\mathcal{T}_{n+1} = \mathcal{T}_n \cup \{y \in E, \exists (\Phi, (t_k)) \in \mathcal{C} \times \mathcal{T}_n^{a(\Phi)}, y = \Phi(t_k)\}$.
Si $y \in \mathcal{T}_n$, on dit que y est de complexité au plus n.

6.b

Remarque : cette complexité ne dépend pas seulement de l'ensemble $\mathcal{T}$, mais surtout de la façon dont il est construit, *i.e.* dépend des constructeurs et des cas de base.

Théorème Description d'un ensemble inductif par niveaux de complexité

On a $\mathcal{T} = \bigcup_{n \in \mathbb{N}} \mathcal{T}_n$.

6.a

Démonstration

facile.

□

Définition (Complexité)

Pour tout $y \in \mathcal{T}$, on appelle *complexité* du terme y de l'ensemble inductif $\mathcal{T}$ comme $\min\{n \in \mathbb{N}, y \in \mathcal{T}_n\}$.

6.c

Théorème Induction structurelle

Soit p un prédicat sur E. On suppose

(1) $\forall\, b \in \mathcal{B}, p(b)$.

(2) Pour tout $(\Phi, (t_1, \ldots, t_{a(\Phi)})) \in \mathcal{C} \times \mathcal{T}^{a(\Phi)}$:

$$(\forall\, k \in [\![1, a(\Phi)]\!], p(t_k)) \Rightarrow p\left(\Phi(t_1, \ldots, t_{a(\Phi)})\right).$$

Le prédicat est alors vrai sur $\mathcal{T}$.

6.b

Démonstration

Il suffit de constater que l'ensemble Ω des éléments de E pour lesquels le prédicat est vrai contient $\mathcal{B}$ et est stable par application des éléments de $\mathcal{C}$.

□

Remarque : on peut appliquer ce résultat à la définition et égalité de fonction(s).
Remarque : lorsqu'on programme une fonction définie sur un ensemble inductif, il est très fréquent que la structure inductive dicte la programmation.

Troisième partie

Analyse des algorithmes

CHAPITRE V

Complexité

1. Généralités

1.1. Comment mesurer la performance d'un programme ?

Bien entendu, la première vertu d'un programme est sa correction, la suivante étant sa terminaison. Une fois ces propriétés satisfaites, il reste à savoir dans quelle mesure le programme proposé est efficace : c'est l'objet du domaine de l'informatique appelé *complexité*.

On distingue surtout deux types de complexité : la *complexité temporelle*, qui évalue la rapidité de l'algorithme, et la *complexité spatiale*, qui évalue l'occupation mémoire de l'algorithme.

Les progrès de l'informatique ont fait perdre de l'importance à la complexité spatiale : nous nous concentrerons surtout sur la complexité temporelle.

Bien entendu, il est hors de question d'évaluer concrètement cette complexité temporelle, en chronométrant un programme : cela n'aurait qu'une valeur empirique, non prédictive, dépendrait fortement du hardware sur lequel le programme tourne, dépendrait des données initiales (par exemple, il est facile de tester si $2^{43112609}$ est premier, ça l'est moins pour $2^{43112609}-1$). Il n'est pas non plus utile de donner précisément la complexité de l'algorithme, mais plutôt son ordre de grandeur.

Il nous faut donc trouver un cadre d'étude théorique pour évaluer la rapidité d'un programme.

1.2. Notion de tailles de données, classes de complexité

La plupart des algorithmes ont un argument entier (test de primalité, factorisation), plusieurs (algorithme d'Euclide, exponentiation) ou leur exécution dépend d'un entier naturel (taille d'une liste, d'un vecteur) : nous noterons n un entier, représentant la taille de données, dont les algorithmes dépendront. Pour un entier, ce peut être le nombre de bits.

Outre la notation de Landau O, nous utiliserons également la notation Θ, signifiant que deux suites ont même ordre : $u_n = \Theta(v_n)$ signifie $u_n = \mathrm{O}(v_n)$ et $v_n = \mathrm{O}(u_n)$.

Selon la taille de données n, l'algorithme va effectuer un certain nombre de tâches, dont certaines auront un poids bien plus grand dans le temps d'exécution. Nous ne compterons que le nombre c_n de ces opérations coûteuses.

On dit qu'un algorithme est

- *logarithmique* si c_n est de l'ordre de $\log_2(n)$.
- *linéaire* si c_n est de l'ordre de n.
- *quasi-linéaire* si c_n est de l'ordre de $n\log n$.
- *quadratique* si c_n est de l'ordre de n^2.
- *polynomial* si c_n est de l'ordre de n^k, pour un entier non nul k.
- *exponentiel* si c_n est de l'ordre de a^n, où $a>1$.

Ces classes de complexité sont données en ordre croissant : les algorithmes exponentiels sont très peu utiles, les logarithmiques finissent en temps raisonnable pour n'importe quelle taille de l'entrée.

Remarque : bien sûr, il faut tempérer ce jugement, puisqu'il peut y avoir des coûts occultes (si par exemple $c_n \sim 10^{100}\log_2(n)$, l'algorithme n'est pas si pratique que cela ...).

1.3. Un raffinement nécessaire

En réalité, la vitesse d'exécution d'un algorithme peut être très sensible à la valeur des données, pas seulement à la taille n de ces données (penser encore une fois à un test de primalité). Ceci nous conduit à introduire des complexités selon les cas : complexité dans le meilleur, le pire des cas, complexité en moyenne.

Dans ce dernier cas, il est important de préciser le contexte probabiliste, afin de pouvoir pondérer les différentes entrées possibles.

2. Un premier exemple

Nous prenons l'exemple de la recherche d'un élément dans un tableau :

```
# let cherche elt t = let i = ref 0 and trouve = ref false in
        while (!i < vect_length t  & !trouve = false) do
                if elt = t.(!i) then trouve := true else i := !i +1
                done;
        !trouve;;
cherche : 'a -> 'a vect -> bool = <fun>

# cherche 3  [| 5; 6; 9|];;
- : bool = false
```

Exercice : programmer par exception la recherche dans un tableau.

n est ici la taille du tableau, et nous allons compter le nombre de comparaisons elt = t.(!i) : dans le meilleur des cas, il n'y en a qu'une, dans le pire, il y en a n.

Supposons que *elt* apparaisse avec une probabilité q, que les éléments de t soient distincts, et que chaque position pour *elt* soit équiprobable.

Pour trouver l'élément en position i, on effectue i comparaisons. La complexité est donc $(1-q)n + q\frac{n+1}{2}$. Elle est donc linéaire, comme dans le pire des cas.

3. Diviser pour régner

3.1. Le principe

La méthode ou le paradigme « diviser pour régner » consiste, étant donné un problème de taille de données n, à :

(1) Diviser ce problème en q sous-problèmes de taille $n/2$ environ.

(2) Traiter ces q sous-problèmes.

(3) Fusionner ces q sous-problèmes

Le gain en complexité peut être énorme dans certains cas. Bien sûr, on applique ceci récursivement (on coupe chaque sous-problème en q sous-problèmes, etc.).

Remarque : cette méthode se rencontre souvent en informatique, et il faut apprendre à la mettre en œuvre et à la proposer spontanément.

Remarque : le plus souvent, $q = 2$.

Un premier exemple d'utilisation de ce paradigme pourrait être la recherche dichotomique (dans un tableau trié, d'un zéro d'une fonction), dont nous reparlerons lors du cours sur les tris. Nous allons voir d'autres exemples : l'exponentiation, la multiplication des polynômes, des grands entiers, des matrices. Nous rencontrerons également des algorithmes de tri efficaces fondés sur cette méthode.

3.2. Gain en termes de complexité

Pour évaluer le gain en termes de complexité, nous allons traiter le cas où $n = 2^m$, où le coût de la division et de la fusion est de $f(n)$, nous notons $C(n) = T(m)$ le coût temporel de l'algorithme.

Nous avons donc la relation de récurrence :

$$T(m+1) = qT(m) + f(2^m)$$

que l'on peut réécrire

$$\frac{T(m+1)}{q^{m+1}} - \frac{T(m)}{q^m} = \frac{f(2^m)}{q^{m+1}}$$

On obtient donc, en sommant :

$$\frac{T(m)}{q^m} = T(0) + \sum_{k=0}^{m} \frac{f(2^k)}{q^{k+1}}.$$

Remarque : observons que $q^m = n^{\log_2(q)}$: si on connaît déjà un algorithme en $\Theta(n^{\log_2(q)})$, le gain sera nul. Tout l'art consistera à minimiser q.

Premier cas : $\frac{f(2^m)}{q^m} = \mathrm{O}(\frac{1}{m^2})$, par exemple $f(n)$ est polynomial en m, ou en α^m, où $\alpha < q$. On montre alors que $T(m) \sim \lambda q^m = \lambda n^{\log_2(q)}$, de sorte que l'algorithme a une complexité au pire linéaire lorsque $q = 2$.

Deuxième cas : $f(n)$ a un coût en $n^{\log_2(q)}$. On montre alors que $C(n) = \Theta(n^{\log_2(q)} \log_2(n))$, de sorte que l'algorithme a une complexité quasi-linéaire lorsque $q = 2$.

Troisième cas : $f(n)$ a un coût en α^m, où $\alpha > q$. On montre alors que $T(m)$ a une complexité dominée par $q^m \left(\frac{\alpha}{q}\right)^{m+1}$, *i.e.* par $\alpha^m = n^{\log_2(\alpha)}$.

Remarque : le fait de prendre une puissance de deux ne nuit pas à la généralité du calcul de complexité, car un encadrement d'un autre entier entre deux puissances de deux permet de conclure.

3.3. L'exponentiation rapide

L'idée la plus évidente pour calculer les puissances d'un entier est la suivante :

```
# let puissance_imperative a b = let p = ref 1 in
        for i = 1 to b do p := !p * a done; !p;;
(* invariant après la i-ème itération : !p = a ^ i *)
puissance_imperative : int -> int -> int = <fun>

# let rec puissance a = function
        | 0 -> 1
        | b -> a * puissance a (b - 1);;
puissance : int -> int -> int = <fun>
```

Ces fonctions donnent bien le résultat voulu, mais leur complexité (comptée en nombre de multiplications) est linéaire en b. Le paradigme diviser pour régner permet de faire bien mieux :

```
# let rec puissance a b = match b with
        | 0 -> 1
        | _ -> let y = puissance a (b / 2) in
                if b mod 2 = 0 then y * y else y * y * a;;
puissance : int -> int -> int = <fun>

# puissance 2 10;;
- : int = 1024
```

De manière impérative, cela pourrait donner

```
# let puiss a b = let res = ref a and expo = ref b and reste = ref 1 in
        while !expo > 1 do
                if !expo mod 2 = 1 then reste := !reste * a;
                res := !res * !res;
                expo := !expo / 2
                done;
                (* Invariant : a^b = !res ^ !expo * !reste *)
        !res * !reste;;
puiss : int -> int -> int = <fun>

# puiss 3 5;;
- : int = 243
```

Dans les deux cas, on trouve une complexité logarithmique (en nombre de multiplications) : le gain est donc conséquent. Ce gain s'explique par le fait que le coût de la fusion vaut 1 ou 2, mais surtout que dans le paradigme diviser pour régner, on a ici $q = 1$!

3.4. L'algorithme de Knuth de multiplication des polynômes

On s'intéresse à la multiplication de polynômes à coefficients entiers, représentés par des tableaux d'entiers (l'indice correspond au degré du monôme associé).

On cherche à multiplier deux polynômes $A = \sum_{k=0}^{p} a_k X^k$ et $B = \sum_{k=0}^{q} b_k X^k$. On sait que :

$$AB = \sum_{i=0}^{p} \sum_{j=0}^{q} a_i b_j X^{i+j},$$

ce qui permet de proposer une première fonction réalisant ce produit :

```
# let produit_poly_simple a b = let p = vect_length a and q = vect_length b in
        let res = make_vect (p + q - 1)  0 in
                for i = 0 to p - 1 do
                        for j = 0 to q - 1 do
                                res.(i + j) <- res.(i + j) + a.(i) * b.(j)
                        done
                done;
        res;;
produit_poly_simple : int vect -> int vect -> int vect = <fun>

# produit_poly_simple [|1; 1; 1|] [|1; 1; 1|];;
- : int vect = [|1; 2; 3; 2; 1|]
```

Cependant, la complexité est clairement de l'ordre de pq, donc au pire quadratique dans le cas de polynômes de même degré.

Knuth a proposé un algorithme de meilleure complexité, en utilisant l'astuce de Karatsuba : on suppose les deux polynômes de degré au plus $2n-1$. On écrit $P = P_0 + X^n P_1$ et $Q = Q_0 + X^n Q_1$ (on a effectué les divisions euclidiennes de P et de Q par X^n). On a donc

$$PQ = P_0Q_0 + X^n(P_0Q_1 + P_1Q_0) + X^{2n}P_1Q_1.$$

Ceci donne quatre multiplications, ce qui n'apporte rien en termes de complexité. L'astuce de Karatsuba consiste cependant à remarquer que les trois produits P_0Q_0, P_1Q_1, et $(P_0 + P_1)(Q_0 + Q_1)$ suffisent à calculer ces quatre produits (au prix d'additions supplémentaires, de coût négligeable par rapport à une multiplication). La complexité (comptée en nombre de multiplications) est de l'ordre de $n^{\log_2(3)} \simeq n^{1.58}$.

Remarque : cet algorithme est utile pour effectuer la multiplication de grands entiers en base b (représentés par des vecteurs d'entiers entre 0 et $b-1$).

3.5. L'algorithme de Strassen de multiplication des matrices

Pour multiplier deux matrices carrées de taille n, la méthode standard, revenant à la définition d'un tel produit, est de complexité en n^3 (on compte encore une fois seulement les multiplications).

On peut mieux faire, en effectuant un produit matriciel par blocs. On cherche à multiplier deux matrices carrées de taille paire n

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \quad \text{et} \quad B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

Les formules de produit par bloc donnent

$$AB = \begin{pmatrix} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{pmatrix},$$

ce qui semble nécessiter huit multiplications, ce qui serait sans grand intérêt. L'idée de Strassen, un peu similaire à l'astuce de Karatsuba, est de réduire le nombre de multiplications à sept. On note :

$$P_1 = A_1(B_2 - B_4), P_2 = (A_1 + A_2)B_4, P_3 = (A_3 + A_4)B_1, P_4 = A_4(B_3 - B_1),$$
$$P_5 = (A_1 + A_4)(B_1 + B_4), P_6 = (A_2 - A_4)(B_3 + B_4), P_7 = (A_1 - A_3)(B_1 + B_2)$$

On a

$$AB = \begin{pmatrix} -P_2 + P_4 + P_5 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 - P_3 + P_5 - P_7 \end{pmatrix}.$$

On obtient donc une complexité en $n^{\log_2(7)}$.

```
# let puissance_imperative a b = let p = ref 1 in
        for i = 1 to b do p := !p * a done; !p;;
(* invariant après la i-ème itération : !p = a ^ i *)
puissance_imperative : int -> int -> int = <fun>

# let rec puissance a = function
        | 0 -> 1
        | b -> a * puissance a (b - 1);;
puissance : int -> int -> int = <fun>

# let rec puissance a b = match b with
        | 0 -> 1
        | _ -> let y = puissance a (b / 2) in
                if b mod 2 = 0 then y * y else y * y * a;;
puissance : int -> int -> int = <fun>

# puissance 2 10;;
- : int = 1024

# let puiss a b = let res = ref a and expo = ref b and reste = ref 1 in
        while !expo > 1 do
                if !expo mod 2 = 1 then reste := !reste * a;
                res := !res * !res;
                expo := !expo / 2
                done;
                (* Invariant : a^b = !res ^ !expo * !reste *)
        !res * !reste;;
puiss : int -> int -> int = <fun>

# puiss 3 5;;
- : int = 243

# let produit_poly_simple a b = let p = vect_length a and q = vect_length b in
        let res = make_vect (p + q - 1)  0 in
                for i = 0 to p - 1 do
                        for j = 0 to q - 1 do
                                res.(i + j) <- res.(i + j) + a.(i) * b.(j)
                        done
                done;
        res;;
produit_poly_simple : int vect -> int vect -> int vect = <fun>

# produit_poly_simple [|1; 1; 1|] [|1; 1; 1|];;
- : int vect = [|1; 2; 3; 2; 1|]
```

CHAPITRE VI

Tris

1. Introduction

Le tri est à la fois l'un des plus anciens et courants problèmes d'informatique. De nombreux algorithmes ont été proposés afin d'effectuer le tri d'un tableau ou d'une liste d'éléments d'un ensemble totalement ordonné.

La complexité d'un algorithme de tri s'effectue en prenant pour taille de données la taille de la liste ou du vecteur à trier, et en comptant le nombre d'échanges ou de comparaisons effectuées (on pourra privilégier, selon le but cherché, l'un ou l'autre). On s'intéressera également à l'espace employé pour l'application de tel ou tel algorithme (un tri sur place étant un tri n'employant qu'un espace constant), et, éventuellement, à la stabilité d'un tel algorithme, *i.e.* s'il lui arrive d'échanger des valeurs égales en des indices distincts.

La structure la plus naturellement adaptée aux tris est celle de vecteur, puisque l'accès aux éléments d'une telle structure s'effectue en temps constant, contrairement au cas des listes.

Nous allons motiver le tri par la recherche dichotomique, se poser le problème de la fusion de deux tableaux ou de deux listes triées, puis proposer de nombreux algorithmes de tri.

2. Utilité du tri : recherche dans un tableau trié

Nous avons déjà vu un algorithme linéaire de recherche dans un tableau. Cependant, si le tableau est trié, une recherche dichotomique (cas particulier de la stratégie diviser pour régner) permet d'effectuer cette recherche en temps logarithmique :

```
# let rec cherche_dicho t (elt : int) = let n = vect_length t in match n with
        | 0 -> false
        | 1 -> if t.(0) = elt then true else false
        | _ -> let n' = n / 2 in
                 if t.(n') <= elt then cherche_dicho (sub_vect t n' (n - n')) elt
                              else cherche_dicho (sub_vect t 0 n') elt;;
cherche_dicho : int vect -> int -> bool = <fun>

# trace "cherche_dicho";;
The function cherche_dicho is now traced.
- : unit = ()

# cherche_dicho [|1; 3; 7; 9; 12; 14|] 12;;
cherche_dicho <-- [|1; 3; 7; 9; 12; 14|]
cherche_dicho --> <fun>
cherche_dicho* <-- 12
cherche_dicho <-- [|9; 12; 14|]
cherche_dicho --> <fun>
cherche_dicho* <-- 12
cherche_dicho <-- [|12; 14|]
cherche_dicho --> <fun>
cherche_dicho* <-- 12
cherche_dicho <-- [|12|]
cherche_dicho --> <fun>
cherche_dicho* <-- 12
cherche_dicho* --> true
cherche_dicho* --> true
cherche_dicho* --> true
cherche_dicho* --> true
- : bool = true
```

Remarque : dans le cas d'une liste chaînée, la recherche dichotomique n'est pas adaptée, puisque l'accès à un élément ne s'effectue pas en temps constant, et que la longueur de la liste n'est pas connue.

3. Insertion dans une liste triée

Comment insérer un élément dans un tableau ou une liste déjà trié ?

```
# let insertion t elt = let n = vect_length t in let t' = make_vect (n + 1) elt in
        let i = ref 0 in (* on place les éléments inférieurs à elt *)
                while !i < n & t.(!i) < elt do
                        t'.(!i) <- t.(!i); i := !i + 1
                        done;
        let j = ref (n - 1) in (* on place les éléments supérieurs à elt *)
                while !j >= 0 & t.(!j) > elt  do
                        t'.(!j + 1) <- t.(!j); j := !j - 1
                        done;
        t';;
insertion : 'a vect -> 'a -> 'a vect = <fun>

# insertion [|3; 4; 9|] 7;;
- : int vect = [|3; 4; 7; 9|]
```

On peut aussi effectuer une insertion pour les listes

```
# let rec insertion_liste l elt = match l with
        | [] -> [elt]
        | a :: q -> if a < elt then a :: insertion_liste q elt else elt :: l;;
insertion_liste : 'a list -> 'a -> 'a list = <fun>

# insertion_liste [3; 4; 9] 7;;
- : int list = [3; 4; 7; 9]
```

Remarque : on effectue $n/2$ comparaisons en moyenne.

4. Fusion de deux listes triées

Le problème de la fusion de deux tableaux triés (ou listes triées), qui généralise en quelque sorte celui de l'insertion, se pose naturellement. Comment programmer ?

Pour les tableaux, on peut proposer

```
# let fusion t t' = let n = vect_length t and n' = vect_length t' in
        let res = make_vect (n + n') 0 (* résultat *) in
        let j = ref 0 and j' = ref 0 in
                for i = 0 to n + n' - 1 do
                (* après le passage pour l'indice i, les termes du résultat,
                d'indices 0 à i, sont bien placés dans res *)
                        if !j < n (* le tableau t n'est pas épuisé *)
                        then
                                if !j' < n' (* le tableau t' n'est pas épuisé *)
                                then
                                        if t.(!j) <= t'.(!j') then
                                                (res.(i) <- t.(!j);
                                                j := !j + 1;)
                                        else
                                                (res.(i) <- t'.(!j');
                                                j' := !j' + 1;)
                                else (* le tableau t' est épuisé *)
                                (res.(i) <- t.(!j); j := !j +1;)
                        else (* le tableau t est épuisé *)
                        (res.(i) <- t'.(!j'); j' := !j' + 1;)
                                done;
```

```
        res;;
fusion : int vect -> int vect -> int vect = <fun>

# fusion [|1; 2; 5; 15; 20|] [|0; 2; 6; 9; 11|];;
- : int vect = [|0; 1; 2; 2; 5; 6; 9; 11; 15; 20|]
```

La complexité est bien sûr en $n + n'$, donc linéaire.

Remarque : on pouvait bien sûr effectuer une succession d'insertions.

Pour les listes, cela pourrait donner :

```
# let rec fusion_listes l l' = match (l,l') with
        | (_,[]) -> l
        | ([],_) -> l'
        | (a :: q, a' :: q') -> if a <= a' then a :: fusion_listes q l'
                                           else a' :: fusion_listes l q';;
fusion_listes : 'a list -> 'a list -> 'a list = <fun>

# fusion_listes [1; 2; 5; 15; 20] [0; 2; 6; 9; 11];;
- : int list = [0; 1; 2; 2; 5; 6; 9; 11; 15; 20]
```

5. Exemples de tris

5.1. Le tri par insertion

Le tri par insertion, celui du joueur de cartes, consiste à insérer un par un chaque terme, tout en conservant l'ordre.

On peut le programmer ainsi :

```
# let tri_insertion t = let t' = ref [||] in
        for i = 0 to  vect_length t - 1 do t' := insertion !t' t.(i) done;
        !t';;
tri_insertion : 'a vect -> 'a vect = <fun>

# tri_insertion [|1; 3; -2; -1; 5; -3; 6|];;
- : int vect = [|-3; -2; -1; 1; 3; 5; 6|]

# let rec tri_insertion_liste = function
        | [] -> []
        | a :: q -> insertion_liste (tri_insertion_liste q) a;;
tri_insertion_liste : 'a list -> 'a list = <fun>

# tri_insertion_liste [1; 3; -2; -1; 5; -3; 6];;
- : int list = [-3; -2; -1; 1; 3; 5; 6]
```

Remarque : cette programmation « à la lettre » du tri par insertion n'est pas optimale. On peut faire bien mieux en programmant par effets de bord.

Remarque : le coût d'une insertion étant linéaire, et puisque nous en faisons n, le coût du tri par insertion est quadratique, ce qui sera le cas de la plupart des tris simples.

Remarque : le tri par insertion est très efficace si le tableau ou le vecteur est presque trié.

5.2. Le tri par sélection

Dans le tri par sélection, on procède de la manière suivante : on cherche d'abord le plus petit élément, que l'on place dans le résultat, puis le suivant, que l'on range en deuxième position, etc.

```
# let echange t i j = let c = t.(i) in t.(i) <- t.(j); t.(j) <- c;;
echange : 'a vect -> int -> int -> unit = <fun>

# let tri_selection t = let n = vect_length t in
        (* programmation par effets de bord *)
        for i = 0 to n - 2 do
        (* après le passage en boucle pour l'indice i, t (0 .. i)
```

```
        est constitué des (i + 1) plus petits termes de t ordonnés *)
                let temp = ref i in
                        for j = i + 1 to n - 1 do
                                if t.(j) < t.(!temp) then temp := j;
                                done;
                        echange t i !temp;
                done;;
tri_selection : 'a vect -> unit = <fun>

# let tab = [|-3; 1; -1; 16; 7; 12|];;
tab : int vect = [|-3; 1; -1; 16; 7; 12|]

# tri_selection tab;;
- : unit = ()

# tab;;
- : int vect = [|-3; -1; 1; 7; 12; 16|]
```

Cet algorithme est encore quadratique pour les comparaisons, mais linéaire pour les échanges.
Voici une version pour les listes :

```
# let rec isole_min = function
        | [] -> failwith "pas de minimum pour la liste vide"
        | [a] -> (a,[])
        | a :: q -> let l' = isole_min q in
                if a <= fst l' then (a,q) else (fst l', a :: snd l');;
isole_min : 'a list -> 'a * 'a list = <fun>

# isole_min [3; 6; -2; 7; 9];;
- : int * int list = -2, [3; 6; 7; 9]

# let rec tri_selection_liste = function
        | [] -> []
        | l -> let l' = isole_min l in
                (fst l') :: (tri_selection_liste (snd l'));;
tri_selection_liste : 'a list -> 'a list = <fun>

# tri_selection_liste [6; 3; -2; 9; 1; 10; 3];;
- : int list = [-2; 1; 3; 3; 6; 9; 10]
```

5.3. Le tri bulle

Le tri bulle est une variante du tri par sélection : on parcourt le tableau en échangeant deux éléments consécutifs mal placés.

```
# let tri_bulle t = let n = vect_length t in
        for i = 0 to (n - 1) do
                for j = 0 to (n - 2) do
                        if t.(j) > t.(j + 1) then echange t j (j + 1)
                        done;
                done;
        t;;
tri_bulle : 'a vect -> 'a vect = <fun>

(* Non optimal, car le tri bulle peut s'arrêter
dès qu'il ne modifie plus rien en une passe *)

# tri_bulle [|3; 5; 1; 6; 3|];;
- : int vect = [|1; 3; 3; 5; 6|]
```

Remarque : le tri bulle a peu d'intérêt, puisqu'il est quadratique en comparaisons et en échanges.

```
(* Recherche dichotomique pour les vecteurs *)

# let rec cherche_dicho t (elt : int) = let n = vect_length t in match n with
        | 0 -> false
        | 1 -> if t.(0) = elt then true else false
        | _ -> let n' = n / 2 in
                 if t.(n') <= elt then cherche_dicho (sub_vect t n' (n - n')) elt
                                  else cherche_dicho (sub_vect t 0 n') elt;;
cherche_dicho : int vect -> int -> bool = <fun>

# trace "cherche_dicho";;
The function cherche_dicho is now traced.
- : unit = ()

# cherche_dicho [|1; 3; 7; 9; 12; 14|] 12;;
cherche_dicho <-- [|1; 3; 7; 9; 12; 14|]
cherche_dicho --> <fun>
cherche_dicho* <-- 12
cherche_dicho <-- [|9; 12; 14|]
cherche_dicho --> <fun>
cherche_dicho* <-- 12
cherche_dicho <-- [|12; 14|]
cherche_dicho --> <fun>
cherche_dicho* <-- 12
cherche_dicho <-- [|12|]
cherche_dicho --> <fun>
cherche_dicho* <-- 12
cherche_dicho* --> true
cherche_dicho* --> true
cherche_dicho* --> true
cherche_dicho* --> true
- : bool = true
```

```
(* Insertion d'un élément dans un vecteur trié *)

# let insertion t elt = let n = vect_length t in let t' = make_vect (n + 1) elt in
        let i = ref 0 in (* on place les éléments inférieurs à elt *)
                while !i < n & t.(!i) < elt do
                        t'.(!i) <- t.(!i); i := !i + 1
                        done;
        let j = ref (n - 1) in (* on place les éléments supérieurs à elt *)
                while !j >= 0 & t.(!j) > elt  do
                        t'.(!j + 1) <- t.(!j); j := !j - 1
                        done;
        t';;
insertion : 'a vect -> 'a -> 'a vect = <fun>

# insertion [|3; 4; 9|] 7;;
- : int vect = [|3; 4; 7; 9|]



(* Fusion de vecteurs triés *)

# let fusion t t' = let n = vect_length t and n' = vect_length t' in
        let res = make_vect (n + n') 0 (* résultat *) in
        let j = ref 0 and j' = ref 0 in
                for i = 0 to n + n' - 1 do
                (* après le passage pour l'indice i, les termes du résultat,
                d'indices 0 à i, sont bien placés dans res *)
                        if !j < n (* le tableau t n'est pas épuisé *)
                        then
                                if !j' < n' (* le tableau t' n'est pas épuisé *)
                                then
                                        if t.(!j) <= t'.(!j') then
                                                (res.(i) <- t.(!j);
                                                j := !j + 1;)
                                        else
                                                (res.(i) <- t'.(!j');
                                                j' := !j' + 1;)
                                else (* le tableau t' est épuisé *)
                                (res.(i) <- t.(!j); j := !j +1;)
                        else (* le tableau t est épuisé *)
                        (res.(i) <- t'.(!j'); j' := !j' + 1;)
                                done;
        res;;
fusion : int vect -> int vect -> int vect = <fun>

# fusion [|1; 2; 5; 15; 20|] [|0; 2; 6; 9; 11|];;
- : int vect = [|0; 1; 2; 2; 5; 6; 9; 11; 15; 20|]
```

```
(* Tri par insertion "à la lettre" pour les vecteurs *)

# let tri_insertion t = let t' = ref [||] in
        for i = 0 to  vect_length t - 1 do t' := insertion !t' t.(i) done;
        !t';;
tri_insertion : 'a vect -> 'a vect = <fun>

# tri_insertion [|1; 3; -2; -1; 5; -3; 6|];;
- : int vect = [|-3; -2; -1; 1; 3; 5; 6|]


(* Tri par sélection pour les vecteurs *)

# let echange t i j = let c = t.(i) in t.(i) <- t.(j); t.(j) <- c;;
echange : 'a vect -> int -> int -> unit = <fun>

# let tri_selection t = let n = vect_length t in
        (* programmation par effets de bord *)
        for i = 0 to n - 2 do
        (* après le passage en boucle pour l'indice i, t (0 .. i)
        est constitué des (i + 1) plus petits termes de t ordonnés *)
                let temp = ref i in
                        for j = i + 1 to n - 1 do
                                if t.(j) < t.(!temp) then temp := j;
                                done;
                        echange t i !temp;
                done;;
tri_selection : 'a vect -> unit = <fun>

# let tab = [|-3; 1; -1; 16; 7; 12|];;
tab : int vect = [|-3; 1; -1; 16; 7; 12|]

# tri_selection tab;;
- : unit = ()

# tab;;
- : int vect = [|-3; -1; 1; 7; 12; 16|]


(* Tri bulle pour les vecteurs *)

# let tri_bulle t = let n = vect_length t in
        for i = 0 to (n - 1) do
                for j = 0 to (n - 2) do
                        if t.(j) > t.(j + 1) then echange t j (j + 1)
                        done;
                done;
        t;;
tri_bulle : 'a vect -> 'a vect = <fun>

# tri_bulle [|3; 5; 1; 6; 3|];;
- : int vect = [|1; 3; 3; 5; 6|]
```

Nous présentons maintenant deux algorithmes de tri évolués, fondés sur la stratégie diviser pour régner (dans lesquels la division et la fusion n'ont pas le même poids).

5.4. Le tri fusion

L'idée consiste à scinder la liste en deux listes de même taille (ou à peu près), à trier chacune de ces deux sous-listes, et à les fusionner (ceci de manière récursive bien entendu).

```
(* Tri fusion (on rappelle que la fusion a déjà été programmée) *)

# let rec tri_fusion t = let n = vect_length t in match n with
        | 0 | 1 -> t
        | _ -> let n' = n / 2 in
               fusion (tri_fusion (sub_vect t 0 n')) (tri_fusion (sub_vect t n' (n - n')));;
tri_fusion : int vect -> int vect = <fun>

# trace "tri_fusion";;
The function tri_fusion is now traced.
- : unit = ()

# tri_fusion [|1; 3; 5; 1; 7; 4; 9; -1|];;
tri_fusion <-- [|1; 3; 5; 1; 7; 4; 9; -1|]
tri_fusion <-- [|7; 4; 9; -1|]
tri_fusion <-- [|9; -1|]
tri_fusion <-- [|-1|]
tri_fusion --> [|-1|]
tri_fusion <-- [|9|]
tri_fusion --> [|9|]
tri_fusion --> [|-1; 9|]
tri_fusion <-- [|7; 4|]
tri_fusion <-- [|4|]
tri_fusion --> [|4|]
tri_fusion <-- [|7|]
tri_fusion --> [|7|]
tri_fusion --> [|4; 7|]
tri_fusion --> [|-1; 4; 7; 9|]
tri_fusion <-- [|1; 3; 5; 1|]
tri_fusion <-- [|5; 1|]
tri_fusion <-- [|1|]
tri_fusion --> [|1|]
tri_fusion <-- [|5|]
tri_fusion --> [|5|]
tri_fusion --> [|1; 5|]
tri_fusion <-- [|1; 3|]
tri_fusion <-- [|3|]
tri_fusion --> [|3|]
tri_fusion <-- [|1|]
tri_fusion --> [|1|]
tri_fusion --> [|1; 3|]
tri_fusion --> [|1; 1; 3; 5|]
tri_fusion --> [|-1; 1; 1; 3; 4; 5; 7; 9|]
- : int vect = [|-1; 1; 1; 3; 4; 5; 7; 9|]
```

Le tri fusion est de complexité (temporelle) quasi-linéaire, puisque fondé sur la stratégie diviser pour régner standard, que la division a un coût nul, et que la fusion a un coût linéaire en termes de comparaisons.
Remarque : malheureusement, le tri fusion prend beaucoup de place en mémoire, à cause de ses nombreux appels récursifs.

5.5. Le tri rapide

L'idée du tri rapide (ou de Hoare, ou par segmentation) consiste à choisir un élément x de la liste (souvent en tête), appelé pivot, à scinder ce qu'il reste de la liste en deux sous-listes, selon leur relation à x.

```
(* Tri rapide *)

(* La fonction pivot compare les éléments de t à son  terme initial,
les répartit dans un tableau, et précise en outre
l'indice où se trouve désormais le pivot *)

# let pivot t ind = let a = t.(ind) and n = vect_length t in
        let j = ref 1 and k = ref (n - 1) and res = make_vect n 0 in
        res.(0) <- a;
        (* on insère les éléments de t dans res selon leur relation à a *)
        for i = 0 to n - 1 do
                if i <> ind then if t.(i) < a then (res.(!j) <- t.(i); j := !j + 1;)
                else (res.(!k) <- t.(i); k:= !k - 1;)
                done;
        echange res 0 !k; (* on replace a à sa place *)
        res,!k;;
pivot : int vect -> int -> int vect * int = <fun>

# pivot [|3; 5; 6; 1; 2; 4; 9|] 3;;
- : int vect * int = [|1; 9; 4; 2; 6; 5; 3|], 0

(* Fonction fusion_triple pour améliorer la lisibilité *)

# let fusion_triple t t' t'' = fusion (fusion t t') t'';;
fusion_triple : int vect -> int vect -> int vect -> int vect = <fun>

(* Fonction tri_rapide suivant l'algorithme à la lettre *)

# let rec tri_rapide t = let n = vect_length t in
        if n <= 1 then t
        else
        let tab,i = pivot t 0 in match i with
                | 0 -> fusion [|tab.(0)|] (tri_rapide (sub_vect tab 1 (n - 1)))
                | k when k = n - 1 -> fusion (* cas de filtrage n - 1 non valide *)
                                (tri_rapide (sub_vect tab 0 (n - 1)))
                                [|tab.(n - 1)|]
                | _ -> fusion_triple
                                (tri_rapide (sub_vect tab 0 i))
                                [|tab.(i)|]
                                (tri_rapide (sub_vect tab (i + 1) (n - i - 1)));;
tri_rapide : int vect -> int vect = <fun>

# trace "tri_rapide";;
The function tri_rapide is now traced.
- : unit = ()

# tri_rapide [|4; 2; 1; 8; 4; 7; 10; 11; -3; 8; 12|];;
tri_rapide <-- [|4; 2; 1; 8; 4; 7; 10; 11; -3; 8; 12|]
tri_rapide <-- [|12; 8; 11; 10; 7; 4; 8|]
tri_rapide <-- [|8; 8; 11; 10; 7; 4|]
tri_rapide <-- [|10; 11; 8|]
tri_rapide <-- [|11|]
tri_rapide --> [|11|]
tri_rapide <-- [|8|]
```

```
tri_rapide --> [|8|]
tri_rapide --> [|8; 10; 11|]
tri_rapide <-- [|4; 7|]
tri_rapide <-- [|7|]
tri_rapide --> [|7|]
tri_rapide --> [|4; 7|]
tri_rapide --> [|4; 7; 8; 8; 10; 11|]
tri_rapide --> [|4; 7; 8; 8; 10; 11; 12|]
tri_rapide <-- [|-3; 2; 1|]
tri_rapide <-- [|1; 2|]
tri_rapide <-- [|2|]
tri_rapide --> [|2|]
tri_rapide --> [|1; 2|]
tri_rapide --> [|-3; 1; 2|]
tri_rapide --> [|-3; 1; 2; 4; 4; 7; 8; 8; 10; 11; 12|]
- : int vect = [|-3; 1; 2; 4; 4; 7; 8; 8; 10; 11; 12|]
```

Dans le pire des cas, l'un des tableaux scindés est vide, ce qui conduit à une complexité quadratique en termes de comparaisons.

Dans le cas où chaque tableau scindé est de taille moitié, on se retrouve avec une stratégie diviser pour régner classique, et on trouve une complexité en $n\log_2(n)$. En effet, la division a un coût linéaire, et la fusion un coût nul en termes de comparaisons.

Remarque : en réalité, cette complexité quasi-linéaire est optimale pour un algorithme de tri générique fondé sur les comparaisons.

Notons c_n le nombre moyen de comparaisons effectuées dans le tri rapide pour la taille de données n : on a

$$\begin{aligned} nc_n &= n(n-1) + \sum_{k=0}^{n-1}(c_k + c_{n-1-k}) \\ &= n(n-1) + 2\sum_{k=0}^{n-1} c_k \\ &= n(n-1) + 2c_{n-1} + (n-1)c_{n-1} - (n-1)(n-2) \\ &= (n+1)c_{n-1} + 2(n-1) \end{aligned}$$

d'où

$$\frac{c_n}{n+1} - \frac{c_{n-1}}{n} \sim \frac{2}{n},$$

puis c_n est quasi-linéaire.

```
# let rec partition l (pivot : int) = match l with
        | [] -> [],[]
        | a :: q -> let l,l' = partition q pivot in
                    if a < pivot then a :: l,l' else l, a :: l';;
partition : int list -> int -> int list * int list = <fun>

# let rec tri_rapide_listes = function
        | [] -> []
        | [a] -> [a] (* facultatif, mais simplifie la trace *)
        | a :: q -> let l,l' = partition q a in
                    tri_rapide_listes l @ [a] @ tri_rapide_listes l';;
tri_rapide_listes : int list -> int list = <fun>

# trace "tri_rapide_listes";;
The function tri_rapide_listes is now traced.
- : unit = ()

# tri_rapide_listes [-3; 5; -2; 9; 3; -1; -2; 3; 12];;
tri_rapide_listes <-- [-3; 5; -2; 9; 3; -1; -2; 3; 12]
tri_rapide_listes <-- [5; -2; 9; 3; -1; -2; 3; 12]
tri_rapide_listes <-- [9; 12]
tri_rapide_listes <-- [12]
```

```
tri_rapide_listes --> [12]
tri_rapide_listes <-- []
tri_rapide_listes --> []
tri_rapide_listes --> [9; 12]
tri_rapide_listes <-- [-2; 3; -1; -2; 3]
tri_rapide_listes <-- [3; -1; -2; 3]
tri_rapide_listes <-- [3]
tri_rapide_listes --> [3]
tri_rapide_listes <-- [-1; -2]
tri_rapide_listes <-- []
tri_rapide_listes --> []
tri_rapide_listes <-- [-2]
tri_rapide_listes --> [-2]
tri_rapide_listes --> [-2; -1]
tri_rapide_listes --> [-2; -1; 3; 3]
tri_rapide_listes <-- []
tri_rapide_listes --> []
tri_rapide_listes --> [-2; -2; -1; 3; 3]
tri_rapide_listes --> [-2; -2; -1; 3; 3; 5; 9; 12]
tri_rapide_listes <-- []
tri_rapide_listes --> []
tri_rapide_listes --> [-3; -2; -2; -1; 3; 3; 5; 9; 12]
- : int list = [-3; -2; -2; -1; 3; 3; 5; 9; 12]
```

```
(* Tri fusion (on rappelle que la fusion a déjà été programmée) *)

# let rec tri_fusion t = let n = vect_length t in match n with
        | 0 | 1 -> t
        | _ -> let n' = n / 2 in
                fusion (tri_fusion (sub_vect t 0 n')) (tri_fusion (sub_vect t n' (n - n')));
tri_fusion : int vect -> int vect = <fun>

# trace "tri_fusion";;
The function tri_fusion is now traced.
- : unit = ()

# tri_fusion [|1; 3; 5; 1; 7; 4; 9; -1|];;
tri_fusion <-- [|1; 3; 5; 1; 7; 4; 9; -1|]
tri_fusion <-- [|7; 4; 9; -1|]
tri_fusion <-- [|9; -1|]
tri_fusion <-- [|-1|]
tri_fusion --> [|-1|]
tri_fusion <-- [|9|]
tri_fusion --> [|9|]
tri_fusion --> [|-1; 9|]
tri_fusion <-- [|7; 4|]
tri_fusion <-- [|4|]
tri_fusion --> [|4|]
tri_fusion <-- [|7|]
tri_fusion --> [|7|]
tri_fusion --> [|4; 7|]
tri_fusion --> [|-1; 4; 7; 9|]
tri_fusion <-- [|1; 3; 5; 1|]
tri_fusion <-- [|5; 1|]
tri_fusion <-- [|1|]
tri_fusion --> [|1|]
tri_fusion <-- [|5|]
tri_fusion --> [|5|]
tri_fusion --> [|1; 5|]
tri_fusion <-- [|1; 3|]
tri_fusion <-- [|3|]
tri_fusion --> [|3|]
tri_fusion <-- [|1|]
tri_fusion --> [|1|]
tri_fusion --> [|1; 3|]
tri_fusion --> [|1; 1; 3; 5|]
tri_fusion --> [|-1; 1; 1; 3; 4; 5; 7; 9|]
- : int vect = [|-1; 1; 1; 3; 4; 5; 7; 9|]
```

```
(* Tri rapide *)

(* La fonction pivot compare les éléments de t à son  terme initial,
les répartit dans un tableau, et précise en outre
l'indice où se trouve désormais le pivot *)

# let pivot t ind = let a = t.(ind) and n = vect_length t in
        let j = ref 1 and k = ref (n - 1) and res = make_vect n 0 in
        res.(0) <- a;
        (* on insère les éléments de t dans res selon leur relation à a *)
        for i = 0 to n - 1 do
                if i <> ind then if t.(i) < a then (res.(!j) <- t.(i); j := !j + 1;)
                else (res.(!k) <- t.(i); k:= !k - 1;)
                done;
        echange res 0 !k; (* on replace a à sa place *)
        res,!k;;
pivot : int vect -> int -> int vect * int = <fun>

# pivot [|3; 5; 6; 1; 2; 4; 9|] 3;;
- : int vect * int = [|1; 9; 4; 2; 6; 5; 3|], 0

(* Fonction fusion_triple pour améliorer la lisibilité *)

# let fusion_triple t t' t'' = fusion (fusion t t') t'';;
fusion_triple : int vect -> int vect -> int vect -> int vect = <fun>
```

```
(* Fonction tri_rapide suivant l'algorithme à la lettre *)

# let rec tri_rapide t = let n = vect_length t in
        if n <= 1 then t
        else
        let tab,i = pivot t 0 in match i with
                | 0 -> fusion [|tab.(0)|] (tri_rapide (sub_vect tab 1 (n - 1)))
                | k when k = n - 1 -> fusion (* cas de filtrage n - 1 non valide *)
                                (tri_rapide (sub_vect tab 0 (n - 1)))
                                [|tab.(n - 1)|]
                | _ -> fusion_triple
                                (tri_rapide (sub_vect tab 0 i))
                                [|tab.(i)|]
                                (tri_rapide (sub_vect tab (i + 1) (n - i - 1)));;
tri_rapide : int vect -> int vect = <fun>

# trace "tri_rapide";;
The function tri_rapide is now traced.
- : unit = ()

# tri_rapide [|4; 2; 1; 8; 4; 7; 10; 11; -3; 8; 12|];;
tri_rapide <-- [|4; 2; 1; 8; 4; 7; 10; 11; -3; 8; 12|]
tri_rapide <-- [|12; 8; 11; 10; 7; 4; 8|]
tri_rapide <-- [|8; 8; 11; 10; 7; 4|]
tri_rapide <-- [|10; 11; 8|]
tri_rapide <-- [|11|]
tri_rapide --> [|11|]
tri_rapide <-- [|8|]
tri_rapide --> [|8|]
tri_rapide --> [|8; 10; 11|]
tri_rapide <-- [|4; 7|]
tri_rapide <-- [|7|]
tri_rapide --> [|7|]
tri_rapide --> [|4; 7|]
tri_rapide --> [|4; 7; 8; 8; 10; 11|]
tri_rapide --> [|4; 7; 8; 8; 10; 11; 12|]
tri_rapide <-- [|-3; 2; 1|]
tri_rapide <-- [|1; 2|]
tri_rapide <-- [|2|]
tri_rapide --> [|2|]
tri_rapide --> [|1; 2|]
tri_rapide --> [|-3; 1; 2|]
tri_rapide --> [|-3; 1; 2; 4; 4; 7; 8; 8; 10; 11; 12|]
- : int vect = [|-3; 1; 2; 4; 4; 7; 8; 8; 10; 11; 12|]
```

Quatrième partie

Structures de données et algorithmes

CHAPITRE VII

Structures de données

1. Notion de structure de données

En informatique, on se pose d'abord la question de la modélisation du sujet étudié : comment représenter un entier, un réel, un produit cartésien, une application, etc.

On utilise souvent les mêmes types/schémas de représentation, que l'on définit principalement par les moyens dont on dispose pour les manipuler. On définit ainsi des *types de données* classiques : liste, pile, file, arbre, etc.

Une fois un tel type défini, il faudra réfléchir à la manière de les représenter en machine, en utilisant une *structure de données*.

Le grand avantage d'avoir défini des types de données abstraits est clair : on s'affranchit de l'implémentation effective, de sorte que le programme ainsi écrit se transpose facilement si on change la manière de représenter ces données, ou de langage de programmation. De plus, les quelques structures de données classiques sont connues de tous, et on peut facilement partager son travail.

Les fonctions de manipulation des structures de données sont de trois classes : les *constructeurs*, les *prédicats*, et les *fonctions de sélection*.

Ce contexte définit la *signature* d'un type ou d'une structure de données. Cette signature serait sans grand intérêt si on ne connaissait pas le rôle précis des fonctions de manipulation : c'est l'objet de la *sémantique*.

Les structures de données, le plus souvent récursives/inductives, dictent les programmes qui les manipulent, comme nous l'avons déjà observé sur les listes.

De même, pour prouver des programmes utilisant des structures de données inductives, on utilisera souvent une démonstration par induction structurelle.

2. Listes

2.1. Définition

Soit E un ensemble. On définit une suite d'ensembles (E_n) par $E_0 = \{\emptyset\}$ et $E_{n+1} = E \times E_n$. On appelle listes d'éléments de E un élément de $\mathcal{L}(E) = \cup E_n$.

Par exemple, $(3, (2, (5, \emptyset)))$ est une liste.

Remarque : on pourrait bien sûr confondre cette dernière liste avec $(3, 2, 5)$, mais il faut bien comprendre que l'accès à 3 est plus rapide que l'accès à 2 ou 5, comme nous n'avons cessé de le répéter pour les listes chaînées.

Remarque : bien que dans cet exemple, il ne soit pas simple de définir $\mathcal{L}(E)$ par induction, il faut comprendre en quoi cette structure est néanmoins inductive. Pour prouver qu'un résultat est vrai pour toute liste, on prouvera qu'il l'est pour la liste vide, et qu'il reste vrai par adjonction d'un élément en tête de liste.

On peut définir les fonctions de sélection Tête et Queue, dont la sémantique est claire.

En informatique, la définition d'une liste pourrait être

$$Liste = nil + Element \times Liste$$

exprimant qu'une liste est vide (nil) ou s'écrit (a, q), où a est un élément, q une liste.

En Caml, ces listes sont déjà implémentées, par les listes chaînées (le type 'a list).

Remarque : en mémoire, la liste [1; 3; 8] par exemple, qui peut s'écrire 1 :: 3 :: 8 :: [] , est représentée par un premier couple $(1, a)$ qui donne la valeur en tête de la liste, et un pointeur a pour déterminer où se trouve la valeur suivante, etc.

Remarque : le « cons » : : est un constructeur [1] de la structure de données liste, contrairement à la concaténation @, c'est pourquoi on évitera si possible cette dernière, au profit du premier (qui est accepté en filtrage). En particulier, l'adjonction en queue de liste est bien plus coûteuse que l'adjonction en tête de liste.

Remarque : dans les prédicats, il faudrait rajouter le test d'égalité, ce qui peut poser plus de problèmes qu'on ne le pense, lorsqu'un même objet peut être représenté de manières différentes.

1. Plus exactement, pour tout $x \in E$, $l \mapsto x :: l$ est un constructeur.

Si l'on voulait redéfinir le type liste en Caml, on pourait écrire :

```
(* Définition d'un type liste *)

# type 'a Liste = Nil | Cons of 'a * ('a Liste);;
Type Liste defined.

(* Sémantique des fonctions de manipulation d'une liste *)

(* Prédicat testant si une liste est vide *)

# let est_vide = function
        | Nil -> true
        | _ -> false;;
est_vide : 'a Liste -> bool = <fun>

# est_vide (Cons (1, Nil));;
- : bool = false

(* Constructeur d'ajout en tête de liste *)

# let ajout a l = Cons (a, l);;
ajout : 'a -> 'a Liste -> 'a Liste = <fun>

# ajout 3.8 Nil;;
- : float Liste = Cons (3.8, Nil)

(* Fonctions de sélection *)

# let tete = function
        | Nil -> failwith "pas de tête pour la liste vide"
        | Cons (a, q) -> a;;
tete : 'a Liste -> 'a = <fun>

# let queue = function
        | Nil -> failwith "pas de tête pour la liste vide"
        | Cons (a, q) -> q;;
queue : 'a Liste -> 'a Liste = <fun>

(* Illustration d'un test d'égalité *)

# Cons (1, Cons (2, Nil)) = ajout 1 (Cons (2, Nil));;
- : bool = true
```

Remarque : en Caml, les fonctions tête et queue sont souvent escamotées par les possibilités de filtrage de ce langage, mais il ne faut pas oublier la structure de données sous-jacente.

2.2. Programmes fondamentaux sur les listes

Revoyons, dans le cadre du type Liste ci-dessus, comment redéfinir les fonctions qui à une liste associent respectivement sa longueur, son maximum, son n-ième élément.

```
(* Longueur, maximum, n-ième terme, concaténation *)

# let l = Cons (4, Cons(9, Cons(3, Nil)));;
l : int Liste = Cons (4, Cons (9, Cons (3, Nil)))

# let rec longueur = function
        | Nil -> 0
        | Cons (a, q) -> 1 + longueur q;;
longueur : 'a Liste -> int = <fun>

# longueur l;;
- : int = 3

# let rec maximum = function
        | Nil -> failwith "pas de maximum pour la liste vide"
        | Cons (a, Nil) -> a
        | Cons (a, q) -> max a (maximum q);;
maximum : 'a Liste -> 'a = <fun>

# maximum l;;
- : int = 9

# let rec nieme_terme n l = match l with
        | Nil -> failwith "pas de nieme terme"
        | Cons (a, q) -> match n with
                | 1 -> a
                | _ -> nieme_terme (n - 1) q;;
nieme_terme : int -> 'a Liste -> 'a = <fun>

# for i = 1 to 4 do print_int (nieme_terme i l); print_newline (); done;;
4
9
3
Uncaught exception: Failure "pas de nieme terme"

# let rec concat l l' = match l with
        | Nil -> l'
        | Cons (a, q) -> Cons (a, concat q l');;
concat : 'a Liste -> 'a Liste -> 'a Liste = <fun>

# let l' = Cons(1, Cons(7, Cons(10, Nil)));;
l' : int Liste = Cons (1, Cons (7, Cons (10, Nil)))

# concat l l';;
- : int Liste =
 Cons (4, Cons (9, Cons (3, Cons (1, Cons (7, Cons (10, Nil))))))
```

```
(* Définition d'un type liste *)

# type 'a Liste = Nil | Cons of 'a * ('a Liste);;

(* Sémantique des fonctions de manipulation d'une liste *)

(* Prédicat testant si une liste est vide *)

# let est_vide = function
        | Nil -> true
        | _ -> false;;
Type Liste defined.

# est_vide (Cons (1, Nil));;
est_vide : 'a Liste -> bool = <fun>

(* Constructeur d'ajout en tête de liste *)

# let ajout a l = Cons (a, l);;
ajout : 'a -> 'a Liste -> 'a Liste = <fun>

# ajout 3.8 Nil;;
- : float Liste = Cons (3.8, Nil)

(* Fonctions de sélection *)

# let tete = function
        | Nil -> failwith "pas de tête pour la liste vide"
        | Cons (a, q) -> a;;
tete : 'a Liste -> 'a = <fun>

# let queue = function
        | Nil -> failwith "pas de tête pour la liste vide"
        | Cons (a, q) -> q;;
queue : 'a Liste -> 'a Liste = <fun>

(* Illustration d'un test d'égalité *)

# Cons (1, Cons (2, Nil)) = ajout 1 (Cons (2, Nil));;
- : bool = true
```

```
(* Longueur, maximum, n-ième terme, concaténation *)

# let l = Cons (4, Cons(9, Cons(3, Nil)));;
l : int Liste = Cons (4, Cons (9, Cons (3, Nil)))

# let rec longueur = function
        | Nil -> 0
        | Cons (a, q) -> 1 + longueur q;;
longueur : 'a Liste -> int = <fun>

# longueur l;;
- : int = 3

# let rec maximum = function
        | Nil -> failwith "pas de maximum pour la liste vide"
        | Cons (a, Nil) -> a
        | Cons (a, q) -> max a (maximum q);;
maximum : 'a Liste -> 'a = <fun>

# maximum l;;
- : int = 9

# let rec nieme_terme n l = match l with
        | Nil -> failwith "pas de nieme terme"
        | Cons (a, q) -> match n with
                | 1 -> a
                | _ -> nieme_terme (n - 1) q;;
nieme_terme : int -> 'a Liste -> 'a = <fun>

# for i = 1 to 4 do print_int (nieme_terme i l); print_newline (); done;;
4
9
3
Uncaught exception: Failure "pas de nieme terme"

# let rec concat l l' = match l with
        | Nil -> l'
        | Cons (a, q) -> Cons (a, concat q l');;
concat : 'a Liste -> 'a Liste -> 'a Liste = <fun>

# let l' = Cons(1, Cons(7, Cons(10, Nil)));;
l' : int Liste = Cons (1, Cons (7, Cons (10, Nil)))

# concat l l';;
- : int Liste =
 Cons (4, Cons (9, Cons (3, Cons (1, Cons (7, Cons (10, Nil))))))
```

3. Piles

3.1. Modèle de données, implémentation

Une *pile* est un modèle de données, dite structure LIFO (last in, first out), dont la signature est donnée par les constructeurs Crée_pile_vide et Empile, les fonctions de sélection sont Sommet et Dépile, le prédicat Est_vide (et éventuellement Egalite).

On ne peut ajouter ou supprimer des éléments qu'au sommet de la pile.

Remarque : on ne peut dépiler un élément que si on a déjà dépilé les éléments qui sont « au-dessus ». Cela se produit en sens inverse, et c'est un peu ce que nous avons fait pour l'image miroir.

Le modèle pile est déjà implémenté dans Caml, sous le nom stack :

```
(* Le type stack donné par Caml *)

# #open "stack";;

# let l = new ();;
l : '_a t = <abstr>

# pop l;;
Uncaught exception: Empty

# push 3 l;;
- : unit = ()

# push 4 l;;
- : unit = ()

# l;;
- : int t = <abstr>

# pop l;;
- : int = 4

# l;;
- : int t = <abstr>

# pop l;;
- : int = 3

(* On dispose aussi des fonctions clear, length, mais pas de top *)
```

Remarque : push et pop agissent par effets de bord sur la pile passée en argument.

Exemples de piles : historique dans un navigateur, expressions parenthésées, % ou # en Maple.

Remarque : le modèle de données pile est isomorphe à celui des listes.

Remarque : il existe une structure proche des piles, celle *file*, fondée sur le principe FIFO (first in, first out). On peut penser à la notion de file d'attente, que l'on retrouve par exemple dans le tampon d'un clavier.

Pour implémenter le type pile, on peut penser à

```
(* Représentation par une liste *)

(* Par "abréviation" *)

# type 'a pile = 'a list;;
Type pile defined.

(* Par structure inductive *)

# type 'a pile = Pile_Vide | Empile of 'a * 'a pile;;
Type pile defined.
```

```
(* Par un type mutable *)

# type 'a pile = {mutable contenu : 'a list}
Type pile defined.


(* Représentation par un vecteur *)

# type 'a pile = { contenu : 'a vect; mutable sommet : int};;
Type pile defined.
```

Nous allons utiliser l'implémentation par un type de contenu une liste mutable :

```
(* Structure de données pile *)

# type 'a pile = {mutable contenu : 'a list};;
Type pile defined.

# let pile_vide () = {contenu = []};;
pile_vide : unit -> 'a pile = <fun>

# let est_vide pile = pile.contenu = [];;
est_vide : 'a pile -> bool = <fun>

# est_vide {contenu = [5; 6]};;
- : bool = false

# est_vide (pile_vide ());;
- : bool = true

# let empile a pile = pile.contenu <- a :: pile.contenu;;
empile : 'a -> 'a pile -> unit = <fun>

# let pile = pile_vide ();;
pile : '_a pile = {contenu = []}

# empile 3 pile;;
- : unit = ()

# empile 9 pile;;
- : unit = ()

# pile;;
- : int pile = {contenu = [9; 3]}

# let depile pile = match pile.contenu with
        | [] -> failwith "Pile vide"
        | a :: q -> pile.contenu <- q;
                    a;;
depile : 'a pile -> 'a = <fun>

# depile pile;; (* On renvoie le sommet de notre pile *)
- : int = 9

# pile;; (* depile a agi par effet de bord sur notre pile *)
- : int pile = {contenu = [3]}

# depile pile;;
```

```
- : int = 3

# pile;;
- : int pile = {contenu = []}

# depile pile;;
Uncaught exception: Failure "Pile vide"

# pile;;
- : int pile = {contenu = []}
```

3.2. Pile de récursivité

Les appels et retours d'une fonction récursive ou même de plusieurs sont gérés par une pile d'exécution. On peut suivre ce processus grâce à la fonction trace.

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

# trace "fibo";;
The function fibo is now traced.
- : unit = ()

# fibo 4;;
fibo <-- 4
fibo <-- 3
fibo <-- 2
fibo <-- 1
fibo --> 1
fibo <-- 0
fibo --> 1
fibo --> 2
fibo <-- 1
fibo --> 1
fibo --> 3
fibo <-- 2
fibo <-- 1
fibo --> 1
fibo <-- 0
fibo --> 1
fibo --> 2
fibo --> 5
- : int = 5
```

3.3. Expressions algébriques postfixées

Nous sommes habitués à la représentation infixe des expressions algébriques :

$$(5.2 - ((4.1 + 8.9) \times (5.2/9.3))) \times (3.3 - 6)$$

On peut définir inductivement la notion d'expression algébrique.

```
(* Définition d'un type expression algébrique *)

# type exp_alg =
        | Var of float
        | Plus of exp_alg * exp_alg
        | Moins of exp_alg * exp_alg
        | Mult of exp_alg * exp_alg
```

```
        | Divise of exp_alg * exp_alg;;
Type exp_alg defined.

# let exemple = Mult
                (Moins
                        (
                        Var 5.2,
                        Mult
                                (
                                Plus (Var 4.1, Var 8.9),
                                Divise (Var 5.2, Var 9.3)
                                )
                        ),
                Moins (Var 3.3, Var 6.)
                );;
exemple : exp_alg =
 Mult
  (Moins (Var 5.2, Mult (Plus (Var 4.1, Var 8.9), Divise (Var 5.2, Var 9.3))),
   Moins (Var 3.3, Var 6.0))
```

La représentation la plus naturelle de cette structure inductive est arborescente. Pour effectuer une représentation linéaire d'un arbre, on peut le parcourir en profondeur, en mettant l'étiquette d'un nœud quand on y arrive la première fois[2] (écriture *préfixe* ou *préfixée*) ou la troisième fois (forme *postfixe*, ou *postfixée*, ou *polonaise inverse*). Ces écritures ont l'avantage de ne pas nécessiter de parenthèses, contrairement à l'écriture infixe.

Lorsqu'on se donne une expression algébrique, on peut bien sûr l'évaluer :

```
(* Evaluation d'une expression algébrique *)

# let rec evalue = function
        | Var x -> x
        | Plus (ea1, ea2) -> evalue ea1 +. evalue ea2
        | Moins (ea1, ea2) -> evalue ea1 -. evalue ea2
        | Mult (ea1, ea2) -> evalue ea1 *. evalue ea2
        | Divise (ea1, ea2) -> evalue ea1 /. evalue ea2;;
evalue : exp_alg -> float = <fun>

# evalue exemple;;
- : float = 5.58580645161
```

Nous allons voir en TD comment traiter une expression algébrique postfixée (EAP) à l'aide d'une pile.
Remarque : on pourrait rajouter des opérateurs unaires (passage à l'opposé, à l'inverse, fonctions usuelles), ou pire, n-aires.

2. C'est d'ailleurs comme cela qu'on la rentré dans Caml

```
(* Le type stack donné par Caml *)

# #open "stack";;

# let l = new ();;
l : '_a t = <abstr>

# pop l;;
Uncaught exception: Empty

# push 3 l;;
- : unit = ()

# push 4 l;;
- : unit = ()

# l;;
- : int t = <abstr>

# pop l;;
- : int = 4

# l;;
- : int t = <abstr>

# pop l;;
- : int = 3

(* On dispose aussi des fonctions clear, length, mais pas de top *)
```

```
(* Représentation par une liste *)

(* Par "abréviation" *)

# type 'a pile == 'a list;;
Type pile defined.

(* Par structure inductive *)

# type 'a pile = Pile_Vide | Empile of 'a * 'a pile;;
Type pile defined.


(* Par un type mutable *)

# type 'a pile = {mutable contenu : 'a list}
Type pile defined.


(* Représentation par un vecteur *)

# type 'a pile = {contenu : 'a vect; mutable sommet : int};;
Type pile defined.
```

```
(* Structure de données pile *)

# type 'a pile = {mutable contenu : 'a list};;
Type pile defined.

# let pile_vide () = {contenu = []};;
pile_vide : unit -> 'a pile = <fun>

# let est_vide pile = pile.contenu = [];;
est_vide : 'a pile -> bool = <fun>

# est_vide {contenu = [5; 6]};;
- : bool = false

# est_vide (pile_vide ());;
- : bool = true

# let empile a pile = pile.contenu <- a :: pile.contenu;;
empile : 'a -> 'a pile -> unit = <fun>

# let pile = pile_vide ();;
pile : '_a pile = {contenu = []}

# empile 3 pile;;
- : unit = ()

# empile 9 pile;;
- : unit = ()

# pile;;
- : int pile = {contenu = [9; 3]}

# let depile pile = match pile.contenu with
        | [] -> failwith "Pile␣vide"
        | a :: q -> pile.contenu <- q;
                    a;;
depile : 'a pile -> 'a = <fun>

# depile pile;; (* On renvoie le sommet de notre pile *)
- : int = 9

# pile;; (* depile a agi par effet de bord sur notre pile *)
- : int pile = {contenu = [3]}

# depile pile;;
- : int = 3

# pile;;
- : int pile = {contenu = []}

# depile pile;;
Uncaught exception: Failure "Pile␣vide"

# pile;;
- : int pile = {contenu = []}
```

```
# type exp_alg =
        | Var of float
        | Plus of exp_alg * exp_alg
        | Moins of exp_alg * exp_alg
        | Mult of exp_alg * exp_alg
        | Divise of exp_alg * exp_alg;;
Type exp_alg defined.

# let exemple = Mult
                (Moins
                        (
                        Var 5.2,
                        Mult
                                (
                                Plus (Var 4.1, Var 8.9),
                                Divise (Var 5.2, Var 9.3)
                                )
                        ),
                Moins (Var 3.3, Var 6.)
                );;
exemple : exp_alg =
 Mult
  (Moins (Var 5.2, Mult (Plus (Var 4.1, Var 8.9), Divise (Var 5.2, Var 9.3))),
   Moins (Var 3.3, Var 6.0))

(* Evaluation d'une expression algébrique *)

# let rec evalue = function
        | Var x -> x
        | Plus (ea1, ea2) -> evalue ea1 +. evalue ea2
        | Moins (ea1, ea2) -> evalue ea1 -. evalue ea2
        | Mult (ea1, ea2) -> evalue ea1 *. evalue ea2
        | Divise (ea1, ea2) -> evalue ea1 /. evalue ea2;;
evalue : exp_alg -> float = <fun>

# evalue exemple;;
- : float = 5.58580645161
```

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

# trace "fibo";;
The function fibo is now traced.
- : unit = ()

# fibo 4;;
fibo <-- 4
fibo <-- 3
fibo <-- 2
fibo <-- 1
fibo --> 1
fibo <-- 0
fibo --> 1
fibo --> 2
fibo <-- 1
fibo --> 1
fibo --> 3
fibo <-- 2
fibo <-- 1
fibo --> 1
fibo <-- 0
fibo --> 1
fibo --> 2
fibo --> 5
- : int = 5
```

Cinquième partie

Logique

CHAPITRE VIII

Calcul propositionnel

Distinction entre syntaxe et sémantique. Dans une première séance, nous allons nous occuper de la syntaxe exclusivement.

1. Préliminaires sur les arbres binaires

Le modèle de données arbre binaire peut être résumé par la formule suivante :

$$Arbre = nil + Arbre \times Element \times Arbre$$

Remarque : encore une fois on a réifié l'ensemble vide pour s'en sortir.

Un arbre peut être représenté par des sommets reliés par des flèches. Étant donné une flèche, on a un *fils* (gauche ou droit) et son *père*. La *racine* est le seul sommet sans père. Un sommet sans fils est une *feuille*. Les descendants d'un nœud sont ses fils, et les descendants de ses fils.

Exemples : représentation arborescente des expressions arithmétiques, listes (chaque fils gauche est une feuille), arbre de décision, et, bientôt, propositions logiques.

Remarque : la structure inductive de cette notion d'arbre binaire fait que nous donnerons des preuves par induction structurelle. De même, nous définirons une fonction sur cet ensemble inductivement.

Remarque : définition de la notion de sous-arbre, de la hauteur d'un arbre.

2. Syntaxe

2.1. Ensemble inductif des propositions logiques

Nous utiliserons un ensemble $\mathcal{V}$, au plus dénombrable, d'éléments appelés *variables propositionnelles*. Nous les noterons en minuscules, et nous noterons les propositions logiques en majuscules.

Plusieurs symboles : la disjonction « ou » notée OR ou $\vee$, la conjonction « et » notée AND ou $\wedge$, et la négation notée NOT ou $\neg$, l'implication $\Rightarrow$, l'équivalence $\Leftrightarrow$, le ou exclusif XOR, le « non et » NAND, le « non ou » NOR.

On définit l'ensemble $\mathcal{P}$ des *propositions* (ou *formules*) inductivement par l'ensemble de base $\mathcal{V}$ et les constructeurs $\vee, \wedge, \neg$, d'arités respectives $2, 2, 1$.

Remarque : on ajoute parfois les constantes V et F (se lisent respectivement vrai et faux).

```
# type binaire = Et | Ou | Impli | Equiv | XOR | NAND;;
Type binaire defined.

# type formule =
        | Const of bool
        | Var of string
        | Non of formule
        | Bin of binaire * formule * formule;;
Type formule defined.

# let P = Bin (Ou, (Const true), Bin (Impli, Var "a", Var "b"));;
P : formule = Bin (Ou, Const true, Bin (Impli, Var "a", Var "b"))
```

Nous pouvons donner une représentation arborescente étiquetée de ces propositions logiques : les feuilles sont étiquetées par des variables propositionnelles, les nœuds d'arité 1 par $\neg$, les nœuds d'arité 2 le sont par les autres connecteurs.

Nous en profitons pour définir les notions de sous-proposition (ou sous-formule) d'une formule, ou la hauteur d'une proposition. On peut décrire inductivement l'ensemble des sous-formules d'une formule :

L'ensemble des sous-formules $s(P)$ est :

- $\{P\}$ si $P = a$.
- $s(Q) \cup \{P\}$ si $P = \neg Q$.
- $s(Q) \cup s(R) \cup \{P\}$ si s est un opérateur binaire.

Programme Caml donnant les sous-propositions :

```
(* Liste des sous-formules d'une formule *)

# let rec sous_formule prop = match prop with
        | Const _ | Var _ -> [prop]
        | Non prop' -> prop :: (sous_formule prop')
        | Bin (op, sa1, sa2) ->
                prop :: (sous_formule sa1) @ (sous_formule sa2);;
sous_formule : formule -> formule list = <fun>

# sous_formule P;;
- : formule list =
 [Bin (Ou, Const true, Bin (Impli, Var "a", Var "b")); Const true;
  Bin (Impli, Var "a", Var "b"); Var "a"; Var "b"]
```

Calcul de la hauteur d'une formule :

```
(* Hauteur d'une formule *)

# let rec hauteur = function
        | Const _ | Var _ -> 0
        | Non prop' -> hauteur prop' + 1
        | Bin (op, sa1, sa2) -> max (hauteur sa1) (hauteur sa2) + 1;;
hauteur : formule -> int = <fun>

# hauteur P;;
- : int = 2
```

2.2. Syntaxe des représentations postfixées des propositions logiques

On peut donner une version postfixée des propositions logiques, pour laquelle les parenthèses sont inutiles. Ces écritures sont des mots sur l'alphabet $\mathcal{V} \cup \{\vee, \wedge, \neg, \Rightarrow, \Leftrightarrow, XOR, NAND, NOR\}$ (ce sont des suites finies de lettres). On définit de manière intuitive la notion de concaténation (ou produit), de préfixe (ou facteur gauche) d'un mot.

Remarque : on définit aussi la notion de suffixe (ou facteur droit), et même la notion de facteur, à ne pas confondre avec la notion de sous-mot.

Remarque : pour montrer l'injectivité de la notation polonaise inverse, on utilise la preuve de Jan Lukasiewicz (pour la notation polonaise ... pas inverse). Pour tout lexème l, on pose $\nu(l) = 1$ si c'est une variable propositionnelle, $\nu(l) = -1$ si l est un opérateur binaire, et $\nu(l) = 0$ si l est l'opérateur de négation.

Pour toute chaîne de caractère $L = l_1 \dots l_n$, on note $\mu(L)$ et appelle *poids* de L la somme $\sum_{i=1}^{n} \nu(l_i)$. C'est donc aussi le nombre de variables propositionnelles moins le nombre de connecteurs binaires de L. C'est un morphisme de monoïdes.

On montre que :

(1) pour toute expression logique L, $\mu(L) = 1$ (par induction structurelle).

(2) pour tout préfixe L' non vide de L, $\mu(L') \geqslant 1$ (par induction structurelles, en séparant selon l'endroit où se trouvent les arguments).

(3) un mot est une expression logique postfixée si et seulement si elle est de poids 1, et si tous ses préfixes non vides sont de poids au moins 1 (par récurrence sur $|L|$ pour la réciproque, en prenant un préfixe propre y de poids 1 de longueur maximale : s'il ne manque qu'un terme, c'est la négation, d'où le résultat. Sinon, L se termine par un connecteur logique b, donc $l = yzb$. Par induction, y est une expression logique postfixée. On montre alors que z est une expression logique postfixée, car elle est de poids 1, et que tous ses facteurs propres sont de poids 1 au moins (il ne peuvent pas être de poids 0)).

(4) montrer que toute expression logique postfixée provient d'une unique proposition logique. Si on a $xyb = x'y'b$, alors, par exemple, x' est un préfixe de x, donc $x = x'z$, où z est de poids nul. Or $y' = zy$ donc z est un préfixe de y' donc z est vide. Il s'ensuit $x = x'$ puis $y = y'$.

Remarque : bien entendu, cette preuve s'adapte au cas des expressions algébriques posfixées. D'ailleurs, on peut considérer les propositions logiques comme un cas particulier d'expression algébrique.

2.3. Représentation infixe (ou parenthésée) des propositions logiques

La représentation arborescente n'est pas toujours très maniable. On s'intéresse ici à une représentation linéaire infixe (donc nécessairement parenthésée) des propositions logiques, déduite du parcours en profondeur de l'arbre :

On associe à une proposition logique (vue comme un arbre binaire étiqueté) une expression linéaire de manière inductive :

```
(* Représentation linéaire infixe parenthésée d'une formule *)

# let print_op = function
        | Et -> " et "
        | Ou -> " ou "
        | Impli -> " -> "
        | Equiv -> " <-> "
        | XOR -> " xor "
        | NAND -> " nand ";;
print_op : binaire -> string = <fun>

# let rec ecriture = function
        | Const true -> "V"
        | Const _ -> "F"
        | Var a -> a
        | Non  prop -> "(non " ^ (ecriture prop) ^ ")"
                (* Les parenthèses sont facultatives pour la négation *)
        | Bin (op, sa1, sa2) ->
                "(" ^ (ecriture sa1) ^ (print_op op) ^ (ecriture sa2) ^ ")";;
ecriture : formule -> string = <fun>

# ecriture P;;
- : string = "(V ou (a -> b))"
```

Une telle expression est donc un mot sur l'alphabet constitué de $\mathcal{V}$, des connecteurs logiques, des parenthèses.

Bien sûr, on définit ainsi une application de l'ensemble des propositions logiques sur l'ensemble des mots construits sur cet alphabet : cette application n'est pas surjective (donner des exemples), mais, pour qu'elle soit pertinente, nous allons vérifier l'injectivité de cette application.

(1) On montre par induction structurelle que le nombre de parenthèses ouvrantes est égal au nombre de parenthèses fermantes.

(2) Pour tout préfixe de $P \in \mathcal{P}$, et si P' est un préfixe de P, alors $o(P') \geqslant f(P')$.

(3) On montre ensuite qu'un préfixe propre d'une expression parenthésée n'est pas une expression parenthésée.

(4) Conclusion.

Remarque : pour alléger l'expression parenthésée, on peut définir un ordre de priorité des opérateurs. J'éviterai autant que possible d'utiliser cette priorité, sauf pour la négation.

```
# type binaire = Et | Ou | Impli | Equiv | XOR | NAND;;
Type binaire defined.

# type formule =
        | Const of bool
        | Var of string
        | Non of formule
        | Bin of binaire * formule * formule;;
Type formule defined.

# let P = Bin (Ou, (Const true), Bin (Impli, Var "a", Var "b"));;
P : formule = Bin (Ou, Const true, Bin (Impli, Var "a", Var "b"))

(* Liste des sous-formules d'une formule *)

# let rec sous_formule prop = match prop with
        | Const _ | Var _ -> [prop]
        | Non prop' -> prop :: (sous_formule prop')
                (* Les parenthèses sont facultatives pour la négation *)
        | Bin (op, sa1, sa2) ->
                prop :: (sous_formule sa1) @ (sous_formule sa2);;
sous_formule : formule -> formule list = <fun>

# sous_formule P;;
- : formule list =
 [Bin (Ou, Const true, Bin (Impli, Var "a", Var "b")); Const true;
  Bin (Impli, Var "a", Var "b"); Var "a"; Var "b"]

(* Hauteur d'une formule *)

# let rec hauteur = function
        | Const _ | Var _ -> 0
        | Non prop' -> hauteur prop' + 1
        | Bin (op, sa1, sa2) -> max (hauteur sa1) (hauteur sa2) + 1;;
hauteur : formule -> int = <fun>

# hauteur P;;
- : int = 2
```

```
(* Représentation linéaire infixe parenthésée d'une formule *)

# let print_op = function
        | Et -> " et "
        | Ou -> " ou "
        | Impli -> " -> "
        | Equiv -> " <-> "
        | XOR -> " xor "
        | NAND -> " nand ";;
print_op : binaire -> string = <fun>

# let rec ecriture = function
        | Const true -> "V"
        | Const _ -> "F"
        | Var a -> a
        | Non  prop -> "(non " ^ (ecriture prop) ^ ")"
                (* Les parenthèses sont facultatives pour la négation *)
        | Bin (op, sa1, sa2) ->
                "(" ^ (ecriture sa1) ^ (print_op op) ^ (ecriture sa2) ^ ")";;
ecriture : formule -> string = <fun>

# ecriture P;;
- : string = "(V ou (a -> b))"
```

3. Sémantique

3.1. Évaluation d'une formule logique

On note $\mathcal{V}$ l'ensemble des variables propositionnelles, $\mathcal{P}$ celui des propositions logiques. On rappelle que par convention, les variables propositionnelles seront en minuscules, les propositions logiques en majuscules.

Jusqu'à présent, nous n'avons pas donné la moindre signification à une proposition logique, et ne nous sommes intéressés qu'à leur syntaxe. Par exemple, nous ne savons pas quel sens attribuer à $a \vee (\neg a)$, et nous ne pouvons pas comparer $a \vee (b \vee c)$ et $(a \vee b) \vee c$.

Définition (Distribution de vérité)

On appelle *distribution de vérité* toute application de $\mathcal{V}$ dans $\{0,1\}$ (intuitivement, la valeur 1 correspond à vrai, 0 à faux).

3.a

Remarque : par commodité de calcul, nous identifions $\{0,1\}$ au corps à deux éléments $\mathbb{F}_2 = \mathbb{Z}/2\mathbb{Z}$ (et nous en utiliserons les opérations).

Définition (Évaluation d'une formule)

Soit σ une distribution de vérité. On définit inductivement la *valeur* (ou *évaluation*) d'une formule en σ par :

(1) $V_\sigma(V) = 1$, $V_\sigma(F) = 0$.
(2) $V_\sigma(a) = \sigma(a)$ pour tout $a \in \mathcal{V}$.
(3) $V_\sigma(\neg A) = 1 - V_\sigma(A)$ pour tout $A \in \mathcal{P}$.
(4) $V_\sigma(A \wedge B) = V_\sigma(A)V_\sigma(B) = \min(V_\sigma(A), V_\sigma(B))$ pour tout $(A,B) \in \mathcal{P}^2$.
(5) $V_\sigma(A \vee B) = V_\sigma(A) + V_\sigma(B) - V_\sigma(A)V_\sigma(B) = \max(V_\sigma(A), V_\sigma(B))$ pour tout $(A,B) \in \mathcal{P}^2$.
(6) $V_\sigma(A \Rightarrow B) = \max(1 - V_\sigma(A), V_\sigma(B))$ pour tout $(A,B) \in \mathcal{P}^2$.
(7) $V_\sigma(A \Leftrightarrow B) = 1 - V_\sigma(A) - V_\sigma(B)$ pour tout $(A,B) \in \mathcal{P}^2$.
(8) $V_\sigma(A \,\mathrm{XOR}\, B) = V_\sigma(A) + V_\sigma(B)$ pour tout $(A,B) \in \mathcal{P}^2$.
(9) $V_\sigma(A \,\mathrm{NAND}\, B) = 1 - V_\sigma(A)V_\sigma(B)$ pour tout $(A,B) \in \mathcal{P}^2$.
(10) $V_\sigma(A \,\mathrm{NOR}\, B) = (1 - V_\sigma(A))(1 - V_\sigma(B))$ pour tout $(A,B) \in \mathcal{P}^2$.

3.b

Remarque : bien évidemment, l'évaluation d'une proposition A ne dépend que des valeurs de vérité attribuées aux variables propositionnelles de A.

Remarque : concrètement, on peut écrire la proposition logique sous forme d'arbre, remplacer les variables propositionnelles par les valeurs associées par σ, les opérateurs par les fonctions associées, et on évalue comme une expression algébrique.

Remarque : pour donner toutes les évaluations possibles d'une formule logique, on peut établir sa *table de vérité*, en ne faisant intervenir que les variables propositionnelles pertinentes. On peut éventuellement utiliser des colonnes intermédiaires pour clarifier les calculs (si par exemple on vous demande la table de vérité de $A \wedge (B \vee (C \Rightarrow (B \vee A)))$).

A	$\neg A$
0	1
1	0

A	B	$A \vee B$	$A \wedge B$	$A \Rightarrow B$	$A \Leftrightarrow B$	$A \Rightarrow B$	$A \Leftrightarrow B$	$A \,\mathrm{XOR}\, B$	$A \,\mathrm{NAND}\, B$	$A \,\mathrm{NOR}\, B$
0	0	0	0	1	1	1	1	0	1	1
0	1	1	0	1	0	1	0	1	1	0
1	0	1	0	0	0	0	0	1	1	0
1	1	1	1	1	1	1	1	0	0	0

Remarque : pour n variables propositionnelles, une table de vérité comprend $2^n (+1)$ lignes, mais on a $2^{(2^n)}$ tables de vérité possibles.

Voir le programme Caml fourni en annexe.

3.2. Propositions équivalentes

Définition (Propositions équivalentes)

Deux propositions A et B sont dites *logiquement équivalentes* (ou *sémantiquement équivalentes*) si pour tout $\sigma \in \{0,1\}^{\mathcal{V}}$, $V_\sigma(A) = V_\sigma(B)$. On note alors $A \equiv B$.

3.c

Remarque : on définit ainsi une relation d'équivalence sur $\mathcal{P}$.
Remarque : si des propositions A et B, dépendant de variables propositionnelles $a_1, \ldots, a_n$, sont logiquement équivalentes, alors on obtient des propositions logiquement équivalentes en substituant à l'une des variables une même proposition quelconque, ce qui permet de créer de nombreuses formules équivalentes.

Voici des exemples classiques de formules équivalentes (où A et B désignent des propositions quelconques) :
- A et $\neg\neg A$ (loi du tiers exclu)
- $\neg(A \wedge B)$ et $(\neg A) \vee (\neg B)$. $\neg(A \vee B)$ et $(\neg A) \wedge (\neg B)$. Ce sont les lois de (De) Morgan.
- $A \Rightarrow B$, $(\neg B) \Rightarrow (\neg A)$ et $(\neg A) \vee B$.
- $A \Leftrightarrow B$, $(A \Rightarrow B) \wedge (B \Rightarrow A)$.
- $A \operatorname{XOR} B$ et $(A \vee B) \wedge (\neg(A \wedge B))$.
- $A \operatorname{NAND} B$ et $\neg(A \wedge B)$.
- $A \operatorname{NOR} B$ et $\neg(A \vee B)$.
- etc.

3.3. Fonction booléenne associée à une formule

Définition (Fonction booléenne)

On appelle *fonction booléenne* toute application de $\{0,1\}^n$ dans $\{0,1\}$.

3.d

Définition (Fonction booléenne associée à une formule)

On considère une proposition A dépendant de variables propositionnelles $a_1, \ldots, a_n$ ordonnées.
On définit alors une fonction $f_A : \{0,1\}^n \to \{0,1\}$, obtenue à partir de la table de vérité.

3.e

Remarque : cette définition est ambiguë, car celle de dépendance (ou de variable) l'est : nous pourrons avoir besoin de rajouter des variables propositionnelles. Cela pourrait poser des problèmes en cas d'implémentation.

3.4. Types particuliers de formules

Définition (Tautologie, antilogie, proposition satisfiable)

Soit $A \in \mathcal{P}$. On dit que A est une *tautologie* si $V_\sigma(A) = 1$ pour toute distribution de vérité σ.
On dit que A est une *antilogie* si $V_\sigma(A) = 0$ pour toute distribution de vérité σ.
A est dite *satisfiable* (ou *satisfaisable*) s'il existe une distribution de vérité σ pour laquelle $V_\sigma(A) = 1$.

3.f

Remarque : A est une tautologie si et seulement si f_A est constante de valeur 1 si et seulement si $A \equiv V$.
Remarque : A et B sont équivalentes si et seulement si $A \Leftrightarrow B$ est une tautologie.

Remarque : encore une fois, si on a une tautologie, alors on obtient des tautologies en substituant certaines de ses variables par des propositions. Par exemple, $a \vee (\neg a)$ est une tautologie, donc, en substituant à a la proposition $A = a \vee b \vee c$, $(a \vee b \vee c) \vee (\neg(a \vee b \vee c))$ est une tautologie.

3.5. Formes normales disjonctives et conjonctives

En ne s'attachant qu'au fond, non plus à la forme, on a constaté que certaines propositions « revenaient au même ». Cependant, cette perte d'unicité peut être nuisible : on cherche donc un représentant normalisé d'une classe d'équivalence.

Définition (Littéral)

On appelle *littéral* une variable propositionnelle ou la négation d'une variable propositionnelle.

3.g

Définition (Clause)

On appelle *clause* toute disjonction de littéraux.

3.h

Définition (Forme conjonctive)

On appelle *forme conjonctive* d'une proposition logique toute conjonction de disjonction de littéraux. On définit de même la notion de *forme disjonctive*.

3.i

Théorème (formes conjonctive et disjonctive)

Toute formule logique est équivalente à une conjonction de disjonction de littéraux, et à une disjonction de conjonctions de littéraux.

3.a

Démonstration

On montre les deux résultats en même temps, par induction structurelle. □

En pratique, pour la mise sous forme disjonctive on peut

(1) tout écrire en fonction des connecteurs logiques $\neg$, $\wedge$ et $\vee$.

(2) on descend le connecteur $\neg$ le plus bas possible grâce aux lois de Morgan.

(3) on utilise la distributivité de $\wedge$ par rapport à $\vee$.

Remarque : on a bien sûr un algorithme équivalent pour la mise sous forme conjonctive.
Remarque : nous verrons également une méthode fondée sur l'utilisation des tables de vérité.

Définition (minterme)

Soit A une proposition de variables propositionnelles $a_1, \ldots, a_n$. On appelle *minterme* toute conjonction de littéraux, où chaque variable propositionnelle apparaît exactement une fois. On définit de même la notion de *maxterme*.

3.j

Remarque : on a donc 2^n mintermes possibles (on fixe un ordre total sur les variables propositionnelles).

Définition (Forme normale disjonctive)

On appelle *forme normale disjonctive* d'une proposition A de variables $a_1, \dots, a_n$ toute disjonction de mintermes distincts, logiquement équivalente à A. On définit de même la notion de forme normale conjontive.

3.k

Théorème (forme normale disjonctive)

Toute proposition logique admet une forme normale disjonctive. De plus, il y a unicité à l'ordre des termes près.

3.b

Démonstration

On peut utiliser les tables de vérité. □

Remarque : retenir l'utilisation des tables de vérité, qui donne directement une forme normale disjonctive.
Remarque : en utilisant les lois de Morgan, on trouve une forme normale conjonctive.

3.6. Déduction logique

Pour montrer qu'une proposition logique C est une conséquence logique de propositions $H_1, \dots, H_n$, *i.e.* $(H_1 \wedge \cdots \wedge H_n) \Rightarrow C$, on peut (n')utiliser (que) la règle (tautologie) dite de résolution suivante :

$$((a \vee b) \wedge (\neg a \vee c)) \Rightarrow (b \vee c),$$

en partant de $H_1 \wedge \cdots \wedge H_n$, pour aboutir sur C. On peut aussi partir de $(H_1 \wedge \cdots \wedge H_n) \wedge (\neg C)$.

Plus précisément, on écrit $H_1, \dots, H_n, \neg C$ sous forme de conjonctions de clauses, puis on utilise cette méthode de résolution pour réduire la proposition à la proposition équivalente F.

```
(* Sémantique d'une proposition *)

(* Opérateur binaire et opération booléenne correspondante *)

# let opere b p p' = match b with
        | Et -> p * p'
        | Ou -> p + p' - p * p'
        | Impli -> max (1 - p) p'
        | Equiv -> (1 + p + p') mod 2
        | XOR -> (p + p') mod 2
        | NAND -> 1 - p * p'
        | NOR -> (1 - p) * (1 - p');;
opere : binaire -> int -> int -> int = <fun>

(* Evaluation d'une formule pour une distribution de vérité donnée *)

# let rec eval p distrib = match p with
        | Const b -> if b then 1 else 0
        | Var s -> distrib s
        | Non p' -> 1 - (eval p' distrib)
        | Bin (b,p',p'') -> opere b (eval p' distrib) (eval p'' distrib);;
eval : formule -> (string -> int) -> int = <fun>

# eval P (function "a" | "b" | _ -> 0);;
- : int = 1

(* Fonction préliminaire à la création d'une table de vérité *)

# let f va vb = function
        | "a" -> va
        | "b" -> vb
        | _ -> failwith "variable non prise en compte";;
f : 'a -> 'a -> string -> 'a = <fun>

(* Création d'une ligne *)

# let ligne a b c =
        print_string "| ";
        print_int a;
        print_string " | ";
        print_int b;
        print_string " || ";
        print_int c;
        print_string " |";
        print_newline ();;
ligne : int -> int -> int -> unit = <fun>

(* Création d'une table de vérité *)

# let table q =
        print_newline ();
        for i = 0 to 1 do
                for j = 0 to 1 do
                        ligne i j (eval q (f i j))
                done;
        done;;
table : formule -> unit = <fun>
```

```
(* Tables des connecteurs logiques binaires élémentaires *)

# let tablebin b = table (Bin (b, Var "a", Var "b"));;
tablebin : binaire -> unit = <fun>

# tablebin Et;;
| 0 | 0 || 0 |
| 0 | 1 || 0 |
| 1 | 0 || 0 |
| 1 | 1 || 1 |
- : unit = ()

# tablebin Ou;;
| 0 | 0 || 0 |
| 0 | 1 || 1 |
| 1 | 0 || 1 |
| 1 | 1 || 1 |
- : unit = ()

# tablebin Impli;;
| 0 | 0 || 1 |
| 0 | 1 || 1 |
| 1 | 0 || 0 |
| 1 | 1 || 1 |
- : unit = ()

# tablebin Equiv;;
| 0 | 0 || 1 |
| 0 | 1 || 0 |
| 1 | 0 || 0 |
| 1 | 1 || 1 |
- : unit = ()

# tablebin XOR;;
| 0 | 0 || 0 |
| 0 | 1 || 1 |
| 1 | 0 || 1 |
| 1 | 1 || 0 |
- : unit = ()

# tablebin NAND;;
| 0 | 0 || 1 |
| 0 | 1 || 1 |
| 1 | 0 || 1 |
| 1 | 1 || 0 |
- : unit = ()

# tablebin NOR;;
| 0 | 0 || 1 |
| 0 | 1 || 0 |
| 1 | 0 || 0 |
| 1 | 1 || 0 |
- : unit = ()
```

CHAPITRE IX

Circuits logiques

1. Circuits combinatoires

1.1. Définitions

Un circuit logique est un dispositif physique (en général électronique) muni d'entrées et de sorties n'ayant que deux états stables notés 0 et 1. Les états des entrées sont imposées par l'extérieur, les états des sorties se déduisent des états d'entrées.

Définition (Circuit combinatoire)

Un circuit électronique est dit *combinatoire* si l'état de ses sorties à un instant donné t ne dépend que de l'état des entrées à cet instant. Un circuit non combinatoire est dit *séquentiel*.

1.a

Remarque : notre étude, très limitée, se limitera à quelques circuits combinatoires.
Remarque : il suffit pour cela que le circuit ne comporte pas de boucle, ce que nous supposerons en pratique.
Remarque : nous allons voir comment les circuits logiques permettent d'implémenter physiquement les propositions logiques. D'ailleurs, il suffit de les implémenter à équivalence logique près.

1.2. Portes logiques élémentaires

Définition (Circuit élémentaire)

Un circuit logique est dit *élémentaire* s'il correspond à un connecteur logique élémentaire $\neg, \vee, \wedge, \text{NOR}, \text{NAND}, \text{XOR}$.

1.b

Remarque : sur les circuits logiques élémentaires à plusieurs entrées (préciser la situation dans le cas de connecteurs non associatifs).
Remarque : nous avons évité l'emploi de circuits élémentaires correspondant à des opérateurs non commutatifs, comme l'implication.

1.3. Propositions logiques et circuits combinatoires

Un circuit combinatoire est constitué en assemblant (correctement) des circuits élémentaires.
Remarque : sur la notation des circuits logiques (point gras lorsque les fils se croisent). Plusieurs sorties possibles.

Bien entendu, on peut associer à toute proposition logique un circuit la représentant physiquement. Il suffit de l'écrire avec les connecteurs logiques autorisés, et de partir de sa représentation arborescente.
Remarque : {NOR} et {NAND} étant des systèmes complets de connecteurs, ces portes logiques sont appréciées en électronique.

Exemple (Réalisation d'un circuit logique correspondant à l'implication, à l'équivalence.)

i

Définition (Profondeur d'un circuit)

La *profondeur* d'un circuit est le nombre maximal de portes traversées pour passer d'une entrée à une sortie.

1.c

Remarque : en utilisant une forme normale d'une proposition, on constate que toute proposition logique peut être représentée par un circuit logique de profondeur au plus 3, en acceptant des portes « et » et « ou » à plusieurs entrées.

2. Tableaux de Karnaugh

Nous avons vu comment définir un circuit représentant toute proposition logique, grâce aux formes normales (qui conservent tout leur intérêt syntaxique). Cependant, cela conduit à une inflation du nombre de portes, ou du nombre d'entrées de ces portes.

On le constate bien sur un exemple simple comme $a \Rightarrow b$: c'est équivalent à $\neg a \vee b$, alors que la forme normale disjonctive est $ab + \bar{a}b + \bar{a}\bar{b}$.

Il est donc légitime de chercher une version simplifiée de notre proposition logique. On peut pour ce faire utiliser les *tableaux de Karnaugh*.

Nous commençons par introduire les nouvelles notations $\bar{a}$, ab, $a+b$ et $a \otimes b$ pour $\neg a, a \wedge b, a \vee b$ et $a \operatorname{XOR} b$ respectivement (attention, ces notations peuvent porter à confusion).

Définition (Monôme)

Un *monôme* en les variables $a_1, \ldots, a_n$ est une conjonction de littéraux sur ces variables.

2.a

Remarque : c'est donc ce que nous avons appelé clause duale précédemment.

Définition (Impliquant)

Soit P une proposition sur les variables $a_1, \ldots, a_n$. On dit qu'un monôme M construit sur ces variables est un *impliquant* de P si $M \Rightarrow P$ est une tautologie.

2.b

Exemple (D'impliquant)

Les impliquants de $a \Rightarrow b$ sont $\bar{a}$, b, $\bar{a}b$, ab, $\bar{a}\bar{b}$.

i

Définition (Impliquant premier)

Soit P une proposition logique. Un impliquant M de P est dit *premier* si aucun monôme obtenu en supprimant un littéral de M n'est un impliquant de P.

2.c

Remarque : autrement dit, en appelant E l'ensemble des impliquants de P, et en définissant une relation d'ordre « est un sous-mot de », les impliquants premiers sont les éléments minimaux pour cette relation d'ordre. E est évidemment bien fondé, puisque fini.

Proposition (Impliquants premiers)

Toute proposition logique est équivalente à la disjonction de ses impliquants premiers.

2.a

Exemple (Impliquants premiers de l'implication)

Les impliquants premiers de $a \Rightarrow b$ sont $\bar{a}$ et b.

ii

Exemple (Impliquants premiers avec trois variables.)

Les impliquants premiers de $a \wedge (b \Rightarrow c)$ sont $a\bar{b}$ et ac.

iii

Comment trouver les impliquants premiers ? Dans le cas de deux à quatre variables, on peut utiliser un *tableau de Karnaugh*.

Remarque : il faut éventuellement utiliser une symétrie « cylindrique » ou « torique ».

Exemple (Tableau de Karnaugh de l'implication)

iv

Exemple (Tableau de Karnaugh avec trois variables)

$P = (a \vee b) \Rightarrow (a \wedge b \wedge c))$ a pour tableau de Karnaugh

$a \backslash (bc)$	00	01	11	10
0	1	1	0	0
1	0	0	1	0

donc P est logiquement équivalente à $\bar{a}\bar{b} + abc$.

v

Exemple (Tableau de Karnaugh avec quatre variables.)

Tableau de Karnaugh de P prenant la valeur 1 si et seulement si au plus une de ses quatre variables prend la valeur 1.

vi

3. Additionneurs

Pour finir, on considère un exemple de circuit arithmétique. On utilise l'écriture binaire d'un entier naturel. On cherche à implémenter l'addition.

3.1. Additionneur 1-bit

Demi-additionneur (sans réflexion). L'unité est $a \otimes b$, la retenue ab.
On en déduit un schéma pour le demi-additionneur 1-bit.
Additionneur 1-bit (trois entrées, deux sorties). L'unité est $a \otimes b \otimes r$, la retenue $ab + r(a \otimes b)$.
On en déduit un schéma pour l'additionneur 1-bit.

3.2. Additionneur n-bits

Version bête, en série, avec un demi-additionneur suivi d'additionneurs.

Remarque : il existe une version bien plus efficace fondée sur une stratégie diviser pour régner.

Sixième partie

Exercices

FEUILLE DE TD 1

Premiers pas en Caml

Exercice 1 (Simple calcul)

Calculer $\sqrt{1+\sqrt{17}}$.

Exercice 2 (Utilisation d'identificateurs locaux) — 0

1 Calculer $\frac{\cos(\ln(3))+\sin(\ln(2))}{\cos^3(\ln(3))-\sin^2(\ln(2))}$.

2 Calculer $12^{\left(\ln(12)^{\ln(\ln(12))}\right)}$ par déclarations emboîtées.

Exercice 3 (Autour de function) — 0

1 Définir la fonction tangente hyperbolique.

2 Définir l'opération d'addition des entiers sous forme curryfiée avec function.

Exercice 4 (Recherche de type)

1 Donner le type de 2, 4, de 2., 2, et de (3, ('a', 4.5), 3).

2 Donner le type des fonctions **let** evalue_en_2 f = f 2 et evalue x f = f x

Exercice 5 (Structure de groupe produit) — 2

Proposer une fonction curryfiée qui à deux lois de composition interne associe la loi produit.

Exercice 6 (Ordre lexicographique) — 2

Proposer une fonction permettant de comparer deux couples d'entiers pour l'ordre lexicographique.

Exercice 7 (Filtrage) — 2

1 Programmer le et logique et l'implication sous forme curryfiée par filtrage.

2 Reconstruire la structure de contrôle conditionnelle (if ... then ... else) par filtrage.

Exercice 8 (Taux d'accroissement) — 0

Définir une fonction taux de type (float −> float) −> float −> float −> float associant le taux d'accroissement de f entre x et y.

Exercice 9 (Composition) — 2

1 Définir l'opération de composition o, que l'on peut mettre sous forme infixe avec la directive #infix "o" (il faut mettre le dièse).

2 Profitez-en pour redéfinir la fonction qui à f associe $x \mapsto f(x+2)+2$.

Exercice 10 (Curryfication) — 1

Définir des fonctions de curryfication et de décurryfication.

Exercice 11 (Détermination compliquée d'un type) — 4

Donner le type des fonctions **let** p x y = y x x et **let** p' y = p p y.

FEUILLE DE TD 2

Premiers pas en récursivité et itération

Exercice 1 (Références)

Que va produire la séquence suivante ?

```
let a = "cou";;
let b = a ^ a;;
a.[1] <- 'r';;
b;;
```

Exercice 2 (Réécriture de fonctions simples)

Réécrire les fonctions hd tl @ mem index.

Exercice 3 (Somme de termes)

1 Écrire une fonction de type int vect -> int renvoyant la somme des termes d'un vecteur d'entiers.

2 Écrire une fonction de type int list -> int renvoyant la somme des termes d'une liste d'entiers.

Exercice 4 (Fonction puissance)

Écrire une fonction puissance curryfiée prenant en arguments deux entiers naturels a et b, et renvoyant a^b :

1 De manière impérative.

2 De manière récursive.

Exercice 5 (Accès à un terme d'une liste)

Écrire une fonction acces telle que acces liste i renvoie le i-ème terme de liste.

Exercice 6 (Enlever le dernier liste d'une liste)

Écrire une fonction qui ôte le dernier terme d'une liste.

Exercice 7 (Éviter les répétitions)

Écrire une fonction qui supprime les répétitions consécutives dans une liste.

Exercice 8 (Ajout de zéros)

1 Écrire la fonction qui ajoute un zéro entre deux termes d'une liste.

2 Écrire la fonction qui ajoute un zéro entre les deux premiers termes d'une liste, puis deux entre le deuxième et le troisième, etc. (elle laisse invariant une liste d'au plus un terme).

Exercice 9 (Recherche dans un tableau)

1 Écrire une fonction recherche de type 'a vect −> 'a −> bool qui détermine la présence d'un élément dans un tableau.

2 Écrire à nouveau une telle fonction qui rattrape l'exception Invalid_argument "vect_item" et une exception trouve que vous aurez préalablement définie.

FEUILLE DE TD 3

Itération

Exercice 1 (Maximum d'un tableau)

1 Définir une fonction max_tableau prenant en argument un tableau d'éléments comparables pour $<$, renvoyant son plus grand élément.

2 Prouver sa correction.

3 Abstraire pour généraliser cet algorithme.

Exercice 2 (Plus petit diviseur premier)

Écrire une fonction qui à un entier $n \geqslant 2$ associe son plus petit diviseur premier. Prouver sa terminaison et sa correction.

```
let f n =
        let x = ref n and y = ref n in
                while not (!y = 0) do
                        x := !x + 2;
                        y := !y - 1;
                done;
        !x;;
```

Exercice 3 (Preuve d'un programme itératif)

1 Que fait la fonction f ? Prouvez-le.

2 Que se passe-t-il si on remplace la ligne x := !x + 2 par x := !x ∗ 2 ? Quel invariant de boucle donneriez-vous dans ce cas ?

Exercice 4 (Bézout itératif)

1 Donner (et prouver la correction et terminaison d')une fonction impérative permettant le calcul du pgcd de deux entiers.

2 Donner (et prouver la correction et terminaison d')une fonction impérative donnant un couple de Bézout pour deux entiers.

Exercice 5 (Algorithme de Hörner)

1 Écrire une procédure evaluation_simple de type float vect -> float -> float permettant d'évaluer en un réel x un polynôme p (à coefficients réels) dont on a entré les coefficients sous forme de tableau (l'indice de l'élément correspondant à l'indice du monôme).

2 Trouver un algorithme d'évaluation d'un polynôme en un point, utilisant moins d'opérations que le premier. C'est (peut-être) la méthode de Hörner.

```
let myst x y = let m = ref 0 and a = ref x and b = ref y in
        while !b >0 do
                if !b mod 2 = 1 then m := !m + !a;
                a := !a * 2;
                b := !b / 2;
                done;
        !m;;
```

Exercice 6 (Une fonction mystérieuse)

Que fait la fonction myst ? Prouvez-le.

FEUILLE DE TD 4

Terminaison et correction des fonctions récursives

Exercice 1 (Ordre produit) — 0

Montrer que l'ordre produit sur $\mathbb{N}^2$, défini, pour tous $(m,n),(m',n') \in \mathbb{N}^2$, par

$$((m,n) \preceq (m',n')) \Leftrightarrow ((m \leqslant m') \wedge (n \leqslant n'))$$

est bien fondé.

```
# let rec morris = fun
        | 0 _ -> 1
        | m n -> morris (m - 1) (morris m n);;
```

Exercice 2 (Fonction de Morris) — 0

La fonction morris ci-dessus termine-t-elle sur $\mathbb{N}^2$?

```
# let rec ackermann = fun
        | 0 p -> p + 1
        | n 0 -> ackermann (n - 1) 1
        | n p -> ackermann (n - 1) (ackermann n (p - 1));;
```

Exercice 3 (Fonction d'Ackermann) — 1

1 Montrer que la fonction ackermann ci-dessus termine sur $\mathbb{N}^2$.
2 Calculer ackermann 1 p, ackermann 2 p et ackermann 3 p.
3 Montrer que pour tout $(n,p) \in \mathbb{N}^2$, ackermann n p $>$ p.

```
# let rec mccarthy = function
        | n when n > 100 -> n - 10
        | n -> mccarthy (mccarthy (n + 11));;
```

Exercice 4 (Fonction de McCarthy) — 2

Montrer que la fonction mccarthy termine sur $\mathbb{Z}$ et calculer sa valeur en tout entier.

FEUILLE DE TD 5

Récursivité terminale

Exercice 1 (Récursivité croisée)

Soit (u_n) et (v_n) les suites définies par $u_0 = a, v_0 = b \in \mathbb{R}_+^*$ (a et b sont préalablement déclarés) et, pour tout $n \in \mathbb{N}$:

$$u_{n+1} = \frac{u_n + v_n}{2}, \quad v_{n+1} = \sqrt{u_n v_n}.$$

Définir ces suites par récursivité croisée.

Exercice 2 (Suite récurrente version terminale)

On suppose disposer d'une fonction f de type `float -> float`. On fixe un réel a, et on considère la suite (u_n) de terme initial $u_0 = a$ et d'itératrice f.
Écrire une fonction récursive terminale permettant le calcul de tout terme de cette suite.

Exercice 3 (Somme des termes d'une liste)

Écrire une fonction récursive terminale qui calcule la somme des termes d'une liste d'entiers.

Exercice 4 (Opérations classiques sur les listes chaînées)

Proposer des versions récursives terminales de l'inversion d'une suite, de la concaténation de deux suites.

Exercice 5 (Aplatissement de listes)

1 Écrire une fonction `aplatir : 'a list list -> 'a list` qui à une liste de listes associe leur concaténée.

2 Proposer une version récursive terminale de l'aplatissement.

FEUILLE DE TD 6

Induction structurelle

Exercice 1 (Fonctions sur des ensembles inductifs)

1 Écrire la fonction pred donnant le prédécesseur d'un objet de type entier (type défini dans le poly).

2 Généraliser l'exemple de la somme de deux nombres (le type nombre est défini dans le poly) par abstraction de l'opération.

3 Écrire une fonction permettant le calcul du nombre de feuilles d'un arbre binaire donné.

4 Écrire la fonction arbre_fibo de type int -> arbre_binaire envoyant l'entier n sur l'arbre binaire des appels à la suite de Fibonacci dans la programmation maladroite de cette suite.

Exercice 2 (Définition d'un type calcul)

Définir un type calcul sous forme d'arbres binaires, comprenant les entiers, les lois d'addition et de multiplication. Définir une fonction qui à un tel calcul associe sa valeur.

FEUILLE DE TD 7

Complexité

Exercice 1 (Tours de Hanoï) — 1

On dispose de n rondelles $r_1, \ldots, r_n$, de diamètres $1, \ldots, n$, évidées en leur centre, et de trois piquets, sur lesquels on peut enfiler les rondelles. Au départ, les rondelles sont toutes empilées sur le premier piquet, selon la taille de leurs diamètres (la plus large en bas). On cherche à déplacer ces rondelles du premier vers le troisième piquet, en respectant les lois suivantes :
– chaque étape consiste en un déplacement d'une unique rondelle vers une autre ;
– on ne peut pas poser une rondelle sur une rondelle plus petite.
C'est le fameux problème des tours de Hanoï (ou des tours de Babel à Plovdiv, une charmante ville bulgare).

1 Proposer une procédure résolvant ce problème.

2 Évaluer la complexité de votre algorithme en nombre de déplacements.

```
# let cherche elt t = let i = ref 0 and trouve = ref false in
        while (!i < vect_length t  & !trouve = false) do
                if elt = t.(!i) then trouve := true else i := !i +1
                done;
        !trouve;;
cherche : 'a -> 'a vect -> bool = <fun>

# cherche 3  [| 5; 6; 9|];;
- : bool = false
```

Exercice 2 (Recherche dans un tableau) — 2

1 On effectue encore une recherche d'un élément dans un tableau, en supposant cette fois-ci que les éléments du tableau de taille n sont des entiers de 1 à p, pouvant apparaître plusieurs fois, avec une probabilité uniforme. Calculer la complexité moyenne de l'algorithme.

2 On revient au cas général, et on suppose le tableau rangé dans l'ordre croissant. Proposer un algorithme de recherche dans ce tableau, de complexité logarithmique.

Exercice 3 (Sommation de relations de comparaison) — 2

Montrer que si $c_n = \mathrm{O}(n^\alpha)$, alors $\sum_{1 \leqslant k \leqslant n} c_k = \mathrm{O}(n^{\alpha+1})$.

Exercice 4 (La traversée du mur) — 3

Vous vous trouvez face à un mur infini, sur lequel se trouve une unique porte permettant de passer de l'autre côté. Vous ne savez pas si cette porte se situe à votre gauche ou à votre droite. De plus, vous ne pouvez la trouver que si vous vous retrouvez en face d'elle. Proposer une stratégie de complexité linéaire pour trouver cette porte.

```
# let rec u1 = function
        | 0 -> 1.0
        | n -> let s = ref 0.0 in
                for k = 1 to n do
                        s := !s +. u1 (n - k) /. float_of_int k
                done;
                !s;;
u1 : int -> float = <fun>

# u1 3;;
- : float = 2.33333333333

# let u2 n = let v = make_vect (n + 1) 0.0 in
        v.(0) <- 1.0;
        for p = 1 to n do
                for k = 1 to p do
                        v.(p) <- v.(p) +. v.(p - k) /. float_of_int k
                done
        done;
        v.(n);;
u2 : int -> float = <fun>

# u2 3;;
- : float = 2.33333333333
```

Exercice 5 (Comparaison de complexités) — 3

On veut calculer les termes de la suite (u_n) définie par :

$$u_0 = 1, \quad \forall n \in \mathbb{N}^*, u_n = \frac{u_{n-1}}{1} + \frac{u_{n-2}}{2} + \cdots + \frac{u_0}{n}.$$

Les deux programmes ci-dessus calculent u_n : comparer leurs complexités temporelles et spatiales.

FEUILLE DE TD 8

Diviser pour régner

Exercice 1 (Suite de Fibonacci, encore et toujours) — 2

En utilisant une stratégie diviser pour régner, montrer comment obtenir le terme d'indice n de la suite de Fibonacci en complexité logarithmique.
Indication : on pourra utiliser la multiplication matricielle.

Exercice 2 (Multiplication polynomiale de Knuth) — 2

On cherche à implémenter la multiplication rapide de Knuth (ou de Karatsuba). On représente les polynômes d'entiers comme des tableaux d'entiers, l'indice correspondant à la puissance de l'indéterminée.

1 Définir l'addition polynomiale (curryfiée, comme les fonctions suivantes).

2 Définir la multiplication par un scalaire d'un polynôme.

3 Définir la soustraction de polynômes.

4 Définir l'addition de trois polynômes, la soustraction de deux polynômes à un troisième.

5 Définir une fonction décalage de type int −> int vect −> int vect tel que decalage n p soit le polynôme $X^n p(X)$.

6 Pour simplifier la programmation, on supposera que les tableaux utilisés auront pour taille une même puissance de deux.
Définir la multiplication polynomiale grâce à l'algorithme de Knuth.

7 Comment utiliser cette multiplication pour effectuer la multiplication d'entiers en base b « arbitrairement grands », représentés par des tableaux d'entiers entre 0 et $b-1$?

FEUILLE DE TD 9

Tris

Exercice 1 (Recherche dichotomique) — 1

Proposer une fonction de recherche d'un élément dans un tableau trié, fondée sur la stratégie diviser pour régner, en complexité logarithmique en nombre de comparaisons.

Exercice 2 (Insertion dans une liste triée) — 1

1 Proposer une fonction insérant (à la bonne place) un élément dans un vecteur trié.
2 Même question pour une liste chaînée.

Exercice 3 (Fusion de listes triées) — 1

1 Proposer une fonction fusionnant deux vecteurs triés.
2 Même question pour les listes chaînées.

Exercice 4 (Tri par insertion) — 1

Le tri par insertion est le tri du joueur de cartes (on prend les cartes une par une, en l'insérant au bon endroit).
1 Proposer un tri par insertion pour les vecteurs.
2 Même question pour les listes chaînées.

Exercice 5 (Tri par sélection) — 1

Le tri par sélection consiste à chercher le plus petit élément, puis le suivant, etc.
1 Proposer un tri par sélection pour les vecteurs.
2 Même question pour les listes chaînées.

Exercice 6 (Tri bulle) — 0

L'algorithme du tri bulle proposé dans le cours effectue un nombre constant de comparaisons : comment améliorer l'algorithme afin qu'il en effectue moins en moyenne ?

Exercice 7 (Tri par insertion en mieux) — 0

L'algorithme du tri par insertion pour les vecteurs suit à la lettre la démarche de cette méthode de tri. En programmant par effets de bord, proposer un algorithme de tri par insertion plus efficace.

Exercice 8 (Stabilité des algorithmes de tri) — 2

Un algorithme de tri est dit *stable* s'il n'échange jamais des termes égaux en des indices distincts. Parmi les algorithmes proposés, lesquels sont-ils stables ? Comment rendre tout algorithme de tri stable ?

FEUILLE DE TD 10

Listes

```
# let rec image_miroir_bof = function
        | [] -> []
        | a :: q -> image_miroir_bof q @ [a];;
image_miroir_bof : 'a list -> 'a list = <fun>

# image_miroir_bof [1; 4; 9; 13];;
- : int list = [13; 9; 4; 1]
```

Exercice 1 (Image miroir) — 1

La fonction image_miroir_bof ci-dessus renverse une liste.

1 Calculer sa complexité en nombre de conses, la complexité de l1 @ l2 étant $|l1|$.

2 Proposer un algorithme de complexité linéaire effectuant l'image miroir.

Exercice 2 (Versions itératives des programmes usuels sur les listes) — 2

En utilisant le type Liste défini dans le cours, proposer des versions impératives des algorithmes fondamentaux sur les listes.

Exercice 3 (Insertion et suppression) — 2

On travaillera avec le type Liste défini dans le cours.

1 Proposer des fonctions d'insertion et de suppression d'un terme dans une liste à une position donnée.

2 Proposer une fonction d'insertion d'un élément dans une liste triée.

Exercice 4 (Programmes itératifs sur les listes) — 3

1 Définir une fonction map_bis, de type (’a −> ’b) −> ’a list −> ’b list telle que map_bis f liste renvoie la liste des applications de f à liste . Cette fonction, sous le nom map, est une primitive Caml.

2 La fonctionnelle it_list , de type (’a −> ’b −> ’a) −> ’a −> ’b list −> ’a, est définie informellement par

it_list f b [e1; e2; ... ; en] = (f (... (f (f b e1) e2) ...) en)

i Programmer cette fonctionnelle.

ii Utiliser cette fonctionnelle pour définir une fonction qui à une liste d’entiers associe la somme de ses termes. Faire de même pour le produit.

3 La fonctionnelle list_it , de type (’a −> ’b −> ’b) −> ’a list −> ’b −> ’b est définie informellement par

list_it f [e1; e2; ... ; en] b = f e1 (f e2 (... (f en b) ...))

i Programmer cette fonctionnelle.

Utiliser cette fonctionnelle pour

ii définir des fonctions somme et produit comme ci-dessus.

iii programmer la concaténation.

iv programmer map_bis.

Exercice 5 (Parties d’une liste) — 2 à 4

Si $L = (a_1, \ldots, a_n)$ est une liste, alors, pour tous i et j tels que $1 \leqslant i \leqslant j \leqslant n$, $(a_k)_{k\in[\![i,j]\!]}$ est une *sous-liste* de L. Nous disons aussi que la liste vide ε est sous-liste de toute liste.
Une *sous-séquence* de la liste L est une liste formée en éliminant zéro ou plusieurs éléments de L. Les éléments restants, qui forment la sous-séquence, doivent apparaître dans le même ordre que L, mais les éléments de la sous-séquence ne sont pas nécessairement consécutifs dans L.
Un *préfixe* d’une liste est une sous-liste commençant par la tête de la liste (*i.e.* $i = 1$ ci-dessus).
Un *suffixe* d’une liste est une sous-liste qui se termine à la fin de la liste (*i.e.* $j = n$ ci-dessus).

1 Programmer des prédicats permettant de tester si une liste est une sous-liste, un préfixe, un suffixe, une sous-séquence de la seconde liste passée en argument (on travaillera avec les listes chaînées de Caml).

2 Dénombrer le nombre maximal de préfixes, suffixes, sous-listes, sous-séquences d’une liste de longueur n.

3 Décrire les listes commutant pour la concaténation.

4 On travaille avec des listes dont les termes appartiennent à un alphabet $\mathcal{A}$ de cardinal N.

i Dénombrer le nombre de listes de taille n.

ii Dénombrer le nombre de listes de tailles respectives 6 et 10 commutant pour la concaténation.

5 On considère une liste de listes d’éléments de $\mathcal{A}$. On vous donne une telle liste, mais on a omis toutes les parenthèses permettant de distinguer les éléments de cette liste.

i Dénombrer les listes possibles n’admettant pas la liste vide pour élément.

ii Dénombrer les listes possibles de taille n.

FEUILLE DE TD 11

Piles

Exercice 1 (Rotation d'une pile) — 0

La rotation d'une pile consiste à prendre la première assiette, et à la mettre tout en bas. Programmer cette rotation pour l'implémentation détaillée dans le cours.

Exercice 2 (EAP pour une loi associative) — 2

On considère une loi de composition interne associative $\star$ sur un ensemble E. Soit $a, b, c, d \in E$. Proposer des expressions algébriques permettant le calcul de $a \star b \star c \star d$.

```
# type operateur = Plus | Moins | Mult | Divise;;
Type operateur defined.

# type Lexeme = Var of float | Op of operateur;;
Type Lexeme defined.
```

Exercice 3 (Expressions algébriques postfixées) — 1

On utilise la structure de données pile donnée en cours. On cherche à évaluer une expression algébrique postfixée (EAP). Pour cela, nous la verrons comme une liste de lexèmes (voir ci-dessus). Proposer une évaluation d'EAP (syntaxiquement correcte) à l'aide d'une pile.

Exercice 4 (Bon parenthésage) — 1

À l'aide d'une pile, vérifier le bon parenthésage (en un sens évident) d'un mot constitué de parenthèses uniquement.

Exercice 5 (Pile et expression préfixée) — 2

Comment utiliseriez-vous une pile pour évaluer des expressions préfixées ?

Exercice 6 (Expressions algébriques) — 2

1 Programmer une fonction permettant de passer d'une expression algébrique infixe à une postfixe.
2 Programmer une fonction réciproque.

Exercice 7 (D'autres représentations des piles) — 3

1 Implémenter les autres structures de données représentant le type de données pile vues en cours. On programmera en particulier le test d'égalité.

2 Reprendre les exercices précédents avec ces structures.

FEUILLE DE TD 12

Logique (syntaxe)

Exercice 1 (Longueur d'une représentation linéaire) — 2

Soit P une proposition sous forme linéaire infixe (la négation n'ajoutant pas de parenthèses), $b(P)$ le nombre d'opérateurs binaires de P, $n(P)$ le nombre de symboles de négation dans P. Montrer que la longueur $|P|$ de P vaut $4b(P) + n(P) + 1$.

Exercice 2 (Encadrement de la longueur d'une proposition à l'aide de sa hauteur) — 2

Étant donné une proposition P de hauteur n, montrer que la longueur de sa représentation infixe parenthésée (la négation n'ajoutant pas de parenthèses) est comprise entre $n + 1$ et $2^{n+2} - 3$.

Exercice 3 (Test de bonne écriture d'une expression logique postfixée) — 2

On implémente la caractérisation par Lukasiewicz de la syntaxe d'une expression logique postfixée. On utilise le type binaire du cours.

1 Définir un type lexème contenant la constante Neg, les variables booléennes Constante, chaîne de caractères Varia et Binaire de type binaire.

2 Définir une fonction EAP qui à une proposition logique (de type formule du cours) associe sa représentation linéaire postfixée, sous forme de liste de lexèmes.

3 Définir une fonction poids qui à une liste de lexèmes associe son poids (au sens vu en cours).

4 Définir un test de bonne syntaxe d'une liste de lexèmes.

FEUILLE DE TD 13

Logique (sémantique)

Exercice 1 (Mise sous forme conjonctive ou disjonctive) — 0

Mettre sous formes conjonctive et disjonctive :
1 $(a \Rightarrow b) \Rightarrow (\neg c \wedge b)$.
2 $(a \Leftrightarrow b) \wedge (a \Rightarrow (\neg c \vee d))$.

Exercice 2 (De la fonction logique à une proposition logique) — 0

Montrer que toute fonction logique $\{0,1\}^n \to \{0,1\}$ est la fonction associée à une proposition logique.

Exercice 3 (Simplification d'une proposition) — 0

Simplifier la proposition $(\neg(a \wedge b)) \wedge (a \vee \neg b) \wedge (a \vee b)$.

Exercice 4 (Tables de vérité, formes normales) — 0

Écrire la table de vérité, les formes normales conjonctives et disjonctives de
1 $\neg((a \Rightarrow b) \Leftrightarrow c)$.
2 $((a \Rightarrow b) \vee c) \Leftrightarrow (a \vee c)$.
3 $(\neg(a \wedge b)) \vee (\neg c \wedge b)$.

Exercice 5 (Système complet de connecteurs) — 2

Un système Σ de connecteurs est dit *complet* si toute proposition logique est équivalente à une proposition construite avec des connecteurs de Σ uniquement. Montrer que $\{\neg, \wedge, \vee\}$, $\{NAND\}$ et $\{NOR\}$ sont des systèmes complets de connecteurs.

FEUILLE DE TD 14

Logique (circuits)

Exercice 1 (Additionneur 1-bit) — 0

1 Écrire un additionneur 1-bit à partir des formes normales disjonctives pour l'unité et la retenue, en utilisant des portes logiques « et » et « ou » à plusieurs entrées et la négation.

2 Faire de même en utilisant les tableaux de Karnaugh.

Exercice 2 (Exemple d'utilisation d'un tableau de Karnaugh) — 0

On considère une proposition logique P de tableau de Karnaugh

$(ab)\backslash(cd)$	00	01	11	10
00	1	0	1	1
01	0	0	0	0
11	0	1	0	0
10	1	0	0	1

1 Donner la forme normale disjonctive de P.

2 Donner les impliquants premiers de P.

3 Construire un circuit logique représentant P.

Exercice 3 (Un autre exemple d'utilisation d'un tableau de Karnaugh) — 0

1 Soit f la fonction booléenne de quatre variables telle que $f(a, b, c, d)$ renvoie 1 si et seulement si $abcd$ est l'écriture binaire d'un entier inférieur ou égal à 10. Construire le tableau de Karnaugh correspondant, en déduire une proposition de fonction booléenne f, et construire un circuit logique correspondant à f.

2 Faire de même avec $f(a, b, c, d) = 1$ si et seulement si $abcd$ est l'écriture binaire d'un nombre congru à 1 modulo 3.

Septième partie

Solution des exercices

FEUILLE DE TD 1

Solution de premiers pas en Caml

```
# sqrt (1. +. sqrt 17.);;
- : float = 2.26342784856

# (a +. b) /. (a ** 3. -. b ** 2.) where a = cos (log 3.) and b = sin (log 2.);;
- : float = -3.48143281984
#let x = 12. in
        let y = log x in
                let z = log y in
                        x ** (y ** z);;(* x ** y ** z donnerait le bon résultat *)
- : float = 295.982482437

# ((x ** (y ** z) where z = log y) where y = log x) where x = 12.;;
(* Noter la nécessité du parenthésage pour les where imbriqués *)
- : float = 295.982482437

# let tanh x = (aux -. 1.) /. (aux +. 1.) where aux = exp (2. *. x);;
tanh : float -> float = <fun>

# tanh 1.;;
- : float = 0.761594155956

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

# 2, 4;;
- : int * int = 2, 4

# 2., 2;;
- : float * int = 2.0, 2

# (3, ('a', 4.5), 3);;
- : int * (char * float) * int = 3, ('a', 4.5), 3

# let evalue_en_2 f = f 2;;
evalue_en_2 : (int -> 'a) -> 'a = <fun>

# let evalue x f = f x;;
evalue : 'a -> ('a -> 'b) -> 'b = <fun>

# let prod loi1 loi2 (x1,x2) (y1,y2) = (loi1 x1 y1, loi2 x2 y2);;
prod :
 ('a -> 'b -> 'c) -> ('d -> 'e -> 'f) -> 'a * 'd -> 'b * 'e -> 'c * 'f =
 <fun>

# let ordre (x,y) (x',y') = (x < x') || ((x = x') && (y < y'));;
ordre : 'a * 'b -> 'a * 'b -> bool = <fun>
```

```
# let et_logique a b = match a with
        | false -> false
        | _ -> b;;
et_logique : bool -> bool -> bool = <fun>

# let implication = fun
        | true false -> false
        | _ _ -> true;;
implication : bool -> bool -> bool = <fun>

# let if_then_else b v v'= match b with
        | true -> v
        | _ -> v';;
if_then_else : bool -> 'a -> 'a -> 'a = <fun>

# if_then_else (0=1) 1 2;;
- : int = 2

# let taux f x y = (f(y) -. f(x))/.(y -. x);;
taux : (float -> float) -> float -> float -> float = <fun>

# let o f g x= f (g x);;
o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# #infix "o";;

# let decale_de_2 f = (add 2) o f o (add 2);;
decale_de_2 : (int -> int) -> int -> int = <fun>

# let triple x = 3 * x;;
triple : int -> int = <fun>

# let h =decale_de_2 triple;;
h : int -> int = <fun>

# h 2;;
- : int = 14

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

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

# let p x y = y x x;;
p : 'a -> ('a -> 'a -> 'b) -> 'b = <fun>

# let p' y = p p y;;
p' :
  (('a -> ('a -> 'a -> 'b) -> 'b) -> ('a -> ('a -> 'a -> 'b) -> 'b) -> 'c) ->
  'c = <fun>
```

FEUILLE DE TD 2

Solution de premiers pas en récursivité et itération

```
# let a = "cou";;
a : string = "cou"

# let b = a ^ a;;
b : string = "coucou"

# a.[1] <- 'r';;
- : unit = ()

# b;;
- : string = "coucou"

# let tete = function
        | [] -> failwith "Pas de tête pour la liste vide"
        | a :: _ -> a;;
tete : 'a list -> 'a = <fun>

# let queue = function
        | [] -> failwith "Pas de queue pour la liste vide"
        | _ :: q -> q;;
queue : 'a list -> 'a list = <fun>

# tete [1; 2; 4];;
- : int = 1

# queue [1; 2; 4];;
- : int list = [2; 4]

# let rec concat l1 l2 = match l1 with
        | [] -> l2
        | a :: q -> a :: concat q l2;;
concat : 'a list -> 'a list -> 'a list = <fun>

# concat [1; 2; 4] [3; 4; 5];;
- : int list = [1; 2; 4; 3; 4; 5]

# let rec membre elt l = match l with
        | [] -> false
        | a :: q -> (elt = a) or (membre elt q);;
membre : 'a -> 'a list -> bool = <fun>

# membre 3 [5; 6; 3];;
- : bool = true

# membre 7 [5; 6; 3];;
- : bool = false

# let rec indice elt l = match l with
        | [] -> failwith "Element non dans la liste"
```

```
        | a :: q -> if elt = a then 1 else indice elt q + 1;;
indice : 'a -> 'a list -> int = <fun>

# indice 3 [5; 6; 3];;
- : int = 3

# indice 7 [5; 6; 3];;
Uncaught exception: Failure "Element non dans la liste"

# let somme_vect t = let s = ref 0 in
        for i = 0 to vect_length t - 1 do s := !s + t.(i) done;
        !s;;
somme_vect : int vect -> int = <fun>

# somme_vect [| 3; 5; 7|];;
- : int = 15

# let rec somme_liste = function
        | [] -> 0
        | a :: q -> a + somme_liste q;;
somme_liste : int list -> int = <fun>

# somme_liste [3; 6; 9];;
- : int = 18

(* accès au i-ème terme d'une liste *)
# let rec acces liste i = match i with
        | 1 -> hd liste
        | _ -> acces (tl liste) (i-1);;
acces : 'a list -> int -> 'a = <fun>

# acces [6; 7; 8; 9] 4;;
- : int = 9

# let rec ampute = function
        | [] -> failwith "on n'ampute pas un manchot cul-de-jatte"
        | [a] -> []
        | a :: q -> a :: ampute q;;
ampute : 'a list -> 'a list = <fun>

# ampute [3; 7; 9; 11];;
- : int list = [3; 7; 9]

# let puissance_imperative a b = let p = ref 1 in for i = 1 to b do p := !p * a done; !p;;
puissance_imperative : int -> int -> int = <fun>

# puissance_imperative 2 10;;
- : int = 1024

# let rec puissance a = function
        | 0 -> 1
        | b -> a * puissance a (b - 1);;
puissance : int -> int -> int = <fun>

# puissance 2 10;;
- : int = 1024

# let rec pasderepet = function
        | [] -> []
```

```
        | [a] -> [a]
        | a :: q -> if a = hd q then pasderepet q else a :: pasderepet q;;
pasderepet : 'a list -> 'a list = <fun>

# pasderepet [1; 3; 3; 3; 5; 5; 10; 10; 10];;
- : int list = [1; 3; 5; 10]

# let rec ajoutsimple = function
        | [] -> []
        | [a] -> [a]
        | a :: q -> a :: 0 :: ajoutsimple q;;
ajoutsimple : int list -> int list = <fun>

# ajoutsimple [4];;
- : int list = [4]

# ajoutsimple [4; 8; 9; 2];;
- : int list = [4; 0; 8; 0; 9; 0; 2]

(* Fonction auxiliaire *)

# let rec ajout l n a=match n with
        | 0 -> l
        | _ -> a :: ajout l (n-1) a;;
ajout : 'a list -> int -> 'a -> 'a list = <fun>

# let rec ajoutde0 l = match l with
        | [] -> failwith "pas d'ajout pour la liste vide"
        | [a] -> [a]
        | a :: q -> a :: (ajout (ajoutde0 q) (list_length l -1) 0);;
ajoutde0 : int list -> int list = <fun>

# let bonajout l = rev (ajoutde0 (rev l));;
bonajout : int list -> int list = <fun>

# bonajout [1; 3; 4; 7; 23];;
- : int list = [1; 0; 3; 0; 0; 4; 0; 0; 0; 7; 0; 0; 0; 0; 23]

# let recherche t elt =
                let b = ref false in
                        let i = ref 0 in
                                while !i < vect_length t & !b = false do
                                        if t.(!i) = elt then b := true; i := !i + 1
                                done;
                                !b;;
recherche : 'a vect -> 'a -> bool = <fun>

# recherche [|1; 2; 3|] 4;;
- : bool = false

# exception trouve;;
Exception trouve defined.

# let recherche' t elt =
                try
                        for i = 0 to vect_length t - 1 do
                                if t.(i) = elt then raise trouve;
                                done;
                        false;
```

```
                with
                        trouve -> true;;
recherche' : 'a vect -> 'a -> bool = <fun>

# recherche' [|1; 2; 3|] 4;;
- : bool = false

# let recherche' t elt =
                try
                        for i = 0 to vect_length t do
                                if t.(i) = elt then raise trouve; done;
                                false;
                with
                        | trouve -> true
                        | Invalid_argument _ -> false;;
recherche' : 'a vect -> 'a -> bool = <fun>

# recherche' [|1; 2; 3|] 4;;
- : bool = false

# let recherche'' t elt =
        try
                let i = ref 0 in
                        while true do
                                if t.(!i) = elt then raise trouve; i := !i + 1 done;
                        false;
        with
                | trouve -> true
                | Invalid_argument _ -> false;;
recherche'' : 'a vect -> 'a -> bool = <fun>

# recherche'' [|1; 2; 3|] 4;;
- : bool = false

# recherche'' [|1; 2; 3|] 2;;
- : bool = true

# exception trouve' of int;;
Exception trouve' defined.

# let indice t elt =
                try
                        for i = 0 to vect_length t do
                                if t.(i) = elt then raise (trouve' i) done;
                with
                        | trouve' i -> print_string "trouvé␣en␣position␣";
                                print_int i; print_newline ();
                        | Invalid_argument _ -> print_string "élément␣non␣trouvé";
                                print_newline ();;
indice : 'a vect -> 'a -> unit = <fun>

# indice [|1; 2; 3|] 2;;
trouvé en position 1
- : unit = ()
```

Solution d'itération

```
# let max_tableau t = let m = ref t.(0) in
        for i = 1 to vect_length t - 1 do
                if !m < t.(i) then m := t.(i); done;
        (* invariant après la i-ème itération : !m vaut max t.(0) ... t.(i) *)
        !m;;
max_tableau : 'a vect -> 'a = <fun>

# max_tableau [|4; -5; 10; 2|];;
- : int = 10

# let max_tableau ordre t = let m = ref t.(0) in
        for i = 1 to vect_length t - 1 do
                if ordre !m t.(i) then m := t.(i); done;
        !m;;
max_tableau : ('a -> 'a -> bool) -> 'a vect -> 'a = <fun>

# let plus_petit n = let m = ref 2 in
        while n mod !m <> 0 do m := !m + 1 done;
        !m;;
plus_petit : int -> int = <fun>

# plus_petit 65;;
- : int = 5

# let f n =
        let x = ref n and y = ref n in
                while not (!y = 0) do
                        x := !x + 2;
                        y := !y - 1;
                done;
        (* invariant : x + 2 y = 3n *)
        !x;;
f : int -> int = <fun>

# let fmod n =
        let x = ref n and y = ref n in
                while not (!y = 0) do
                        x := !x * 2;
                        y := !y - 1;
                done;
        (* invariant : !x*2^(!y)=n*2^n *)
        !x;;
fmod : int -> int = <fun>

# fmod 3;;
- : int = 24

# let pgcd a b = let m = ref a and n = ref b in
        while !n <> 0 do let c = !n in (n := !m mod !n; m := c;) done;
```

```
        !m;;
pgcd : int -> int -> int = <fun>

# pgcd 10 15;;
- : int = 5

# let cmod a b q = (fst(!a) - q * fst (!b),snd(!a) - q * snd (!b));;
cmod : (int * int) ref -> (int * int) ref -> int -> int * int = <fun>

# let bezout a b = let r = ref (a,b) and uv0 = ref (1,0) and uv1 = ref (0,1) in
                   while snd(!r) <> 0 do
                   let q = fst(!r) / snd(!r) and c = !uv1 in
                           (uv1 := cmod uv0 uv1 q; uv0 := c;);
                   r := (snd(!r),fst(!r) mod snd(!r));
                   done;
                   (* invariant : fst !uv0 * a + snd !uv0 * b = fst !r
                   et
                   fst !uv1 * a + snd !uv1 * b = snd !r *)
        !uv0;;
bezout : int -> int -> int * int = <fun>

# bezout 113 15;;
- : int * int = 2, -15

# let evalue p x = let v = ref 0. and xp = ref 1. in
        for i = 0 to vect_length p - 1 do
                v := !v +. p.(i) *. !xp;
                xp := x *. !xp;
                done;
        (* invariant après le passage en boucle pour l'indice i :
        !v = somme pour k allant de 0 à i de p.k * x ^ k et !xp = x ^ (i + 1) *)
        !v;;
evalue : float vect -> float -> float = <fun>

# evalue [| 3.; 2.; 1. |] 2.;;
- : float = 11.0

# let horner p x = let m = (vect_length p) - 1 in let v = ref p.(m) in
        for i = m - 1 downto 0 do
                v := !v *. x +. p.(i)
                done;
        (* invariant après la i-ème itération :
        !v = somme pour k allant de 0 à i de p.(m - k) * x ^ (i - k) *)
        !v;;
horner : float vect -> float -> float = <fun>

# horner [| 3.; 2.; 1. |] 2.;;
- : float = 11.0

# let myst x y = let m = ref 0 and a = ref x and b = ref y in
        while !b >0 do
                if !b mod 2 = 1 then m := !m + !a;
                a := !a * 2;
                b := !b / 2;
                done;
        (* invariant : !m + !a * !b = x * y *)
        !m;;
myst : int -> int -> int = <fun>
```

```
# myst 5 9;;
- : int = 45

# let g n = let p = ref 2 in
                while (n mod !p != 0) do p := !p + 1 done;
                !p;;
g : int -> int = <fun>

# g 113;;
- : int = 113
```

FEUILLE DE TD 4

Solution de terminaison et correction des fonctions récursives

Exercice 1 (Ordre produit) — 0

Première méthode : soit (m_0, n_0) un élément donné de $\mathbb{N}^2$. L'ensemble $\{(m,n) \in \mathbb{N}^2, (m,n) \preceq (m_0,n_0)\}$ étant fini (de cardinal $(m_0+1)(n_0+1)$), il n'existe pas de suite strictement décroissante d'éléments de $\mathbb{N}^2$ pour cet ordre, qui est donc bien fondé.
Deuxième méthode : soit Ω une partie non vide de $\mathbb{N}^2$. On pose $m_0 = \min\{m \in \mathbb{N}, \exists n \in \mathbb{N}, (m,n) \in \Omega\}$ (qui existe car Ω est non vide et toute partie non vide de $\mathbb{N}$ admet un minimum). On pose alors $n_0 = \min\{n \in \mathbb{N}, (m_0,n) \in \Omega\}$. L'élément (m_0,n_0) est alors minimal dans Ω : en effet si $(m,n) \in \Omega$ vérifie $(m,n) \preceq (m_0,n_0)$, alors $m \leqslant m_0$ par définition de $\preceq$, et $m_0 \leqslant m$ par définition de m_0 : $m = m_0$. Il vient alors, par un raisonnement analogue, $n = n_0$.
Troisième méthode : supposons l'existence d'une suite infinie $((m_p,n_p))_{p\in\mathbb{N}}$ strictement décroissante d'éléments de $\mathbb{N}^2$. L'application

$$\begin{array}{rccc} \Phi : & \mathbb{N}^2 & \to & \mathbb{N} \\ & (m,n) & \mapsto & m+n \end{array}$$

étant strictement croissante, la suite $(\Phi(m_p,n_p))_{p\in\mathbb{N}}$ d'entiers naturels est strictement décroissante, ce qui est absurde.

Exercice 2 (Fonction de Morris) — 0

La fonction morris ne termine pas, puisque pour le calcul de morris 1 1 par exemple, qui n'est pas un cas de base, la fonction s'appelle d'abord pour le même argument.

Exercice 3 (Fonction d'Ackermann) — 1

On note A la fonction ackermann, en version non curryfiée.

1 Cette fonction termine, comme le montre un raisonnement par induction bien fondée, en prenant l'ordre lexicographique sur $\mathbb{N}^2$. En effet, elle termine dès que l'un des deux arguments est nul, et, si on fixe $(m_0, n_0) \in (\mathbb{N}^*)^2$, et que l'on suppose que cette fonction termine pour tout couple $(m, n) \in \mathbb{N}^2$ tel que $(m, n) \prec (m_0, n_0)$, alors $A(m_0, n_0)$ appelle A pour le couple $(m_0, (n_0 - 1)) \prec (m_0, n_0)$, puis pour un couple $(m_0 - 1, n_0')$ (pour un certain entier n_0'), strictement inférieur à (m_0, n_0).

2 Pour tout entier naturel non nul p,

$$A(1, p) = A(0, (A(1, (p - 1)))) = A(1, (p - 1)) + 1.$$

Comme $A(1, 0) = A(0, 1) = 2$, on obtient $A(1, p) = p + 2$ par une récurrence immédiate.
On observe déjà que $A(2, 0) = 3$, puis que pour tout $p \in \mathbb{N}^*$, $A(2, p) = A(2, (p - 1)) + 2$, donc $A(2, p) = 3 + 2p$ pour tout $p \in \mathbb{N}$ par une récurrence immédiate.
Par un calcul simple, $A(3, 0) = 5$, et pour tout $p \in \mathbb{N}^*$, $A(3, p) = 2\,A(3, (p-1)) + 3$, d'où $A(3, p) + 3 = 2^{p+3}$ par une récurrence immédiate.

3 On raisonne par induction bien fondée sur $\mathbb{N}^2$ muni de l'ordre lexicographique. Le résultat est vrai pour tout couple d'entiers naturels dont l'un est nul, et, si on fixe $(m, n) \in (\mathbb{N}^*)^2$, et que l'on suppose le résultat prouvé pour tous les éléments qui lui sont strictement inférieurs, alors $A(n, (p - 1)) \geqslant p$ par hypothèse d'induction, puis, toujours par hypothèse d'induction : $A(n, p) > A(n, (p - 1)) \geqslant p$, donc $A(n, p) > p$.
Le résultat est donc bien prouvé.

Exercice 4 (Fonction de McCarthy) — 2

On note M la fonction mccarthy, en version non curryfiée.
La fonction mccarthy termine clairement pour tout entier $n \geqslant 101$, et elle y vaut $n - 10$. Montrons que pour tout entier $n \leqslant 100$, $M(n) = 91$. On raisonne par l'absurde, en supposant l'existence d'un entier $n \leqslant 100$ tel que $M(n)$ ne termine pas ou diffère de 91. On choisit le plus grand entier $N \leqslant 100$ tel que $M(N)$ ne termine pas ou diffère de 91.
Si $N + 11 > 100$, alors $M(N + 11) = N + 1$, puis $M(N) = M(M(N + 11)) = M(N + 1)$, ce qui impose $N = 100$ par maximalité de N, or $M(100) = 91$, ce qui est exclu.
Si $N + 11 \leqslant 100$, alors $M(N + 11) = 91$, donc $M(N)$ termine et vaut $M(91)$. Or un calcul simple montre que $M(91) = 91$ (car $M(n) = M(n + 1)$ pour tout $n \in [\![91, 100]\!]$), ce qui est encore une fois absurde.

Solution de récursivité terminale

```
# let a = 1. and b = 3.;;
a : float = 1.0
b : float = 3.0

# let rec u = function
        | 0 -> a
        | n -> (u (n - 1) +. v (n - 1)) /. 2.
and
v = function
        | 0 -> b
        | n -> sqrt (u (n - 1) *. v (n - 1));;
u : int -> float = <fun>
v : int -> float = <fun>

# u 3;;
- : float = 1.86361756099

# v 3;;
- : float = 1.8636160055

# let rec iter a f = function
        | 0 -> a
        | n -> iter (f a) f (n - 1);;
iter : 'a -> ('a -> 'a) -> int -> 'a = <fun>

# iter 3 (function x -> 2 * x + 1) 2;;
- : int = 15

# let f x = 3. *. x +. 2.;;
f : float -> float = <fun>

# let u = iter a f;;
u : int -> float = <fun>

# u 2;;
- : float = 17.0

# let somme = let rec somme_aux accu = function
        | [] -> accu
        | a :: q -> somme_aux (a + accu) q
in somme_aux 0;;
somme : int list -> int = <fun>

# somme [1; 2; 3; 4; 5];;
- : int = 15

# let reverse = let rec reverse_temp l l' = match l' with
        | [] -> l
        | a :: q -> reverse_temp (a :: l) q
```

```
in reverse_temp [];;
reverse : '_a list -> '_a list = <fun>

# reverse [1; 2; 3; 4; 5];;
- : int list = [5; 4; 3; 2; 1]

# let rec concat_rev l = function
        | [] -> l
        | a :: q -> concat_rev (a :: l) q;;
concat_rev : 'a list -> 'a list -> 'a list = <fun>

# concat_rev [1; 2; 3] [4; 5; 6];;
- : int list = [6; 5; 4; 1; 2; 3]

# let concat l l' = concat_rev l' (rev l);;
concat : 'a list -> 'a list -> 'a list = <fun>

# concat [1; 2; 3] [4; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6]

# let concat2 =
        let rec concat_temp2 = fun
                | [] [] l -> l
                | accu (a :: q) l -> concat_temp2 (a :: accu) q l
                | (a :: q) [] l -> concat_temp2 q [] (a :: l)
        in concat_temp2 [];;
concat2 : '_a list -> '_a list -> '_a list = <fun>
# concat2 [1; 2; 3] [4; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6]

# let rec aplatir = function
        | [] -> []
        | a :: q -> concat a (aplatir q);;
aplatir : 'a list list -> 'a list = <fun>

# aplatir [[1; 2; 3; 4]; []; [5; 6]; []];;
- : int list = [1; 2; 3; 4; 5; 6]

# let aplatir_term = let rec aplatir_term_temp accu = function
        | [] -> accu
        | a :: q -> aplatir_term_temp (concat accu a) q
        in aplatir_term_temp [];;
aplatir_term : '_a list list -> '_a list = <fun>

# aplatir_term [[1; 2; 3; 4]; []; [5; 6]; []];;
- : int list = [1; 2; 3; 4; 5; 6]
```

FEUILLE DE TD 6

Induction structurelle

```
# let pred = function
        | Zero -> failwith "0 n'a pas de predecesseur"
        | succ x -> x;;
pred : entier -> entier = <fun>

# pred (succ (succ (succ Zero)));;
- : entier = succ (succ Zero)

# let operation f = fun
        | (Entier a) (Entier b) -> Entier (fst f a b)
        | (Reel a) (Entier b) -> Reel (snd f a (float_of_int b))
        | (Entier a) (Reel b) -> Reel (snd f (float_of_int a)  b)
        | (Reel a) (Reel b) -> Reel (snd f a b);;
operation :
 (int -> int -> int) * (float -> float -> float) ->
 nombre -> nombre -> nombre = <fun>

# let prod x y = x * y;;
prod : int -> int -> int = <fun>

# let prodbis x y = x *. y;;
prodbis : float -> float -> float = <fun>

# operation (prod,prodbis) (Entier 3) (Reel 5.5);;
- : nombre = Reel 16.5

# let rec nombrefeuilles = function
        | Feuille n -> 1
        | Noeud (a, branche1, branche2) -> nombrefeuilles branche1 + nombrefeuilles branche2
nombrefeuilles : arbre_binaire -> int = <fun>

# let valeurracine = function
        | Feuille n -> n
        | Noeud (a, branche1, branche2) -> a;;
valeurracine : arbre_binaire -> int = <fun>

# let rec arbre_fibo = function
        | 0 | 1 -> (Feuille 1)
        | n -> Noeud (valeurracine (arbre_fibo (n - 2)) + valeurracine (arbre_fibo (n - 1)),
                arbre_fibo (n - 2), arbre_fibo (n - 1));;
arbre_fibo : int -> arbre_binaire = <fun>

# arbre_fibo 4;;
- : arbre_binaire =
 Noeud
  (5, Noeud (2, Feuille 1, Feuille 1),
   Noeud (3, Feuille 1, Noeud (2, Feuille 1, Feuille 1)))

# nombrefeuilles (arbre_fibo 5);;
```

```
- : int = 8

# type oper = | addition | multiplication ;;
Type oper defined.

# type calcul = | Valeur of int | Fonc of oper * calcul * calcul;;
Type calcul defined.

# let rec compute = function
        | Valeur n -> n
        | Fonc (addition, souscalcul1, souscalcul2) -> compute souscalcul1 +
              compute souscalcul2
        | Fonc (multiplication, souscalcul1, souscalcul2) ->
              compute souscalcul1 * compute souscalcul2;;
compute : calcul -> int = <fun>

# compute (Fonc( addition, (Valeur 12312), (Valeur 35453)));;
- : int = 47765

# compute (Fonc( addition, (Valeur 12), (Fonc (multiplication, Valeur 2, Valeur 35))));;
- : int = 82
```

FEUILLE DE TD 7

Solution de complexité

Solution de diviser pour régner

Exercice 1 (Suite de Fibonacci, encore et toujours) — 2

Il suffit d'observer qu'on obtient les termes de la suite de Fibonacci en calculant les puissances de $M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$, car, pour tout entier naturel n,

$$\begin{pmatrix} F_{n+1} \\ F_{n+2} \end{pmatrix} = M \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}.$$

En utilisant l'exponentiation rapide, on obtient bien une complexité logarithmique.

```
(* Multiplication de Knuth *)

(* Addition des polynômes *)

# let rec add_poly a b = let p = vect_length a and q = vect_length b in
        if p < q then add_poly b a else let s = make_vect p 0 in
                for i = 0 to (q - 1) do
                        s.(i) <- a.(i) + b.(i)
                        done;
                for i = q to (p - 1) do
                        s.(i) <- a.(i)
                done;
        s;;
add_poly : int vect -> int vect -> int vect = <fun>

# add_poly [| 1; 3; -4|] [|3; 6; 4|];;
- : int vect = [|4; 9; 0|]

(* Multiplication par d'un polynôme par un entier *)

# let multi lambda a =  let p = vect_length a in
        let m = make_vect p 0 in
                for i = 0 to (p - 1) do m.(i) <- lambda * a.(i) done;
        m;;
multi : int -> int vect -> int vect = <fun>

(* Soustraction, en utilisant la multiplication par -1 *)

# let mun = multi (-1);;
mun : int vect -> int vect = <fun>

# mun [|1; 8|];;
- : int vect = [|-1; -8|]

# let subs a b = add_poly a (mun b);;
subs : int vect -> int vect -> int vect = <fun>
```

```
(* Addition de trois polynômes *)

# let add3 a b c = add_poly (add_poly a b) c;;
add3 : int vect -> int vect -> int vect -> int vect = <fun>

(* Soustraction de deux polynômes à un troisième *)

# let sub2 a b c = subs (subs a b) c;;
sub2 : int vect -> int vect -> int vect -> int vect = <fun>

(* Décalage *)

# let decale a n = let p = vect_length a in
        let d = make_vect (p + n) 0 in
                for i = 0 to (p - 1) do
                        d.(i + n) <- a.(i)
                done;
        d;;



# let complete a n = let p = vect_length a in
        let d = make_vect (p + n) 0 in
                for i = 0 to (p - 1) do
                        d.(i) <- a.(i)
                done;
        d;;
decale : int vect -> int -> int vect = <fun>

(* Multiplication de Knuth : attention, ceci ne fonctionne que

si ces deux tableaux ont pour même taille une puissance de deux

Cela peut s'arranger en complétant nos tableaux par des 0 *)

# let rec mult_poly a b = let m = vect_length a in match m with
        | 1 -> [|a.(0) * b.(0)|]
        | _ -> let n = m / 2  in
                let p0 = sub_vect a 0 n and p1 = sub_vect a n n
                and q0 = sub_vect b 0 n and q1 = sub_vect b n n in
                let prod0 = mult_poly p0 q0 and prod1 = mult_poly p1 q1
                and prod01 = mult_poly (add_poly p0 p1) (add_poly q0 q1) in
        add3 prod0 (decale prod1 (2 * n)) (decale (sub2 prod01 prod0 prod1) n);;
mult_poly : int vect -> int vect -> int vect = <fun>

# mult_poly [|1; -1|] [|1; 1|];;
- : int vect = [|1; 0; -1|]

# mult_poly [|1; 2; 1; 0|] [|1 ; 3; 3; 1|];;
- : int vect = [|1; 5; 10; 10; 5; 1; 0|]
```

FEUILLE DE TD 9

Solution de tris

Exercice 1 (Tri bulle) — 0

L'algorithme du tri bulle proposé dans le cours effectue un nombre constant de comparaisons : comment améliorer l'algorithme afin qu'il en effectue moins en moyenne ?

Exercice 2 (Tri par insertion) — 0

L'algorithme du tri par insertion pour les vecteurs suit à la lettre la démarche de cette méthode de tri. En programmant par effets de bord, proposer un algorithme de tri par insertion plus efficace.

Exercice 3 (Tris pour les listes) — 1

1 Proposer un algorithme de fusion de deux listes ordonnées.

2 Proposer un algorithme de recherche dichotomique dans une liste chaînée triée.

3 Faire de même pour les tris par insertion, par sélection, et à bulles pour les listes chaînées.

4 Programmer les tris fusion et rapide pour les listes chaînées.

Exercice 4 (Stabilité des algorithmes de tri) — 2

Un algorithme de tri est dit *stable* s'il n'échange jamais des termes égaux en des indices distincts. Parmi les algorithmes proposés, lesquels sont-ils stables ? Comment rendre tout algorithme de tri stable ?

Solution de listes

```
(* Image miroir *)

# let rec image_miroir_bof = function
        | [] -> []
        | a :: q -> image_miroir_bof q @ [a];;
image_miroir_bof : 'a list -> 'a list = <fun>

# image_miroir_bof [1; 4; 9; 13];;
- : int list = [13; 9; 4; 1]

# let rec miroir l (accu : int list) = match l with
        | [] -> accu
        | a :: q -> miroir q (a :: accu);;
miroir : int list -> int list -> int list = <fun>

# let image_miroir l = miroir l [];;
image_miroir : int list -> int list = <fun>

# trace "miroir";;
The function miroir is now traced.
- : unit = ()

# image_miroir [1; 3; 8; 9];;
miroir <-- [1; 3; 8; 9]
miroir --> <fun>
miroir* <-- []
miroir <-- [3; 8; 9]
miroir --> <fun>
miroir* <-- [1]
miroir <-- [8; 9]
miroir --> <fun>
miroir* <-- [3; 1]
miroir <-- [9]
miroir --> <fun>
miroir* <-- [8; 3; 1]
miroir <-- []
miroir --> <fun>
miroir* <-- [9; 8; 3; 1]
miroir* --> [9; 8; 3; 1]
miroir* --> [9; 8; 3; 1]
miroir* --> [9; 8; 3; 1]
miroir* --> [9; 8; 3; 1]
miroir* --> [9; 8; 3; 1]
- : int list = [9; 8; 3; 1]

(* Définition d'un type liste *)

# type 'a Liste = Nil | Cons of 'a * ('a Liste);;
Type Liste defined.
```

```
(* Sémantique des fonctions de manipulation d'une liste *)

(* Prédicat testant si une liste est vide *)

# let est_vide = function
        | Nil -> true
        | _ -> false;;
est_vide : 'a Liste -> bool = <fun>

# est_vide (Cons (1, Nil));;
- : bool = false

(* Constructeur d'ajout en tête de liste *)

# let ajout a l = Cons (a, l);;
ajout : 'a -> 'a Liste -> 'a Liste = <fun>

# ajout 3.8 Nil;;
- : float Liste = Cons (3.8, Nil)

(* Fonctions de sélection *)

# let tete = function
        | Nil -> failwith "pas de tête pour la liste vide"
        | Cons (a, q) -> a;;
tete : 'a Liste -> 'a = <fun>

# let queue = function
        | Nil -> failwith "pas de tête pour la liste vide"
        | Cons (a, q) -> q;;
queue : 'a Liste -> 'a Liste = <fun>

# let longueur_iter l = let temp = ref l and i = ref 0 in
        while !temp <> Nil do
                i:= !i + 1;
                temp := queue !temp;
                done;
        !i;;
longueur_iter : 'a Liste -> int = <fun>

# longueur_iter (Cons (1, (Cons (3, Cons( 8, Nil)))));;
- : int = 3

(* Insertion et suppression *)

# let rec insertion elt pos l = match (l, pos) with
        | (_, 1) -> elt :: l
        | ([], _) -> failwith "Mauvaise position d'insertion"
        | (a :: q, _)-> a :: (insertion elt (pos - 1) q);;

# insertion 4 2 [1; 2; 3; 8];;
- : int list = [1; 4; 2; 3; 8]

# let rec suppression pos l = match (l, pos) with
        | ([], _) -> failwith "Rien à supprimer dans cette liste à cette position"
        | (a :: q, 1) -> q
        | (a :: q, _) -> a :: (suppression (pos - 1) q);;
insertion : 'a -> int -> 'a list -> 'a list = <fun>
```

```
suppression : int -> 'a list -> 'a list = <fun>

# suppression 4 [1; 4; 9; 6; 12];;
- : int list = [1; 4; 9; 12]

(* Programmes itératifs sur les listes *)

# let rec map_bis f = function
        | [] -> []
        | a :: q -> f a :: (map_bis f q);;
map_bis : ('a -> 'b) -> 'a list -> 'b list = <fun>

# map_bis (function x -> x * x) [1; 4; 9];;
- : int list = [1; 16; 81]

# let rec itlist f b = function
        | [] -> b
        | a :: q -> itlist f (f b a) q;;
itlist : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

# let somme = itlist (prefix +) 0;;
somme : int list -> int = <fun>

# somme [1; 4; 9];;
- : int = 14

# let produit = itlist (prefix *) 1;;
produit : int list -> int = <fun>

# produit [2; 4; 5];;
- : int = 40

# let rec listit f l b = match l with
        | [] -> b
        | a :: q -> f a (listit f q b);;
listit : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>

# let somme_bis l = listit (prefix +) l 0;;
somme_bis : int list -> int = <fun>

# somme_bis [1; 4; 9];;
- : int = 14

# let produit_bis l = listit (prefix *) l 1;;
produit_bis : int list -> int = <fun>

# produit_bis [2; 4; 5];;
- : int = 40

# let conse a l = a :: l;;
conse : 'a -> 'a list -> 'a list = <fun>

# let concat = listit conse;;
concat : '_a list -> '_a list -> '_a list = <fun>

# concat [1; 3; 5] [2; 4; 6];;
- : int list = [1; 3; 5; 2; 4; 6]

# let map_ter f l = let g a q = f a :: q in listit g l [];;
```

```
map_ter : ('a -> 'b) -> 'a list -> 'b list = <fun>

# map_ter (function x -> x * x) [2; 5; 9];;
 - : int list = [4; 25; 81]
```

FEUILLE DE TD 11

Solution de piles

```
(* Rotation d'une pile *)

# let transfert pile1 pile2 = while not est_vide pile1 do empile (depile pile1) pile2 done;;
transfert : 'a pile -> 'a pile -> unit = <fun>

# let rotation pile = let a = depile pile and aux = pile_vide () in
        transfert pile aux;
        empile a pile;
        transfert aux pile;;
rotation : 'a pile -> unit = <fun>

# let p = {contenu = [1; 2; 3; 4; 5]};;
p : int pile = {contenu = [1; 2; 3; 4; 5]}

# rotation p;;
- : unit = ()

# p;;
- : int pile = {contenu = [2; 3; 4; 5; 1]}

(* Expressions algébriques postfixées *)

# type operateur = Plus | Moins | Mult | Divise;;
Type operateur defined.

# type Lexeme = Var of float | Op of operateur;;
Type Lexeme defined.

# let valeur ope (a,b) = match ope with
        | Plus -> a +. b
        | Moins -> a -. b
        | Mult -> a *. b
        | Divise -> a /. b;;
valeur : operateur -> float * float -> float = <fun>

# let action_sur_pile op pile = match op with
        | Var c -> empile c pile;
        | Op ope -> empile (valeur ope (depile pile, depile pile)) pile;;
action_sur_pile : Lexeme -> float pile -> unit = <fun>

# let pile = pile_vide ();; empile 3. pile;; empile 2. pile;;
pile : '_a pile = {contenu = []}
- : unit = ()
- : unit = ()

# action_sur_pile (Op Mult) pile;;
- : unit = ()

# pile;;
```

```
- : float pile = {contenu = [6.0]}

# let rec eval pile = function
        | [] -> ()
        | a :: q -> action_sur_pile a pile; eval pile q;;
eval : float pile -> Lexeme list -> unit = <fun>

# let evalue_EAP l = let temp = pile_vide () in eval temp l; depile temp;;
evalue_EAP : Lexeme list -> float = <fun>

# evalue_EAP [Var 1.; Var 2.; Op Divise];;
- : float = 0.5

# evalue_EAP [Var 3.5; Var 4.5; Op Moins];;
- : float = -1.0

# evalue_EAP
        [Var 1.; Var 2.; Op Divise; Var 2.; Var 3.4; Var 4.2; Op Plus; Op Mult; Op Mult];;
- : float = 7.6

(* Bon parenthésage *)

# let depile_sans_effet pile = match pile.contenu with
        | [] -> failwith "Pile vide"
        | a :: q -> pile.contenu <- q;;
depile_sans_effet : 'a pile -> unit = <fun>

# let rec bon_parenthesage mot = let temp = pile_vide () in
        for i = 0 to (string_length mot - 1) do
                if mot.[i] = '(' then empile 1 temp else depile_sans_effet temp;
                done;
        est_vide temp;;
bon_parenthesage : string -> bool = <fun>

# bon_parenthesage "(((())))";;
- : bool = false

# bon_parenthesage "(()())())";;
Uncaught exception: Failure "Pile vide"

# bon_parenthesage "((()()))()";;
- : bool = true
```

Solution de logique (syntaxe)

Exercice 1 (Longueur d'une représentation linéaire) — 2

Preuve par induction structurelle, l'étape la plus difficile étant l'ajout d'un nouvel opérateur binaire. Si P et Q sont deux propositions vérifiant l'hypothèse d'induction, alors $(PbinQ)$ est de longueur $4(b(P)+b(Q))+n(P)+n(Q)+2+3$ (le 3 venant des deux parenthèses et de *bin*), *i.e.* $4b((PbinQ))+n((PbinQ))+1$.

Exercice 2 (Encadrement de la longueur d'une proposition à l'aide de sa hauteur) — 2

La taille minimale est obtenue lorsque chaque père a un unique fils, ce que l'on obtient en ne prenant que l'opérateur de négation. La taille maximale lorsque chaque père a deux fils, ce qui donne aucune négation, que des opérateurs binaires (on en a donc 2^n-1, sauf aux 2^n feuilles. D'après l'exercice précédent, sa longueur est $4\cdot(2^n-1)+0+1=2^{n+2}-3$.

```
(* Test de bonne écriture d'une expression logique postfixée *)

(* Définition d'un type lexeme *)

# type lexeme = Neg | Constante of bool | Varia of string | Binaire of binaire;;
Type lexeme defined.

(* Expression postfixée d'une formule sous forme de liste de lexèmes *)

# let rec EAP = function
        | Const c -> [Constante c]
        | Var v -> [Varia v]
        | Non prop' -> (EAP prop') @ [Neg]
        | Bin (op, sa1, sa2) -> (EAP sa1) @ (EAP sa2) @ [Binaire op];;
EAP : formule -> lexeme list = <fun>

# EAP P;;
- : lexeme list =
 [Constante true; Varia "a"; Varia "b"; Binaire Impli; Binaire Ou]

(* Poids d'une formule logique postfixée *)

# let rec poids = function
        | [] -> 0
        | a :: q -> let temp = poids q in
                match a with
                | Constante _ | Varia _ -> 1 + temp
                | Neg -> temp
                | _ -> temp - 1;;
poids : lexeme list -> int = <fun>
```

```
# poids (EAP P);;
- : int = 1

(* Test de bonne syntaxe d'une liste de lexèmes *)

# let transfert (l1, l2) = match l2 with
        | [] -> (l1, l2)
        | a :: q -> (a :: l1, q);;
transfert : 'a list * 'a list -> 'a list * 'a list = <fun>

# let rec bonne_syntaxe eap = let temp = ref ([hd eap], tl eap) in
        while poids (fst !temp) >= 1 && snd !temp <> []
                do temp := transfert !temp done;
        snd !temp = [] && poids (fst !temp) = 1;;
bonne_syntaxe : lexeme list -> bool = <fun>

# bonne_syntaxe (EAP P);;
- : bool = true

# bonne_syntaxe [Varia "a"; Varia "b"; Binaire Impli; Binaire Ou];;
- : bool = false

# bonne_syntaxe [Constante true; Varia "a"; Varia "b"; Binaire Impli];;
- : bool = false
```

Solution de logique (sémantique)

```
# table_ter (Bin (Impli, Bin (Impli, Var "a", Var "b"), (Bin (Et, Non (Var "c"), Var "b"))));;
| 0 | 0 | 0 || 0 |
| 0 | 0 | 1 || 0 |
| 0 | 1 | 0 || 1 |
| 0 | 1 | 1 || 0 |
| 1 | 0 | 0 || 1 |
| 1 | 0 | 1 || 1 |
| 1 | 1 | 0 || 1 |
| 1 | 1 | 1 || 0 |
- : unit = ()
```

Exercice 1 (Mise sous forme conjonctive ou disjonctive) — 0

1 On peut tout exprimer en fonction de $\neg, \wedge, \vee$, puis utiliser les règles algébriques dans notre algèbre de Boole. On trouve par exemple immédiatement « la » forme disjonctive suivante :

$$(a \wedge \neg b) \vee (\neg c \wedge b),$$

et on peut en déduire par distributivité « la » forme conjonctive suivante :

$$(a \vee \neg c) \wedge (a \vee b) \wedge (\neg b \vee \neg c).$$

On peut aussi utiliser la table de vérité ci-dessus.

2 $(a \Rightarrow (\neg c \vee d))$ est équivalent à la disjonction $\neg a \vee \neg c \vee d$.

$a \Leftrightarrow b$ se met sous forme conjonctive $(\neg a \vee b) \wedge (a \vee \neg b)$, d'où une forme conjonctive pour la proposition considérée :

$$(\neg a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg c \vee d).$$

$a \Leftrightarrow b$ se met sous forme disjonctive $(\neg a \wedge \neg b) \vee (a \wedge b)$. On obtient une forme disjonctive de la proposition considérée en distribuant :

$$(a \wedge b \wedge \neg c) \vee (a \wedge b \wedge d) \vee (\neg a \wedge \neg b).$$

Exercice 2 (De la fonction logique à une proposition logique) — 0

Il suffit de reconnaître une fonction logique donnée comme fonction booléenne associée à une disjonction de mintermes (comme on obtient la forme normale disjonctive d'une proposition logique).

Exercice 3 (Simplification d'une proposition) — 0

$(\neg(a \wedge b)) \wedge (a \vee \neg b) \wedge (a \vee b)$ se simplifie en $(\neg(a \wedge b)) \wedge a$ (en factorisant par b la dernière conjonction), puis simplement en $\neg b \wedge a$ grâce à une loi de Morgan, puis en distribuant.

```
# table_ter (Non (Bin (Equiv, Bin (Impli, Var "a", Var "b"), Var "c")));;
| 0 | 0 | 0 || 1 |
| 0 | 0 | 1 || 0 |
| 0 | 1 | 0 || 1 |
| 0 | 1 | 1 || 0 |
| 1 | 0 | 0 || 0 |
| 1 | 0 | 1 || 1 |
| 1 | 1 | 0 || 1 |
| 1 | 1 | 1 || 0 |
- : unit = ()

# table_ter
(Bin (Equiv, Bin (Ou, Bin (Impli, Var "a", Var "b"), Var "c"), Bin (Ou, Var "a", Var "c")));
| 0 | 0 | 0 || 0 |
| 0 | 0 | 1 || 1 |
| 0 | 1 | 0 || 0 |
| 0 | 1 | 1 || 1 |
| 1 | 0 | 0 || 0 |
| 1 | 0 | 1 || 1 |
| 1 | 1 | 0 || 1 |
| 1 | 1 | 1 || 1 |
- : unit = ()

# table_ter (Bin (Ou, Non (Bin (Et, Var "a", Var "b")), Bin (Et, Non (Var "c"), Var "b")));;
| 0 | 0 | 0 || 1 |
| 0 | 0 | 1 || 1 |
| 0 | 1 | 0 || 1 |
| 0 | 1 | 1 || 1 |
| 1 | 0 | 0 || 1 |
| 1 | 0 | 1 || 1 |
| 1 | 1 | 0 || 1 |
| 1 | 1 | 1 || 0 |
- : unit = ()
```

Exercice 4 (Tables de vérité, formes normales) — 0

Voir les tables de vérité ci-dessus, sur lesquelles on lit immédiatement la forme normale disjonctive, et avec un peu d'entraînement, la forme normale conjonctive.

Exercice 5 (Système complet de connecteurs) — 2

Il est connu depuis longtemps que $\{\neg, \wedge, \vee\}$ est un système complet de connecteurs ($a \Rightarrow b$ équivaut à $\neg a \vee b$, $a \Leftrightarrow b$ équivaut à $(a \wedge b) \vee (\neg a \wedge \neg b)$ par exemple). Les lois de Morgan montrent même que $\{\neg, \wedge\}$ et $\{\neg, \vee\}$ sont des systèmes complets de connecteurs.
Pour montrer que $\{NAND\}$ est un système complet de connecteurs, il suffit de montrer que l'on atteint $\neg, \wedge$ et $\vee$:
- $\neg a \equiv (a \,\mathrm{NAND}\, a)$.
- $(a \vee b) \equiv \neg a \,\mathrm{NAND}\, \neg b \equiv (a \,\mathrm{NAND}\, a) \,\mathrm{NAND} (b \,\mathrm{NAND}\, b)$.
- (facultatif) $(a \wedge b) \equiv \neg(a \,\mathrm{NAND}\, b) \equiv (a \,\mathrm{NAND}\, b) \,\mathrm{NAND} (a \,\mathrm{NAND}\, b)$.

De même, on a :
- $\neg a \equiv (a \,\mathrm{NOR}\, a)$.
- $(a \vee b) \equiv \neg(a \,\mathrm{NOR}\, b) \equiv (a \,\mathrm{NOR}\, b) \,\mathrm{NOR} (a \,\mathrm{NOR}\, b)$.

Solution de logique (circuits)

Exercice 1 (Additionneur 1-bit) — 0

1 Écrire un additionneur 1-bit à partir des formes normales disjonctives pour l'unité et la retenue, en utilisant des portes logiques « et » et « ou » à plusieurs entrées et la négation.

2 Faire de même en utilisant les tableaux de Karnaugh.

Exercice 2 (Exemple d'utilisation d'un tableau de Karnaugh) — 0

On considère une proposition logique P de tableau de Karnaugh

$(ab)\backslash(cd)$	00	01	11	10
00	1	0	1	1
01	0	0	0	0
11	0	1	0	0
10	1	0	0	1

1 Donner la forme normale disjonctive de P.

2 Donner les impliquants premiers de P.

3 Construire un circuit logique représentant P.

Exercice 3 (Un autre exemple d'utilisation d'un tableau de Karnaugh) — 0

1 Soit f la fonction booléenne de quatre variables telle que $f(a, b, c, d)$ renvoie 1 si et seulement si $abcd$ est l'écriture binaire d'un entier inférieur ou égal à 10. Construire le tableau de Karnaugh correspondant, en déduire une proposition de fonction booléenne f, et construire un circuit logique correspondant à f.

2 Faire de même avec $f(a, b, c, d) = 1$ si et seulement si $abcd$ est l'écriture binaire d'un nombre congru à 1 modulo 3.