Algorithmique

MICHEL QUERCIA

Ancien élève de l'École Normale Supérieure
Professeur en classes préparatoires au lycée Champollion de Grenoble

Corrections

La version « livre » diffusée par Vuibert comporte quelques erreurs, découvertes après impression, et qui sont corrigées dans cette version électronique.

p. 62 : inversion des seconds membres dans les formules de distributivité.
p. 154 (automate produit) : A et B sont supposés complets.
p. 248 (exercice 10-5) : correction de l'expression régulière pour L.

Table des matières

Cours et exercices

Problèmes

Travaux pratiques

Solutions des exercices

Solutions des problèmes

Solutions des travaux pratiques

Annexes

Préface

Cet ouvrage présente un cours d'algorithmique dispensé successivement au lycée Carnot de Dijon puis au lycée Poincaré de Nancy en classes préparatoires aux grandes écoles, option informatique. Il s'adresse donc en premier lieu aux étudiants de ces classes préparatoires, mais aussi aux étudiants du premier cycle de l'Université suivant un enseignement d'informatique. Enfin, il peut constituer une base de départ pour un enseignant de l'option informatique voulant construire son propre cours.

Le plan du livre est celui adopté pour l'exposé des cours donnés à Dijon et à Nancy, les chapitres 1 à 5 recouvrent le programme de première année (méthodes de programmation, structure de liste, logique booléenne) et les chapitres 6 à 11 celui de deuxième année (complexité des algorithmes, structure d'arbre, manipulation d'expressions formelles, langage et automates). A la suite de ces onze chapitres, j'ai inclus quelques sujets de problèmes et quelques sujets de travaux pratiques sélectionnés pour leur originalité parmi les sujets qui ont été donnés aux étudiants de Dijon ou de Nancy (à l'exception du sujet de TP intitulé « Recherche de contradictions par la méthode des consensus » qui est inspiré d'un exercice similaire présenté dans le cours de LUC ALBERT, et qui a été donné aux étudiants du lycée Champollion de Grenoble). Tous les sujets proposés sont accompagnés d'un corrigé plus ou moins détaillé. Ces corrigés sont regroupés en fin d'ouvrage de façon à éviter au lecteur la tentation de s'y reporter trop vite : un corrigé n'est en aucune façon *la* solution unique et incontournable au problème posé, mais plutôt *une* réponse possible que le lecteur comparera avec sa propre réponse.

Conformément à l'esprit du programme officiel, l'accent a été mis sur l'écriture effective de programmes implémentant les algorithmes étudiés. En effet, les raisonnements théoriques sur le comportement d'un algorithme et notamment sur sa complexité temporelle ou spatiale sont facilités par la précision que confère la rédaction d'un véritable programme. En outre, la programmation effective d'un algorithme et son exécution sur machine permettent de confronter la théorie à la réalité par une expérimentation pratique, de détecter et de corriger les erreurs élémentaires ou les insuffisances d'un raisonnement ainsi que d'affermir la compréhension du fonctionnement de l'algorithme. Le langage de programmation servant de support à ce cours est le langage CAML.

Le nom CAML est l'acronyme de *Categorical Abstract Machine Language*, par référence à un modèle de machine abstraite utilisé dans les années 1980 pour implémenter le premier compilateur CAML. Les implémentations actuelles de CAML n'utilisent plus ce modèle de machine abstraite, mais le nom est resté. Le système Caml-Light est une implémentation de CAML pouvant tourner sur micro-ordinateur, il a été développé à partir de 1990 par XAVIER LEROY. C'est ce système qui est actuellement utilisé dans la majorité des classes préparatoires aux grandes écoles pour l'enseignement de l'option informatique. Caml-Light est distribué gracieusement par l'INRIA via son site Internet :

`http://pauillac.inria.fr/caml/index-fra.html`

Le cours introduit au fur et à mesure les éléments du langage CAML nécessaires à l'exposé, et seulement ces éléments. Il ne constitue donc pas un manuel de programmation en CAML. Pour une présentation plus complète de ce langage, on pourra consulter le manuel de référence disponible sous forme électronique sur le site de l'INRIA, ainsi le livre de PIERRE WEIS et XAVIER LEROY cité en bibliographie. Le présent ouvrage comporte en annexe un aide mémoire pour le langage CAML, mis en page de façon à pouvoir être photocopié en recto-verso sur une feuille de format A4.

Le mode de diffusion de ce livre est assez inhabituel : le texte complet est disponible gratuitement sous forme électronique sur le serveur de l'INRIA, et les éditions Vuibert ont accepté de le publier sous forme « papier » permettant un plus grand confort de lecture. Je tiens à remercier les éditions Vuibert pour l'aide qu'elles apportent ainsi, dans des conditions commerciales incertaines, à la diffusion de ce cours. Je remercie aussi chaleureusement JEAN-PIERRE CARPENTIER qui fut mon professeur de mathématiques puis mon collègue en mathématiques et informatique au lycée Carnot de Dijon, et qui a bien voulu relire ce livre et a minutieusement contrôlé toutes les démonstrations du cours, ainsi que les corrigés des exercices et problèmes. Ses remarques et critiques, toujours pertinentes, ont permis d'améliorer notablement la présentation de certains points obscurs du cours, et de corriger les erreurs que comportait la version initiale. Je remercie enfin messieurs GÉRARD DUCHAMP professeur à l'université de Rouen, et OLIVIER BOUVEROT professeur en classes préparatoires au lycée Jules Ferry de Versailles, qui ont participé à la relecture finale de ce livre.

MICHEL QUERCIA, Juillet 2002

Cours et exercices

Chapitre 1

Méthodes de programmation

1-1 Description d'un algorithme

Un *algorithme* est la description non ambiguë en un nombre fini d'étapes d'un calcul ou de la résolution d'un problème. Un *programme* est la traduction d'un algorithme dans un langage de programmation « compréhensible » par une machine. Une *fonction* est une relation entre chaque élément d'un ensemble dit *de départ* et un unique élément d'un autre ensemble dit *d'arrivée*. Considérons par exemple le problème de la résolution dans $\mathbf{R}$ d'une équation du second degré à coefficients réels :

La fonction : $f : \begin{cases} \mathbf{R}^3 & \longrightarrow & \mathcal{P}(\mathbf{R}) \\ (a,b,c) & \longmapsto & \{x \text{ tq } ax^2+bx+c=0\} \end{cases}$

L'algorithme :

> *degré_2(a,b,c) :* calcule les racines x', x'' de l'équation $ax^2+bx+c=0$
> *si* $a = 0$ *alors ERREUR*
> *sinon, soit* $\Delta = b^2 - 4ac$ *:*
> *si* $\Delta \geqslant 0$ *alors* $x' = \dfrac{-b+\sqrt{\Delta}}{2a}$, $x'' = \dfrac{-b-\sqrt{\Delta}}{2a}$
> *sinon ERREUR*
> *fin*

La description est finie (il n'y a pas de points de suspension), elle est non ambiguë dans la mesure où l'on sait réaliser les opérations mathématiques $(+,-,*,/,\sqrt{\ })$ et où l'opération ERREUR est connue.

Le programme (en CAML) :

```
let degré_2(a,b,c) =
  if a = 0.
  then failwith "équation incorrecte"
  else let delta = b **. 2. -. 4. *. a *. c in
    if delta >= 0.
    then ((-. b +. sqrt(delta))/.(2. *. a),
          (-. b -. sqrt(delta))/.(2. *. a))
    else failwith "discriminant négatif"
;;
```

Les symboles `+.`, `-.`, `*.`, `/.` et `**.` désignent en CAML les opérations usuelles sur les nombres réels : addition, soustraction, multiplication, division et exponentiation. les opérations entre nombres entiers sont notées `+`, `-`, `*`, `/` (division entière) et `mod` (reste). Il n'y a pas d'élévation à une puissance entière prédéfinie en CAML.

La version suivante est un peu plus efficace car elle économise la répétition des calculs de $\sqrt{\Delta}$ et de $2a$:

```
let degré_2(a,b,c) =
  if a = 0.
  then failwith "équation incorrecte"
  else let delta = b **. 2. -. 4. *. a *. c in
       if delta >= 0.
       then let d = sqrt(delta) and deux_a = 2. *. a in
            ((d -. b)/.deux_a, (-. d -. b)/.deux_a)
       else failwith "discriminant négatif"
;;
```

Remarquons que l'algorithme ne correspond pas tout à fait à la définition de f : $f(a,b,c)$ est correctement définie même si $a = 0$ ou $b^2 - 4ac < 0$ mais ces situations ont été éliminées pour pouvoir plus facilement traduire l'algorithme en un programme (CAML permet de produire un résultat qui peut être multiforme (vide, $\mathbf{R}$, un singleton ou une paire) mais au prix d'une complication importante). Par ailleurs la définition de f n'indique aucun moyen de calcul effectif. Il pourrait exister plusieurs algorithmes foncièrement différents réalisant le calcul de f (dans le cas de l'équation de degré 2 on peut effectuer une résolution par approximations successives au lieu d'appliquer la formule $(-b \pm \sqrt{\Delta})/(2a)$), ou même aucun. La fonction f ci dessous est bien définie mais n'est pas calculable :

$$f(x) = \begin{cases} 1 & \text{si } \pi^x \text{ comporte une infinité de 13 dans son développement décimal ;} \\ 0 & \text{sinon.} \end{cases}$$

On peut affirmer que $f(0) = 0$, $f(\ln(13/99)/\ln \pi) = 1$, mais en l'état actuel des connaissances on ne peut pas déterminer $f(1)$, $f(2)$, $f(\pi)$, ...

L'algorithmique consiste, partant de la définition d'une fonction f, à écrire un ou plusieurs algorithmes décrivant les étapes du calcul de $f(x)$, à prouver que ces algorithmes sont corrects, c'est-à-dire que le résultat calculé est bien $f(x)$, et à déterminer le temps et la quantité de mémoire nécessaires à l'exécution de ces algorithmes. Le temps est mesuré en nombre d'opérations considérées comme

« élémentaires », et la place mémoire en nombre de valeurs « élémentaires » dont on a besoin. Par exemple, l'algorithme *degré_2* effectue 4 opérations pour calculer Δ, 5 opérations pour calculer x' et 5 pour x'', soit en tout 14 opérations, 16 si l'on compte comme opérations les tests *si* $a = 0$ et *si* $\Delta \geqslant 0$ (on ne compte pas les opérations effectuées en cas d'erreur). L'espace mémoire nécessaire est de 3 nombres : Δ, x', x''. En pratique il est plus facile de raisonner sur un programme que sur un algorithme car le texte d'un programme est plus précis. Par exemple les deux programmes `degré_2` ci-dessus correspondent au même algorithme mais le deuxième effectue trois opérations de moins puisque la racine carrée de `delta` et le dénominateur `2.*.a` ne sont calculés qu'une fois, et le numérateur de x' est calculé à l'aide d'une soustraction au lieu d'un changement de signe suivi d'une addition. En contrepartie, ce deuxième programme requiert deux places mémoires supplémentaires pour conserver les valeurs `d` et `deux_a`.

1-2 Itération

L'itération consiste à répéter plusieurs fois un sous-algorithme. Le nombre de répétitions peut être défini lors de la rédaction de l'algorithme, mais on peut aussi indiquer à quelle condition l'itération doit se poursuivre ou non. Dans ce cas il est nécessaire de s'assurer que la condition d'arrêt sera remplie au bout d'un nombre fini de *tours de boucle* pour garantir que l'algorithme comporte un nombre fini d'étapes (condition de terminaison).

Exemple, calcul de x^n :

```
(* calcule x^n pour x et n entiers, n >= 1 *)
let puissance(x,n) =
  let p = ref x in
  for i = 2 to n do p := !p * x done;
  !p
;;
```

`p = ref x` signifie que `p` fait *référence* à un nombre entier variable et valant initialement `x`. Ce nombre est désigné par `!p` dans la suite de la fonction `puissance` et l'instruction `p := !p * x` a pour effet de modifier la valeur de l'entier auquel `p` fait référence. En langage machine on dit que `p` est un *pointeur* vers une zone mémoire et que cette zone mémoire doit être interprétée comme la représentation machine d'un nombre entier.

L'identificateur `i` est appelé *indice de boucle*. Il désigne une valeur entière variable qui augmente de 1 à chaque itération. Bien que représentant tous les deux des nombres variables, `p` et `i` n'ont pas le même statut : `p` est l'adresse d'une variable et `!p` est la valeur courante de cette variable, tandis que `i` est la valeur courante de la variable indice de boucle. On n'a pas accès en CAML à l'adresse à laquelle `i` est rangé en mémoire.

La fonction `puissance` telle qu'elle est définie ci dessus ne produit un résultat correct que lorsque l'argument n est entier au moins égal à 1. Cette restriction figure en commentaire dans la définition de puissance, mais il serait peut-être plus prudent de contrôler la validité de n dans le corps de puissance par un test approprié. Quoi qu'il en soit, si l'on fournit un argument n négatif ou nul alors la boucle n'est pas exécutée, c'est une convention du langage CAML, et donc puissance renvoie le résultat x. Une autre possibilité de résultat incorrect est due à la limitation des nombres entiers en machine : le calcul de `!p * x` peut produire un résultat supérieur au plus grand nombre entier représentable en machine. Dans ce cas, soit l'ordinateur détecte ce fait et interrompt le programme, soit, le plus souvent, ce débordement n'est pas détecté et les calculs continuent avec une valeur incorrecte pour `!p`. Sur un micro-ordinateur 32 bits c'est le cas et on observe que `puissance (10,10) = -737418240` ...

Boucle avec arrêt conditionnel :

```
(* cherche le plus petit diviseur > 1 de n, n >= 2 *)
let diviseur(n) =
  let d = ref 2 in
  while n mod !d <> 0 do d := !d + 1 done;
  !d
;;
```

Preuve de correction : la boucle est finie car il existe au moins un diviseur de n à savoir n lui même et `!d` avance de 1 en 1 à partir de 2 avec $2 \leqslant n$ par hypothèse. Lorsque la boucle se termine, `!d` contient un diviseur de n puisque c'est la condition d'arrêt, et c'est le plus petit à part 1 car sinon la boucle se serait terminée plus tôt.

Calcul sur les polynômes. Étant donnés un polynôme :

$$P = a_0 + a_1X + \ldots + a_nX^n$$

à coefficients réels et un nombre réel x, on veut calculer la valeur $P(x)$. La méthode naïve conduit à calculer séparément les puissances de x : $1, x, \ldots, x^n$, à les multiplier par les coefficients a_i correspondants et à additionner le tout :

```
(* calcule p(x) pour un polynôme p en un point x *)
let valeur p x =
  let n = vect_length(p) - 1 (* n = deg(p) *)
  and xk = ref 1.0           (* !xk = x^k  *)
  and vk = ref p.(0)         (* !vk = a0 + .. + ak*xk *)
  in

  for k = 1 to n do
    xk := x *. !xk;
    vk := !vk +. p.(k) *. !xk
  done;
  !vk (* résultat *)
;;
```

Les coefficients de P sont conservés dans le *vecteur* `p`, c'est-à-dire que `p.(i)` est la valeur du coefficient a_i. `p.(i)` est appelé *élément d'indice* `i` *de* `p`, `i` étant un nombre entier compris entre 0 et `vect_length(p)-1` où `vect_length(p)` est le nombre d'éléments de `p`. Par ailleurs la fonction `valeur` est définie comme une fonction à deux arguments « détachés » : `valeur p x` et non `valeur(p,x)`. Ceci permet (c'est une spécificité de CAML) de l'utiliser avec le premier argument seul sous la forme :

```
let f = valeur p;;
```

ce qui définit `f` comme une fonction à un argument de sorte que `f(x)` équivaut à `valeur p x`. On peut ainsi ne pas spécifier le polynôme `p` à chaque calcul si l'on doit en faire plusieurs avec le même polynôme.

Démontrons que cet algorithme calcule effectivement le nombre $P(x)$: pour cela on montre par récurrence sur k que, à l'entrée dans la boucle, on a les relations :

$$(*) \qquad !xk = x^{k-1}, \qquad\qquad !vk = a_0 + a_1 x + \ldots + a_{k-1}x^{k-1}$$

et à la sortie :

$$(**) \qquad !xk = x^{k}, \qquad\qquad !vk = a_0 + a_1 x + \ldots + a_{k}x^{k}$$

ce qui résulte clairement des expressions affectées à `xk` et à `vk` dans le programme. La dernière valeur de `!vk` est donc $a_0 + a_1x + \ldots + a_nx^n = P(x)$. Les propriétés $(*)$ et $(**)$ sont appelées *invariant de boucle en entrée* et *invariant de boucle en sortie* : tout algorithme comportant une boucle non triviale doit être accompagné de la spécification d'un tel invariant de boucle permettant de vérifier aisément la correction de l'algorithme.

Temps d'exécution : si l'on néglige le temps de calcul de `vect_length(p)-1` et la mise à jour de k au cours de l'itération, l'algorithme effectue une addition et deux multiplication réelles à chaque tour de boucle, donc au total n additions et $2n$ multiplications.

Algorithme de HÖRNER : on peut conduire plus rapidement le calcul de $P(x)$ en effectuant les opérations *à l'envers* :

```
(* calcule p(x) pour un polynôme p en un point x *)
let Hörner p x =
  let n = (vect_length p) - 1 in               (* n = deg(p) *)
  let s = ref p.(n) in        (* !s = a_k + ... + a_n*x^(n-k) *)
  for k = n-1 downto 0 do s := !s *. x +. p.(k) done;
  !s
;;
```

On démontre comme précédemment que `Hörner` calcule effectivement $P(x)$, mais à présent il n'est effectué qu'une multiplication par tour de boucle donc le temps de calcul est de n additions et n multiplications réelles. On peut espérer un gain en vitesse compris entre 33% et 50% en utilisant cet algorithme suivant que le temps d'une addition est négligeable devant celui d'une multiplication ou lui est comparable.

1-3 Récursivité

Récursivité simple

Un algorithme est dit *récursif* lorsqu'il intervient dans sa description, c'est-à-dire lorsqu'il est défini en fonction de lui même. Très souvent un algorithme récursif est lié à une relation de récurrence permettant de calculer la valeur d'une fonction pour un argument n à l'aide des valeurs de cette fonction pour des arguments inférieurs à n. Reprenons l'exemple du calcul de x^n vu précédemment ; on peut définir x^n par récurrence à partir des relations :

$$x^0 = 1, \qquad x^n = x * x^{n-1} \text{ si } n \geqslant 1.$$

La traduction en CAML de ces relations est :

```
(* calcule x^n pour x et n entiers, n >= 0 *)
let rec puissance(x,n) =
  if n = 0 then 1 else x * puissance(x,n-1)
;;
```

ou plus élégamment :

```
let rec puissance(x,n) = match n with
 | 0 -> 1
 | _ -> x * puissance(x,n-1)
;;
```

(_ désigne une valeur quelconque en CAML). Observons cet algorithme tourner :

```
trace "puissance";; puissance(5,3);;
The function puissance is now traced.
- : unit = ()
#puissance <-- 5, 3
puissance <-- 5, 2
puissance <-- 5, 1
puissance <-- 5, 0
puissance --> 1
puissance --> 5
puissance --> 25
puissance --> 125
- : int = 125
```

La machine applique la règle : `puissance(x,n) = x * puissance(x,n-1)` tant que l'exposant est différent de 0, ce qui introduit des calculs intermédiaires jusqu'à aboutir au *cas de base* : `puissance(x,0) = 1`. Les calculs en suspens sont alors achevés dans l'ordre inverse jusqu'à obtenir le résultat final. Comme pour les boucles avec condition d'arrêt, il faut s'assurer que le cas de base sera atteint en un nombre fini d'étapes sinon l'algorithme « boucle ». Pour la fonction `puissance` précédente la terminaison est garantie si n est entier positif ou nul, il y a bouclage si $n < 0$. On peut éliminer ce risque de bouclage en remplaçant le test `if n = 0` par `if n <= 0`. Dans ce cas l'algorithme ne boucle plus mais il ne fournit pas pour autant un résultat correct lorsque $n < 0$.

Les fonctions vérifiant une relation de récurrence de la forme :

$$f(0) = f_0, \qquad f(n) = g(n, f(n-1)) \text{ si } n \geqslant 1,$$

se prêtent aussi facilement à un codage itératif que récursif. Les performances des programmes correspondants sont généralement comparables en termes de temps d'exécution, mais une programmation récursive produit autant de calculs en suspens que le nombre d'étapes nécessaires pour arriver au cas de base, c'est-à-dire n dans cet exemple. La quantité de mémoire nécessaire pour exécuter l'algorithme récursif est donc proportionnelle à n ce qui peut être gênant si n est grand, alors qu'elle est constante pour l'algorithme itératif.

Récursivité double

```
(* calcule le coefficient du binôme C(n,p), n >= p >= 0 *)
let rec binome(n,p) =
  if p = 0 or p = n then 1
  else binome(n-1,p) + binome(n-1,p-1)
;;
```

L'algorithme de calcul de C_n^p est déduit la relation de PASCAL avec deux cas de base regroupés : $C_n^0 = C_n^n = 1$. Montrons que cet algorithme termine et fournit le résultat correct dans tous les cas où $n \geqslant p \geqslant 0$:

– On remarque d'abord que si $0 < p < n$ alors $0 < p \leqslant n-1$ et $0 \leqslant p-1 < n-1$ donc si le cas de base n'est pas atteint les deux appels récursifs de `binome` ont des arguments convenables.

– Ensuite, on démontre par récurrence sur n la propriété suivante :

  si $0 \leqslant p \leqslant n$ *alors* `binome`(n, p) *termine et retourne le coefficient* C_n^p.

  C'est évident si $n = 0$ puisqu'alors $p = 0$, et si c'est vrai pour un entier $n-1$ alors ça l'est aussi pour n car pour $p = 0$ ou $p = n$ on obtient le résultat correct $C_n^p = 1$, et si $0 < p < n$ alors la formule :

  ```
  binome(n-1,p) + binome(n-1,p-1)
  ```

  est calculée par hypothèse de récurrence en un nombre fini d'étapes et fournit la bonne valeur pour C_n^p. ■

La correction de `binome` est donc prouvée. Cependant cet algorithme est très peu efficace. La figure 1 montre la trace du calcul de C_5^3. La fonction `binome` a été appelée 19 fois : en effet, on a $C_5^3 = C_4^3 + C_4^2$ et $C_4^3 = C_3^3 + C_3^2$, $C_4^2 = C_3^2 + C_3^1$, d'où $C_5^3 = C_3^3 + 2C_3^2 + C_3^1$ mais le calcul de C_3^2 est effectué 2 fois. Au rang suivant, on a $C_5^3 = C_3^3 + 2C_2^2 + 3C_2^1 + C_2^0$ et l'on constate que C_2^2 est calculé deux fois et C_2^1 l'est 3

```
trace "binome";; binome(5,3);;
The function binome is now traced.
- : unit = ()
#binome <-- 5, 3
binome <-- 4, 2
binome <-- 3, 1
binome <-- 2, 0
binome --> 1
binome <-- 2, 1
binome <-- 1, 0
binome --> 1
binome <-- 1, 1
binome --> 1
binome --> 2
binome --> 3
binome <-- 3, 2
binome <-- 2, 1
binome <-- 1, 0
binome --> 1
binome <-- 1, 1
binome --> 1
binome --> 2
binome <-- 2, 2
binome --> 1
binome --> 3
binome --> 6
binome <-- 4, 3
binome <-- 3, 2
binome <-- 2, 1
binome <-- 1, 0
binome --> 1
binome <-- 1, 1
binome --> 1
binome --> 2
binome <-- 2, 2
binome --> 1
binome --> 3
binome <-- 3, 3
binome --> 1
binome --> 4
binome --> 10
- : int = 10
```

Figure 1 : calcul récursif de C_n^p

fois... Tout se passe comme si la machine n'avait « aucune mémoire » et refaisait sans cesse les mêmes calculs, ce qui est effectivement le cas car le programme n'indique pas qu'il faut mémoriser les calculs intermédiaires. La programmation efficace d'une fonction doublement récursive nécessite de coder la mémorisation des valeurs devant servir plusieurs fois, par exemple dans un vecteur. L'exercice **1-7** propose une meilleure programmation du calcul de C_n^p.

Récursivité mutuelle

Deux algorithmes sont dits *mutuellement récursifs* lorsque chacun des deux fait appel à l'autre. Par exemple :

```
let rec pair(x) = match x with
| 0 -> true
| _ -> impair(x-1)

and impair(x) = match x with
| 0 -> false
| _ -> pair(x-1)
;;
```

Les deux définitions doivent être données en une seule instruction de la forme : `let rec ... and ...`. On peut spécifier un nombre quelconque de fonctions dépendant les unes des autres. Les fonctions mutuellement récursives apparaissent naturellement dans les problèmes de parcours d'arbres ou d'analyse syntaxique, en particulier dans les évaluateurs de formules et les compilateurs.

1-4 Diviser pour régner

La méthode « diviser pour régner » consiste à ramener la résolution d'un problème dépendant d'un entier n à la résolution de plusieurs problèmes identiques portant sur des entiers $n' < n$; en général $n' \approx n/2$. Il s'agit d'un cas particulier de récursion avec une décroissance très rapide vers le cas de base.

Exemple, calcul de x^n. On a vu que x^n peut être défini par la relation de récurrence :

$$x^0 = 1, \qquad x^n = x * x^{n-1}.$$

On peut aussi utiliser la relation mathématiquement plus compliquée :

$$x^0 = 1, \qquad x^1 = 1, \qquad x^{2k} = (x^k)^2, \qquad x^{2k+1} = x * x^{2k}.$$

Ici $n = 2k$ ou $2k+1$ et $n' = k \approx n/2$.

```
let rec puiss_2(x,n) = match n with
 | 0 -> 1
 | 1 -> x
 | _ -> let y  = puiss_2(x,n/2) in
        if n mod 2 = 0 then y*y else x*y*y
 ;;
```

La différence entre `puissance` telle qu'elle est définie en **1-2** ou en **1-3** et `puiss_2` se fait sentir dès que n dépasse la dizaine :

`puissance(x,10)` ->	$x*x*x*x*x*x*x*x*x*x$:	9 multiplications
`puiss_2(x,10)` ->	$(x*(x^2)^2)^2$:	4 multiplications
`puiss_2(x,100)` ->	$((x*(((x*x^2)^2)^2)^2)^2)^2$:	8 multiplications

Si l'on considère que le temps d'une multiplication est constant (c'est-à-dire indépendant des opérandes), alors `puiss_2` est deux fois plus rapide que `puissance` pour $n = 10$ et douze fois plus rapide pour $n = 100$. Cette hypothèse de constance du temps de multiplication est en pratique vérifiée si l'on opère sur des nombres de taille machine fixe, ce qui n'est pas réaliste lorsque n est grand. Elle est par contre vérifiée pour toute valeur de n dans le problème de *l'exponentiation modulaire* : calculer $x^n \bmod a$ où a est un entier naturel non nul donné, en remplaçant les expressions `y*y` et `x*y*y` par `y*y mod a` et `x*y*y mod a`. L'exercice **1-9** propose l'étude des performances comparées de `puissance` et `puiss_2` dans le cas de nombres de tailles arbitrairement grandes.

Multiplication de polynômes

Soient $P(x) = a_0 + a_1x + \ldots + a_px^p$ et $Q(x) = b_0 + b_1x + \ldots + b_qx^q$ deux polynômes donnés par leurs coefficients (entiers, réels, ...) dont on veut calculer le produit $R(x) = c_0 + c_1x + \ldots + c_rx^r$. En développant le produit $P(x)Q(x)$ on obtient la relation :

$$R(x) = \sum_{i=0}^{p} \sum_{j=0}^{q} a_i b_j x^{i+j}$$

qui est implémentée dans le programme suivant :

```
(* calcule le produit polynomial des vecteurs p et q          *)
let produit(p,q) =
  let dp = vect_length(p) - 1            (* degré de p       *)
  and dq = vect_length(q) - 1 in         (* degré de q       *)

  let r = make_vect (dp + dq + 1) 0 in   (* initialisation   *)
  for i = 0 to dp do
    for j = 0 to dq do                   (* calcule tous les *)
      r.(i+j) <- r.(i+j) + p.(i) * q.(j) (* produits ai * bj *)
    done
  done;
  r (* résultat *)
;;
```

Le temps d'exécution de ce programme est $(p+1)(q+1)$ multiplications « élémentaires » et autant d'additions, soit de l'ordre de $2n^2$ opérations pour deux polynômes de degré n. Un algorithme de type « diviser pour régner » a été présenté par KNUTH en 1969 : on suppose que les polynômes P et Q sont de degrés au plus $n-1$, on note $k = \lfloor n/2 \rfloor$, $\ell = n - k = \lceil n/2 \rceil$ où $\lfloor u \rfloor$ et $\lceil u \rceil$ désignent les parties entières inférieure et supérieure d'un réel u, et on décompose les polynômes P et Q de la manière suivante :

$$P(x) = (a_0 + \ldots + a_{k-1}x^{k-1}) + x^k(a_k + \ldots + a_{n-1}x^{\ell-1}) = P_0 + x^k P_1 \ ;$$
$$Q(x) = (b_0 + \ldots + b_{k-1}x^{k-1}) + x^k(b_k + \ldots + b_{n-1}x^{\ell-1}) = Q_0 + x^k Q_1.$$

On a alors :

$$\begin{aligned} R(x) &= (P_0 + x^k P_1)(Q_0 + x^k Q_1) \\ &= P_0Q_0 + x^k(P_0Q_1 + P_1Q_0) + x^{2k}P_1Q_1 \\ &= P_0Q_0 + x^k((P_0+P_1)(Q_0+Q_1) - P_0Q_0 - P_1Q_1) + x^{2k}P_1Q_1. \end{aligned}$$

Cette formule fait apparaître trois multiplications de polynômes de degré au plus $k-1$, ou $\ell-1$, deux additions de polynômes de degré au plus $\ell-1$ et trois additions de polynômes de degré au plus $2\ell-2$ (le calcul de $P_0Q_0 + x^{2k}P_1Q_1$ peut être effectué sans addition). Elle est implémentée dans le programme suivant :

```
(* multiplication de Knuth des polynômes p et q       *)
(* on suppose que p et q ont même longueur n          *)
let rec mult_Knuth p q n =
  let r = make_vect (2*n-1) 0 in  (* résultat <- 0   *)

  (* cas de base : p et q sont constants              *)
  if n = 1 then r.(0) <- p.(0) * q.(0)

  else begin (* cas général : on divise pour régner *)
    let k  = n/2 and l = (n+1)/2 in
```

```
    let p0 = sub_vect p 0 k     (* découpe p,q en 2 *)
    and p1 = sub_vect p k l
    and q0 = sub_vect q 0 k
    and q1 = sub_vect q k l in

    let p01 = make_vect l 0  (* calcule p0+p1,q0+q1 *)
    and q01 = make_vect l 0 in
    for i = 0 to k-1 do
      p01.(i) <- p0.(i) + p1.(i);
      q01.(i) <- q0.(i) + q1.(i)
    done;
    if k < l then begin
      p01.(k) <- p1.(k);
      q01.(k) <- q1.(k)
    end;

    let r0 = mult_Knuth p0  q0  k      (* récursion *)
    and r1 = mult_Knuth p01 q01 l
    and r2 = mult_Knuth p1  q1  l in

    (* assemble les produits *)
    for i = 0 to 2*k-2 do r.(i)     <- r0.(i) done;
    for i = 0 to 2*l-2 do r.(i+2*k) <- r2.(i) done;
    for i = 0 to 2*k-2 do
      r.(i+k) <- r.(i+k) + r1.(i) - r0.(i) - r2.(i)
    done;
    for i = 2*k-1 to 2*l-2 do
      r.(i+k) <- r.(i+k) + r1.(i) - r2.(i)
    done;
  end;
  r
;;
```

Temps de calcul : on se limite pour simplifier au cas où n est une puissance de 2 (voir la section **6-3** pour le cas général). Soit $M(p)$ le nombre de multiplications et $A(p)$ le nombre d'additions/soustractions effectuées pour multiplier deux polynômes de longueur $n = 2^p$. On a :

$$\begin{aligned} &M(0) = 1, && A(0) = 0, \\ &M(p) = 3M(p-1), && A(p) = 3A(p-1) + 2\cdot 2^{p-1} + 3\cdot(2^p - 1) \\ & && = 3A(p-1) + 2^{p+2} - 3. \end{aligned}$$

On obtient alors par récurrence : $M(p) = 3^p$ et $A(p) = \frac{13}{2}\cdot 3^p - 2^{p+3} + \frac{3}{2}$. On a $3^p = 2^{p\log_2(3)} = n^{\log_2(3)}$, donc le nombre total d'opérations effectuées est environ égal à $\frac{15}{2}n^{\log_2(3)}$. Pour $n \geqslant 25$, ce nombre est inférieur au nombre d'opérations effectuées par `produit`. Toutefois la complication de l'algorithme de Knuth le rend moins performant que `produit` pour de petites valeurs de n. Expérimentalement, `mult_Knuth` est aussi rapide que `produit` pour $n = 128$ et devient plus rapide pour $n \geqslant 256$.

n	1	2	4	8	16	32	64	128	256	
produit	1	4	16	64	256	1024	4096	16384	65536	additions
mult_Knuth	0	5	28	113	400	1325	4228	13193	40600	
produit	1	4	16	64	256	1024	4096	16384	65536	multiplications
mult_Knuth	1	3	9	27	81	243	729	2187	6561	

Note historique : l'algorithme de multiplication rapide décrit ci-dessus, bien que dû à KNUTH, est généralement appelé « algorithme de KARATSUBA » car il est dérivé d'une idée similaire présentée par KARATSUBA en 1961.

1-5 Exercices

Exercice 1-1 : classement de trois nombres
Écrire un algorithme ou un programme CAML permettant de classer trois nombres en effectuant le moins possible de comparaisons.

Exercice 1-2 : nombres parfaits
Un nombre entier $n \geqslant 2$ est dit parfait s'il est égal à la somme de tous ses diviseurs stricts, 1 compris. Écrire un programme qui teste si son argument est un nombre parfait.

Exercice 1-3 : algorithme de HÖRNER et translation de polynômes
On représente un polynôme $P = p_0 + p_1X + \ldots + p_{n-1}X^{n-1}$ à coefficients entiers par le vecteur `p = [|`$p_0; \ldots; p_{n-1}$`|]`. En s'inspirant de l'algorithme de HÖRNER, écrire une fonction CAML :

```
translate : int vect -> int -> int vect
```

telle que `translate p a` calcule le polynôme Q défini par $Q(x) = P(x + a)$.

Exercice 1-4 : décodage de nombres entiers

1. Soit s une chaîne de caractères représentant un nombre entier naturel par son écriture décimale. Écrire une fonction CAML calculant la valeur (de type `int`) associée à s. La valeur d'un caractère représentant un chiffre décimal peut être obtenue à l'aide de la fonction suivante :

```
let valeur_chiffre(c) = int_of_char(c) - int_of_char(`0`);;
```

2. Soient s et s' deux chaînes de caractères représentant les entiers naturels a et b. Il s'agit dans cette question de comparer a et b. Une première solution est de calculer en machine les nombres a et b puis de comparer les résultats, mais cette solution échoue si a ou b dépasse la capacité d'une variable de type `int`. On demande ici d'écrire un programme qui compare a et b sans les calculer et en fournissant un résultat correct quelles que soient les longueurs de s et s'.

Exercice 1-5 : racine carrée

La racine carrée d'un réel $a > 0$ peut être calculée de façon approchée par l'algorithme de HERON :

choisir $x_0 > 0$ *et calculer* $x_1, \ldots, x_n, \ldots$ *par la relation* $x_{k+1} = (x_k + a/x_k)/2$. *Alors la suite* (x_n) *converge vers* $\sqrt{a}$.

1. Montrer que la suite (x_n) est décroissante à partir du rang 1 et qu'elle converge effectivement vers $\sqrt{a}$. L'algorithme de HERON mérite-t-il vraiment l'appellation d'algorithme ?

2. On suppose maintenant que a est entier et l'on souhaite calculer la racine carrée entière de a, c'est-à-dire l'entier naturel b tel que $b^2 \leqslant a < (b+1)^2$. Plutôt que d'appliquer « l'algorithme » de HERON, on lui substitue une version entière :

   choisir x_0 *entier strictement positif et calculer* $x_1, \ldots, x_n, \ldots$ *par la relation* $x_{k+1} = \lfloor (x_k + a/x_k)/2 \rfloor$ *où* $\lfloor u \rfloor$ *désigne la partie entière de* u. *Arrêter les calculs dès que l'on obtient deux valeurs successives* x_n *et* x_{n+1} *telles que* $x_n \leqslant x_{n+1}$ *et* $n > 0$. *Retourner* x_n.

   Démontrer que cet algorithme est valide (c'est-à-dire qu'il termine et fournit le résultat correct).

Exercice 1-6 : calcul récursif de C_n^p

Le programme suivant calcule C_n^p pour n et p entiers naturels en utilisant la formule de PASCAL :

```
let rec binome(n,p) =
  if p = 0 or p = n then 1
  else binome(n-1,p) + binome(n-1,p-1)
;;
```

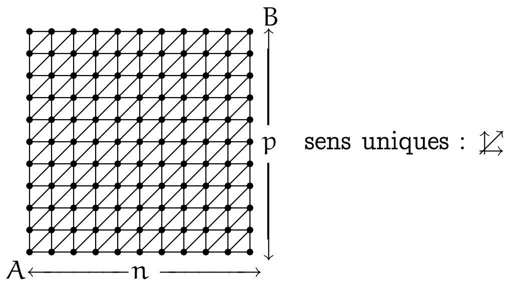
Déterminer le nombre d'appels à `binome` effectués lors du calcul de `binome(n,p)` en fonction de n et p.

Exercice 1-7 : calcul amélioré de C_n^p

Comment peut-on calculer C_n^p plus efficacement ?

Exercice 1-8 : dénombrement de chemins

La figure ci-dessous donne le plan d'une ville. Écrire un programme calculant le nombre de chemins reliant A à B.



Exercice 1-9 : exponentiation rapide
Les deux programmes suivants calculent x^n pour n entier naturel :

```
let rec puiss_1(x,n) = match n with
| 0 -> 1
| _ -> x*puiss_1(x,n-1)
;;

let rec puiss_2(x,n) = match n with
| 0 -> 1
| 1 -> x
| _ -> let y  = puiss_2(x,n/2) in
       if n mod 2 = 0 then y*y else x*y*y
;;
```

On considère que la multiplication de deux nombres de a et b chiffres prend un temps proportionnel à ab et que le résultat est codé systématiquement sur $a + b$ chiffres. Comparer les temps d'exécution de `puiss_1(x,n)` et `puiss_2(x,n)` lorsque x est un nombre de a chiffres (on se limitera au cas où n est une puissance de 2).

Exercice 1-10 : exponentiation itérative
Écrire un programme *itératif* calculant x^n en temps logarithmique par rapport à n (on suppose ici que le temps d'une multiplication est constant).

Exercice 1-11 : exponentiation optimale
Quel est le nombre minimal de multiplications nécessaires pour calculer x^{15} ?

Exercice 1-12 : suite de FIBONACCI
La *suite de* FIBONACCI est définie par les relations :

$$F_0 = F_1 = 1, \qquad F_{n+1} = F_n + F_{n-1} \text{ pour } n \geqslant 1.$$

1. Écrire un programme récursif calculant F_n pour $n \geqslant 0$.
2. Pour améliorer le temps d'exécution, écrire un programme récursif calculant le *couple* (F_{n-1}, F_n).
3. Démontrer la relation : $\forall\, n, p \geqslant 1,\ F_{n+p} = F_n F_p + F_{n-1} F_{p-1}$.
4. En déduire un programme de calcul de F_n selon la méthode « diviser pour régner ».

Exercice 1-13 : une fonction mystérieuse

```
let rec f(x) =
   if x <= 1 then 1
   else if x mod 2 = 0 then 2*f(x/2)
   else 1 + f(x+1)
;;
```

1. Montrer que l'appel $f(x)$ termine quel que soit l'entier x.
2. Montrer que pour tout $x \in \mathbf{N}$ le nombre $f(x) + x$ est une puissance de 2.
3. Pouvez vous décrire mathématiquement la fonction f ?

Exercice 1-14 : calcul de ppcm

Soient $a_0, \dots, a_{n-1}$, n nombres entiers naturels dont on veut calculer le plus petit commun multiple. On suppose disposer d'une fonction `ppcm2` calculant le ppcm de deux entiers.

1. Écrire un programme itératif calculant $\mathrm{ppcm}(a_0, \dots, a_{n-1})$. Les nombres a_i seront placés dans un vecteur transmis en argument à ce programme.

2. Écrire de même un programme utilisant la méthode « diviser pour régner » et comparer les performances de ces deux programmes.

Exercice 1-15 : multiplication rapide

La multiplication de Karatsuba est construite sur l'application récursive de la décomposition du produit $(a_0 + a_1x)(b_0 + b_1x)$ en trois multiplications :

$$(a_0 + a_1x)(b_0 + b_1x) = a_0b_0 + ((a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1)x + a_1b_1x^2.$$

Chercher un algorithme permettant de calculer $(a_0 + a_1x + a_2x^2)(b_0 + b_1x + b_2x^2)$ en cinq multiplications et en déduire un algorithme de multiplication polynomiale asymptotiquement plus rapide que celui de Karatsuba. Indication : le polynôme à calculer est de degré au plus 4, donc il peut être déduit de la connaissance de ses valeurs en 5 points distincts.

Chapitre 2

Structure de liste

2-1 Définitions

Une *liste* est une suite finie d'*éléments* notée $L = (a_0, a_1, \dots, a_{n-1})$ où $n \in \mathbf{N}$ est la *longueur* de la liste L et a_0, a_1, ..., a_{n-1} sont le premier, le deuxième, ..., le $n^{\text{ème}}$ objet de L. Lorsque $n = 0$ la liste ne contient aucun objet, on dit que c'est une *liste vide*. Le premier élément d'une liste est aussi appelé *tête* de la liste, et la *sous-liste* $L' = (a_1, \dots, a_{n-1})$ est la *queue* de L.

Exemples

- Coordonnées d'un point dans le plan : $L = (x, y)$. La longueur de L est 2 ; le premier objet est x (abscisse), c'est un nombre réel ; le deuxième objet est y (ordonnée), c'est aussi un nombre réel.

- Polygone dans un plan à n sommets : $L = (P_1, \dots, P_n)$. Le $i^{\text{ème}}$ objet est $P_i = (x_i, y_i)$, un point repéré par ses coordonnées. Par exemple la liste : $L = \big(\ (0,0),\ (1,0),\ (1,1),\ (0,1)\ \big)$ représente un carré. Remarquer que les éléments de L sont eux-mêmes des listes.

- Liste des élèves de la classe : chaque objet représente un élève sous la forme de son nom (chaîne de caractères), son prénom (chaîne de caractères) et sa moyenne générale (nombre décimal). L'ordre de rangement dans la liste est par exemple l'ordre alphabétique des noms.

- Liste des termes d'un polynôme : chaque objet t_i est un couple (a, e) représentant un monôme aX^e avec $a \neq 0$. Les exposants sont distincts, et les termes sont classés par ordre croissant des exposants. Par exemple le polynôme $P = X^3 - X^2 + 2$ est représenté par la liste $L = \big(\ (2,0),\ (-1,2),\ (1,3)\ \big)$.

Remarque : une liste peut contenir plusieurs fois le même objet à des positions différentes, par exemple un point du plan peut avoir deux coordonnées égales.

Opérations usuelles sur les listes

– Création et initialisation d'une liste.

– Parcours d'une liste pour effectuer un même traitement sur chaque élément. Par exemple imprimer l'élément.

– Recherche d'un élément particulier dans une liste : cet élément est spécifié par son indice ou une partie de sa valeur. Par exemple dans la liste des élèves de la classe on peut chercher le troisième dans l'ordre alphabétique, ou celui dont le nom commence par `MARTIN`. On peut aussi chercher l'élève ayant la plus forte moyenne générale. Lorsque plusieurs éléments de la liste vérifient le critère de recherche, on peut les chercher tous ou seulement le premier ou le dernier.

– Insertion d'un nouvel élément : en début, en fin ou au milieu d'une liste. Si la liste ne doit pas contenir de répétition alors l'insertion peut être précédée d'une recherche de l'élément à insérer pour s'assurer qu'il n'est pas déjà présent dans la liste.

– Permutation des éléments d'une liste pour respecter un nouvel ordre de classement. Par exemple trier les élèves de la classe par moyenne décroissante.

– Concaténation ou fusion de deux listes L_1 et L_2 : la concaténation produit une liste L constituée de tous les éléments de L_1 puis tous les éléments de L_2 ; la fusion produit une liste L' constituée de tous les éléments de L_1 et de L_2 classés suivant un certain ordre. Pour la fusion on suppose que L_1 et L_2 sont déjà triées suivant l'ordre choisi.

2-2 Représentation en mémoire

Le langage CAML définit trois structures de données permettant de représenter des listes.

Représentation par un n-uplet

La déclaration :

```
let A = (0,0) and B = (1,0) and C = (1,1) and D = (0,1);;
let L = (A,B,C,D);;
```

définit A, B, C, D comme étant des listes à deux éléments entiers et L comme étant une liste de quatre éléments couples d'entiers. Les listes définies de cette manière sont appelées « n-uplets » où n est la longueur de la liste ; elles sont essentiellement utilisées pour grouper plusieurs objets participant à un même traitement, par exemple les deux coordonnées d'un point dans un programme de dessin. Les éléments d'un n-uplet ne sont pas nécessairement de même type. L'accès aux éléments d'un n-uplet se fait en indiquant leur position :

```
let (_,_,x,_) = L in ...
```

« sélectionne » le troisième élément de L et le nomme x pour la suite des calculs. La longueur d'un n-uplet est définie implicitement par son écriture.

Représentation par un vecteur

Un vecteur est une liste d'objets de même type rangés « consécutivement » en mémoire.

```
let V = [| 1.0; 2.0; 3.0; 4.0 |];;
```

déclare une variable V de type vecteur réel dont les éléments sont $\texttt{V.}(0) = 1.0$, $\texttt{V.}(1) = 2.0$, $\texttt{V.}(2) = 3.0$ et $\texttt{V.}(3) = 4.0$. La longueur de V est $\texttt{vect\_length}(\texttt{V}) = 4$. En mémoire V est représenté par un bloc de 5 cellules consécutives :

4	1.0	2.0	3.0	4.0
ℓ	V.(0)	V.(1)	V.(2)	V.(3)

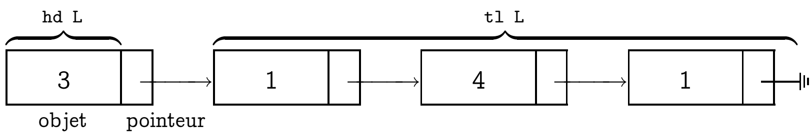
La première cellule contient la longueur du vecteur, les cellules suivantes contiennent les éléments placés par indice croissant. La longueur d'un vecteur n'est pas modifiable mais chaque élément peut être modifié « sur place », c'est-à-dire sans avoir à construire un nouveau vecteur.

Les vecteurs permettent donc de représenter des listes de longueur constante, mais on peut aussi les utiliser pour représenter des listes de longueur variable en ne « remplissant » que le début du vecteur et en conservant dans une variable à part le nombre d'éléments effectivement présents.

Représentation par une liste chaînée

Une liste chaînée est une liste d'objets de même type rangés en mémoire de manière non nécessairement consécutive. A chaque élément de la liste autre que le dernier est associé un *pointeur* qui contient l'adresse mémoire de l'élément suivant ; au dernier élément est associé un pointeur spécial marquant la fin de la liste.

```
let L = [ 3; 1; 4; 1 ];;
```



Le premier élément de la liste est $\texttt{hd L} = 3$ et la queue de la liste est $\texttt{tl L} = [1; 4; 1]$ (hd et tl sont les contractions de *head* et *tail*). Le deuxième élément est donc hd(tl L) $= 1$, le troisième hd(tl(tl L)) $= 4$ et ainsi de suite. Dans un programme CAML les pointeurs sont représentés par l'opérateur `::` et les déclarations suivantes sont équivalentes :

```
let L = [ 3; 1; 4; 1 ];;
let L = 3 :: [ 1; 4; 1 ];;
let L = 3 :: 1 :: [ 4; 1 ];;
let L = 3 :: 1 :: 4 :: [ 1 ];;
let L = 3 :: 1 :: 4 :: 1 :: [];;
```

mais : `let M = [ 3; 1 ] :: [ 4; 1 ]` est interprétée comme la demande de création de la liste : `[ [ 3; 1 ]; 4; 1 ]` ce qui provoque une erreur de typage (liste hétérogène).

2-3 Parcours d'une liste

Soit $L = (a_0, \dots, a_{n-1})$ une liste. Parcourir L consiste à « passer en revue » chaque élément a_i pour effectuer un traitement. Par exemple pour connaître la longueur d'une liste chaînée, on la parcourt en incrémentant un compteur à chaque élément rencontré. De même le calcul de la somme ou du maximum d'une liste de nombres est une opération de type « parcours » de la liste.

```
(* Applique la fonction traitement aux éléments du vecteur v *)
let parcours_vect traitement v =
    for i=0 to vect_length(v)-1 do traitement(v.(i)) done
;;

(* idem pour une liste chaînée *)
let rec parcours_liste traitement l = match l with
| [] -> ()
| a::suite -> traitement(a); parcours_liste traitement suite
;;
```

Les fonctions `parcours_vect` et `parcours_liste` sont implémentées dans la bibliothèque standard de CAML sous les noms `do_vect` et `do_list`.

2-4 Recherche d'un élément dans une liste

Recherche d'un élément par son rang

Étant donnés une liste L et un entier i on veut connaître le $i^{\text{ème}}$ élément de L. Cette situation se présente par exemple lorsque L est une table de valeurs d'une fonction f définie sur les entiers de $[\![1, n]\!]$: l'élément cherché est alors la valeur f(i).

```
(* recherche dans un vecteur *)
let cherche_vect v i =
    if (i <= 0) or (i > vect_length(v))
       then failwith "rang incorrect"
       else v.(i-1)
;;

(* recherche dans une liste *)
let rec cherche_liste l i =
    if (i <= 0) or (l = []) then failwith "rang incorrect"
    else if i = 1           then hd l
                            else cherche_liste (tl l) (i-1)
;;
```

Comparaison de ces versions : la version « vecteur » s'exécute en un temps constant tandis que la version « liste chaînée » nécessite un temps d'exécution environ proportionnel à i (pour i grand). Une liste chaînée se comporte comme une bande magnétique, il faut dérouler la bande depuis le début jusqu'à la position désirée. De plus le test de non débordement est effectué à chaque étape dans le cas d'une liste chaînée car la longueur de la liste n'est pas connue.

Recherche d'un élément par sa valeur

Étant donné une liste L on veut savoir s'il existe un ou plusieurs éléments de L vérifiant une certaine propriété et éventuellement connaître ces éléments. Considérons par exemple le travail d'un compilateur analysant un texte source. Il tient à jour la liste des identificateurs déjà déclarés, et doit accéder à cette liste :

- lors de la déclaration d'un nouvel identificateur pour déterminer s'il n'y a pas double déclaration ;

- lors de la compilation d'une expression faisant intervenir une variable pour vérifier que cette variable a été déclarée et déterminer son type, sa taille et son adresse en mémoire.

Dans cet exemple la propriété à vérifier est l'égalité du nom de l'identificateur avec le nom analysé. En principe l'identificateur cherché figure au plus une fois dans la liste des identificateurs déclarés, mais il peut être présent plusieurs fois si le compilateur supporte la notion de *portée locale des identificateurs*, auquel cas la recherche doit retourner l'identificateur le plus récemment déclaré ayant le nom analysé. On peut supposer que la liste des identificateurs est organisée de sorte que les déclarations les plus récentes figurent en début de liste, donc la recherche doit s'arrêter sur le premier élément convenant en partant du début de la liste.

```
(* Recherche si le vecteur v contient un objet convenant, et *)
(* renvoie le premier objet convenant s'il existe, sinon     *)
(* déclenche l'exception "non trouvé".                       *)
(* "convient" est une fonction booléenne disant si un objet  *)
(* convient.                                                 *)
let cherche_vect convient v =
    let i = ref 0 in
    while (!i < vect_length(v)) & not(convient(v.(!i))) do
       i := !i+1
    done;
    if !i < vect_length(v) then v.(!i)
                           else failwith "non trouvé"
;;

(* idem pour une liste chaînée *)
let rec cherche_liste convient l = match l with
    | []       -> failwith "non trouvé"
    | a::suite -> if convient(a) then a
                  else cherche_liste convient suite
;;
```

Dans les deux versions on examine successivement tous les éléments de L jusqu'à en trouver un convenant. Le temps de recherche est donc proportionnel à la position de l'objet trouvé ou à la longueur de la liste si aucun élément ne convient. Pour une liste « aléatoire » de longueur n le temps de recherche est en moyenne de $n/2$ appels à `convient` s'il y a un objet convenant, et le temps d'une recherche infructueuse est toujours de n appels à `convient`.

2-5 Insertion et suppression

Insérer un objet x dans une liste $L = (a_0, \dots, a_{n-1})$ consiste à construire une nouvelle liste L' contenant l'objet x et tous les éléments de L. Il existe plusieurs types d'insertion :

– insertion en début de liste : $L' = (x, a_0, \dots, a_{n-1})$;
– insertion en fin de liste : $L' = (a_0, \dots, a_{n-1}, x)$;
– insertion après le i-ème élément : $L' = (a_0, \dots, a_{i-1}, x, a_i, \dots, a_{n-1})$.

La suppression d'un objet est l'opération inverse. Lorsque l'ordre de classement des éléments dans L est sans importance on choisit généralement l'insertion et la suppression en début ou en fin de liste car ces positions sont les plus accessibles. Par contre si L est une liste triée et si l'on veut que L' soit aussi triée, alors il faut chercher dans L deux éléments consécutifs encadrant x et l'insérer entre ces éléments. La suppression dans une liste triée, quant à elle, ne modifie pas le caractère trié. Par ailleurs si L ne doit pas comporter de répétitions il faut s'assurer que $x \notin L$ avant de procéder à une insertion, en employant l'une des méthodes de recherche vues précédemment.

Insertion et suppression à une extrémité de la liste

```
let insère_début_vect v x =
  let n = vect_length(v)    in
  let w = make_vect (n+1) x in
  for i = 0 to n-1 do w.(i+1) <- v.(i) done;
  w
;;

let supprime_début_vect v =
  let n = vect_length(v) in
  if n = 0 then failwith "vecteur vide"
  else if n = 1 then [||]
  else begin
    let w = make_vect (n-1) v.(1) in
    for i = 2 to n-1 do w.(i-1) <- v.(i) done;
    w
  end
;;
(* insère_fin_vect et supprime_fin_vect sont analogues *)

let insère_début_liste l x = x :: l;;

let supprime_début_liste l = match l with
| []       -> failwith "liste vide"
| _::suite -> suite
;;

let rec insère_fin_liste l x = match l with
| []       -> [x]
| a::suite -> a :: (insère_fin_liste suite x)
;;

let rec supprime_fin_liste l = match l with
| []       -> failwith "liste vide"
| [_]      -> []
| a::suite -> a :: (supprime_fin_liste suite)
;;
```

Remarquer que les opérations « en fin de liste chaînée » imposent de parcourir toute la liste pour accéder au dernier élément donc elles ont un temps d'exécution environ proportionnel à la longueur de la liste tandis que les opérations « en début de liste chaînée » s'exécutent en temps constant. En ce qui concerne les vecteurs, l'insertion et la suppression modifient la longueur du vecteur considéré et imposent d'allouer en mémoire un nouveau vecteur et d'y recopier la partie utile de l'ancien, donc ont un temps d'exécution environ proportionnel à la longueur du vecteur initial. Par contre si l'on utilise un vecteur partiellement rempli alors l'insertion et la suppression peuvent être effectuées « sur place » en temps constant si la position d'insertion ou de suppression est l'extrémité « libre » du vecteur partiellement rempli.

Concaténation

Concaténer $L_1 = (a_0, \dots, a_{n-1})$ et $L_2 = (b_0, \dots, b_{p-1})$ consiste à construire la liste $(a_0, \dots, a_{n-1}, b_0, \dots, b_{p-1})$.

```
let concat_vect v1 v2 =
  let n1 = vect_length(v1) and n2 = vect_length(v2) in
  if (n1 = 0) & (n2 = 0) then [| |]
  else begin
    let x = if n1 > 0 then v1.(0) else v2.(0) in
    let w = make_vect (n1 + n2) x             in
    for i = 0 to n1-1 do w.(i)    <- v1.(i) done;
    for i = 0 to n2-1 do w.(i+n1) <- v2.(i) done;
    w
  end
;;

let rec concat_liste l1 l2 = match l1 with
| []       -> l2
| a::suite -> a :: (concat_liste suite l2)
;;
```

La fonction `concat_vect` comporte une difficulté technique car on doit fournir à `make_vect` un élément servant à initialiser le vecteur créé, même quand la longueur $n_1 + n_2$ est nulle. On utilise pour cela le premier élément de l'un des deux vecteurs v_1 ou v_2 s'il en existe, et on créée explicitement un vecteur vide dans le cas contraire.

Le temps d'exécution de `concat_vect` est environ proportionnel à la somme des longueurs des vecteurs à concaténer ; le temps d'exécution de `concat_liste` est environ proportionnel à la longueur de L_1. La bibliothèque standard de CAML comporte des fonctions de concaténation notées `concat_vect` et `@` (opérateur infixe) codées sensiblement de la même manière que celles données ci-dessus.

2-6 Exercices

Exercice 2-1 : maximum d'une liste
Écrire une fonction `maximum` qui renvoie le plus grand élément d'un vecteur entier. Même question avec une liste chaînée.

Exercice 2-2 : itération sur une liste
La *fonctionnelle* CAML standard `do_list` prend en argument une fonction `f` et une liste $\mathtt{l} = [a_0; \ldots; a_{n-1}]$ et calcule dans cet ordre les expressions $\mathtt{f}(a_0)$, $\mathtt{f}(a_1)$, ..., $\mathtt{f}(a_{n-1})$ (sans conserver les résultats). Écrire une fonctionnelle `do_list_rev` telle que `do_list_rev f` $[a_0; \ldots; a_{n-1}]$ calcule successivement les expressions $\mathtt{f}(a_{n-1})$, ..., $\mathtt{f}(a_1)$, $\mathtt{f}(a_0)$.

Exercice 2-3 : image miroir
Soit $L = (a_0, a_1, \ldots, a_{n-1})$ une liste. *L'image miroir* de L est $L' = (a_{n-1}, \ldots, a_0)$. La bibliothèque standard de CAML comporte une fonction `rev` calculant l'image miroir d'une liste. Donner deux implémentations possibles en CAML de `rev`, une utilisant la concaténation en queue `@` et une utilisant la concaténation en tête `::`. Déterminer les complexités asymptotiques de ces deux implémentations.

Exercice 2-4 : rotation d'une liste
Soit $L = (a_0, \ldots, a_{n-1})$ une liste et $k \in [\![1, n-1]\!]$. On veut *faire tourner* L de k positions vers la gauche pour obtenir la liste $L' = (a_k, \ldots, a_{n-1}, a_0, \ldots, a_{k-1})$.

1. Donner des algorithmes résolvant ce problème pour une liste chaînée et pour un vecteur ; étudier leurs complexités asymptotiques.

2. Dans le cas d'un vecteur, on peut effectuer une rotation *sur place* c'est-à-dire sans utiliser de deuxième vecteur, le vecteur initial étant alors perdu. Écrire un programme réalisant une telle rotation.

3. Le professeur TOURNESOL a présenté le programme suivant :

```
let rotation v k =
(* fait tourner v de k positions vers la gauche *)
  let n = vect_length v
  and compte = ref 0
  and i0 = ref 0 in
  while !compte < n do
    let temp = v.(!i0)
    and i = ref !i0
    and j = ref((!i0 + k) mod n) in
    while !j <> !i0 do
      v.(!i) <- v.(!j);
      i := !j;
      j := (!i + k) mod n;
      compte := !compte + 1
    done;
    v.(!i) <- temp;
```

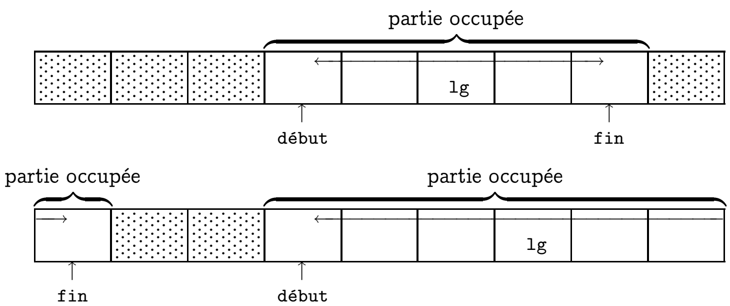
```
    compte := !compte + 1;
    i0 := !i0 + 1
  done
;;
```

Malheureusement, il a oublié de rédiger les commentaires permettant de comprendre son programme. Pourriez vous l'aider ? Indication : le faire tourner *à la main* sur quelques exemples, étudier le cas particulier où k est un diviseur de n, puis le cas général. Voyez vous un intérêt à ce programme ?

Exercice 2-5 : recherche d'un élément dans une liste
Un *prédicat* est une fonction retournant un résultat booléen, `true` si le ou les arguments passés vérifient un certain critère, `false` sinon. Étant donnés un prédicat `f` à un argument et une liste chaînée `l` on veut déterminer si `l` contient au moins un élément satisfaisant le critère défini par `f`, et le cas échéant retourner le dernier élément de `l` satisfaisant ce critère. Écrire des fonctions CAML `contient` et `dernier` résolvant ces problèmes.

Exercice 2-6 : recherche d'une sous-liste
Soient $A = (a_0, \dots, a_{n-1})$ et $B = (b_0, \dots, b_{p-1})$ deux listes dont les éléments ont même type, B étant non vide. On désire savoir si B est une sous-liste de A, c'est-à-dire s'il existe i tel que $B = (a_i, \dots, a_{i+p-1})$. Écrire des fonctions CAML `cherche_sous_liste` et `cherche_sous_vect` qui répondent à cette question pour des listes chaînées ou pour des vecteurs et renvoient le rang de la première apparition de B dans A s'il y en a une.

Exercice 2-7 : listes à double entrée
Une liste à double entrée est une structure de données permettant de représenter une liste $L = (a_0, \dots, a_{n-1})$ et d'effectuer en temps constant les opérations d'insertion en tête et en queue et l'extraction du premier ou du dernier élément de la liste. Les listes à double entrée ne sont pas prédéfinies en CAML mais elles peuvent être implémentées par des *vecteurs circulaires* c'est-à-dire des vecteurs dont les deux extrémités sont conceptuellement « reliées ». Une liste occupe un segment de ce vecteur défini par son indice de début et son indice de fin, ou mieux, par son indice de début et sa longueur effective (on peut ainsi distinguer une liste vide d'une liste pleine).



Donner les déclarations en CAML pour ce type de données et les programmes associés effectuant les opérations d'insertion en tête et en queue et d'extraction du premier et du dernier élément.

Chapitre 3

Listes triées

Une *relation de comparaison* sur un ensemble E est une relation binaire $\preccurlyeq$ sur E réflexive, transitive et telle que pour deux éléments quelconques a, b de E l'une au moins des relations $a \preccurlyeq b$ ou $b \preccurlyeq a$ est vraie. On dit que a et b sont *équivalents* pour $\preccurlyeq$ si l'on a à la fois $a \preccurlyeq b$ et $b \preccurlyeq a$, ce qui est noté $a \approx b$. Une relation d'ordre total est une relation de comparaison pour laquelle chaque élément n'est équivalent qu'à lui-même.

Par exemple un dictionnaire peut être vu comme un ensemble de définitions, une définition étant un couple `(nom,texte)` de chaînes de caractères : le nom que l'on définit et la définition proprement dite. L'ordre alphabétique sur les noms est une relation de comparaison sur les objets du dictionnaire, c'est une relation d'ordre totale si chaque nom n'a qu'une seule définition.

Une liste $L = (a_0, \dots, a_{n-1})$ d'éléments de E est dite *triée* selon $\preccurlyeq$ si l'on a $a_i \preccurlyeq a_{i+1}$ pour tout i compris entre 0 et $n-2$. Les listes triées permettent de représenter des parties d'un ensemble totalement ordonné avec la propriété que deux parties sont égales si et seulement si les listes associées le sont.

3-1 Insertion dans une liste triée

Soit L une liste triée par ordre croissant pour une relation de comparaison $\preccurlyeq$. On veut insérer un objet x « à la bonne place » dans L pour que la nouvelle liste soit encore triée. L'idée est de chercher x dans L pour déterminer à quelle place il doit être inséré, puis de procéder à l'insertion. On suppose disposer d'une fonction `compare` telle que `compare a b` renvoie l'un des résultats `PLUSGRAND` lorsque $a \succ b$, `PLUSPETIT` lorsque $a \prec b$ et `EQUIV` lorsque $a \approx b$ où `PLUSGRAND`, `PLUSPETIT` et `EQUIV` sont des *constantes symboliques* introduites par la définition suivante :

```
(* résultat d'une comparaison *)
type cmp_res = PLUSGRAND | PLUSPETIT | EQUIV;;

(* insertion dans un vecteur trié *)
let insère_vect compare v x =
  let n = vect_length(v)    in
  let w = make_vect (n+1) x in

  (* recopie les éléments de v inférieurs à x *)
  let i = ref(0) in
  while (!i < n) & (compare v.(!i) x = PLUSPETIT) do
    w.(!i) <- v.(!i);
    i := !i + 1
  done;

  (* recopie la fin de v (x est déjà en place) *)
  while !i < n do
    w.(!i+1) <- v.(!i);
    i := !i + 1
  done;

  w
;;

(* insertion dans une liste chaînée triée *)
let rec insère_liste compare l x = match l with
| []       -> [x]
| a::suite -> match compare a x with
    | PLUSPETIT -> a :: (insère_liste compare suite x)
    | _         -> x :: l
;;
```

Temps d'insertion : le temps d'insertion dans un vecteur trié de longueur n est $O(n)$ puisqu'il faut recopier tout le vecteur. Le temps d'insertion dans une liste triée de longueur n est approximativement proportionnel au nombre d'éléments de `l` inférieurs à x, c'est aussi $O(n)$ dans le pire des cas.

3-2 Recherche dans une liste triée

Considérons le problème de la recherche d'un objet x dans une liste L triée par valeurs croissantes pour une relation d'ordre total $\preccurlyeq$. L'algorithme de recherche vu au chapitre précédent peut être modifié pour interrompre la recherche dès que l'on rencontre un élément a_i de L tel que $a_i \succ x$. En effet dans ce cas x ne peut pas figurer dans L après a_i puisque L est triée par valeurs croissantes, donc on est sûr que $x \notin L$. Cette amélioration ne diminue que le temps des recherches infructueuses : pour une liste aléatoire de longueur n, le temps moyen d'une recherche infructueuse passe de n comparaisons pour une liste quelconque à $n/2$

comparaisons pour une liste triée. Une autre amélioration consiste à rechercher x dans L par dichotomie : on divise $L = (a_0, \dots, a_{n-1})$ en deux sous-listes $L_1 = (a_0, \dots, a_{i-1})$, $L_2 = (a_{i+1}, \dots, a_{n-1})$ séparées par un élément a_i puis on compare x à a_i :

- si $a_i \prec x$, soit $x \notin L$, soit $x \in L_2$;
- si $a_i \succ x$, soit $x \notin L$, soit $x \in L_1$;
- si $a_i \approx x$, alors $x \in L$ et on l'a trouvé.

Dans les deux premiers cas on itère la méthode de dichotomie à la sous-liste L_1 ou L_2 qui a été déterminée.

```
(* Recherche si l'objet x figure dans le vecteur trié v.      *)
(* Si oui, renvoie un indice de x dans v,                     *)
(* sinon lève l'exception "non trouvé".                       *)
(* "compare" est la fonction définissant la relation d'ordre. *)
let dichotomie compare v x =
    let a = ref 0
    and b = ref (vect_length(v)-1)
    and trouvé = ref false in

    while (!a <= !b) & not(!trouvé) do
      (* x n'a pas encore été trouvé, et s'il appartient à v *)
      (* alors son indice est compris entre a et b inclus.   *)
      let c = (!a + !b)/2 in
      match (compare v.(c) x) with
      | PLUSGRAND -> b := c - 1
      | PLUSPETIT -> a := c + 1
      | EQUIV     -> trouvé := true; a := c
    done;

    if !trouvé then !a else failwith "non trouvé"
;;
```

Preuve de l'algorithme : le commentaire placé dans dans la boucle `while` énonce un invariant de boucle en entrée. Vérifions sa validité.

- Au début, x n'est pas encore trouvé et s'il appartient à v alors son indice est compris entre 0 et $\texttt{vect\_length}(v) - 1$, c'est-à-dire entre a et b.
- Si la propriété est vérifiée au début de la $k^{\text{ème}}$ itération, et s'il y a une $(k+1)$-ème itération alors :
  1. L'indice c calculé est effectivement compris entre a et b que $a + b$ soit pair ou impair,
  2. Les deux premiers cas du `match` modifient a ou b sans changer le fait que x, s'il appartient à v, se trouve entre les objets d'indices a et b de v.
- La boucle `while` se termine nécessairement car tant que x n'est pas trouvé la quantité $b - a$ décroît strictement à chaque itération. En effet,

si $a + b = 2p$ alors $c = p = (a + b)/2$ donc :

$$(c - 1) - a = b - (c + 1) = (b - a)/2 - 1 < b - a\ ;$$

si $a + b = 2p + 1$ alors $c = p = (a + b - 1)/2$ donc :

$$(c - 1) - a = (b - a - 3)/2 < b - a;$$
$$b - (c + 1) = (b - a - 1)/2 < b - a.$$

– Quand la boucle est terminée, si `trouvé` vaut `true` alors x a été trouvé (troisième cas du `match`) et son indice est a tandis que si `trouvé` vaut `false` alors on a $a > b$ donc il n'existe aucun élément de v d'indice compris entre a et b dans cet ordre et en particulier $x \notin v$. ∎

Remarque : si v comporte plusieurs éléments égaux à x alors `dichotomie` en trouve un, mais on ne sait pas si c'est le premier, le dernier ou un autre.

Temps d'exécution : notons n la longueur de v et a_k, b_k les valeurs de a, b au début de la $k^{\text{ème}}$ itération de la boucle `while` : on a $b_1 - a_1 + 1 = n$ et tant que la boucle continue : $b_{k+1} - a_{k+1} + 1 \leqslant (b_k - a_k + 1)/2$ d'après les calculs précédents. Donc par récurrence on a $b_k - a_k + 1 \leqslant n/2^{k-1}$ s'il y a au moins k itérations, mais $b_k - a_k + 1 \geqslant 1$ donc $2^{k-1} \leqslant n$ c'est-à-dire $k \leqslant \log_2(n) + 1$. Ainsi, le nombre de comparaisons effectuées, qui est égal au nombre d'itérations, est au plus égal à $\lfloor \log_2(n) + 1 \rfloor$ que x appartienne à v ou non. Ce minorant est d'ailleurs atteint si x est strictement plus grand que le dernier élément de v car on a alors par récurrence $a_k = \lfloor (n + 1)(1 - 2^{1-k}) \rfloor$ et $b_k = n - 1$ au début de la $k^{\text{ème}}$ itération, donc $a_k \leqslant \lfloor n - 1/n \rfloor = b_k$ tant que $2^{k-1} \leqslant n$.

Lorsque n est grand on a $\log_2(n) + 1 = o(n)$ donc la méthode de recherche dichotomique est bien plus efficace que la recherche dans une liste non triée. Voici par exemple quelques valeurs du nombre maximal de comparaisons effectuées en fonction de n :

n	10	100	1000	10000
$\lfloor \log_2(n) + 1 \rfloor$	4	7	10	14

3-3 Fusion de listes triées

Étant données deux listes $L_1 = (a_0, \dots, a_{n-1})$ et $L_2 = (b_0, \dots, b_{p-1})$ d'objets de même type triées par ordre croissant pour une relation de comparaison $\preccurlyeq$, on veut constituer la liste L formée de tous les éléments de L_1 et L_2, triée elle aussi par ordre croissant. L'opération passant de (L_1, L_2) à L est appelée *fusion de* L_1 *et* L_2 *dans* L. La fusion apparaît dans certains algorithmes de tri, mais aussi dans le cas de mise à jour de fichiers triés : par exemple on peut vouloir fusionner les listes des élèves de deux classes pour obtenir la liste de tous les élèves de ces deux classes.

La méthode générale de fusion consiste à comparer les têtes de L_1 et L_2, a_0 et b_0, et à placer en tête de L le plus petit de ces éléments. Si c'est a_0 alors il reste à fusionner la queue de L_1 avec L_2, si c'est b_0 alors il reste à fusionner L_1 avec la queue de L_2.

Fusion de vecteurs

```
(* fusionne les deux vecteurs triés v1 et v2 dans v *)
(* "compare" est la fonction de comparaison.       *)
let fusion_vec compare v1 v2 v =

    let i = ref 0    (* indice de v1 *)
    and j = ref 0    (* indice de v2 *)
    and k = ref 0    (* indice de v  *)
    in

    while (!i < vect_length(v1)) & (!j < vect_length(v2)) do
      match compare v1.(!i) v2.(!j) with
      | PLUSGRAND -> v.(!k) <- v2.(!j); k := !k+1; j := !j+1
      | _         -> v.(!k) <- v1.(!i); k := !k+1; i := !i+1
    done;

    (* ici, un des deux vecteurs est épuisé *)
    (* on recopie la fin de l'autre         *)

    while !i < vect_length(v1) do
       v.(!k) <- v1.(!i); k := !k+1; i := !i+1
    done;

    while !j < vect_length(v2) do
       v.(!k) <- v2.(!j); k := !k+1; j := !j+1
    done
;;
```

Preuve du programme

On démontre par une récurrence immédiate la propriété :

> *à l'entrée de la première boucle* `while`*, les éléments* $v_1.(0\ ..\ !i-1)$ *et* $v_2.(0\ ..\ !j-1)$ *sont les* $!k$ *plus petits éléments de* $v_1 \cup v_2$ *et ont été placés dans le bon ordre dans* $v.(0\ ..\ !k-1)$.

La boucle s'arrête quand $!i \geqslant$ `vect_length`(v_1) ou $!j \geqslant$ `vect_length`(v_2) ce qui se produit en un nombre fini d'étapes car l'une des quantités $!i$ ou $!j$ augmente d'une unité à chaque itération. A la fin de la boucle un au moins des vecteurs v_1 ou v_2 a été transféré entièrement dans v (et un seulement si v_1 et v_2 ne sont pas initialement vides, car la boucle ne place dans v qu'un seul élément à la fois). Les deux boucles `while` suivantes recopient s'il y a lieu la fin de l'autre vecteur à la fin de v (au plus une de ces deux boucles est effectivement exécutée). ∎

Temps d'exécution

La première boucle `while` effectue une comparaison entre un objet de v_1 et un objet de v_2 à chaque fois qu'elle place un objet dans v. Les deux autres boucles n'effectuant pas de comparaisons, le nombre total de comparaisons effectuées, N_{comp}, vérifie :

$$\min(\ell_1, \ell_2) \leqslant N_{\mathrm{comp}} \leqslant \ell_1 + \ell_2 - 1$$

où ℓ_1 et ℓ_2 désignent les longueurs des vecteurs v_1 et v_2. Ces deux bornes peuvent être atteintes : $N_{\mathrm{comp}} = \min(\ell_1, \ell_2)$ lorsque le plus court des deux vecteurs vient entièrement avant l'autre dans v, et $N_{\mathrm{comp}} = \ell_1 + \ell_2 - 1$ lorsque le dernier élément d'un vecteur est strictement compris entre les deux derniers éléments de l'autre.

Nombre de transferts : un transfert est la copie d'un objet d'un vecteur dans un autre. Comme chaque élément de v est placé par un seul transfert, le nombre total de transferts effectués est $N_{\mathrm{transf}} = \ell_1 + \ell_2$.

Le temps d'exécution d'une itération de la première boucle `while` est la somme des temps d'une comparaison, d'un transfert et de la gestion des variables i, j, k. En CAML le temps d'un transfert est constant puisque le transfert se résume à une copie de pointeur. Ainsi, en supposant que `compare` a un temps d'exécution borné, on voit que le temps de fusion de deux vecteurs de longueurs ℓ_1 et ℓ_2 est $O(\ell_1 + \ell_2)$.

Fusion de listes chaînées

```
(* fusionne les listes l1 et l2.           *)
(* "compare" est la fonction de comparaison. *)
let rec fusion_liste compare l1 l2 = match (l1,l2) with
| ([], _) -> l2
| (_, []) -> l1
| ((a::suite1),(b::suite2)) -> match compare a b with
    | PLUSGRAND -> b :: (fusion_liste compare l1 suite2)
    | _         -> a :: (fusion_liste compare suite1 l2)
;;
```

Temps d'exécution : chaque appel à `fusion_liste` « place » un élément dans la liste résultat en cours de construction par une comparaison et une concaténation en tête, donc le temps de fusion de deux listes L_1, L_2 de longueurs ℓ_1, ℓ_2 est $O(\ell_1 + \ell_2)$ si l'on suppose que `compare` a un temps d'exécution borné.

3-4 Tri d'une liste

Soit $L = (a_0, \dots, a_{n-1})$ une liste d'éléments comparables. *Trier* L consiste à construire une liste L' triée contenant les mêmes éléments que L, en conservant les répétitions. *Trier sur place* un vecteur consiste à permuter les éléments du vecteur de sorte qu'ils forment une liste triée, en utilisant une mémoire auxiliaire de taille bornée (indépendante de la taille du vecteur à trier).

Les algorithmes usuels de tri se rangent en deux catégories :

- tri en temps quadratique : le temps de tri est asymptotiquement proportionnel au carré de la taille de la liste à trier (tri à bulles) ;
- tri en temps quasi linéaire : le temps de tri dans le pire des cas ou en moyenne est asymptotiquement proportionnel à $n\ln(n)$ où n est la taille de la liste à trier (tri par fusion, quicksort).

Un algorithme de *tri par comparaisons* est un algorithme de tri dans lequel il n'est effectué que des comparaisons entre éléments pour décider quelle permutation de la liste L doit être réalisée. On démontre qu'un tel algorithme doit effectuer au moins $\lceil \log_2(n!) \rceil \approx n\log_2(n)$ comparaisons pour trier une liste quelconque de longueur n (cf. section **7-4**). Un algorithme de tri par comparaisons ne peut donc avoir une complexité dans le pire des cas négligeable devant $n\ln(n)$. Le tri par fusion atteint cette complexité minimale si l'on suppose que le temps d'une comparaison est constant (constant signifie ici indépendant de n) ; on dit que c'est un algorithme de tri *optimal*.

En ce qui concerne la complexité moyenne, on démontre que le nombre moyen de comparaisons effectuées par un algorithme de tri par comparaisons est supérieur ou égal à $\log_2(n!-1)$ lorsque la moyenne est calculée sur les $n!$ permutations des éléments d'un ensemble totalement ordonné de cardinal n (cf. exercice **7-19**). La complexité moyenne d'un algorithme de tri par comparaisons ne peut donc, elle non plus, être négligeable devant $n\ln(n)$ dans ce modèle d'équirépartition des permutations. Le tri par fusion et le quicksort atteignent cette complexité en moyenne lorsque le temps d'une comparaison est constant, ce sont des tris *optimaux en moyenne*.

Par ailleurs il existe des algorithmes de tri ne procédant pas par comparaisons, et donc pour lesquels les minorations de complexité précédentes ne sont pas applicables. En particulier, le tri par distribution est un algorithme de tri ayant un temps d'exécution approximativement proportionnel à la taille de la liste à trier (cf. le problème « Tri par distribution » dans la partie « Problèmes »).

Un algorithme de tri est dit *stable* si la disposition relative des éléments équivalents dans L est conservée dans L'. Par exemple, si l'on dispose d'une liste alphabétique des élèves de la classe que l'on veut trier par moyenne décroissante, il est souhaitable que des élèves ayant même moyenne restent classés par ordre alphabétique pour ne pas faire de jaloux. Remarquons que l'on peut toujours rendre stable un algorithme de tri en définissant la fonction de comparaison de telle sorte qu'elle tienne compte des indices des objets qu'elle compare lorsqu'ils sont équivalents (ceci peut être obtenu en ajoutant à chaque objet un champ contenant son indice initial).

3-5 Tri à bulles

On parcourt la liste à trier en examinant si chaque couple d'éléments consécutifs, (a_i, a_{i+1}), est dans le bon ordre $(a_i \preccurlyeq a_{i+1})$ ou est mal classé $(a_i \succ a_{i+1})$. Dans ce dernier cas on échange les éléments du couple et le processus est répété tant qu'il reste des couples mal classés.

```
(* trie le vecteur v par ordre croissant   *)
(* compare est la fonction de comparaison  *)
let tri_bulles compare v =

    let fini = ref false in
    while not !fini do

      fini := true;  (* espérons *)

      for i = 0 to vect_length(v)-2 do
        if (compare v.(i) v.(i+1)) = PLUSGRAND
        then begin
           (* un couple est mal classé, échange les éléments *)
           let x = v.(i) in
           v.(i)   <- v.(i+1);
           v.(i+1) <- x;
           fini := false       (* il faut refaire une passe *)
        end
      done (* for i *)

    done   (* while not !fini *)
;;
```

Preuve de l'algorithme : notons N_{inv} le nombre de couples (i, j) tels que $i < j$ et $a_i \succ a_j$ (N_{inv} est le nombre d'*inversions* du vecteur v) et examinons une itération de la boucle `while` :

– s'il existe un indice i tel que $a_i \succ a_{i+1}$ alors ces éléments sont échangés et `fini` est mis à `false` donc le nombre d'inversions diminue d'une unité et le test de fin de boucle sera négatif ;

– si pour tout i on a $a_i \preccurlyeq a_{i+1}$ alors v est trié et `fini` garde la valeur `true` tout au long de la boucle, donc cette itération est la dernière.

Donc l'algorithme ne s'arrête que lorsque le vecteur est trié, et il s'arrête nécessairement puisque N_{inv} diminue strictement à chaque itération tant que le vecteur n'est pas trié. ∎

On peut prouver l'arrêt d'une autre manière en vérifiant par récurrence que, à la fin de la $k^{\text{ème}}$ itération de la boucle `while`, les k derniers éléments de v sont les k éléments les plus grands et sont placés dans l'ordre croissant. Il en résulte que le nombre d'itérations est majoré par la longueur de v. Par ailleurs, on peut remplacer la borne `vect_length(v)-2` de la boucle `for` par une borne variable, initialisée à `vect_length(v)-2` et décrémentée d'une unité à la fin de chaque itération de la boucle `while`.

Temps d'exécution : le temps total passé dans le bloc `begin`... `end` est environ proportionnel au nombre d'échanges et on a remarqué que chaque échange diminue d'une unité le nombre d'inversions, donc il y a exactement N_{inv} échanges effectués, et $N_{inv} \leqslant C_n^2$ (nombre de couples (i,j) tels que $i < j$). Le nombre d'appels à la fonction `compare` est égal au produit de $n-1$ par le nombre d'itérations donc est majoré par $n(n-1)$. Ces majorants sont atteints quand le vecteur v est « trié à l'envers » c'est-à-dire quand $a_i \succ a_{i+1}$ pour tout i. Par conséquent le temps d'exécution maximal est $O(n^2)$ en supposant que `compare` a un temps d'exécution borné.

Stabilité : soient $a_i \approx a_j$ avec $i \neq j$. A chaque échange la distance entre a_i et a_j ne peut diminuer que d'une unité au plus, et si ces éléments deviennent adjacents alors ils ne peuvent être échangés puisqu'un échange porte toujours sur des éléments non équivalents. Donc la disposition relative de a_i et a_j reste constante au cours du tri ; l'algorithme de tri à bulles est stable.

Complexité spatiale : la fonction `tri_bulles` utilise seulement trois variables locales (`fini`, `i` et `j`) donc le tri à bulles est un tri sur place.

3-6 Tri par fusion

On divise la liste $L = (a_0, \ldots, a_{n-1})$ que l'on veut trier en deux demi listes, $L_1 = (a_0, \ldots, a_{p-1})$ et $L_2 = (a_p, \ldots, a_{n-1})$ que l'on trie séparément récursivement, puis on fusionne les deux listes triées obtenues L_1' et L_2'.

Tri par fusion pour un vecteur

```
(* fusionne les sous-vecteurs triés v1(a..c) et v1(c+1..b) *)
(* dans v2(a..b). On suppose a <= c < b.                    *)
let fusion compare v1 v2 a c b =
  ...
(* adapter ici le code de fusion_vect donné en section 3-3 *)
;;

(* Trie par ordre croissant v1(a..b) à l'aide du vecteur   *)
(* auxiliaire v2. Si vers_v1 = true, le résultat est placé *)
(* dans v1(a..b), sinon dans v2(a..b). On suppose a <= b.  *)
let rec tri_partiel compare v1 v2 a b vers_v1 =

    if a < b then begin
       let c = (a+b)/2 in
       tri_partiel compare v1 v2   a   c (not vers_v1);
       tri_partiel compare v1 v2 (c+1) b (not vers_v1);
```

```
        if vers_v1 then fusion compare v2 v1 a c b
                   else fusion compare v1 v2 a c b
     end

     (* cas a = b : transfère l'élément au besoin *)
     else if not vers_v1 then v2.(a) <- v1.(a)
;;

(* trie par ordre croissant le vecteur v *)
let tri_fusion compare v =
    let n  = vect_length(v) in
    if n > 1 then begin
       let v' = make_vect n v.(0) in
       tri_partiel compare v v' 0 (n-1) true
    end
;;
```

Le tri est réalisé avec un vecteur auxiliaire v' permettant de fusionner les deux sous-vecteurs qui ont été récursivement triés. A chaque niveau de récursion la fusion se fait de v vers v' ou de v' vers v suivant la valeur de l'indicateur `vers_v1`.

Preuve de l'algorithme : on démontre la correction de `tri_partiel` par récurrence sur $n = b - a + 1$. Pour $n = 1$ on a $b = a$ donc le sous-vecteur $v_1(a\ ..\ b)$ est déjà trié, et il est correctement placé dans v_1 ou dans v_2 selon la valeur de l'indicateur `vers_v1`.

Supposons que `tri_partiel` fournit un résultat juste pour tous les sous-vecteurs de longueur $p \in [\![1, n[\![$ avec $n \geqslant 2$, et considérons le tri d'un sous-vecteur $v_1(a\ ..\ b)$ avec $b = a + n - 1$. On a $c = \lfloor (a+b)/2 \rfloor = a + \lfloor (n-1)/2 \rfloor \in [\![a, b[\![$ donc les appels :

```
tri_partiel compare v1 v2   a   c (not vers_v1);
tri_partiel compare v1 v2 (c+1) b (not vers_v1);
```

sont licites et portent sur des sous-vecteurs de longueurs :

$$\lfloor (n-1)/2 \rfloor + 1 = \lceil n/2 \rceil \quad \text{et} \quad n - \lceil n/2 \rceil = \lfloor n/2 \rfloor.$$

Ces longueurs sont strictement inférieures à n car $n \geqslant 2$, donc, d'après l'hypothèse de récurrence, les appels récursifs trient $v_1(a\ ..\ c)$ et $v_1(c+1\ ..\ b)$ en plaçant le résultat dans v_2 ou dans v_1 selon que `vers_v1` = `true` ou `vers_v1` = `false`. La fusion les deux sous-vecteurs obtenus fournit alors le résultat attendu : un sous-vecteur trié et placé dans le vecteur désigné par l'indicateur `vers_v1`. Ceci achève la démonstration de validité. ∎

Temps d'exécution. Le temps nécessaire pour trier un vecteur v est de la forme :

$$T = \alpha N_{\mathrm{comp}} + \beta N_{\mathrm{transf}} + \gamma N_{\mathrm{rec}}$$

étape	v	v'	a c b	vers_v1	à faire
1	31415	xxxxx	0 2 4	`true`	2: `tri_partiel v v' 0 2 false` 3: `tri_partiel v v' 3 4 false` 4: `fusion v' v 0 2 4`
2	31415	xxxxx	0 1 2	`false`	5: `tri_partiel v v' 0 1 true` 6: `tri_partiel v v' 2 2 true` 7: `fusion v v' 0 1 2`
5	31415	xxxxx	0 0 1	`true`	8: `tri_partiel v v' 0 0 false` 9: `tri_partiel v v' 1 1 false` 10: `fusion v' v 0 0 1`
8	31415	xxxxx	0 0	`false`	$v'(0) \longleftarrow v(0)$
9	31415	3xxxx	1 1	`false`	$v'(1) \longleftarrow v(1)$
10	31415	31xxx	0 0 1		$v(0..1) \longleftarrow \text{fusion}(v'(0..0), v'(1..1))$
6	13415	31xxx	2 2	`true`	
7	13415	31xxx	0 1 2		$v'(0..2) \longleftarrow \text{fusion}(v(0..1), v(2..2))$
3	13415	134xx	3 3 4	`false`	11: `tri_partiel v v' 3 3 true` 12: `tri_partiel v v' 4 4 true` 13: `fusion v v' 3 3 4`
11	13415	134xx	3 3	`true`	
12	13415	134xx	4 4	`true`	
13	13415	134xx	3 3 4		$v'(3..4) \longleftarrow \text{fusion}(v(3..3), v(4..4))$
4	13415	13415	0 2 4		$v(0..4) \longleftarrow \text{fusion}(v'(0..2), v'(3..4))$
	11345				

Figure 3 : tri par fusion du vecteur $[|3;1;4;1;5|]$

où N_{comp} est le nombre d'appels à la fonction `compare`, N_{transf} est le nombre de transferts d'un élément de l'un des vecteurs v_1 ou v_2 dans l'autre vecteur, et N_{rec} est le nombre d'appels récursifs à la fonction `tri_partiel`. Les coefficients α, β, γ représentent les durées d'une comparaison, d'un transfert et de la gestion des variables locales aux fonctions `tri_partiel` et `fusion`.

Notons $N_{comp}(n)$, $N_{transf}(n)$ et $N_{rec}(n)$ les valeurs maximales de N_{comp}, N_{transf} et N_{rec} pour le tri d'un sous-vecteur $v_1(a\ ..\ b)$ de longueur $n = b - a + 1$. Si $n = 1$ on a :

$$N_{comp}(1) = 0, \qquad N_{transf}(1) = 1, \qquad N_{rec}(1) = 1.$$

Si $n \geqslant 2$ alors on divise $v_1(a\ ..\ b)$ en deux sous-vecteurs $v_1(a\ ..\ c)$ et $v_1(c+1\ ..\ b)$ de longueurs $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor$, que l'on trie récursivement, d'où :

$$\begin{aligned} N_{comp}(n) &= N_{comp}(\lceil n/2 \rceil) + N_{comp}(\lfloor n/2 \rfloor) + \lceil n/2 \rceil + \lfloor n/2 \rfloor - 1, \\ N_{transf}(n) &= N_{transf}(\lceil n/2 \rceil) + N_{transf}(\lfloor n/2 \rfloor) + \lceil n/2 \rceil + \lfloor n/2 \rfloor, \\ N_{rec}(n) &= N_{rec}(\lceil n/2 \rceil) + N_{rec}(\lfloor n/2 \rfloor) + 1. \end{aligned}$$

On en déduit, par récurrence sur n :

$$N_{\text{rec}}(n) = 2n - 1 ;$$
$$N_{\text{comp}}(n) + N_{\text{rec}}(n) = N_{\text{transf}}(n) ;$$
$$\text{si } 2^k \leqslant n \leqslant 2^{k+1} \text{ alors } (k+1)n \leqslant N_{\text{transf}}(n) \leqslant (k+2)n.$$

D'après ces relations, $N_{\text{transf}}(n) \sim n\log_2(n)$, $N_{\text{rec}}(n) = o(n\log_2 n)$ et donc $N_{\text{comp}}(n) \sim n\log_2(n)$. Finalement :

le tri par fusion d'un vecteur de taille n *s'exécute en temps* $O(n\log_2 n)$ *si le temps d'une comparaison est constant.*

Stabilité : la position relative de deux éléments équivalents ne peut être modifiée qu'au cours d'une fusion, et l'algorithme de fusion décrit en section **3-3** ne permute jamais deux éléments équivalents appartenant au même sous-vecteur, et place en premier les éléments du premier sous-vecteur en cas d'équivalence avec des éléments du deuxième. Il en résulte que l'algorithme de tri par fusion tel qu'il a été décrit est stable.

Complexité spatiale : l'algorithme de tri par fusion nécessite un vecteur auxiliaire v' de longueur n et une pile de récursion dans laquelle sont stockées les variables locales aux différentes instances de `tri_partiel` et de `fusion` actives. La taille de cette pile de récursion est proportionnelle au nombre maximum d'appels à `tri_partiel` imbriqués, et on voit par récurrence sur n que ce nombre est au plus égal à $1 + \lceil \log_2 n \rceil$. Donc *le tri par fusion d'un vecteur de longueur* n *a une complexité spatiale asymptotiquement proportionnelle à* n. En particulier, le tri par fusion n'est pas un tri sur place.

Tri par fusion pour une liste chaînée

```
(* Découpe l en deux sous-listes de tailles n et lg(l)-n *)
(* Il est supposé que lg(l) >= n                         *)
let rec découpe l n = match n,l with
| 0,_        -> ([], l)
| _,x::suite -> let (a,b) = découpe suite (n-1) in (x::a, b)
| _,_        -> failwith "cas impossible"
;;

(* Trie par fusion la liste l de n éléments *)
let rec tri_partiel compare l n =
  if n <= 1 then l
  else let n1 = n/2  in
       let n2 = n-n1 in
       let (l1, l2) = découpe l n1 in
       fusion_liste compare (tri_partiel compare l1 n1)
                            (tri_partiel compare l2 n2)
;;
```

```
(* Trie par fusion la liste l *)
let tri_fusion compare l = tri_partiel compare l (list_length l)
;;
```

3-7 Tri rapide

Le *tri rapide*, aussi appelé *tri de* HOARE, *tri par segmentation, ou quick-sort,* consiste à réorganiser une liste $L = (a_0, \ldots, a_{n-1})$ en trois sous-listes :

$$L_1 = (b_0, \ldots, b_{p-1}), \qquad L_2 = (a_0), \qquad L_3 = (c_0, \ldots, c_{q-1})$$

dont la concaténation est une permutation de L et telles que tous les éléments de L_1 sont inférieurs ou équivalents à a_0 et tous les éléments de L_3 sont supérieurs ou équivalents à a_0. Pour trier L, il reste à trier séparément L_1 et L_3 ce que l'on fait récursivement.

```
(* échange les éléments d'indices i et j dans v *)
let échange v i j = let x = v.(i) in v.(i) <- v.(j); v.(j) <- x;;

(* On suppose a < b et on note x = v.(a).                      *)
(* Réorganise v(a..b) de sorte qu'en sortie tous les éléments *)
(* d'indice < c soient inférieurs à x, tous ceux d'indice > c *)
(* soient plus grands que x et v(c) = x.                       *)
(* Retourne c en résultat.                                     *)
let segmentation compare v a b =

    let i = ref(a+1) and j = ref(b) and x = v.(a) in
    while !i <= !j do

      (* ici, tous les éléments de v(a+1..i-1) sont <= x *)
      (*   et tous les éléments de v(j+1..b)   sont >= x *)
      (* avance i et recule j tant que cette propriété   *)
      (* reste vraie et que l'on a i <= j.               *)
      while (!i <= !j) & (compare x v.(!i)) <> PLUSPETIT
      do i := !i+1 done;
      while (!j > !i)  & (compare x v.(!j)) <> PLUSGRAND
      do j := !j-1 done;

      (* échange les éléments trouvés *)
      if !i < !j then begin
         échange v !i !j;
         i := !i+1;
         j := !j-1;
       end
       else if !i = !j then j := !j-1;

    done; (* while !i <= !j *)
```

```
        (* met x en place et retourne sa nouvelle position *)
        if a <> !j then échange v a !j;
        !j
;;

(* trie le sous-vecteur v(a..b) par ordre croissant *)
(* on suppose a < b                                 *)
let rec tri_partiel compare v a b =
    let c = segmentation compare v a b in
    if a < c-1 then tri_partiel compare v a (c-1);
    if c+1 < b then tri_partiel compare v (c+1) b
;;

(* trie le vecteur v par ordre croissant  *)
let tri_rapide compare v =
    let n = vect_length(v) in
    if n >= 2 then tri_partiel compare v 0 (n-1)
;;
```

Preuve de correction pour la fonction `segmentation` **:** la propriété placée en commentaire au début de la boucle `while !i <= !j` se démontre aisément par récurrence sur le nombre d'itérations effectuées. Lorsque cette boucle est terminée, on a $!i = !j+1$, $!j \geqslant a$ et $v.(!j) \preccurlyeq x$. L'échange de $v.(!j)$ et $v.(a)$ est donc licite et produit les listes L_1, L_2 et L_3 annoncées. La figure 4 illustre le tri rapide du vecteur $[|1;4;1;4;2;1;3;5|]$. ■

Temps d'exécution. Le temps nécessaire pour trier un vecteur v de longueur n avec $n \geqslant 2$ est de la forme :

$$T = \alpha N_{\mathrm{comp}} + \beta N_{\mathrm{ech}} + \gamma N_{\mathrm{rec}}$$

où N_{comp} est le nombre d'appels à la fonction `compare`, N_{ech} est le nombre d'appels à la fonction `échange` et N_{rec} est le nombre d'appels à la fonction `tri_partiel`. Les coefficients α, β, γ représentent les durées d'une comparaison, d'un échange et de la gestion des variables locales aux fonctions `tri_partiel` et `segmentation`.

Chaque appel à `tri_partiel` pour un sous-vecteur $v(a\,..\,b)$ tel que $a < b$ met à sa place définitive au moins un élément, $v(c)$. On a donc $N_{\mathrm{rec}} \leqslant n$.

Montrons par récurrence sur n que le nombre d'échanges est majoré par $\frac{1}{2} n\log_2(n)$. C'est immédiat si $n = 2$. Si $n > 2$, soit c l'indice retourné par la segmentation de v (avec $a = 0$ et $b = n-1$). Si $2 \leqslant c \leqslant n-3$, alors :

$$N_{\mathrm{ech}} \leqslant \frac{c\log_2(c) + (n-c-1)\log_2(n-c-1)}{2} \leqslant \frac{(n-1)\log_2(n)}{2} \leqslant \frac{n\log_2(n)}{2}$$

par hypothèse de récurrence. On obtient la même inégalité lorsque $c < 2$ ou $c > n-3$ en remplaçant le terme $c\log_2(c)$ ou $(n-c-1)\log_2(n-c-1)$ par 0. ■

<table>
<tr><th colspan="8">v</th><th>action</th></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td><code>segmentation</code>(0, 7)</td></tr>
<tr><td>[|1</td><td>$\underset{i}{4}$</td><td>1</td><td>4'</td><td>2</td><td>1</td><td>3</td><td>$\underset{j}{5}$|]</td><td><code>while !i <= !j</code></td></tr>
<tr><td>[|$\underset{j}{1}$</td><td>$\underset{i}{4}$</td><td>1</td><td>4'</td><td>2</td><td>1</td><td>3</td><td>5|]</td><td><code>done</code></td></tr>
<tr><td>1</td><td>[|4</td><td>1</td><td>4'</td><td>2</td><td>1</td><td>3</td><td>5|]</td><td>c = 0</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td><code>segmentation</code>(1, 7)</td></tr>
<tr><td>1</td><td>[|4</td><td>$\underset{i}{1}$</td><td>4'</td><td>2</td><td>1</td><td>3</td><td>$\underset{j}{5}$|]</td><td><code>while !i <= !j</code></td></tr>
<tr><td>1</td><td>[|4</td><td>1</td><td>4'</td><td>2</td><td>1</td><td>$\underset{j}{3}$</td><td>$\underset{i}{5}$|]</td><td><code>done</code></td></tr>
<tr><td>1</td><td>[|4</td><td>1</td><td>4'</td><td>2</td><td>1</td><td>$\underset{j}{3}$</td><td>$\underset{i}{5}$|]</td><td><code>échange</code>(1, 6)</td></tr>
<tr><td>1</td><td>[|3</td><td>1</td><td>4'</td><td>2</td><td>1|]</td><td>4</td><td>[|5|]</td><td>c = 6</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td><code>segmentation</code>(1, 5)</td></tr>
<tr><td>1</td><td>[|3</td><td>$\underset{i}{1}$</td><td>4'</td><td>2</td><td>$\underset{j}{1}$|]</td><td>4</td><td>5</td><td><code>while !i <= !j</code></td></tr>
<tr><td>1</td><td>[|3</td><td>1</td><td>$\underset{i}{4'}$</td><td>2</td><td>$\underset{j}{1}$|]</td><td>4</td><td>5</td><td><code>échange</code>(3, 5)</td></tr>
<tr><td>1</td><td>[|3</td><td>1</td><td>1</td><td>$\underset{ij}{2}$</td><td>4'|]</td><td>4</td><td>5</td><td><code>while !i <= !j</code></td></tr>
<tr><td>1</td><td>[|3</td><td>1</td><td>1</td><td>$\underset{j}{2}$</td><td>$\underset{i}{4'}$|]</td><td>4</td><td>5</td><td><code>done</code></td></tr>
<tr><td>1</td><td>[|3</td><td>1</td><td>1</td><td>$\underset{j}{2}$</td><td>$\underset{i}{4'}$|]</td><td>4</td><td>5</td><td><code>échange</code>(1, 4)</td></tr>
<tr><td>1</td><td>[|2</td><td>1</td><td>1|]</td><td>3</td><td>[|4'|]</td><td>4</td><td>5</td><td>c = 4</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td><code>segmentation</code>(1, 3)</td></tr>
<tr><td>1</td><td>[|2</td><td>$\underset{i}{1}$</td><td>$\underset{j}{1}$|]</td><td>3</td><td>4'</td><td>4</td><td>5</td><td><code>while !i <= !j</code></td></tr>
<tr><td>1</td><td>[|2</td><td>1</td><td>$\underset{j}{1}$|]</td><td>$\underset{i}{\mathbf{3}}$</td><td>4'</td><td>4</td><td>5</td><td><code>done</code></td></tr>
<tr><td>1</td><td>[|2</td><td>1</td><td>$\underset{j}{1}$|]</td><td>$\underset{i}{\mathbf{3}}$</td><td>4'</td><td>4</td><td>5</td><td><code>échange</code>(1, 3)</td></tr>
<tr><td>1</td><td>[|1</td><td>1|]</td><td>2[| |]</td><td>3</td><td>4'</td><td>4</td><td>5</td><td>c = 3</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td><code>segmentation</code>(1, 2)</td></tr>
<tr><td>1</td><td>[|1</td><td>$\underset{ij}{1}$|]</td><td>2</td><td>3</td><td>4'</td><td>4</td><td>5</td><td><code>while !i <= !j</code></td></tr>
<tr><td>1</td><td>[|1</td><td>$\underset{j}{1}$|]</td><td>$\underset{i}{\mathbf{2}}$</td><td>3</td><td>4'</td><td>4</td><td>5</td><td><code>done</code></td></tr>
<tr><td>1</td><td>[|1</td><td>$\underset{j}{1}$|]</td><td>$\underset{i}{\mathbf{2}}$</td><td>3</td><td>4'</td><td>4</td><td>5</td><td><code>échange</code>(1, 2)</td></tr>
<tr><td>1</td><td>[|1|]</td><td>1[| |]</td><td>2</td><td>3</td><td>4'</td><td>4</td><td>5</td><td>c = 2</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>2</td><td>3</td><td>4'</td><td>4</td><td>5</td><td></td></tr>
</table>

Figure 4 : tri rapide du vecteur [|1; 4; 1; 4; 2; 1; 3; 5|]

En remarquant que `segmentation` effectue $n-1$ appels à `compare` pour un vecteur de longueur n, on montre de même que $N_{comp} \leqslant \frac{1}{2}n(n-1)$. Ce majorant est atteint, entre autres, lorsque le vecteur v est déjà trié : en effet dans ce cas, la segmentation de v compare $v(0)$ aux $n-1$ autres éléments de v et retourne $c=0$, donc on est ramené au tri du sous-vecteur $v(1\,..\,n-1)$ lui aussi déjà trié.

En conclusion, le tri rapide d'un vecteur de taille n *s'exécute en temps* $O(n^2)$ *si le temps d'une comparaison est constant, et cette complexité quadratique est atteinte lorsque le vecteur est déjà trié.*

Ainsi le tri rapide n'est pas particulièrement rapide si l'on considère la complexité dans le pire des cas. Cependant, on montre que pour une liste « aléatoire » de n éléments distincts, le nombre moyen de comparaisons effectuées est de l'ordre de $2n\ln(n) \approx 1.4n\log_2(n)$ (cf. exercice **8-6** et aussi [KNUTH] vol. 3, p. 114). Il en résulte que le temps moyen d'exécution du tri rapide est $O(n \ln n)$ si le temps d'une comparaison est constant.

Complexité spatiale : la segmentation d'un sous vecteur est effectuée sur place puisque la fonction `segmentation` utilise une nombre fixe de variables locales. Par contre, la fonction `tri_partiel` ne s'exécute pas sur place du fait des appels récursifs. Dans le pire des cas il peut y avoir jusqu'à $n-1$ instances de `tri_partiel` en suspens si le vecteur v est initialement trié à l'envers (c'est-à-dire $v(i) \succ v(i+1)$ pour tout i), donc la complexité mémoire de `tri_rapide` dans le pire des cas est asymptotiquement proportionnelle à n. L'exercice **3-12** montre que l'on peut effectuer un tri rapide avec $O(\ln n)$ unités de mémoire auxiliaire seulement, en ordonnant convenablement les appels récursifs à `tri_partiel`. En tout état de cause, le tri rapide n'est pas un tri sur place.

Stabilité : l'exemple du tri de $[|1;4;1;4;2;1;3;5|]$ montre que *le tri rapide n'est pas stable* (les deux 4 sont intervertis). On peut le rendre stable en utilisant un *vecteur d'indexation*, c'est-à-dire un vecteur p à éléments entiers dans lequel on calcule une permutation de $[\![0, n-1]\!]$ telle que la liste $(v.(p.(0)), \ldots, v.(p.(n-1)))$ soit triée, avec conservation de la disposition relative des éléments équivalents. Le vecteur p s'obtient en triant la permutation identité avec une fonction de comparaison entre indices qui compare les éléments de v ayant ces indices, puis compare les indices en cas d'équivalence.

```
(* Calcule une permutation p telle que la liste *)
(* [v(p(0)), .., v(p(n-1))] est triée par ordre *)
(* croissant. En cas d'équivalence, utilise les *)
(* indices des éléments pour les départager.    *)
let tr_stable compare v =

  let n = vect_length(v) in
  let p = make_vect n 0  in
  for i = 1 to n-1 do p.(i) <- i done;

  let comp x y = match compare v.(x) v.(y) with
  | EQUIV when x < y -> PLUSPETIT
```

```
    | EQUIV when x > y -> PLUSGRAND
    | res              -> res
    in

    tri_rapide comp p; p
  ;;
```

Avec cette méthode, on obtient un algorithme de tri stable qui ne modifie pas le vecteur à trier, ce qui peut être intéressant dans certaines applications.

3-8 Exercices

Exercice 3-1 : insertion sans répétition
Modifier les fonctions `insère_vect` et `insère_liste` présentées à la section **3-1** pour qu'elles ne procèdent pas à l'insertion si le vecteur ou la liste fourni contient déjà un élément équivalent à `x`.

Exercice 3-2 : fusion sans répétition
Programmer en CAML les opérations de fusion sans répétition pour des listes chaînées et pour des listes représentées par des vecteurs (on supposera que les listes à fusionner ne comportent pas de répétitions).

Exercice 3-3 : opérations sur les ensembles
On représente les parties d'un ensemble totalement ordonné E par des listes triées sans répétitions. Comment peut-on réaliser efficacement les opérations de réunion, intersection, différence et différence symétrique ?

Exercice 3-4 : polynômes creux
On représente les polynômes à une variable à coefficients entiers par des listes chaînées de monômes, un monôme aX^e étant représenté par le couple d'entiers (a, e). Les monômes d'un polynôme sont classés par degré croissant et chaque monôme a un coefficient $a \neq 0$. Écrire une fonction d'addition de deux polynômes pour cette représentation.

Exercice 3-5 : fusion multiple
Étudier les problèmes suivants (on cherchera à minimiser le temps d'exécution compté en nombre maximal de comparaisons effectuées) :
1. Fusion de trois listes triées, L_1, L_2, L_3.
2. Fusion de quatre listes triées, L_1, L_2, L_3, L_4.

Exercice 3-6 : tri à bulles
Programmer l'algorithme du tri à bulles dans le cas du tri d'une liste chaînée.

Exercice 3-7 : complexité moyenne du tri à bulles

On suppose que les éléments d'un vecteur $v = [|a_0, \dots, a_{n-1}|]$ sont des entiers compris entre 0 et $K-1$ où K et n sont des constantes, et que les K^n vecteurs de n éléments sont équiprobables. Déterminer le nombre moyen d'échanges effectués lors du tri à bulles de v.

Exercice 3-8 : liste presque triée

Soit $p \in \mathbf{N}$. On dit qu'une liste $L = (a_0, \dots, a_{n-1})$ est p-*presque triée à gauche* si, pour tout entier i, le nombre d'indices $j < i$ tels que $a_j \succ a_i$ est majoré par p. Soit L une liste p-presque triée à gauche.

1. Est-ce que L est aussi p-presque triée à droite, c'est-à-dire le nombre d'indices $j > i$ tels que $a_j \prec a_i$ est-il majoré par p ?
2. Montrer que la complexité asymptotique du tri à bulles d'une liste p-presque triée à gauche est $O(np)$.

Exercice 3-9 : le tri de CAML

La bibliothèque standard de CAML contient la fonction de tri suivante :

```
(* Merging and sorting *)

let merge order =
  merge_rec where rec merge_rec = fun
    [] l2 -> l2
  | l1 [] -> l1
  | (h1::t1 as l1) (h2::t2 as l2) ->
      if order h1 h2 then h1 :: merge_rec t1 l2
                     else h2 :: merge_rec l1 t2
;;

let sort order l =
  let rec initlist = function
      [] -> []
    | [e] -> [[e]]
    | e1::e2::rest ->
           (if order e1 e2 then [e1;e2] else [e2;e1])
        :: initlist rest in
  let rec merge2 = function
      l1::l2::rest -> merge order l1 l2 :: merge2 rest
    | x -> x in
  let rec mergeall = function
      [] -> []
    | [l] -> l
    | llist -> mergeall (merge2 llist) in
  mergeall(initlist l)
;;
```

Expliquer et justifier le fonctionnement de `sort`.

Exercice 3-10 : tri par fusion naturelle
Soit $L = (a_0, \ldots, a_{n-1})$ une liste d'éléments comparables. On appelle *séquence croissante* toute sous-liste $(a_i, \ldots, a_j)$ triée par ordre croissant, et *séquence croissante maximale* toute séquence croissante qui n'est pas contenue dans une autre séquence croissante. Étudier l'algorithme de tri suivant :

Pour trier L *:*
- *parcourir* L *en fusionnant deux par deux les séquences croissantes maximales ;*
- *recommencer jusqu'à ce qu'il ne reste plus qu'une séquence croissante maximale.*

L'étude consiste à écrire un programme implémentant cet algorithme, à en prouver la correction et à en déterminer la complexité asymptotique.

Exercice 3-11 : tri rapide
Programmer l'algorithme du tri rapide pour des listes chaînées.

Exercice 3-12 : dérécursification du tri rapide
Pour éliminer les appels récursifs dans le tri rapide d'un vecteur, on utilise une liste auxiliaire dans laquelle on place les indices limites des sous-vecteurs restant à trier. Écrire une fonction itérative effectuant le tri d'un vecteur suivant cette méthode. Montrer que la longueur de la liste auxiliaire peut être majorée par $1 + \log_2 n$ si l'on choisit de trier en premier le plus petit des sous-vecteurs produits par la phase de segmentation.

Chapitre 4

Évaluation d'une formule

4-1 Structure de pile

Une pile est une liste ne permettant des insertions ou des suppressions qu'à une seule extrémité, appelée *sommet de la pile*. *Empiler* un objet sur une pile P consiste à insérer cet objet au sommet de P, et *dépiler* un objet de P consiste à supprimer de P l'objet placé au sommet. L'objet dépilé est retourné par la fonction de dépilement pour être traité par le programme.

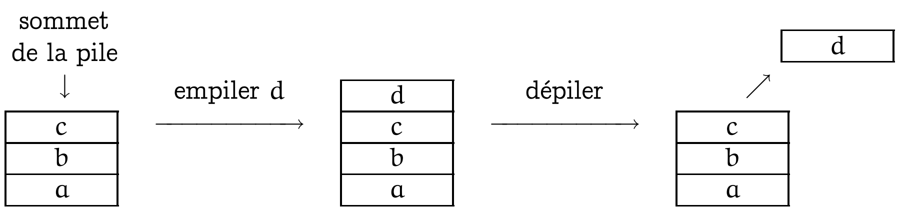


Figure 5 : fonctionnement d'une pile

Une propriété remarquable des piles est qu'un objet ne peut être dépilé qu'après avoir dépilé tous les objets qui sont placés « au dessus » de lui, ce qui fait que les objets quittent la pile dans l'ordre inverse de leur ordre d'arrivée. Pour cette raison une pile est aussi appelée *structure LIFO* (Last In, First Out). On peut comparer le fonctionnement d'une pile à celui d'un journal télédiffusé où le présentateur donne la parole à un reporter. Le reporter rend compte de son enquête et donne à son tour la parole à un témoin. Lorsque le témoin a fini de témoigner, le reporter reprend la parole pour conclure et la rend enfin au présentateur.

En informatique une pile sert essentiellement à stocker des données qui ne peuvent pas être traitées immédiatement car le programme a une tâche plus urgente ou préalable à accomplir auparavant. En particulier les appels et retours de fonctions sont gérés grâce à une pile appelée *pile d'exécution* : si au cours de l'exécution d'une fonction A la machine doit exécuter une fonction B alors elle place au sommet de la pile d'exécution l'adresse à laquelle le code de A est interrompu, exécute le code de B et, lorsque B est terminée, reprend l'exécution à l'adresse figurant au sommet de la pile, c'est-à-dire là où A a été interrompue. Ce mécanisme permet d'imbriquer des fonctions théoriquement sans limitation de profondeur, en pratique dans les limites de la mémoire allouée à la pile d'exécution. La gestion des variables locales est réalisée de la même manière en rangeant ces variables dans la pile d'exécution ou dans une pile dédiée.

Représentation d'une pile par un vecteur

On peut représenter une pile en mémoire par un couple constitué d'un vecteur sur-dimensionné et d'un entier indiquant la première position libre sur la pile. La représentation ci-dessous utilise un *record* CAML c'est-à-dire un couple dont les deux composantes sont désignées par les noms symboliques `objet` et `sommet` plutôt que par leur position dans le couple. `objet` et `sommet` sont appelés *champs* du record. Le champ `sommet` est déclaré *mutable* ce qui signifie qu'on peut le modifier sur place.

```
type 'a v_pile = {objet:'a vect; mutable sommet:int};;

(* création d'une pile vide de taille maximale n      *)
(* x est un objet du type de ceux qui seront empilés  *)
let crée_pile n x = {objet = make_vect n x; sommet = 0};;

(* empile x *)
let empile pile x =
    if pile.sommet >= vect_length(pile.objet)
    then failwith "pile pleine"
    else begin
      pile.objet.(pile.sommet) <- x;
      pile.sommet <- pile.sommet+1
    end
;;

(* dépile le sommet de pile et le retourne comme résultat *)
let dépile pile =
    if pile.sommet <= 0
    then failwith "pile vide"
    else begin
      pile.sommet <- pile.sommet-1;
      pile.objet.(pile.sommet)
    end
;;

(* dit si la pile est vide *)
let est_vide pile = (pile.sommet = 0);;
```

Représentation d'une pile par une liste chaînée

On peut aussi représenter une pile par une référence sur liste chaînée, le sommet de la pile étant la tête de la liste.

```
type 'a ch_pile == 'a list ref;;

(* création d'une pile vide *)
let crée_pile() = ref [];;

(* empile x  *)
let empile pile x = pile := x :: !pile;;

(* dépile le sommet de pile et le retourne comme résultat *)
let dépile pile = match !pile with
    | []       -> failwith "pile vide"
    | a::suite -> pile := suite; a
;;

(* dit si la pile est vide *)
let est_vide pile = !pile = [];;
```

4-2 Représentation linéaire d'une formule

Une formule mathématique peut être représentée linéairement de trois manières :

- forme infixe : les opérateurs binaires sont placés entre leurs arguments, des parenthèses peuvent imposer un ordre d'exécution des opérations ;
- forme préfixe : tous les opérateurs sont placés avant leurs arguments ;
- forme postfixe : tous les opérateurs sont placés après leurs arguments.

Par exemple la formule :

$$\frac{3+2\sqrt{36}}{6}$$

est représentée sous ...

forme infixe : (3 + 2 ∗ `sqrt` (36)) / 6
forme préfixe : / + 3 ∗ 2 `sqrt` 36 6
forme postfixe : 3 2 36 `sqrt` ∗ + 6 /

Formellement une formule est représentée par une suite de *lexèmes*, un lexème étant un nombre, un opérateur unaire, un opérateur binaire ou une parenthèse. La *valeur* d'une formule est définie récursivement à l'aide de règles de réduction remplaçant une partie de la formule par sa valeur. En notant x, y des nombres, f_1 un opérateur unaire, f_2 un opérateur binaire, $f_1[x]$ et $f_2[x,y]$ les résultats des applications de ces opérateurs, on a les règles de réduction suivantes :

réductions d'une formule infixe

$$\begin{array}{llll} (& x &) & \longrightarrow x \\ f_1 & x & & \longrightarrow f_1[x] \\ x & f_2 & y & \longrightarrow f_2[x,y] \end{array}$$

réductions d'une formule préfixe

$$\begin{array}{llll} f_1 & x & & \longrightarrow f_1[x] \\ f_2 & x & y & \longrightarrow f_2[x,y] \end{array}$$

réductions d'une formule postfixe

$$\begin{array}{llll} x & f_1 & & \longrightarrow f_1[x] \\ x & y & f_2 & \longrightarrow f_2[x,y] \end{array}$$

Une suite quelconque de lexèmes, f, constitue une formule infixe, préfixe ou postfixe correcte si et seulement si elle peut être réduite en un nombre à l'aide des règles précédentes. Dans le cas des formules préfixes ou postfixes le nombre obtenu est indépendant de la suite de réductions utilisée (cf. exercice **4-1**) et est la valeur de f. Dans le cas des formules infixes le nombre obtenu dépend de la suite de réductions utilisée, la valeur d'une formule infixe est rendue non ambiguë par utilisation de règles de priorité et d'associativité interdisant certaines réductions.

4-3 Évaluation d'une formule postfixe

Une formule postfixe peut être évaluée à l'aide d'une pile par l'algorithme suivant :

- *initialiser la pile à vide et parcourir la formule « de gauche à droite » ;*
- *à chaque fois que l'on trouve une valeur, empiler cette valeur ;*
- *à chaque fois que l'on trouve un opérateur unaire* f, *dépiler le sommet de la pile, soit* x, *et empiler la valeur* $f[x]$ *;*
- *à chaque fois que l'on trouve un opérateur binaire* g, *dépiler les deux valeurs au sommet de la pile, soit* x, y *avec* x *dépilée en premier, et empiler la valeur* $g[y, x]$ *;*
- *lorsque le parcours de la formule est terminé, la pile ne contient plus qu'une valeur qui est la valeur de la formule.*

Par exemple la formule postfixe :

$3 \quad 2 \quad 36 \quad \texttt{sqrt} \quad * \quad + \quad 6 \quad /$

est évaluée selon les étapes :

pile	formule
[ ]	3 2 36 `sqrt` $*$ $+$ 6 /
[3]	2 36 `sqrt` $*$ $+$ 6 /
[3 2]	36 `sqrt` $*$ $+$ 6 /
[3 2 36]	`sqrt` $*$ $+$ 6 /
[3 2 6]	$*$ $+$ 6 /
[3 12]	$+$ 6 /
[15]	6 /
[2.5]	/
[2.5]	

On prouve la validité de cet algorithme en remarquant qu'à chaque étape la concaténation de la pile et du reste de la formule est une formule déduite de la formule initiale selon les règles de réduction des formules postfixes. Le nombre d'étapes est égal à la longueur de la formule initiale, il est donc fini. Enfin, si la formule initiale est une formule postfixe correcte, alors la formule finale obtenue est une formule postfixe correcte constituée uniquement de valeurs, donc elle contient une seule valeur qui est la valeur de la formule initiale.

```
(* éléments syntaxiques d'une formule portant *)
(* sur des valeurs de type 'a                 *)
type 'a lexème =
| VALEUR     of 'a
| OP_UNAIRE  of 'a -> 'a
| OP_BINAIRE of 'a -> 'a -> 'a
;;

(* évalue une formule postfixe *)
let évalue formule =

  let pile = crée_pile()   (* pile conservant les valeurs en attente *)
  and reste = ref(formule) (* reste de la formule *)
  in

  while !reste <> [] do

    (* traite un lexème : si c'est une valeur, l'empile *)
    (* si c'est un opérateur, effectue l'opération et   *)
    (* empile le résultat.                              *)
    begin match hd(!reste) with
    | VALEUR a     -> empile pile a
    | OP_UNAIRE  f -> let x = dépile pile in empile pile (f x)
    | OP_BINAIRE g -> let x = dépile pile in
                      let y = dépile pile in empile pile (g y x)
    end;
    reste := tl(!reste) (* retire le lexème traité *)

  done;
```

```
    (* normalement, la pile ne contient plus que le résultat *)
    let v = dépile pile in
    if est_vide(pile) then v else failwith "pile non vide"

;;

(* Exemple *)
évalue [ VALEUR     3.0;
         VALEUR     2.0;
         VALEUR     36.0;
         OP_UNAIRE  sqrt;
         OP_BINAIRE mult_float;
         OP_BINAIRE add_float;
         VALEUR     6.0;
         OP_BINAIRE div_float ]
;;
- : float = 2.5
```

En présence d'une formule incorrecte, `évalue` et déclenche l'erreur `"pile vide"` s'il manque un argument à un opérateur, et l'erreur `"pile non vide"` s'il y a plus qu'une valeur dans la pile après évaluation de tous les opérateurs.

Temps d'exécution : en supposant que l'évaluation des opérations intervenant dans `formule` et les opérations sur les piles ont un temps d'exécution constant, le temps d'évaluation d'une formule postfixe de longueur n est $O(n)$.

Remarque : l'usage des fonctions `crée_pile`, `empile` et `dépile` permet de coder l'algorithme d'évaluation d'une formule postfixe indépendamment de l'implémentation effective des piles (à ce détail près que dans l'implémentation des piles sous forme de vecteurs, la fonction `crée_pile` prend un paramètre de taille et un objet du type de ceux qui seront empilés).

4-4 Exercices

Exercice 4-1 : non ambiguïté des formules postfixes
Démontrer que la valeur d'une formule postfixe correcte est bien définie, c'est-à-dire est indépendante de l'ordre des réductions effectuées.

Exercice 4-2 : expressions conditionnelles postfixes
On envisage d'ajouter à la liste des opérateurs un *opérateur conditionnel* du type : `si` *condition* `alors` exp_1 `sinon` exp_2. Quelles sont les modifications à apporter à l'algorithme d'évaluation d'une formule postfixe pour traiter ce type d'expressions ?

Exercice 4-3 : évaluation d'une formule préfixe
Étudier le problème de l'évaluation d'une formule préfixe.

Exercice 4-4 : évaluation d'une formule infixe à l'aide de deux piles
L'évaluation d'une formule infixe non complètement parenthésée impose de définir des règles de priorité de façon à rendre non ambiguë les formules du type :

$$x \quad f \quad y \quad g \quad z$$

où x, y, z sont des nombres et f, g des opérateurs binaires. On adopte ici la convention d'associativité à gauche :

> *évaluer les opérations prioritaires en premier ; en cas de priorités égales évaluer l'opération de gauche en premier.*

On peut alors évaluer une formule infixe à l'aide de deux piles, une pile de valeurs et une pile d'opérateurs de la manière suivante :

> *parcourir la formule en plaçant les valeurs rencontrées dans la pile des valeurs, les opérateurs et parenthèses ouvrantes dans la pile des opérateurs et en effectuant toutes les opérations prioritaires empilées lorsqu'on doit empiler un opérateur binaire ou une parenthèse fermante. Une opération unaire est effectuée au moment où l'on doit empiler l'argument correspondant. A la fin du parcours, effectuer les opérations en instance et retourner le sommet de la pile des valeurs.*

Coder cet algorithme en CAML. On utilisera la déclaration suivante pour représenter les éléments d'une formule infixe :

```
type 'a lexème =
| VALEUR     of 'a
| OP_UNAIRE  of 'a -> 'a
| OP_BINAIRE of int * ('a -> 'a -> 'a) (* priorité, opération *)
| PARENTHÈSE_OUVRANTE
| PARENTHÈSE_FERMANTE
;;
```

Exercice 4-5 : compilation d'une formule infixe
Écrire une fonction `compile` qui prend en argument une formule infixe f supposée bien formée et renvoie une formule postfixe f' constituée des mêmes nombres et des mêmes opérateurs et ayant même valeur que f.

Exercice 4-6 : non ambiguïté des formules infixes
Montrer que si l'on remplace les valeurs par des variables indépendantes et si l'on ne fait aucune hypothèse sur la commutativité ou l'associativité des opérateurs alors une formule infixe bien formée f admet une unique formule postfixe f' équivalente.

Chapitre 5

Logique booléenne

5-1 Propositions

Une proposition est une phrase non ambiguë à laquelle on peut attribuer une valeur de vérité : `vrai` ou `faux`. Cette valeur peut dépendre de paramètres contenus dans la proposition. Par exemple les phrases :

- *Il fait beau aujourd'hui.*
- x *admet une racine carrée entière.*
- π *est un nombre négatif.*

sont des propositions, la deuxième dépendant du paramètre x. Par contre les phrases :

- *Cette phrase est fausse.*
- *Pourquoi pas ?*

n'en sont pas.

En logique booléenne on ne s'intéresse qu'à la valeur de vérité d'une proposition et on ignore le sens de la phrase associée. Deux propositions p et q ayant la même valeur de vérité sont dites *identiques*, ce que l'on note : $p \equiv q$. Par exemple avec :

p : Mercredi vient après Mardi.
q : Mardi vient après Mercredi.
r : Noël est un jour férié.
s : $\pi < 0$.

on a $p \equiv r$ et $q \equiv s$. Si les propositions p et q dépendent de paramètres $x_1, \dots, x_n$, on convient que $p \equiv q$ si et seulement si, pour toute valeur de $(x_1, \dots, x_n)$, $p(x_1, \dots, x_n)$ et $q(x_1, \dots, x_n)$ ont même valeur de vérité. Une proposition est une *tautologie* si elle est identique à `vrai`.

Connecteurs logiques

Étant données deux propositions p et q, on convient que les expressions suivantes sont aussi des propositions :

– non p : noté aussi $\neg p$ ou $\overline{p}$. (non p) est `vrai` lorsque p est `faux`, `faux` lorsque p est `vrai`.

– p et q : noté aussi $p \wedge q$ ou pq. (p et q) est `vrai` lorsque p et q valent `vrai`, (p et q) est `faux` dans les autres cas. L'opération et est appelée *conjonction* ou *produit booléen*.

– p ou q : noté aussi $p \vee q$ ou $p + q$. (p ou q) est `vrai` lorsque l'une au moins des deux propositions vaut `vrai`, (p ou q) est `faux` lorsque les deux valent `faux`. L'opération ou est appelée *disjonction* ou *somme booléenne*.

– p `oubien` q : noté aussi $p \oplus q$. (p `oubien` q) est `vrai` lorsqu'une et une seule des propositions p, q est `vrai`. On peut aussi dire que (p `oubien` q) vaut `vrai` si et seulement si p et q ont des valeurs de vérités différentes.

– $p \Longleftrightarrow q$: $(p \Longleftrightarrow q)$ est `vrai` lorsque p et q ont la même valeur de vérité. Cette notion est différente de l'identité $\equiv$: $p \equiv q$ est un fait tandis que $p \Longleftrightarrow q$ est une proposition qui peut être vraie ou fausse.

– $p \Longrightarrow q$: $(p \Longrightarrow q)$ est une proposition identique à ((non p) ou q), ce qui se lit : *l'implication* $p \Longrightarrow q$ *vaut* `faux` *si et seulement si* p *vaut* `vrai` *et* q *vaut* `faux` *; dans tous les autres cas, en particulier lorsque* p *vaut* `faux`*, l'implication vaut* `vrai`. Remarquons que l'implication booléenne ainsi définie ne traduit pas un lien de cause à effet entre p et q mais seulement une comparaison de leurs valeurs de vérité. Par exemple les implications :

(Mercredi vient après Mardi) $\Longrightarrow (2 + 2 = 4)$
(Il neige en Novembre) $\Longrightarrow$ *(Noël sera en Décembre)*

valent `vrai`. De même, étant donné trois propositions p, q, r quelconques et sans aucun rapport entre elles, la proposition :

$(p \Longrightarrow q)$ ou $(q \Longrightarrow r)$

vaut toujours `vrai`.

Tables de vérité

Soit $f(p_1, \ldots, p_n)$ une proposition dépendant de paramètres $p_1, \ldots, p_n$ qui sont des propositions indéterminées. Les propositions $p_1, \ldots, p_n$ sont appelées : *variables propositionnelles de* f. On dit aussi que f est une *fonction booléenne* de n variables. La *table de vérité* de f est un tableau donnant la valeur de vérité de $f(p_1, \ldots, p_n)$ pour chaque valeur possible du n-uplet $(p_1, \ldots, p_n)$. Chaque variable propositionnelle peut prendre la valeur `vrai` ou `faux`, donc l'ensemble des valeurs possibles pour $(p_1, \ldots, p_n)$ est $\{\mathtt{vrai}, \mathtt{faux}\}^n$. En particulier, la table de vérité de f a 2^n lignes. Ces lignes sont généralement classées par ordre lexicographique du n-uplet valeur de $(p_1, \ldots, p_n)$. Les tables de vérité des connecteurs logiques définis à la section précédente sont :

p	non p
V	F
F	V

p	q	p et q	p ou q	p oubien q	$p \iff q$	$p \implies q$
V	V	V	V	F	V	V
V	F	F	V	V	F	F
F	V	F	V	V	F	V
F	F	F	F	F	V	V

où V et F représentent `vrai` et `faux`.

Une table de vérité peut être utilisée pour définir une fonction booléenne ou pour vérifier une identité remarquable, c'est-à-dire une tautologie. Par exemple la table :

p	q	$\overline{p}$	$\overline{q}$	$\overline{q} \implies \overline{p}$	$p \implies q$
V	V	F	F	V	V
V	F	F	V	F	F
F	V	V	F	V	V
F	F	V	V	V	V

démontre l'identité : $(p \implies q) \equiv (\overline{q} \implies \overline{p})$ qui est à la base du raisonnement par contraposition. On démontre de même les identités suivantes :

$$p \oplus q \equiv \text{non}(p \iff q) \equiv (\overline{p} \iff q) \equiv (\overline{p}q + p\overline{q})$$
$$p \equiv \text{non}(\text{non } p)$$

$$\text{lois de DE MORGAN} \begin{cases} \text{non}(p \text{ et } q) \equiv \overline{p} \text{ ou } \overline{q} \\ \text{non}(p \text{ ou } q) \equiv \overline{p} \text{ et } \overline{q} \end{cases}$$

$$\text{associativité} \begin{cases} p \text{ et } (q \text{ et } r) \equiv (p \text{ et } q) \text{ et } r \\ p \text{ ou } (q \text{ ou } r) \equiv (p \text{ ou } q) \text{ ou } r \end{cases}$$

$$\text{distributivité} \begin{cases} p \text{ et } (q \text{ ou } r) \equiv (p \text{ et } q) \text{ ou } (p \text{ et } r) \\ p \text{ ou } (q \text{ et } r) \equiv (p \text{ ou } q) \text{ et } (p \text{ ou } r) \end{cases}$$

On convient souvent de représenter les valeurs de vérité `vrai` et `faux` par les nombres 1 et 0 ce qui donne les tables de vérité numériques :

p	q	p et q	p ou q	$p \oplus q$	$p \implies q$
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	1

Ce tableau justifie la notation pq pour (p et q). L'opération ou ne correspond à une addition que si l'on admet que $1 + 1 = 1$ (!) et il aurait été plus judicieux de noter $\max(p, q)$ pour p ou q. L'opération $\oplus$ correspond à l'addition modulo 2, en particulier elle est associative et distributive sur et. Enfin l'implication peut être vue comme une opération de comparaison : $(p \implies q) \equiv (p \leqslant q)$.

5-2 Circuits logiques

Un circuit logique est un dispositif physique (électronique, mécanique, optique ou autre) muni *d'entrées* et de *sorties* n'ayant que deux états stables notés 0 et 1. Les états des entrées sont imposés par « l'extérieur », les états des sorties sont imposés par le circuit en fonction des états des entrées. Le circuit est dit *combinatoire* si les états des sorties à un instant donné ne dépendent que des états des entrées à cet instant, et *séquentiel* si les états des sorties dépendent aussi des états antérieurs des entrées. Les circuits logiques élémentaires sont représentés sur la figure 6.

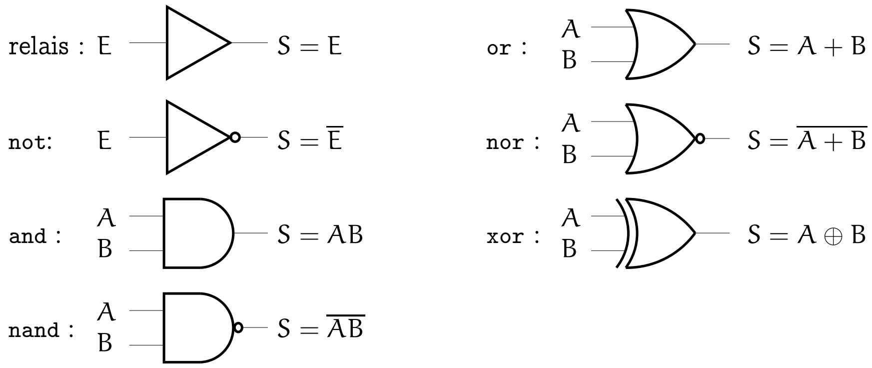


Figure 6 : circuits élémentaires

Le relais est utilisé pour amplifier un signal logique dans un montage comportant beaucoup de circuits, il n'a pas d'utilité au sens logique. Les portes `et` et `ou` sont appelées ainsi car elles permettent de *bloquer* ou *laisser passer* un signal logique (cf. figure 7).

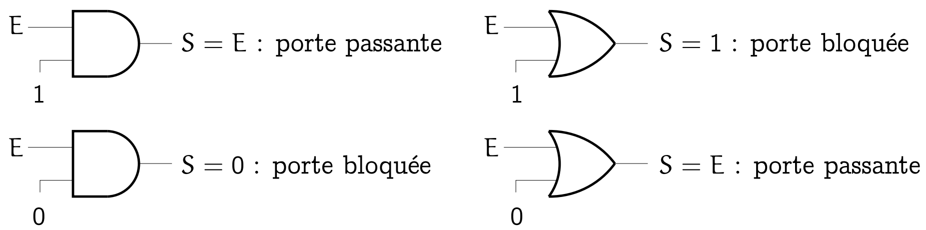
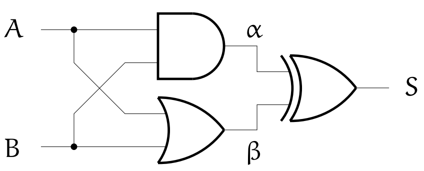


Figure 7 : portes

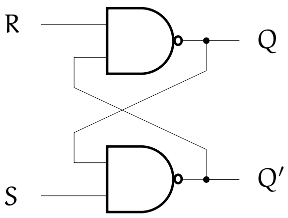
Les portes `nand` et `nor` agissent comme des portes inverseuses : soit la porte est bloquée, soit elle laisse passer la négation de son entrée libre. La porte `oubien` transmet ou inverse un signal présent sur une entrée suivant que l'autre entrée est maintenue à 0 ou à 1.

Un circuit logique non élémentaire est constitué de circuits élémentaires inter-connectés de sorte que toute entrée d'un composant est reliée soit à une entrée du circuit complet soit à une sortie d'un autre composant, et toute sortie d'un composant est reliée à une sortie du circuit complet ou à une ou plusieurs entrées d'autres composants. On démontre qu'un circuit est combinatoire si aucune sortie d'un composant n'est reliée à une des entrées de ce composant, directement ou indirectement à travers d'autres portes (c'est une condition suffisante seulement).



circuit combinatoire

A	B	α	β	S
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0



circuit séquentiel

R	S	Q	Q′
1	1	x	$\overline{x}$
1	0	0	1
0	1	1	0
0	0	1	1

Le circuit séquentiel ci-dessus est appelé *bascule RS*. Il constitue une mémoire simplifiée : on enregistre un 0 en mettant R à 1 et S à 0 puis en ramenant S à 1. De même, en maintenant S à 1, on enregistre un 1 en mettant R à 0 puis en le ramenant à 1. Lorsque $R = S = 1$, le bit mémorisé est disponible sur Q et son complément est disponible sur Q'. Les états de Q et Q' dépendent donc du passé.

Remarque technique

Par assemblage de circuits élémentaires on peut réaliser n'importe quelle fonction booléenne (voir la section suivante). Cependant, lors de la réalisation physique, il faut tenir compte des points suivants :

– Une sortie ne peut alimenter qu'un nombre limité d'entrées. La *sortance* d'un circuit est le nombre maximal d'entrées qui peuvent être connectées sur une sortie de ce circuit ; elle est généralement comprise entre 10 et 100 suivant la technologie employée. Dans un circuit complexe il peut être nécessaire d'intercaler des relais amplificateurs sur les sorties très demandées.

– Chaque porte a un temps de propagation non nul : lorsqu'une entrée change de valeur la sortie n'est garantie stable qu'au bout de ce délai de propagation. Le temps de réponse d'un assemblage de portes élémentaires est donc proportionnel au plus grand nombre de portes intercalées entre une entrée et une sortie du circuit complet. Ce nombre est appelé *profondeur* du circuit et on a généralement intérêt à minimiser cette profondeur pour obtenir un fonctionnement rapide.

5-3 Synthèse des fonctions booléennes

Représentation `et-ou-non`

Théorème : *soit* f *une fonction booléenne des variables* $p_1, \dots, p_n$. *Alors* f *peut être exprimée uniquement à l'aide des connecteurs* `et`, `ou`, `non`, *des variables* p_i *et des propositions constantes* `vrai` *et* `faux`.

Conséquence : *toute fonction booléenne peut être réalisée par assemblage de portes élémentaires* `and`, `or` *et* `not`.

Démonstration : on procède par récurrence sur n. Pour $n = 0$, f est une proposition sans variable, donc $f \equiv \texttt{vrai}$ ou $f \equiv \texttt{faux}$. Si le théorème est vérifié pour toute fonction booléenne à $n-1$ variables, considérons une fonction booléenne f à n variables et les deux fonctions :

$$\begin{aligned} g &: (p_1, \dots, p_{n-1}) \longmapsto f(p_1, \dots, p_{n-1}, \texttt{vrai}), \\ h &: (p_1, \dots, p_{n-1}) \longmapsto f(p_1, \dots, p_{n-1}, \texttt{faux}), \end{aligned}$$

de sorte que :

$$f(p_1, \dots, p_n) \equiv (p_n \texttt{ et } g(p_1, \dots, p_{n-1})) \texttt{ ou } (\overline{p_n} \texttt{ et } h(p_1, \dots, p_{n-1})).$$

L'hypothèse de récurrence appliquée à g et h fournit alors une décomposition de f à l'aide des connecteurs `et`, `ou` et `non`, des variables p_i et des constantes `vrai` et `faux`. ■

Exemple : additionneur 1-bit. Étant donnés deux nombres A et B codés sur un bit (c'est-à-dire compris entre 0 et 1) et une retenue, C codée aussi sur un bit, on veut calculer la somme $A+B+C$ (+ a ici le sens de l'addition des entiers naturels). Cette somme est comprise entre 0 et 3, donc peut être représentée sur deux bits : $A+B+C = 2S_1 + S_0$. On peut écrire la table de vérité de (S_1, S_0) :

A	B	C	S_1	S_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

La première moitié donne la table de vérité de (S_0, S_1) lorsque $A \equiv 0$. Il apparaît que dans ce cas $S'_0 \equiv B \oplus C$ et $S'_1 \equiv BC$. De même, dans le cas $A \equiv 1$ on obtient : $S''_0 \equiv \overline{B \oplus C}$ et $S''_1 \equiv B + C$ (addition logique) ce qui donne :

$$\begin{aligned} S_0 &\equiv A(\overline{B \oplus C}) + \overline{A}(B \oplus C) \,; \\ S_1 &\equiv A(B + C) + \overline{A}(BC). \end{aligned}$$

On a de même : $B \oplus C \equiv B\overline{C} + \overline{B}C$, ce qui donne le circuit additionneur de la figure 8.

Additionneur n-bits

Soient $a = \overline{a_{n-1} \dots a_0}^2$ et $b = \overline{b_{n-1} \dots b_0}^2$ deux nombres entiers naturels décomposés en base 2 et $c = \overline{c_0}^2 \in \{0, 1\}$. Pour calculer la somme $a + b + c$, on peut

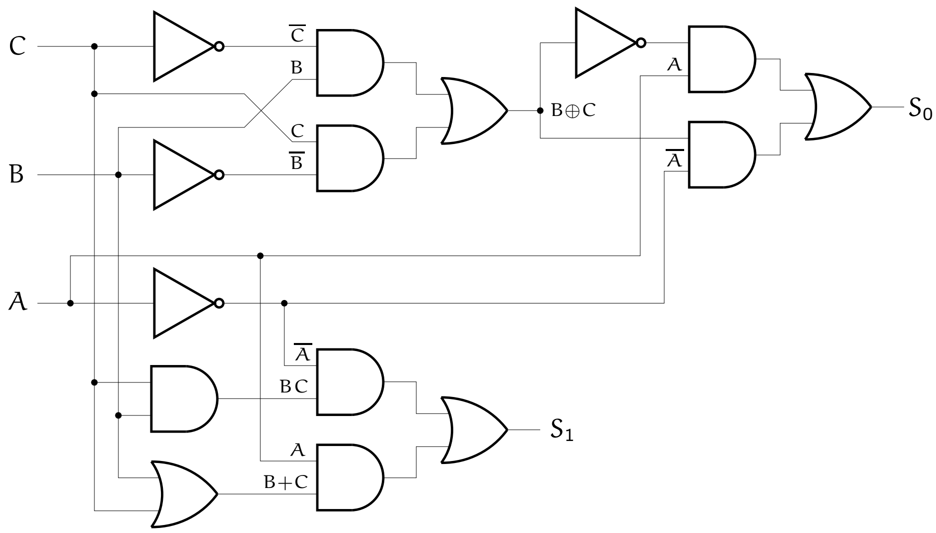


Figure 8 : additionneur 1-bit

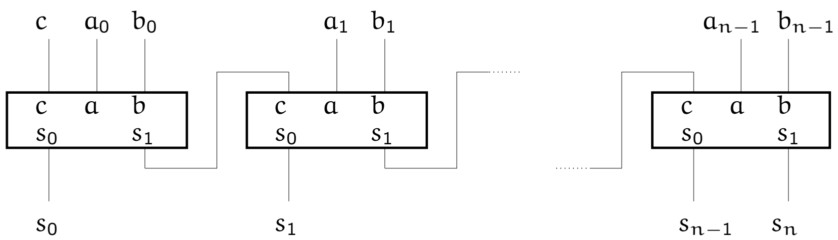


Figure 9 : additionneur n-bits

additionner les bits de même rang de a et b en propageant les retenues, c jouant le rôle de *retenue entrante*. Il suffit donc de disposer n additionneurs 1-bit en cascade (cf. figure 9). L'inconvénient de ce montage est que son temps de réponse est proportionnel à n car chaque additionneur 1-bit doit « attendre » que la retenue de l'additionneur précédent soit disponible pour produire un résultat correct. On peut réaliser l'addition de deux nombres de n bits et d'une retenue entrante plus rapidement en calculant au préalable toutes les retenues. Notons c_k la retenue entrant dans le k-ème additionneur 1-bit, c'est-à-dire la sortie s_1 du $(k-1)$-ème additionneur dans le montage précédent (on convient ici de numéroter les additionneurs à partir de 0). c_k est une fonction booléenne des entrées $a_0, \dots, a_{k-1}$, $b_0, \dots, b_{k-1}$ et c_0 et on démontre par récurrence sur k que l'on a :

$$c_k = f_k(a_0, \dots, b_{k-1}) + c_0 g_k(a_0, \dots, b_{k-1})$$

avec :

$$x_i = a_i + b_i, \qquad y_i = a_i b_i,$$
$$g_k(a_0, \dots, b_{k-1}) = x_0 \dots x_{k-1},$$
$$f_k(a_0, \dots, b_{k-1}) = y_0 x_1 \dots x_{k-1} + y_1 x_2 \dots x_{k-1} + \dots + y_{k-2} x_{k-1} + y_{k-1}.$$

Notons $\Phi_n(a, b, c_0) = (c_0, \dots, c_{n-1})$ et supposons que n est pair, $n = 2p$. En décomposant $a = a' + 2^p a''$ et $b = b' + 2^p b''$, on obtient les relations de récurrence :

$$\begin{aligned} f_n(a, b) &= f_p(a', b') \cdot g_p(a'', b'') + f_p(a'', b''), \\ g_n(a, b) &= g_p(a', b') \cdot g_p(a'', b''), \\ \Phi_n(a, b, c_0) &= \Phi_p(a', b', c_0) \mathbin{@} \Phi_p(a'', b'', f_p(a', b') + c_0 \cdot g_p(a', b')), \end{aligned}$$

où @ désigne la concaténation des listes. Ces relations permettent de construire un circuit calculant récursivement les coefficients $f_n(a, b)$, $g_n(a, b)$ et la liste complète des retenues selon la stratégie « diviser pour régner » (cf. figure 10). Soient P_n la profondeur de ce circuit et K_n le nombre de portes qu'il contient. On a :

$$P_1 = 1, \quad K_1 = 2, \qquad P_n = P_{n/2} + 2, \quad K_n = 2K_{n/2} + 5.$$

D'où, lorsque n est une puissance de 2, $P_n = 1 + 2\log_2(n)$ et $K_n = 7n - 5$. Il est ainsi possible de calculer la somme de deux nombres de n bits en temps logarithmique par rapport à n et avec un nombre de portes linéaire en n (voir l'exercice **5-8** pour une minoration de la profondeur d'un additionneur n-bits quelconque).

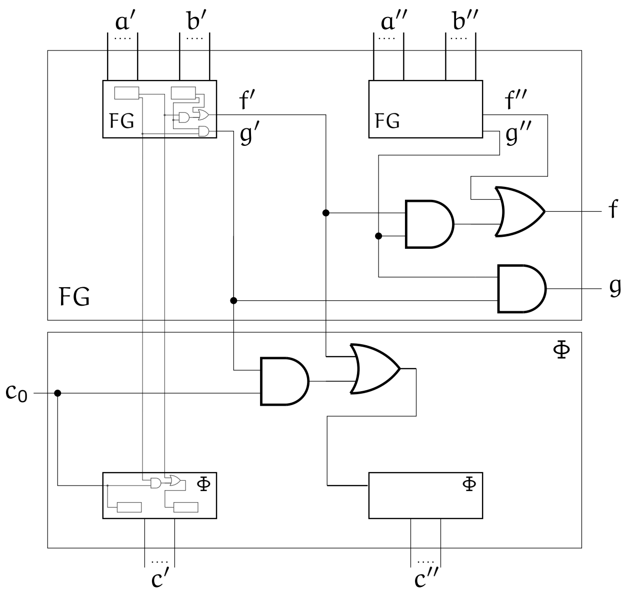


Figure 10 : calcul rapide des retenues

Formes normales

Soient $p_1, \dots, p_n$ des variables propositionnelles. On appelle *littéraux* les propositions $p_1, \dots, p_n$ et $\overline{p_1}, \dots, \overline{p_n}$. Un *monôme* est un produit de littéraux comme par exemple $m = p_1 p_2 \overline{p_3}$. Un *minterme* est un monôme dans lequel chaque variable p_i apparaît une et une seule fois, avec ou sans négation. Il y a donc 2^n mintermes de n variables, et un minterme m donné vaut `vrai` pour une et une seule distribution de vérité des variables $p_1, \ldots, p_n$: celle pour laquelle p_i vaut `vrai` si p_i est facteur de m et `faux` si $\overline{p_i}$ est facteur de m.

Soit f une fonction booléenne des variables $p_1, \ldots, p_n$. Un monôme m est appelé *impliquant de* f si la proposition $m \Longrightarrow f$ est une tautologie, c'est-à-dire si f vaut `vrai` à chaque fois que m vaut `vrai`.

Théorème : *toute fonction booléenne* f *est la somme booléenne des mintermes l'impliquant.*

En effet f et la somme de ses mintermes ont la même table de vérité. Ceci permet d'exprimer f comme une disjonction de mintermes, donc uniquement à l'aide des connecteurs `et`, `ou` et `non`. La décomposition de f ainsi obtenue est appelée *forme normale disjonctive*. Par exemple, pour l'additionneur 1-bit étudié précédemment, on a d'après la table de vérité de (S_1, S_0) :

$$S_0 \equiv \overline{A}.\overline{B}.C + \overline{A}.B.\overline{C} + A.\overline{B}.\overline{C} + A.B.C$$
$$S_1 \equiv \overline{A}.B.C + A.\overline{B}.C + A.B.\overline{C} + A.B.C$$

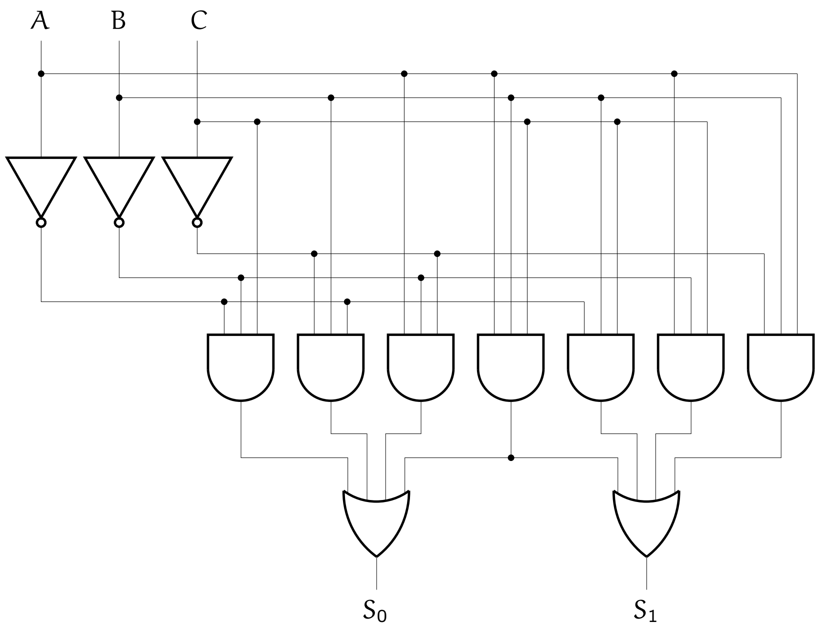


Figure 11 : additionneur 1-bit

Cette expression permet de réaliser l'additionneur à l'aide de portes à 3 ou 4 entrées (cf. figure 11). De manière générale, en utilisant la forme normale disjonctive on peut réaliser n'importe quelle fonction booléenne avec un circuit de profondeur au plus 3, mais ceci impose d'utiliser des portes `et` et `ou` à plus que deux entrées (jusqu'à n entrées pour les portes `et` et jusqu'à 2^n entrées pour les portes `ou`) ce qui peut poser des problèmes de réalisation technique.

En intervertissant les rôles des connecteurs `et` et `ou` on peut aussi représenter une fonction booléenne de n variables comme un produit de sommes de n littéraux où chaque variable apparaît une et une seule fois dans chaque somme. Cette représentation est appelée *forme normale conjonctive*. Pour obtenir la forme normale conjonctive de f il suffit d'appliquer les règles de de Morgan à la négation de la forme normale disjonctive de $\overline{f}$.

5-4 Manipulation des formules logiques

Une formule logique est une représentation d'une fonction booléenne au même titre qu'une expression arithmétique est une représentation d'une fonction algébrique. Par exemple si p, q, r sont trois variables booléennes alors les formules :

$$p \text{ et } (q \text{ et } r), \qquad (p \text{ et } q) \text{ et } r$$

sont différentes par leur forme mais elles représentent la même fonction booléenne, celle qui vaut `vrai` si et seulement si les propositions p, q et r valent `vrai`. On étudie ici les problèmes suivants :

- représentation en mémoire d'une formule logique ;
- évaluation d'une formule lorsque les valeurs de vérité des variables de la formule sont connues ;
- satisfiabilité : existe-t-il une distribution de vérité pour laquelle cette formule vaut `vrai` ?
- tautologie : la formule étudiée vaut-elle `vrai` pour toute distribution de vérité ?
- identité fonctionnelle : deux formules logiques représentent-elles la même fonction booléenne ?

Les trois derniers problèmes sont liés. En effet, la formule f est une tautologie si et seulement si $\overline{f}$ n'est pas satisfiable, et les formules f et g sont identiques si et seulement si $f \Longleftrightarrow g$ est une tautologie. Un problème qui n'est pas abordé est celui de la simplification d'une formule logique, ou de la recherche d'une formule logique « la plus simple possible » représentant une fonction booléenne donnée.

Représentation en mémoire

La notion de formule logique est intrinsèquement récursive : une formule logique est soit une valeur constante `vrai` ou `faux`, soit une variable propositionnelle, soit l'application d'un connecteur logique (`non`, `et`, `ou`, `oubien`, `nand`, `nor`, $\Longrightarrow$, $\Longleftrightarrow$) à une ou deux formules logiques. On définira donc le type `formule` en caml par :

```
type formule =
| Const of bool                  (* valeur constante         *)
| Var   of string                (* variable                 *)
| Mono  of op1 * formule         (* connecteur à un argument *)
| Bin   of op2 * formule * formule (* connecteur binaire     *)

and op1 == bool -> bool
and op2 == bool -> bool -> bool
;;
```

Les connecteurs logiques sont représentés par des fonctions opérant sur le type `bool`. Les connecteurs usuels peuvent être définis par :

```
let non   = fun true        -> false | _   -> true;;
let et    = fun true  true  -> true  | _ _ -> false;;
let ou    = fun false false -> false | _ _ -> true;;
let equiv = fun false false -> true  | true true -> true | _ _ -> false
;;

let nand   x y = non(et x y);;
let nor    x y = non(ou x y);;
let oubien x y = non (equiv x y);;
let impl   x y = ou (non x) y;;
```

La formule logique $f = (p$ `et` $(q \Longleftrightarrow \overline{r}))$ est donc représentée par :

```
Bin(et, (Var "p"), Bin(equiv, (Var "q"), Mono(non, Var "r")))
```

En interne la formule est représentée par un ensemble de cellules reliées par des pointeurs comme pour les listes chaînées (cf. figure 12). Pour des raisons de commodité typographique, on préfère décrire les formules par des *arbres syntaxiques* tels que celui de la figure 13.

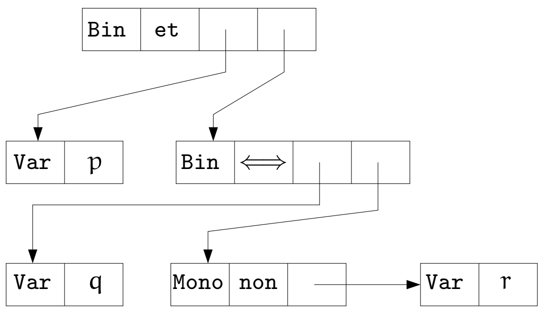


Figure 12 : représentation mémoire d'une formule

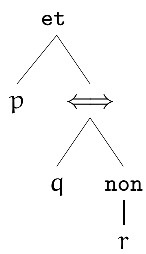


Figure 13 : représentation abstraite

Pour analyser une formule, l'évaluer ou plus généralement procéder à un traitement quelconque, on est amené à *parcourir* la formule, ce qui se fait suivant le schéma récursif :

```
let rec parcours f = match f with
| Const(c)    -> traite_constante(c)
| Var(v)      -> traite_variable(v)
| Mono(op,g)  -> traite_mono(op,g)
| Bin(op,g,h) -> traite_bin(op,g,h)
;;
```

où `traite_constante`, `traite_variable`, `traite_mono` et `traite_bin` sont des actions appropriées au traitement à effectuer. En général les actions `traite_mono` et `traite_bin` procèdent au parcours des sous-formules g et h.

Évaluation d'une formule logique

Pour évaluer une formule logique il faut disposer des valeurs de toutes les variables apparaissant dans la formule. On peut représenter une distribution de vérité par une liste de couples (*nom de la variable, valeur*) transmise en argument à la fonction d'évaluation. Comme les connecteurs sont directement représentés par des fonctions CAML, il suffit d'évaluer récursivement les arguments d'un connecteur puis d'appliquer la fonction associée aux valeurs obtenues :

```
(* Recherche la variable de nom "nom" dans la distribution *)
(* de vérité "distribution" et renvoie sa valeur.         *)
let rec valeur_variable distribution nom = match distribution with
| []           -> failwith "variable inconnue"
| (x,y)::suite -> if x = nom then y else valeur_variable suite nom
;;

(* Évalue la formule f *)
let rec évalue distribution f = match f with
| Const(c)    -> c
| Var(nom)    -> valeur_variable distribution nom
| Mono(op,g)  -> op (évalue distribution g)
| Bin(op,g,h) -> op (évalue distribution g) (évalue distribution h)
;;
```

Complexité : le temps nécessaire à l'évaluation de f dépend de plusieurs paramètres. Chaque variable apparaissant dans f donne lieu à une recherche séquentielle dans la distribution de vérité, nécessitant en moyenne $n/2$ comparaisons. L'évaluation d'une constante ou d'un connecteur prend un temps constant une fois que les arguments du connecteur sont évalués, donc le temps total d'évaluation est de la forme :

$$T = aN + bVn$$

où a, b sont des constantes, N est le nombre de connecteurs et de constantes dans la formule, V est le nombre de variables de f comptées avec répétitions et n est la longueur de la liste `distribution`. Pour une formule de longueur ℓ le temps d'évaluation est donc $O(n\ell)$.

Satisfiabilité, reconnaissance de tautologies

Étant donnée une formule logique f dépendant de variables $p_1, \dots, p_n$, on veut déterminer une distribution de vérité pour $p_1, \dots, p_n$ qui rend f `vrai`. La méthode la plus simple est de « parcourir » la table de vérité de f jusqu'à trouver la valeur `vrai`. Il n'est pas nécessaire de construire explicitement cette table de vérité, il suffit d'engendrer l'une après l'autre toutes les distributions de vérité pour $(p_1, \dots, p_n)$ et d'évaluer à chaque fois f.

```
type résultat =
| TROUVÉ of (string * bool) list       (* dist. de vérité trouvée *)
| NONTROUVÉ                            (* formule non satisfaite  *)
;;

(* cherche une distribution de vérité satisfaisant la formule f   *)
(* vlibres est la liste des variables non encore affectées        *)
(* vliées est la distribution de vérité en construction           *)

let rec satisfait vlibres vliées f = match vlibres with
| [] -> if évalue vliées f then TROUVÉ(vliées) else NONTROUVÉ
| p :: suite -> match satisfait suite ((p,true)  :: vliées) f with
       | NONTROUVÉ -> satisfait suite ((p,false) :: vliées) f
       | res       -> res
;;

(* détection d'une tautologie,
   voir l'exercice 5-5 pour la définition de liste_variables *)
let tautologie f =
  (satisfait (liste_variables f) [] (Mono(non,f))) = NONTROUVÉ
;;
```

Complexité : pour une formule f à n variables de longueur ℓ, la recherche d'une distribution de vérité satisfaisant f peut nécessiter jusqu'à 2^n étapes, chacune de ces étapes consistant à construire une distribution de vérité dans la liste `vliées` puis à évaluer f. Le temps d'exécution d'une étape est $O(n\ell)$ donc le temps nécessaire pour constater qu'une formule à n variables est une tautologie est $O(n2^n\ell)$ car dans ce cas on parcourt effectivement toute la table de vérité. On peut obtenir un temps $O(2^n\ell)$ si la distribution de vérité est stockée dans un vecteur au lieu d'une liste chaînée, mais même avec cette amélioration la détection des tautologies est impraticable pour des valeurs de n dépassant la vingtaine.

Il existe d'autres méthodes pour reconnaître les tautologies, mais elles ont elles aussi un temps d'exécution exponentiel dans le pire des cas : la méthode de DAVIS et PUTNAM consiste à simplifier la formule à contrôler à chaque fois que l'on choisit la valeur de vérité d'une variable. On utilise pour cela les règles de simplification des connecteurs `et` et `ou` ayant un opérande constant ainsi que des règles similaires pour les autres connecteurs (cf. exercice **5-6**). Ceci permet d'obtenir une formule plus courte, ayant au moins une variable de moins que la formule de départ, la variable que l'on vient d'affecter. L'ordre dans lequel on affecte les variables n'est pas indifférent : on a intérêt à choisir parmi les variables encore libres celle qui donnera après simplification la formule la plus simple en nombre de variables libres restant. La recherche de la meilleure variable à affecter complique le code du vérificateur et coûte du temps, mais semble être la méthode la plus efficace connue à ce jour pour la vérification de tautologies.

Une autre méthode de vérification d'une tautologie consiste à mettre la formule f sous forme conjonctive, non nécessairement normale, c'est-à-dire à transformer f en un produit de sommes de littéraux. On peut obtenir une telle forme pour f en traduisant tous les connecteurs à l'aide des connecteurs de base `et`, `ou`, `non`, et en « faisant remonter » tous les `et` et « descendre » tous les `non` par les règles de DE MORGAN et la distributivité de `ou` sur `et`. Après élimination des sommes contenant un littéral et sa négation, il ne reste plus que des sommes non triviales qui fournissent les cas où f vaut `faux`. Donc f est une tautologie si et seulement si la forme conjonctive obtenue pour f est vide après simplifications. Cette méthode a aussi un coût exponentiel dans le pire des cas, car une forme conjonctive à n variables peut contenir jusqu'à 2^{n-1} facteurs différents (cf. exercice **5-3**). Donc cette méthode est non seulement coûteuse en temps, mais aussi en mémoire.

5-5 Exercices

Exercice 5-1 : synthèse des fonctions booléennes
Montrer que toute fonction booléenne peut être exprimée uniquement à l'aide du connecteur `nand`. Montrer qu'il existe des fonctions booléennes non exprimables uniquement à l'aide des connecteurs `et` et `ou`.

Exercice 5-2 : logique ternaire
Les Normands utilisent un système logique à trois valeurs de vérité : `vrai`, `faux` et `peutetre` avec les connecteurs suivants :

- $\mathrm{non}(\mathtt{vrai}) = \mathtt{faux}$, $\mathrm{non}(\mathtt{faux}) = \mathtt{vrai}$, $\mathrm{non}(\mathtt{peutetre}) = \mathtt{peutetre}$;
- p `et` q vaut `vrai` quand $p = q = \mathtt{vrai}$, `faux` quand p ou q vaut `faux` et `peutetre` dans tous les autres cas ;
- $p \ \mathtt{ou}\ q = \overline{\overline{p}\ \mathtt{et}\ \overline{q}}$;
- $p \ \mathtt{oubien}\ q = (p \ \mathtt{ou}\ q) \ \mathtt{et}\ \mathrm{non}(p \ \mathtt{et}\ q)$.

1. Calculer `vrai et (faux oubien peutetre)`.
2. Écrire en CAML les définitions du type `normand` et des connecteurs précédents.
3. Vérifier que les identités usuelles des connecteurs logiques sont encore valables en Normandie (associativité, commutativité, distributivité). On pourra effectuer cette vérification par programme.
4. Est-ce que toute « fonction normande » est exprimable à l'aide des connecteurs `et` `ou` et `non` ?

Exercice 5-3 : taille d'une forme conjonctive
1. Montrer que toute forme conjonctive (conjonction de disjonctions de littéraux, non nécessairement normale) qui équivalente à la formule logique $f = p_1 \oplus p_2 \oplus \ldots \oplus p_n$ comporte au moins 2^{n-1} facteurs.
2. Existe-t-il des fonctions booléennes à n variables nécessitant encore plus de facteurs, même après simplifications ?

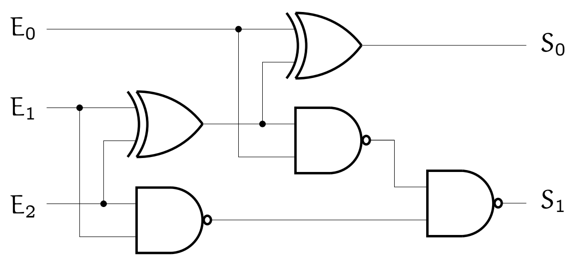
Exercice 5-4 : forme normale exclusive
Montrer que toute formule logique peut être transformée en une *somme exclusive* (à l'aide du connecteur $\oplus$) de produits de variables et des constantes `vrai` et `faux`. Montrer qu'il y a unicité d'une telle forme à l'ordre des termes près si l'on retire les produits nuls (c'est-à-dire contenant un littéral et sa négation) et les produits répétés (car $f \oplus f \equiv 0$). Peut-on utiliser cette mise en forme pour détecter les tautologies ?

Exercice 5-5 : variables d'une formule logique
Soit f une formule logique. Écrire une fonction CAML `liste_variables` calculant la liste sans répétition des variables de f.

Exercice 5-6 : simplification d'une formule logique
Écrire une fonction `simplifie` qui simplifie une formule logique en évaluant les opérandes constants. Noter que si un seul des opérandes d'un connecteur binaire est constant alors la formule peut quand même être simplifiée :

$$p \texttt{ et vrai} \equiv p, \qquad p \texttt{ et faux} \equiv \texttt{faux}, \qquad p \texttt{ oubien vrai} \equiv \overline{p}, \qquad \ldots$$

Exercice 5-7 : additionneur 1-bit
Vérifier que le circuit ci-dessous réalise l'addition de deux bits et d'une retenue. Existe-t-il un additionneur 1-bit comportant moins de 5 portes élémentaires ?



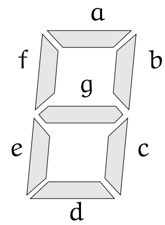
Exercice 5-8 : profondeur minimale d'un additionneur
Montrer que tout additionneur n-bits constitué de portes à une ou deux entrées a une profondeur au moins égale à $\log_2(n)$.

Exercice 5-9 : division par 3
Construire un circuit logique calculant le reste de la division par 3 d'un nombre entier naturel exprimé sur n bits.

Exercice 5-10 : comparateur
Un *comparateur* n-bits est un circuit logique comportant $2n$ entrées $A_0 \ldots A_{n-1}$, $B_0 \ldots B_{n-1}$, et trois sorties : I, E, S telles que si a et b sont les nombres représentés en binaire par $A_0, \ldots, A_{n-1}, B_0, \ldots, B_{n-1}$ alors : $I = 1$ si et seulement si $a < b$, $E = 1$ si et seulement si $a = b$, et $S = 1$ si et seulement si $a > b$.

1. Construire un comparateur 1-bit.
2. Construire un comparateur 2-bits.
3. Construire un comparateur n^2-bits à l'aide de $n + 1$ comparateurs n-bits.

4. Peut-on utiliser la technique précédente pour construire un comparateur n-bits quelconque ?



Exercice 5-11 : affichage 7 segments
Concevoir un circuit logique à 4 entrées, A, B, C, D et 7 sorties a, b, c, d, e, f, g permettant de représenter un entier $n = A + 2B + 4C + 8D$ supposé compris entre 0 et 9 sur un afficheur 7 segments disposés comme ci-contre (un segment est allumé lorsque son entrée vaut 1).

Chapitre 6

Complexité des algorithmes

6-1 Généralités

Afin de comparer plusieurs algorithmes résolvant un même problème, on introduit des mesures de ces algorithmes appelées *complexités*.

- *Complexité temporelle :* c'est le nombre d'opérations « élémentaires » effectuées par une machine qui exécute l'algorithme.
- *Complexité spatiale :* c'est le nombre de « positions mémoire » utilisées par une machine qui exécute l'algorithme.

Ces deux complexités dépendent de la machine utilisée mais aussi des données traitées. Considérons par exemple les algorithmes `prod1` et `prod2` suivants qui calculent le produit des éléments d'un vecteur :

```
let prod1(v) =
  let p = ref(1.0) in
  for i = 0 to (vect_length(v)-1) do p := !p *. v.(i) done;
  !p
;;

let prod2(v) =
  let p = ref(1.0) and i = ref(0) in
  while (!i < vect_length(v)) & (!p <> 0.0) do
    p := !p *. v.(!i);
    i := !i + 1
  done;
  !p
;;
```

Si l'on fait exécuter ces deux algorithmes par une machine CAML, la complexité temporelle de `prod1` est $T_1 = 4n + 2$ où n est la longueur de `v` (on a compté 4 opérations élémentaires dans le corps de la boucle : incrémentation de `i`, accès à `v.(i)`, multiplication et affectation du résultat à `p`) tandis que la complexité temporelle de `prod2` est $T_2 = 6n + 4$ si `v` ne comporte pas d'élément nul, et $T_2 = 6m + 5$ si `v` comporte un zéro, en notant m le rang d'apparition du premier zéro de `v`. Si les deux algorithmes sont exécutés sur la même machine alors `prod2` est plus efficace que `prod1` s'il y a un zéro dans les deux premiers tiers de `v`, moins efficace s'il n'y a pas de zéro ou s'il est dans le dernier tiers. Cette limite approximative $\frac{2}{3}$ ne vaut que pour la machine considérée où chaque opération prend une unité de temps : si l'on utilise une machine où une multiplication compte pour 100 opérations élémentaires, on obtient $T_1 = 103n + 2$ et $T_2 = 105n + 4$ ou $T_2 = 105m + 5$ donc l'avantage revient à `prod2` si `v` comporte un zéro situé à au moins 2% de la fin du vecteur.

Si l'on veut s'abstraire de la dépendance de la complexité par rapport à la machine, on considère que `prod1` et `prod2` effectuent un nombre constant d'opérations élémentaires à chaque itération, auquel cas les complexités sont $T_1 = a_1 n + b_1$ et $T_2 = a_2 n + b_2$ ou $T_2 = a_2 m + b'_2$ pour certaines constantes a_1, b_1, a_2, b_2 et b'_2. Il n'est plus possible dans ce cas de dire si `prod1` est plus ou moins rapide que `prod2`.

Pour s'abstraire de la dépendance de la complexité par rapport aux données précises traitées on considère généralement la complexité dans le pire des cas, c'est-à-dire la complexité maximale pour toutes les valeurs de `v` possibles. Si la taille n de `v` est bornée alors `prod1` et `prod2` s'exécutent en *temps constant*, constant signifiant en fait borné. Si la taille n'est pas bornée à priori, on considère le pire des cas pour un vecteur quelconque de taille n, ce qui donne : $T_1 = a_1 n + b_1$ et $T_2 = a_2 n + b_2$. Comme les constantes a_1, a_2, b_1 et b_2 sont indéterminées (c'est-à-dire dépendent de la machine utilisée), on peut ne retenir que les complexités *asymptotiques* : $T_1 = O(n)$ et $T_2 = O(n)$.

On peut aussi étudier la *complexité en moyenne*, c'est-à-dire la moyenne des complexités d'un algorithme pour toutes les valeurs possibles des données à traiter selon un certain modèle de probabilité. Supposons par exemple que les éléments de `v` sont en fait des entiers aléatoires compris entre 0 et 9, chaque élément étant indépendant des autres. Il y a donc à n fixé 10^n vecteurs `v` possibles qui se rangent en catégories suivant le rang d'apparition du premier zéro :

$m = 1$: 10^{n-1} vecteurs,
$m = 2$: $9 \times 10^{n-2}$ vecteurs,
...
$m = k$: $9^{k-1} \times 10^{n-k}$ vecteurs,
...
$m = n$: 9^{n-1} vecteurs.
pas de zéro : 9^n vecteurs.

La complexité moyenne de `prod2` est alors :

$$\begin{aligned}
T_2 &= \frac{1}{10^n}\sum_{k=1}^{n}(a_2k+b_2)\times 9^{k-1}\times 10^{n-k}+\frac{9^n}{10^n}(a_2n+b_2)\\
&= \frac{1}{10}\sum_{k=1}^{n}(a_2k+b_2)\times 0,9^{k-1}+0.9^n(a_2n+b_2)\\
&\xrightarrow[n\to\infty]{} \frac{1}{10}\sum_{k=1}^{\infty}(a_2k+b_2)\times 0,9^{k-1}=10a_2+b_2.
\end{aligned}$$

La complexité moyenne de `prod2` est donc bornée, alors que celle de `prod1` est asymptotiquement proportionnelle à n. Ce phénomène se produit quelle que soit la distribution de probabilité retenue, pourvu que les éléments de `v` soient indépendants et que la probabilité d'apparition de zéro soit non nulle. Incidemment, on apprend que le rang moyen du premier zéro est environ égal à 10 (ou plus généralement $1/p$ où p est la probabilité d'apparition d'un zéro) lorsque n est grand, et non $n/2$.

La complexité spatiale mesure la quantité de mémoire nécessaire à l'exécution d'un algorithme, en comptant les variables temporaires et le résultat, mais pas les données (de même qu'on ne compte pas le temps d'introduction des données dans la complexité temporelle). L'unité de mesure est le « mot mémoire », mais cette unité dépend de la machine utilisée et on calcule généralement une complexité mémoire asymptotique. Sur une machine séquentielle où il y a un seul processeur exécutant une seule instruction par unité de temps, la complexité mémoire est au plus égale à la complexité temporelle puisqu'il faut au moins une unité de temps pour « remplir » une position mémoire. Ceci n'est pas vrai sur des machines hautement parallèles disposant d'un nombre arbitraire de processeurs. Par exemple le calcul du produit des éléments d'un vecteur de longueur n peut être effectué selon la méthode « diviser pour régner » avec $n/2$ processeurs calculant en même temps les produits de deux termes consécutifs, puis en calculant par la même méthode le produit des $n/2$ sous-produits obtenus (cf. figure 14).

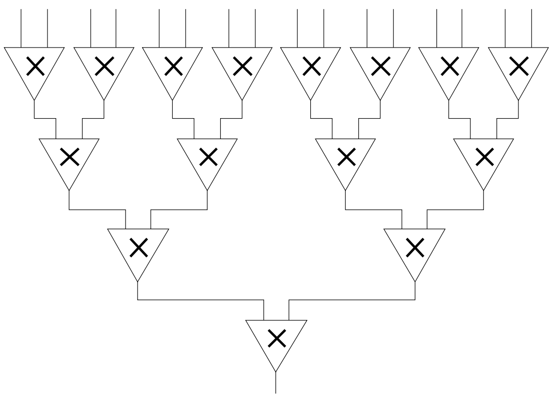


Figure 14 : multiplication parallèle

On obtient ainsi le produit des termes d'un vecteur de longueur $n = 2^p$ en p unités de temps, avec $1+2+4+\ldots+2^{p-1} = n-1$ processeurs utilisant chacun une position mémoire pour leur résultat. Donc la complexité temporelle est $T = O(\ln n)$ tandis que la complexité spatiale est $M = O(n)$.

L'exemple précédent montre que l'on peut changer la complexité temporelle d'un problème en augmentant la complexité spatiale (et la complexité matérielle). Un autre exemple est le calcul d'une suite vérifiant une relation de récurrence double :

```
let rec fib1(n) = if n < 2 then 1 else fib1(n-1) + fib1(n-2);;

let fib2(n) =
  let v = make_vect (n+1) 1 in
  for i = 2 to n do v.(i) <- v.(i-1) + v.(i-2) done;
  v.(n)
;;
```

`fib1` et `fib2` calculent toutes les deux le n-ème terme de la suite de FIBONACCI, mais `fib1` a une complexité temporelle exponentielle (cf. exercice **6-4**) tandis que `fib2` a une complexité temporelle linéaire. Dans cet exemple il n'y a pas augmentation de la complexité spatiale car `fib1` a une complexité spatiale linéaire due au stockage des calculs en suspens. Par ailleurs, on peut écrire une fonction `fib3` calculant F_n en temps linéaire et mémoire constante, et même, par la technique « diviser pour régner », une fonction `fib4` calculant F_n en temps logarithmique et mémoire constante (cf. exercice **1-12**).

6-2 Équation de récurrence $T(n) = aT(n-1) + f(n)$

Le calcul de la complexité d'un algorithme conduit généralement à une relation de récurrence, en particulier si cet algorithme est récursif. On étudie ici le cas où le temps T d'exécution d'un algorithme pour une donnée de taille n suit une relation de la forme :

$$T(n) = aT(n-1) + f(n)$$

où f est une fonction à valeurs entières donnée. Cela signifie que la résolution du problème pour une donnée de taille n se ramène à la résolution de a sous-problèmes pour des données de taille $n-1$. $f(n)$ représente le temps nécessaire pour découper le problème initial en sous-problèmes et pour recombiner les résultats de ces sous-problèmes.

Exemples

Parcours d'une liste de n éléments : $T(n) = T(n-1) + b$ où b est le temps nécessaire pour traiter un élément.

L'algorithme de tri par sélection consiste à parcourir un vecteur de longueur n en repérant la position du plus grand élément, à déplacer cet élément en dernière position par échange, puis à trier le sous-vecteur de taille $n-1$ restant. Le temps de recherche du plus grand élément est proportionnel à n (il faut parcourir tout le vecteur), le temps d'un échange est constant, donc le temps de tri par sélection suit la relation : $T(n) = T(n-1) + bn + c$.

Le problème des tours de Hanoi : n disques de tailles distinctes sont empilés sur un piquet A par tailles croissantes, il faut les empiler sur un piquet B en utilisant un piquet intermédiaire C avec les contraintes de ne déplacer qu'un disque à la fois et de respecter à tout instant la condition de croissance des tailles des disques sur chaque piquet. Ce problème se ramène à trois sous-problèmes :

- déplacer les $n-1$ premiers disques de A vers C en utilisant B comme piquet intermédiaire ;
- déplacer le dernier disque de A vers B ;
- déplacer les $n-1$ disques de C vers B en utilisant A comme piquet intermédiaire.

Le nombre total de déplacements effectués vérifie donc : $T(n) = 2T(n-1) + 1$.

On supposera dans toute la suite que a est un entier supérieur ou égal à 1, et que le temps de séparation-recombinaison, $f(n)$, est strictement positif. L'objectif de l'étude de ces équations de récurrence est d'obtenir, dans la mesure du possible, une expression asymptotique pour $T(n)$.

Manipulation d'inégalités par récurrence

On considère deux fonctions, T, U définies sur $\mathbf{N}$ et vérifiant les équations :

$$T(n) = aT(n-1) + f(n), \qquad U(n) = aU(n-1) + g(n),$$

pour $n \geqslant 1$, où $a \in \mathbf{N}^*$ et f, g sont des fonctions de $\mathbf{N}^*$ dans $\mathbf{N}^*$. On a alors les propriétés suivantes :

1. *Les fonctions* T *et* U *sont strictement croissantes.*
2. *Si* $T(0) \leqslant U(0)$ *et si* $f(n) \leqslant g(n)$ *pour tout* $n \in \mathbf{N}^*$ *alors* $T(n) \leqslant U(n)$ *pour tout* $n \in \mathbf{N}$.
3. *Si* $f(n) = O(g(n))$ *pour* $n \to \infty$ *alors* $T(n) = O(U(n))$ *pour* $n \to \infty$.

En effet, *1* et *2* sont évidentes. Pour *3*, puisque $f(n) = O(g(n))$ et $g(n) > 0$, il existe une constante C telle que $f(n) \leqslant Cg(n)$ pour tout n et, quitte à augmenter C, on peut supposer que $T(1) \leqslant CU(1)$ (car $U(1) = aU(0) + g(1) > 0$) donc $T(n) \leqslant CU(n)$ pour tout $n \geqslant 1$ par récurrence. ■

Cette propriété permet de remplacer une équation de récurrence compliquée par une équation plus simple où l'on ne garde que le terme dominant du temps de séparation-recombinaison. Par exemple le coefficient c peut être ignoré dans l'étude du tri par sélection si l'on ne veut qu'une estimation asymptotique du temps de tri. Plus précisément, si T et U sont solution des équations :

$$T(n) = T(n-1) + bn + c, \qquad U(n) = U(n-1) + bn,$$

alors on a $T(n) = O(U(n))$, mais aussi $U(n) = O(T(n))$, c'est-à-dire que l'on ne risque pas d'obtenir une majoration trop grossière en calculant U à la place de T. On utilise la notation : $T(n) = \Theta(U(n))$ pour indiquer que les deux relations $T(n) = O(U(n))$ et $U(n) = O(T(n))$ ont lieu.

Équation $T(n) = T(n-1) + f(n)$

Cette équation se résout de manière immédiate :

$$T(n) = T(0) + \sum_{k=1}^{n} f(k).$$

Il reste à estimer asymptotiquement la somme $\sum\limits_{k=1}^{n} f(k)$.

Théorème : *(lemme de* CÉSARO*)*
Soient (u_n) *et* (v_n) *deux suites de réels telles que :*

$$v_n > 0, \quad \frac{u_n}{v_n} \xrightarrow[n\to\infty]{} \ell \in [0, +\infty] \text{ et } v_0 + v_1 + \ldots + v_n \xrightarrow[n\to\infty]{} +\infty.$$

Alors le rapport $\dfrac{u_0 + u_1 + \ldots + u_n}{v_0 + v_1 + \ldots + v_n}$ *tend vers* ℓ *quand* $n \to \infty$.

Théorème : *(variante)*
Soient (u_n) *et* (v_n) *deux suites de réels telles que :*

$$v_n > 0, \quad u_n = O(v_n) \text{ pour } n \to \infty \text{ et } v_0 + v_1 + \ldots + v_n \xrightarrow[n\to\infty]{} +\infty.$$

Alors $u_0 + u_1 + \ldots + u_n = O(v_0 + v_1 + \ldots + v_n)$ *pour* $n \to \infty$.

Ces deux énoncés sont admis. Ils permettent d'obtenir un équivalent ou un majorant asymptotique d'une somme en remplaçant le terme général par celui d'une somme connue ou plus facile à calculer. Par exemple pour l'équation de récurrence $T(n) = T(n-1) + f(n)$, si l'on sait que $f(n) \sim \alpha n^p$ pour $n \to \infty$ avec $\alpha > 0$ et $p \geqslant 0$ alors on obtient : $T(n) \sim \alpha \sum_{k=1}^{n} k^p$ pour $n \to \infty$. La somme des puissances p-èmes des premiers entiers est connue pour les petites valeurs de p :

$$\sum_{k=1}^{n} k^0 = n, \qquad \sum_{k=1}^{n} k^1 = \frac{n(n+1)}{2}, \qquad \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6},$$

ce qui suffit pour les récurrences usuelles. Ainsi le temps de parcours d'une liste vérifie : $T(n) \sim bn$ et celui du tri par sélection : $T(n) \sim \frac{1}{2}bn^2$. Dans le cas général, $\sum_{k=1}^{n} k^p$ peut être estimé par comparaison série-intégrale, et l'on a :

$$\sum_{k=1}^{n} k^p \sim \frac{n^{p+1}}{p+1} \quad \text{pour } p \geqslant 0 \text{ et } n \to \infty.$$

Équation $T(n) = aT(n-1) + f(n)$

On se ramène à une équation du type précédent en posant $U(n) = T(n)/a^n$ ce qui donne :

$$U(n) = U(n-1) + \frac{f(n)}{a^n},$$
$$U(n) = U(0) + \sum_{k=1}^{n} \frac{f(k)}{a^k},$$
$$T(n) = a^n \Big(T(0) + \sum_{k=1}^{n} \frac{f(k)}{a^k}\Big).$$

Si $a \geqslant 2$ alors le temps d'exécution de l'algorithme étudié est au moins exponentiel. Par exemple pour le problème des tours de Hanoi :

$$T(n) = 2^n \Big(\sum_{k=1}^{n} 2^{-k}\Big) = 2^n - 1.$$

Dans le cas général on peut obtenir une conclusion plus précise selon la vitesse de croissance de f :

- *si la série* $\sum_{k=1}^{\infty} f(k)/a^k$ *converge (ce qui est en particulier le cas lorsque* $f(n)$ *est à croissance polynomiale) alors* $T(n) \sim \lambda a^n$ *pour une certaine constante* λ *en général non calculable explicitement ;*
- *si* $f(n) \sim \lambda a^n$ *pour* $n \to \infty$ *alors* $T(n) \sim \lambda n a^n$ *;*
- *si* $f(n) \sim \lambda b^n$ *pour* $n \to \infty$ *avec* $b > a$ *alors* $T(n) \sim \dfrac{\lambda b}{b-a} b^n$.

6-3 Récurrence diviser pour régner

Le principe « diviser pour régner » conduit généralement à ramener la résolution d'un problème de taille n à celle d'un ou plusieurs sous-problèmes de taille approximativement $n/2$ puis à combiner les résultats. C'est en particulier le cas pour la recherche par dichotomie dans un vecteur trié, pour le tri par fusion, la multiplication rapide des polynômes (méthodes de KNUTH et transformation de FOURIER rapide), et dans une moindre mesure pour le tri « rapide » : dans ce dernier cas, il n'est pas garanti que la segmentation produise des sous-vecteurs de taille approximativement $n/2$. Les équations de récurrence pour les algorithmes de type diviser pour régner sont généralement de la forme :

$$T(n) = aT(\lfloor n/2 \rfloor) + bT(\lceil n/2 \rceil) + f(n) \quad (n \geqslant 2)$$

où a et b sont des entiers naturels tels que $a + b \geqslant 1$ et f est une fonction de $\mathbf{N}^*$ dans $\mathbf{N}^*$ (le cas de base correspond à $n = 1$ plutôt qu'à $n = 0$).

Cas où n est une puissance de 2

Lorsque n est une puissance de 2, $n = 2^p$, on introduit une fonction auxiliaire U définie par $U(p) = T(2^p)$ et qui vérifie donc :

$$U(p) = (a+b)U(p-1) + f(2^p).$$

On obtient alors :

$$T(2^p) = U(p) = (a+b)^p \Big(U(0) + \sum_{k=1}^{p} \frac{f(2^k)}{(a+b)^k} \Big).$$

Posons $a + b = 2^\alpha$. D'après la section **6-2**, le comportement asymptotique de $T(2^p)$ est lié à celui de la série $\sum_{k=1}^{\infty} f(2^k)/2^{k\alpha}$.

- *Si la série* $\sum_{k=1}^{\infty} f(2^k)/2^{k\alpha}$ *converge, ce qui est réalisé entre autres si* $f(n) = O(n^\beta)$ *avec* $\beta < \alpha$*, alors* $U(p) \sim \lambda 2^{p\alpha}$ *soit* $T(n) \sim \lambda n^\alpha$ *pour un certain* $\lambda > 0$*.*
- *Si* $f(n) \sim \lambda n^\alpha$ *alors* $U(p) \sim \lambda p 2^{p\alpha}$*, soit* $T(n) \sim \lambda n^\alpha \log_2(n)$*.*
- *Si* $f(n) \sim \lambda n^\beta$ *avec* $\beta > \alpha$ *alors* $U(p) \sim \mu 2^{p\beta}$*, soit* $T(n) \sim \mu n^\beta$ *avec* $\mu = \lambda \dfrac{2^\beta}{2^\beta - 2^\alpha}$*.*

Par exemple le tri par fusion vérifie $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn + d$ donc $\alpha = 1$ et $f(n) \sim cn^1$ ce qui implique : $T(n) \sim cn\log_2(n)$ lorsque n est une puissance de 2. Pour la multiplication polynomiale de KNUTH, on a :

$$T(n) = T(\lfloor n/2 \rfloor) + 2T(\lceil n/2 \rceil) + cn + d,$$

donc $\alpha = \log_2(3) \approx 1.58$ et $f(n) \sim cn^1$ d'où $T(n) \sim \lambda n^{\log_2 3}$ pour un certain $\lambda > 0$.

Cas où n n'est pas une puissance de 2

Si f est croissante, on prouve par récurrence que la fonction T est croissante. L'énoncé de récurrence est :

$$H_n \iff \textit{pour } 1 \leqslant p \leqslant q \leqslant n, \textit{ on a } T(p) \leqslant T(q).$$

$n = 1$: il n'y a rien à démontrer.
$n = 2$: il suffit de constater que $T(2) \geqslant T(1)$ ce qui résulte de l'hypothèse $a + b \geqslant 1$.
$n - 1 \Longrightarrow n$: par transitivité de la relation $\leqslant$ et en utilisant H_{n-1}, il suffit de considérer le cas $p = n - 1$ et $q = n$. Comme $n \geqslant 3$ on a $2 \leqslant p < q$ donc on peut appliquer la relation de récurrence à $T(p)$ et $T(q)$:

$$T(p) = aT(\lfloor p/2 \rfloor) + bT(\lceil p/2 \rceil) + f(p)$$
$$T(q) = aT(\lfloor q/2 \rfloor) + bT(\lceil q/2 \rceil) + f(q).$$

Et l'on a $1 \leqslant \lfloor p/2 \rfloor \leqslant \lfloor q/2 \rfloor < n$, $1 \leqslant \lceil p/2 \rceil \leqslant \lceil q/2 \rceil < n$ donc d'après H_{n-1} et la croissance de f, on a bien $T(p) \leqslant T(q)$. ∎

L'hypothèse de croissance de f peut être difficile à prouver si f est compliquée, mais, comme à la section **6-2**, on constate que l'on peut remplacer f par une fonction équivalente sans modifier le comportement asymptotique de $T(n)$, et si l'on obtient un équivalent de la forme λn^β alors la croissance de l'équivalent est évidente.

Maintenant que l'on sait que T est croissante, il suffit d'encadrer n par deux puissances successives de 2 pour obtenir une estimation asymptotique de $\mathsf{T}(n)$. Si $2^p \leqslant n < 2^{p+1}$ et $\mathsf{T}(2^p) \sim \lambda 2^{p\beta}$, alors :

$$\frac{\mathsf{T}(2^p)}{2^{(p+1)\beta}} \leqslant \frac{\mathsf{T}(n)}{n^\beta} \leqslant \frac{\mathsf{T}(2^{p+1})}{2^{p\beta}}$$

et les deux termes extrêmes convergent pour $n \to \infty$ vers $\lambda/2^\beta$ et $\lambda 2^\beta$ donc le rapport $\mathsf{T}(n)/n^\beta$ est borné, c'est-à-dire : $\mathsf{T}(n) = \Theta(n^\beta)$. On obtient une conclusion similaire lorsque $\mathsf{T}(2^p) \sim \lambda p 2^{p\beta}$, d'où le résultat général :

Théorème : *Soit* T *une fonction de* $\mathbf{N}^*$ *dans* $\mathbf{N}^*$ *vérifiant une équation de récurrence de la forme :* $\mathsf{T}(n) = a\mathsf{T}(\lfloor n/2 \rfloor) + b\mathsf{T}(\lceil n/2 \rceil) + f(n)$ *avec* $a, b \in \mathbf{N}$ *et* $a + b \geqslant 1$. *On note* $\alpha = \log_2(a+b)$.

- *Si* $f(n) = O(n^\beta)$ *avec* $\beta < \alpha$ *alors* $\mathsf{T}(n) = \Theta(n^\alpha)$.
- *Si* $f(n) = \Theta(n^\alpha)$ *alors* $\mathsf{T}(n) = \Theta(n^\alpha \ln(n))$.
- *Si* $f(n) = \Theta(n^\beta)$ *avec* $\beta > \alpha$ *alors* $\mathsf{T}(n) = \Theta(n^\beta)$.

Schématiquement, on peut retenir que $\mathsf{T}(n)$ est généralement $\Theta(n^\alpha)$, sauf si le temps de séparation-recombinaison des sous-problèmes est comparable ou supérieur à cette borne, auquel cas $\mathsf{T}(n)$ est $\Theta(n^\alpha \ln(n))$ (cas comparable) ou $\Theta(f(n))$ (cas supérieur).

6-4 Exercices

Exercice 6-1 : calcul de déterminant

1. On veut calculer un déterminant $n \times n$ en développant récursivement suivant la première colonne. Quelle est la complexité temporelle de ce projet ?
2. Pour améliorer le temps de calcul, on décide de mémoriser tous les déterminants mineurs nécessaires de façon à éviter de les calculer plusieurs fois. Quelle est la nouvelle complexité temporelle, et quelle est la complexité spatiale ?

Exercice 6-2 : multiplication matricielle

Pour multiplier deux matrices $n \times n$ il faut effectuer n^3 multiplications scalaires et $n^2(n-1)$ additions en utilisant la formule du produit matriciel. STRASSEN a présenté un algorithme permettant de multiplier deux matrices 2×2 avec 7 multiplications seulement et 18 additions/soustractions :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} p_5 - p_7 - p_2 - p_3 & p_1 + p_2 \\ p_4 - p_3 & p_1 - p_4 + p_5 - p_6 \end{pmatrix}$$

avec :

$$\begin{array}{lll} p_1 = a(f-h), & p_4 = (c+d)e, & p_7 = (d-b)(g+h). \\ p_2 = (a+b)h, & p_5 = (a+d)(e+h), & \\ p_3 = d(e-g), & p_6 = (a-c)(e+f), & \end{array}$$

De plus, cet algorithme peut s'étendre à des matrices $n \times n$ en effectuant 7 multiplications et 18 additions/soustractions de matrices $(n/2) \times (n/2)$.

Si l'on utilise la méthode de STRASSEN récursivement pour calculer les produits des matrices $(n/2) \times (n/2)$, quelle complexité atteint-on pour le produit de deux matrices $n \times n$?

Exercice 6-3 : temps moyen d'une comparaison
Pour classer deux chaînes de caractères par ordre alphabétique on compare leurs caractères de même rang jusqu'à épuiser une chaîne ou à trouver deux caractères différents. Quel est le nombre moyen de comparaisons de caractères effectuées pour comparer deux chaînes de longueur n en supposant que toutes les chaînes sont équiprobables ? Étudier de même le temps moyen de comparaison de deux chaînes de longueur inférieure ou égale à n.

Exercice 6-4 : relation de récurrence
On considère la relation de récurrence :

$$T(n) = T(n-1) + T(n-2) + f(n)$$

où f est une fonction positive donnée à croissance polynomiale.

1. Donner un exemple d'algorithme dont le temps d'exécution vérifie cette relation.
2. Chercher une estimation asymptotique de $T(n)$.

Exercice 6-5 : relation de récurrence
Résoudre asymptotiquement la relation de récurrence :

$$T(n) = 2T(\lceil n/2 \rceil) + n\log_2(n).$$

Exercice 6-6 : relation de récurrence
Résoudre asymptotiquement la relation de récurrence :

$$T(n) = 2T(\lfloor (n-1)/2 \rfloor) + n.$$

Exercice 6-7 : comparaison d'algorithmes
On veut calculer les termes de la suite (u_n) définie par :

$$u_0 = 1, \qquad u_n = \frac{u_{n-1}}{1} + \frac{u_{n-2}}{2} + \ldots + \frac{u_0}{n} \quad \text{pour } n \geqslant 1.$$

Les deux programmes suivants calculent u_n :

```
let rec u_rec(n) = match n with
| 0 -> 1.0
| _ -> let s = ref 0.0 in
       for k = 1 to n do
         s := !s +. u_rec(n-k)/.float_of_int(k)
       done;
       !s
;;
```

```
let u_iter(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)
 ;;
```

Calculer les complexités asymptotiques temporelles et spatiales de ces fonctions.

Exercice 6-8 : comparaison d'algorithmes
Même question avec la relation de récurrence et les programmes suivants :

$$u_0 = 1, \qquad u_n = u_{\lfloor n/2 \rfloor} + u_{\lfloor n/3 \rfloor} \quad \text{pour } n \geqslant 1.$$

```
let rec u_rec(n) =
   if n = 0 then 1 else u_rec(n/2) + u_rec(n/3)
;;

let u_iter(n) =
   let v = make_vect (n+1) 0 in
   v.(0) <- 1;
   for i = 1 to n do v.(i) <- v.(i/2) + v.(i/3) done;
   v.(n)
;;
```

Chapitre 7

Arbres

7-1 Définitions

Arbres généraux

Un *arbre général* est un ensemble non vide et fini d'objets appelés *nœuds* et liés par une « relation de parenté » :

$$\begin{array}{lll} x\,\mathcal{F}\,y & \Longleftrightarrow & x \text{ est } \textit{un} \text{ fils de } y \\ & \Longleftrightarrow & y \text{ est } \textit{le} \text{ père de } x\ ; \end{array}$$

$$\begin{array}{lll} x\,\mathcal{F}^*\,y & \Longleftrightarrow & x \text{ est } \textit{un} \text{ descendant de } y \\ & \Longleftrightarrow & y \text{ est } \textit{un} \text{ ancêtre (ascendant) de } x \\ & \Longleftrightarrow & x = y \text{ ou il existe un } \textit{chemin} : x = x_0\,\mathcal{F}\,x_1\,\mathcal{F}\ldots\mathcal{F}\,x_n = y \\ & & \text{tel que chaque élément est le père de l'élément précédent ;} \end{array}$$

avec les propriétés :

- il existe un unique élément n'ayant pas de père, appelé *racine* de l'arbre ;
- tout élément à part la racine a un et un seul père ;
- tout élément est un descendant de la racine.

Vocabulaire :

- Un nœud n'ayant pas de fils est appelé *feuille* ou *nœud terminal* et un nœud ayant au moins un fils est appelé *nœud interne* ou *nœud intérieur*.
- Les fils d'un même nœud sont appelés *frères*.
- Le *degré* d'un nœud est son nombre de fils, le *degré maximal* de l'arbre est le plus grand des degrés des nœuds.
- La *profondeur* d'un nœud est le nombre d'ascendants stricts de ce nœud, la *hauteur* d'un arbre est la profondeur maximale de ses nœuds.
- Le nombre de nœuds dans un arbre est la *taille* de l'arbre.
- L'ensemble des descendants d'un nœud x forme un *sous-arbre* de racine x.

- Une *forêt* est un ensemble fini d'arbres sans nœuds communs ; l'ensemble des descendants stricts d'un nœud x forme une forêt, vide si le nœud est terminal. Les arbres de cette forêt sont appelés *branches* issues de x.
- Un arbre *ordonné* est un arbre pour lequel l'ensemble des branches issues d'un nœud est totalement ordonné.
- Un arbre est *étiqueté* lorsqu'à chaque nœud est associée une information appelée *étiquette du nœud*. Des nœuds distincts peuvent porter la même étiquette.

Exemples d'arbres : (cf. figure 15)

- L'arbre d'une expression arithmétique ou logique est un arbre dont les nœuds intérieurs sont étiquetés avec les opérateurs composant cette expression et les branches représentent les opérandes associés. C'est un arbre ordonné.

- Un système de fichiers est représenté par un arbre dont les nœuds sont étiquetés avec des noms de fichiers ou de répertoires. Les nœuds intérieurs correspondent aux répertoires non vides et les feuilles aux fichiers de données et aux répertoires vides.

- Une *taxinomie* classe des objets suivant une description hiérarchique. Chaque nœud correspond à une sous-classe incluse dans celle du nœud père.

- Un *arbre de décision* modélise la résolution d'un problème par une suite de tests imbriqués.

- *L'arbre d'appels* d'une fonction récursive représente le calcul de cette fonction, les feuilles correspondant aux cas de base et les nœuds intérieurs aux cas récursifs.

- Un arbre *dynastique* représente les descendants (généralement mâles) d'un individu et correspond à la notion usuelle de parenté. Un arbre *généalogique* représente les ascendants d'un individu. Par dérogation un arbre généalogique a sa racine en bas, mais il ne s'agit réellement d'un arbre que si l'on ne remonte pas trop loin dans le passé sinon un même ascendant risque d'apparaître dans plusieurs branches.

Arbres binaires

Un *arbre binaire* est un ensemble fini, éventuellement vide, de nœuds liés par une relation de parenté orientée :

$$\begin{array}{lcll} x\,\mathcal{F}_d\,y & \Longleftrightarrow & x \text{ est } \textit{le fils droit} \text{ de } y & \Longrightarrow \quad y \text{ est } \textit{le père} \text{ de } x\ ; \\ x'\,\mathcal{F}_g\,y & \Longleftrightarrow & x' \text{ est } \textit{le fils gauche} \text{ de } y & \Longrightarrow \quad y \text{ est } \textit{le père} \text{ de } x'\ ; \end{array}$$

avec les propriétés :

- si l'arbre est non vide alors il existe un unique élément n'ayant pas de père, appelé racine de l'arbre ;
- tout élément à part la racine a un et un seul père ;

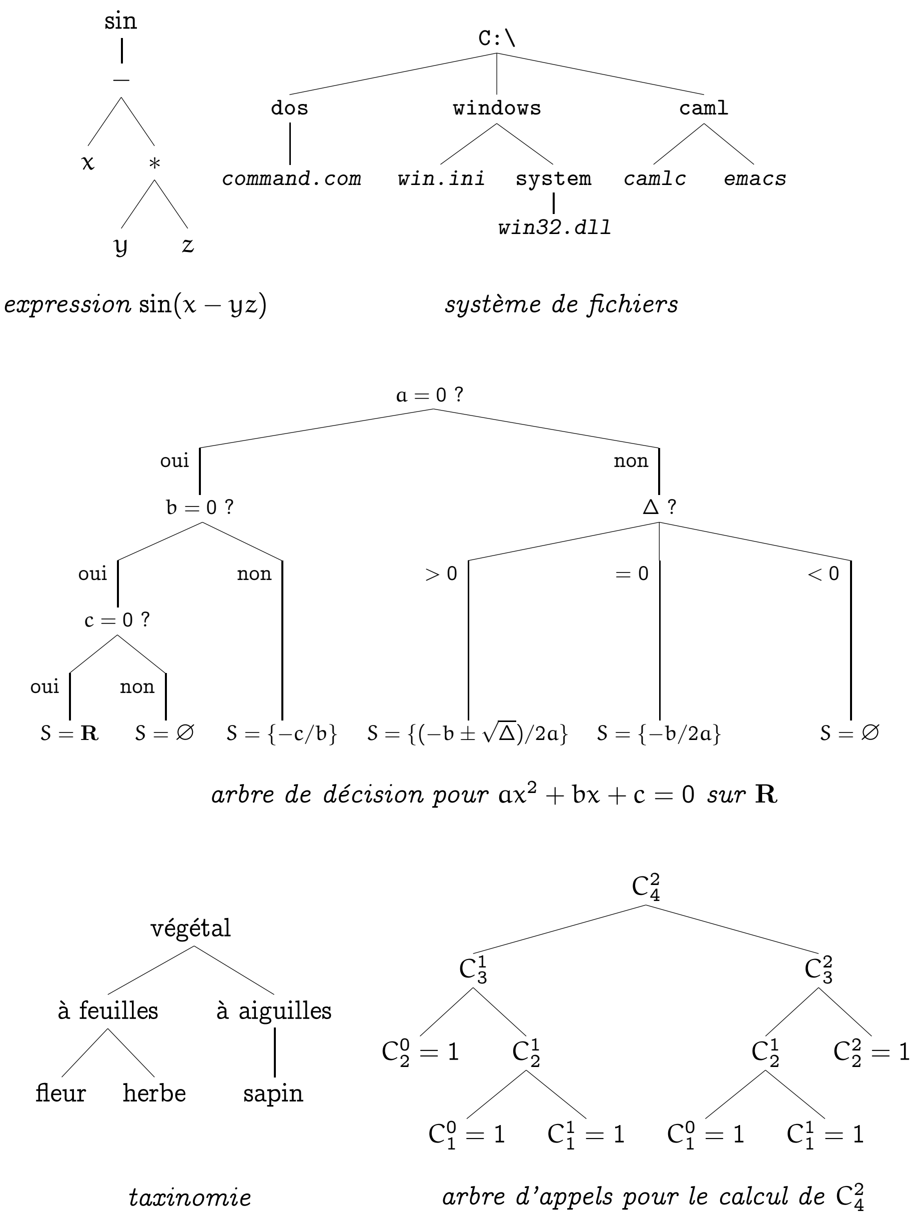
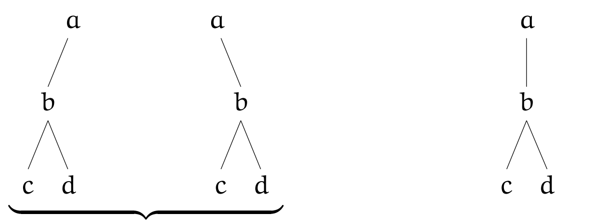


Figure 15 : exemples d'arbres

– tout élément a au plus un fils droit et au plus un fils gauche ;
– tout élément est un descendant de la racine.

Si x est un nœud d'un arbre binaire, alors x possède deux branches qui sont des arbres binaires : la *branche droite* est l'arbre des descendants de son fils droit s'il existe, l'arbre vide sinon ; la *branche gauche* est l'arbre des descendants de son fils gauche s'il existe, l'arbre vide sinon. Les notions d'arbre général ordonné et d'arbre binaire diffèrent par l'existence de l'arbre vide dans la catégorie des arbres binaires, et par le fait qu'un nœud d'un arbre binaire a toujours deux branches, éventuellement vides. En particulier, un arbre binaire n'a pas de feuilles.



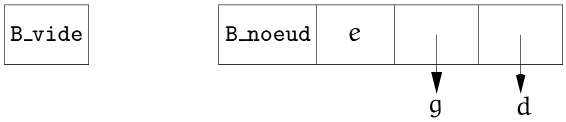
arbres binaires différents *arbre général*

7-2 Représentation en mémoire

Représentation des arbres binaires

```
type 'a b_arbre =
 | B_vide
 | B_noeud of 'a * ('a b_arbre) * ('a b_arbre)
;;

let étiquette(a) = match a with
| B_vide -> failwith "arbre vide"
| B_noeud(e,_,_) -> e

and gauche(a) = match a with
| B_vide -> failwith "arbre vide"
| B_noeud(_,g,_) -> g

and droite(a) = match a with
| B_vide -> failwith "arbre vide"
| B_noeud(_,_,d) -> d
;;
```

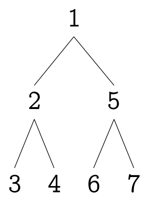
Un arbre binaire non vide est représenté par un triplet (e, g, d) où e est l'étiquette de la racine et g, d sont les branches gauche et droite issues de la racine. `B_vide` et `B_noeud` sont des identificateurs symboliques appelés *constructeurs* permettant de discriminer les deux cas possibles pour un arbre binaire. En mémoire un arbre binaire est représenté par le code du constructeur suivi des représentations des arguments dans le cas `B_noeud`. En CAML, avec la définition précédente, les sous-arbres gauche et droit sont matérialisés par des pointeurs, de même que l'étiquette e si elle est plus complexe qu'un entier, ce qui permet de garantir un temps d'accès constant à chacune des composantes d'un nœud.



La construction explicite d'un arbre peut être pénible. Le code suivant :

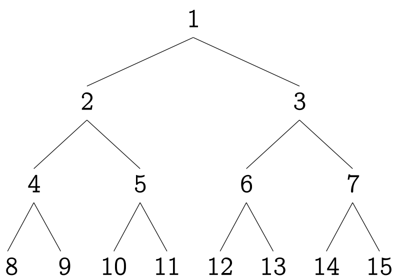
```
let a = B_noeud(1,
          B_noeud(2, B_noeud(3, B_vide, B_vide),
                     B_noeud(4, B_vide, B_vide)),
          B_noeud(5, B_noeud(6, B_vide, B_vide),
                     B_noeud(7, B_vide, B_vide)))
;;
```

définit l'arbre à étiquettes entières :



Représentation par un vecteur

Les nœuds d'un arbre binaire peuvent être numérotés par profondeur croissante et « de gauche à droite » à profondeur égale :



De manière générale, le nœud numéro i a pour fils éventuels les nœuds numéro $2i$ et $2i+1$, et le père du nœud numéro j est le nœud numéro $\lfloor j/2 \rfloor$. Cette numérotation permet de mettre un arbre binaire a en correspondance avec un vecteur `v` : l'étiquette du nœud numéro i est placée dans la cellule $v.(i)$ et, lorsqu'un nœud n'est pas présent dans a, la cellule correspondante est ignorée. Cette représentation est surtout utilisée pour les arbres binaires *parfaits* où tous les niveaux de profondeur sauf le dernier sont complets et où les nœuds du dernier niveau sont situés le plus à gauche possible.

Représentation des arbres généraux

```
type 'a g_arbre = G_noeud of 'a * ('a g_arbre list);;

let étiquette(a) = match a with
| G_noeud(e,_) -> e

and forêt(a) = match a with
| G_noeud(_,f) -> f
;;
```

Un arbre est représenté par l'étiquette de sa racine et la liste des branches issues de la racine. Pour un arbre ordonné, l'ordre des branches dans cette liste est important et un ordre différent définit un arbre différent ; pour un arbre non ordonné l'ordre des branches doit être ignoré et un même arbre admet plusieurs représentations, ce qui doit être pris en compte lors des tests d'égalité entre arbres. Une feuille est un nœud dont la liste des branches est vide. L'expression arithmétique $\sin(x - yz)$ peut être mise en arbre par la déclaration :

```
let expr = G_noeud( "sin",
             [G_noeud( "-",
               [G_noeud( "x", []);
                G_noeud( "*", [G_noeud("y", []); G_noeud("z",[])])
               ])
             ])
;;
```

Le constructeur `G_noeud` est en principe inutile puisqu'il n'y a qu'une forme possible pour un arbre : *(racine, branches)*. On l'a introduit pour insister sur le fait qu'on manipule un arbre et non un couple quelconque. Cette coquetterie ne coûte rien car le compilateur CAML actuel réserve toujours une place pour le code du constructeur, qu'il y en ait un explicite ou non.

En pratique cette définition est trop générale, et on préfère souvent définir un type particulier mieux adapté aux arbres que l'on doit manipuler. Par exemple pour les expressions arithmétiques, on peut définir plusieurs types de feuilles (entiers, réels, variables, ...) et plusieurs types de nœuds internes (opérateurs unaires, binaires, ...) ce qui permet de mieux refléter la structure d'une expression :

```
type 'a expression =
| Const of 'a
| Variable of string
| Op1 of ('a -> 'a) * ('a expression)
| Op2 of ('a -> 'a -> 'a) * ('a expression) * ('a expression)
;;
```

(cf. l'étude des expressions booléennes, section **5-4**). L'avantage d'une description spécialisée est une meilleure lisibilité du code, l'inconvénient est que les algorithmes généraux de manipulation et parcours d'arbres doivent être réécrits à chaque nouvelle définition de type, et que la longueur du code augmente avec la multiplication des cas à tester. On pourrait d'ailleurs spécialiser encore plus la description d'une expression en listant tous les opérateurs possibles ($+$, $-$, $*$, ...). Quoi qu'il en soit, la représentation interne d'un nœud est semblable à celle vue pour les nœuds d'arbres binaires, mais les différentes branches issues d'un nœud sont chaînées entre elles et le temps d'accès à une branche particulière dépend de son rang (cf. figure 16). On pourrait utiliser un vecteur au lieu d'une liste chaînée pour stocker les pointeurs vers les branches, une branche quelconque et le degré d'un nœud seraient alors accessibles en un temps constant.

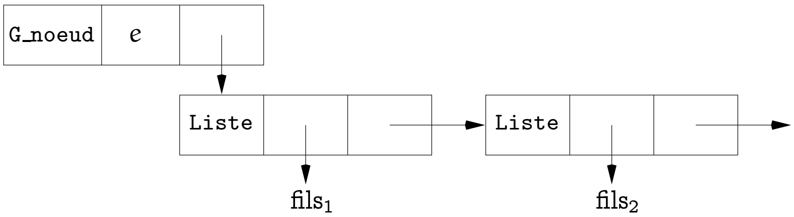


figure 16 : représentation d'un nœud dans un arbre général

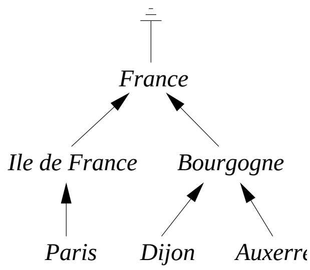
Représentation ascendante

La représentation *(racine,branches)* permet un accès facile aux nœuds de l'arbre en partant de la racine, ce qui correspond au mode d'accès le plus fréquent. Il n'y a aucun moyen avec cette représentation de connaître le père d'un nœud à partir du nœud lui même. Dans certains cas pourtant, on peut avoir besoin de remonter vers la racine, par exemple à partir du nom d'une ville retrouver son département, sa région, son pays, ... Les arbres binaires parfaits rangés dans un vecteur permettent cette remontée. Dans le cas général on adopte une représentation inverse :

```
type 'a a_noeud = A_vide | A_noeud of 'a * ('a a_noeud);;

let étiquette(n) = match n with
| A_vide -> failwith "noeud vide"
| A_noeud(e,_) -> e

and père(n) = match n with
| A_vide -> failwith "noeud vide"
| A_noeud(_,p) -> p
;;
```

L'arbre géographique de la France est déclaré par :

```
let france = A_noeud("France", A_vide);;
let ilefr  = A_noeud("Île de France", france);;
let paris  = A_noeud("Paris", ilefr);;
let bourg  = A_noeud("Bourgogne", france);;
let dijon  = A_noeud("Dijon", bourg);;
let auxr   = A_noeud("Auxerre", bourg);;
```



La racine de l'arbre est reconnue comme telle par son père particulier : `A_vide`. Dans cette représentation il n'est pas garanti que deux nœuds appartiennent bien au même arbre, il faut remonter jusqu'aux racines pour le constater. On représente donc en fait une forêt non ordonnée plutôt qu'un unique arbre. Enfin, dans le cas où l'on a besoin de pouvoir monter et descendre dans un arbre, on peut utiliser une représentation mixte où chaque nœud contient un lien vers son père et un lien vers ses fils.

Représentation « fils gauche - frère droit »

Dans cette représentation chaque nœud contient un lien vers son premier fils et un lien vers son frère suivant appelé « frère droit » (cf. figure 17). Chaque nœud n'a plus que deux liens, éventuellement vides, ce qui transforme un arbre ordonné quelconque en un arbre binaire dont la racine n'a pas de fils droit et une forêt d'arbres ordonnés en un arbre binaire quelconque. Cette représentation n'est pas fondamentalement différente de celle où tous les fils sont rangés dans une liste chaînée, mais elle est plus économique en termes de mémoire.

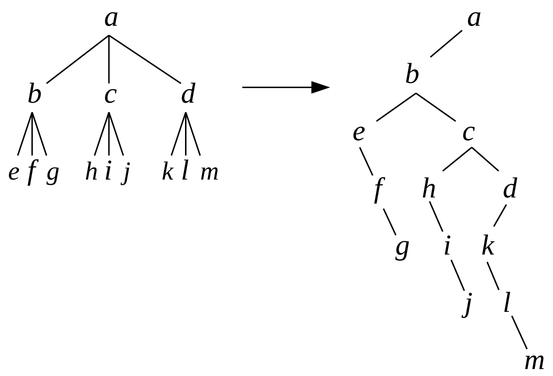


Figure 17 : représentation fils gauche - frère droit

7-3 Parcours d'un arbre

Pour appliquer un même traitement à tous les nœuds de cet arbre, par exemple les compter, les imprimer ou comparer leur étiquette à une valeur donnée, il faut *parcourir* l'arbre. On distingue deux méthodes de parcours.

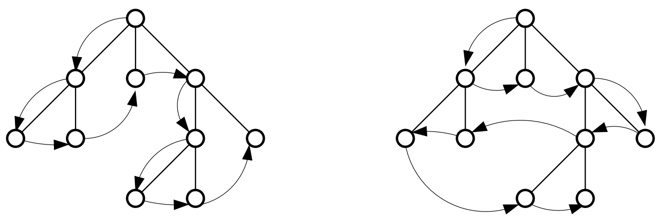


Figure 18 : parcours en profondeur et en largeur

- *Le parcours en profondeur d'abord* : partant de la racine, on descend le plus bas possible en suivant la branche la plus à gauche de chaque nœud, puis on remonte pour explorer les autres branches en commençant par la plus basse parmi celles non encore parcourues.

- *Le parcours en largeur d'abord* : partant de la racine, on explore tous les nœuds d'un niveau avant de passer au niveau suivant.

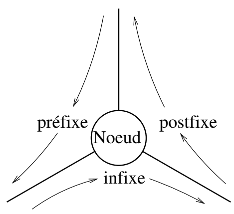
La complexité temporelle d'un parcours d'arbre est au minimum proportionnelle au nombre de nœuds visités, donc à la taille de l'arbre si le parcours n'est pas interrompu prématurément. Il y a proportionnalité lorsque le temps de traitement d'un nœud, non compris le temps de traitement de sa descendance, est constant. La complexité spatiale dépend du type de parcours : pour un parcours en profondeur il faut conserver la liste des branches non encore explorées, ce qui requiert un nombre fixe de mots mémoire par ancêtre strict du nœud visité (un pointeur vers l'ancêtre et le numéro de la première branche non explorée par exemple), donc la complexité spatiale est proportionnelle à la hauteur de l'arbre. Pour un parcours en largeur il faut conserver une liste de pointeurs vers les branches du niveau en cours, et la complexité spatiale est proportionnelle à la *largeur* de l'arbre (le plus grand nombre de nœuds de même niveau).

Ces estimations de complexité peuvent être modifiées par une représentation particulière de l'arbre, par exemple un arbre binaire parfait stocké dans un vecteur permet un parcours en largeur ou en profondeur à mémoire constante.

Parcours en profondeur d'un arbre binaire

```
let rec parcours(a) = match a with
| B_vide -> ()
| B_noeud(e,g,d) -> traitement préfixe;
                    parcours(g);
                    traitement infixe;
                    parcours(d);
                    traitement postfixe
;;
```

traitement préfixe, *traitement infixe* et *traitement postfixe* désignent des actions ou calculs éventuels à effectuer. L'ordre d'application de ces traitements est :

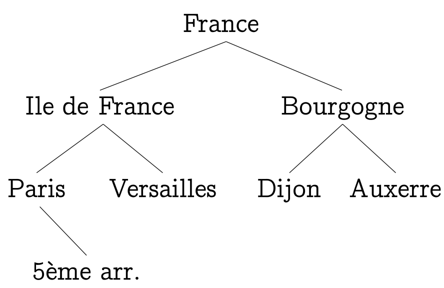


préfixe : nœud, descendants droits, descendants gauches.
infixe : descendants droits, nœud, descendants gauches.
postfixe : descendants droits, descendants gauches, nœud.

L'ordre préfixe, l'ordre infixe et l'ordre postfixe constituent des relations d'ordre total sur les nœuds d'un arbre binaire. L'ordre infixe est aussi appelé *ordre symétrique* (cf. exercice **7-8**). Le numéro préfixe infixe ou postfixe d'un nœud est son rang dans l'ordre correspondant. Par exemple le code suivant imprime les étiquettes d'un arbre avec les numéros postfixe :

```
(* a = arbre, n = numéro postfixe       *)
(* retourne le prochain numéro postfixe *)
let rec imprime a n = match a with
| B_vide        -> n
| B_noeud(e,g,d) -> let n’ = imprime g n  in
                    let n’’= imprime d n’ in
                    print_int(n’’);
                    print_char(‘:‘);
                    print_string(e);
                    print_newline();
                    n’’+1
;;

let a = B_noeud("France",
          B_noeud("Ile de France",
            B_noeud("Paris",
              B_vide,
              B_noeud("5ème arr.", B_vide,B_vide)),
            B_noeud("Versailles",B_vide,B_vide)),
          B_noeud("Bourgogne",
            B_noeud("Dijon",B_vide,B_vide),
            B_noeud("Auxerre",B_vide,B_vide)))
in imprime a 1;;
```

```
1:5ème arr.
2:Paris
3:Versailles
4:Ile de France
5:Dijon
6:Auxerre
7:Bourgogne
8:France
- : int = 9
```



Remarque : l’arbre d’appels de la fonction `parcours` est un arbre binaire isomorphe à l’arbre parcouru.

Parcours en profondeur d’un arbre général

La méthode est la même : effectuer un traitement préfixe sur la racine, parcourir chaque branche en effectuant éventuellement un traitement entre deux branches, puis effectuer un traitement postfixe avant de quitter la racine. Si l’arbre est ordonné, c’est-à-dire si l’ordre de succession des branches est défini, alors on peut définir l’ordre et la numérotation préfixe et postfixe des nœuds. Il n’y a pas d’ordre infixe. Par exemple l’évaluation d’une expression arithmétique mise sous forme d’arbre consiste à parcourir l’arbre en évaluant les branches puis en évaluant les opérations une fois que tous les opérandes sont connus. L’ordre de traitement des nœuds est donc un ordre postfixe (le nœud est traité après sa descendance, mais celle-ci peut l’être dans n’importe quel ordre) et l’algorithme d’évaluation est la transcription en termes d'arbres de l'évaluation d'une formule postfixe, la pile de récursion de caml remplaçant la pile de valeurs.

```
type 'a formule =
| Valeur of 'a
| Op1 of ('a -> 'a) * ('a formule)
| Op2 of ('a -> 'a -> 'a) * ('a formule) * ('a formule)
| Opn of ('a -> 'a -> 'a) * 'a * ('a formule list)
     (* Opn = opérateur binaire associatif : +,*,... *)
;;   (* le 2ème champ est l'élément neutre associé   *)

let rec évalue(f) = match f with
| Valeur(v)        -> v
| Op1(op,g)        -> op (évalue g)
| Op2(op,g,h)      -> op (évalue g) (évalue h)
| Opn(op,neutre,l) -> évalue_liste(op, neutre, l)

and évalue_liste(op,accu,liste) = match liste with
| [] -> accu
| f::suite -> évalue_liste(op, op accu (évalue f), suite)
;;
```

Une formule est représentée par un arbre dont les feuilles sont étiquetées par des valeurs et les nœuds internes sont étiquetés par des opérateurs unaires ou binaires. Le constructeur `Opn` correspond à la répétition d'un opérateur binaire associatif pour lequel on spécifie l'élément neutre qui est retourné en guise de valeur dans le cas d'une liste vide. Le caractère postfixe du traitement effectué n'apparaît pas clairement dans le code de `évalue`, mais résulte de la convention caml comme quoi les arguments d'une fonction sont calculés avant que le code de la fonction soit exécuté. Par exemple la ligne :

```
    | Op2(op,g,h)      -> op (évalue g) (évalue h)
```

est traduite par le compilateur caml en l'équivalent machine de :

```
Si f est de la forme Op2(op,g,h) alors :
  calculer x = évalue(g)
  calculer y = évalue(h)
  retourner op x y.
```

De même, la fonction `évalue_liste` effectue l'évaluation récursive des branches d'une opération multiple et un traitement infixe entre chaque branche, à savoir le calcul et stockage des résultats partiels dans `accu`. Exemple :

```
let f = Opn(add_int, 0,
           [Op2(mult_int, Valeur 1, Valeur 2);
            Op2(div_int,  Valeur 9, Valeur 3);
            Valeur 5;
            Opn(mult_int, 1, [Valeur 2; Valeur 3; Valeur 4])])
in évalue(f);;
- : int = 34
```

Parcours en largeur d'abord

Le parcours en largeur d'abord est surtout utilisé dans les situations de recherche dans un arbre où l'on ne peut pas déterminer quelle est la branche à explorer, et où l'on ne veut pas explorer l'arbre en entier. Par exemple la recherche des feuilles de profondeur minimum dans un arbre de jeu de grande hauteur (quel est le moyen le plus rapide de « mater » aux échecs à partir d'une position donnée ?) conduit à un parcours en largeur. Ce type de parcours est mal adapté à la représentation hiérarchique d'un arbre, mais peut-être effectué en tenant à jour la liste des branches restant à visiter :

```
(* constitue la liste des branches d'une liste de noeuds *)
let rec liste_fils lnoeuds = match lnoeuds with
| [] -> []
| G_noeud(_,f)::suite -> f @ liste_fils(suite)
;;

(* parcours en largeur d'une forêt *)
let rec parcours traitement forêt = match forêt with
| [] -> ()
| _  -> do_list traitement forêt;
        parcours traitement (liste_fils forêt)
;;

(* parcours en largeur d'un arbre *)
parcours traitement [arbre];;
```

On utilise ici la représentation *(étiquette, liste des fils)* d'un arbre général. `do_list` est une fonction prédéfinie en CAML qui parcourt une liste et applique la fonction `traitement` à chaque élément. Par récurrence sur sa hauteur, `parcours` applique `traitement` à chaque nœud de `forêt`, par ordre de niveau et dans un même niveau « de gauche à droite ». Si l'on veut effectuer une recherche ou un calcul sur les nœuds de la forêt alors il suffit de modifier en conséquence l'instruction : `do_list traitement forêt`.

Complexité : on suppose que `traitement` a un temps d'exécution constant et donc que l'itération de `traitement` sur une liste a un temps d'exécution proportionnel à la longueur de cette liste. Le temps d'exécution d'un appel à `liste_fils`, sans compter les appels récursifs qui en découlent, est de la forme $T = ad + b$ où a et b sont des constantes et d le degré du nœud considéré. Le temps d'exécution de `parcours` pour une forêt de k nœuds ayant ensemble k' fils, non compris le parcours de la forêt des fils, est de la forme : $T' = ak' + ck$. Dans le parcours complet d'un arbre, chaque nœud coûte donc $a + c$ unités de temps (c seulement pour la racine) et le temps total de parcours est linéaire par rapport à la taille de l'arbre.

7-4 Dénombrements sur les arbres binaires

Calculs élémentaires

Théorème :

1. *Un arbre binaire à* n *nœuds possède* $n+1$ *branches vides.*
2. *Dans un arbre binaire à* n *nœuds, le nombre de nœuds sans fils est inférieur ou égal à* $(n+1)/2$. *Il y a égalité si et seulement si tous les nœuds ont zéro ou deux fils.*
3. *La hauteur d'un arbre binaire non vide à* n *nœuds est comprise entre* $\lfloor \log_2 n \rfloor$ *et* $n-1$.

Démonstration :

1. Par récurrence sur n.

2. Soient p le nombre de nœuds ayant un seul fils et q le nombre de nœuds sans fils. $p + 2q = n + 1$ (nombre de branches vides) donc $q = (n + 1 - p)/2$.

3. Dans un arbre de taille n la profondeur d'un nœud est son nombre d'ascendants stricts donc est inférieure ou égale à $n-1$. La hauteur d'un arbre est la plus grande profondeur donc est aussi majorée par $n-1$.

   Un arbre binaire non vide de hauteur h a au plus $1 + \ldots + 2^h = 2^{h+1} - 1$ nœuds. Soit a un arbre binaire de taille n et $h = \lfloor \log_2 n \rfloor$: tout arbre binaire non vide de hauteur inférieure ou égale à $h-1$ a au plus $2^h - 1 < n$ nœuds donc la hauteur de a est au moins égale à h. ■

Application aux calculs de complexité

Considérons un arbre de décision binaire (chaque test n'a que deux issues) ayant N sorties. Une sortie est une branche vide (il n'y a plus de test à faire) donc cet arbre possède $N-1$ nœuds et a une hauteur au moins égale à $\lfloor \log_2(N-1) \rfloor$. Ceci signifie que le nombre de tests à effectuer dans le pire des cas est au moins égal à $1 + \lfloor \log_2(N-1) \rfloor = \lceil \log_2 N \rceil$. On obtient ainsi une minoration de la *complexité intrinsèque* d'un problème.

Par exemple le tri comparaisons de n éléments distincts a une complexité dans le pire des cas au moins égale à $\lceil \log_2(n!) \rceil$, complexité comptée en nombre de comparaisons, puisqu'une issue du tri est une permutation des éléments les remettant dans l'ordre, permutation bien définie si les éléments sont distincts, et toutes les permutations sont possibles (voir l'exercice **7-19** pour une construction formelle de l'arbre de décision associé à un algorithme de tri par comparaisons). Cela prouve que la complexité intrinsèque dans le pire des cas du tri par comparaisons est $\Theta(n \ln n)$.

Le minorant obtenu par cette méthode n'est pas toujours aussi pertinent. La recherche du plus grand élément dans une liste de taille n produit un arbre de décision à n issues (une issue est la la position de ce plus grand élément), elle nécessite donc au moins $\lceil \log_2 n \rceil$ comparaisons. Mais en réalité il est impossible de connaître le plus grand élément sans les avoir tous examinés, donc la complexité intrinsèque de ce problème est $\Theta(n)$.

Dans le même ordre d'idées, un calcul dépendant de n variables et réalisé par des opérations binaires ne peut être conduit en moins de $\log_2(n)$ unités de temps dans le pire des cas *même si l'on effectue autant d'opérations binaires en parallèle que l'on veut.* L'arbre exprimant le calcul à effectuer, dont les nœuds sont les opérateurs binaires et les feuilles sont les variables, possède n feuilles donc a une hauteur au moins égale à $\lceil \log_2 n \rceil$. Ainsi il est impossible de déterminer par comparaisons le plus grand élément d'une liste de taille n sur une machine parallèle en moins de $\lceil \log_2 n \rceil$ unités de temps, et le minorant est cette fois pertinent (cf. multiplication en parallèle, section **6-1**).

Arbres équilibrés

Un arbre binaire est dit *équilibré en hauteur* s'il est vide ou si pour tout nœud x, les branches gauche et droite issues de x ont même hauteur à une unité près.

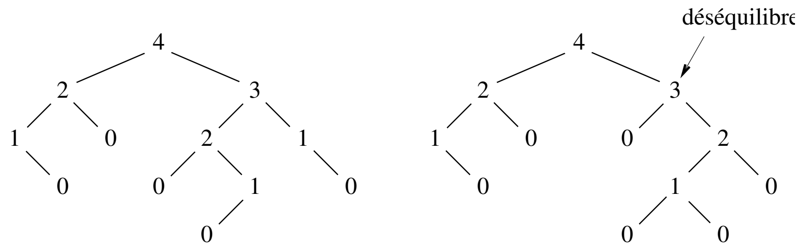


Figure 19 : arbres équilibré et déséquilibré

Remarquons que la propriété d'équilibre est une propriété globale valable pour l'arbre entier et pour tous ses sous-arbres.

Théorème : *Soit* h *la hauteur d'un arbre équilibré en hauteur à* n *nœuds et* $\phi = \frac{1+\sqrt{5}}{2}$*. Alors* $\log_2(n+1) \leqslant h+1 < \log_\phi(n+2)$.

Démonstration :

Pour $h \in \mathbf{N}$ soient $m(h)$ et $M(h)$ le plus petit et le plus grand nombre de nœuds d'un arbre équilibré de hauteur h. On a pour $h \geqslant 2$:

$$\begin{aligned} m(h) &= 1 + m(h-1) + \min(m(h-1), m(h-2)), \\ M(h) &= 1 + M(h-1) + \max(M(h-1), M(h-2)). \end{aligned}$$

Par conséquent les fonctions m et M sont croissantes sur $\mathbf{N}^*$ et $m(0) = M(0) = 1$, $m(1) = 2$, $M(1) = 3$, donc elles sont croissantes sur $\mathbf{N}$. On en déduit :

$$\begin{aligned} m(h) + 1 &= (m(h-1) + 1) + (m(h-2) + 1), \\ M(h) + 1 &= 2(M(h-1) + 1), \end{aligned}$$

et donc :

$$m(h)+1 = a\phi^h + b\overline{\phi}^h,$$
$$M(h)+1 = c2^h,$$

où a, b, c sont des constantes indépendantes de h, $\phi = \frac{1+\sqrt{5}}{2}$ et $\overline{\phi} = \frac{1-\sqrt{5}}{2}$ (voir l'étude de la suite de FIBONACCI dans le cours de mathématiques). Compte tenu des conditions initiales, on obtient :

$$m(h)+1 = (\phi^{h+3} - \overline{\phi}^{h+3})/\sqrt{5},$$
$$M(h)+1 = 2^{h+1}.$$

La deuxième égalité donne $h+1 = \log_2(M(h)+1) \geqslant \log_2(n+1)$. La première implique $\phi^{h+3} = (m(h)+1)\sqrt{5} + \overline{\phi}^{h+3} < (m(h)+2)\sqrt{5}$ car $\overline{\phi}^{h+3} < 1 < \sqrt{5}$ donc :

$$h+1 < \log_\phi(m(h)+2) + \log_\phi(\sqrt{5}) - 2 < \log_\phi(m(h)+2) \leqslant \log_\phi(n+2).$$

∎

En conséquence, la hauteur d'un arbre équilibré en hauteur est $\Theta(\ln n)$ ce qui signifie que les algorithmes de recherche en profondeur dans un tel type d'arbre ont une complexité logarithmique.

Dénombrement des arbres binaires à n nœuds

Soit C_n le nombre d'arbres binaires non étiquetés à n nœuds. On a $C_0 = 1$, $C_1 = 1$, $C_2 = 2$, $C_3 = 5$ et $C_4 = 14$. De manière générale, un arbre binaire à n nœuds s'obtient en choisissant deux arbres binaires à k et $n-k-1$ nœuds indépendamment, et en les reliant à une racine, l'arbre à k nœuds étant à gauche. On a donc la relation de récurrence :

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-k-1} \quad (n \geqslant 1).$$

Notons $C(z) = \sum_{n=0}^{\infty} C_n z^n$ la série génératrice associée. En multipliant la relation précédente par z^{n-1} et en sommant de $n=1$ à ∞, on obtient :

$$\frac{C(z)-C_0}{z} = C(z)^2 \iff zC(z)^2 - C(z) + 1 = 0 \iff C(z) = \frac{1 \pm \sqrt{1-4z}}{2z}$$

et C étant de classe $\mathcal{C}^\infty$ au voisinage de 0, le $\pm$ est en fait un $-$. Il est ainsi prouvé que, si la série C a un rayon de convergence non nul alors, $C(z) = (1-\sqrt{1-4z})/2z$. Considérons alors la fonction définie par :

$$f(z) = \frac{1-\sqrt{1-4z}}{2z}.$$

f est développable en série entière,

$$f(z) = \sum_{n=0}^{\infty} \frac{-1}{2}\binom{1/2}{n+1}(-4)^{n+1} z^n = \sum_{n=0}^{\infty} \frac{C_{2n}^n}{n+1} z^n = \sum_{n=0}^{\infty} a_n z^n \qquad \text{pour } |z| < 1,$$

et les coefficients (a_n) vérifient la relation :

$$a_n = \sum_{k=0}^{n-1} a_k a_{n-k-1}$$

pour $n \geqslant 1$ car les deux membres sont les coefficients des développements en série entière de $(f(z) - a_0)/z$ et $f(z)^2$, et ces fonctions sont égales par construction de f. Enfin, comme $a_0 = 1 = C_0$, on peut conclure :

$$\forall\, n \in \mathbf{N},\ C_n = a_n = \frac{C_{2n}^n}{n+1}.$$

Le nombre C_n est appelé n-*ème nombre de* CATALAN, et l'on a d'après la formule de WALLIS :

$$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}} \quad (n \to \infty).$$

Profondeur moyenne d'un nœud

La méthode « série génératrice » précédente peut aussi être utilisée pour déterminer la profondeur moyenne d'un nœud dans un arbre binaire de taille n. Notons P_n la somme des profondeurs de tous les nœuds de tous les arbres binaires à n éléments et $P(z) = \sum_{n=0}^{\infty} P_n z^n$. En décomposant un arbre à n nœuds en son sous-arbre gauche à k nœuds et son sous-arbre droit à $n - k - 1$ nœuds, et en observant que la profondeur d'un nœud augmente de 1 quand on passe du sous-arbre à l'arbre complet, on obtient :

$$\begin{aligned} P_n &= \sum_{k=0}^{n-1} \left(C_{n-k-1}(P_k + kC_k) + C_k(P_{n-k-1} + (n-k-1)C_{n-k-1})\right) \\ &= (n-1)C_n + \sum_{k=0}^{n-1} C_{n-k-1}P_k + \sum_{k=0}^{n-1} C_k P_{n-k-1}, \end{aligned}$$

ce qui se traduit en termes de séries génératrices par :

$$\frac{P(z) - P_0}{z} = C'(z) - \frac{C(z) - C_0}{z} + 2P(z)C(z)$$

avec $P_0 = 0$, d'où :

$$P(z) = \frac{zC'(z) - C(z) + 1}{1 - 2zC(z)} = \frac{1}{1-4z} - \frac{1}{z\sqrt{1-4z}} + \frac{1}{\sqrt{1-4z}} + \frac{1}{z}.$$

Comme précédemment, la fonction définie par l'expression ci-dessus est développable en série entière et ses coefficients vérifient par construction la même équation de récurrence que les P_n, donc on peut les identifier, ce qui donne :

$$P_n = 4^n - (n+2)C_{n+1} + (n+1)C_n \quad (n \geqslant 1).$$

La profondeur moyenne d'un nœud dans un arbre binaire quelconque à n *éléments est donc :*

$$p_n = \frac{P_n}{nC_n} \sim \sqrt{n\pi} \quad (n \to \infty).$$

Arbres binaires construits aléatoirement

Le résultat précédent a été établi en supposant que tous les arbres binaires à n nœuds sont équiprobables. Un autre modèle de probabilité consiste à considérer qu'un arbre binaire donné a été construit par adjonction un à un de nœuds dans un ordre aléatoire. Plus précisément, on suppose qu'un arbre a de taille n s'obtient à partir d'un arbre a' de taille $n-1$ par transformation d'une branche vide en une branche à un nœud, et que les n branches vides de a' sont choisies avec la même probabilité $1/n$. Cette hypothèse est vérifiée dans le cas des arbres binaires de recherche si l'on procède à l'insertion « aux feuilles » des entiers $1, 2, \dots, n$ dans un ordre aléatoire (cf. section **8-2**). Dans ce modèle de probabilité, il y a $n!$ arbres binaires à n nœuds (non tous distincts puisque $C_n < n!$) et la profondeur moyenne d'un nœud est la somme des profondeurs des tous les nœuds de ces $n!$ arbres, divisée par $n\,n!$.

Pour un arbre binaire a on note $L_i(a)$ la somme des profondeurs des n nœuds *(longueur du chemin interne de* a*)* et $L_e(a)$ la somme des profondeurs des $n+1$ branches vides *(longueur du chemin externe de* a*)*, en convenant que la profondeur d'une branche vide est égale à la profondeur du nœud dont elle est issue augmentée d'une unité. On note aussi $L_i(n)$ et $L_e(n)$ les sommes des quantités $L_i(a)$ et $L_e(a)$ pour les $n!$ arbres binaires à n nœuds.

Si a' est un arbre binaire à $n-1$ nœuds et si a se déduit de a' par transformation d'une branche de profondeur p en un nœud alors on a :

$$L_i(a) = L_i(a') + p, \qquad L_e(a) = L_e(a') + p + 2.$$

On en déduit :

$$L_i(n) = nL_i(n-1) + L_e(n-1), \qquad L_e(n) = (n+1)L_e(n-1) + 2\,n!$$

avec les conditions initiales : $L_i(1) = 0$, $L_e(1) = 2$. On obtient alors successivement :

$$\frac{L_i(n)}{n!} = \frac{L_i(n-1)}{(n-1)!} + \frac{L_e(n-1)}{n!}, \qquad \frac{L_e(n)}{(n+1)!} = \frac{L_e(n-1)}{n!} + \frac{2}{n+1},$$

$$\frac{L_e(n)}{(n+1)!} = 2\Big(\frac{1}{2} + \dots + \frac{1}{n+1}\Big), \qquad \frac{L_i(n)}{n!} = 2\sum_{k=2}^{n}\Big(\frac{1}{2} + \dots + \frac{1}{k}\Big).$$

Comme $1/2 + \dots + 1/k \sim \ln(k)$ lorsque $k \to \infty$, on peut appliquer le lemme de CÉSARO :

$$\frac{L_i(n)}{n!} \sim 2\sum_{k=2}^{n} \ln(k) = 2\ln(n!) \sim 2n\ln(n).$$

Ainsi, *la profondeur moyenne d'un nœud dans un arbre binaire de* n *éléments* construit aléatoirement *est :*

$$\frac{L_i(n)}{n\,n!} \sim 2\ln(n).$$

7-5 Exercices

Exercice 7-1 : relation entre les nombres de nœuds
Si un arbre a contient n_0 nœuds de degré 0 (feuilles), n_1 nœuds de degré 1, ..., n_p nœuds de degré p, quelle relation y a-t-il entre n_0, n_1, ..., n_p ?

Exercice 7-2 : nombre moyen de fils
On suppose ici que tous les arbres binaires à n nœuds sont équiprobables. Dans un arbre binaire de taille n, combien un nœud a-t-il de fils en moyenne ? Combien a-t-il de fils droits en moyenne ?

Exercice 7-3 : ordre des feuilles
Montrer que l'ordre préfixe et l'ordre postfixe induisent la même relation d'ordre sur les feuilles d'un arbre.

Exercice 7-4 : reconstitution d'un arbre à partir des ordres préfixe et postfixe
Soient x, y deux nœuds d'un arbre a. Montrer que x est un ascendant de y si et seulement si x précède y en ordre préfixe et succède à y en ordre postfixe.

Exercice 7-5 : représentation fils gauche, frère droit
Écrire une fonction CAML transformant une forêt d'arbres généraux en un arbre binaire suivant la convention « fils gauche - frère droit » et une fonction effectuant la transformation inverse.

Exercice 7-6 : représentation fils gauche, frère droit
Soient a un arbre général ordonné et a' l'arbre binaire associé à a dans la représentation fils gauche, frère droit. Quel rapport y a-t-il entre les ordres préfixe et postfixe des nœuds de a et les ordres préfixe, postfixe et infixe des nœuds de a' ?

Exercice 7-7 : arbres binaires parfaits
Soit a un arbre binaire parfait à n nœuds rangé dans un vecteur `v`. Peut-on calculer en mémoire constante les indices des successeurs d'un nœud d'indice i pour les ordres préfixe, infixe, postfixe ?

Exercice 7-8 : ordres de parcours inverses
Le parcours préfixe inverse d'un arbre consiste à traiter un nœud d'abord puis ensuite à parcourir les branches issues de ce nœud en commençant par la dernière branche. On définit de même l'ordre postfixe inverse et, pour un arbre binaire, l'ordre infixe inverse. Comparer ces ordres aux ordres de parcours directs.

Exercice 7-9 : fonctions de parcours
Étudier les fonctions de parcours suivantes (quel est l'ordre de parcours, quelle est la complexité asymptotique du parcours réalisé en supposant que `traitement` a un temps d'exécution constant) :

```
let rec parcours traitement forêt = match forêt with
| [] -> ()
| G_noeud(e,f)::suite ->
    traitement(e); parcours traitement (f @ suite)
;;
```

```
let rec courspar traitement forêt = match forêt with
| [] -> ()
| G_noeud(e,f)::suite ->
    traitement(e); courspar traitement (suite @ f)
;;
```

Exercice 7-10 : transformation d'un arbre binaire en liste
Écrire une fonction `liste_préfixe` qui constitue la liste des étiquettes des nœuds d'un arbre binaire suivant l'ordre préfixe. Analyser sa complexité temporelle (il existe une solution de complexité linéaire par rapport à la taille de l'arbre).

Exercice 7-11 : impression textuelle d'un arbre
Écrire une fonction `imprime_arbre_binaire` qui prend en argument une fonction de conversion `'a -> string` et un arbre binaire à étiquettes de type `'a` et l'imprime sous forme semi graphique. Par exemple, on devra avoir :

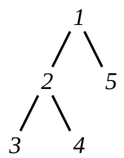
```
imprime_arbre_binaire
       string_of_int
       (B_noeud(1, B_noeud(2, ...)));;

            ______1______
           /             \
        __2__           __9__
       /     \         /     \
      3       6      10       13
     / \     / \     / \     / \
    4   5   7   8  11  12  14  15
```

Exercice 7-12 : reconnaissance de sous-arbres
On donne deux arbres binaires a et b.
1. Écrire une fonction `est_sous_arbre` qui dit si a est un sous-arbre de b (un sous-arbre est l'ensemble des descendants d'un nœud).
2. Écrire une fonction `est_inclus` qui dit si a est inclus dans b (c'est-à-dire si a s'obtient en supprimant certaines branches d'un certain sous-arbre de b).

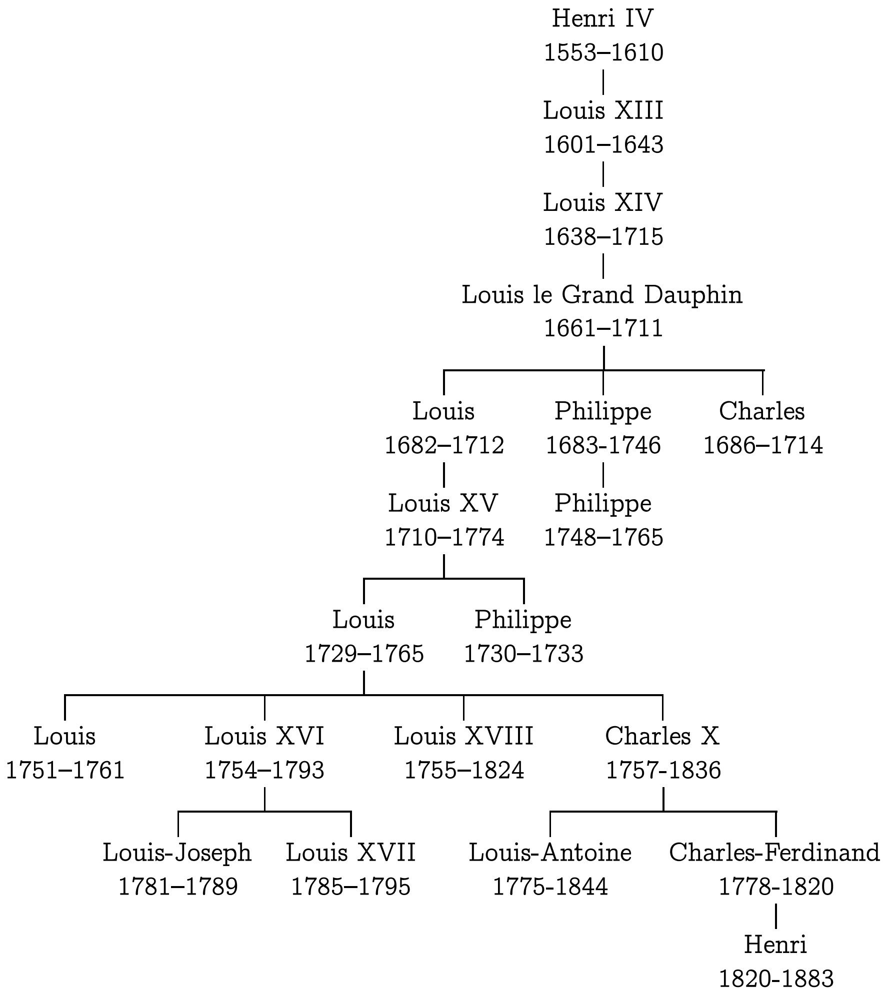
Exercice 7-13 : arbre dynastique
La figure page suivante présente un extrait de l'arbre dynastique de la maison de BOURBON. Écrire une fonction `liste_rois` qui, à partir d'un tel arbre, donne la liste des personnes ayant régné.

Exercice 7-14 : représentation d'un arbre par une liste parenthésée
On représente un arbre a par une expression parenthésée :

$$a \equiv (\textit{racine}\ (\textit{fils-}1)\ (\textit{fils-}2)\ \ldots\ (\textit{fils-}n))$$



où *racine* est l'étiquette de la racine et *(fils-*i*)* est la représentation de la i-ème branche. Par exemple l'arbre ci-contre est représenté par : `(1 (2 (3) (4)) (5))`.



arbre dynastique des BOURBONS

source : *Initiation à l'histoire de la France*, PIERRE GOUBERT

On demande :

1. Une fonction CAML qui calcule l'expression parenthésée d'un arbre.
2. Une fonction CAML qui calcule l'arbre associé à une expression parenthésée.

Les expressions parenthésées seront représentées par des listes selon la déclaration :

```
type 'a terme = Pargauche | Pardroite | Etiquette of 'a;;
type 'a expression_parenthésée == 'a terme list;;
```

Exercice 7-15 : impression d'une formule avec le minimum de parenthèses
On représente les expressions arithmétiques par le type suivant :

```
type formule =
| Valeur of string            (* valeur *)
| Op1 of string * formule     (* opérateur unaire, argument *)
| Op2 of string * int * formule * formule
;;                         (* op. binaire, priorité, arguments *)
```

où les champs de type `string` contiennent la représentation en chaîne de caractères de l'élément considéré et le champ de type `int` code la priorité d'un opérateur binaire. Écrire une fonction d'impression pour ce type de formules en imprimant le minimum de parenthèses nécessaires pour respecter la priorité des opérateurs binaires. L'argument d'un opérateur unaire sera systématiquement mis entre parenthèses.

Exercice 7-16 : arbre binaire équilibré en poids
Soit $\frac{1}{2} \leqslant \alpha < 1$. On note $|A|$ la taille d'un arbre binaire A et on dit qu'un arbre A est α-équilibré en poids si pour tout sous-arbre B de branches G, D, on a : $|G| \leqslant \alpha|B|$ et $|D| \leqslant \alpha|B|$. Montrer qu'un arbre α-équilibré de taille n a une hauteur $\Theta(\ln n)$.

Exercice 7-17 : arbre binaire équilibré en hauteur
Un arbre binaire est dit équilibré en hauteur à k près si pour tout nœud x les branches gauche et droite de x ont même hauteur à $\pm k$ près ($k \geqslant 1$). Montrer que la hauteur d'un tel arbre à n nœuds est $\Theta(\ln n)$.

Exercice 7-18 : dénombrement des arbres à n nœuds
Soient G_n le nombre d'arbres généraux ordonnés non étiquetés de taille n et B_n le nombre d'arbres binaires non étiquetés de taille n.

1. Quelle relation y a-t-il entre ces quantités ?
2. A un arbre général A on associe l'expression parenthésée, $u = [u_0, \dots, u_{2n-1}]$ où u_i est une parenthèse ouvrante ou fermante (cf. exercice **7-14**). Montrer qu'une suite de $2n$ parenthèses représente effectivement un arbre de taille n si et seulement si elle contient autant de parenthèses ouvrantes que de fermantes et si toute sous-suite $[u_0, \dots, u_i]$ avec $i < 2n-1$ contient strictement plus de parenthèses ouvrantes que de fermantes. Une telle suite sera dite « admissible ».

3. Soit $u = [u_0, \dots, u_{2n-1}]$ une suite de parenthèses contenant autant d'ouvrantes que de fermantes avec $u_0 = ($. On définit la suite :

$$T(u) = [u_0, u_2, u_3, \dots, u_{2n-1}, u_1]$$

(on fait tourner toutes les parenthèses sauf la première). Montrer qu'une et une seule des suites $u, T(u), T^2(u), \dots, T^{2n-2}(u)$ est admissible (faire un dessin représentant la différence entre le nombre d'ouvrantes et de fermantes dans $[u_0, \dots, u_{i-1}]$ en fonction de i et interpréter géométriquement l'opération T).

4. Retrouver ainsi l'expression de B_n en fonction de n.

Exercice 7-19 : Complexité en moyenne du tri par comparaisons

1. Soit a un arbre binaire non vide à n nœuds. Montrer que la longueur du chemin externe de a est minimale (à n fixé) si et seulement si tous les niveaux de a sauf peut-être le dernier sont complets. En déduire que, dans le cas général, on a $L_e(a) \geqslant (n+1)\lfloor \log_2 n \rfloor$.

2. On considère un algorithme de tri par comparaisons $\mathcal{A}$ opérant sur des listes de n entiers avec $n \geqslant 2$. Si σ est une permutation de $[\![1, n]\!]$, on note T_σ le nombre de comparaisons effectuées par $\mathcal{A}$ pour trier la liste $(\sigma(1), \dots, \sigma(n))$. Le nombre moyen de comparaisons effectuées par $\mathcal{A}$ est :

$$T_n = \frac{1}{n!} \sum_\sigma T_\sigma$$

où la somme porte sur toutes les permutations de $[\![1, n]\!]$. En modélisant $\mathcal{A}$ par un arbre de décision, montrer que $T_n \geqslant \lfloor \log_2(n! - 1) \rfloor$.

Chapitre 8

Arbres binaires de recherche

8-1 Recherche dans un arbre binaire

Un arbre binaire non vide étiqueté est appelé *arbre binaire de recherche* si les étiquettes appartiennent à un ensemble totalement ordonné et s'il vérifie l'une des deux conditions équivalentes suivantes :

1. *la liste des étiquettes en ordre infixe est croissante au sens large ;*
2. *pour tout nœud* x *d'étiquette* e*, les étiquettes des éléments de la branche gauche de* x *sont inférieures ou égales à* e *et les étiquettes des éléments de la branche droite de* x *sont supérieures ou égales à* e*.*

Remarques :

– La condition *2* implique que l'étiquette d'un nœud est comprise entre celles des ses fils gauche et droit quand ils existent, mais il n'y a pas équivalence.
– Par définition même, tout sous-arbre non vide d'un arbre binaire de recherche est encore un arbre binaire de recherche.
– Pour un ensemble donné d'étiquettes il existe plusieurs arbres binaires de recherche contenant ces étiquettes, en particulier deux arbres *linéaires* assimilables à des listes ordonnées et des arbres « mieux équilibrés » (cf. figure 21).

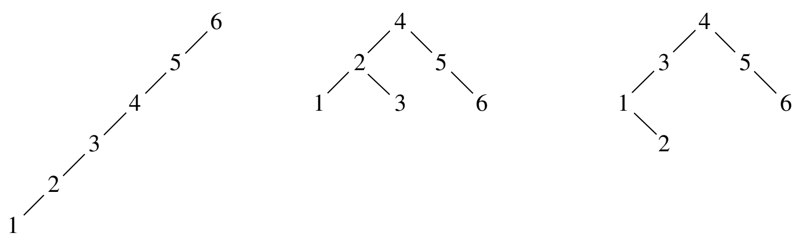


Figure 21 : arbres binaires de recherche ayant mêmes étiquettes

On utilise les arbres binaires de recherche pour représenter des ensembles finis ou des tables d'association. Leur intérêt vient de la propriété *2* : pour déterminer si une étiquette x donnée figure dans un arbre binaire de recherche, il suffit de la comparer à l'étiquette r de la racine de l'arbre, puis si $x \neq r$ de chercher récursivement x dans le sous-arbre gauche ou droit selon le sens de la comparaison entre x et r. On élimine ainsi une branche à chaque comparaison, ce qui élimine environ la moitié des nœuds si l'arbre est équilibré.

```
(* détermine si l'étiquette x appartient à l'arbre donné *)
(* compare est la relation d'ordre entre étiquettes      *)
let rec recherche compare x arbre = match arbre with
| B_vide          -> false
| B_noeud(e,g,d) -> match compare x e with
   | EQUIV     -> true (* EQUIV = égal *)
   | PLUSGRAND -> recherche compare x d
   | PLUSPETIT -> recherche compare x g
;;
```

La correction et la terminaison de `recherche` s'établissent par récurrence sur la hauteur de l'arbre considéré. Le temps d'exécution est proportionnel au nombre de sous-arbres examinés, soit $O(h)$ pour un arbre de hauteur h. Lorsque l'arbre est équilibré on a $h = O(\ln n)$ et le temps de recherche est logarithmique par rapport à la taille de l'arbre, mais si l'arbre est filiforme alors $h = O(n)$ et la complexité de `recherche` est linéaire par rapport à la taille de l'arbre. La complexité spatiale est en principe constante car la fonction `recherche` est *récursive terminale*, c'est-à-dire qu'il n'y a rien à faire après l'appel récursif.

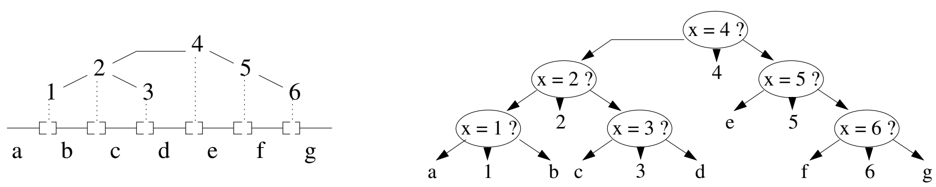


Figure 22 : arbre de recherche et arbre de décision

Un arbre binaire de recherche peut être vu comme un arbre de décision, les nœuds correspondant aux comparaisons à effectuer avec trois branches par nœud : continuer la recherche à droite, à gauche, ou conclure à la présence de x. Les feuilles de cet arbre de décision sont les nœuds de l'arbre de recherche et les branches vides représentant les intervalles délimités par les étiquettes de l'arbre (cf. figure 22). Remarquer par ailleurs la similitude entre la recherche dans un arbre binaire de recherche et la recherche par dichotomie dans un vecteur trié. Dans ce dernier cas l'arbre correspondant est équilibré en hauteur puisque les branches issues d'un pivot ont même taille à une unité près.

A la différence d'un vecteur, un arbre binaire ne permet pas d'accéder en temps constant à un élément par son rang. Toutefois on peut trouver le plus grand ou le plus petit élément en $O(h)$ étapes : il suffit de « descendre » le plus à gauche possible (recherche du minimum) ou le plus à droite possible (recherche du maximum) :

```
let rec minimum(a) = match a with
| B_vide              -> failwith "arbre vide"
| B_noeud(e,B_vide,_) -> e
| B_noeud(_,g,_)      -> minimum(g)
;;
```

8-2 Insertion

Étant donnés un arbre binaire de recherche a et une étiquette x, on veut construire un nouvel arbre binaire de recherche b contenant les étiquettes des nœuds de a et x. Noter que b n'est pas entièrement spécifié dans ce problème car plusieurs arbres peuvent correspondre au même ensemble d'étiquettes. La méthode la plus simple consiste à déterminer dans quelle branche issue de la racine de a on peut insérer x, et à réaliser cette insertion récursivement.

```
let rec insère compare x a = match a with
| B_vide         -> B_noeud(x,B_vide,B_vide)
| B_noeud(e,g,d) -> match compare x e with
   | PLUSGRAND -> B_noeud(e,g,insère compare x d)
   | PLUSPETIT -> B_noeud(e,insère compare x g,d)
   | EQUIV     -> a
;;
```

La correction de `insère` est immédiate par récurrence sur la hauteur de a. En pratique, `insère` suit le même chemin dans l'arbre que la fonction `recherche` jusqu'à trouver une branche vide, et remplace cette branche par un nœud d'étiquette x. Le nouvel arbre est alors reconstruit de proche en proche depuis cette branche vers la racine, sans modifier les branches non explorées. Le retour vers la racine est effectué par le jeu normal de dépilement des appels imbriqués sans qu'il soit nécessaire de le coder explicitement.

La complexité temporelle de `insère` est $O(h)$ pour un arbre de hauteur h et la complexité spatiale également pour deux raisons : il faut une pile de récursion de hauteur $O(h)$ pour atteindre le point d'insertion car la récursivité n'est pas terminale, et il y a création de $O(h)$ nouveaux nœuds.

La méthode d'insertion ci-dessus est appelée *insertion aux feuilles* car elle place le nouvel élément sur une branche vide de l'ancien arbre (certains auteurs appellent « feuilles » les branches vides d'un arbre binaire) en modifiant « aussi peu que possible » la structure de ce dernier. En particulier la racine est conservée sauf dans le cas d'un arbre vide. Le modèle probabiliste d'arbre construit aléatoirement correspond à ce type d'insertion : si σ est une permutation aléatoire des entiers $1, 2, \ldots, n$ et si l'on crée un arbre binaire de recherche en insérant successivement les étiquettes $\sigma(1), \ldots, \sigma(n)$ dans un arbre initialement vide, alors $\sigma(n)$ est inséré

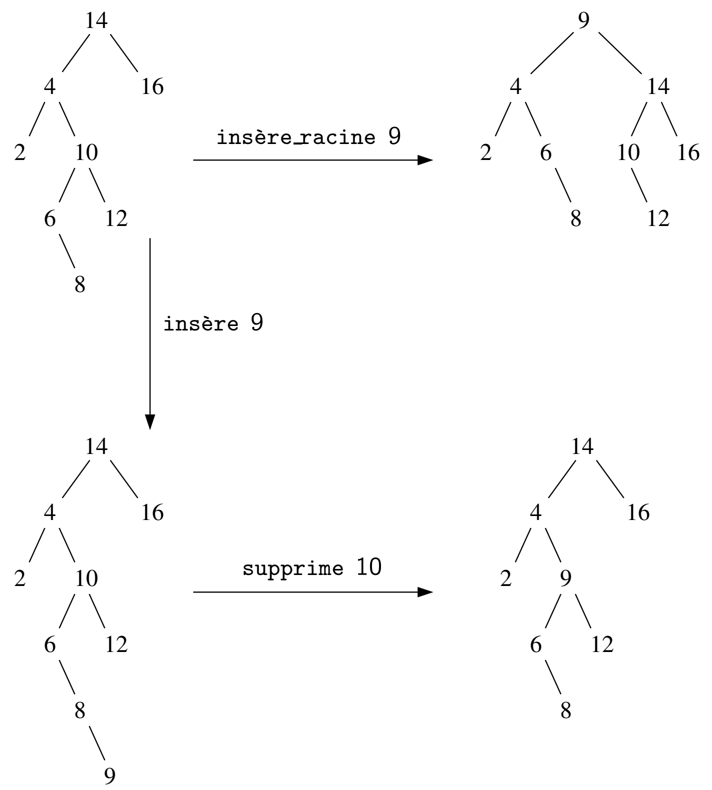


Figure 23 : insertion et suppression dans un arbre binaire

sur la k-ème branche vide en ordre infixe si et seulement $\sigma(n) = k$, ce qui se produit avec une probabilité $1/n$ indépendante de k.

Une autre méthode d'insertion, appelée *insertion à la racine*, consiste à placer l'étiquette à insérer à la racine du nouvel arbre. Pour cela on découpe l'arbre initial en deux arbres dont les étiquettes sont respectivement inférieures à x et supérieures ou égales à x, puis on rassemble ces arbres sous une nouvelle racine :

```
(* découpe l'arbre a en deux arbres séparés par l'étiquette x *)
let rec découpe compare x a = match a with
| B_vide         -> (B_vide,B_vide)
| B_noeud(e,g,d) -> if (compare x e) = PLUSGRAND
   then let (b,c) = découpe compare x d in (B_noeud(e,g,b),c)
   else let (b,c) = découpe compare x g in (b,B_noeud(e,c,d))
;;

(* insère l'étiquette x dans a à la racine *)
let insère_racine compare x a =
  let (b,c) = découpe compare x a in B_noeud(x,b,c);;
```

Comme pour l'insertion aux feuilles, les complexités temporelle et spatiale sont $O(h)$.

8-3 Suppression

Soit a un arbre binaire de recherche et x un nœud de a que l'on veut supprimer. Si x n'a pas de fils on le remplace par une branche vide, et s'il n'a qu'un fils, y, on remplace la branche de racine x par celle de racine y. Dans les deux cas on obtient encore un arbre binaire de recherche car aucun nœud ne change de position par rapport à ses ascendants. Lorsque x a un fils gauche y et un fils droit z, soient y' le prédécesseur et z' le successeur de x dans l'ordre de parcours infixe. y' est le plus à droite des descendants de y et z' est le plus à gauche des descendants de z. On peut alors au choix : retirer y' de la descendance de y ou retirer z' de la descendance de z et mettre l'étiquette du nœud retiré à la place de celle de x. Dans les deux cas la propriété d'un arbre de recherche est conservée : toutes les étiquettes à gauche sont inférieures à celle de la racine qui est inférieure à toutes les étiquettes à droite.

```
(* supprime l'élément le plus à droite d'un arbre   *)
(* renvoie l'étiquette supprimée et le nouvel arbre *)
let rec suppmax(a) = match a with
| B_vide              -> failwith "suppression impossible"
| B_noeud(e,g,B_vide) -> (e,g)
| B_noeud(e,g,d)      -> let (e',d') = suppmax(d) in
                         (e',B_noeud(e,g,d'))
;;

(* recherche si l'étiquette x figure dans a et la supprime *)
let rec supprime compare x a = match a with
| B_vide         -> failwith "x n'est pas dans a"
| B_noeud(e,g,d) -> match compare x e with
   | PLUSGRAND -> B_noeud(e,g,supprime compare x d)
   | PLUSPETIT -> B_noeud(e,supprime compare x g,d)
   | EQUIV     -> if g = B_vide then d
             else if d = B_vide then g
             else let (e',g') = suppmax(g) in B_noeud(e',g',d)
;;
```

Compte tenu de ce qui précède, la correction de `supprime` est équivalente à celle de `suppmax` et celle-ci s'établit par récurrence sur la hauteur de l'arbre considéré. La complexité temporelle dans le pire des cas de `suppmax` est proportionnelle à la hauteur de l'arbre dont on supprime le maximum, et la complexité temporelle dans le pire des cas de `supprime` est proportionnelle à la hauteur h de l'arbre initial. Il y a création de $O(h)$ nouveaux nœuds et récursion non terminale, donc la complexité spatiale est aussi $O(h)$.

Si l'on part d'un arbre vide et si l'on insère des étiquettes distinctes dans un ordre aléatoire, alors la profondeur moyenne d'un nœud est $O(\ln n)$ et donc *l'opération* de recherche, insertion ou suppression suivante a une complexité temporelle et spatiale moyenne $O(\ln n)$. Mais cette propriété n'est pas récurrente : après une suite d'insertions et de suppressions aléatoires la distribution de probabilité des arbres obtenus n'est pas celle des arbres « construits aléatoirement », et n'est pas connue théoriquement. Expérimentalement il semble que les arbres bien équilibrés sont plus fréquents que dans le modèle « arbres construits aléatoirement » et que la profondeur moyenne reste $O(\ln n)$ ([KNUTH] vol. 3, ch. 6).

8-4 Exercices

Exercice 8-1 : contrôle d'un arbre de recherche
Écrire une fonction testant si un arbre binaire est bien un arbre de recherche.

Exercice 8-2 : complexité de la création d'un arbre de recherche
Soit T_n le temps maximal de création d'un arbre binaire de recherche à partir d'une liste quelconque de n éléments en procédant uniquement à des comparaisons entre ces éléments pour décider des positions dans l'arbre où les placer. Montrer que $n \ln n = O(T_n)$.

Exercice 8-3 : transformation d'un vecteur trié en arbre de recherche
On dispose d'un vecteur trié de n éléments. Écrire une fonction CAML construisant un arbre binaire de recherche équilibré contenant ces éléments.

Exercice 8-4 : comparaison entre l'insertion aux feuilles et à la racine
On constitue un arbre binaire de recherche par insertion de n éléments dans un arbre initialement vide. Quel rapport y a-t-il entre les arbres obtenus par les deux méthodes d'insertion ?

Exercice 8-5 : fusion de deux arbres de recherche
Écrire une fonction réalisant la fusion de deux arbres binaires de recherche. Analyser sa complexité temporelle.

Exercice 8-6 : analyse du tri rapide
Soit σ une permutation des entiers de $[\![1, n]\!]$. Montrer que le nombre de comparaisons effectuées lors du tri rapide de la liste $(\sigma(1), \dots, \sigma(n))$ est égal au nombre de comparaisons effectuées lors de l'insertion aux feuilles de $\sigma(1), \dots, \sigma(n)$ dans cet ordre dans un arbre binaire de recherche initialement vide. En déduire que le nombre moyen de comparaisons effectuées pour trier rapidement une permutation aléatoire de $[\![1, n]\!]$ est $\Theta(n \ln n)$. Le code du tri rapide est :

```
let rec découpe a liste = match liste with
| []       -> [],[]
| x::suite -> let (u,v) = découpe a suite in
              if x < a then (x::u,v) else (u,x::v)
;;

let rec quicksort liste = match liste with
| []       -> []
| a::suite -> let (u,v) = découpe a suite in
              (quicksort u) @ (a :: (quicksort v))
;;
```

Chapitre 9

Manipulation d'expressions formelles

9-1 Introduction

On s'intéresse ici à la manipulation d'expressions arithmétiques contenant des constantes, des variables, des fonctions et les opérateurs binaires usuels `+`, `-`, `*`, `/` et éventuellement `^` (puissance). Un système de calcul formel est défini par deux caractéristiques :

- la représentation interne des expressions ;
- les manipulations que l'on veut effectuer.

La représentation interne, généralement sous forme d'arbre, n'est importante que par son influence sur la complexité temporelle et spatiale des manipulations que l'on veut effectuer, elle doit donc être choisie une fois que ces manipulations sont spécifiées. Parmi les manipulations courantes on peut citer :

- L'évaluation d'une formule quand certaines variables sont connues. Lorsque toutes les variables sont connues, l'évaluation peut produire une constante, mais aussi une formule sans variables. Par exemple l'évaluation de la formule $\sqrt{x+1}+\sqrt{y-1}$ avec les conditions $x = 1$, $y = 2$ peut produire la constante 2.4142136 ou la formule $1+\sqrt{2}$.

- Les transformations élémentaires : développement d'un produit, factorisation, substitution. Le développement est relativement simple à effectuer mais produit généralement une formule de grande taille : $(1+x)^{100}$ est transformé en une somme de 101 termes. La factorisation est nettement plus délicate : on peut facilement factoriser une somme autour d'un facteur commun présent dans chaque terme, mais factoriser un polynôme tel que

x^4+x^2+1 en $(x^2+x+1)(x^2-x+1)$ est bien plus délicat. De même que le développement, la factorisation peut produire une formule plus grande que celle dont on est parti, par exemple :

$$\begin{aligned} x^{20}-1 = (x-1)(x+1)(x^2+1)(x^4+x^3+x^2+x+1) \\ *(x^4-x^3+x^2-x+1)(x^8-x^6+x^4-x^2+1). \end{aligned}$$

- La simplification : transformer une formule donnée en une formule *équivalente* mais plus simple, par exemple :

$$\frac{x-1}{x^2-1} \equiv \frac{1}{x+1} \; ;$$
$$\ln(x+\sqrt{x^2+1}) + \ln(-x+\sqrt{x^2+1}) \equiv 0.$$

  Comme la factorisation, la simplification est une opération difficile, et ceci d'autant plus que le résultat attendu est défini de manière ambiguë. On peut d'ailleurs objecter que $(x-1)/(x^2-1)$ et $1/(x+1)$ ne sont pas réellement équivalentes, la deuxième expression est définie pour $x = 1$ mais pas la première.

- La dérivation par rapport à une variable : là le calcul est facile à conduire en appliquant les règles de dérivation. Le problème essentiel de la dérivation est la simplification du résultat. L'opération inverse, la primitivation, est très difficile à conduire car on ne dispose pas d'algorithme pour cela (mais il existe des cas particuliers importants dans lesquels la primitivation peut être automatisée, notamment pour les fractions rationnelles et pour les expressions dites « élémentaires » constituées uniquement d'exponentielles, de logarithmes et de racines d'équations polynomiales (algorithme de RISCH)).

L'évaluation a déjà été étudiée lors du parcours d'un arbre général (cf. section **7-3**). On se limitera dans ce chapitre à la dérivation formelle et aux simplifications élémentaires : regrouper les termes identiques dans une somme ou dans un produit, regrouper les constantes, éliminer les opérations inutiles ($0+x \equiv x$, $x^0 \equiv 1$, ...).

9-2 Représentation des formules

On pourrait reprendre la description des formules vue en **7-3**, mais pour automatiser les simplifications algébriques il est préférable de définir de manière symbolique les opérateurs binaires :

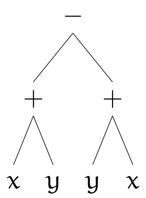
```
type 'a expression =
| Const of 'a                              (* constante  *)
| Var   of string                          (* variable   *)
| Fct   of string * ('a expression)        (* fonction   *)
| Plus  of ('a expression) * ('a expression)  (* somme      *)
```

```
  | Moins of ('a expression) * ('a expression)  (* différence *)
  | Mult  of ('a expression) * ('a expression)  (* produit    *)
  | Div   of ('a expression) * ('a expression)  (* quotient   *)
  | Puiss of ('a expression) * ('a expression)  (* puissance  *)
  ;;
```

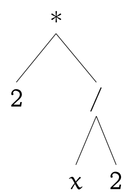
Une fonction est identifiée par son nom. On supposera qu'il existe des fonctions CAML `feval` et `fder` renvoyant le code d'évaluation et la dérivée d'une fonction de nom donné.

Le type des constantes n'est pas spécifié (`'a`) ce qui permet théoriquement d'écrire des algorithmes indépendants de la représentation effective des constantes, mais il faudra bien coder les opérations binaires (`Plus`, `Moins`, `Mult`, `Div` et `Puiss`) entre constantes. Une possibilité est de transmettre en argument aux fonctions de manipulation les opérateurs associés ainsi que les constantes élémentaires 0, 1 et -1, mais elle alourdirait le code de ces fonctions. On supposera pour simplifier que les constantes sont de type `float`. Il s'agit certainement d'un mauvais choix car les calculs sur les « réels machine » sont approchés et les erreurs d'arrondi peuvent gêner la simplification. En pratique, il vaudrait mieux prendre des constantes de type entier et accepter des expressions constantes comme $3-\sqrt{2}$, mais cela conduirait à développer une bibliothèque de calcul sur les entiers de longueur arbitraire et les nombres algébriques sortant largement du cadre de ce cours.

Le problème principal de la représentation précédente est la complication qu'elle entraîne pour les simplifications élémentaires : $(x+y)-(y+x)$ est représenté par l'arbre :



et il n'est pas immédiat de constater sur l'arbre que les x et les y s'en vont. De même, $2*(x/2)$ est représenté par :



et les 2 qui se simplifient sont relativement loin l'un de l'autre. Ce problème est dû au fait que l'on ne prend pas en compte dans la représentation interne la commutativité et l'associativité de $+$ et $*$, ni les caractères « réciproques » de $+$ et $-$ ou $*$ et $/$. Une meilleure solution consiste à définir deux opérateurs n-aires, la combinaison linéaire et le produit avec exposants d'une liste d'expressions :

```
type 'a expression =
| Const of 'a                          (* constante *)
| Var   of string                      (* variable  *)
| Fct   of string * ('a expression)    (* fonction, argument *)
| CL    of ('a * 'a expression) list   (* comb. linéaire     *)
| PG    of ('a * 'a expression) list   (* produit généralisé *)
;;
```

CL $[(a_1, e_1); \ldots; (a_n e_n)]$ représente l'expression $a_1 e_1 + \ldots + a_n e_n$ où les a_i sont des constantes et les e_i des expressions. De même, PG $[(a_1, e_1); \ldots; (a_n e_n)]$ représente le produit généralisé $e_1^{a_1} * \ldots * e_n^{a_n}$. La notation est alourdie, $f + g$ est codée par CL $[(1, f); (1, g)]$, mais en contrepartie il n'y a plus que deux opérateurs au lieu de cinq. Remarquons qu'on a ainsi perdu l'élévation à une puissance non constante, mais on peut la retrouver par la forme exponentielle. L'associativité et la commutativité de $+$ et $*$ ne sont pas directement prises en compte, il faut pour cela des *conventions de représentation* :

– aucun terme d'une combinaison linéaire n'est lui-même une combinaison linéaire ;
– aucun facteur d'un produit généralisé n'est lui-même un produit généralisé ;
– il est défini une relation d'ordre total sur les expressions, et les termes d'une combinaison linéaire ou d'un produit généralisé sont triés suivant cet ordre, deux termes successifs étant distincts.

La condition de tri des termes permet de garantir une sorte d'unicité de la représentation mémoire d'une formule ce qui est indispensable pour tester si deux formules sont égales et aussi pour les classer, mais elle ne suffit pas. La principale cause de non unicité est la présence des constantes, et on ajoute les conventions suivantes :

– dans une combinaison linéaire aucun coefficient n'est nul et toutes les constantes sont regroupées en une seule avec coefficient 1 si elle est non nulle, en aucune sinon ;
– dans un produit généralisé aucun exposant n'est nul et il n'y a pas de facteur constant ;
– un produit généralisé comporte au moins deux facteurs ou un seul facteur avec un exposant différent de 1.

En effet, les facteurs constants dans un produit généralisé peuvent être « sortis » du produit et placés en coefficient d'une combinaison linéaire, donc cette interdiction de facteurs constants dans un produit ne porte pas à conséquence (mais elle ne serait pas défendable si l'on voulait gérer des constantes symboliques comme $2^{1/2}$ sans les convertir en approximation numérique).

Les conventions précédentes font partie de la représentation interne des expressions, même si elles n'apparaissent pas en tant que telles dans la déclaration CAML du type `expression`. Il est donc possible de construire des expressions « non normalisées », mais il faut s'engager à les mettre sous forme normale à la fin des calculs. Remarquons enfin que les simplifications du type :

$$\frac{x-1}{x^2-1} \equiv \frac{1}{x+1},$$
$$\ln(x+\sqrt{x^2+1}) + \ln(-x+\sqrt{x^2+1}) \equiv 0,$$

ne sont pas prises en compte : la première relève d'algorithmes de factorisation et la deuxième de règles de simplification fonctionnelles.

Relation d'ordre

Une manière simple de comparer deux expressions consiste à les parcourir simultanément en profondeur d'abord jusqu'à ce que l'on trouve une différence. On déclare la première expression plus grande ou plus petite que l'autre en fonction de cette première différence si l'on en trouve une, égale à l'autre si l'on n'en trouve pas :

```
let rec compare e e' = match (e,e') with

| (Const(a),Const(b)) -> if a < b then PLUSPETIT
                      else if a > b then PLUSGRAND else EQUIV
| (Const(_),_)        -> PLUSPETIT
| (_,Const(_))        -> PLUSGRAND

| (Var(a),Var(b))     -> if a < b then PLUSPETIT
                      else if a > b then PLUSGRAND else EQUIV
| (Var(_),_)          -> PLUSPETIT
| (_,Var(_))          -> PLUSGRAND

| (Fct(f,u),Fct(g,v)) -> if f < g then PLUSPETIT
                      else if f > g then PLUSGRAND else compare u v
| (Fct(_),_)          -> PLUSPETIT
| (_,Fct(_))          -> PLUSGRAND

| (CL(l),CL(l'))      -> compare_listes l l'
| (CL(_),_)           -> PLUSPETIT
| (_,CL(_))           -> PLUSGRAND

| (PG(l),PG(l'))      -> compare_listes l l'

and compare_listes l l' = match (l,l') with
| ([],[]) -> EQUIV
| ([],_)  -> PLUSPETIT
| (_,[])  -> PLUSGRAND
| ((a,x)::b,(a',x')::b') -> if a < a' then PLUSPETIT
                         else if a > a' then PLUSGRAND
                         else match compare x x' with
                         | EQUIV -> compare_listes b b'
                         | c     -> c
;;
```

< désigne à la fois l'opérateur de comparaison entre réels et celui entre chaînes de caractères. La fonction `compare` ci-dessus compare successivement tous les nœuds des expressions e et e' dans l'ordre préfixe, et termine dès qu'une différence est détectée, soit dans un constructeur soit dans une étiquette (valeur d'une constante ou d'un coefficient, nom d'une variable ou d'une fonction). L'ordre parmi les constructeurs est défini arbitrairement par :

$$\texttt{Const} < \texttt{Var} < \texttt{Fct} < \texttt{CL} < \texttt{PG}.$$

Par récurrence sur la taille de e, `compare e e'` termine et renvoie un résultat `PLUSPETIT`, `PLUSGRAND` ou `EQUIV`, ce dernier si et seulement si $e = e'$. La relation ainsi définie est bien une relation d'ordre total : total puisqu'un appel à `compare` termine ; réflexive, transitive et antisymétrique par récurrence sur la plus grande taille des formules à comparer. Il s'agit en fait d'un cas particulier de relation d'ordre sur un produit cartésien appelée *ordre lexicographique* parce que c'est l'ordre utilisé pour classer les mots dans un dictionnaire. Dans le langage CAML la relation < est en fait une relation *polymorphe* classant deux objets de même type quel que soit ce type pourvu qu'il ne contienne pas de valeurs fonctionnelles. Elle est grosso modo implémentée comme la fonction `compare` ci-dessus, mais les comparaisons « élémentaires » portent sur les représentations binaires des objets à comparer ce qui lui confère son caractère polymorphe. En pratique on pourra donc utiliser < à la place de `compare`, le résultat d'une comparaison étant alors un booléen.

On peut donc comparer deux expressions. Le temps de comparaison dans le pire des cas est proportionnel à la plus petite des tailles des expressions comparées, mais si l'on compare deux expressions aléatoires alors on peut considérer que le temps moyen de comparaison est constant (cf. exercice **6-3**).

9-3 Dérivation

Soit e une expression et x une variable (une chaîne de caractères). L'objectif est de calculer une expression e' équivalente à $\partial e/\partial x$. Notons qu'il n'y a pas unicité du résultat, même si l'on impose à e' de respecter les conventions de représentation des expressions normales. Par exemple l'expression :

$$e = x\cos(x)\sin(x)$$

admet pour dérivées par rapport à x les deux expressions suivantes (entre autres) :

$$e'_1 = \cos(x)\sin(x) + x\cos(x)^2 - x\sin(x)^2,$$
$$e'_2 = \cos(x)\sin(x) + x(\cos(x)^2 - \sin(x)^2).$$

Par ailleurs, bien qu'il soit souhaitable de retourner une expression sous forme normale, on se limitera dans un premier temps au calcul d'une expression de $\partial e/\partial x$ non simplifiée, quitte à la simplifier après coup (voir cependant l'exercice **9-7** à ce sujet).

Le calcul de e' s'effectue récursivement en appliquant les règles usuelles de dérivation :

- $\dfrac{\partial c}{\partial x} = 0$ pour toute constante c ;
- $\dfrac{\partial x}{\partial x} = 1$;
- $\dfrac{\partial y}{\partial x} = 0$ si y est une variable différente de x ;
- $\dfrac{\partial f(u)}{\partial x} = f'(u) * \dfrac{\partial u}{\partial x}$ pour une fonction f et une expression u ;
- $\dfrac{\partial}{\partial x}(a_1e_1 + \ldots + a_ne_n) = a_1\dfrac{\partial e_1}{\partial x} + \ldots + a_n\dfrac{\partial e_n}{\partial x}$ pour des expressions $e_1, \ldots, e_n$ et des constantes $a_1, \ldots, a_n$;
- $\dfrac{\partial}{\partial x}(e_1^{a_1} \ldots e_n^{a_n}) = a_1\dfrac{\partial e_1}{\partial x}e_1^{a_1-1}e_2^{a_2}\ldots e_n^{a_n} + \ldots + a_n\dfrac{\partial e_n}{\partial x}e_1^{a_1}\ldots e_{n-1}^{a_{n-1}}e_n^{a_n-1}$ pour des expressions $e_1, \ldots, e_n$ et des constantes $a_1, \ldots, a_n$.

```
let rec dérive x e = match e with
| Const(_)  -> Const(0.0)
| Var(v)    -> if v = x then Const(1.0) else Const(0.0)
| Fct(f,u)  -> PG [(1.0,dérive x u); (1.0,fder f u)]
| CL(liste) -> CL(map (dériveCL x) liste)
| PG(liste) -> CL(map (dérivePG liste x) liste)

and dériveCL x (a,e) = (a,dérive x e)
and dérivePG liste x (a,e) =
     (a, PG( (1.0,dérive x e) :: (-1.0,e) :: liste ))
;;
```

`map` applique la fonction passée en premier argument à la liste passée en deuxième argument. La fonction `dérivePG` accole à `liste` (représentant $e_1^{a_1} * \ldots * e_n^{a_n}$) les facteurs $\partial e_i/\partial x$ et e_i^{-1} et place le coefficient a_i, en accord avec la formule de dérivation logarithmique d'un produit. L'expression n'est certainement pas simplifiée puisque $e_i^{a_i}$ figure dans `liste`, la fonction de simplification décrite à la section **9-4** rectifiera ceci.

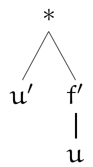
La dérivation des fonctions d'une variable fait appel à une fonction `fder` chargée de constituer la dérivée d'une fonction spécifiée par son nom et appliquée à une expression u. Le code de `fder` est de la forme :

```
let fder f u = match f with
| "exp" -> Fct("exp",u)             (* exp'(u) = exp(u)  *)
| "ln"  -> PG [(-1.0,u)]            (* ln'(u)  = u^-1    *)
| "cos" -> CL [(-1.0,Fct("sin",u))] (* cos'(u) = -sin(u) *)
| "sin" -> Fct("cos",u)             (* sin'(u) = cos(u)  *)
   ......                   (* autres dérivées usuelles *)
| _     -> Fct(f^"'",u)             (* f'(u)   = f'(u)   *)
;;
```

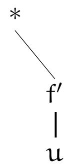
La dernière ligne constitue la dérivée d'une fonction inconnue en accolant une prime à son nom. On peut ainsi dériver des fonctions formelles.

Correction et complexité

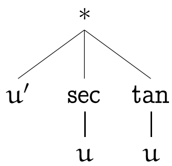
La correction de `dérive` s'établit par récurrence sur la taille ou la hauteur d'une expression. Il suffit de constater que l'on a effectivement codé les règles de dérivation des fonctions usuelles et des opérations (combinaison linéaire et produit). La complexité est plus délicate à évaluer : `dérive x e` s'exécute en temps constant lorsque e est une constante ou une variable. En admettant que `map` a une complexité linéaire, le temps de dérivation d'une combinaison linéaire, non compte tenu des appels récursifs, est proportionnel au nombre de termes. La dérivation de $f(u)$ produit l'arbre :



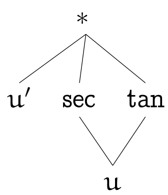
donc il faut compter le temps de construction du sous-arbre :



qui est à priori au moins égal à la taille de u. Lorsque f' est compliquée, u peut apparaître plusieurs fois. Par exemple si l'on dispose dans la batterie des fonctions usuelles de la fonction sécante ($\sec x = 1/\cos x$) alors la dérivation de $\sec(u)$ produit l'arbre :



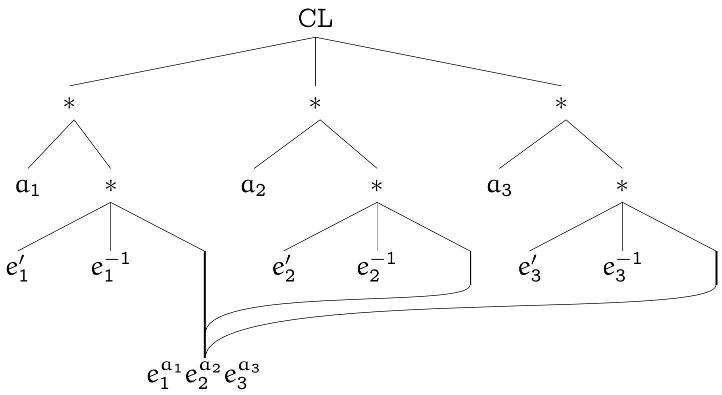
car $\sec'(x) = \sec(x)\tan(x)$ (ou $\sin(x)/\cos^2(x)$). En fait, dans l'implémentation de CAML, les sous-arbres ne sont pas recopiés et toutes les données complexes sont référencées par des pointeurs, donc l'arbre précédent est en réalité un *graphe* :



```
let e = expression "x^2 + cos(2*x+t)";;
e : float expression =
       _____CL_____
      /            \
    _*_          ___*____
   /   \        /        \
 1.0    PG    1.0         cos
        |                  |
        ^                _CL__
       / \              /     \
      x  2.0           *       *
                      / \     / \
                   2.0  x  1.0  t

#let e' = dérive "x" e;;
e' : float expression =
          ____________________CL_____________________
         /                                           \
    ____*_____                              _____________*__________...
   /          \                            /
 1.0           CL                       1.0                ______...
               |                                          /
         ______*______                              ____^_____
        /             \                            /          \
      2.0       ______PG_______               __CL___        1.0
               /       |       \             /       \
              ^_       ^_       ^           *_        *_
             /  \     /  \     / \         /  \      /  \
           1.0  1.0  x   -1.0 x  2.0      2.0  1.0  1.0  0.0

#simplifie(e');;
- : float expression =
      ___CL____
     /         \
    *        ___*____
   / \      /        \
 2.0  x   -2.0        sin
                       |
                     _CL__
                    /     \
                   *       *
                  / \     / \
               1.0  t  2.0  x
```

Figure 24 : dérivation et simplification

Pour cette implémentation, la dérivation de $f(u)$ prend un temps constant une fois que l'on connaît u', avec des fonctions usuelles raisonnables et en nombre déterminé à l'avance, la dérivation d'une fonction formelle prenant un temps proportionnel à la longueur du nom de cette fonction (car il y a recopie du nom pour accoler une prime). De même, la dérivation d'un produit généralisé produit un arbre dans lequel le produit complet est partagé entre les différents termes de la dérivée :



Le temps de construction de cet « arbre » est proportionnel au nombre de facteurs. La complexité temporelle de la dérivation d'une expression e est donc proportionnelle au nombre de nœuds de e s'il n'y a pas de fonction formelle. Sous la même hypothèse, la complexité spatiale est également linéaire par rapport au nombre de nœuds puisqu'il n'est créé qu'un nombre borné de nouveaux nœuds à chaque étape ou aussi parce que la complexité spatiale sur une machine séquentielle est majorée par la complexité temporelle. Notons que la taille mémoire de e peut être bien plus faible que son nombre de nœuds s'il y a déjà partage de sous-arbres dans e, dans ce cas on calcule plusieurs fois la dérivée d'une même sous-expression.

9-4 Simplification

L'exemple figure 24 illustre le caractère catastrophique de la dérivation formelle sans simplification. Étant donnée une expression e obtenue par dérivation, substitution, évaluation partielle ou toute autre manipulation formelle, il est nécessaire de transformer e en une expression e' équivalente à e et normalisée selon les conventions de la section **9-2**. On considère ici que deux formules sont équivalentes si l'on peut passer de l'une à l'autre par application des règles usuelles de commutativité, associativité, distributivité de $*$ sur $+$ et de ^ sur $*$, et évaluation d'expressions constantes. Le schéma général de simplification est :

parcourir e *en profondeur d'abord ; pour chaque nœud* n *simplifier récursivement les branches de* n *puis simplifier* n *lui-même.*

L'ordre de simplification est donc postfixe. On peut aussi procéder à une simplification préfixe, par exemple transformer un produit contenant zéro en zéro sans avoir besoin de simplifier les autres facteurs, mais il n'est pas garanti que cela suffise, il se peut qu'un facteur s'avère nul après simplification seulement. La validité de la simplification postfixe est justifiée ci-après.

Algorithme de simplification

– *Une constante et une variable sont déjà simplifiées.*

– *Soit* $e = f(u)$ *où* f *est une fonction et* u *une expression : simplifier* u *en* u' *puis, si* u' *est une constante et* f *est évaluable, remplacer* e *par la constante* $f[u']$ *sinon par l'expression* $f(u')$*.*

– *Soit* $e = a_1e_1 + \ldots + a_ne_n$ *où les* a_i *sont des coefficients et les* e_i *des expressions : simplifier chaque* e_i *en* e_i'*. Si* e_i' *est une combinaison linéaire, distribuer le coefficient* a_i*. On obtient ainsi une combinaison linéaire* $b_1f_1 + \ldots + b_pf_p$ *où les* b_j *sont des coefficients et les* f_j *des expressions simplifiées et ne sont pas des combinaisons linéaires. Trier la liste et regrouper les termes correspondant à la même sous-expression, supprimer les termes ayant un coefficient nul, regrouper les constantes en une seule avec coefficient* 1 *si elle est non nulle, aucune sinon. Si la liste obtenue est vide, remplacer* e *par zéro ; s'il ne reste plus qu'un terme,* f*, avec un coefficient* 1*, remplacer* e *par* f *sinon remplacer* e *par la combinaison linéaire obtenue.*

– *Soit* $e = e_1^{a_1} * \ldots * e_n^{a_n}$ *où les* a_i *sont des exposants et les* e_i *des expressions : simplifier chaque* e_i *en* e_i'*. Si* e_i' *est un produit généralisé, distribuer l'exposant* a_i*. Si* e_i' *est une combinaison linéaire à un seul terme,* $e_i' = af$*, la transformer en produit* $a*f$ *puis distribuer l'exposant* a_i *(il est possible ici que* f *soit aussi un produit, distribuer l'exposant sur chaque facteur dans ce cas). Trier la liste des facteurs obtenue en regroupant les facteurs égaux et en supprimant les facteurs ayant un exposant nul. Regrouper toutes les constantes en une seule,* c*, et soit* f *la liste des facteurs restants. Si* f *est vide, remplacer* e *par la constante* c *; si* $c = 0$ *remplacer* e *par zéro ; si* $c = 1$ *et* f *ne contient qu'un terme* g *avec un exposant aussi égal à* 1*, remplacer* e *par* g *; si* $c = 1$ *et* f *contient plusieurs facteurs ou un facteur avec un exposant différent de* 1*, remplacer* e *par* $\mathtt{PG}(f)$ *; si* $c \neq 1$ *et* f *ne contient qu'un terme,* g*, qui est une combinaison linéaire avec un exposant* 1*, distribuer* c *et remplacer* e *par la combinaison linéaire obtenue ; dans tous les autres cas remplacer* e *par la combinaison linéaire* $c\mathtt{PG}(f)$*.*

Il s'agit presque d'un algorithme, il suffit de préciser certains détails comme la manière de reconnaître si les expressions intermédiaires sont de la forme attendue, la manière d'effectuer les opérations entre constantes, la manière de distribuer un coefficient ou un exposant et la relation de comparaison utilisée pour les tris. Le codage de cet « algorithme » en CAML est fastidieux mais ne pose pas de problème particulier, il fait l'objet de l'exercice **9-6**.

Terminaison

Montrons par récurrence sur la hauteur h de e que la simplification de e termine : pour $h = 0$, e est une feuille, donc une constante ou une variable et la terminaison est évidente. Si la simplification termine pour toute expression de hauteur $h-1$ et si e est de hauteur h alors e est de la forme $f(u)$ ou $a_1e_1+\ldots+a_ne_n$ ou $e_1^{a_1}\ldots e_n^{a_n}$: la branche u ou les branches e_i sont simplifiées par hypothèse en un temps fini, et il reste à vérifier qu'on effectue un nombre fini d'opérations après simplification des branches. C'est immédiat si $e = f(u)$.

Si $e = a_1e_1 + \ldots + a_ne_n$ il faut vérifier qu'on n'appliquera pas une infinité de fois la règle de distributivité : on constate ici que l'algorithme est incomplètement spécifié puisqu'il n'est pas dit si l'on doit distribuer récursivement le coefficient a_i dans le cas où e'_i contient des combinaisons linéaires imbriquées. On peut remarquer que ceci ne peut pas se produire si l'algorithme est correct puisque e'_i est supposée simplifiée, mais la correction de l'algorithme n'est pas prouvée ... Cette situation peut se régler de plusieurs manières : démontrer d'abord la correction ou la démontrer ensuite mais en vérifiant qu'on n'utilise pas l'hypothèse de terminaison, démontrer en même temps la correction et la terminaison puisqu'ici on a besoin de l'hypothèse de correction au rang $h-1$ et non h, revoir la description de l'algorithme en précisant qu'on ne distribue pas récursivement (cette précision ne changeant rien si l'algorithme est correct) ou remarquer que, même si l'on distribue récursivement, il y a un nombre fini de e'_i chacun ayant un nombre fini de descendants, donc on ne pourra pas appliquer une infinité de fois cette règle de distributivité. Une fois la distribution des coefficients effectuée, le tri des termes, leur regroupement et l'élimination des constantes prennent un temps manifestement fini donc la simplification d'une combinaison linéaire de hauteur h termine.

Si e est un produit généralisé de hauteur h, on constate comme dans le cas précédent qu'on obtient en un temps fini une liste de facteurs isolés, la transformation des combinaisons linéaires à un seul terme en produit ne pouvant être effectuée une infinité de fois même si elle est appliquée récursivement. L'élimination des facteurs constants et les simplifications finales ont aussi un temps d'exécution fini donc par récurrence, l'algorithme de simplification termine ! ∎

Correction

Déjà, on montre par une récurrence immédiate que l'expression e' déduite de e par simplification est équivalente à e *sous l'hypothèse que les calculs sur les constantes sont exacts*. Avec des constantes de type `float` il n'y a aucune garantie d'équivalence et e' est seulement une approximation de e ce qui ne veut rien dire dans l'absolu. En supposant les calculs exacts, il reste à prouver que e' satisfait aux conventions de normalisation définies en **9-2**, ce qui se fait encore par récurrence sur la hauteur h de e :

– *Aucun terme d'une combinaison linéaire n'est lui-même une combinaison linéaire* : si e est une combinaison linéaire, les termes simplifiés e'_i ne contiennent pas de combinaisons linéaires imbriquées par hypothèse de récurrence et si e'_i est une combinaison linéaire elle est éclatée par distributivité donc e' ne contient pas de combinaison linéaire imbriquée. Si e est un produit généralisé, les facteurs e'_i ne contiennent pas de combinaisons linéaires imbriquées et e' est soit un produit de certains de ces termes (ou de leurs descendants), soit une combinaison linéaire de produit avec plusieurs facteurs ou un seul si son exposant est différent de 1, soit l'un des e'_i ou un descendant. En aucun cas, e' ne contient de combinaison linéaire imbriquée. Lorsque e est une constante, une variable, ou de la forme $f(u)$ il n'y a pas création de nouvelle combinaison linéaire donc là encore e' ne contient pas de combinaison linéaire imbriquée.

- *Aucun facteur d'un produit généralisé n'est lui-même un produit généralisé* : par un raisonnement analogue.

- *Les sous-expressions d'une combinaison linéaire ou d'un produit généralisé sont triées, deux expressions successives étant distinctes* ;
- *dans une combinaison linéaire, aucun coefficient n'est nul et toutes les constantes sont regroupées en une seule (avec coefficient* 1*) si elle est non nulle, en aucune sinon* ;
- *dans un produit généralisé, aucun exposant n'est nul et il n'y a pas de facteur constant* ;
- *un produit généralisé comporte au moins deux facteurs ou un seul facteur avec un exposant différent de* 1 :

par construction. ∎

Complexité

Soit e une expression de taille n que l'on simplifie en e'. Chacune des règles de simplification réduit le nombre de nœuds de l'expression manipulée, sauf peut-être la règle stipulant de remplacer un produit ayant un coefficient multiplicatif $c \neq 1$ par une combinaison linéaire à un terme de coefficient c. Mais il est impossible d'obtenir un coefficient $c \neq 1$ s'il n'y a pas de facteur constant dans le produit après simplification des facteurs et éclatement des produits imbriqués. On crée un nœud supplémentaire (la combinaison linéaire) en ayant supprimé au moins un facteur (constant) donc cette dernière règle n'augmente pas le nombre de nœuds de l'expression à simplifier. Ainsi, il est garanti que e' a moins de nœuds que e et que, au moins en ce sens, e' est plus simple que e.

Soit $T(e)$ le temps de simplification de e. On a les relations de récurrence :

$$\begin{aligned}
T(e) &\leqslant \alpha \quad \text{lorsque } e \text{ est une constante ou une variable,}\\
T(f(u)) &\leqslant \alpha + T(u),\\
T(a_1 e_1 + \ldots + a_p e_p) &\leqslant T(e_1) + \ldots + T(e_p) + \beta n \ln n,\\
T(e_1^{a_1} \ldots e_p^{a_p}) &\leqslant T(e_1) + \ldots + T(e_p) + \beta n \ln n,
\end{aligned}$$

en supposant que le temps d'évaluation d'une fonction f est constant et que le temps du tri d'une liste de longueur inférieure ou égale à n est $O(n \ln n)$. Cette deuxième hypothèse ne va pas de soi car il faut tenir compte du temps de comparaison de deux expressions de tailles arbitraires, ce temps est considéré ici comme constant en moyenne. Si T_n désigne le temps de simplification d'une expression de taille n dans le pire des cas, on a donc :

$$
\begin{aligned}
& T_1 \leqslant \alpha, \\
& T_n \leqslant \max(\alpha + T_{n-1}, \\
& \qquad\qquad T_{n_1} + \ldots + T_{n_p} + \beta n \ln n,\ p \geqslant 1,\ n_i \geqslant 1,\ n_1 + \ldots + n_p = n - 1), \\
& \quad \text{pour } n \geqslant 2.
\end{aligned}
$$

Soit (X_n) la suite définie par $X_1 = \alpha$ et $X_n = X_{n-1} + \alpha + \beta n \ln n$ pour $n \geqslant 2$. On a $X_n = n\alpha + \beta \sum_{k=2}^{n} k \ln k$, et si $n_1 + \ldots + n_p = n - 1$ alors :

$$
\begin{aligned}
& X_{n_1} + \ldots + X_{n_p} + \beta n \ln n \\
& \qquad = (n_1 + \ldots + n_p)\alpha + \beta \underbrace{\left(\sum_{k=2}^{n_1} k \ln k + \ldots + \sum_{k=2}^{n_p} k \ln k\right)}_{n-p-1 \text{ termes}} + \beta n \ln n \\
& \qquad \leqslant X_n.
\end{aligned}
$$

Ceci permet de prouver par récurrence que $T_n \leqslant X_n$ et l'on a $X_n = \Theta(n^2 \ln n)$ par comparaison série-intégrale, donc $T_n = O(n^2 \ln n)$. Cette borne est atteinte par exemple lorsque e représente une addition en chaîne (cf. figure 25).

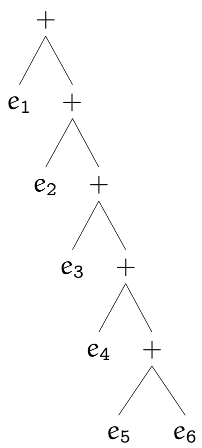


Figure 25 : cas extrême de simplification

Le temps de simplification d'une expression à n nœuds dans le pire des cas est donc $\Theta(n^2 \ln n)$, si les comparaisons entre sous-expressions lors des tris prennent un temps moyen constant.

9-5 Exercices

Exercice 9-1 : compatibilité de la relation de comparaison avec les transformations d'expressions
Soient e, e' et f trois expressions et x une variable. On note $\texttt{subst}(x = f, e)$ l'expression obtenue en remplaçant chaque occurrence de la variable x dans e par l'expression f. On suppose $e < e'$ où $<$ désigne la relation d'ordre entre expressions définie dans le cours. A-t-on, avant simplification, $e + f < e' + f$? $ef < e'f$? $\texttt{subst}(x = f, e) < \texttt{subst}(x = f, e')$? Et après simplification ?

Exercice 9-2 : substitution de sous-expressions
Soient e et f deux expressions. On dit que f *figure* dans e si f est une sous-expression de e ou si f est une combinaison linéaire ou un produit généralisé inclus dans une combinaison linéaire ou un produit généralisé de e. Par exemple $f = x + z$ figure dans $\sin(x + y + z)$ mais pas dans $\sin(2x + y + z)$.

1. Écrire une fonction `figure` qui dit si f figure dans e. e et f seront supposées sous forme normale.
2. Écrire une fonction `substitue` qui remplace dans e chaque occurrence de f par une autre expression g. Il n'est pas demandé de simplifier le résultat.

Exercice 9-3 : complexité de la dérivation
Montrer que les complexités temporelle et spatiale de `dérive` sont $O(n^2)$ pour une expression de taille n s'il y a recopie des sous-arbres lors de la dérivation de $f(u)$ ou de $e_1^{a_1} \dots e_p^{a_p}$.

Exercice 9-4 : dérivation optimisée
Si une expression e comporte des sous-arbres partagés, peut-on éviter de calculer plusieurs fois la dérivée d'un sous-arbre ?

Exercice 9-5 : taille d'un arbre avec partage
Si l'on admet le partage des sous-arbres et si l'on considère qu'un nœud à p fils occupe $p + 1$ mots mémoire, quel est le plus gros arbre que l'on peut stocker dans n mots mémoire ?

Exercice 9-6 : simplification
Transcrire en CAML l'algorithme de simplification du cours.

Exercice 9-7 : dérivation et simplification
Quelle est la meilleure stratégie : dériver puis simplifier, dériver et simplifier en même temps ?

Chapitre 10

Langages réguliers

La notion de langage est liée au problème de *codage* des informations : la langue française permet de coder des idées par des phrases ou des textes (suites de mots). Les mots eux-mêmes sont codés par des suites de lettres ou des suites de phonèmes pour la langue parlée. Un langage de programmation permet de coder un algorithme par une suite de déclarations, ces déclarations étant elles-mêmes codées par des suites de caractères alphanumériques. L'identité génétique d'un individu est codée par une suite de molécules constituant une chaîne d'ADN. Une image filmée par une caméra est codée par une suite de nombres représentant la luminosité et la couleur de chaque point du capteur de la caméra.

De manière informelle, un langage définit quels sont les symboles de base (lettres), quelles suites de lettres constituent des mots valides (lexèmes) et quelles suites de lexèmes constituent des phrases valides. Il s'agit ici d'une description grammaticale : quelles sont les phrases bien formées, ou comment former une phrase correcte. A un niveau supérieur, on peut aussi définir une *interprétation* pour chaque phrase correcte, c'est le *décodage* de l'information transportée par la phrase. Considérons par exemple les phrases :

1. *Il fait beau*
2. *kzw30 xxx tfgrd*
3. *la maison mange le chat*
4. `x=3`

Vues comme suites de caractères, chacune de ces phrases est bien formée ; vues comme suites de mots français, les phrases 1 et 3 sont bien formées, la phrase 2 ne l'est pas car elle contient au moins un mot non français. D'un point de vue grammatical, les phrases 1 et 3 sont aussi bien formées : sujet, verbe, complément d'objet ; mais d'un point de vue *sémantique* la phrase 3 n'a pas d'interprétation valide. La phrase 4 est bien formée selon les règles du langage CAML, et admet une interprétation valide (expression booléenne disant si `x` vaut 3), elle est aussi bien formée pour le langage C et admet une interprétation différente (placer la valeur 3 dans la variable `x`). Les problèmes liés aux langages sont :

- Définir un langage du point de vue *lexical* (lettres autorisées, règles d'assemblage des lettres en lexèmes), *syntaxique* (règles d'assemblage des lexèmes en phrases) et *sémantique* (interprétation d'une phrase valide). Lorsque le langage n'autorise qu'un nombre fini de suites, il suffit de les lister toutes et de donner leur interprétation, c'est ce que fait un dictionnaire pour la description lexicale du français. Par contre lorsque le langage est infini, il s'agit de donner une description finie et non ambiguë de toutes les suites autorisées, et de leur interprétation.

- Décomposer une suite de lettres en lexèmes (analyse lexicale), reconnaître si une suite de lexèmes est valide (analyse syntaxique) et déterminer son interprétation (analyse sémantique).

- Traduire une phrase valide pour un langage en une phrase valide pour un autre langage en respectant au mieux le sens de la phrase.

La division de la description d'un langage en une description lexicale et une description syntaxique permet de simplifier la description globale du langage. Par exemple pour le français il est plus simple de donner la liste des mots, chaque mot étant qualifié par un *attribut* (nom, adjectif, verbe transitif,...) et les règles de formation des phrases :

sujet, verbe, complément d'objet, "."
verbe, sujet, complément d'objet, "?"
verbe, complément d'objet, "!"
...

plutôt que de donner toutes les phrases possibles (qui sont en nombre infini dans le cas du français compte tenu des propositions subordonnées).

En ce qui concerne les langages informatiques, l'analyse lexicale et syntaxique d'un texte est effectuée par un *compilateur* ou un *interpréteur*. Le compilateur reconnaît un programme source valide et le traduit en un programme exécutable (liste d'instructions machine). L'interpréteur reconnaît une phrase valide, la traduit en instructions machine et exécute ces instructions avant de passer à la phrase suivante. Il n'y a pas d'analyse sémantique, aucun logiciel (connu de l'auteur à ce jour) n'est en mesure de dire que le programme :

```
let rec mystère(a,b) = if b = 0 then a else mystère(b,a mod b);;
```

calcule le plus grand diviseur commun aux entiers strictement positifs a et b, ni même que ce programme est équivalent en termes de résultat produit à :

```
let rec énigme(a,b) = if b = 0 then a else énigme(b,abs(a-b));;
```

L'un des objectifs de la théorie des langages est de faciliter la réalisation de « compilateurs de compilateurs », c'est-à-dire de programmes prenant en entrée la description sous une forme adéquate d'un langage de programmation et produisant en sortie un compilateur pour ce langage. Dans ce cours, on se limitera à l'étude des langages admettant une description simple et à la réalisation d'analyseurs lexicaux pour ces langages.

10-1 Définitions

Un *alphabet* est un ensemble fini non vide dont les éléments sont appelés *lettres*. Un *mot* sur l'alphabet A est une suite finie, éventuellement vide, de lettres. On écrit $u = a_1a_2\dots a_n$ pour désigner la suite $(a_1, \dots, a_n)$. Le mot vide est noté ε. La longueur d'un mot u est notée $|u|$. Si u est un mot sur A et x une lettre de A, on note $|u|_x$ le nombre d'occurrences de x dans u. L'ensemble de tous les mots sur A est noté A^*. Un *langage* sur A est un ensemble de mots, c'est-à-dire un sous-ensemble de A^*. Il peut être fini ou infini.

Exemples :

- $A = \{0, 1\}$ (chiffres binaires). A^* est l'ensemble de toutes les suites finies constituées de 0 et de 1. L'ensemble L des suites de 0 et de 1 commençant par un 1 constitue un langage sur A. L peut être interprété comme l'ensemble des représentations en base 2 des entiers naturels non nuls, ceci constitue *une* interprétation de L. On peut aussi interpréter un mot de L comme la représentation binaire d'un entier impair en lisant les chiffres de droite à gauche, ou comme une suite de tirages à pile ou face commençant par pile.

- $A = \{a, b, c, \dots, z\}$, $L = \{$mots français non accentués et sans trait d'union$\}$. Ici L est un langage fini.

- $A = \{0, 1, \dots, 9, +, -, *, /, (,)\}$, L est l'ensemble des expressions arithmétiques correctement parenthésées (ceci constitue une description ambiguë qu'il faudrait préciser). Un mot de L peut en général être interprété de plusieurs manières : $3+4*5$ peut être interprété comme $3+(4*5)$ ou $(3+4)*5$, il faut fixer des règles de priorité pour décider quelle est la « bonne » interprétation.

Opérations sur les mots

Deux mots $u = a_1\dots a_n$ et $v = b_1\dots b_p$ sur un même alphabet A peuvent être juxtaposés pour former le mot $uv = a_1\dots a_nb_1\dots b_p$ appelé *concaténation* de u et v. En identifiant les lettres de A et les mots à une lettre, l'écriture $a_1\dots a_n$ peut être interprétée comme le mot constitué des lettres indiquées ou comme le produit de concaténation des mots à une lettre indiqués. Ces deux interprétations définissent clairement le même mot. On a les propriétés :

$|uv| = |u| + |v|$;

ε est élément neutre à droite et à gauche pour la concaténation ;

la concaténation est associative mais non commutative, sauf si A ne contient qu'une lettre ;

$(uv = uw \Longrightarrow v = w)$ et $(vu = wu \Longrightarrow v = w)$.

Pour $u \in A^*$ et $n \in \mathbf{N}^*$, on note $u^0 = \varepsilon$ et $u^n = \underbrace{u\,u \dots u}_{n \text{ facteurs}}$.

Si u et v sont des mots sur A, on dit que u est un *facteur* de v s'il existe deux mots v', v'', éventuellement vides, tels que $v = v'uv''$. Lorsque $v' = \varepsilon$ on dit que u est un *préfixe* de v, et lorsque $v'' = \varepsilon$ on dit que u est un *suffixe* de v.

Si $u = a_1 \dots a_n$ où les a_i sont des lettres, l'image miroir de u est $\tilde{u} = a_n \dots a_1$.

10-2 Opérations sur les langages

Soient L, L' deux langages sur un alphabet A. On définit les langages suivants :

$L + L'$: union ensembliste de L et L'.

$L \cap L'$: intersection de L et L'.

$\overline{L} = \{u \in A^* \text{ tq } u \notin L\}$.

$LL' = \{uv \text{ tq } u \in L \text{ et } v \in L'\}$: LL' est l'ensemble de tous les produits de concaténation d'un mot de L avec un mot de L'. LL' est vide si et seulement si l'un des langages L ou L' est vide.

$L^* = \{\varepsilon\} + L + LL + LLL + \dots = \bigcup_{n \in \mathbf{N}} L^n$. L^* est l'ensemble des produits d'un nombre fini de mots de L. Il est toujours non vide, même si L est vide ($\varnothing^* = \{\varepsilon\}$). La notation A^* pour l'ensemble de tous les mots sur A est compatible avec cette notation si l'on considère A comme un ensemble de mots à une lettre.

$L^+ = \bigcup_{n \geqslant 1} L^n$. $L^+ = L^*$ si et seulement si L contient le mot vide.

$\tilde{L} = \{\tilde{u} \text{ tq } u \in L\}$.

Les propriétés suivantes sont immédiates :

$L^+ = LL^* = L^*L$;

$L^* = L^+ + \{\varepsilon\}$;

$(L^*)^* = L^*$;

$L(M + N) = LM + LN$;

$(M + N)L = ML + NL$;

$(L + M)^* = (L^*M)^*L^* = L^*(ML^*)^*$ (règle de Lazard).

L'intersection et le complémentaire n'ont par contre pas de « bonne » propriété vis-à-vis de la concaténation : $\overline{L}^* \neq \overline{L^*}$ (ε appartient au premier et pas au second) et $L(M \cap N) \neq (LM) \cap (LN)$ (par exemple $L = A^*$, M et N les sous-ensembles des mots de longueur paire, de longueur impaire).

Expressions régulières

Une *expression régulière* sur l'alphabet A est une formule de longueur finie correctement parenthésée utilisant uniquement les opérations somme, concaténation et étoile, le langage vide $\varnothing$ et les langages réduits à un mot : $\{u\}$ où u est un mot quelconque sur A. Pour alléger les notations, on écrira u pour le langage $\{u\}$. Un langage L est dit *régulier* s'il peut être décrit par une expression régulière. Un langage régulier est aussi dit *rationnel* pour des raisons liées à la théorie des séries formelles.

Exemples :

– Sur un alphabet A, le langage L constitué des mots de longueur paire : $L = \{u \text{ tq } |u| \in 2\mathbf{N}\}$ est régulier car il est décrit par l'expression $L = (AA)^*$ (ici A est une abréviation pour la somme de toutes les lettres, ce qui constitue une expression régulière puisque A est fini).

– Sur l'alphabet $A = \{0, 1\}$ le langage L des mots commençant par 1 est décrit par l'expression régulière $L = 1(0 + 1)^*$, il est donc régulier. Sur le même alphabet, le langage L' constitué des mots contenant exactement trois fois la lettre 1 est décrit par l'expression régulière $L' = 0^*10^*10^*10^*$.

– Sur l'alphabet $A = \{a, \dots, z\}$ le langage des mots contenant comme facteur l'un des mots : `facteur` ou `factrice` est défini par l'expression régulière : $L = A^*\mathtt{fact}(\mathtt{eur} + \mathtt{rice})A^*$. Les expressions régulières :

$$(A^*\mathtt{facteur}A^*) + (A^*\mathtt{factrice}A^*)$$

$$A^*(\mathtt{facteur})^*(\mathtt{facteur} + \mathtt{factrice})(\mathtt{factrice})^*A^*$$

définissent le même langage, ce qui montre qu'il n'y a pas unicité d'une expression régulière définissant un langage régulier donné.

– Le langage des parenthèses emboîtées est défini par $L = \{a^n b^n \text{ tq } n \in \mathbf{N}\}$ où $A = \{a, b\}$, a désignant une parenthèse ouvrante et b une parenthèse fermante. Ce langage n'est pas régulier, voir une démonstration à la section **11-6** et une autre démonstration dans l'exercice **10-8**.

Par définition même, la somme, le produit de deux langages réguliers et l'étoile d'un langage régulier sont encore réguliers, mais il n'est pas évident que l'intersection de deux langages réguliers ou le complémentaire d'un langage régulier le soient. On verra au chapitre suivant que c'est cependant le cas. Par ailleurs, tout langage fini est régulier puisqu'il est la somme des mots qu'il contient. Enfin, si L est un langage régulier, alors $\tilde{L}$ l'est aussi : on obtient une expression régulière pour $\tilde{L}$ en permutant récursivement les opérandes des concaténations et en remplaçant les mots par leurs images miroir dans une expression régulière pour L

10-3 Appartenance d'un mot à un langage régulier

Soit L un langage régulier sur un alphabet A et $u \in A^*$. On veut déterminer par programme si $u \in L$. On suppose que L est défini par une expression régulière représentée par un arbre dont les nœuds internes sont étiquetés par les opérateurs somme, concaténation et étoile, et dont les feuilles sont étiquetées par le langage vide ou par un mot sur A, un mot étant représenté par un vecteur de lettres.

```
type 'a exp = (* 'a est le type des lettres *)
| Vide                  (* langage ∅       *)
| Mot    of ('a vect)   (* langage à un mot *)
| Somme  of ('a exp) * ('a exp)
| Concat of ('a exp) * ('a exp)
| Etoile of ('a exp)
;;

(* dit si le mot u appartient au langage régulier défini par exp *)
let rec appartient(exp,u) = match exp with
| Vide          -> false
| Mot(m)        -> u = m
| Somme(e1,e2)  -> appartient(e1,u) or appartient(e2,u)
| Concat(e1,e2) -> découpe(e1,e2,u,0)
| Etoile(e)     -> u = [||] or découpe(e,exp,u,1)

(* dit si l'on peut découper le mot u après la position i de    *)
(* sorte que la partie gauche est dans e1 et la droite dans e2 *)
and découpe(e1,e2,u,i) =
  if i > vect_length(u) then false
  else let v = sub_vect u 0 i
       and w = sub_vect u i (vect_length(u) - i) in
       (appartient(e1,v) & appartient(e2,w)) or découpe(e1,e2,u,i+1)
;;
```

`découpe` essaie récursivement tous les découpages possibles de `u` en deux facteurs jusqu'à trouver un découpage élément de E_1E_2 où E_i est le langage défini par e_i. Remarquer que l'expression booléenne :

```
(appartient(e1,v) & appartient(e2,w)) or découpe(e1,e2,u,i+1)
```

est un raccourci pour :

```
if appartient(e1,v)
then if appartient(e2,w) then true else découpe(e1,e2,u,i+1)
else découpe(e1,e2,u,i+1)
```

c'est-à-dire que `appartient(e2,w)` n'est pas évalué si `appartient(e1,v)` retourne `false`, et `découpe(e1,e2,u,i+1)` n'est évalué que si l'un des deux appels à `appartient` a retourné `false`. On remarque que chaque appel récursif à `appartient` porte sur une expression plus petite (en nombre de nœuds) et un mot plus petit, l'une des deux inégalités étant toujours stricte car on traite à part l'appartenance du mot vide à une expression E^*. Donc `appartient` termine et on démontre par récurrence sur $\max(|\exp|, |u|)$ que le booléen retourné est correct.

Il ne s'agit certainement pas d'un algorithme efficace : par exemple pour constater que le mot `aab` n'appartient pas au langage défini par l'expression $(\mathtt{a}+\mathtt{b})^*\mathtt{c}$, on essaie successivement :

$\varepsilon \in (\mathtt{a}+\mathtt{b})^*$ et $\mathtt{aab} \in \mathtt{c}$?
$\mathtt{a} \in (\mathtt{a}+\mathtt{b})$ et $\varepsilon \in (\mathtt{a}+\mathtt{b})^*$ et $\mathtt{ab} \in \mathtt{c}$?
$\mathtt{a} \in (\mathtt{a}+\mathtt{b})$ et $\mathtt{a} \in (\mathtt{a}+\mathtt{b})$ et $\varepsilon \in (\mathtt{a}+\mathtt{b})^*$ et $\mathtt{b} \in \mathtt{c}$?
$\mathtt{a} \in (\mathtt{a}+\mathtt{b})$ et $\mathtt{a} \in (\mathtt{a}+\mathtt{b})$ et $\mathtt{b} \in (\mathtt{a}+\mathtt{b})$ et $\varepsilon \in (\mathtt{a}+\mathtt{b})^*$ et $\varepsilon \in \mathtt{c}$?

Il y a donc répétition des mêmes tests, répétition due aux « points de découpage » multiples pour reconnaître une concaténation et pour reconnaître une étoile. Plus précisément, si l'expression régulière est la concaténation de k expressions simples (mots ou sommes de mots), alors on teste dans le pire des cas toutes les possibilités de placer k points de découpage entre les lettres de u, soit pour un mot de longueur n, C_{n+1}^k essais. Compte tenu des court-circuits booléens effectués par le compilateur CAML, il n'est pas évident que tous ces essais soient effectués, mais cela se produit certainement si chaque essai échoue lors du dernier test compilé. A k fixé la complexité dans le pire des cas de la reconnaissance pour un mot de longueur n à un langage de la forme $L_1 \dots L_k$ croît au moins aussi vite que n^k.

Si l'expression régulière est de la forme L^*, alors on teste dans le pire des cas tous les découpages de u en facteurs de longueur supérieure ou égale à 1, donc tous les sous-ensembles de positions distinctes entre les lettres de u et il y a 2^n tels sous-ensembles. Dans le cas général, la complexité de `appartient` pour une expression régulière donnée comportant au moins une étoile et pour un mot arbitraire de longueur n est au moins exponentielle.

10-4 Exercices

Exercice 10-1 : mots commutants

1. Lemme de LEVI : soient $u, v, x, y \in A^*$ tels que $uv = xy$. Montrer qu'il existe $z \in A^*$ tel que ($u = xz$ et $y = zv$) ou ($x = uz$ et $v = zy$).
2. Soient $x, y \in A^*$. Montrer que x et y commutent si et seulement s'ils sont puissance d'un même mot.

Exercice 10-2 : expression régulière

Définir par une expression régulière le langage des mots sur $\{a, \dots, z\}$ contenant les lettres `a`, `v`, `i`, `o`, `n` dans cet ordre (non nécessairement consécutives). Définir par une expression régulière le langage des mots ne contenant pas les lettres `a,v,i,o,n` dans cet ordre.

Exercice 10-3 : expression régulière
Définir par une expression régulière le langage des mots sur $\{0, \dots, 9\}$ ne contenant pas le facteur 13. On prouvera que cette expression est correcte.

Exercice 10-4 : expression régulière

1. Donner une expression régulière définissant le langage des suites croissantes sur l'alphabet $\{0, \dots, 9\}$.
2. Donner une expression régulière définissant le langage des suites arithmétiques sur l'alphabet $\{0, \dots, 9\}$.

Exercice 10-5 : expression régulière
En TURBO-PASCAL, un commentaire est un texte placé entre les caractères `(*` et `*)` ou un texte placé entre les caractères `{` et `}`. Donner une expression régulière définissant le langage de ces commentaires.

Exercice 10-6 : langage défini par un système d'équations.
Soient K, L deux langages sur un alphabet A tels que K ne contient pas le mot vide.

1. Montrer qu'il existe un unique langage X sur A solution de l'équation : $X = KX + L$, et décrire explicitement X.

2. De même, si K, K', L, L', M, M' sont des langages sur A tels que K, K', L, L' ne contiennent pas le mot vide, montrer que le système :

$$\begin{cases} X = KX + LY + M \\ Y = K'X + L'Y + M' \end{cases}$$

admet une solution unique (que l'on ne demande pas d'expliciter), et que si K, K', L, L', M, M' sont réguliers, alors X et Y sont aussi réguliers.

Exercice 10-7 : expression sans $\varnothing$ ni ε
Soit L un langage régulier non vide et ne contenant pas le mot vide. Montrer qu'il existe une expression régulière définissant L ne faisant intervenir ni $\varnothing$ ni ε.

Exercice 10-8 : langage non régulier
Soient $L = \{a^n b^n \text{ tq } n \in \mathbf{N}\}$ le langage des parenthèses emboîtées et pour $p \in \mathbf{N}$, $L_p = Lb^p$ et $L_{-p} = a^p L$. Montrer que tout langage régulier inclus dans un L_p $(p \in \mathbf{Z})$ est fini. Ceci démontre que L n'est pas régulier.

Chapitre 11

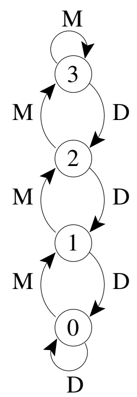
Automates finis

11-1 Définitions

Un *automate fini* est un dispositif physique ou abstrait pouvant se trouver dans un nombre fini d'*états* et susceptible de changer d'état en réaction à un événement extérieur. Mathématiquement, un automate est défini par la donnée :

– de l'ensemble des évènements susceptibles d'influer sur l'automate. Cet ensemble est supposé fini non vide, donc constitue un alphabet appelé *alphabet d'entrée*.
– de l'ensemble des états de l'automate. C'est également un ensemble fini non vide.
– d'une *fonction de transition* qui, pour chaque état et chaque lettre de l'alphabet d'entrée indique les états possibles vers lesquels l'automate peut évoluer. Cette fonction peut aussi spécifier des transitions « spontanées » (l'automate évolue sans événement extérieur) appelées *transitions vides* ou *ε-transitions*. la fonction de transition est aussi appelée *programme* de l'automate.

On représente généralement un automate par un *graphe de transition* dont les nœuds sont les états de l'automate et les flèches sont les transitions autorisées, étiquetées par la ou les lettres correspondant à ces transitions. Par exemple la figure ci-contre présente un automate d'ascenseur simplifié dont les états sont les étages où peut se trouver la cabine et l'alphabet d'entrée est constitué des lettres `M` (montée) et `D` (descente) correspondant aux boutons d'appel de la cabine.



automate d'ascenseur

Un automate est qualifié de *complet* si pour chaque état il est défini au moins une transition par lettre d'entrée, et de *déterministe* s'il n'a pas de ε-transitions et si pour chaque état il est défini au plus une transition par lettre d'entrée. Si un automate incomplet reçoit une lettre pour laquelle il n'y a pas de transition à partir de l'état courant alors il se *bloque* et cesse de fonctionner. Si

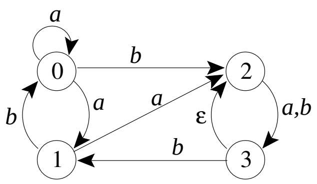


figure 26 : automate non déterministe

un automate non déterministe se trouve dans un état d'où part une ε-transition alors il peut changer d'état selon cette transition sans attendre l'arrivée d'une lettre d'entrée. S'il reçoit une lettre pour laquelle plusieurs transitions sont définies alors il peut suivre n'importe laquelle de ces transitions. La figure 26 présente un automate incomplet et non déterministe. Si cet automate est placé dans l'état 0 et reçoit les lettres a, b, a, a dans cet ordre, alors il peut suivre l'un des chemins :

$$0 \xrightarrow{a} 0 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{a} \text{bloqué}$$
$$0 \xrightarrow{a} 0 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{\varepsilon} 2 \xrightarrow{a} 3$$
$$0 \xrightarrow{a} 1 \xrightarrow{b} 0 \xrightarrow{a} 0 \xrightarrow{a} 0$$
$$0 \xrightarrow{a} 1 \xrightarrow{b} 0 \xrightarrow{a} 0 \xrightarrow{a} 1$$
$$0 \xrightarrow{a} 1 \xrightarrow{b} 0 \xrightarrow{a} 1 \xrightarrow{a} 2$$

Au contraire, l'automate d'ascenseur est complet et déterministe, donc il ne peut jamais se bloquer et partant d'un état donné il ne peut suivre qu'un seul chemin pour une suite de lettres donnée.

Dans la vie quotidienne, on utilise des automates pour piloter des appareils physiques ou de manière générale effectuer des tâches *automatiques*, l'automate d'ascenseur relève de ce type d'utilisation (dans les ascenseurs modernes il n'est pas nécessaire d'appuyer sur un bouton à chaque étage, l'exercice **11-4** propose l'étude d'un modèle plus réaliste). Un ordinateur peut être considéré comme un automate ayant un nombre fini d'états, toutes les configurations possibles de sa mémoire, et changeant d'état lorsque l'on frappe une touche sur un clavier. Les ordinateurs réels disposent de plusieurs organes d'entrée que l'on peut assimiler à un ensemble de claviers, mais aussi d'une horloge permettant de dater les évènements reçus. Cette notion de temps n'est pas prise en compte dans le modèle mathématique, elle pourrait l'être en considérant les tops de l'horloge comme des frappes sur une touche particulière.

Soit $\mathcal{A}$ un automate d'alphabet d'entrée A, $u = a_1 \ldots a_n \in A^*$ et x, y deux états de $\mathcal{A}$. On dit que u fait passer $\mathcal{A}$ de l'état x à l'état y si x et y sont reliés par une chaîne de transitions correspondant aux lettres $a_1, \ldots, a_n$ et au mot vide ε, les lettres figurant dans le même ordre que dans u. On dit qu'il existe une *transition généralisée*, notée $x \overset{u}{\twoheadrightarrow} y$, d'étiquette u menant de x à y. Remarquer que si $\mathcal{A}$ est un automate complet et déterministe alors pour x et u donnés il existe un unique état y tel que $x \overset{u}{\twoheadrightarrow} y$. Le langage menant de x à y est l'ensemble des mots u faisant passer $\mathcal{A}$ de x à y, il est noté $\mathcal{L}_{\mathcal{A}}(x, y)$.

Un automate *reconnaisseur* est un automate dans lequel on choisit un état q_0 appelé *état initial* et un ensemble F d'états appelés *états finals* ou *états* *acceptants* : *$\mathcal{A}$ reconnaît* un mot u s'il existe un état final $f \in F$ tel que $q_0 \xrightarrow[*]{u} f$. Le langage reconnu par $\mathcal{A}$ est le langage menant de l'état initial à un état final quelconque. Dans le cas d'un automate non déterministe, un mot est reconnu s'il existe au moins une transition généralisée indexée par u de q_0 vers F, même si d'autres transitions généralisées peuvent conduire de q_0 vers un état non final ou bloquer l'automate. Deux automates reconnaissant le même langage sont dits *équivalents*. Dans la représentation graphique d'un automate reconnaisseur, on indique par une flèche entrante l'état initial et par une flèche sortante ou un cercle double chaque état final. Par exemple les deux automates de la figure 27 sur l'alphabet $\{a, b, c\}$ reconnaissent le langage $L = a^*bc^*$; le premier est déterministe, le deuxième non.

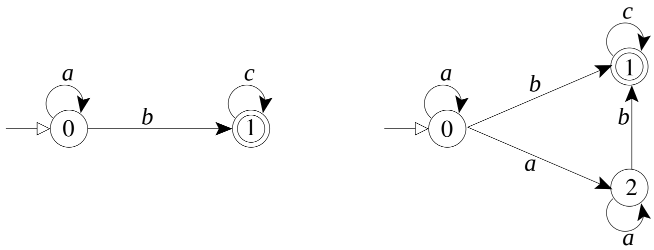


figure 27 : automates reconnaissant le langage a^*bc^*

L'intérêt des automates reconnaisseurs, et surtout des automates reconnaisseurs déterministes, est qu'ils permettent de savoir facilement si un mot donné appartient au langage de l'automate : on place l'automate dans l'état initial et on lui transmet une à une les lettres de u. Si l'automate arrive dans un état final alors le mot appartient au langage ; sinon, et si toutes les possibilités de transition ont été essayées dans le cas d'un automate non déterministe, alors le mot n'appartient pas au langage. Une autre application des automates reconnaisseurs est de permettre le découpage d'un mot en facteurs appartenant au langage de l'automate : à chaque fois que l'automate atteint un état final, il *émet* le sous-mot reconnu et est replacé dans l'état initial pour analyser la suite du mot d'entrée. Les analyseurs lexicaux procèdent généralement de cette manière. Par exemple l'automate de la figure 28 reconnaît les expressions arithmétiques non parenthésées conformes à la syntaxe de CAML. L'état final 1 correspond aux constantes entières, les états finals 4 et 8 correspondent aux constantes flottantes, l'état final 2 aux opérations entières, l'état final 5 aux opérations flottantes et l'état final 3 aux identificateurs écrits en minuscules. Bien que cet automate soit déterministe, il y a ambiguïté sur la manière de découper un mot en lexèmes, par exemple le mot `12.3E+07` peut être découpé en `1`, `2.3E+0`, `7`. Ce découpage ne correspond pas à la syntaxe usuelle des expressions arithmétiques, mais cette syntaxe ne figure pas dans la description de l'automate. Généralement les analyseurs lexicaux choisissent en cas d'ambiguïté le sous-mot le plus long reconnaissable, et dans le cas d'un automate non déterministe, s'il y a plusieurs états finals correspondant à ce sous-mot, ils choisissent entre ces états selon une règle de priorité inscrite dans l'analyseur.

Enfin un automate peut aussi servir à chercher un mot dans un texte. L'automate de la figure 29 reconnaît les textes contenant le mot *ennemi*. Si on lui

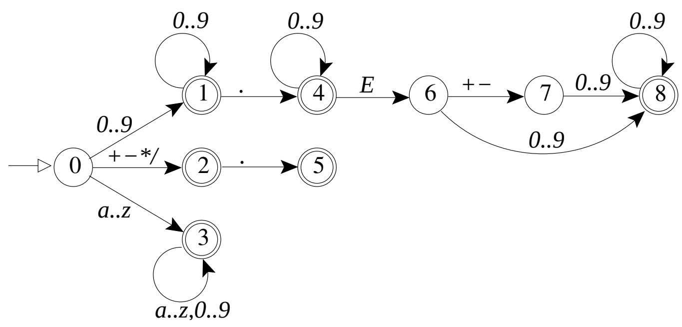


figure 28 : automate reconnaissant les expressions arithmétiques simplifiées

soumet un tel texte alors il aboutit à l'état final n°6 sur la première occurrence du mot *ennemi* et reste dans cet état pour lire le reste du texte. A partir d'un état, les transitions étiquetées _ correspondent à toutes les lettres autres que celles explicitement indiquées comme partant de l'état considéré.

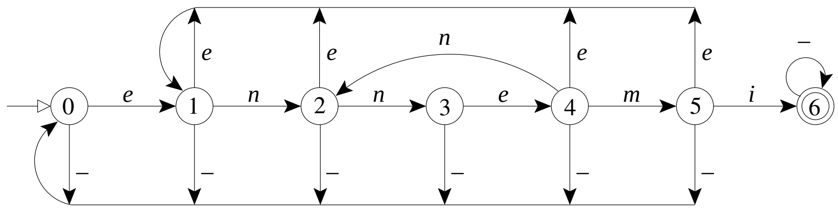


figure 29 : automate reconnaissant la première occurrence de ennemi

11-2 Simulation d'un automate fini.

On peut simuler le fonctionnement d'un automate sur un ordinateur, il suffit de conserver en mémoire les états et transitions de l'automate et de suivre ces transitions en parcourant un mot à analyser. Les problèmes algorithmiques liés à cette simulation sont de choisir sous quelle forme on mémorise les transitions, et de quelle manière on gère le non-déterminisme.

Simulation d'un automate déterministe

Dans le cas d'un automate déterministe il suffit pour connaître l'automate de spécifier l'état initial, les états finals et la fonction de transition qui indique à partir d'un état et d'une lettre vers quel état l'automate doit évoluer. En pratique, si les états et les lettres sont des nombres entiers alors on peut définir la fonction de transition par une *matrice de transition* `M` telle `M.(e).(l)` contient l'état accessible à partir de l'état `e` et de la lettre `l`, et un entier particulier, par exemple -1, pour les transitions non définies. On peut généralement se ramener facilement à ce cas puisque l'ensemble des états et l'alphabet d'entrée sont finis, donc numérotables.

Par exemple l'automate « ennemi » de la figure 29 est représenté par la matrice de transition :

	e	n	m	i	_
0	1	0	0	0	0
1	1	2	0	0	0
2	1	3	0	0	0
3	4	0	0	0	0
4	1	2	5	0	0
5	1	0	0	6	0
6	−1	−1	−1	−1	−1

On a comprimé la matrice en notant _ une lettre quelconque autre que e, n, m, i. Sans cette compression ou pour un automate « surveillant » un plus grand nombre de lettres, on obtiendrait une matrice énorme contenant essentiellement des transitions indéfinies (-1) ou menant à l'état initial (0). Une méthode de stockage plus élégante qui n'impose pas le codage des états et des lettres par des entiers consiste à écrire une fonction de transition directement en CAML. La fonction de transition de « ennemi » peut être codée de la manière suivante :

```
(* e = état, l = lettre *)
let transition(e,l) = match (e,l) with
| (0,`e`) -> 1 | (0,_)   -> 0
| (1,`e`) -> 1 | (1,`n`) -> 2 | (1,_)   -> 0
| (2,`e`) -> 1 | (2,`n`) -> 3 | (2,_)   -> 0
| (3,`e`) -> 4 | (3,_)   -> 0
| (4,`e`) -> 1 | (4,`n`) -> 2 | (4,`m`) -> 5 | (4,_) -> 0
| (5,`e`) -> 1 | (5,`i`) -> 6 | (5,_)   -> 0
| (6,_)   -> 6
| _       -> failwith "état inconnu"
;;
```

De cette manière on n'indique que les transitions existantes, et on peut éventuellement regrouper plusieurs cas identiques à l'aide de tests appropriés. L'inconvénient de la méthode « fonction » par rapport à la méthode « matrice » est que le temps de calcul d'une transition est plus long puisqu'il faut examiner tous les cas successivement avant de trouver celui qui s'applique. On peut aussi mélanger les deux méthodes en écrivant une fonction qui élimine rapidement les cas simples et consulte une ou plusieurs tables dans les autres cas. On utilisera ici la méthode « fonction » pour stocker l'automate.

```
(* 'a est le type des états, 'b est celui des lettres *)
type ('a,'b) automate = {
  départ : 'a;                 (* état initial            *)
  final  : 'a -> bool;         (* dit si un état est final *)
  transition : ('a * 'b) -> 'a (* fonction de transition   *)
};;
```

```
(* a est un automate déterministe, u un mot à analyser *)
(* dit si u appartient au langage de a.                *)

let analyse(a,u) = try
  let e = ref(a.départ) in
  for i = 0 to vect_length(u)-1 do
    e := a.transition(!e,u.(i))
  done;
  a.final(!e)
with Failure "transition indéfinie" -> false
;;
```

La correction de `analyse` est immédiate, et sa complexité pour un automate `a` donné est proportionnelle à la longueur de u sous réserve que la fonction `a.transition` s'exécute en un temps constant.

On peut aussi représenter un automate par un ensemble de fonctions mutuellement récursives où chaque appel récursif correspond à une transition :

```
let rec état_0(u) = match u with
| `e` :: v -> état_1(v)
| _   :: v -> état_0(v)
| _        -> false

and état_1(u) = match u with
| `e` :: v -> état_1(v)
| `n` :: v -> état_2(v)
| _   :: v -> état_0(v)
| _        -> false

and état_2(u) = match u with
| `e` :: v -> état_1(v)
| `n` :: v -> état_3(v)
| _   :: v -> état_0(v)
| _        -> false

and état_3(u) = match u with
| `e` :: v -> état_4(v)
| _   :: v -> état_0(v)
| _        -> false

and état_4(u) = match u with
| `e` :: v -> état_1(v)
| `n` :: v -> état_2(v)
| `m` :: v -> état_5(v)
| _   :: v -> état_0(v)
| _        -> false

and état_5(u) = match u with
| `e` :: v -> état_1(v)
| `i` :: v -> état_6(v)
| _   :: v -> état_0(v)
| _        -> false

and état_6(u) = true
;;
```

Un mot u (représenté ici par une liste chaînée de lettres) est reconnu si et seulement si l'appel `état_0(u)` retourne `true`. Ce codage est moins compact que la fonction de transition définie plus haut, mais chaque fonction de transition s'exécute plus rapidement puisqu'elle n'a à examiner que la première lettre du mot proposé et non le couple *(état, lettre)*. Tous les appels récursifs sont terminaux, donc l'analyse d'un mot peut être compilée de manière à s'exécuter en mémoire constante. Le logiciel `camllex` est un générateur de programmes : à partir d'une expression régulière fournie en entrée, il construit un automate déterministe reconnaissant le langage associé et *rédige* un programme CAML simulant cet automate par la technique des fonctions mutuellement récursives.

Simulation d'un automate non déterministe

Un automate non déterministe peut être représenté par une fonction de transition retournant pour un état et une lettre donnés la liste des états accessibles, et une deuxième fonction retournant la liste des états accessibles à partir d'un état donné par une ε-transition. Dans la mesure où l'automate est constant on peut calculer, lors de la construction de l'automate, pour chaque état la liste de tous les états accessibles par un nombre quelconque de ε-transitions et adapter en conséquence la fonction de transition sur lettre. Il est alors plus commode de considérer que l'automate dispose de plusieurs états initiaux, exactement tous les états accessibles à partir de l'état initial « officiel » par des ε-transitions. On est donc ramené au problème suivant :

> *étant donnés un automate non déterministe sans ε-transitions, un ensemble* I *d'états initiaux, un ensemble* F *d'états finals et un mot* u, *déterminer si l'on peut passer d'un état de* I *à un état de* F *en suivant les transitions correspondant aux lettres de* u.

Ce problème s'apparente à un *parcours de graphe* où l'on cherche un chemin menant d'une source à un but dans un graphe orienté à priori quelconque. Il existe essentiellement deux solutions à ce problème, le *parcours en profondeur d'abord* du graphe : on suit un chemin aussi loin que possible, puis on revient récursivement sur les choix effectués en cas d'impasse ; et le *parcours en largeur d'abord* : on explore en parallèle tous les chemins possibles. La méthode du parcours en profondeur d'abord s'apparente à l'algorithme présenté à la section **10-3** pour la reconnaissance d'une expression régulière. L'algorithme de parcours en largeur est simple : déterminer tous les successeurs autorisés de tous les états de I compte tenu de la première lettre de u, et appliquer récursivement l'algorithme avec cet ensemble de successeurs et la suite de u.

```
(* 'a est le type des états, 'b est celui des lettres *)
type ('a,'b) automate_nd = {
  départ : 'a list;               (* états initiaux     *)
  final  : 'a -> bool;            (* état final ?       *)
  transition : ('a * 'b) -> 'a list (* fct. de transition *)
};;


(* cherche tous les successeurs d'une liste d'états *)
let rec successeurs(a,liste,lettre) = match liste with
| [] -> []
| x::suite -> union (a.transition(x,lettre))
                    (successeurs(a,suite,lettre))
;;
```

```
(* a est un automate non déterministe, u un mot à analyser *)
(* dit si u appartient au langage de a.                    *)
let analyse_nd(a,u) =
  let l = ref(a.départ) in
  for i = 0 to vect_length(u)-1 do
    l := successeurs(a,!l,u.(i))
  done;
  exists a.final !l
;;
```

union et exists sont des fonctions de la bibliothèque standard de CAML réalisant l'union sans répétition de deux listes et cherchant si une liste contient un élément satisfaisant une propriété. La fonction de transition de l'automate doit renvoyer une liste de successeurs dans tous les cas, éventuellement une liste vide pour une transition indéfinie, car si l'automate se bloque sur un état il faut pouvoir continuer l'analyse avec les autres états stockés dans !l. Par exemple l'automate de la figure 27 est défini par :

```
let a = {
  départ = [0];
  final  = (fun x -> x = 1);
  transition = (function (0,`a`) -> [0;2]
                       | (0,`b`) -> [1]
                       | (1,`c`) -> [1]
                       | (2,`a`) -> [2]
                       | (2,`b`) -> [1]
                       | _       -> [])
};;
```

Complexité : chaque lettre de u donne lieu à un appel à la fonction successeurs sur la liste des états atteints, et cette liste est de taille bornée puisque l'automate est fini et la fonction union réalise une union sans répétition. La complexité temporelle de analyse_nd est donc linéaire par rapport à la taille de u, et il en est de même de la complexité spatiale puisqu'on crée pour chaque lettre de u une nouvelle liste !l de taille bornée par le nombre d'états de l'automate. Comme l'ancienne liste n'est plus utilisée, la mémoire qu'elle occupe peut être « recyclée » et dans ce cas la complexité spatiale de analyse_nd est constante.

Ainsi, aussi bien avec un automate déterministe qu'avec un automate non déterministe, on peut reconnaître l'appartenance d'un mot de longueur n à un langage donné en $O(n)$ opérations, ce qui améliore nettement la complexité obtenue par parcours d'une expression régulière (cf. section **10-3**). A priori les langages reconnus ne sont pas de même type, l'algorithme de la section **10-3** ne reconnaît que l'appartenance à un langage régulier tandis que les deux versions de analyse traitent de langages dits *reconnaissables*, c'est-à-dire reconnaissables par un automate fini. On verra qu'en fait un langage est régulier si et seulement s'il est reconnaissable (théorème de Kleene) donc la comparaison des algorithmes de reconnaissance est pertinente.

11-3 Déterminisation d'un automate

Les fonctions `analyse` et `analyse_nd` vues à la section précédente sont très semblables : on part d'un état initial et on applique la fonction de transition à cet état pour chaque lettre du mot u à analyser, ou on part d'une liste d'états initiaux et on applique la fonction `successeurs` à cette liste pour chaque lettre de u. Cette similitude suggère qu'un automate fini non déterministe $\mathcal{A}$ est formellement équivalent à un automate déterministe $\mathcal{A}'$ construit de la manière suivante :

- les états de $\mathcal{A}'$ sont les sous-ensembles d'états de $\mathcal{A}$ (si $\mathcal{A}$ a n états alors $\mathcal{A}'$ a 2^n états) ;
- pour un ensemble E d'états de $\mathcal{A}$ et une lettre a, $\mathcal{A}'$ définit une unique transition partant de E étiquetée par a, arrivant sur l'ensemble E' constitué de tous les états de $\mathcal{A}$ accessibles à partir d'un état élément de E par une transition généralisée d'étiquette a ;
- l'état initial de $\mathcal{A}'$ est l'ensemble de tous les états de $\mathcal{A}$ accessibles à partir des états initiaux de $\mathcal{A}$ par des ε-transitions ;
- un état E de $\mathcal{A}'$ est final si et seulement s'il contient un état final de $\mathcal{A}$.

Toutes ces règles peuvent être appliquées de manière automatique puisque $\mathcal{A}$ a un nombre fini d'états, c'est-à-dire qu'il est possible de « construire » l'automate $\mathcal{A}'$ à partir d'une description convenable de $\mathcal{A}$. Les détails de cette construction font l'objet de l'exercice **11-5**.

Complexité : partant d'un automate $\mathcal{A}$ à n états, ...

- On construit les 2^n états de $\mathcal{A}'$ ce qui nécessite $O(n2^n)$ opérations (en supposant qu'un état de $\mathcal{A}'$ est représenté par une liste d'états de $\mathcal{A}$ ou un vecteur de booléens, il faut $O(n)$ opérations pour créer cette liste ou ce vecteur).

- On cherche ensuite pour chaque état de $\mathcal{A}$ tous les états accessibles par ε-transitions, ce qui peut être fait en $O(n^3)$ opérations (algorithme de WARSHALL, cf. exercice **11-5** et [FROID] ch. 19).

- Puis, pour chaque état et chaque lettre de l'alphabet d'entrée, on détermine les états accessibles depuis cet état en suivant cette lettre et les ε-transitions. Il y a pour un état et une lettre donnés au maximum n transitions sur lettre à considérer et donc n sous-ensembles de taille inférieure ou égale à n à réunir ; au total, l'alphabet d'entrée étant de taille fixée, il faut effectuer $O(n^3)$ opérations.

- Enfin on détermine les transitions des états de $\mathcal{A}'$ par réunion des ensembles précédents, ce qui nécessite $O(n^2 2^n)$ opérations.

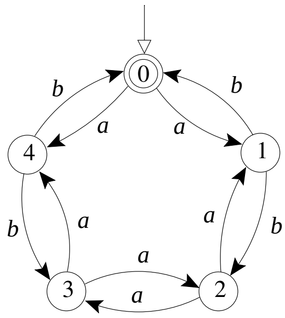
La complexité totale de l'algorithme de déterminisation est donc $O(n^2 2^n)$ *pour un automate non déterministe à* n *états.*

Lorsque n est petit cet algorithme peut être exécuté en un temps raisonnable, mais si n dépasse quelques dizaines il est impraticable. Il y a deux solutions à ce problème :

1. Conserver l'automate non déterministe : l'algorithme de reconnaissance n'est plus lent que d'un facteur constant entre un automate non déterministe et un automate déterministe équivalent ; si l'on a peu de mots à tester alors le gain à attendre de la déterminisation ne vaut pas l'effort qu'il faut fournir.

2. Appliquer l'algorithme de déterminisation en ne générant que les états utiles, c'est-à-dire les sous-ensembles d'états de $\mathcal{A}$ effectivement obtenus à partir de l'état initial en suivant toutes les ε-transitions et les transitions sur lettre possibles. Si l'automate n'est pas trop compliqué, on peut espérer obtenir un automate déterministe n'ayant pas trop d'états en $O(n^3 + n^2 N)$ opérations où N est le nombre d'états de l'automate déterministe effectivement construit (cf. exercice **11-5**). Il existe cependant des cas pathologiques où tout automate déterministe équivalent à $\mathcal{A}$ a au moins 2^{n-1} états (cf. section **11-6**).

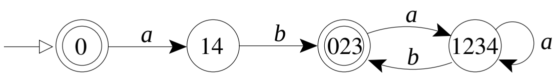
Exemple



automate non déterministe

départ	lettre	arrivée
$\{0\}$	a	$\{1,4\}$
$\{0\}$	b	$\varnothing$
$\{1,4\}$	a	$\varnothing$
$\{1,4\}$	b	$\{0,2,3\}$
$\{0,2,3\}$	a	$\{1,2,3,4\}$
$\{0,2,3\}$	b	$\varnothing$
$\{1,2,3,4\}$	a	$\{1,2,3,4\}$
$\{1,2,3,4\}$	b	$\{0,2,3\}$

table de transition



automate déterministe équivalent

11-4 Le théorème de Kleene

Théorème : *soit* L *un langage sur un alphabet* A*. Il y a équivalence entre :*
1. L *est défini par une expression régulière.*
2. L *est reconnu par un automate fini.*

Démonstration de $1 \Longrightarrow 2$ (construction de Thompson)

L est représenté par une expression régulière $\mathcal{E}$ que l'on assimile à un arbre comme à la section **10-3**. On construit récursivement un automate non déterministe $\mathcal{E}'$ qui simule le parcours de $\mathcal{E}$: à chaque nœud n de $\mathcal{E}$ on associe deux états dans $\mathcal{E}'$, un état d'entrée et un état de sortie (un seul état pour un nœud de type `Etoile`). On place des ε-transitions entre ces états et ceux des fils de n comme indiqué figure 30, et des transitions sur lettre entre les lettres d'une feuille de type `Mot` (ou une ε-transition dans le cas du mot vide). Les feuilles correspondant au langage vide n'ont pas de transition définie. Il est clair, par récurrence sur la taille de $\mathcal{E}$, que $\mathcal{E}'$ reconnaît exactement le langage décrit par l'expression régulière $\mathcal{E}$. ∎

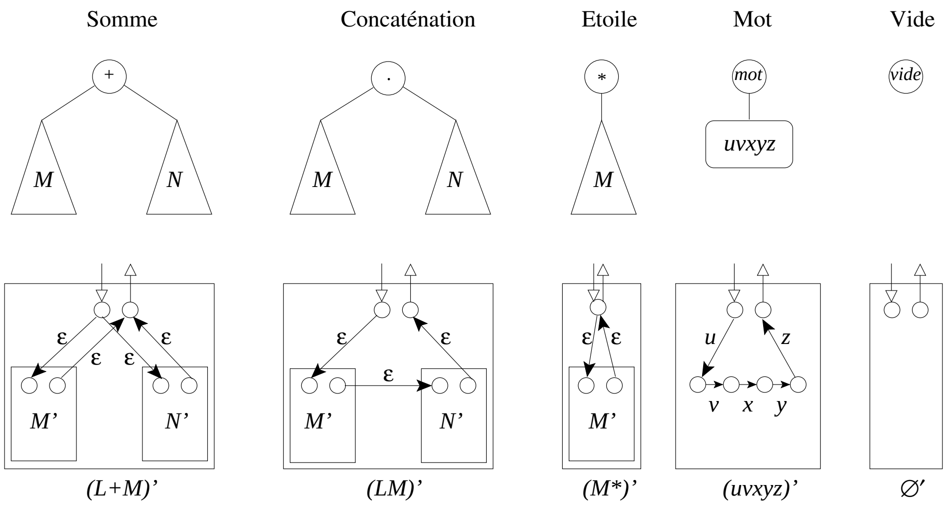


figure 30 : transformation d'une expression régulière en automate

La figure 31 illustre la transformation de l'expression régulière $(a+b)^*(a+c)^*$ en automate reconnaisseur. Dans un premier temps on obtient un automate à 16 états, mais on peut supprimer des états « inutiles » et raccourcir les transitions associées : $2 \xrightarrow{\varepsilon} 4 \xrightarrow{a} 5 \xrightarrow{\varepsilon} 3$ et $2 \xrightarrow{\varepsilon} 6 \xrightarrow{b} 7 \xrightarrow{\varepsilon} 3$ sont rassemblées en une seule transition $2 \xrightarrow{a,b} 3$ et de même pour la branche $(a + c)$, ce qui élimine 8 états. On peut aussi déplacer les points d'entrée et de sortie de l'automate vers le point d'entrée de $(a + b)^*$ et le point de sortie de $(a + c)^*$ et enfin raccourcir les boucles reconnaissant $(a + b)^*$ et $(a + c)^*$ ce qui fournit un automate non déterministe à deux états. Cet automate peut être transformé selon l'algorithme de déterminisation en un automate déterministe à trois états.

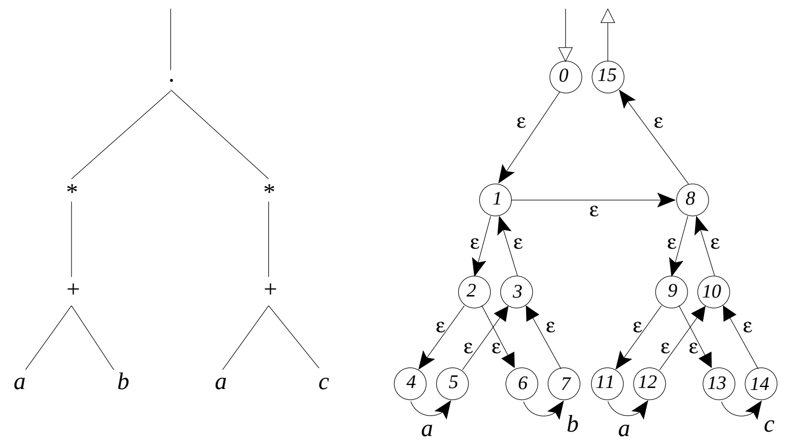


figure 31 : automate reconnaissant l'expression $(a + b)^*(a + c)^*$

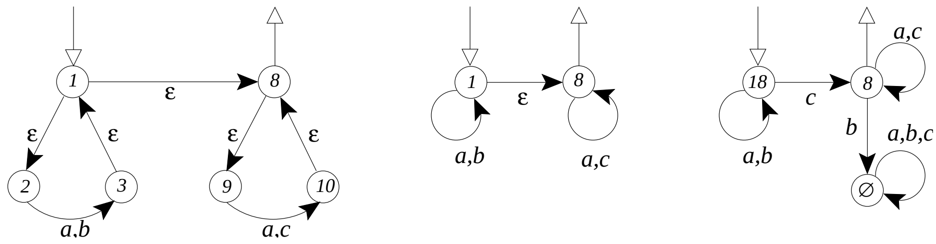


figure 32 : automates simplifiés

Automate indexé par des expressions régulières

Pour démontrer la réciproque (un automate reconnaît un langage régulier), on généralise la notion d'automate en considérant des automates indexés par des expressions régulières sur l'alphabet A. Un tel automate $\mathcal{A}$ est défini par un ensemble d'états fini non vide, et une fonction de transition qui associe à tout couple d'états (x, y) une expression régulière $\mathcal{E}_{x,y}$ sur A, ce que l'on note : $x \xrightarrow{\mathcal{E}_{x,y}} y$ avec l'interprétation suivante : partant de l'état p, l'automate peut passer dans l'état q en lisant un mot u si et seulement s'il existe une suite d'états $(x_0, \dots, x_n)$ telle que $x_0 = p$, $x_n = q$, et u appartient au langage décrit par l'expression régulière : $\mathcal{E}_{x_0,x_1} \dots \mathcal{E}_{x_{n-1},x_n}$. On note $\mathcal{L}_\mathcal{A}(p, q)$ le langage des mots faisant passer $\mathcal{A}$ de l'état p à l'état q. Remarquons qu'un automate « ordinaire » est un cas particulier d'automate indexé par des expressions régulières où les expressions $\mathcal{E}_{x,y}$ sont $\varnothing$ ou des sommes finies de lettres et de ε.

Démonstration de $2 \Longrightarrow 1$ (méthode par suppression d'états)

Soit $\mathcal{A}$ un automate ordinaire fini. Comme le langage reconnu par $\mathcal{A}$ est une somme de langages $\mathcal{L}_\mathcal{A}(q_0, f)$ où q_0 est un état initial et f un état final quelconque, il suffit de prouver que pour tous états x, y de $\mathcal{A}$ le langage $\mathcal{L}_\mathcal{A}(x, y)$ est régulier. On fixe donc deux états x, y de $\mathcal{A}$ et on considère désormais que $\mathcal{A}$ est indexé par des expressions régulières.

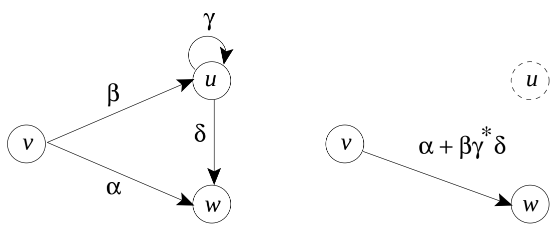


figure 33 : suppression de l'état u

Considérons un état u de $\mathcal{A}$ autre que x ou y. On construit un automate $\mathcal{A}'$, lui aussi indexé par des expressions régulières, ayant pour états tous les états de $\mathcal{A}$ sauf u, et tel que pour tous états v, w de $\mathcal{A}'$ on ait $\mathcal{L}_{\mathcal{A}'}(v, w) = \mathcal{L}_\mathcal{A}(v, w)$. Soient v, w deux états de $\mathcal{A}'$ tels qu'il y a une transition $v \xrightarrow{\alpha} w$ et une chaîne de transitions $v \xrightarrow{\beta} u \xrightarrow{\gamma} u \xrightarrow{\delta} w$ dans $\mathcal{A}$. Alors on remplace la transition $v \xrightarrow{\alpha} w$

par $v \xrightarrow{\alpha+\beta\gamma^*\delta} w$ (cf. figure 33). Ceci étant fait pour tous couples (v, w), on obtient un automate $\mathcal{A}'$ qui vérifie $\mathcal{L}_{\mathcal{A}'}(v, w) = \mathcal{L}_{\mathcal{A}}(v, w)$ pour tous états v, w. Le même procédé peut être appliqué à $\mathcal{A}'$ et de proche en proche on aboutit à l'un des automates $\mathcal{A}''$ ou $\mathcal{A}'''$ de la figure 34 selon que $x = y$ ou $x \neq y$ où α, β, γ, δ et ϕ sont des expressions régulières.

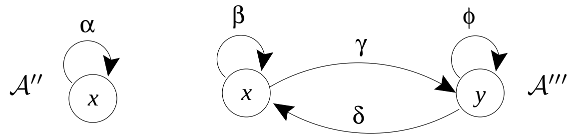


figure 34 : automates réduits

Donc $\mathcal{L}_{\mathcal{A}}(x, x) = \mathcal{L}_{\mathcal{A}''}(x, x) = \alpha^*$ ou $\mathcal{L}_{\mathcal{A}}(x, y) = \mathcal{L}_{\mathcal{A}'''}(x, y) = \beta^*\gamma(\phi+\delta\beta^*\gamma)^*$ sont des langages réguliers et ceci achève la démonstration du théorème de KLEENE. ■

Exemple : considérons l'automate ordinaire non déterministe figure 35 (à gauche). En supprimant d'abord les états 1 et 4 puis l'état 2 et enfin l'état 3, on obtient l'imposante expression :

$$ab + ab(ab)^*ab + (ab + ab(ab)^*a(ab + a(ab)^*a)^*(ab + a(ab)^*ab))^*$$

pour le langage reconnu. En appliquant la même technique à l'automate déterministe équivalent obtenu en **11-3**, on obtient une expression plus simple :

$$\varepsilon + ab(aa^*b)^*.$$

C'est un excellent exercice de patience que de montrer *à la main* l'équivalence de ces deux expressions. Ainsi, le langage reconnu par l'automate est formé du mot vide et de l'ensemble des mots constitués des lettres a et b, commençant par ab, se terminant par b et ne contenant pas deux b consécutifs.

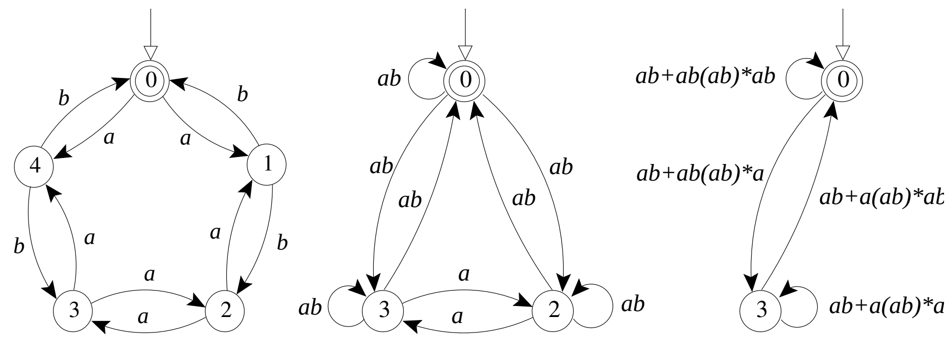


figure 35 : recherche d'une expression régulière

11-5 Stabilité et algorithmes de décision

Théorème : *soient* L, L' *deux langages réguliers sur un alphabet* A. *Alors les langages* $\overline{L}$, $L \cap L'$ *et* $L \setminus L'$ *sont réguliers.*

Démonstration : on a $L \cap L' = \overline{\overline{L} + \overline{L'}}$ et $L \setminus L' = \overline{\overline{L} + L'}$ donc il suffit de prouver que $\overline{L}$ est régulier. Soit $\mathcal{A}$ un automate déterministe reconnaissant L. On peut supposer que $\mathcal{A}$ est complet car si ce n'est pas le cas, on peut ajouter à $\mathcal{A}$ un *état rebut* et diriger vers cet état toutes les transitions indéfinies dans $\mathcal{A}$, et faire boucler l'état rebut sur lui-même pour toutes les lettres de A. On suppose donc que pour tout mot $u \in A^*$, il existe un unique état x_u de l'automate complété tel que $q_0 \overset{u}{\twoheadrightarrow} x_u$ où q_0 est l'état initial de $\mathcal{A}$. Alors L est l'ensemble des mots u tels que $x_u \in F$ où F désigne l'ensemble des états finals de $\mathcal{A}$, et $\overline{L}$ est l'ensemble des mots u tels que $x_u \in \overline{F}$, c'est-à-dire que $\overline{L}$ est reconnu par le même automate que L en changeant l'ensemble des états finals en son complémentaire. ∎

Il est ainsi possible, à partir d'une expression régulière $\mathcal{E}$ définissant un langage L, de construire *par programme* une expression régulière $\mathcal{E}'$ définissant le langage $\overline{L}$: on construit un automate reconnaissant L, on le transforme en automate déterministe et on le complète, on échange les états finals et non finals et on reconstruit une expression régulière, probablement abominable, pour l'automate obtenu. De même on peut par programme trouver une expression régulière reconnaissant l'intersection ou la différence de deux langages définis par des expressions régulières.

Compte tenu de la taille des expressions obtenues, les algorithmes précédents sont pratiquement de peu d'utilité. Cependant, d'un point de vue théorique, il existe donc des algorithmes répondant aux questions suivantes :

- Un langage défini par une expression régulière est-il vide ? $L = M + N$ est vide si et seulement si M et N le sont ; $L = MN$ est vide si et seulement si M ou N l'est ; $L = M^*$ n'est jamais vide. On peut donc ramener le problème de la vacuité au même problème pour des expressions régulières plus courtes et les cas de base, $\varnothing$, ε, *lettre* ou *mot* sont évidents. En termes d'automates, on peut déterminer si le langage reconnu par un automate fini est non vide en cherchant s'il existe un chemin menant d'un état initial de l'automate à un état final, avec l'algorithme de WARSHALL (de complexité $O(n^3)$ où n est la taille de l'automate). Donc la question de la vacuité est résoluble algorithmiquement.

- Un langage défini par une expression régulière est-il fini ? Là aussi, ce problème se ramène au même problème pour des expressions régulières plus courtes.

- Deux expressions régulières définissent-elles le même langage ? Si L et L' sont les langages définis par ces expressions régulières, il suffit de tester si $L \setminus L'$ et $L' \setminus L$ sont vides, ce qui est décidable puisque l'on peut obtenir des expressions régulières pour ces deux langages.

Ainsi, l'égalité *sémantique* de deux expressions régulières est décidable par programme.

11-6 Langages non réguliers

D'après le théorème de KLEENE, un langage régulier est reconnaissable par un automate fini, et cet automate ne peut « mémoriser » qu'un nombre borné de lettres d'un mot à analyser puisqu'il a un nombre fini d'états. En conséquence, tout langage défini par une condition imposant de mémoriser un nombre arbitrairement grand de lettres est non reconnaissable par automate fini et donc non régulier. Les exemples les plus simples de langages non réguliers sont :

– le langage des parenthèses emboîtées : $L = \{a^n b^n \text{ tq } n \in \mathbf{N}\}$;
– le langage des parenthèses correctement imbriquées : L est défini comme étant le plus petit langage sur $\{a, b\}$ solution de l' équation : $L = \varepsilon + aLb + LL$;
– le langage des palindromes : $L = \{u(a + \varepsilon)\tilde{u} \text{ tq } a \in A \text{ et } u \in A^*\}$ où A est un alphabet ayant au moins deux lettres.

Formellement, les démonstrations d'irrégularité utilisent la notion de langage résiduel ou le lemme de l'étoile.

Soit L un langage sur A et $u \in A^*$. Le *langage résiduel* associé à u et L est le langage noté $u^{-1}L$ défini par :

$$u^{-1}L = \{v \in A^* \text{ tq } uv \in L\}.$$

$u^{-1}L$ est l'ensemble des mots pouvant compléter u en un mot élément de L. C'est un langage (sous-ensemble de A^*) qui peut être vide ou non selon les cas.

Théorème : *soit* L *un langage régulier sur un alphabet* A. *Alors pour tout mot* $u \in A^*$ *le langage résiduel* $u^{-1}L$ *est régulier et l'ensemble des langages résiduels obtenus lorsque* u *décrit* A^* *est fini.*

Démonstration : soit $\mathcal{A}$ un automate fini déterministe complet reconnaissant L, q_0 l'état initial et F l'ensemble des états finals de $\mathcal{A}$. Si $u \in A^*$, soit x_u l'état atteint à partir de q_0 en lisant les lettres de u. Alors $u^{-1}L$ est le langage faisant passer $\mathcal{A}$ de x_u à l'un des états finals, donc est régulier. Et il y a au plus autant de langages résiduels distincts associés à L qu'il y a d'états dans $\mathcal{A}$. ■

Remarque : la réciproque est vraie, c'est-à-dire que si un langage admet un nombre fini de résiduels, alors il est régulier (cf. [STERN], ch. 1.4).

Application : soit L le langage des parenthèses emboîtées. Pour $n \in \mathbf{N}$ on note $L_n = (a^n)^{-1}L$. Alors les ensembles L_n sont deux à deux distincts car si $n \neq p$ alors $b^n \in L_n$ et $b^n \notin L_p$. Ainsi, L admet une infinité de langages résiduels distincts, ce qui prouve que L n'est pas régulier. L'irrégularité des autres langages cités ci-dessus se démontre de la même manière en exhibant une infinité de résiduels distincts.

Autre application : soit L_n le langage sur $\{a, b\}$ constitué des mots dont la n-ème lettre avant la fin est un a. L_n est régulier : $L_n = (a + b)^* a(a + b)^{n-1}$, et l'automate non déterministe de la figure 36 reconnaît L_n. Si u est un mot de n lettres sur $\{a, b\}$, alors la connaissance de $u^{-1}L_n$ permet de reconstituer u car

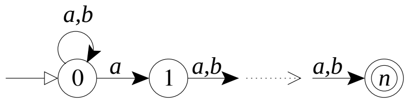


figure 36 : automate à $n+1$ états dont le déterminisé a au moins 2^n états

la i-ème lettre de u est un a si et seulement si $a^{i-1} \in u^{-1}L_n$. Cela signifie que L_n a au moins 2^n langages résiduels distincts, donc tout automate déterministe complet reconnaissant L_n a au moins 2^n états. L'automate non déterministe de la figure 36 constitue un exemple sur lequel toute méthode de déterminisation a une complexité exponentielle.

Théorème : *(lemme de l'étoile) soit L un langage régulier. Alors il existe un entier n tel que tout mot $u \in L$ de longueur supérieure ou égale à n se décompose en trois parties : $u = xyz$ avec $|xy| \leqslant n$, $|y| \geqslant 1$ et $xy^kz \in L$ pour tout $k \in \mathbf{N}$.*

Autrement dit, dans un langage régulier, pour tout mot u suffisamment long on peut supprimer ou dupliquer la partie centrale de u un nombre arbitraire de fois sans quitter L. Contrairement au théorème sur les résiduels en nombre fini, le lemme de l'étoile ne constitue pas une caractérisation des langages réguliers (cf. exercice **11-13**).

Démonstration : puisque L est régulier, il admet un nombre n fini de langages résiduels. Soit $u \in L$ de longueur supérieure ou égale à n. u admet $n+1$ préfixes de longueur inférieure ou égale à n donc il existe deux préfixes produisant le même langage résiduel. Notons x et xy ces préfixes, (donc $|xy| \leqslant n$ et $|y| \geqslant 1$) et soit $u = xyz$. Alors $(xy)^{-1}L = x^{-1}L$ c'est-à-dire :

$$\forall\, v \in A^*,\ xyv \in L \iff xv \in L.$$

En particulier, $xyz \in L$ donc $xz \in L$ mais aussi $x(yz) \in L$ donc $xy(yz) \in L$ et de proche en proche, pour tout $k \in \mathbf{N}$, $xy^kz \in L$. ■

Exemple : le langage des parenthèses emboîtées n'est pas régulier (troisième démonstration en comptant celle de l'exercice **10-8**). En effet, soit $n \in \mathbf{N}^*$ quelconque et $u = a^nb^n \in L$ que l'on découpe en $u = xyz$ avec $|xy| \leqslant n$ et $|y| \geqslant 1$. Le facteur y est de la forme $y = a^p$ avec $p \geqslant 1$, donc $xz = a^{n-p}b^n \notin L$.

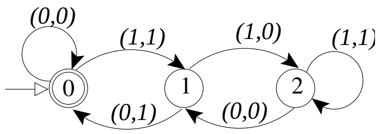
11-7 Exercices

Exercice 11-1 : multiples de 3

Soit $A = \{0, 1\}$. On interprète un mot sur A comme la représentation binaire d'un entier naturel avec le bit de poids fort en tête. Construire un automate reconnaissant les multiples de 3.

Exercice 11-2 : automate séquentiel

Un *automate séquentiel* est un automate déterministe muni de deux alphabets : l'*alphabet d'entrée*, A, et l'*alphabet de sortie*, B. Chaque transition est étiquetée par un couple (a, w) où $a \in A$ et $w \in B^*$. L'automate peut utiliser cette transition s'il reçoit la lettre a et dans ce cas, il émet le mot w (penser à un ordinateur muni d'un clavier et d'une imprimante). Que fait l'automate séquentiel suivant ?



Exercice 11-3 : automate produit

Soient A, B deux automates finis déterministes complets. L'automate produit $A \times B$ a pour états tous les couples (a, b) où a est un état de A et b un état de B et pour transitions : $(a, b) \xrightarrow{x} (a', b')$ si $a \xrightarrow{x} a'$ est une transition de A et $b \xrightarrow{x} b'$ une transition de B. Si L_A et L_B sont les langages reconnus par A et B, comment peut-on reconnaître les langages $L_A \cap L_B$, $L_A \cup L_B$, $L_A \setminus L_B$ à l'aide de $A \times B$?

Exercice 11-4 : ascenseur

Dans un immeuble de trois étages (Rdc, 1er, 2ème) on installe un ascenseur avec un bouton d'appel à chaque étage et un bouton pour chaque étage dans la cabine. Concevoir l'automate régissant cet ascenseur de sorte que :

– l'ascenseur se rend aux étages demandés ;
– l'ascenseur s'arrête aux étages intermédiaires s'il est appelé en cours de voyage à un tel étage.

On supposera qu'il n'y a pas d'appels simultanés.

Exercice 11-5 : algorithme de déterminisation

Soit $\mathcal{A}$ un automate non déterministe sur l'alphabet $\{0, \ldots, p-1\}$ dont les états sont numérotés de 0 à $n-1$, donné par :

1. le nombre n d'états et le nombre p de lettres ;
2. la liste des ε-transitions (liste de couples *(départ,arrivée)*) ;
3. la liste des transitions sur lettre (liste de triplets *(départ, lettre, arrivée)*) ;
4. la liste des états initiaux ;
5. la liste des états finals.

Écrire une fonction CAML produisant un automate déterministe $\mathcal{A}'$ équivalent à $\mathcal{A}$. $\mathcal{A}'$ sera défini par :

1. le nombre q d'états ;
2. la liste des transitions sur lettre (liste de triplets *(départ, lettre, arrivée)*) ;
3. le numéro de l'état initial ;
4. la liste des états finals.

Exercice 11-6 : simplification d'un automate
Soit $\mathcal{A}$ un automate et q un état de $\mathcal{A}$. On appelle *langage d'entrée* de q le langage faisant passer $\mathcal{A}$ d'un état initial à q et *langage de sortie* de q le langage faisant passer $\mathcal{A}$ de q à un état final de $\mathcal{A}$. On dit qu'un état est *accessible* si son langage d'entrée est non vide et *coaccessible* si son langage de sortie est non vide. Un état non coaccessible est appelé *état rebut*.

1. Montrer que l'automate $\mathcal{A}'$ déduit de $\mathcal{A}$ en retirant tous les états non accessibles ou non coaccessibles et les transitions issues de ces états ou aboutissant à ces états est équivalent à $\mathcal{A}$.

2. Donner un algorithme en français permettant de construire $\mathcal{A}'$ à partir de $\mathcal{A}$.

3. Caractériser la propriété « $\mathcal{A}$ est déterministe » en termes de langage d'entrée.

4. Soit $\mathcal{A}$ quelconque. On construit successivement : $\mathcal{B}$ l'automate miroir de $\mathcal{A}$ (on retourne les flèches et on intervertit les états d'entrée et de sortie) ; $\mathcal{C}$ le déterminisé de $\mathcal{B}$ par la méthode des sous-ensembles sans état rebut ; $\mathcal{D}$ l'automate miroir de $\mathcal{C}$; $\mathcal{E}$ le déterminisé de $\mathcal{D}$ par la méthode des sous-ensembles sans état rebut.

   Montrer que $\mathcal{E}$ est équivalent à $\mathcal{A}$, que tous les états de $\mathcal{E}$ sont coaccessibles, et que deux états de $\mathcal{E}$ distincts ont des langages de sortie distincts.

5. Démontrer que tout automate déterministe équivalent à $\mathcal{A}$ a au moins autant d'états que $\mathcal{E}$ ($\mathcal{E}$ est appelé *automate déterministe minimal* équivalent à $\mathcal{A}$).

Exercice 11-7 : langage à une lettre
Soit L un langage sur l'alphabet à une lettre $\{a\}$. Montrer que L est régulier si et seulement s'il existe deux langages finis F et G et un entier n tels que $L = F + G(a^n)^*$.

Exercice 11-8 : racine carrée d'un langage
Soit L un langage sur un alphabet A. On note $\sqrt{L} = \{u \in A^* \text{ tq } u^2 \in L\}$. Montrer que si L est régulier alors $\sqrt{L}$ l'est aussi.

Exercice 11-9 : résiduels d'un langage régulier
Démontrer qu'un langage régulier n'a qu'un nombre fini de résiduels en utilisant uniquement la définition des expressions régulières (c'est-à-dire sans utiliser d'automate reconnaisseur).

Exercice 11-10 : langage des parenthèses emboîtées
1. Écrire une fonction CAML prenant un mot u (vecteur de lettres) et disant s'il appartient au langage $L = \{a^n b^n,\ n \in \mathbf{N}\}$ (langage des parenthèses emboîtées).
2. Un ordinateur exécutant la fonction précédente constitue un automate reconnaissant le langage des parenthèses emboîtées, connu pour être non régulier. Comment est-ce possible ?

Exercice 11-11 : Congruences
Montrer que $L_5 = \{u \in \{a, b\}^* \text{ tq } |u|_a \equiv |u|_b \bmod 5\}$ est un langage régulier.
Montrer que $L = \{u \in \{a, b\}^* \text{ tq } |u|_a = |u|_b\}$ ne l'est pas.

Exercice 11-12 : Un automate sait-il compter ?

1. On représente un triplet d'entiers (m, n, p) par le mot $a^m b^n c^p$ sur l'alphabet $\{a, b, c\}$. Montrer que le langage des additions exactes :

$$L = \{a^m b^n c^p \text{ tq } m + n = p\}$$

n'est pas reconnaissable par automate fini (un automate *ne peut donc pas* vérifier une addition).

2. On représente un triplet (m, n, p) par le mot $m_0n_0p_0m_1n_1p_1 \dots m_qn_qp_q$ sur l'alphabet $\{0, 1\}$ où $m_q \dots m_0$, $n_q \dots n_0$ et $p_q \dots p_0$ sont les écritures binaires de m, n et p prolongées par des zéros pour avoir même longueur. Montrer que le langage des additions exactes pour cette représentation est reconnaissable par automate fini (un automate *peut donc* vérifier une addition).

Exercice 11-13 : contre-exemple au lemme de l'étoile
Soit A un alphabet ayant au moins deux lettres. Démontrer que le langage $L = \{p\tilde{p}q \text{ tq } p \in A^+,\ q \in A^*\}$ vérifie le lemme de l'étoile, mais n'est pas régulier.

Exercice 11-14 : test de finitude pour un langage régulier
Soit L un langage défini par une expression régulière $\mathcal{E}$ sur un alphabet A. On remplace dans $\mathcal{E}$ les symboles $\varnothing$ par des zéros, les symboles ε par des 1, et chaque lettre par un 2. Soit $\mathcal{E}'$ l'expression obtenue. On interprète alors $\mathcal{E}'$ comme une formule « algébrique » sur l'ensemble $X = \{0, 1, 2, \infty\}$ avec les opérations :

$$x + y = \max(x, y)\ ;$$
$$0x = x0 = 0, \qquad 1x = x1 = x, \qquad 22 = 2, \qquad 2\infty = \infty 2 = \infty\ ;$$
$$0^* = 1, \qquad 1^* = 1, \qquad 2^* = \infty, \qquad \infty^* = \infty.$$

Montrer que l'évaluation de $\mathcal{E}'$ retourne 0 si et seulement si L est vide. Caractériser les langages L pour lesquels l'évaluation de $\mathcal{E}'$ retourne 1, retourne 2, retourne ∞.

Problèmes

Tri par distribution

Question 1

Soit a un entier naturel. On sait que a admet une unique écriture binaire :

$$a = a_0 + 2a_1 + 4a_2 + \ldots + 2^{n-1}a_{n-1},$$

où $a_i \in \{0, 1\}$ et n est un entier naturel suffisamment grand. Le nombre a_i est appelé i-*ème bit de* a ou aussi *bit* i *de* a.

a) Donner une expression simple calculant a_0 en fonction de a.

b) Quelle est l'écriture binaire de $\lfloor a/2 \rfloor$ en fonction de celle de a ($\lfloor\ \rfloor$ désigne la partie entière) ?

c) En déduire une fonction CAML `bit : int -> int -> int` telle que `bit` p a retourne le bit p de a si a et p sont des entiers naturels. On donnera deux versions : une version récursive et une version itérative.

Question 2

Soit $v = [|v_0; v_1; \ldots; v_{n-1}|]$ un vecteur à éléments entiers naturels. On veut réordonner ce vecteur de sorte que tous les entiers v_i pairs soient placés en tête et tous les v_i impairs en queue, l'ordre relatif des éléments étant conservé à l'intérieur de chaque groupe. Par exemple si :

$$v = [|3; 1; 4; 1; 5; 9; 2; 6|]$$

alors on souhaite obtenir après transformations :

$$v = [|4; 2; 6; 3; 1; 1; 5; 9|].$$

Il est autorisé pour ce faire d'utiliser un vecteur auxiliaire `w` de longueur n.

a) Donner un algorithme en français effectuant la transformation demandée. On démontrera que cet algorithme répond effectivement au problème posé.

b) Écrire une fonction CAML `permute : int vect -> unit` implémentant cet algorithme.

c) On appelle *transfert* une opération `v.(i) <-` *qqch* ou `w.(j) <-` *qqch*. Calculer le nombre de transferts effectués par votre programme en fonction des nombres a et b d'entiers pairs et impairs dans `v`.

d) On généralise le problème en ajoutant à `permute` un paramètre p entier naturel de sorte que l'appel `permute` p `v` place en tête de `v` les éléments dont le bit p vaut 0 et en queue ceux dont le bit p vaut 1, l'ordre initial des entiers étant conservé à l'intérieur de chaque groupe. Indiquer quelles sont les modifications à apporter à votre code donné en **2-b**.

Question 3
Soit `v` $= [|\nu_0; \nu_1; \ldots; \nu_{n-1}|]$ un vecteur à éléments entiers naturels que l'on veut trier par ordre croissant. On exécute l'algorithme suivant :

α. *Calculer* $M = \max(\nu_0, \nu_1, \ldots, \nu_{n-1})$.
β. *Déterminer un entier* K *tel que* $M < 2^K$.
γ. *Pour* $p = 0, 1, \ldots, K-1$ *faire* `permute p v` *fin pour*

a) Exécuter cet algorithme sur le vecteur `v` $= [|3; 1; 4; 1; 5; 9; 2; 6|]$. On indiquera les valeurs de M, K, et la valeur de `v` à la fin de chaque itération de la boucle γ.

b) Traduire cet algorithme en CAML. On rappelle que CAML dispose d'une fonction `max : int -> int -> int` qui retourne le plus grand de ses deux arguments. Il n'y a pas d'élévation à la puissance dans les entiers et il est interdit d'utiliser celle des flottants.

c) Montrer qu'à la fin de chaque itération de la boucle γ la propriété suivante est vérifiée :

la suite $(\nu_0 \bmod 2^{p+1}, \nu_1 \bmod 2^{p+1}, \ldots, \nu_{n-1} \bmod 2^{p+1})$ *est croissante.*

En déduire que le vecteur `v` est trié par ordre croissant à la fin de l'algorithme.

d) On suppose que tous les entiers ν_i sont compris entre 0 et $2^{30}-1$ (taille maximale d'un entier CAML sur une machine 32 bits). Quelle est la complexité asymptotique de cet algorithme de tri comptée en nombre de transferts effectués ?

Interpolation de LAGRANGE et multiplication rapide

Soient $n \in \mathbf{N}^*$, $P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}$ un polynôme à coefficients complexes de degré strictement inférieur à n et $X = (x_0, x_1, \ldots, x_{n-1})$ une suite de complexes deux à deux distincts. La liste des valeurs de P aux points x_i est la suite notée $Y = P\{X\} = (P(x_0), \ldots, P(x_{n-1}))$. On étudie dans ce problème des algorithmes permettant de calculer $P\{X\}$ à partir de P et X, et des algorithmes permettant de calculer P à partir de X et $P\{X\}$.

Les algorithmes pourront être écrits en français ou en CAML. On supposera que le langage CAML dispose d'un type de données `complexe` représentant les nombres complexes et des opérations arithmétiques usuelles sur les complexes notées `+:`, `-:`, `*:` et `/:`, ainsi que de fonctions de conversion :

`complexe_of_floats` : $(a, b) \longmapsto a + ib$
`floats_of_complexe` : $a + ib \longmapsto (a, b)$.

Les polynômes et les suites de valeurs seront représentés par des vecteurs CAML à éléments complexes :

Le vecteur représentant P est $p = [|\ a_0; a_1; \ldots; a_{n-1}\ |]$;
Le vecteur représentant X est $x = [|\ x_0; x_1; \ldots; x_{n-1}\ |]$;
Le vecteur représentant $P\{X\}$ est $y = [|\ P(x_0); P(x_1); \ldots; P(x_{n-1})\ |]$.

Question 1
Écrire un algorithme calculant la valeur $P(z)$ pour un polynôme P et un complexe z donnés. On prouvera la validité de cet algorithme et on donnera son temps d'exécution mesuré en nombre d'opérations complexes en fonction de la longueur n de P.

Question 2
Écrire un algorithme calculant $Y = P\{X\}$ en fonction du polynôme P et de la suite de points X. Il est supposé que P et X ont même longueur n. Donner la complexité de cet algorithme en fonction de n (en nombre d'opérations complexes).

Question 3
Étant donné deux suites $X = (x_0, \ldots, x_{n-1})$ et $Y = (y_0, \ldots, y_{n-1})$ de n complexes, les x_i étant distincts, il existe un unique polynôme P de degré inférieur ou égal à $n - 1$ tel que $P\{X\} = Y$. Ce polynôme peut être calculé par l'une des relations :

a) *Formule de* LAGRANGE *:* $P(x) = \sum_{i=0}^{n-1} y_i L_i(x)$ avec $L_i(x) = \prod_{\substack{0 \leqslant j < n \\ j \neq i}} \left(\dfrac{x - x_j}{x_i - x_j}\right)$.

b) *Formule de* NEWTON *:* $P(x) = Q(x) + \dfrac{y_{n-1} - Q(x_{n-1})}{U(x_{n-1})} U(x)$

où Q est le polynôme défini par $\deg(Q) \leqslant n-2$, $Q\{x_0, \ldots, x_{n-2}\} = (y_0, \ldots, y_{n-2})$ et $U(x) = (x - x_0)(x - x_1)\ldots(x - x_{n-2})$.

c) *Formule d'*AITKEN *:* $P(x) = \dfrac{x - x_{n-1}}{x_0 - x_{n-1}} P_{0,n-2}(x) + \dfrac{x - x_0}{x_{n-1} - x_0} P_{1,n-1}(x)$

où $P_{a,b}$ est le polynôme de degré inférieur ou égal à $b - a$ tel que :

$$P_{a,b}\{x_a, \ldots, x_b\} = (y_a, \ldots, y_b).$$

Il n'est pas demandé la démonstration de ces formules. Écrire les algorithmes correspondant à ces formules et déterminer leur complexité asymptotique en fonction de n en nombre d'opérations complexes. On pourra utiliser si nécessaire des fonctions auxiliaires calculant la somme de deux polynômes et le produit d'un polynôme par un polynôme de degré 1, en indiquant à part le code de ces fonctions et leur complexité. *Aucun des algorithmes présentés ne doit conduire à calculer plusieurs fois le même polynôme.*

Question 4
On suppose dans cette question que n est une puissance de 2 : $n = 2^\alpha$, et que X est la suite des racines n-èmes de 1 dans $\mathbf{C}$: $x_k = e^{2ik\pi/n}$. Soient $Y = (y_0, \dots, y_{n-1})$ une suite de n complexes quelconques et P le polynôme de degré strictement inférieur à n défini par $P\{X\} = Y$. On note Y_0 et Y_1 les sous-suites paire et impaire associées à Y : $Y_0 = (y_0, y_2, y_4, \dots, y_{n-2})$, $Y_1 = (y_1, y_3, y_5, \dots, y_{n-1})$, X_0 la sous-suite paire associée à X et P_0, P_1 les polynômes de degrés strictement inférieurs à $n/2$ définis par $P_0\{X_0\} = Y_0$ et $P_1\{X_0\} = Y_1$.

a) Montrer que $P(x) = \dfrac{1+x^{n/2}}{2}P_0(x) + \dfrac{1-x^{n/2}}{2}P_1(xe^{-2i\pi/n})$.

b) En déduire un algorithme récursif permettant de calculer P en fonction de Y.

c) Montrer que le temps d'exécution de cet algorithme (en nombre d'opérations complexes) est $O(n\ln(n))$ (on rappelle que n est de la forme 2^α).

d) Montrer de même que l'on peut calculer la suite $Y = P\{X\}$ à partir de P selon la stratégie « diviser pour régner » en $O(n\ln(n))$ opérations.

Question 5
Soient P, Q deux polynômes à coefficients complexes de degrés strictement inférieurs à $n = 2^{\alpha-1}$. En utilisant la question précédente, montrer que l'on peut calculer les coefficients du polynôme PQ en $O(n\ln(n))$ opérations. Expliquer l'intérêt de ce résultat.

Plus longue sous-séquence commune

Définitions

– Soit $A = (a_0, \dots, a_{n-1})$ une liste de n éléments. On appelle : *sous-séquence de* A toute liste obtenue en retirant un nombre quelconque, éventuellement nul, d'éléments à A et en conservant l'ordre relatif des éléments restants. Par exemple, si $A = (3,1,4,1,5,9)$ alors les listes A, $(1,1,5)$ et $()$ (la liste vide) sont des sous-séquences de A, tandis que la liste $(5,1,9)$ n'en est pas une.

– Soient $A = (a_0, \dots, a_{n-1})$ et $B = (b_0, \dots, b_{p-1})$ deux listes. On appelle : *sous-séquence commune à* A *et* B toute liste $C = (c_0, \dots, c_{q-1})$ qui est à la fois une sous-séquence de A et de B. Une *plus longue sous-séquence commune à* A *et* B (en abrégé PLSC(A, B)) est une sous-séquence commune de longueur maximale. La longueur d'une PLSC(A, B) est notée LC(A, B). Remarquer que A et B peuvent avoir plusieurs PLSC, mais qu'elles ont toutes même longueur.

L'objet du problème est l'étude d'algorithmes permettant de calculer la longueur $\textsc{lc}(A,B)$, et de trouver *une* $\textsc{plsc}(A,B)$. Ce problème a des applications pratiques entre autres dans les domaines suivants :

– Biologie moléculaire : déterminer si deux chaînes d'ADN diffèrent « notablement », et sinon, identifier les différences.
– Correction orthographique : déceler les fautes d'orthographe dans un mot incorrect par comparaison avec les mots « voisins » figurant dans un dictionnaire.
– Mise à jour de fichiers : A et B sont deux versions successives d'un fichier informatique ayant subi « peu » de modifications. Soit C une $\textsc{plsc}(A,B)$. Pour diffuser B auprès des détenteurs de A, il suffit de transmettre les listes des éléments de A et de B ne figurant pas dans C, ce qui est généralement plus économique que de transmettre B en totalité.

Les algorithmes pourront être écrits en français ou en CAML. Les calculs de complexité seront faits en supposant que les opérations élémentaires sur les éléments d'une liste (recopie dans une variable, comparaison de deux éléments) ont un temps d'exécution constant. Les résultats seront donnés sous forme asymptotique par rapport aux longueurs des listes traitées.

Question 1 : reconnaissance d'une sous-séquence
On dispose de deux listes $A = (a_0, \dots, a_{n-1})$ et $C = (c_0, \dots, c_{q-1})$.

a) Écrire un algorithme permettant de savoir si C est ou non une sous-séquence de A. La validité de cet algorithme devra être justifiée.

b) En supposant que C est bien une sous-séquence de A, écrire un algorithme calculant *une* liste I des indices des éléments de A à supprimer pour obtenir C. Lorsqu'il existe plusieurs listes I solution, une seule doit être retournée.

c) Exécuter les algorithmes précédents sur l'exemple :

$$A = (3,1,4,1,5,9), \quad C = (1,5,9).$$

Question 2 : calcul de $\textsc{lc}(A,B)$
On dispose des listes $A = (a_0, \dots, a_{n-1})$ et $B = (b_0, \dots, b_{p-1})$. Pour $0 \leqslant i \leqslant n$ et $0 \leqslant j \leqslant p$ on note $A_i = (a_0, \dots, a_{i-1})$, $B_j = (b_0, \dots, b_{j-1})$ et $\ell_{i,j} = \textsc{lc}(A_i, B_j)$ avec la convention : $A_0 = B_0 = ()$.

a) Démontrer pour $i \in [\![1,n]\!]$ et $j \in [\![1,p]\!]$:

$$\ell_{i,j} = \begin{cases} 1 + \ell_{i-1,j-1} & \text{si } a_{i-1} = b_{j-1} \text{ ;} \\ \max(\ell_{i-1,j},\ \ell_{i,j-1}) & \text{sinon.} \end{cases}$$

b) En déduire un algorithme efficace calculant la matrice $L = (\ell_{i,j})_{0 \leqslant i \leqslant n,\ 0 \leqslant j \leqslant p}$. On prouvera la validité de cet algorithme et on donnera sa complexité asymptotique par rapport à n et p. Les listes A, B seront représentées au choix par des listes chaînées ou par des vecteurs.

c) Écrire un algorithme permettant, à partir des listes A, B et de la matrice L de calculer une $\textsc{plsc}(A, B)$.

d) Exécuter les algorithmes précédents sur l'exemple :

$$A = (p, i, o, l, e, t), \quad B = (p, l, i, e, r).$$

Question 3 : pré-traitement de A et B
Pour accélérer le calcul d'une $\textsc{plsc}(A, B)$, on supprime de A tous les éléments ne figurant pas dans B et on supprime de B tous ceux ne figurant pas dans A. En notant A', B' les listes obtenues, toute $\textsc{plsc}(A, B)$ est une $\textsc{plsc}(A', B')$ et réciproquement.

a) Ici on suppose que les éléments de A et B appartiennent à un ensemble E de cardinal suffisamment petit pour qu'il soit envisageable de créer en mémoire des listes ou des vecteurs de longueur $\operatorname{card}(E)$. Ceci est le cas si A et B sont des listes de caractères ; E est alors l'ensemble de tous les caractères représentables en machine, de cardinal 256 (sur les machines courantes). Écrire un algorithme efficace calculant les listes A' et B' à partir de A et B.

b) Ici on suppose que les éléments de A et B appartiennent à un ensemble E trop gros pour tenir en mémoire, mais muni d'une relation d'ordre total notée $\leqslant$. Ceci est le cas si A et B sont des listes d'entiers avec $E = [\![-10^9, 10^9]\!]$. On suppose avoir trié A et B, c'est-à-dire calculé des listes $A^t = (\sigma(0), \dots, \sigma(n-1))$ et $B^t = (\tau(0), \dots, \tau(p-1))$ où σ et τ sont des permutations de $[\![0, n-1]\!]$ et $[\![0, p-1]\!]$ telles que les listes $(a_{\sigma(0)}, \dots, a_{\sigma(n-1)})$ et $(b_{\tau(0)}, \dots, b_{\tau(p-1)})$ sont croissantes. Écrire un algorithme calculant A' et B' à partir de A, B, A^t, B^t.

Question 4
Déterminer la complexité asymptotique du pré-traitement selon **a** ou **b**. Pour **b**, la complexité du tri devra être prise en compte : il n'est pas demandé d'écrire un algorithme de tri, vous pouvez citer un algorithme vu en cours. Est-il raisonnable d'effectuer un pré-traitement (la réponse devra être argumentée) ?

Arbres de priorité équilibrés

Étant donné un arbre binaire a, on note $N(a)$ le nombre de nœuds de a et $H(a)$ la hauteur de a. On pose par convention :

$$N(\varnothing) = 0, \qquad H(\varnothing) = -1.$$

Soit a un arbre binaire dont les étiquettes appartiennent à un ensemble totalement ordonné. On dit que a est un *arbre de priorité équilibré* (APE) s'il vérifie pour tout nœud $n \in a$:

1. l'étiquette de n est supérieure ou égale à celles de ses fils s'il en a ;
2. si g et d désignent les branches gauche et droite issues de n alors :

$$N(g) - 1 \leqslant N(d) \leqslant N(g).$$

Par convention un arbre vide est un APE. Ce type d'arbre intervient dans les problèmes de files d'attente avec priorité où l'on doit effectuer plusieurs tâches selon un ordre de priorité. Si chaque tâche est représentée par un nœud d'un APE étiqueté par la priorité de la tâche, alors la tâche représentée par la racine de l'arbre est l'une des tâches les plus prioritaires. La condition 2. signifie que les autres tâches sont réparties équitablement entre les branches gauche et droite, la branche gauche étant « plus lourde d'un nœud » en cas d'inégalité. On étudie dans ce problème les opérations suivantes :

1. insertion d'une nouvelle tâche ayant une certaine priorité ;
2. extraction d'une tâche prioritaire (celle associée à la racine) et reconstitution d'un APE avec les tâches restantes.

Les APE seront représentés en CAML par des arbres binaires ordinaires suivant la déclaration :

```
type 'a arbre = B_vide | B_noeud of 'a * ('a arbre) * ('a arbre);;
```

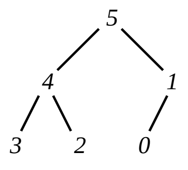
où `'a` est le type ordonné représentant les tâches avec leur priorité.

`B_vide` représente un APE vide ;
`B_noeud`(e, g, d) représente un APE dont la racine porte l'étiquette e et dont les branches gauche et droite sont g et d.

On supposera que deux étiquettes peuvent être comparées par les opérateurs usuels `<`, `<=`, `>` et `>=`.

Question 1
Soit a un APE non vide. Montrer que $H(a) = \lfloor \log_2(N(a)) \rfloor$.

Question 2 : insertion
Écrire une fonction `insertion` telle que si a désigne un APE et e une étiquette alors `insertion(a,e)` renvoie un APE a' contenant tous les nœuds de a et un nœud supplémentaire d'étiquette e. On prouvera la correction de cette fonction, en particulier on prouvera que l'arbre a' retourné est bien un APE, et on donnera sa complexité temporelle. Exécuter « à la main » votre fonction sur l'exemple :

$a =$  $e = 2.$

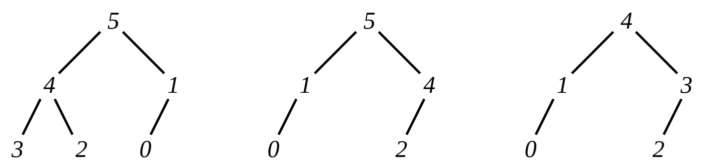
Question 3 : modification de racine
Soit $a = \texttt{B\_noeud}(r, g, d)$ un APE non vide et e une étiquette. On souhaite supprimer l'étiquette r et insérer l'étiquette e pour obtenir un nouvel APE, a'. Dans un premier temps on constitue l'arbre $a_1 = \texttt{B\_noeud}(e, g, d)$ qui n'est pas en général un APE, et on transforme cet arbre en un APE a' ayant les mêmes nœuds. Écrire une fonction `transforme` qui constitue a' à partir de a_1. On prouvera la correction de cette fonction et on donnera sa complexité temporelle en fonction de $N(a)$.

Question 4 : extraction de la racine
Soit a un APE dont on souhaite retirer la racine. On procède en deux temps :

1. On supprime le nœud le plus à gauche dans a et on permute les branches gauche et droite de chaque nœud ascendant du nœud supprimé. Soient a' l'arbre obtenu et e l'étiquette du nœud supprimé.

2. On remplace l'étiquette r de la racine de a' par e et on reconstitue un APE, a'', selon l'algorithme de la question précédente.

Exemple : a a' a''



a) Justifier que l'arbre a' obtenu en 1. est bien un APE.

b) Écrire le code de la fonction `extraction` qui retourne l'arbre a'' à partir de a. Donner la complexité temporelle de `extraction` en fonction de $N(a)$.

Question 5 : utilisation des APE pour le tri
Pour trier une liste $[e_0; e_2; \ldots; e_{n-1}]$ d'éléments appartenant à un ensemble totalement ordonné, on insère un à un ces éléments dans un APE initialement vide puis on extrait par ordre de priorité décroissante les éléments de l'APE obtenu et on les insère dans une nouvelle liste initialement vide. Quelle est la complexité temporelle de cette méthode de tri ?

Compilation d'une expression

On considère une expression arithmétique constituée de constantes, de variables, d'opérateurs unaires (fonctions) et d'opérateurs binaires (addition, multiplication, etc). La *compilation* d'une telle formule consiste à transformer cette formule en une liste d'instructions, le *code compilé*, dont l'exécution par un processeur fournit la valeur de la formule. Les instructions à produire dépendent du processeur exécutant ; on supposera dans ce problème que ce processeur est constitué d'une *unité de calcul* et d'une *pile opérationnelle* dans laquelle sont stockés les opérandes et les résultats des calculs effectués par l'unité de calcul. Les instructions du processeur sont :

– empiler une constante c, notée `empile_c` c ;

– empiler la valeur d'une variable v, notée `empile_v` v ;

– effectuer une opération unaire op, notée `unaire` op, l'opérande est le sommet de la pile et l'unité de calcul le remplace dans la pile par le résultat de l'opération ;

– effectuer une opération binaire op, notée `binaire` op, les deux opérandes, a et b sont pris au sommet de la pile (et retirés de la pile), b étant dépilé le premier puis le résultat, a op b, est placé au sommet de la pile restante.

– effectuer une opération binaire op avec inversion des opérandes, notée `binaire_inv` op, identique à l'opération précédente à ceci près que le résultat empilé est b op a.

Par exemple la formule $a/(b+c)$ peut être compilée de deux manières :

– `empile_v` a; `empile_v` b; `empile_v` c; `binaire` $+$; `binaire` $/$.
– `empile_v` b; `empile_v` c; `binaire` $+$; `empile_v` a; `binaire_inv` $/$.

Dans les deux cas la valeur de la formule est disponible à la fin de l'exécution du programme au sommet de la pile. On remarque sur cet exemple que le deuxième code n'utilise que deux niveaux de pile pour évaluer la formule tandis que le premier en utilise trois. Le but de ce problème est de déterminer, pour une formule f quelconque, quel est le nombre $N(f)$ minimal de niveaux de pile requis pour évaluer cette formule, et de produire un code correspondant à cette utilisation minimale de la pile.

Question 1
Soit l'expression $a+(b-c)*(d-e)$. Combien y a-t-il de codes compilés différents correspondants ? Donner, en le justifiant, un code utilisant le moins possible de niveaux de pile.

Question 2
Déterminer une relation entre $N(f)$ et $N(g)$ où f et g sont deux formules telles que $f = op_1(g)$, op_1 désignant un opérateur unaire. Déterminer de même une relation entre $N(f)$, $N(g)$ et $N(h)$ lorsque $f = g\ op_2\ h$, op_2 désignant une opération binaire.

Question 3

a) Déduire de la question précédente un algorithme de compilation optimal d'une formule f. Cet algorithme sera donné en français, puis codé en CAML en introduisant les définitions de type adéquates pour représenter les formules et les codes compilés. On ne cherchera pas à optimiser le temps de construction du code, mais on évitera la répétition de calculs identiques.

b) Proposer une méthode permettant d'éviter les concaténations en queue de liste dans la production du code. Le codage en CAML de cette amélioration n'est pas demandé.

Question 4
Si la pile opérationnelle d'un processeur comporte 8 niveaux, quelle est la taille minimale, comptée en nombre de nœuds, d'une formule dont le calcul est impossible ?

Recherche d'une chaîne de caractères dans un texte

De nombreux problèmes de traitement de texte ou de reconnaissance de formes impliquent de déterminer si une chaîne de caractères figure dans un texte et si oui, de trouver la première occurrence ou toutes les occurrences de la chaîne dans le texte. Formellement, soit A un alphabet, $m = a_0 \dots a_{|m|-1}$ et $t = b_0 \dots b_{|t|-1}$ deux mots sur A ; on dit que m *figure* dans t s'il existe un entier i tel que :

$$a_0 = b_i, \quad a_1 = b_{i+1}, \dots, \quad a_{|m|-1} = b_{i+|m|-1}. \qquad (*)$$

Un tel entier i est appelé *une position* de m dans t. On étudie dans ce problème divers algorithmes de recherche de la première position de m dans t quand elle existe. Dans les programmes, on pourra supposer que les lettres de A sont représentées par les entiers de $[\![0, |A|-1]\!]$ où $|A|$ est le cardinal de A. Les mots m et t seront représentés par des vecteurs. Les calculs de complexité seront faits en supposant que $|A|$ est constant et que $|m|$ et $|t|$ sont variables avec $|m| = o(|t|)$. Enfin, on supposera que $|A| \geqslant 2$.

Question 1
Le premier algorithme consiste à essayer successivement toutes les valeurs de i jusqu'à constater les égalités $(*)$ ou atteindre la fin du texte t.

a) Écrire une fonction CAML implémentant cet algorithme. Donner sa complexité asymptotique dans le pire des cas en fonction de $|m|$ et $|t|$.

b) On suppose dans cette question seulement que toutes les lettres de m sont différentes. Cette information permet-elle d'améliorer l'algorithme précédent ? Si oui, donner la complexité de l'algorithme modifié.

c) Analyse en moyenne : on note $\mu(t,m)$ le nombre de comparaisons de lettres effectuées par l'algorithme du **1-a** pour tester $(*)$ pour des valeurs données de t et m. En admettant que tous les mots m de longueur ℓ sont équiprobables, le nombre moyen de comparaisons à t, ℓ fixés est :

$$\overline{\mu}(t,\ell) = \frac{1}{|A|^\ell} \sum_{|m|=\ell} \mu(t,m)$$

où la somme porte sur tous les mots m de longueur ℓ. Montrer que l'on a :

$$\overline{\mu}(t,\ell) \leqslant \frac{|A|}{|A|-1}|t|.$$

d) Majorer de même : $\tilde{\mu}(\ell,m) = \frac{1}{|A|^\ell} \sum_{|t|=\ell} \mu(t,m)$ (nombre moyen de comparaisons pour m fixé et t aléatoire de longueur ℓ).

Question 2

Le deuxième algorithme consiste à construire un automate déterministe complet $\mathcal{A}_m$ reconnaissant le langage $\mathcal{L}_m = A^*m$ et à lui faire lire le texte t. Chaque passage par un état final de $\mathcal{A}_m$ correspond à une occurrence de m dans t.

a) Exemple : $A = \{a,b,c\}$ et $m = aabaac$. Construire un automate non déterministe reconnaissant $\mathcal{L}_m$ et en déduire un automate déterministe complet reconnaissant le même langage.

b) Écrire une fonction CAML prenant en argument un automate $\mathcal{A}_m$ (sous une forme à définir) et le texte t et retournant la première position de m dans t si elle existe et déclenchant une erreur dans le cas contraire.

Construction automatique de $\mathcal{A}_m$:

Pour $m,u \in A^*$ on note $S(m,u)$ le plus long mot v qui soit à la fois un préfixe de m et un suffixe de u ($v = \varepsilon$ éventuellement). On construit $\mathcal{A}_m$ de la manière suivante :

- les états de $\mathcal{A}_m$ sont les préfixes de m (il y a $|m|+1$ préfixes en comptant ε et m) ;
- si u est un préfixe de m et a une lettre alors $\mathcal{A}_m$ contient la transition $u \xrightarrow{a} S(m,ua)$;
- l'état initial de $\mathcal{A}_m$ est ε et l'état final m.

c) Montrer que $\mathcal{A}_m$ reconnaît effectivement $\mathcal{L}_m$.

d) Soient u un préfixe de m différent de ε et u' le mot obtenu en supprimant la première lettre de u. Montrer que l'on a pour toute lettre a :

$$S(m,ua) = \begin{cases} ua & \text{si } ua \text{ est un préfixe de } m, \\ S(m,S(m,u')a) & \text{sinon.} \end{cases}$$

e) On représente en machine un préfixe $a_0 \dots a_{i-1}$ de m par l'entier i (0 représentant ε) et la fonction de transition de $\mathcal{A}_m$ par une matrice M à $|m|+1$ lignes et $|A|$ colonnes telle que $M.(i).(a) = j$ représente la transition :

$$a_0 \dots a_{i-1} \xrightarrow{a} a_0 \dots a_{j-1}$$

(les lettres de A sont supposées être représentées par des nombres entiers). Déduire de la question précédente un algorithme *efficace* de calcul de M à partir de m et A. Déterminer sa complexité asymptotique en fonction de $|m|$. Exécuter votre algorithme sur le mot $m = \mathtt{aabaac}$.

f) Montrer que l'automate $\mathcal{A}_m$ construit par la méthode précédente est optimal, c'est-à-dire qu'il n'existe pas d'automate déterministe complet à moins de $|m|+1$ états reconnaissant $\mathcal{L}_m$.

Question 3

Cet algorithme est une variante du premier algorithme : on teste les positions i éventuelles par valeurs croissantes, mais les tests de $(*)$ sont effectués de droite à gauche, c'est-à-dire qu'on effectue dans cet ordre les comparaisons :

$$a_{|m|-1} = b_{i+|m|-1}, \dots, \quad a_1 = b_{i+1}, \quad a_0 = b_i. \tag{$*'$}$$

En cas d'échec, soit $a_k \neq b_{i+k}$ la première différence constatée et a_{k-p} la dernière occurrence de la lettre b_{i+k} dans m avant la position k (par convention, $k-p=-1$ si b_{i+k} ne figure pas dans m avant la position k). On remplace alors i par $i+p$ et on reprend les tests $(*')$.

a) Exécuter cet algorithme avec $m = \mathtt{aabaac}$ et $t = \mathtt{abaabaababcbac}$.

b) Justifier la validité de l'algorithme et donner sa complexité asymptotique dans le pire des cas en fonction de $|m|$ et $|t|$.

c) Programmer cet algorithme. On définira les structures de données et fonctions auxiliaires nécessaires.

d) Peut-on exploiter simultanément les idées des algorithmes 2 et 3 ?

Travaux pratiques

Chemins dans $\mathbf{Z}^2$

On appelle « chemin dans $\mathbf{Z}^2$ » toute suite $\Gamma = (A_0, \dots, A_n)$ de points du plan à coordonnées entières telle que l'on passe de chaque point au suivant par une translation dont le vecteur appartient à l'ensemble :

$$\{N = (0,1), S = (0,-1), E = (1,0), W = (-1,0)\} \qquad \text{(Nord, Sud, Est et West).}$$

Le *mot de* Γ est la suite des vecteurs joignant deux points successifs :

$$\gamma = (\overrightarrow{A_0A_1}, \overrightarrow{A_1A_2}, \dots, \overrightarrow{A_{n-1}A_n}).$$

On dit que :

- Γ a des points multiples s'il existe i, j distincts tels que $A_i = A_j$;
- Γ est fermé si $A_0 = A_n$;
- Γ est fermé simple si $A_0 = A_n$ et si les points A_0, $A_1, \dots$, A_{n-1} sont distincts.

L'objet de ce TP est d'étudier quelques algorithmes sur les chemins : dire si un chemin comporte des points multiples, supprimer les boucles éventuelles en conservant les extrémités d'un chemin, remplir la surface du plan délimitée par un chemin fermé simple. Un chemin Γ sera représenté en CAML soit par la liste des points (couples d'entiers) soit par le mot γ associé, le point initial étant pris par convention égal à $(0,0)$. On utilisera les fonctions de la bibliothèque `graphics` et les fonctions `random_chemins`, `dessine` et `noircit` listées en annexe pour visualiser les chemins étudiés.

1. Conversion mot $\to$ liste de points

Écrire une fonction `points : (int*int) -> direction list -> (int*int) list` qui calcule les points d'un chemin dont l'origine et le mot sont passés en arguments. Tirer des mots de chemins au hasard et les faire afficher (effacer la fenêtre graphique entre deux tracés).

2. Détection et élimination des boucles

a) Écrire une fonction `multiples : (int * int) list -> bool` qui teste si un chemin contient des points multiples. On utilisera la fonction CAML `mem : 'a -> 'a list -> bool` qui dit si une liste contient un élément donné en premier argument.

b) Pour supprimer les boucles dans le chemin $\Gamma = (A_0, \dots, A_n)$ on utilise l'algorithme suivant :

1. Constituer la liste de couples $\ell = ((A_1, A_0), (A_2, A_1), \dots, (A_n, A_{n-1}))$. *Cette liste sera utilisée comme « table de prédécesseurs » grâce à la fonction standard* `assoc` *:* `assoc` M ℓ *renvoie pour un point* M *le premier point* N *tel que le couple* (M, N) *appartient à* ℓ*, c'est-à-dire le premier prédécesseur de* M *dans* Γ *(si* M *n'a pas de prédécesseur,* `assoc` *déclenche une erreur).*

2. Constituer de proche en proche le chemin sans boucle $\Gamma' = (A'_0, \dots, A'_p)$ *associé à* Γ *à l'aide des relations :*

$$A'_p = A_n, \quad A'_i = \text{premier prédécesseur}(A'_{i+1}), \quad A'_0 = A_0.$$

Programmer cet algorithme. On écrira une fonction :

```
sans_boucle : (int*int) list -> (int*int) list
```

qui calcule Γ' à partir de Γ.

Remarque : la complexité asymptotique de `sans_boucle` est $O(n^2)$ car le temps d'une recherche effectuée par `assoc` est linéaire en la taille de la liste d'association. On pourrait améliorer cette complexité en utilisant une structure de données plus efficace qu'une liste d'association, par exemple une table de hachage ou un arbre binaire de recherche.

La fonction suivante retourne un chemin fermé simple aléatoire d'au moins n segments :

```
let rec random_boucle(n) =
  let ch = random_chemin (n,n,n,n-1)    in
  let pt = sans_boucle(points (0,0) ch) in
  if list_length(pt) >= n then pt @ [(0,0)] else random_boucle(n)
;;
```

On tire un chemin non fermé au hasard, on élimine les boucles et on le ferme avec un segment supplémentaire. Si le chemin obtenu est trop court alors on recommence (cela peut boucler indéfiniment mais la théorie des probabilités montre que cette éventualité a une probabilité nulle ; en pratique si n n'est pas trop grand on obtient un chemin convenable assez rapidement).

3. Remplissage

Soit Γ un chemin fermé simple ; on veut « noircir » la région bornée délimitée par Γ (en fait la remplir avec la dernière couleur sélectionnée par `set_color`). Voici un algorithme possible pour ce faire :

1. Déterminer les ordonnées minimale et maximale, y_1 *et* y_2*, des points de* Γ.

2. Pour $y = y_1, y_1 + 1, \ldots, y_2 - 1$ *faire :*

2.1. Déterminer les abscisses (x_i) *des segments verticaux de* Γ *coupant la droite horizontale d'ordonnée* $y + \frac{1}{2}$ *et classer ces abscisses par valeurs croissantes. On obtient une liste* $(x_0, x_1, \ldots, x_{2n-1})$.

2.2 Noircir tous les rectangles $[x_{2i}, x_{2i+1}] \times [y, y+1]$.

fin pour

On se convaincra intuitivement que le nombre d'abscisses trouvées est forcément pair et que les zones noircies sont les bonnes. Programmer cet algorithme. Le tri d'une liste ℓ d'entiers est obtenu par l'expression : `sort__sort (prefix <=)` ℓ.

4. Annexe : fonctions fournies. On entrera les définitions suivantes :

```
#open "graphics";;
open_graph "";;
type direction = N | S | E | W;;

                         (* +--------------------+
                            | Chemin aléatoire   |
                            +--------------------+ *)

(* chemin comportant n fois N, s fois S, e fois E et w fois W *)
let rec random_chemin (n,s,e,w) =
  let t = n+s+e+w in
  if t = 0 then []
  else begin
    let x = random__int(t) in
    if      x < n     then N :: random_chemin(n-1,s,e,w)
    else if x-n < s   then S :: random_chemin(n,s-1,e,w)
    else if x-n-s < e then E :: random_chemin(n,s,e-1,w)
    else                   W :: random_chemin(n,s,e,w-1)
  end
;;
                             (* +-------------+
                                | Affichage   |
                                +-------------+ *)

let pas = 20 and rayon1 = 5 and rayon2 = 3;;
let dessine points =

  (* origine = centre de la fenêtre *)
  let x0 = size_x()/2 and y0 = size_y()/2 in
  let (a,b) = match points with
  | [] -> (x0,y0)
  | (x,y)::suite -> (x0 + x*pas, y0 + y*pas)
  in
  moveto a b;
  fill_circle a b rayon1;
```

```
    (* tracé des segments et des points *)
    let rec trace = function
      | []           -> ()
      | (x,y)::suite ->
          let r = if suite = [] then rayon1 else rayon2 in
          fill_circle (x0 + x*pas) (y0 + y*pas) r;
          lineto (x0 + x*pas) (y0 + y*pas);
          trace(suite)
    in trace(points)
  ;;

  (* noircit un rectangle de diagonale [(x1,y1),(x2,y2)] *)
  let noircit (x1,y1) (x2,y2) =
    let x0 = size_x()/2       and y0 = size_y()/2       in
    let x3 = min x1 x2        and y3 = min y1 y2        in
    let dx = (max x1 x2) - x3 and dy = (max y1 y2) - y3 in
    fill_rect (x0 + x3*pas) (y0 + y3*pas) (dx*pas) (dy*pas)
  ;;
```

Le rôle de `#open "graphics"` est de rendre accessibles les fonctions graphiques (elles sont chargées en mémoire lorsqu'on lance `camlgraph` mais leurs noms courts ne sont connus de l'interpréteur qu'après ce `#open "graphics"`).

`random_chemin` tire au hasard un mot de chemin, γ, comportant n fois Nord, s fois Sud, e fois Est et w fois West. On obtient un mot de chemin fermé si (et seulement si) $n = s$ et $e = w$.

`dessine` trace dans la fenêtre graphique un chemin Γ (liste de points) passé en argument. Les quantités `pas`, `rayon1` et `rayon2` déterminent l'unité d'échelle et la taille des disques matérialisant les points du chemin.

`noircit` trace un rectangle dont on donne deux sommets opposés en tenant compte du facteur d'échelle `pas`.

Files d'attente et suite de HAMMING

1. Implémentation des files d'attente

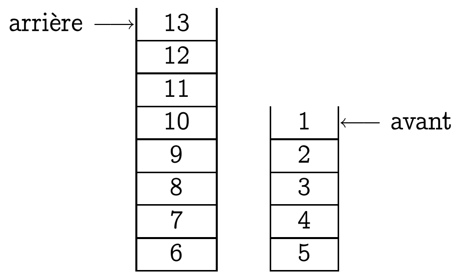
Une file d'attente est liste dont on limite l'usage aux opérations suivantes :

- la liste est-elle vide ?
- extraire la tête de la liste ;
- ajouter un élément en queue de liste.

On rencontre souvent des files d'attente dans la vie courante, par exemple à la caisse d'un supermarché ou à un feu rouge. En informatique une file d'attente permet de stocker des informations qui ne peuvent être traitées de suite, mais qui devront l'être en respectant leur ordre d'arrivée dans la file (structure de données FIFO : *first in, first out*).

On peut réaliser ces opérations avec le type liste chaînée fourni en standard par CAML, mais l'opération « insertion en queue » a une mauvaise complexité sur ce type de listes, aussi utilise-t-on des structures de données mieux adaptées aux fonctionnalités des files d'attente. Dans ce TP on implémentera une file d'attente par un couple de listes chaînées : l'avant de la file, classée par ordre d'arrivée et l'arrière, classée par ordre inverse d'arrivée.

Les nouveaux arrivants seront placés en tête de la liste arrière, les premiers arrivés seront extraits en tête de la liste avant. Lorsque la liste avant est épuisée on retourne la liste arrière, on la place à l'avant et on réinitialise la liste arrière à `[]`. Cette opération de retournement a une complexité linéaire en la taille de la liste arrière, donc le temps d'extraction du premier élément d'une file ainsi implémentée n'est pas constant, mais comme un élément de la file n'est retourné qu'une seule fois, le temps cumulé de n extractions en tête de la file est linéaire en n.




```
type 'a file = {
  mutable avant   : 'a list;
  mutable arrière : 'a list
};;
```

Dans la déclaration ci-dessus les champs `avant` et `arrière` sont déclarés *mutables* de façon à pouvoir être modifiés sur place. Si `f` est une variable de type `'a file` alors on modifie sa liste avant par une instruction de la forme : `f.avant <- qqch`.

Recopier la définition du type `'a file` et programmer les fonctions suivantes :

`nouv_file: unit -> 'a file`	crée une file initialement vide ;
`est_vide : 'a file -> bool`	dit si une file est vide ;
`longueur : 'a file -> int`	renvoie la longueur de la file ;
`ajoute   : 'a file -> 'a -> unit`	ajoute un élément en queue de la file ;
`retire   : 'a file -> 'a`	retire le premier élément de la file et le renvoie ;
`premier  : 'a file -> 'a`	renvoie le premier élément de la file sans le retirer.

Les fonctions `premier` et `retire` déclencheront des erreurs appropriées si la file passée en argument est vide.

2. La suite de HAMMING

Un entier de HAMMING est un entier naturel non nul dont les seuls facteurs premiers éventuels sont $2, 3, 5$. Le *problème de* HAMMING consiste à énumérer les n premiers entiers de HAMMING par ordre croissant. Pour cela on remarque que le premier entier de HAMMING est 1 et que tout autre entier de HAMMING est le double, le triple ou le quintuple d'un entier de HAMMING plus petit (ces cas n'étant pas exclusifs). Il suffit donc d'utiliser trois files d'attente, h_2, h_3 et h_5 contenant initialement le seul nombre 1 puis d'appliquer l'algorithme :

> *déterminer le plus petit des trois nombres en tête des files d'attente, soit* x. *Imprimer* x*, le retirer de chacune des files le contenant et insérer en queue de* h_2*,* h_3 *et* h_5 *respectivement* $2x$*,* $3x$ *et* $5x$*.*

a) Programmer cet algorithme et faire afficher les n premiers entiers de HAMMING. Observer l'évolution des files d'attente à chaque étape.

b) L'algorithme précédent est très dispendieux car il place la plupart des entiers de HAMMING dans les trois files alors qu'une seule suffirait. En effet, si x est un entier de HAMMING divisible à la fois par 2, 3 et 5 alors x a été placé dans h_2 au moment où l'on extrayait $x/2$, dans h_3 au moment où l'on extrayait $x/3$ et dans h_5 au moment où l'on extrait $x/5$. Modifier votre programme de sorte qu'un même entier de HAMMING ne soit inséré que dans une seule des files d'attente.

c) Les entiers CAML sont limités à un milliard environ ce qui ne permet pas d'aller très loin dans la suite de HAMMING. Pour pouvoir traiter des grands nombres on convient de représenter un entier de HAMMING x par le triplet (a, b, c) tel que $x = 2^a 3^b 5^c$. Reprendre votre programme avec cette convention et calculer le millionième entier de HAMMING. Pour comparer deux entiers de HAMMING x et y connus par leurs exposants il suffit de comparer les réels $\ln x$ et $\ln y$ (ln est noté `log` en CAML). On admettra que la précision des calculs sur les flottants est suffisante pour ne pas induire de comparaison erronée.

d) Déterminer expérimentalement la complexité mémoire de l'algorithme de recherche du n-ème nombre de HAMMING. Cette complexité sera mesurée par la somme ℓ des longueurs des trois files h_2, h_3 et h_5. Pour cela on relèvera les valeurs de ℓ pour $n = 1000$, $n = 2000$, $n = 4000$ etc, et on conjecturera une relation simple entre ℓ et n.

Recherche de contradictions par la méthode des consensus

1. Présentation

Soient $p_1, \dots, p_n$ des variables booléennes. On appelle :

- *littéraux* les propositions p_i et leurs complémentaires $\overline{p_i}$;
- *clause* toute disjonction de littéraux, par exemple $p_1 + p_2 + \overline{p_3}$;
- *système de clauses* toute famille finie de clauses.

La clause c est dite *triviale* s'il existe une variable p telle que p et $\overline{p}$ apparaissent dans c. Dans ce cas la valeur de vérité de c est vraie quelles que soient les valeurs de vérité des p_i. La clause *Faux* est la clause constituée d'aucun littéral, sa valeur de vérité est fausse quelles que soient les valeurs des variables p_i. Dans ce TP on ne considérera que des clauses non triviales. Ces clauses seront représentées par un couple de deux listes : la liste des variables apparaissant *positivement* dans c (p_1 et p_2 dans l'exemple précédent) et la liste des variables apparaissant *négativement* (p_3 dans l'exemple précédent). Pour une clause non triviale ces listes sont donc disjointes, et elles sont toutes deux vides pour la clause *Faux* et pour elle seulement.

Un système de clauses est dit *contradictoire* si la conjonction de toutes les clauses du système est égale à la clause *Faux*. Par exemple le système :

$$\{c + \overline{d},\ \overline{c} + \overline{m},\ d + \overline{m},\ m\}$$

est contradictoire comme on peut s'en rendre compte en développant à la main le produit :

$$\begin{aligned}(c + \overline{d})(\overline{c} + \overline{m})(d + \overline{m})m = {} & c\,\overline{c}\,d\,m + c\,\overline{c}\,\overline{m}\,m + c\,\overline{m}\,d\,m + c\,\overline{m}\,\overline{m}\,m \\ & + \overline{d}\,\overline{c}\,d\,m + \overline{d}\,\overline{c}\,\overline{m}\,m + \overline{d}\,\overline{m}\,d\,m + \overline{d}\,\overline{m}\,\overline{m}\,m.\end{aligned}$$

Cette méthode de développement complet n'est pas praticable lorsque n est grand : la longueur d'une clause non triviale peut atteindre n, et il y a 2^n clauses de longueur n ; le produit de toutes ces clauses comporte donc dans sa forme développée n^{2^n} termes... On peut aussi se rendre compte qu'un système est contradictoire en calculant la table de vérité de la conjonction des clauses, mais pour n variables booléennes il y a quand même 2^n valeurs de vérité à calculer. L'objectif de ce TP est d'étudier un autre algorithme de reconnaissance de contradictions, appelé *méthode des consensus*, qui permet de déterminer si un système est contradictoire en général plus efficacement qu'en calculant le produit de toutes les clauses du système ou en cherchant la table de vérité de leur conjonction.

Si c et c' sont deux clauses, on dit que c *implique* c' si tout littéral apparaissant dans c apparaît aussi dans c'. Ceci correspond à la notion usuelle d'implication : $c \Longrightarrow c'$ si et seulement si tout choix des variables rendant c vraie rend aussi c' vraie. Par convention la clause *Faux* implique toutes les autres. Dans un système S, une clause $c \in S$ est dite *minimale* s'il n'existe pas de clause $c' \in S \setminus \{c\}$ telle que $c' \Longrightarrow c$.

Si c et c' sont deux clauses telles qu'il existe une variable p apparaissant positivement dans une clause et négativement dans l'autre, par exemple $c = p + c_1$ et $c' = \overline{p} + c_1'$ (c_1 et c_1' ne contenant ni p ni $\overline{p}$), on appelle *consensus* des clauses c et c' la clause $c'' = c_1 + c_1'$. c'' a la propriété d'être vraie chaque fois que c et c' le sont, et c'est la plus petite clause pour l'ordre d'implication parmi les clauses indépendantes de p ayant cette propriété (c'est-à-dire si d est une clause indépendante de p, on a $cc' \Longrightarrow d$ si et seulement si $c'' \Longrightarrow d$). S'il existe une autre variable q apparaissant positivement dans l'une des clauses c, c' et négativement dans l'autre alors $c_1 + c'_1$ contient $q + \overline{q}$ donc est la clause triviale. Ceci montre qu'il existe trois cas possibles pour deux clauses c et c' :

– ou bien elle n'ont pas de consensus (aucune variable n'apparaît positivement dans une clause et négativement dans l'autre) ;
– ou bien elles ont un unique consensus ;
– ou bien elles ont plusieurs consensus mais alors ils sont tous triviaux.

Soit S un système de clauses. On détermine si S est contradictoire de la manière suivante :

1. simplifier S en retirant toutes les clauses non minimales ;
2. former tous les consensus non triviaux entre deux clauses de S et les ajouter à S ;
3. recommencer les étapes 1 et 2 tant que l'on obtient en sortie de 2 un système différent de celui en entrée de 1 ;
4. le système d'origine est contradictoire si et seulement si le système final est réduit à $\{Faux\}$.

La terminaison de cet algorithme est immédiate (il n'y a qu'un nombre fini de systèmes de clauses pour un ensemble de variables donné), sa correction l'est moins, ce pourrait être un très bon sujet de problème.

Exemple : *quand ils ont un devoir, les élèves apprennent leur cours. S'ils ont appris leur cours, ils n'ont pas de mauvaises notes. S'ils n'ont pas de devoir, ils n'ont pas non plus de mauvaise notes. Montrer que les élèves n'ont jamais de mauvaises notes.*

On modélise ce problème avec trois variables booléennes : d = « avoir un devoir » ; m = « avoir une mauvaise note » ; c = « apprendre son cours » ; et on dispose des hypothèses :

$$h_1 \equiv (d \Longrightarrow c \equiv \overline{c} + d), \quad h_2 \equiv (c \Longrightarrow \overline{m} \equiv \overline{c} + \overline{m}), \quad h_3 \equiv (\overline{d} \Longrightarrow \overline{m} \equiv d + \overline{m}).$$

Il s'agit de montrer que $h_1\,h_2\,h_3 \Longrightarrow \overline{m}$ c'est-à-dire que le système :

$$S = \{c + \overline{d},\ \overline{c} + \overline{m},\ d + \overline{m},\ m\}$$

est contradictoire. En déroulant à la main l'algorithme des consensus on obtient successivement :

1. $S = \{c + \overline{d},\ \overline{c} + \overline{m},\ d + \overline{m},\ m\}$
2. $S = \{c + \overline{d},\ \overline{c} + \overline{m},\ d + \overline{m},\ m,\ \overline{d} + \overline{m},\ c + \overline{m},\ \overline{c},\ d\}$
1. $S = \{c + \overline{d},\ m,\ \overline{d} + \overline{m},\ c + \overline{m},\ \overline{c},\ d\}$
2. $S = \{c + \overline{d},\ m,\ \overline{d} + \overline{m},\ c + \overline{m},\ \overline{c},\ d,\ \overline{d},\ c,\ \overline{m}\}$
1. $S = \{m,\ \overline{c},\ d,\ \overline{d},\ c,\ \overline{m}\}$
2. $S = \{m,\ \overline{c},\ d,\ \overline{d},\ c,\ \overline{m},\ Faux\}$
1. $S = \{Faux\}$
2. $S = \{Faux\}$

Dans cet exemple on aurait aussi pu partir du système $S' = \{c+\overline{d},\ \overline{c}+\overline{m},\ d+\overline{m}\}$ et le « simplifier » par l'algorithme des consensus, on obtient en sortie un système contenant $\overline{m}$ ce qui suffit à prouver $h_1\, h_2\, h_3 \Longrightarrow \overline{m}$.

2. Types de données et fonctions fournies

Les variables booléennes seront représentées par des chaînes de caractères, les clauses par des couples de listes de chaînes de caractères et les systèmes de clauses par des listes de tels couples :

```
                (* v. positives, v. négatives *)
type clause  == (string list)   *  (string list);;
type systeme == clause list;;

(* affichage d'une clause s *)

let rec concat sep liste = match liste with
| [] -> ""
| [x] -> x
| x::suite -> x^sep^(concat sep suite);;

let string_of_clause(p,n) =
  (if n = [] then "" else (concat "." n)^" => ") ^
  (if p = [] then "Faux" else concat "+" p);;

let print_clause(c) = format__print_string(string_of_clause c);;
install_printer "print_clause";;

(* construction de clauses *)
let prefix =>    a b = ([b],  [a])     (* a => b          *)
and prefix =>~   a b = ([],   [a;b])   (* a => non b      *)
and prefix ~=>   a b = ([a;b],[])      (* non a => b      *)
and prefix ~=>~  a b = ([a],  [b])     (* non a => non b *)
;;
```

La fonction `print_clause` affiche une clause c sous la forme $c_1 \Longrightarrow c_2$ où c_1 est la conjonction des variables négatives de c et c_2 la disjonction des variables positives de c (de fait, $(\overline{p}+\overline{q}+r+s) \equiv (p.q \Longrightarrow r+s)$). Le rôle de l'instruction `install_printer "print_clause"` est d'indiquer au système qu'il doit dorénavant utiliser cette fonction à chaque fois qu'il doit afficher une clause. Les fonctions `=>`, `~=>`, `=>~` et `~=>~` permettent de saisir des clauses sous une forme presque mathématique, on écrira :

```
let s' = ["d" => "c"; "c" =>~ "m"; "d" ~=>~ "m"];;
```

pour désigner le système $S' = \{h_1, h_2, h_3\}$ présenté en exemple. En réponse, le système interactif affiche :

```
s' : (string list * string list) list = [d => c; c.m => Faux; m => d]
```

conformément au code de `print_clause`. On pourra par ailleurs utiliser les fonctions à caractère ensembliste faisant partie de la bibliothèque standard de CAML : `union`, `intersect`, `subtract`, `mem` et `except`.

3. Programmation de l'étape 1
Écrire les fonctions suivantes :

```
implique  : clause  -> clause  -> bool
present   : clause  -> systeme -> bool
ajoute    : clause  -> systeme -> systeme
simplifie : clause  -> systeme -> systeme
```

– `implique` c c' dit si $c \Longrightarrow c'$.
– `present` c s dit s'il existe une clause c' dans s telle que $c' \Longrightarrow c$.
– `ajoute` c s retourne un système s' équivalent au système $s \cup \{c\}$ en retirant de $s \cup \{c\}$ toutes les clauses $c' \neq c$ telles que $c \Longrightarrow c'$. De la sorte, si l'on part d'un système s ne contenant que des clauses minimales, le système s' retourné a aussi cette propriété.
– `simplifie` s retourne un système s' équivalent à s et ne contenant que des clauses minimales. Pour ce faire, on ajoutera une à une les clauses de s à un système initialement vide.

4. Programmation des étapes 2 et 3
Écrire les fonctions suivantes :

```
ajoute_cons      : clause  -> clause  -> systeme -> systeme
ajoute_consensus : clause  -> systeme -> systeme -> systeme
cloture          : systeme -> systeme
```

– `ajoute_cons` c c' s calcule le consensus éventuel entre c et c' et, s'il est non trivial, l'ajoute au système s.
– `ajoute_consensus` c s s' calcule tous les consensus non triviaux entre c et une des clauses de s et les ajoute au système s'.
– `cloture` s calcule la clôture par consensus du système s, c'est-à-dire le système s' que l'on obtient en sortie de l'étape 3. On utilisera la fonction `subtract` pour détecter si deux systèmes sont égaux indépendamment de l'ordre dans lequel sont rangées les clauses dans chaque système.

5. Applications

a) Vérifier avec votre programme que les élèves ne peuvent jamais avoir de mauvaises notes (sous les hypothèses de l'exemple donné en introduction).

b) Les règles d'admission dans un club écossais sont les suivantes :

> *Tout membre non écossais porte des chaussettes rouges.*
> *Tout membre qui porte des chaussettes rouges porte un kilt.*
> *Les membres mariés ne sortent pas le dimanche.*
> *Un membre sort le dimanche si et seulement s'il est écossais.*
> *Tout membre qui porte un kilt est écossais et marié.*
> *Tout membre écossais porte un kilt.*
> (LEWIS CARROLL)

Montrer que les règles de ce club sont si contraignantes que personne ne peut en être membre.

6. Affichage des étapes d'une contradiction

Le programme précédent dit que le club écossais ne peut pas avoir de membres, mais on ne sait pas vraiment pourquoi : on sait seulement que la machine a trouvé une contradiction entre toutes les règles. En fait il n'est pas difficile d'obtenir une démonstration en bonne et due forme de la contradiction trouvée, il suffit chaque fois qu'on produit un nouveau consensus de mémoriser à partir de quelles clauses il a été obtenu. Lorsqu'on aboutit à la clause *Faux*, il ne reste plus qu'à afficher ces étapes dans le bon ordre. Modifier le type `clause` de la manière suivante :

```
type clause == (string list) * (string list) * raison
and  raison  =  H | D of clause * clause;;
```

Une clause est à présent constituée de la liste de ses variables positives, de la liste de ses variables négatives, et d'un composante de type `raison` indiquant son origine : `H` pour une hypothèse, `D(c1,c2)` pour une clause déduite par consensus des clauses c_1 et c_2.

Modifier en conséquence votre programme de recherche de contradiction et écrire une fonction d'affichage imprimant, lorsqu'une contradiction a été trouvée, toutes les étapes ayant produit cette contradiction. Dans le cas du club écossais on obtiendra par exemple :

```
let syst = cloture ["e" ~=>  "r";
                    "r"  =>  "k";
                    "m"  =>~ "d";
                    "d"  =>  "e"; "e" => "d";
                    "k"  =>  "e"; "k" => "m";
                    "e"  =>  "k"]
          in explique (hd syst);;

J'ai l'hypothèse : k => e
J'ai l'hypothèse : r => k
De r => k et k => e je déduis : r => e
J'ai l'hypothèse : e+r
De e+r et r => e je déduis : e
J'ai l'hypothèse : e => d
J'ai l'hypothèse : m.d => Faux
De m.d => Faux et e => d je déduis : m.e => Faux
J'ai l'hypothèse : e => k
J'ai l'hypothèse : k => m
De k => m et e => k je déduis : e => m
De m.e => Faux et e => m je déduis : e => Faux
De e et e => Faux je déduis : Faux
- : unit = ()
```

Modélisation d'un tableur

Un tableur est un programme permettant de gérer un *tableau* constitué d'une matrice de valeurs et d'une matrice de formules. Voici un exemple de tableau :

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

`valeurs`

	0	1	2
0		A_{22}	A_{00}
1	$2A_{20}$	$A_{20}+A_{22}$	$A_{10}+A_{21}$
2	A_{02}	$A_{11}+A_{01}$	

`formules`

Chaque formule indique comment calculer l'élément de la matrice des valeurs ayant même position, par exemple la cellule ligne 1, colonne 2 contient la formule $A_{10}+A_{21}$ ce qui signifie que la valeur ligne 1, colonne 2 doit être égale à la somme des valeurs ligne 1, colonne 0 et ligne 2, colonne 1. Manifestement ce n'est pas le cas et le rôle du tableur est de remplacer les éléments de la matrice des valeurs qui sont associés à une formule par l'évaluation de la formule en question. A priori il suffit d'évaluer chaque formule et de modifier en conséquence la matrice des valeurs, mais un problème délicat se pose : dans quel ordre doit-on évaluer les formules ?

Ce problème est appelé *tri topologique :* étant donnés un ensemble fini d'objets et un ensemble fini de relations de dépendance entre ces objets, classer les objets de sorte qu'aucun ne dépende d'un objet situé après lui dans le classement. L'algorithme du tri topologique est le suivant :

1. *Noter pour chaque objet* o *de combien d'objets il dépend et quels sont les objets qui dépendent de lui. Ces objets seront appelés* successeurs de o. *Initialiser la liste résultat à* $\varnothing$.
2. *Tant qu'il reste des objets ayant un compte de dépendance nul : sélectionner un objet ayant un compte de dépendance nul, le placer dans la liste résultat et diminuer d'une unité les comptes de dépendance de tous ses successeurs.*
3. *S'il reste des objets non classés alors le problème n'a pas de solution, sinon retourner la liste résultat.*

La terminaison et la correction de cet algorithme sont évidentes. L'objet de ce TP est d'implémenter l'algorithme du tri topologique dans le cas particulier d'un tableur. On utilisera les déclarations CAML suivantes :

```
type lcell  == (int*int) list;;     (* liste de cellules     *)
type formule = Rien | Somme of lcell;;
type tableau = {
          n  : int; p : int;        (* dimensions            *)
          v  : int vect vect;       (* valeurs des cellules  *)
          f  : formule vect vect;   (* formules              *)
```

```
            dep: int vect vect;       (* nombre de dépendances *)
            suc: lcell vect vect;     (* successeurs           *)
    mutable lt : lcell                (* ordre de calcul       *)
  };;
```

Si t est une variable de type tableau et i, j sont deux entiers on aura :

- t.n est le nombre de lignes du tableau ;
- t.p est le nombre de colonnes du tableau ;
- t.v.(i).(j) est la valeur ligne i, colonne j ;
- t.f.(i).(j) est la formule ligne i, colonne j ;
- t.dep.(i).(j) est le compte de dépendance de la cellule ligne i, colonne j ;
- t.suc.(i).(j) est la liste des successeurs de la cellule ligne i, colonne j ;
- t.lt est la liste des cellules triée par ordre topologique ;
- si t.f.(i).(j) = Rien alors la valeur t.v.(i).(j) n'est soumise à aucune contrainte ;
- si t.f.(i).(j) = Somme[$c_0; \ldots; c_{k-1}$] alors la valeur t.v.(i).(j) doit être égale à la somme des valeurs des cellules $c_0, \ldots, c_{k-1}$.

Ici on s'est limité aux formules de type addition de cellules, d'autres types de formules pourraient être naturellement implémentées. Le tableau donné en exemple est défini par :

```
let t = {
  n = 3; p = 3;
  v = [| [|  1;   2;   3  |];
         [|  4;   5;   6  |];
         [|  7;   8;   9  |] |];
  f = [| [|Rien;           Somme[2,2];      Somme[0,0]     |];
         [|Somme[2,0; 2,0]; Somme[2,0; 2,2]; Somme[1,0; 2,1]|];
         [|Somme[0,2];      Somme[1,1; 0,1]; Rien           |] |];
  dep = make_matrix 3 3 0;
  suc = make_matrix 3 3 [];
  lt  = []
};;
```

1. Tri topologique

Programmer l'algorithme du tri topologique. Voici à titre indicatif une décomposition possible des opérations à effectuer :

`dépendances : tableau -> unit`
: calcule les matrices dep et suc d'un tableau à partir de sa matrice f.

`place : tableau -> (int*int) -> unit`
: place une cellule dans la liste lt et décrémente le compte de dépendance de chaque successeur de la cellule placée.

`décompte : tableau -> lcell -> unit`
: décrémente les comptes de dépendance des cellules transmises en deuxième argument et appelle place pour chaque cellule dont le compte de dépendance devient nul.

`tri_topo : tableau -> unit`
effectue le tri topologique du tableau transmis.

`place` et `décompte` sont deux fonctions mutuellement récursives, elles devront être déclarées de façon conjointe par :

```
let rec place t (i,j) = ... and décompte t liste = ...;;
```

2. Calcul du tableau
Maintenant que le tableau est « trié topologiquement » il reste à évaluer les formules dans l'ordre et à modifier en conséquence la matrice `v` du tableau. Programmer cela. On doit trouver pour le tableau servant d'exemple les valeurs suivantes :

	0	1	2
0	1	9	1
1	2	10	21
2	1	19	9

3. Complément
Lorsque l'on modifie la valeur d'une cellule qui n'est pas soumise à contraintes, il faut réévaluer toutes les formules qui font intervenir cette cellule, directement ou indirectement. Une solution naïve consiste à recalculer tout le tableau, mais on peut faire mieux en notant pour chaque cellule non contrainte la liste des cellules à recalculer, triée par ordre topologique.

Analyse syntaxique

1. Présentation
L'objectif de ce TP est d'effectuer l'analyse syntaxique d'un programme puis de l'exécuter à l'aide d'un interpréteur à partir de l'expression arborescente fournie par l'analyseur. Le langage de programmation que devra reconnaître l'analyseur est le suivant :

– *Les valeurs définies dans le langage sont les nombres entiers et les fonctions à un argument.*
– *Les opérations définies dans le langage sont les opérations arithmétiques usuelles entre nombres entiers : $+$, $-$, $*$, / et % (modulo) ainsi que l'application de fonction : si e_1 et e_2 sont deux expressions alors*

$$\mathrm{e}_1\ \mathrm{e}_2$$

désigne l'application de la fonction e_1 *à la valeur* e_2*. Une telle application n'est valide que si* e_1 *désigne effectivement une fonction et* e_2 *une valeur appartenant au domaine de* e_1*, mais cette restriction ne sera pas prise en compte dans le cadre de l'analyse syntaxique ; c'est l'interpréteur devant évaluer une expression qui signalera une erreur à l'exécution en présence d'une application invalide. De même, les opérations arithmétiques impossibles telle que l'addition d'une fonction et d'un nombre entier ne seront signalées qu'à l'exécution.*

– *Les priorités entre opérations binaires sont les priorités algébriques habituelles : l'application de fonction est prioritaire devant la multiplication et la division qui sont prioritaires devant l'addition et la soustraction, elles-mêmes prioritaires devant le modulo. Les opérations de même priorité sont évaluées de gauche à droite et des parenthèses peuvent modifier l'ordre d'évaluation d'une expression.*
– *Les constructions syntaxiques du langage sont la liaison locale d'une expression à un identificateur (*`let x =` e_1 `in` e_2*), la définition de fonction (*`fun x ->` e_1*) et l'alternative (*`if` e_1 `then` e_2 `else` e_3*) où* e_1*,* e_2*,* e_3 *sont des expressions et* `x` *un identificateur. Dans le cas de l'alternative, l'interpréteur devra évaluer* e_1*, vérifier qu'il s'agit d'un nombre entier, puis évaluer* e_2 *si* $e_1 = 0$ *ou* e_3 *si* $e_1 \neq 0$*. L'expression non sélectionnée ne doit pas être évaluée. Les définitions introduites par* `let` *seront systématiquement considérées comme récursives.*

Ce langage, bien que réduit, est suffisant pour coder des calculs intéressants. Par exemple :

```
let f = fun x -> if x*(x-1) then 1 else f(x-1) + f(x-2) in f 5
```

2. Analyse lexicale

La première étape de l'analyse d'une phrase est le découpage de cette phrase en lexèmes. On définit le type suivant pour les éléments lexicaux du langage :

```
type lexème =
| Nombre of int                     (* constante      *)
| Ident  of string                  (* identificateur *)
| Opr    of int * char              (* opr. infixe    *)
| Let | In | Fun | If | Then | Else (* mots-clés      *)
| Egal | Flèche                     (* = et ->        *)
| ParO | ParF                       (* parenthèses    *)
;;
```

Écrire une fonction `lexèmes : string -> lexème list` qui décompose une chaîne de caractères en liste de lexèmes. Un identificateur est une suite de caractères alphabétiques la plus longue possible. Les mots `let`, `in`, `fun`, `if`, `then` et `else` sont des *mots-clés* et non des identificateurs, ils seront codés par les lexèmes correspondants. Un opérateur infixe est constitué d'un unique caractère, `+ - * / %`, auquel on attache un niveau de priorité : 1 pour le modulo, 2 pour l'addition et la soustraction, 3 pour la multiplication et la division. Les parenthèses et le symbole d'égalité sont codés par le caractère correspondant, la flèche de définition de fonction est codée par les caractères `->`. La chaîne transmise en argument à `lexèmes` peut aussi contenir des blancs (espace, tabulation, retour à la ligne codés respectivement `‘ ‘`, `‘\t‘` et `‘\n‘` en CAML) qui seront considérés comme des séparateurs. Si un caractère de la chaîne n'est pas reconnu alors la fonction `lexèmes` devra déclencher une erreur. Exemple :

```
lexèmes "let f = fun x -> if x*(x-1)
         then 1 else f(x-1) + f(x-2) in f 5";;
- : lexème list =
[Let; Ident "f"; Egal; Fun; Ident "x"; Flèche; If; Ident "x";
  Opr (3, ‘*‘); ParO; Ident "x"; Opr (2, ‘-‘); Nombre 1; ParF;
  ...
```

3. Analyse syntaxique

L'analyseur syntaxique a pour rôle de transformer une liste de lexèmes telle que celle retournée par la fonction `lexèmes` précédente en une expression codée sous forme arborescente selon la définition du type `expression` suivante :

```
type expression =
| Const of int                               (* nombre           *)
| Var   of string                            (* identificateur   *)
| Fct   of string * expression               (* fonction         *)
| Bin   of expression * char * expression    (* opérateur binaire *)
| Appl  of expression * expression           (* appl. de fonction *)
| Test  of expression * expression * expression (* sélection     *)
| Letin of string * expression * expression  (* définition locale *)
;;
```

La transformation s'effectue par réécritures de la liste à analyser selon les règles :

1. $\texttt{Nombre}(n) \longrightarrow \texttt{Const}(n)$ et $\texttt{Ident}(x) \longrightarrow \texttt{Var}(x)$
2. (*expression*) $\longrightarrow$ *expression*
3. exp_1 $\texttt{Opr}(p, op)$ $exp_2 \longrightarrow \texttt{Bin}(exp_1, op, exp_2)$
4. exp_1 $exp_2 \longrightarrow \texttt{Appl}(exp_1, exp_2)$
5. `Fun` $\texttt{Ident}(x)$ `Flèche` *expression* $\longrightarrow \texttt{Fct}(x, expression)$
6. `Let` $\texttt{Ident}(x)$ `Egal` exp_1 `In` $exp_2 \longrightarrow \texttt{Letin}(x, exp_1, exp_2)$
7. `If` exp_1 `Then` exp_2 `Else` $exp_3 \longrightarrow \texttt{Test}(exp_1, exp_2, exp_3)$

En principe il suffit d'appliquer ces règles tant que possible jusqu'à réduire la liste à une unique expression. Mais on doit aussi tenir compte des priorités des opérateurs binaires lors de l'application des règles 3 et 4 et préciser de quelle manière on repère la fin de la dernière expression pour chacune des règles 5,6 et 7. On adoptera la convention de CAML : une expression derrière `Flèche`, `In` ou `Else` s'étant aussi loin que possible, c'est-à-dire jusqu'au premier des évènements suivants :

- la fin de la phrase ;
- la rencontre d'une parenthèse fermante non associée à une ouvrante ;

– la rencontre d'un `In` non associé à un `Let` ;
– la rencontre d'un `Then` ou d'un `Else` non associé à un `If`.

Ainsi, `if` a `then` b `else` $c + d$ doit être transformé en : `Test`$(a, b,$ `Bin`$(c, `+`, d))$.

Algorithme d'analyse syntaxique : *parcourir la liste à analyser et empiler les lexèmes sur une pile contenant les parties de la phrase déjà traduites en expressions et les lexèmes délimitant un groupe non encore complet. Les nombres et identificateurs seront traduits en constantes ou variables lors de l'empilement. Effectuer sur la pile, au moment de l'empilement, les réductions qui peuvent l'être compte-tenu des règles de priorité, les lexèmes* `ParF`*,* `In`*,* `Then` *et* `Else` *étant considérés comme des opérateurs de priorité* 0 *pour l'application de la règle* 3*.*

Programmation : la pile de réduction sera implémentée par une liste dont la tête est le sommet de pile. Comme elle doit contenir à la fois des lexèmes et des expressions, on définira un type mixte pour les éléments de la pile :

```
type lexème_ou_expression = L of lexème | E of expression;;
type pile == lexème_ou_expression list;;
```

Écrire une fonction `empile : lexème_ou_expression -> pile -> pile` qui empile un lexème ou une expression sur la pile passée en argument, procède récursivement aux réductions éventuelles et retourne la nouvelle pile. La fonction `expression` ci-dessous utilise `empile` pour calculer l'expression associée à une liste de lexèmes :

```
let expression liste =
   let pile = ref [L ParO] in
   do_list (fun x -> pile := empile (L x) !pile) liste;
   pile := empile (L ParF) !pile;
   match !pile with
   | [E e] -> e
   | _     -> failwith "erreur de syntaxe"
;;
```

On empile une parenthèse ouvrante, tous les lexèmes de la phrase à analyser puis une parenthèse fermante, ce qui a pour effet de réduire la pile à une seule expression si la liste à analyser est correcte. C'est cette expression qui est retournée. Exemple :

```
expression (lexèmes "let f = fun x -> if x*(x-1)
                     then 1 else f(x-1) + f(x-2) in f 5");;
- : expression =
 Letin
  ("f",
   Fct
    ("x",
     Test
      (Bin (Var "x", `*`, Bin (Var "x", `-`, Const 1)), Const 1,
```

```
        Bin
         (Appl (Var "f", Bin (Var "x", `-`, Const 1)), `+`,
          Appl (Var "f", Bin (Var "x", `-`, Const 2))))),
    Appl (Var "f", Const 5))
```

4. Évaluation

Pour évaluer une expression comportant des variables on a besoin de connaître les valeurs de ces variables. On utilisera ici une liste d'association formée de couples (*nom*,*valeur*) avec la convention qu'une variable x a pour valeur la valeur indiquée par le premier couple (x,*qqch*) figurant dans la liste. S'il n'y a pas de couple commençant par x alors le programme d'évaluation provoquera une erreur. Une telle liste est appelée *liste d'environnement*. On définit le type des valeurs possibles pour une expression :

```
type valeur =
  | Int of int
  | Cloture of environnement * expression
and environnement == (string * valeur) list
;;
```

Une valeur est soit un nombre entier x codé `Int`(x), soit une fonction f codée `Cloture`(*env*,*expr*) où *expr* est l'expression donnée pour la définition de f dans `fun` x `->` *expr* et *env* est la liste d'environnement qui était en vigueur lors de la définition de f. Il faut mémoriser cet environnement car la liste d'environnement en vigueur lors de l'évaluation d'une application f(x) n'est pas nécessairement la même, et c'est celle à la définition qui doit prévaloir dans une expression telle que :

```
let a = 1 in let f = fun x -> x + a in let a = 2 in f 3
```

Écrire une fonction `valeur : environnement -> expression -> valeur` qui calcule la valeur de l'expression fournie en deuxième paramètre à l'aide de la liste d'environnement fournie en premier paramètre. On aura par exemple :

```
valeur [] (expression (lexèmes
 "let f = fun x -> if x*(x-1) then 1 else f(x-1) + f(x-2) in f 5"));;
- : valeur = Int 8
```

Solutions des exercices

Exercice 1-1

```
let classe_3(a,b,c) =
  let (a',b') = if a <= b then (a,b) else (b,a) in
       if c <= a' then (c,a',b')
  else if c <= b' then (a',c,b')
  else                 (a',b',c)
;;
```

Il est effectué deux ou trois comparaisons pour classer la liste (a, b, c) ce qui est minimal car un algorithme effectuant au plus deux comparaisons ne peut retourner que quatre permutations de (a, b, c) parmi les six possibles.

Exercice 1-2

```
(* dit si n est parfait, n >= 2 *)
let parfait(n) =
  let s = ref 0 in (* somme des diviseurs déjà vus *)
  for d = 1 to n-1 do
    if n mod d = 0 then s := !s + d;
  done;
  n = !s  (* résultat = true ou false *)
;;
```

Exercice 1-3

L'algorithme de HÖRNER consiste à calculer par indices décroissants les nombres :

$$y_k = p_k + p_{k+1}x + \ldots + p_{n-1}x^{n-k-1}$$

selon la relation de récurrence :

$$y_n = 0, \qquad y_k = p_k + xy_{k+1} \quad \text{si } 0 \leqslant k < n.$$

Ici on veut calculer le *polynôme* $Q(X) = P(X + a)$ donc il suffit de calculer les polynômes successifs définis par :

$$Q_n(X) = 0, \qquad Q_k(X) = p_k + (X + a)Q_{k+1}(X) \quad \text{si } 0 \leqslant k < n,$$

et l'on a $Q = Q_0$.

```
(* calcule le polynôme q tel que q(x) = p(x+a) *)
let translate p a =
  let n = vect_length(p) in
  let q = make_vect n 0  in

  for k = n-1 downto 0 do
  (* ici on a : q(x) = p_{k+1} + p_{k+2}(x + a) + ... + p_{n-1}(x + a)^{n-k-2}  *)
    for i = n-k-1 downto 1 do q.(i) <- a * q.(i) + q.(i-1) done;
    q.(0) <- a * q.(0) + p.(k)
  done;

  q (* résultat *)
;;
```

Exercice 1-4

1. Décodage d'une chaîne de caractères : l'écriture décimale d'un entier naturel n est par définition la suite $(n_0, n_1, \dots, n_{p-1})$ d'entiers compris entre 0 et 9 telle que :

$$n = n_0 + 10n_1 + \dots + 10^{p-1}n_{p-1},$$

cette écriture étant unique à prolongation par des zéros près. Le calcul de n à partir de la suite $(n_0, n_1, \dots, n_{p-1})$ se ramène donc à l'évaluation en $x = 10$ du polynôme :

$$P(x) = n_0 + n_1x + \dots + n_{p-1}x^{p-1}$$

que l'on peut réaliser par exemple avec l'algorithme de HÖRNER.

```
let valeur(s) =
  let p = string_length(s) in
  let v = ref(0)           in
  for i = 0 to p-1 do v := !v*10 + valeur_chiffre(s.[i]) done;
  !v
;;
```

2. Comparaison de deux entiers : après élimination des zéros non significatifs la chaîne la plus longue représente le nombre le plus grand lorsque les longueurs des chaînes diffèrent. Si s et s' ont même longueur alors la comparaison des entiers représentés par s et s' coïncide avec la *comparaison lexicographique* de s et s' : le premier caractère différant entre s et s' décide du sens de la comparaison.

```
(* compare les nombres a et b représentés par s et s' *)
(* retourne 1 si a > b, -1 si a < b et 0 si a = b     *)
let compare s s' =
  let n  = string_length(s) and n' = string_length(s') in

  (* saute les zéros non significatifs *)
  let d = ref(0) and d' = ref(0) in
  while (!d < n)   & (s.[!d] = `0`)   do d  :=  !d  + 1 done;
  while (!d' < n') & (s'.[!d'] = `0`) do d' :=  !d' + 1 done;

  (* si les longueurs résiduelles sont différentes *)
  (* alors la comparaison est terminée             *)
  if      n - !d > n' - !d' then  1
  else if n - !d < n' - !d' then -1
  else begin

    (* comparaison lexicographique *)
    let i = ref(0) in
    while (!i < n - !d) & (s.[!d + !i] = s'.[!d' + !i])
    do i := !i + 1 done;
```

```
        if !i = n - !d then 0
        else if s.[!d + !i] > s'.[!d' + !i] then 1 else -1

      end
    ;;
```

Exercice 1-5

1. La décroissance de (x_n) et sa convergence vers $\sqrt{a}$ s'établissent de façon classique par étude de la fonction associée $f : x \longmapsto \frac{1}{2}(x + a/x)$. L'algorithme de HERON n'est pas réellement un algorithme car il implique de calculer une infinité de termes pour obtenir la limite. Par contre on peut en déduire un véritable algorithme de calcul approché de $\sqrt{a}$ à ε près en testant la condition :

$$(x_n - \varepsilon)^2 < a$$

et en arrêtant les calculs dès qu'elle est réalisée.

2. Si l'algorithme boucle alors il engendre une suite infinie (x_k) entière, strictement décroissante et positive, c'est absurde. Il existe donc un rang $n > 0$ tel que $x_{n+1} \geqslant x_n$. Par ailleurs $(x_{n-1} + a/x_{n-1})/2 \geqslant \sqrt{a}$ donc $x_n \geqslant \lfloor\sqrt{a}\rfloor$ et si $x_n > \lfloor\sqrt{a}\rfloor$ alors $x_n \geqslant \lfloor\sqrt{a}\rfloor + 1 > \sqrt{a}$ donc $x_{n+1} \leqslant f(x_n) < x_n$ ce qui est contradictoire avec la définition de n. Ainsi $x_n = \lfloor\sqrt{a}\rfloor$.

Exercice 1-6

Notons $T(n,p)$ le nombre cherché. On a :

$$\begin{aligned} &T(n,0) = T(n,n) = 1, \\ &T(n,p) = 1 + T(n-1,p) + T(n-1,p-1) \quad \text{si } 0 < p < n. \end{aligned}$$

Si l'on pose $T'(n,p) = T(n,p) + 1$ alors on obtient :

$$\begin{aligned} &T'(n,0) = T'(n,n) = 2, \\ &T'(n,p) = T'(n-1,p) + T'(n-1,p-1) \quad \text{si } 0 < p < n. \end{aligned}$$

Il en résulte par récurrence que $T'(n,p) = 2C_n^p$ et donc $T(n,p) = 2C_n^p - 1$. Ainsi la fonction `binome` fait environ deux fois plus de calculs que le résultat qu'elle fournit !

Exercice 1-7

Si l'on utilise la relation de récurrence de PASCAL alors il faut mémoriser les valeurs déjà calculées ce qui peut être fait en utilisant un vecteur. L'exercice **1-3** adapté au calcul de $(X+1)^n$ fournit une solution relativement efficace si l'on veut *tous* les coefficients C_n^p pour un n donné. Une solution pour calculer un seul coefficient C_n^p est d'utiliser une autre formule de récurrence, par exemple :

$$C_n^p = \frac{n!}{p!\,(n-p)!} = \frac{n}{p} C_{n-1}^{p-1}$$

ce qui donne le programme :

```
(* calcule C(n,p) pour 0 <= p <= n *)
let rec binome(n,p) = match p with
| 0 -> 1
| _ -> (n * binome(n-1,p-1))/p
;;
```

Ce programme présente toutefois un risque de débordement de capacité lorsque nC_{n-1}^{p-1} est supérieur au plus grand entier représentable en machine, ce qui peut se produire même si C_n^p est inférieur à ce plus grand entier. Noter par ailleurs que la formule `n * (binome(n-1,p-1)/p)` est incorrecte car il est possible que C_{n-1}^{p-1} ne soit pas divisible par p (par exemple si n est pair et $p = 2$). Le manuel de référence de CAML précise que l'expression ambiguë : `n * binome(n-1,p-1)/p` est interprétée comme `(n * binome(n-1,p-1))/p`.

Exercice 1-8

Soit $N(n, p)$ le nombre cherché. Compte tenu des sens uniques, on a :

$$N(0, p) = N(n, 0) = 1 \ ;$$
$$N(n, p) = N(n-1, p) + N(n-1, p-1) + N(n, p-1).$$

Ces relations ne sont pas directement exploitables car elle conduiraient à refaire de nombreuses fois les mêmes calculs (on montre que le nombre d'appels à une fonction récursive implémentant naïvement ces relations est égal à $\frac{3}{2}N(n, p) - \frac{1}{2}$ comme à l'exercice **1-6**). Pour éviter la répétition des calculs, on peut utiliser une matrice $(n+1) \times (p+1)$ pour conserver les valeurs déjà calculées, ou mieux, travailler sur une colonne que l'on « déplace » de $i = 0$ à $i = n$:

```
(* Calcule le nombre de chemins de A(0,0) à B(n,p) *)
let Nchemins(n,p) =
  let colonne = make_vect (p+1) 1 (* sauvegarde de N(i-1,0..p) *)
  and v = ref 1                   (* sauvegarde de N(i-1,j-1)  *)
  in

  for i = 1 to n do
    v := 1;
    for j = 1 to p do
      (* ici on a colonne.(0..j-1) = N(i,0..j-1),
                  colonne.(j..p)   = N(i-1,j..p),
                  !v = N(i-1,j-1).               *)
      let w = colonne.(j) in
      colonne.(j) <- colonne.(j) + !v + colonne.(j-1);
      v := w
    done
  done;

  colonne.(p) (* résultat *)
;;
```

Exercice 1-9

Soit $n = 2^p$ et P_n et P'_n les temps d'exécution cherchés. On a $P_{n+1} = P_n + Kna^2$ puisque x^n est codé sur na chiffres, d'où :

$$P_n = Ka^2(1 + 2 + \ldots + (n-1)) = K\frac{n(n-1)}{2}a^2 \approx \frac{Kn^2a^2}{2}.$$

Par ailleurs $P'_{2k} = P'_k + K(ka)^2$, donc :

$$P'_n = Ka^2(1 + 2^2 + 4^2 + \ldots + (2^{p-1})^2) = K\frac{n^2-1}{3}a^2 \approx \frac{Kn^2a^2}{3}.$$

Ainsi, avec cette hypothèse sur la durée d'une multiplication, `puiss_2` a un temps d'exécution de l'ordre de $\frac{2}{3}$ du temps d'exécution de `puiss_1`.

Remarque : la conclusion demeure pour n quelconque car on a les relations :

$$P'_{2n} = P'_n + Ka^2n^2, \qquad P'_{2n+1} = P'_n + Ka^2(n^2 + 2n),$$

et l'on obtient par récurrence :

$$\forall\ n \in \mathbf{N}^*,\ \frac{Ka^2(n-1)^2}{3} \leqslant P'_n \leqslant \frac{Ka^2(n+1)^2}{3}.$$

Exercice 1-10

Notons $n = n_0 2^0 + n_1 2^1 + \ldots + n_{k-1}2^{k-1}$ l'écriture binaire de n. On calcule de proche en proche les quantités :

$$y_i = x^{n_0 2^0 + n_1 2^1 + \ldots + n_{i-1}2^{i-1}} \qquad \text{et} \quad z_i = x^{2^i}$$

par les relations :

$$y_0 = 1,\ z_0 = x, \quad z_{i+1} = z_i^2,\ y_{i+1} = y_i z_i^{n_i}.$$

```
let puiss_3(x,n) =
  let y = ref 1
  and z = ref x
  and p = ref n in
  while !p > 0 do
    if !p mod 2 = 1 then y := !y * !z;
    z := !z * !z;
    p := !p / 2
  done;
  !y
;;
```

Exercice 1-11

Le programme puiss_2 de l'exercice **1-9** requiert 6 multiplications. En remarquant que $15 = 3 * 5$, on calcule en 2 multiplications $x^3 = y$ puis en 3 multiplications supplémentaires $y^5 = x^{15}$, soit en tout 5 multiplications. Est-ce le minimum ? Il suffit de chercher toutes les puissances de x que l'on peut obtenir en moins de 4 multiplications ce qui peut être fait à la main. On trouve les exposants 1,2,3,4,5,6,7,8,9,10,12 et 16. Donc le nombre minimal demandé est bien 5.

Exercice 1-12

1.
```
let rec fibonacci(n) =
   if n < 2 then 1 else fibonacci(n-1) + fibonacci(n-2);;
```

Le nombre d'appels à fibonacci pour calculer F_n est égal à $2F_n - 1$ (démonstration similaire à celle de l'exercice **1-6**).

2.
```
(* fibo(n) calcule le couple (Fn-1, Fn) *)
let rec fibo(n) = match n with
| 0 -> (0,1)
| 1 -> (1,1)
| _ -> let (u,v) = fibo(n-1) in (v,u+v)
;;

let fibonacci(n) = let (u,v) = fibo(n) in v;;
```

Il y a récursion simple, donc le nombre d'appels à fibo pour calculer F_n est égal à n pour $n \geqslant 1$.

3. Par récurrence sur p.

4. Avec $p = n$, $p = n - 1$, $p = n + 1$ on obtient les relations :

$$F_{2n} = F_n^2 + F_{n-1}^2, \quad F_{2n-1} = F_{n-1}(2F_n - F_{n-1}), \quad F_{2n+1} = F_n(F_n + 2F_{n-1}).$$

Notons $T(n) = (F_{n-1}, F_n)$. On a donc :

$$T_{2n} = (F_{n-1}(2F_n - F_{n-1}), F_n^2 + F_{n-1}^2), \quad T_{2n+1} = (F_n^2 + F_{n-1}^2, F_n(F_n + 2F_{n-1})),$$

c'est-à-dire que T_{2n} et T_{2n+1} se calculent facilement en fonction de T_n. On en déduit le programme suivant :

```
(* fibo(n) calcule le couple (Fn-1, Fn) *)
let rec fibo(n) = match n with
| 0 -> (0,1)
| 1 -> (1,1)
| _ -> let (u,v) = fibo(n/2) in
       if n mod 2 = 0 then (u*(2*v - u), u*u + v*v)
                      else (u*u + v*v, v*(v + 2*u))
;;
```

Soit T_n le nombre d'appels à fibo pour calculer F_n. On a $T_n = 1 + T_{\lfloor n/2 \rfloor}$ et $T_1 = 1$, d'où $T_n = 1 + \lfloor \log_2 n \rfloor$.

Exercice 1-13

1. Si x est pair strictement supérieur à 1 alors on a $f(x) = 2f(x/2)$ et $x/2 < x$. Si x est impair strictement supérieur à 1 alors on a $f(x) = 1 + 2f((x+1)/2)$ et $(x+1)/2 < x$. Donc dans les deux cas récursifs on rappelle f en une ou deux étapes avec un argument strictement inférieur à x ; ceci prouve la terminaison du calcul de $f(x)$ par récurrence sur x.

2. Soit $g(x) = f(x) + x$. En remplaçant $f(x)$ par $g(x) - x$ dans la définition de f et en simplifiant on obtient un programme calculant g :

```
let rec g(x) =
   if x <= 1 then 1+x
   else if x mod 2 = 0 then 2*g(x/2)
   else g(x+1)
;;
```

   et il est immédiat de constater que $g(x)$ est toujours une puissance de 2.

3. Par récurrence sur x on montre pour $x > 0$ que $g(x)$ est la première puissance de 2 supérieure ou égale à $2x$ et $f(x) = g(x) - x$ est le complément à additionner à x pour obtenir cette puissance de 2.

Exercice 1-14

1.
```
(* calcule le ppcm des éléments du vecteur a      *)
(* on suppose que a contient au moins un élément. *)
let ppcm_i a =
  let p = ref a.(0) in
  for i = 1 to vect_length(a) - 1 do p := ppcm2(!p,a.(i)) done;
  !p
;;
```

2.
```
(* calcule le ppcm des éléments a.(i)..a.(j) *)
(* on suppose i <= j.                        *)
let rec ppcm_rec a i j =
  if      j = i   then a.(i)
  else if j = i+1 then ppcm2(a.(i),a.(j))
  else let k = (i+j)/2 in ppcm2(ppcm_rec a i     k,
                                ppcm_rec a (k+1) j)
;;

let ppcm_dr(a) = ppcm_rec a 0 (vect_length(a)-1);;
```

Performances : `ppcm_i` effectue $n-1$ appels à `ppcm2` pour un vecteur de longueur n. Pour `ppcm_dr`, si $T(n)$ est le nombre d'appels à `ppcm2` effectués, alors on a :

$$T(1) = 0, \qquad T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \text{ si } n \geqslant 2.$$

On constate immédiatement par récurrence que $T(n) = n - 1$ donc les algorithmes sont de performances identiques si l'on ne compte que l'opération `ppcm2`. `ppcm_dr` est probablement plus lente que `ppcm_i` compte tenu des opérations non comptées (division de `a` en 2, gestion des appels récursifs) et certainement plus compliquée à écrire et à lire ...

Exercice 1-15

Soient $A(x) = a_0 + a_1x + a_2x^2$ et $B(x) = b_0 + b_1x + b_2x^2$: il s'agit de déterminer les cinq coefficients du polynôme $C(x) = A(x)B(x)$, ce qui peut se faire en calculant les valeurs de C en cinq points et en calculant le polynôme de LAGRANGE associé. Par exemple, en prenant les points 0, 1, -1, 2 et -2 on obtient :

$$\begin{aligned}
C(x) &= c_0 + c_1x + c_2x^2 + c_3x^3 + c_4x^4\\
&= \frac{C(0)}{4}(x^4 - 5x^2 + 4)\\
&\quad - \frac{C(1)}{6}(x^4 + x^3 - 4x^2 - 4x)\\
&\quad - \frac{C(-1)}{6}(x^4 - x^3 - 4x^2 + 4x)\\
&\quad + \frac{C(2)}{24}(x^4 + 2x^3 - x^2 - 2x)\\
&\quad + \frac{C(-2)}{24}(x^4 - 2x^3 - x^2 + 2x),
\end{aligned}$$

d'où :

$$\begin{aligned}
c_0 &= C(0)\\
c_1 &= \frac{2}{3}(C(1) - C(-1)) + \frac{1}{12}(C(-2) - C(2))\\
c_2 &= -\frac{5}{4}C(0) + \frac{2}{3}(C(1) + C(-1)) - \frac{1}{24}(C(2) + C(-2))\\
c_3 &= \frac{1}{6}(C(-1) - C(1)) + \frac{1}{12}(C(2) - C(-2))\\
c_4 &= \frac{1}{4}C(0) - \frac{1}{6}(C(1) + C(-1)) + \frac{1}{24}(C(2) + C(-2)),
\end{aligned}$$

avec :

$$\begin{aligned}
C(0) &= a_0b_0\\
C(1) &= (a_0 + a_1 + a_2)(b_0 + b_1 + b_2)\\
C(-1) &= (a_0 - a_1 + a_2)(b_0 - b_1 + b_2)\\
C(2) &= (a_0 + 2a_1 + 4a_2)(b_0 + 2b_1 + 4b_2)\\
C(-2) &= (a_0 - 2a_1 + 4a_2)(b_0 - 2b_1 + 4b_2).
\end{aligned}$$

On en déduit un algorithme calculant le produit de deux polynômes de longueurs inférieures ou égales à $3n$ en cinq multiplications de polynômes de longueurs inférieures ou égales à n :

Soient A *et* B *les polynômes à multiplier. Les décomposer sous la forme :*

$$A = A_0 + x^n A_1 + x^{2n} A_2, \qquad B = B_0 + x^n B_1 + x^{2n} B_2$$

avec $\deg(A_i) < n$, $\deg(B_i) < n$. *Calculer les produits :*

$$\begin{aligned}
P_0 &= A_0 B_0 \\
P_1 &= (A_0 + A_1 + A_2)(B_0 + B_1 + B_2) \\
P_2 &= (A_0 - A_1 + A_2)(B_0 - B_1 + B_2) \\
P_3 &= (A_0 + 2A_1 + 4A_2)(B_0 + 2B_1 + 4B_2) \\
P_4 &= (A_0 - 2A_1 + 4A_2)(B_0 - 2B_1 + 4B_2).
\end{aligned}$$

Alors $AB = C_0 + x^n C_1 + x^{2n} C_2 + x^{3n} C_3 + x^{4n} C_4$ *avec :*

$$\begin{aligned}
C_0 &= P_0 \\
C_1 &= \frac{1}{12}(8(P_1 - P_2) + (P_4 - P_3)) \\
C_2 &= \frac{1}{24}(-30P_0 + 16(P_1 + P_2) - (P_3 + P_4)) \\
C_3 &= \frac{1}{12}(2(P_2 - P_1) + (P_3 - P_4)) \\
C_4 &= \frac{1}{24}(6P_0 - 4(P_1 + P_2) + (P_3 + P_4)).
\end{aligned}$$

Si les polynômes A et B sont à coefficients entiers alors on vérifie aisément que les divisions par 12 ou 24 dans les expressions de C_1, C_2, C_3 et C_4 sont des divisions exactes, donc le calcul de AB peut être conduit sans fractions. En utilisant cet algorithme de multiplication récursivement pour calculer $P_0, \dots, P_5$, on obtient un algorithme de multiplication de polynômes effectuant $5^{\lceil \log_3(n) \rceil}$ multiplications de coefficients pour calculer le produit de polynômes de longueurs inférieures ou égales à n. La méthode de KNUTH calcule le même produit en $3^{\lceil \log_2(n) \rceil}$ multiplications de coefficients, et on a :

$$\begin{aligned}
5^{\lceil \log_3(n) \rceil} &\leqslant 5^{1+\log_3(n)} = 5n^{\log_3(5)}, \\
3^{\lceil \log_2(n) \rceil} &\geqslant 3^{\log_2(n)} = n^{\log_2(3)},
\end{aligned}$$

avec $\log_3(5) \approx 1.47 < \log_2(3) \approx 1.58$, donc cette méthode de multiplication récursive est asymptotiquement meilleure que celle de KNUTH. Toutefois, les calculs à effectuer sont plus complexes et elle ne bat expérimentalement la méthode de KNUTH que pour de grandes valeurs de n, ce qui la rend de peu d'utilité pratique La multiplication par transformation de FOURIER rapide (voir le problème « Interpolation de LAGRANGE et multiplication rapide ») bat ces deux algorithmes de multiplication dès n dépasse quelques milliers).

Exercice 2-1

Version vecteur :

```
(* renvoie le plus grand élément de v *)
let maximum v =
    if vect_length(v) = 0 then failwith "vecteur vide"
    else begin
      let m = ref v.(0) in
      for i = 1 to vect_length(v)-1 do m := max !m v.(i) done;
      !m
    end
;;
```

Version liste chaînée :

```
(* renvoie le plus grand élément de l *)
let rec maximum l = match l with
| []          -> failwith "liste vide"
| [a]         -> a
| a :: suite -> max a (maximum suite)
;;
```

Pour les listes chaînées, on peut obtenir une fonction récursive terminale (pouvant s'exécuter en mémoire constante) en ajoutant un deuxième paramètre contenant le maximum du début de la liste :

```
let rec maxi2 m l = match l with
| []        -> m
| a::suite -> maxi2 (max a m) suite
;;

let maximum l = match l with
| []          -> failwith "liste vide"
| a :: suite -> maxi2 a suite
;;
```

Exercice 2-2

```
(* applique f à chaque élément de l en commençant par la fin *)
let rec do_list_rev f l = match l with
| []        -> ()
| a::suite -> do_list_rev f suite; let _ = f(a) in ()
;;
```

Exercice 2-3

Avec concaténation en queue :

```
let rec rev_q l = match l with
| []       -> []
| a::suite -> rev_q(suite) @ [a]
;;
```

Complexité : on considère que la concaténation en queue de deux listes chaînées a un temps d'exécution égal à $a\ell$ où ℓ est la longueur de la première liste et a une constante. Soit $T(n)$ le temps de calcul de l'image miroir d'une liste de n éléments à l'aide de la fonction `rev_q`. On a l'équation de récurrence :

$$T(n) = T(n-1) + a(n-1).$$

D'où :

$$T(n) = T(0) + a(0 + 1 + \ldots + (n-1)) = T(0) + a\frac{n(n-1)}{2} \approx \frac{an^2}{2}.$$

Avec concaténation en tête :

```
(* calcule "miroir(l) @ r" par concaténations en tête *)
let rec rev1 l r = match l with
| []       -> r
| a::suite -> rev1 suite (a::r)
;;
let rev_t l = rev1 l [];;
```

Complexité : le temps de calcul de `rev1 l r` est manifestement proportionnel à la longueur de `l` donc pour une liste à n éléments le temps de calcul de `rev_t l` est proportionnel à n.

Exercice 2-4

1. Rotation d'une liste chaînée L. On peut découper L en deux listes :

$$L_1 = (a_0, \ldots, a_{k-1}), \qquad L_2 = (a_k, \ldots, a_{n-1})$$

et l'on a $L' = L_2 \mathbin{@} L_1$. Le découpage s'effectue en temps proportionnel à k et la concaténation s'effectue en temps proportionnel à $n - k$, donc la rotation a une complexité proportionnelle à n.

Rotation d'un vecteur V dans un deuxième vecteur W.

Pour $i = 0..n-1$ *faire* $W.(i) \longleftarrow V.((i+k) \bmod n)$ *fin.*

Là aussi la complexité est proportionnelle à n.

2. Rotation d'un vecteur V sur place. On peut effectuer une rotation d'une position par l'algorithme :

$x \longleftarrow V.(0)$
Pour $i = 0..n-2$ *faire* $V.(i) \longleftarrow V.(i+1)$ *fin pour.*
$V.(n-1) \longleftarrow x.$

La rotation sur place de k positions peut alors être effectuée par k rotations d'une position avec une complexité totale proportionnelle à kn.

3. Rotation TOURNESOL. Soit σ la permutation à effectuer. σ admet une décomposition en cycles à supports disjoints :

$$\sigma = c_1 \circ c_2 \circ \ldots \circ c_p,$$

où chaque c_i est un cycle de la forme :

$$c_i = (j, \quad (j+k) \bmod n, \quad (j+2k) \bmod n, \quad \ldots)$$

pour un certain entier j que l'on peut supposer compris entre 0 et $k-1$. Une itération de la boucle `while` :

```
let temp = v.(!i0)
...
v.(!i) <- temp;
compte := !compte + 1;
```

effectue sur place la permutation circulaire (`!i0`, $($`!i0` $+ k) \bmod n$, $\ldots$) donc l'un des cycles c_i composant σ, plus précisément celui contenant `!i0`, tout en cumulant dans `compte` le nombre d'éléments déplacés. A la fin de cette boucle `while` on a donc déplacé au moins n éléments et il reste à vérifier que les cycles effectués sont distincts pour prouver que chaque élément a été déplacé une et une seule fois.

Notons d le pgcd de n et k. Alors deux entiers i et j appartiennent au même cycle si et seulement s'ils sont congrus modulo d donc le nombre d'éléments d'un cycle est n/d et les cycles contenant $0, 1, \ldots, d-1$ sont distincts. Ce sont exactement les cycles réalisés par le programme TOURNESOL puisqu'on atteint ainsi $d \times (n/d) = n$ éléments déplacés. Ainsi le programme est valide.

Intérêt : le temps de réalisation d'un cycle de longueur n/d est de la forme $a + b(n/d)$ où a et b sont des constantes (a est le temps d'initialisation et b est la durée d'exécution d'un cycle), donc le temps de réalisation de la rotation complète est de la forme $ad + bn + c = O(n)$ au lieu λkn pour l'algorithme donné à la question précédente.

Exercice 2-5

Test de présence :

```
let rec contient f l = match l with
| []       -> false
| a::suite -> (f a) or (contient f suite)
;;
```

L'expression booléenne `(f a) or (contient f suite)` est évaluée en CAML comme :

```
if (f a) then true else (contient f suite)
```

c'est-à-dire que la partie `(contient f suite)` n'est pas évaluée si `(f a)` retourne `true`. Ainsi, la fonction `contient` s'arrête dès qu'un élément satisfaisant le prédicat `f` est trouvé, sans parcourir le reste de la liste.

Dernier élément vérifiant le critère : on peut calculer l'image miroir de `l` (cf. exercice **2-3**) puis chercher le premier élément de cette liste vérifiant le critère à l'aide du programme `cherche_liste` présenté à la section **2-4**. On peut aussi parcourir la liste en conservant le dernier élément trouvé jusque là :

```
(* retourne le dernier élément de l vérifiant f *)
(* ou à défaut x                                *)
let rec dernier_1 f x l = match l with
| []       -> x
| a::suite -> if f(a) then dernier_1 f a suite
                      else dernier_1 f x suite
;;

(* retourne le dernier élément de l vérifiant f *)
let rec dernier f l = match l with
| []       -> failwith "non trouvé"
| a::suite -> if f(a) then dernier_1 f a suite
                      else dernier   f   suite
;;
```

Exercice 2-6
Une méthode simple est de comparer le début de A et B : s'ils coïncident alors B est une sous-liste de A au rang 0. Sinon on cherche si B est une sous-liste de la queue de A et on adapte le rang éventuellement trouvé.

Version liste chaînée :

```
(* dit si le début de a est égal à la liste b *)
let rec est_début a b = match (a,b) with
| (_,[])         -> true
| ([],_)         -> false
| (x::sa, y::sb) -> (x = y) & (est_début sa sb)
;;

(* cherche si b est une sous-liste de a et renvoie  *)
(* le premier rang d'apparition de b dans a si oui. *)
(* Sinon, renvoie -1.                               *)
let rec cherche_sous_liste a b = match a with
| []       -> -1
| _::suite -> if est_début a b then 0
              else let i = cherche_sous_liste suite b in
                   if i >= 0 then i+1 else -1
;;
```

Version vecteur :

```
(* cherche si b est un sous-vecteur de a et renvoie *)
(* le premier rang d'apparition de b dans a si oui. *)
(* Sinon, renvoie -1. On suppose b non vide.        *)
let cherche_sous_vect a b =

    let la = vect_length(a)
    and lb = vect_length(b)
    and i  = ref 0
    and trouve = ref false in

    while (!i+lb <= la) & (not !trouve) do
      let j = ref 0 in
      while (!j < lb) & (a.(!i + !j) = b.(!j)) do j := !j+1 done;
      trouve := (!j = lb);
      i := !i+1
    done;

    if !trouve then !i-1 else -1
;;
```

Exercice 2-7

```
(* vecteur circulaire avec indice de début et longueur *)
type 'a cercle = {v:'a vect; mutable d:int; mutable l:int};;

(* insère x en tête de c *)
let ins_tete c x =
  let n = vect_length(c.v) in
  if c.l >= n then failwith "plus de place"
  else begin
    c.d <- (c.d + n - 1) mod n;
    c.v.(c.d) <- x;
    c.l <- c.l + 1
  end
;;

(* insère x en queue de c *)
let ins_queue c x =
  let n = vect_length(c.v) in
  if c.l >= n then failwith "plus de place"
  else begin
    c.v.((c.d + c.l) mod n) <- x;
    c.l <- c.l + 1
  end
;;
```

```
(* extrait la tête de c *)
let extr_tete c =
  let n = vect_length(c.v) in
  if c.l = 0 then failwith "liste vide"
  else begin
    let x = c.v.(c.d) in
    c.d <- (c.d + 1) mod n;
    c.l <- c.l - 1;
    x
  end
;;

(* extrait le dernier élément de c *)
let extr_dernier c =
  let n = vect_length(c.v) in
  if c.l = 0 then failwith "liste vide"
  else begin
    c.l <- c.l - 1;
    c.v.((c.d + c.l) mod n)
  end
;;
```

Remarques :

– `ins_tete` décrémente l'indice de début par l'instruction :

```
c.d <- (c.d + n - 1) mod n;
```

et non `(c.d - 1) mod n` car l'opérateur `mod` en CAML retourne un résultat négatif si son premier argument est négatif.

– L'insertion d'un élément en tête ou en queue peut échouer si le vecteur `c.v` est entièrement rempli. On peut remédier à ce problème en recopiant le vecteur `c.v` dans un vecteur plus grand en cas de débordement :

```
type 'a cercle = {mutable v:'a vect; mutable d:int; mutable l:int};;

(* allonge c.v par doublement de taille *)
let allonge c = c.v <- concat_vect c.v c.v;;
```

Le temps d'une insertion n'est alors plus constant, mais le temps cumulé T_n de n insertions reste asymptotiquement proportionnel à n car `concat_vect v1 v2` s'exécute en un temps asymptotiquement proportionnel à la somme des longueurs de `v1` et `v2` donc on a l'équation de récurrence :

$$T_{2n} = T_n + \Theta(n)$$

qui a pour solution $T_n = \Theta(n)$ (cf. section **6-3**).

– Une autre possibilité d'implémentation des listes à double entrée est d'utiliser deux listes chaînées L_1 et L_2 telles que $L = L_1$ @ $\mathrm{miroir}(L_2)$ (cf. exercice **2-3**). Dans ce cas les opérations d'insertion et d'extraction peuvent être effectuées de manière *fonctionnelle*, c'est-à-dire sans perdre la liste initiale :

```
type 'a dbliste = {début:'a list; fin:'a list};;

let ins_tete  l x = {début = x::l.début; fin = l.fin};;
let ins_queue l x = {début = l.début; fin = x::l.fin};;

let extr_tete(l) = match l.début with
| x::suite -> (x, {début=suite; fin=l.fin})
| []       -> match rev(l.fin) with
              | x::suite -> (x, {début=suite; fin=[]})
              | []       -> failwith "liste vide"
;;

let extr_dernier(l) = match l.fin with
| x::suite -> (x, {début=l.début; fin=suite})
| []       -> match rev(l.début) with
              | x::suite -> (x, {début=[]; fin=suite})
              | []       -> failwith "liste vide"
;;
```

Avec cette implémentation, les opérations d'insertion en tête et en queue sont effectuées en temps constant, une suite de k extractions à la même extrémité s'effectue dans le pire des cas en un temps asymptotiquement proportionnel à la longueur n de la liste, et une suite de k extractions sans contrainte d'extrémité s'effectue dans le pire des cas en un temps asymptotiquement proportionnel à kn.

Exercice 3-1

```
(* insertion sans répétition dans un vecteur trié *)
let insère_vect compare v x =
  let n = vect_length(v)
  and i = ref(0)
  and c = ref(PLUSPETIT) in

  (* cherche à quelle place insérer x *)
  while (!i < n) & (!c = PLUSPETIT) do
    c := compare v.(!i) x;
    i := !i + 1
  done;

  (* élément déjà présent ? *)
  if !c = EQUIV then copy_vect v

  (* sinon effectue l'insertion *)
  else begin
    let w = make_vect (n+1) x in
    for j = 0    to !i-2 do w.(j)   <- v.(j) done;
    for j = !i-1 to n-1  do w.(j+1) <- v.(j) done;
    w
  end
;;
```

```
(* insertion sans répétition dans une liste chaînée triée *)
let rec insère_liste compare l x = match l with
| []      -> [x]
| a::suite -> match compare a x with
    | PLUSPETIT -> a :: (insère_liste compare suite x)
    | EQUIV     -> l
    | _         -> x :: l
;;
```

Exercice 3-2

```
(* fusion sans répétition de deux listes chaînées *)
let rec fusion_sr compare l1 l2 = match (l1,l2) with
| ([],_) -> l2
| (_,[]) -> l1
| (a::s1, b::s2) -> match compare a b with
    | EQUIV     -> fusion_sr compare l1 s2
    | PLUSPETIT -> a :: (fusion_sr compare s1 l2)
    | _         -> b :: (fusion_sr compare l1 s2)
;;

(* fusion sans répétition de deux vecteurs *)
let fusion_vect_sr compare v1 v2 =

  let v = make_vect (vect_length(v1)+vect_length(v2)) v1.(0) in
  let i = ref 0    (* indice de v1 *)
  and j = ref 0    (* indice de v2 *)
  and k = ref 0    (* indice de v  *)
  in

  while (!i < vect_length(v1)) & (!j < vect_length(v2)) do
  match compare v1.(!i) v2.(!j) with
  | EQUIV     -> v.(!k) <- v1.(!i); k:= !k+1; i:= !i+1; j:= !j+1
  | PLUSGRAND -> v.(!k) <- v2.(!j); k:= !k+1; j:= !j+1
  | _         -> v.(!k) <- v1.(!i); k:= !k+1; i:= !i+1
  done;

  (* ici, un des deux vecteurs est épuisé *)
  (* on recopie la fin de l'autre         *)
  while !i < vect_length(v1) do
     v.(!k) <- v1.(!i); k := !k+1; i := !i+1
  done;
  while !j < vect_length(v2) do
     v.(!k) <- v2.(!j); k := !k+1; j := !j+1
  done;

  sub_vect v 0 !k (* résultat *)
;;
```

Remarque : `fusion_vect_sr` initialise le vecteur résultat avec `v1.(0)` ce qui déclenche une erreur si `v1` est de longueur nulle. On peut corriger ce défaut en traitant à part le cas où l'un des vecteurs est vide et en retournant une copie de l'autre dans ce cas.

Exercice 3-3
La fusion sans répétition fournit la réunion de deux ensembles. Le calcul de l'intersection peut être conduit par un algorithme similaire :

```
let rec intersection compare e1 e2 = match (e1,e2) with
| ([],_) -> []
| (_,[]) -> []
| (a::s1, b::s2) -> match compare a b with
    | EQUIV     -> a :: (intersection compare e1 s2)
    | PLUSPETIT -> (intersection compare s1 e2)
    | _         -> (intersection compare e1 s2)
;;
```

La différence et la différence symétrique se traitent de manière analogue. Toutes ces opérations ont une complexité asymptotique $O(\ell_1 + \ell_2)$ où ℓ_1 et ℓ_2 sont les cardinaux des ensembles e_1 et e_2.

Exercice 3-4
Comme dans l'exercice **3-3**, une adaptation de l'algorithme de fusion donne le résultat :

```
let rec addition p q = match (p,q) with
| ([], _) -> q
| (_, []) -> p
| ((a,e)::p', (b,f)::q') ->
    if      e < f then (a,e) :: (addition p' q )
    else if e > f then (b,f) :: (addition p  q')
    else let c = a + b in
         if c = 0 then  addition p' q'
         else (c,e) :: (addition p' q')
;;
```

Exercice 3-5
Fusion de trois listes : on peut fusionner L_1 et L_2 dans une liste L puis fusionner L et L_3. Le nombre maximal de comparaisons est :

$$N_{\mathrm{comp}} = (\ell_1 + \ell_2 - 1) + (\ell_1 + \ell_2 + \ell_3 - 1) = 2\ell_1 + 2\ell_2 + \ell_3 - 2.$$

Il y a intérêt avec cette stratégie à prendre pour L_3 la liste la plus longue, mais si les longueurs des listes ne sont pas connues à l'avance, alors le temps de détermination de la liste la plus longue peut être plus coûteux que l'économie réalisée en temps de comparaisons.

Une autre possibilité est de fusionner les trois listes en parallèle suivant l'algorithme :

```
 renvoie le numéro de la liste contenant le plus petit élément
fonction minimum(L1,L2,L3 : liste)
  i ⟵ 0
  si L1 ≠ ∅ alors i ⟵ 1
  si L2 ≠ ∅ alors
    si i = 0 alors i ⟵ 2
    si i = 1 et si tête(L2) < tête(L1) alors i ⟵ 2
  si L3 ≠ ∅ alors
    si i = 0 alors i ⟵ 3
    si i = 1 et si tête(L3) < tête(L1) alors i ⟵ 3
    si i = 2 et si tête(L3) < tête(L2) alors i ⟵ 3
  retourner i
fin

 fusionne L1, L2, L3 dans L
fonction fusion3(L1,L2,L3 : liste)
  i ⟵ minimum(L1, L2, L3)
  si i = 0 alors []
  si i = 1 alors tête(L1) :: fusionne(queue(L1), L2, L3)
  si i = 2 alors tête(L2) :: fusionne(L1, queue(L2), L3)
  si i = 3 alors tête(L3) :: fusionne(L1, L2, queue(L3))
fin
```

Chaque élément placé dans la liste résultat donne lieu à au plus deux comparaisons (une seule pour l'avant dernier et aucune pour le dernier) d'où :

$$N_{\text{comp}} = 2(\ell_1 + \ell_2 + \ell_3) - 3.$$

Cet algorithme est donc moins bon.

Fusion de quatre listes : on peut envisager au moins quatre algorithmes.

a. Fusion de L_1 et L_2 dans L, fusion de L et L_3 dans L' et fusion de L' et L_4 avec $3\ell_1 + 3\ell_2 + 2\ell_3 + \ell_4 - 3$ comparaisons.

b. Fusion de L_1 et L_2 dans L, fusion de L_3 et L_4 dans L' puis fusion de L et L' avec $2\ell_1 + 2\ell_2 + 2\ell_3 + 2\ell_4 - 3$ comparaisons.

c. Fusion en parallèle comme pour trois listes, le minimum de 4 objets étant déterminé en au plus 3 comparaisons soit au total $3\ell_1 + 3\ell_2 + 3\ell_3 + 3\ell_4 - 6$ comparaisons.

d. Fusion en parallèle avec un calcul de minimum plus astucieux : supposons que l'on ait classé les têtes de L_1, L_2, L_3 et L_4. On connaît alors le minimum qui peut être inséré dans L, et on peut classer la nouvelle tête de la liste d'où vient ce minimum par rapport aux trois autres en seulement 2 comparaisons. On fusionne ainsi les quatre listes avec au maximum $2\ell_1 + 2\ell_2 + 2\ell_3 + 2\ell_4 - 3$ comparaisons.

Le meilleur algorithme parmi ceux envisagés pour fusionner quatre listes est *b* ou *d* avec une préférence pour *b* qui est plus simple à programmer. Plus généralement, on peut fusionner p listes triées de longueurs $\ell_1, \dots, \ell_p$ avec moins de $\lceil \log_2 p \rceil(\ell_1 + \dots + \ell_p)$ comparaisons en fusionnant les listes deux par deux, puis en fusionnant les listes obtenues deux par deux, et ainsi de suite.

Exercice 3-6

```
(* applique une passe du tri à bulles à l, retourne la liste *)
(* obtenue et l'indicateur fini qui dit si l était triée     *)
let rec bulles1 compare l fini = match l with
| []  -> [],  fini
| [a] -> [a], fini
| a::b::suite -> if compare a b = PLUSGRAND
   then let l,f = bulles1 compare (a::suite) false in (b::l),f
   else let l,f = bulles1 compare (b::suite) fini  in (a::l),f
;;

let rec bulles_liste compare l =
  let l',fini = bulles1 compare l true in
  if fini then l' else bulles_liste compare l'
;;
```

Remarque : à l'issue d'une passe du tri à bulles le dernier élément de la liste obtenue est le maximum de cette liste donc il est à la bonne place et peut être ignoré lors des passes suivantes. Cette optimisation pourrait être programmée en ajoutant un troisième paramètre à `bulles1` indiquant combien d'éléments restent à comparer.

Exercice 3-7

Soit $\mu(n)$ la moyenne cherchée. Considérons une liste $L = (a_0, \dots, a_n)$ à $n+1$ éléments et soient $L' = (a_0, \dots, a_{n-1})$, $L'' = (b_0, \dots, b_{n-1})$ la liste triée associée à L', et p la position finale de a_n dans la liste triée associée à L. Le tri à bulles de L effectue $N_{\mathrm{inv}}(L)$ échanges, et l'on a $N_{\mathrm{inv}}(L) = N_{\mathrm{inv}}(L') + n - p$. On en déduit la relation :

$$\mu(n+1) = \mu(n) + \frac{1}{K^{n+1}} \sum_{L} (n-p) = \mu(n) + \frac{1}{K^{n+1}} \sum_{L'} \sum_{a_n=0}^{K-1} (n-p).$$

Lorsque a_n varie de 0 à $K-1$, la liste L' étant fixée, on a :

$$\begin{aligned} n-p &= n && \text{si } 0 \leqslant a_n < b_0 \ ; \\ n-p &= n-1 && \text{si } b_0 \leqslant a_n < b_1 \ ; \\ &\dots \\ n-p &= 0 && \text{si } b_{n-1} \leqslant a_n < K. \end{aligned}$$

donc :

$$\begin{aligned} \sum_{a_n=0}^{K-1} (n-p) &= nb_0 + (n-1)(b_1 - b_0) + \dots + 1(b_{n-1} - b_{n-2}) \\ &= b_0 + \dots + b_{n-1} \\ &= a_0 + \dots + a_{n-1}. \end{aligned}$$

On a alors :

$$\frac{1}{K^{n+1}}\sum_{L'}\sum_{a_n=0}^{K-1}(n-p) = \frac{1}{K^{n+1}}\sum_{L'}(a_0+\ldots+a_{n-1}) = n\frac{K-1}{2K}$$

car $\frac{1}{K^n}\sum_{L'} a_i = \dfrac{K-1}{2}$ (valeur moyenne de a_i sur l'ensemble des listes L). Donc $\mu(n+1) = \mu(n) + n\dfrac{K-1}{2K}$ avec $\mu(1) = 0$, et finalement :

$$\mu(n) = n(n-1)\frac{K-1}{4K}.$$

Lorsque K est grand, $\frac{K-1}{4K} \approx \frac{1}{4}$ donc le nombre moyen d'échanges effectués par le tri à bulles d'un vecteur de longueur n est de l'ordre de $n^2/4$, ce qui implique que la complexité en moyenne du tri à bulles est quadratique (sous l'hypothèse d'équidistribution de tous les vecteurs de longueur n).

Exercice 3-8

1. Non en général. Par exemple si a_0 est le plus grand élément de L et si $(a_1,\ldots,a_{n-1})$ est triée par ordre croissant, alors L est 1-presque triée à gauche mais pas à droite.

2. La complexité du tri à bulles sur une liste L est proportionnelle au nombre d'inversions de L et, lorsque L est p-presque triée à gauche, ce nombre d'inversions est majoré par np. On a le même résultat pour une liste p-presque triée à droite.

Exercice 3-9

On remarque que `merge` effectue la fusion de deux listes chaînées suivant la relation d'ordre définie par la fonction booléenne `order` (`order x y` renvoie `true` si et seulement si $x \preccurlyeq y$).

`sort` fonctionne de la manière suivante : on constitue par `initlist l` une liste de sous-listes de l triées et de longueur 2, la dernière étant éventuellement de longueur 1. Puis on fusionne ces sous-listes deux par deux avec `merge2`, et on recommence tant qu'il reste au moins deux sous-listes (fonction `mergeall`). Par récurrence sur la profondeur de récursion, on montre que la liste renvoyée par `mergeall` est constituée de sous-listes triées formant une partition de la liste initiale l. La récursion est finie car le nombre de sous-listes renvoyées décroît strictement à chaque appel jusqu'à ce qu'il ne reste plus qu'une seule sous-liste qui est la liste triée associée à l.

Exercice 3-10

```
(* Cherche la plus grande séquence croissante initiale   *)
(* de la liste l et renvoie cette séquence et le reste   *)
(* de la liste.                                          *)
let rec cherche_séquence compare l = match l with
| []  -> ([],  [])
| [a] -> ([a], [])
| a::(b::suite as l') ->
    if compare a b = PLUSGRAND then [a],l'
    else let (s,r) = cherche_séquence compare l' in  (a::s, r)
;;

(* fusionne les sous-séquences croissantes de l deux par deux *)
let rec fusionne_séquences compare l = match l with
| [] -> []
| _  -> let (l1, r1) = cherche_séquence compare l  in
        let (l2, r2) = cherche_séquence compare r1 in
        (fusion compare l1 l2) @ (fusionne_séquences compare r2)
;;

(* code principal : on fusionne les séquences deux par *)
(* deux tant qu'il y en a plusieurs                    *)
let rec fusion_naturelle compare l =
   let (l1,r1) = cherche_séquence compare l in
    if r1 = [] then l
    else let (l2,r2) = cherche_séquence compare r1 in
    fusion_naturelle compare
       ((fusion compare l1 l2) @ (fusionne_séquences compare r2))
;;
```

On établit séparément la validité de `cherche_séquence`, `fusionne_séquences` et `fusion_naturelle` par récurrence sur la longueur de la liste passée en argument. La complexité asymptotique de `cherche_séquence` est proportionnelle à la taille de la séquence initiale trouvée donc la complexité des opérations :

```
let (l1, r1) = cherche_séquence compare l  in
let (l2, r2) = cherche_séquence compare r1 in fusion l1 l2
```

est proportionnelle à la somme des tailles des listes `l1` et `l2`. Il en est de même pour la concaténation en queue dans :

```
(fusion compare l1 l2) @ (fusionne_séquences compare r2)
```

et donc `fusionne_séquences compare l` a une complexité asymptotique $O(p)$ où p est la taille de l. Enfin, `fusion_naturelle` itère `fusionne_séquences` tant qu'il reste au moins deux séquences croissantes maximales, et le nombre de telles séquences est divisé par deux ou plus à l'arrondi supérieur près à chaque itération. Donc le nombre d'itérations est $O(\ln n)$ et le temps d'exécution de `fusion_naturelle` est $O(n \ln n)$.

Remarque : plutôt que de rechercher systématiquement les séquences croissantes à chaque appel à `fusionne_séquences`, on pourrait découper dans un premier temps l en listes de séquences croissantes, puis les fusionner deux par deux comme dans l'exercice **3-9**.

Exercice 3-11

```
(* découpe l en deux listes séparées par le pivot p *)
let rec découpe compare p l = match l with
| []       -> [],[]
| a::suite ->
     let (l1,l2) = découpe compare p suite in
     if (compare a p) = PLUSPETIT then (a::l1,l2) else (l1,a::l2)
;;

let rec tri compare l = match l with
| []       -> []
| a::suite -> let (l1,l2) = découpe compare a suite in
              (tri compare l1) @ (a :: (tri compare l2))
;;
```

Remarques :

– Cet algorithme de tri est stable, car `découpe` place dans la liste de droite les éléments équivalents à p en conservant leurs dispositions relatives.

– L'usage de la concaténation en queue, `@`, n'est pas pénalisant ici car la complexité de cette concaténation dans le calcul de :

```
(tri compare l1) @ (a :: (tri compare l2))
```

est $O(n)$ où n est la taille de l et `découpe` a une complexité $\Theta(n)$ donc le coût temporel de `découpe` suivi de `@` reste $\Theta(n)$. On peut toutefois éviter la concaténation en queue avec le code suivant :

```
(* trie la liste l et la concatène à m *)
let rec tri compare l m = match l with
| []       -> m
| a::suite -> let (l1,l2) = découpe compare a suite in
              tri compare l1 (a :: (tri compare l2 m))
;;
```

Exercice 3-12

```
let tri_rapide_itératif compare v =

  (* empile le premier appel *)
  let pile = ref [(0,vect_length(v)-1)] in

  while !pile <> [] do
    let (a,b) = hd(!pile) in  (* bornes du vecteur à traiter *)
    if a < b
```

```
    then let c = segmentation compare v a b in
         (* empile le plus grand sous-vecteur en premier *)
         if c-a < b-c then pile := (a,c-1)::(c+1,b)::(tl !pile)
                      else pile := (c+1,b)::(a,c-1)::(tl !pile)
    else pile := tl !pile
  done
;;
```

Taille maximale de la pile : soient $\ell_1, \dots, \ell_i$ les longueurs des sous-vecteurs empilés à la fin d'une itération de la boucle `while`. Si $\ell_i \geqslant 2$ alors le sous-vecteur de longueur ℓ_i est remplacé lors de l'itération suivante par deux sous-vecteurs de longueurs ℓ'_i et ℓ'_{i+1} avec $\ell'_i \geqslant \ell'_{i+1}$ et $\ell'_i + \ell'_{i+1} = \ell_i - 1$. Si $\ell_i \leqslant 1$ alors le sous-vecteur de longueur ℓ_i est retiré de la pile. On a donc par récurrence :

$$\ell_i + \ell_{i-1} + \ldots + \ell_{i-p} \leqslant \ell_{i-p-1} - p \leqslant \ell_{i-p-1}$$

pour $0 \leqslant p < i$, d'où l'on déduit :

$$n \geqslant \ell_1 + \ldots + \ell_i \geqslant 2(\ell_2 + \ldots + \ell_i) \geqslant \ldots \geqslant 2^{i-1}\ell_i.$$

Si $i > \log_2 n$ alors $\ell_i < 2$ et l'itération suivante de la boucle `while` ne provoquera pas de nouvel empilement. Le nombre de sous-vecteurs empilés est donc toujours majoré par $1 + \log_2 n$.

Exercice 4-1
On raisonne par récurrence sur la longueur d'une formule postfixe f :

– si f est de longueur 1 alors f est un nombre et il n'y a aucune réduction à effectuer ;

– si f est de longueur au moins égale à 2 supposons qu'il y ait deux réductions différentes possibles, par exemple :

$$\begin{array}{lll} (1) & x \quad \varphi \longrightarrow & \varphi[x] \\ (2) & y \quad \psi \longrightarrow & \psi[y] \end{array}$$

où x, y sont des nombres et φ, ψ des opérateurs unaires (le fait que φ et ψ soient des opérateurs unaires ne restreint pas la généralité du raisonnement qui suit). On note f_1 la formule déduite de f par application de (1), f_2 la formule déduite de f par application de (2), f_3 la formule déduite de f_1 par application de (2) et f_4 la formule déduite de f_2 par application de (1). Comme les sous-listes $[x,\ \varphi]$ et $[y,\ \psi]$ ne se chevauchent pas dans f, on a $f_3 = f_4$. Par hypothèse de récurrence f_1 et f_2 admettent des valeurs bien définies, et comme f_3 est obtenue à la fois par réduction de f_1 et de f_2, les valeurs de f_1 et f_3 sont égales de même que les valeurs de f_2 et f_3. On en déduit que f_1 et f_2 ont même valeur ce qu'il fallait démontrer.

Exercice 4-2

Ceci impose de prévoir plusieurs types de valeurs dont le type booléen et les opérations associées pour pouvoir évaluer *condition*, et donc de tester à chaque opération si les opérandes ont le type requis. Par ailleurs, la forme postfixe associée à une formule conditionnelle est :

$$exp_1,\ exp_2,\ condition,\ \texttt{si}$$

à l'ordre des termes près, et donc les deux expressions exp_1 et exp_2 devront être évaluées avant de savoir laquelle est à retenir, ce qui ne permet pas d'évaluer une formule conditionnelle telle que :

$$si\ x \neq 0\ alors\ \frac{\sin x}{x}\ sinon 1.$$

Le langage POSTSCRIPT résout ce problème en introduisant un type de valeur *procédure* constitué d'une formule postfixe représentant les calculs que doit exécuter la procédure. Une procédure placée sur la pile est évaluée par des opérateurs spécialisés tels `exec`, `loop` et `if`.

Exercice 4-3

Une méthode simple consiste à évaluer l'image miroir de cette formule à l'aide de l'algorithme d'évaluation postfixe en permutant les opérandes lors de l'évaluation d'une opération binaire.

Exercice 4-4

```
let évalue(f) =

  (* piles des valeurs et des opérateurs *)
  let pv = ref []
  and po = ref [] in

  (* empile une valeur en effectuant les opérations *)
  (* unaires en attente                             *)
  let rec pushv x = match !po with
  | OP_UNAIRE(f)::so -> po := so; pushv(f x)
  | _                -> pv := x :: !pv
  in

  (* dépile une valeur *)
  let popv() = match !pv with
  | []    -> failwith "formule incorrecte"
  | x::sv -> pv := sv; x
  in

  (* effectue les opérations binaires de priorité   *)
  (* supérieure ou égale à p                        *)
  let rec réduit(p) = match !po with
  | OP_BINAIRE(q,f)::so when q >= p ->
```

```
        let y = popv() in
        let x = popv() in
        po := so;
        pushv (f x y);
        réduit(p)

    | _ -> ()
    in

    (* empile un opérateur ou une parenthèse en        *)
    (* effectuant les opérations binaires prioritaires *)
    let pusho lexeme = match lexeme with
    | VALEUR(_)           -> failwith "cas impossible"
    | OP_UNAIRE(_)        -> po := lexeme :: !po
    | OP_BINAIRE(p,_)     -> réduit(p); po := lexeme :: !po
    | PARENTHÈSE_OUVRANTE -> po := lexeme :: !po
    | PARENTHÈSE_FERMANTE ->
       réduit(-1);
       match !po with
       | PARENTHÈSE_OUVRANTE::so -> po := so; pushv(popv())
       | _ -> failwith "formule incorrecte"
    in

    (* traitement d'un lexème *)
    let traitement(lexeme) = match lexeme with
    | VALEUR(x) -> pushv x
    | _         -> pusho lexeme
    in

    (* parcours de la formule et extraction du résultat *)
    traitement PARENTHÈSE_OUVRANTE;
    do_list traitement f;
    traitement PARENTHÈSE_FERMANTE;
    match (!pv, !po) with
    | [x],[] -> x
    | _      -> failwith "formule incorrecte"
  ;;
```

Remarques : on a supposé que toutes les priorités des opérateurs binaires sont positives, donc l'instruction `réduit(-1)` effectue toutes les opérations unaires et binaires en instance jusqu'à rencontrer une parenthèse ouvrante dans la pile des opérateurs. De même, l'introduction d'une parenthèse ouvrante avant le parcours de f et d'une parenthèse fermante après ce parcours permet d'effectuer simplement les dernières opérations en instance.

Exercice 4-5
Il suffit de remplacer dans l'exercice **4-4** la pile de valeurs `pv` par une pile de lexèmes et d'y empiler les opérateurs retirés de `po` dans les fonctions `réduit` et `pushv` au lieu de calculer les résultats de ces opérations. Lorsque f a été entièrement parcourue, `pv` contient l'image miroir de la liste f' désirée.

Exercice 4-6
L'exercice **4-5** fournit un algorithme de construction de f', ce qui prouve son existence. On en démontre l'unicité par récurrence sur la longueur de f.

– Une formule infixe de longueur 1 bien formée est réduite à une variable, et la seule formule postfixe équivalente est elle aussi réduite à cette variable.

– Considérons une formule infixe bien formée f de longueur $n > 1$ comportant donc au moins un opérateur. Les règles de priorité déterminent de manière non ambiguë le dernier opérateur, *op*, à évaluer : c'est un opérateur unaire si f est de la forme $f = op(qqch)$ et c'est l'opérateur binaire de niveau le plus externe, de priorité la plus basse et le plus à droite dans les autres cas. Cet opérateur doit donc être placé en dernière position de f' et le début de f' est constitué d'une ou de deux formules postfixes calculant le ou les opérandes de *op* dans le bon ordre. Ces formules sont les formules postfixes associées aux sous-formules de f définissant les opérandes de *op*, donc elles sont uniques par hypothèse de récurrence.

Exercice 5-1
On a $\overline{p} = p \texttt{ nand } p$; $p \texttt{ et } q = \overline{p \texttt{ nand } q}$; $p \texttt{ ou } q = \overline{p} \texttt{ nand } \overline{q}$. Donc le connecteur `nand` permet d'exprimer les trois connecteurs `et`, `ou`, `non` et donc toutes les formules booléennes.

La négation ne peut être exprimée uniquement à l'aide de `et` et de `ou` car toute formule non constante constituée de ces connecteurs vaut `faux` lorsque toutes les variables valent `faux`.

Exercice 5-2
1. `peutetre`

2.
```
type normand = Vrai | Faux | Peutetre;;

let non = fun
| Vrai     -> Faux
| Faux     -> Vrai
| Peutetre -> Peutetre
;;

let et = fun
| Vrai Vrai -> Vrai
| Faux _    -> Faux
| _    Faux -> Faux
| _    _    -> Peutetre
;;
```

```
let ou p q = non(et (non p) (non q));;
let oubien p q = et (ou p q) (non(et p q));;
```

3.

```
#(* vérifie que deux fonctions normandes de trois *)
(* variables sont égales                           *)
let est_égal f g =
  let val = [|Vrai; Faux; Peutetre|] in
  let res = ref(true)                in
  for i = 0 to 2 do for j = 0 to 2 do for k = 0 to 2 do
    res := !res & (  f val.(i) val.(j) val.(k)
                   = g val.(i) val.(j) val.(k) )
  done done done;
  !res
;;
est_égal :
 (normand -> normand -> normand -> 'a) ->
 (normand -> normand -> normand -> 'a) -> bool = <fun>

#(* associativité *)
est_égal (fun p q r -> et (et p q) r)
         (fun p q r -> et p (et q r));;
- : bool = true
#est_égal (fun p q r -> ou (ou p q) r)
          (fun p q r -> ou p (ou q r));;
- : bool = true

#(* distributivité *)
est_égal (fun p q r -> et (ou p q) r)
         (fun p q r -> ou (et p r) (et q r));;
- : bool = true

(* etc *)
```

4. Non : on montre par récurrence sur la longueur d'une expression `et-ou-non` que si la fonction f associée est non constante alors la valeur de f est `peutetre` lorsque toutes les variables valent `peutetre`.

Exercice 5-3

1. f vaut `faux` si et seulement si parmi $p_1, \dots, p_n$ il y a un nombre pair de variables valant `vrai`. Soit g un facteur de f : g spécifie un cas où f vaut `faux`, donc spécifie les valeurs de *toutes* les variables $p_1, \dots, p_n$. Le nombre minimal de facteurs de f est donc le nombre de résultats `faux` dans la table de vérité de f, soit 2^{n-1}.

2. Non : partant de la forme normale conjonctive d'une fonction booléenne f, on peut toujours regrouper deux facteurs qui ne diffèrent que par la variable p_n. En effet, si u est une proposition quelconque, alors on a :

$$(u + p_n)u \equiv (u + p_n)(u + \overline{p_n}) \equiv u(u + \overline{p_n}) \equiv u.$$

Exercice 5-4
On a : $\overline{p} \equiv p \oplus$ `vrai`, $p + q \equiv p \oplus q \oplus pq$ et $(p \oplus q)(r \oplus s) \equiv pr \oplus ps \oplus qr \oplus qs$. Ces trois règles permettent de mettre récursivement une formule exprimée à l'aide de `et`, `ou` et `non` sous la forme d'une somme exclusive des constantes `vrai`, `faux` et de produit de variables.

On peut aussi considérer qu'une fonction booléenne est une application de $(\mathbf{Z}/2\mathbf{Z})^n$ dans $\mathbf{Z}/2\mathbf{Z}$, et qu'une telle application est polynomiale car $\mathbf{Z}/2\mathbf{Z}$ est un corps fini pour les lois $\oplus$ et $\times$. La transformation d'une formule logique en somme exclusive correspond donc à une écriture polynomiale de la fonction associée.

On démontre l'unicité à ordre près d'une telle écriture, après suppression des termes nuls ou répétés, par récurrence sur n. Pour $n = 1$ il suffit de constater que les quatre expressions possibles, `vrai`, `faux`, p et `vrai` $\oplus\, p$ définissent des fonctions booléennes de p distinctes. Supposons l'unicité établie pour toute formule booléenne à $n - 1$ variables et considérons une formule booléenne f à n variables $p_1, \dots, p_n$, ayant deux décompositions en sommes exclusives simplifiées. En regroupant dans chaque décomposition les termes ayant p_n en facteur, on a :

$$f \equiv a \oplus bp_n \equiv c \oplus dp_n$$

où a, b, c, d sont des sommes exclusives simplifiées en les variables $p_1, \dots, p_{n-1}$. En prenant $p_n = 0$ on obtient $a \equiv c$ donc $a = c$ par hypothèse de récurrence, et en prenant $p_n = 1$ on obtient $a + b \equiv c + d$ avec $a = c$ d'où $b \equiv d$ et donc $b = d$ par hypothèse de récurrence encore ($=$ désigne ici l'égalité à ordre près). ■

Puisqu'il y a unicité on peut tester si f représente une tautologie en calculant sa forme somme exclusive simplifiée : ou bien elle se réduit à `vrai` et f représente bien une tautologie, ou bien elle se réduit à `faux` et f n'est jamais satisfiable, ou bien elle contient d'autres termes et sa valeur n'est pas constante. Comme pour la méthode « forme conjonctive », cette méthode a un coût mémoire (et donc aussi temps) exponentiel dans le pire des cas. Par exemple la forme somme exclusive réduite de :

$$(p_1 \oplus p_2)(p_1 \oplus p_3)\dots(p_1 \oplus p_n)$$

est :

$$\Big(\bigoplus p_1 p_2^{\alpha_2} \dots p_n^{\alpha_n}\Big) \oplus p_2 \dots p_n$$

où $(\alpha_2, \dots, \alpha_n)$ décrit $\{0,1\}^{n-1}$, donc cette forme comporte $2^{n-1} + 1$ termes.

Exercice 5-5
La seule difficulté est de réaliser la réunion sans répétition des listes de variables de deux sous-formules. Comme les chaînes de caractères peuvent être comparées par l'ordre lexicographique (ordre total qui est celui que fournit `<=` en CAML), on peut utiliser la méthode de fusion sans répétition sur les listes de variables triées selon cet ordre.

```
let rec liste_variables f = match f with
| Const(_)   -> []
| Var(p)     -> [p]
```

```
| Mono(_,g)  -> liste_variables g
| Bin(_,g,h) -> union (liste_variables g) (liste_variables h)
                where rec union l l' = match (l,l') with
                | [],_ -> l'
                | _,[] -> l
                | p::suite, p'::suite' ->
                    if      p < p' then p  :: (union suite l')
                    else if p > p' then p' :: (union l suite)
                    else                p  :: (union suite suite')
;;
```

Exercice 5-6

```
let rec simplifie f = match f with
| (Const(_) | Var(_)) -> f

| Mono(op,g) -> begin
    match simplifie g with
    | Const(x) -> Const(op x)
    | g'       -> Mono(op,g')
    end

| Bin(op,g,h) -> begin
    match (simplifie g, simplifie h) with
    | Const(x),Const(y) -> Const(op x y)
    | Const(x),h'       -> simplifie_à_gauche op x h'
    | g',Const(x)       -> simplifie_à_droite op g' x
    | g',h'             -> Bin(op,g',h')
    end

where simplifie_à_gauche op x f =
    match (op x true),(op x false) with
    | true,true   -> Const(true)
    | true,false  -> f
    | false,true  -> Mono(non,f)
    | false,false -> Const(false)

and simplifie_à_droite op f x =
    match (op true x),(op false x) with
    | true,true   -> Const(true)
    | true,false  -> f
    | false,true  -> Mono(non,f)
    | false,false -> Const(false)
;;
```

Les fonctions auxiliaires `simplifie_à_gauche` et `simplifie_à_droite` testent si les expressions `op x f` ou `op f x` dans lesquelles f est inconnue dépendent effectivement de `f`, et si oui de quelle manière. Ces fonctions sont très générales puisqu'il n'est fait aucune hypothèse sur les connecteurs utilisés, mais en contrepartie on doit « redécouvrir » à chaque fois les règles de simplification. Si le temps de calcul d'un connecteur est constant, ce qui est probable, la complexité asymptotique de cette méthode de simplification est $O(N)$ où N est la longueur de la formule à simplifier. Un codage « en dur » des règles de simplification n'apporterait pas d'amélioration significative. Remarquons que les doubles négations ne sont pas simplifiées, et que les fonctions `simplifie_à_droite` et `simplifie_à_gauche` peuvent créer des doubles négations supplémentaires. On pourrait corriger ce défaut par la même méthode à l'aide d'une fonction de simplification d'un opérateur unaire.

Exercice 5-7

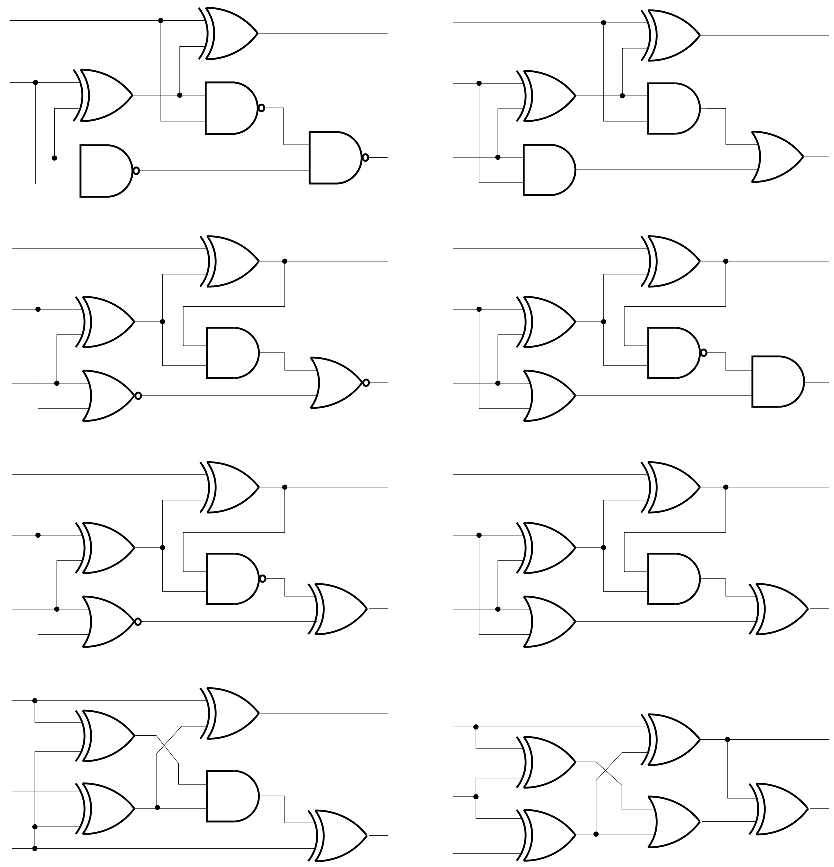
On vérifie aisément que le circuit proposé convient. Je n'ai pas de meilleure réponse à la deuxième question que de passer en revue tous les circuits à quatre portes ou moins et de constater qu'aucun ne convient. Voici un programme de recherche listant tous les circuits de taille minimale à n portes ou moins générant les fonctions S_0 et S_1 :

```
(* tables de vérité *)
type table == bool list;;

(* un circuit est une liste de tables de vérité *)
(* classées par ordre lexicographique          *)

(* ajoute une table à un circuit *)
let rec ajoute t circ = match circ with
| []       -> [t]
| a::suite -> if t = a then circ
         else if t < a then t::circ
         else a :: ajoute t suite
;;

(* variables et fonctions à obtenir *)
let e0 = [true;  true;  true;  true;  false; false; false; false]
and e1 = [true;  true;  false; false; true;  true;  false; false]
and e2 = [true;  false; true;  false; true;  false; true;  false]
and s0 = [true;  false; false; true;  false; true;  true;  false]
and s1 = [true;  true;  true;  false; true;  false; false; false]
;;

(* fonctions de transfert *)
let non   a   = map (fun x -> not x) a;;
let et    a b = map2 (prefix &)  a b;;
let ou    a b = map2 (prefix or) a b;;
let ouex  a b = map2 (prefix <>) a b;;
let nonet a b = non(et a b);;
let nonou a b = non(ou a b);;
let binops = [et; ou; ouex; nonet; nonou];;
```

```
(* liste des paires de points d'un circuit *)
let rec paires = function
| []       -> []
| a::suite -> (map (fun b -> (a,b)) suite) @ paires(suite)
;;

(*
 * ensembles de circuits
 * déjavus   = circuits de taille < n examinés
 * trouvés   = circuits contenant s0 et s1
 * possibles = circuits de taille < n-1 ou de taille
 *             n-1 contenant s0 ou s1
*)
let déjavus   = ref(set__empty eq__compare);;
let trouvés   = ref(set__empty eq__compare);;
let possibles = ref(set__empty eq__compare);;

(* examine un circuit *)
let nouveau n circ =
  if not(set__mem circ !déjavus) then begin
    let l = list_length(circ)
    and a0 = mem s0 circ
    and a1 = mem s1 circ in
    if l < n   then déjavus := set__add circ !déjavus;
    if a0 & a1 then trouvés := set__add circ !trouvés;
    if (l < n-1) or ((l = n-1) & (a0 or a1))
               then possibles := set__add circ !possibles
  end
;;

(* engendre tous les circuits déduits de *)
(* "circ" par adjonction d'une porte     *)
let successeurs n circ =

  (* ajoute un inverseur ou une porte binaire *)
  let job1 a       = nouveau n (ajoute (non a) circ)
  and job2 (a,b) f = nouveau n (ajoute (f a b) circ) in
  do_list job1 circ;
  do_list (fun p -> do_list (job2 p) binops) (paires circ)
;;

(* engendre récursivement tous les circuits  *)
(* par nombre croissant de portes jusqu'à la *)
(* la taille n et retourne tous les circuits *)
(* de taille minimale contenant s0 et s1     *)
let cherche(n) =
  déjavus   := set__empty eq__compare;
  trouvés   := set__empty eq__compare;
```

```
    possibles := set__empty eq__compare;

    déjavus   := set__add [e2;e1;e0] !déjavus;
    possibles := set__add [e2;e1;e0] !possibles;

    while   set__is_empty !trouvés
      & not(set__is_empty !possibles) do
        let p = !possibles in
        possibles := set__empty eq__compare;
        set__iter (successeurs n) p
    done;
    set__elements !trouvés
  ;;
```

`cherche(7)` recherche tous les circuits à 4 portes ou moins contenant S_0 et S_1 et n'en trouve aucun. `cherche(8)` trouve 24 circuits à 5 portes convenant qui se réduisent après élimination des symétries aux 8 circuits suivants :



Exercice 5-8

Soit p la profondeur du circuit calculant S_n : par récurrence le nombre d'entrées de ce circuit est au plus égal à 2^p et toutes les entrées $A_0 \ldots A_{n-1}$, $B_0 \ldots B_{n-1}$ doivent être prises en compte car chacune de ces entrées est susceptible d'influer sur S_n. On a donc $2^p \geqslant 2n$ d'où le résultat.

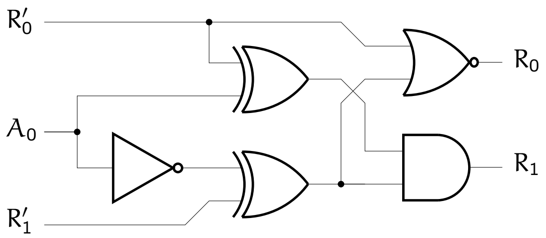
Exercice 5-9

Première méthode

Soit $N = \overline{A_{n-1} \ldots A_1 A_0}^2$ le nombre à diviser par 3 et $R = \overline{R_1 R_0}^2$ le reste à obtenir avec $0 \leqslant R \leqslant 2$. On procède par récurrence en supposant disposer du reste $R' = \overline{R'_1 R'_0}^2$ de la division par 3 de $N' = \overline{A_{n-1} \ldots A_1}^2$:

A_0	R'_0	R'_1	R_0	R_1
0	0	0	0	0
0	1	0	0	1
0	0	1	1	0
1	0	0	1	0
1	1	0	0	0
1	0	1	0	1

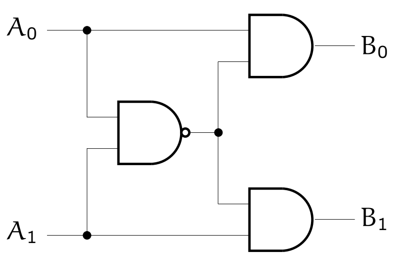
Il apparaît que $R_0 = 1$ si et seulement si $R'_0 = 0$, $R'_1 = \overline{A_0}$ et que $R_1 = 1$ si et seulement si $R'_0 = \overline{A_0}$, $R'_1 = A_0$, ce qui donne la cellule de division élémentaire :



Un diviseur n-bits s'obtient alors par mise en cascade de telles cellules, ce qui donne un circuit à $5n$ portes de profondeur $3n$.

Deuxième méthode

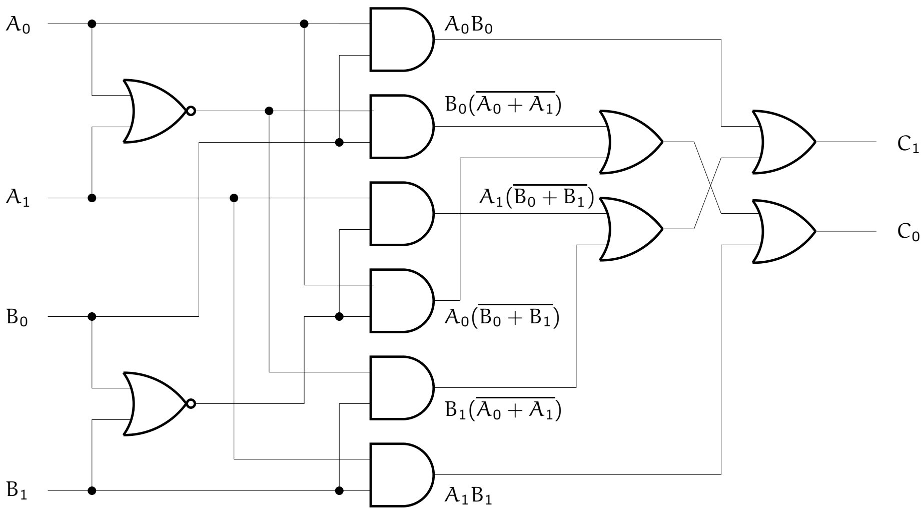
Comme $4 \equiv 1$ [3], on a $N \equiv \overline{A_1 A_0}^2 + \overline{A_3 A_2}^2 + \ldots + \overline{A_{2p-1} A_{2p-2}}^2$ [3] où $p = \lceil n/2 \rceil$ et $A_{2p-1} = 0$ si n est impair. Donc, pour obtenir le résidu de N modulo 3, il suffit de grouper les bits de N deux par deux, de calculer les résidus modulo 3 de chaque groupe, puis d'additionner modulo 3 les résidus obtenus. Le circuit logique ci-dessous calcule le résidu $\overline{B_1 B_0}^2$ de $\overline{A_1 A_0}^2$ modulo 3.



Considérons à présent le problème de l'addition modulo 3 : étant donnés deux nombres $A = \overline{A_1A_0}^2$ et $B = \overline{B_1B_0}^2$ compris entre 0 et 2, on veut obtenir $C = \overline{C_1C_0}^2$ lui aussi compris entre 0 et 2 tel que $A + B \equiv C\ [3]$. On construit la table de vérité de (C_0, C_1) :

A_0	A_1	B_0	B_1	C_0	C_1
0	0	0	0	0	0
0	0	1	0	1	0
0	0	0	1	0	1
1	0	0	0	1	0
1	0	1	0	0	1
1	0	0	1	0	0
0	1	0	0	0	1
0	1	1	0	0	0
0	1	0	1	1	0

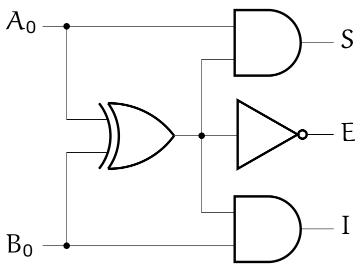
On en déduit : $C_0 = B_0$ si $A_0 = A_1 = 0$, $C_0 = \overline{B_0 + B_1}$ si $A_0 = 1$, et $C_0 = B_1$ si $A_1 = 1$, soit dans tous les cas : $C_0 = B_0(\overline{A_0 + A_1}) + A_0(\overline{B_0 + B_1}) + A_1B_1$. On obtient de même : $C_1 = B_1(\overline{A_0 + A_1}) + A_1(\overline{B_0 + B_1}) + A_0B_0$ et l'on peut construire un circuit « additionneur modulo 3 » implémentant ces formules :



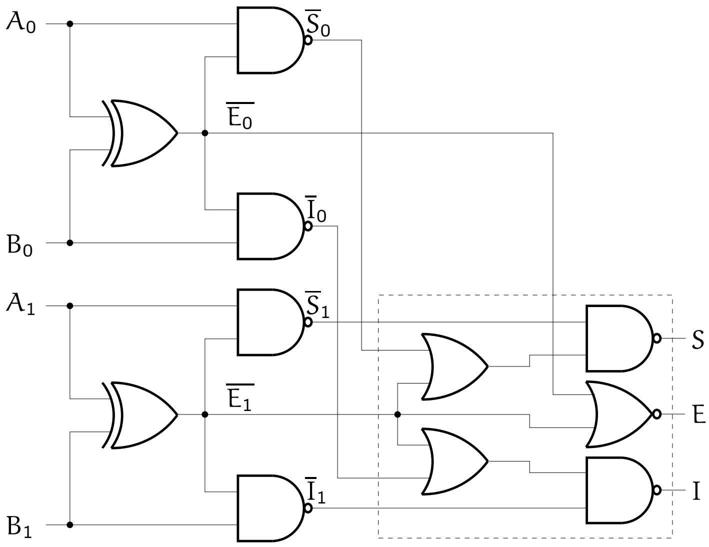
Enfin, on calcule la somme modulo 3 des p résidus en assemblant $p - 1$ additionneurs modulo 3 en arbre binaire comme le multiplieur parallèle décrit à la section **6-1**, figure 14. On obtient ainsi un circuit calculant le résidu modulo 3 d'un nombre de n bits ayant une profondeur $2 + 4\lceil \log_2 p \rceil = 4\lceil \log_2 n \rceil - 2$ et un nombre de portes élémentaires égal à $3\lfloor n/2 \rfloor + 12(p - 1) \leqslant \frac{15}{2}n$.

Exercice 5-10

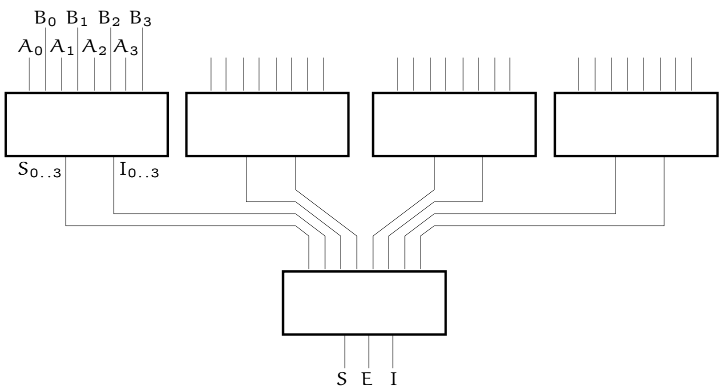
1.



2.



3. On compare les bits par groupes de n puis on compare avec un dernier comparateur n-bits les n sorties S, I obtenues :



4. C'est possible, mais il existe de meilleures solutions : la construction d'un comparateur 256-bits par association de comparateurs requiert 17 comparateurs 16-bits et a une profondeur double de celle de ces comparateurs, ce qui donne récursivement 255 comparateurs 2-bits, soit 2805 portes et une profondeur de 32.

Une autre solution est de fusionner les résultats de deux comparateurs 128-bits en utilisant le circuit additionnel de la question *2* à condition que ces comparateurs fournissent les sorties $\overline{S}, \overline{E}$ et $\overline{I}$. Le coût de cette association est de 5 portes et la profondeur est augmentée de 2. En réalisant les comparateurs 128-bits suivant le même principe, on peut obtenir un comparateur 256-bits avec 256 comparateurs 1-bit et 255 modules de fusion, soit au total 2043 portes et une profondeur de 18.

D'ailleurs il n'est pas nécessaire de calculer la sortie I pour les comparateurs intermédiaires, $\overline{S}$ et $\overline{E}$ contiennent toute l'information nécessaire pour réaliser les fusions. La sortie I finale peut être calculée par : $I = \overline{\overline{S} + \overline{E}}$ ce qui nécessite une seule porte et une profondeur de 1. Avec cette économie, on obtient un comparateur 256-bits constitué de 1278 portes ayant une profondeur de 19 ...

Exercice 5-10

La table de vérité du circuit demandé est la suivante (on n'a marqué que les 1 dans les colonnes a–g pour une meilleure lisibilité) :

n	D	C	B	A	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	
1	0	0	0	1		1	1				
2	0	0	1	0	1	1		1	1		1
3	0	0	1	1	1	1	1	1			1
4	0	1	0	0		1	1			1	1
5	0	1	0	1	1		1	1		1	1
6	0	1	1	0	1		1	1	1	1	1
7	0	1	1	1	1	1	1				
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1		1	1

En distinguant selon la valeur du couple (D, C), on obtient la table suivante :

D	C	a	b	c	d	e	f	g
0	0	$\overline{A}+B$	1	$A+\overline{B}$	$\overline{A}+B$	$\overline{A}$	$\overline{A+B}$	B
0	1	$A+B$	$\overline{A\oplus B}$	1	$A\oplus B$	$\overline{A}B$	$\overline{AB}$	$\overline{AB}$
1	0	1	1	1	1	$\overline{A}$	1	1

D'où les formules :

$$
\begin{aligned}
a &= (C \oplus \overline{A}) + B + D,\\
b &= \overline{C(A \oplus B)},\\
c &= A + \overline{B} + C + D,\\
d &= \overline{C}(\overline{A} + B) + C(A \oplus B) + D,\\
e &= \overline{A}(B + \overline{C}),\\
f &= \overline{C}\,\overline{A + B} + C\overline{AB} + D,\\
g &= \overline{C}B + C\overline{AB} + D.
\end{aligned}
$$

Exercice 6-1

1. On a $T(n) = nT(n-1) + an + b$ donc $\dfrac{T(n)}{n!} = \dfrac{T(n-1)}{(n-1)!} + \dfrac{a}{(n-1)!} + \dfrac{b}{n!}$. Comme la série $\sum_{k=0}^{\infty} 1/k!$ est convergente, on en déduit qu'il existe $\lambda > 0$ tel que $T(n) \sim \lambda n!$.

2. Il y a n mineurs d'ordre $n-1$ à calculer, C_n^2 mineurs d'ordre $n-2$ et de manière générale C_n^k mineurs d'ordre $n-k$. Le temps de calcul d'un mineur d'ordre k connaissant tous les mineurs d'ordre $k-1$ est $t_k = ak + b$ en supposant que les mineurs d'ordre $k-1$ sont mémorisés de telle sorte qu'ils puissent être retrouvés en temps constant. Donc le temps de calcul d'un déterminant $n \times n$ est :

$$T(n) = \sum_{k=0}^{n} C_n^k t_{n-k} = \sum_{k=0}^{n} C_n^k t_k = \sum_{k=0}^{n} (ak+b) C_n^k = n2^{n-1}a + 2^n b.$$

La complexité spatiale est proportionnelle à :

$$\max(C_n^k,\ 0 \leqslant k \leqslant n) = C_n^{\lfloor n/2 \rfloor} \sim \frac{2^{(n-1)/2}}{\sqrt{\pi n}}.$$

La méthode du pivot de GAUSS qui a une complexité $O(n^3)$ est nettement plus performante, mais elle produit des fractions et des résultats approchés, sauf si l'on effectue le calcul en gérant à part numérateur et dénominateur mais dans ce cas les coefficients manipulés sont de plus en plus gros et il faut effectuer les opérations en multiprécision, ce qui modifie la complexité.

Exercice 6-2

Si n est impair, on augmente les matrices d'une ligne et d'une colonne nulle. Le temps de calcul vérifie donc l'équation : $T(n) = 7T(\lceil n/2 \rceil) + O(n^2)$ où $O(n^2)$ représente le temps des additions/soustractions et des recopies de coefficients. On en déduit : $T(n) = \Theta(n^{\log_2 7})$.

Exercice 6-3

Si on se limite à des chaînes de longueur exactement n, en considérant qu'il y a 256 caractères possibles, cela fait 256^n chaînes différentes supposées équiprobables. La probabilité de trouver deux caractères différents à une position donnée dans chaque chaîne est $p = 255/256$ donc le rang moyen d'apparition de la première différence est inférieur ou égal à $256/255 \approx 1.004$ (et supérieur ou égal à 1). Le nombre moyen de comparaisons de caractères effectuées est égal à ce rang moyen, donc est compris entre 1 et 1.004.

Si les longueurs ne sont pas fixées, soit p_k la probabilité que la première chaîne soit de longueur k. On peut ajouter un marqueur de fin à chaque chaîne (pris en dehors des 256 caractères usuels) et comparer les $k+1$ premiers caractères des deux chaînes. Le nombre moyen de caractères comparés (pour une première chaîne de longueur k) est :

$$\begin{aligned} c_k &= \frac{255}{256}\Big(\sum_{m=1}^{k} \frac{m}{256^{m-1}}\Big) + \frac{k+1}{256^k} = \frac{255}{256}\Big(\sum_{m=1}^{k} \frac{m}{256^{m-1}} + \sum_{m=k+1}^{\infty} \frac{k+1}{256^{m-1}}\Big) \\ &\leqslant \frac{255}{256}\Big(\sum_{m=1}^{\infty} \frac{m}{256^{m-1}}\Big) = \frac{256}{255} \end{aligned}$$

et le nombre moyen de caractères comparés pour une première chaîne de longueur quelconque est :

$$N = \frac{p_0c_0 + p_1c_1 + \ldots + p_nc_n}{p_0 + p_1 + \ldots + p_n} \leqslant \frac{256}{255}.$$

Exercice 6-4

1. Le calcul naïf de la suite de FIBONACCI ($u_n = u_{n-1} + u_{n-2}$) donne

$$T(n) = T(n-1) + T(n-2) + K.$$

2. D'après la relation de récurrence la fonction T est croissante, donc on peut écrire : $T(n) \geqslant 2T(n-2) + f(n)$ et par récurrence, $T(n) \geqslant 2^{n/2}\lambda$. Le temps d'exécution est au moins exponentiel.

On peut obtenir une estimation plus précise en considérant les racines de l'équation caractéristique $x^2 = x + 1$ associée à l'équation de récurrence $u_n = u_{n-1} + u_{n-2}$, c'est-à-dire :

$$\phi = \frac{1+\sqrt{5}}{2} \quad \text{et} \quad \overline{\phi} = \frac{1-\sqrt{5}}{2} = -\frac{1}{\phi} = 1 - \phi.$$

On a $T(n) = (\phi + \overline{\phi})T(n-1) - \phi\overline{\phi}T(n-2) + f(n)$, donc en posant :

$$u(n) = T(n) - \overline{\phi}T(n-1) \quad \text{et} \quad v(n) = T(n) - \phi T(n-1)$$

on obtient les deux relations :

$$u(n) = \phi u(n-1) + f(n), \qquad v(n) = \overline{\phi}v(n-1) + f(n).$$

Ceci permet de calculer exactement $u(n)$, $v(n)$ puis $T(n) = \dfrac{\phi u(n) - \overline{\phi}v(n)}{\phi - \overline{\phi}}$.

On obtient :

$$T(n) = \frac{\phi^n u(1) - \overline{\phi}^n v(1)}{\phi - \overline{\phi}} + \sum_{k=2}^{n} f(k)\frac{\phi^{n-k} - \overline{\phi}^{n-k}}{\phi - \overline{\phi}}.$$

Donc si f est à croissance polynomiale alors $T(n) \sim \lambda\phi^n$ pour un certain $\lambda > 0$.

Exercice 6-5

$T(2^p) = 2T(2^{p-1}) + p2^p$ donc $T(2^p) = 2^p\left(T(1) + \sum\limits_{k=1}^{p} k\right) \sim p^2 2^{p-1}$.

Par monotonie, il vient : $T(n) = \Theta(n(\ln n)^2)$.

Exercice 6-6
On pose $n_p = 2^p - 1$ de sorte que $\lfloor (n_p - 1)/2 \rfloor = 2^{p-1} - 1 = n_{p-1}$. On peut donc étudier la relation partielle :

$$T(n_p) = 2T(n_{p-1}) + 2^p - 1,$$

qui a pour solution :

$$T(n_p) = 2^p \left(T(n_0) + \sum_{k=0}^{p} \frac{2^k - 1}{2^k} \right) \sim p2^p.$$

On en déduit, par monotonie : $T(n) = \Theta(n \ln(n))$.

Exercice 6-7
Soient $T_{rec}(n)$, $M_{rec}(n)$, $T_{iter}(n)$, $M_{iter}(n)$, les complexités temporelles et spatiales de `u_rec` et `u_iter`. On a immédiatement $T_{iter}(n) = O(n^2)$ et $M_{iter}(n) = O(n)$. Par ailleurs il existe des constantes a et b telles que :

$$\begin{aligned} T_{rec}(n) &= a + bn + \sum_{k=1}^{n} T_{rec}(n-k) = a + bn + \sum_{k=0}^{n-1} T_{rec}(k) \\ &= 2T_{rec}(n-1) + b \end{aligned}$$

donc $T_{rec}(n) \sim \lambda 2^n$ pour un certain $\lambda > 0$.

En supposant une récupération systématique de la mémoire utilisée par les variables locales dès qu'une fonction est terminée, comme `u_rec` n'utilise qu'un nombre fixe de mémoires locales (`n`, `s`, `k` et l'adresse de retour), le nombre de positions mémoire nécessaires au calcul de `u_rec` est proportionnel au nombre maximal d'appels imbriqués, soit $M_{rec}(n) = O(n)$.

Exercice 6-8
On a de manière immédiate $T_{iter}(n) = O(n)$ et $M_{iter}(n) = O(n)$. Par ailleurs T_{rec} vérifie la relation :

$$T_{rec}(n) = a + T_{rec}(\lfloor n/2 \rfloor) + T_{rec}(\lfloor n/3 \rfloor).$$

Soit $\alpha > 0$. En posant $U(n) = \dfrac{T_{rec}(n) + a}{n^\alpha}$ on obtient :

$$U(n) \leqslant \frac{1}{2^\alpha} U(\lfloor n/2 \rfloor) + \frac{1}{3^\alpha} U(\lfloor n/3 \rfloor)$$

donc si l'on a $1/2^\alpha + 1/3^\alpha \leqslant 1$ alors la fonction U est bornée et l'on en déduit que $T_{rec}(n) = O(n^\alpha)$. Comme l'équation $1/2^\alpha + 1/3^\alpha = 1$ admet une unique racine $\alpha_0 \approx 0.787$, on obtient finalement $T_{rec}(n) = O(n^{\alpha_0})$. Enfin $M_{rec}(n)$ est proportionnel au nombre maximal d'appels imbriqués, soit $M_{rec}(n) = O(\ln n)$.

La méthode récursive naïve s'avère donc plus efficace que la méthode itérative avec stockage des résultats intermédiaires ! Dans le cas particulier de cette équation de récurrence, on peut obtenir un algorithme plus rapide que `u_rec` en remarquant que le calcul de u_n ne nécessite que la connaissance des termes précédents de la forme $u_{\lfloor n/2^a 3^b \rfloor}$ et donc qu'il suffit de construire une matrice indexée par a et b, de taille $\lceil \log_2 n \rceil \times \lceil \log_3 n \rceil$, pour éviter le recalcul de termes. On obtient alors un algorithme calculant u_n en temps $O(\ln^2 n)$ et en mémoire $O(\ln^2 n)$.

Exercice 7-1

Le nombre total de nœuds est $n = n_0 + n_1 + \ldots + n_p$ et le nombre de fils est $f = 0 \times n_0 + 1 \times n_1 + \ldots + p \times n_p$. Donc si a n'est pas vide, on a $n = f + 1$ soit :

$$n_0 = 1 + n_2 + \ldots + (p-1)n_p.$$

Exercice 7-2

S'il y a n nœuds alors il y a $n-1$ fils, donc chaque nœud a en moyenne $(n-1)/n$ fils. Le nombre moyen de fils droit dépend de l'arbre considéré mais, si l'on fait la moyenne sur tous les arbres binaires de taille n, on obtient par symétrie $(n-1)/2n$ fils droit en moyenne. Le nombre moyen de frères (moyenne sur tous les nœuds et sur tous les arbres binaires de taille n) semble plus compliqué.

Exercice 7-3

Ces ordres ne diffèrent que par le moment où l'on traite un nœud intérieur, ils coïncident donc sur les feuilles.

Exercice 7-4

La condition donnée est nécessaire par définition des ordres préfixe et postfixe. Pour la réciproque, considérons deux nœuds x et y tels que x précède y en ordre préfixe et succède à y en ordre postfixe, et soit z le premier ascendant commun à x et y en remontant vers la racine : si z n'est égal ni à x ni à y alors x et y appartiennent à deux branches distinctes issues de z et leurs ordres préfixe et postfixe relatifs coïncident, c'est absurde. On obtient aussi une contradiction si $z = y$ et donc nécessairement $z = x$.

Exercice 7-5

```
(* transforme une forêt générale en arbre binaire *)
let rec bin_of_g_foret f = match f with
| [] -> B_vide
| G_noeud(x,fils)::suite ->
    B_noeud(x, bin_of_g_foret fils, bin_of_g_foret suite)
;;

(* transforme un arbre général en arbre binaire *)
let bin_of_g(a) = bin_of_g_foret([a]);;

(* transforme un arbre binaire en forêt générale *)
let rec g_foret_of_bin a = match a with
| B_vide -> []
| B_noeud(x,g,d) ->
    G_noeud(x, g_foret_of_bin g) :: g_foret_of_bin d
;;
```

Exercice 7-6

Les ordres préfixe sur a et a' coïncident ; l'ordre postfixe sur a coïncide avec l'ordre infixe sur a' ; l'ordre postfixe sur a' n'a pas d'interprétation simple pour a.

Exercice 7-7

successeur préfixe

```
let rec succ_pref(i,n) =
  if 2*i <= n then 2*i                          (* fils gauche *)
  else if (i mod 2 = 0) & (i+1 <= n) then i+1  (* frère droit *)
  else if i > 1 then succ_pref(i/2, 2*(i/2)-1)
            (* successeur du père après suppression des fils *)
  else failwith "plus de successeur"
;;
```

La correction de la fonction `succ_pref` se démontre par récurrence sur n. Notons $f_n(i) = \mathtt{succ\_pref}(i,n)$: f_1 est manifestement correcte. Si f_n est correcte (c'est-à-dire si $f_n(i)$ est correct pour $1 \leqslant i \leqslant n$), soit p le prédécesseur de $n+1$ dans la numérotation préfixe : il faut montrer que $f_{n+1}(i) = f_n(i)$ pour $1 \leqslant i \leqslant n$ et $i \neq p$, que $f_{n+1}(p) = n+1$ et que $f_{n+1}(n+1) = f_n(p)$.

- Si $n+1$ est pair, alors $p = (n+1)/2$, donc pour $i \neq p$ et $i \leqslant n$ les résultats des trois tests sont inchangés si l'on remplace n par $n+1$, d'où $f_{n+1}(i) = f_n(i)$. On a aussi clairement $f_{n+1}(p) = n+1$ et $f_{n+1}(n+1) = f_{2p-1}(p) = f_n(p)$.

- Si $n+1$ est impair alors $p = n$, donc pour $i < n$ les résultats des trois tests sont là aussi inchangés et l'on a $f_{n+1}(i) = f_n(i)$. Enfin, $f_{n+1}(p) = n+1$ et $f_{n+1}(n+1) = f_{n-1}(n/2) = f_n(n)$. ■

Le calcul de `succ_pref` peut s'effectuer en mémoire constante puisque c'est une fonction récursive terminale. Par contre le temps d'exécution de `succ_pref` n'est pas constant et le temps maximal de calcul de $\mathtt{succ\_pref}(i,n)$ est clairement $\Theta(\ln n)$. Cependant on constate que, lors du parcours en ordre préfixe du vecteur, un entier i est « appelé » par `succ_pref` (c'est-à-dire que l'on demande un calcul $\mathtt{succ\_pref}(i,p)$ pour un certain entier p) au plus deux fois : une fois pour obtenir le successeur normal de i, et une deuxième fois pour obtenir le successeur « anormal » $\mathtt{succ\_pref}(i, 2i-1)$ lorsque l'on a fini de visiter la descendance de i s'il en a une. Ceci découle directement de la correction de `succ_pref`. Donc le temps cumulé de calcul de tous les successeurs lors de ce parcours est $O(n)$.

successeur postfixe

```
(* cherche le noeud le plus à gauche descendant de i *)
let rec descend(i,n) = if 2*i <= n then descend(2*i,n) else i;;

let succ_post(i,n) =
  if i = 1 then failwith "plus de successeur"
  else if (i mod 2 = 1) or (i = n) then i/2       (* père *)
  else descend(i+1,n)  (* descendant gauche du frère droit *)
;;
```

Les corrections de descend et succ_post sont immédiates. Comme la fonction descend est récursive terminale, descend et succ_post peuvent s'exécuter en mémoire constante.

successeur infixe

```
let rec succ_inf(i,n) =
  if 2*i+1 <= n then descend(2*i+1,n)  (* desc. gauche du fils droit *)
  else if i mod 2 = 0 then i/2         (* père *)
  else if i > 1 then succ_inf(i/2,i-1)
                 (* successeur du père après suppression de ce fils *)
  else failwith "plus de successeur"
;;
```

On établit la correction de succ_inf par récurrence sur n de la même manière que celle de succ_pref. Cette fonction a elle aussi une complexité spatiale constante puisque récursive terminale.

Exercice 7-8
x précède y dans l'ordre préfixe (infixe, postfixe) inverse si et seulement si x succède à y dans l'ordre postfixe (infixe, préfixe) direct.

Exercice 7-9
parcours effectue un parcours en profondeur d'abord en ordre préfixe d'une forêt. courspar effectue un parcours en largeur d'abord en respectant l'ordre naturel des branches. Soient n le nombre de nœuds et ℓ la largeur de la forêt : si le temps de traitement d'un nœud est constant alors la complexité temporelle de parcours est $\Theta(n)$ et celle de de courspar est $O(n + n\ell)$ (et $\Theta(n^2)$ pour un arbre binaire complet).

Exercice 7-10

```
let rec liste_préfixe(a) = match a with
| B_vide         -> []
| B_noeud(e,g,d) -> e :: ((liste_préfixe g) @ (liste_préfixe d))
;;
```

Complexité : soit $n_g(i)$ le nombre de descendants gauche d'un nœud i. Le temps de calcul de

```
e :: ((liste_préfixe g) @ (liste_préfixe d))
```

non compris le temps passé dans les appels récursifs est de la forme $\alpha n_g(i) + \beta$ donc le temps total de calcul pour un arbre a de taille n est :

$$T(a) = \alpha \sum_{i \in a} n_g(i) + \beta n \leqslant \alpha n h + \beta n$$

où h la hauteur de l'arbre (en effet, dans $\sum n_g(i)$ un nœud donné est compté autant de fois qu'il a d'ascendants à sa droite, donc au plus h fois). La complexité en nh est effectivement atteinte pour un arbre en forme de peigne gauche (chaque branche droite contient un seul nœud). Une meilleure solution consiste à parcourir l'arbre en ordre postfixe inverse (cf. exercice **7-8**) en accumulant les nœuds rencontrés, ce qui permet de ne faire que des insertions en tête de liste et donne une complexité linéaire en n :

```
let rec liste_préfixe(a,l) = match a with
| B_vide -> l
| B_noeud(e,g,d) -> let l'  = liste_préfixe(d,l)  in
                    let l'' = liste_préfixe(g,l') in
                    e :: l''
;;
```

Exercice 7-11
Voir le fichier `arbres.ml` disponible sur le serveur de l'INRIA :

```
http://pauillac.inria.fr/~quercia/automatx.tgz
```

Exercice 7-12

1.
```
let rec est_sous_arbre(a,b) = match (a,b) with
| (B_vide,B_vide) -> true
| B_noeud(ea,ga,da), B_noeud(eb,gb,db) ->
     (a = b) or est_sous_arbre(a,gb) or est_sous_arbre(a,db)
| _ -> false
;;
```

L'égalité entre arbres est prédéfinie en CAML (comme l'égalité entre deux structures quelconques de même type non fonctionnel). Si ce n'était pas le cas, on pourrait la définir par :

```
let rec égal(a,b) = match (a,b) with
| (B_vide,B_vide) -> true
| (B_noeud(ea,ga,da),B_noeud(eb,gb,db)) ->
     (ea = eb) & égal(ga,gb) & égal(da,db)
| _ -> false
;;
```

2.
```
let rec est_inclus(a,b) = match (a,b) with
| (B_vide,_) -> true
| (_,B_vide) -> false
| (B_noeud(ea,ga,da),B_noeud(eb,gb,db)) ->
      ((ea = eb) & coïncide(ga,gb) & coïncide(da,db))
   or est_inclus(a,gb) or est_inclus(a,db)

and coïncide(a,b) = match (a,b) with
| (B_vide,_) -> true
| (_,B_vide) -> false
| (B_noeud(ea,ga,da),B_noeud(eb,gb,db)) ->
      (ea = eb) & coïncide(ga,gb) & coïncide(da,db)
;;
```

Exercice 7-13

On ne peut pas répondre précisément à la question car les données fournies (années de naissance et de décès des individus) ne permettent pas de savoir dans quel ordre ont eu lieu les naissances et les décès dans une même année, ni à quelles dates la royauté était effectivement en vigueur (Louis XVII, et les descendants de Charles X n'ont pas régné car le régime politique en vigueur alors était le régime républicain). Le programme ci-dessous fait abstraction des changements de régime et suppose que dans une même année toutes les naissances ont lieu avant tous les décès et que, en cas de décès multiples, les individus meurent en respectant l'ordre dynastique. On parcourt l'arbre en ordre préfixe et on imprime tous les noms d'individus encore en vie après le décès du dernier roi rencontré jusqu'alors.

```
type individu == string*int*int;; (* nom, naissance, décès *)

(* imprime les noms des rois à partir de la date d *)
(* retourne la date de décès du dernier roi        *)
let rec imprime(d, (G_noeud((x,_,b), fils))) =
   if b >= d then print_endline(x);
   let f = ref(fils) and d' = ref(max b d) in
   while !f <> [] do
     d' := imprime(!d', hd(!f));
     f := tl(!f)
   done;
   !d'
;;
```

Remarquer que l'arbre donné est incohérent : le cousin de Louis XV est censé être né deux ans après le décès de son père, cette incohérence figure dans le document cité en référence.

Exercice 7-14

1. On constitue l'expression parenthésée par parcours préfixe inverse comme à l'exercice **7-10** pour garantir une complexité linéaire (rappel : `rev` calcule l'image d'une liste de longueur n en temps $\Theta(n)$).

```
let rec exp_of_arbre a accu = match a with
| G_noeud(e,f) ->  Pargauche :: Etiquette(e)
                :: exp_of_forêt (rev f) (Pardroite :: accu)

and exp_of_forêt f accu = match f with
| []       -> accu
| n::suite -> exp_of_forêt suite (exp_of_arbre n accu)
;;
```

2.
```
let rec arbre_of_expr liste = match liste with
| Pargauche :: Etiquette(e) :: suite ->
   let (f,reste) = forêt_of_expr(suite) in (G_noeud(e,f), reste)
| _ -> failwith "expression mal formée"
```

```
and forêt_of_expr liste = match liste with
| Pardroite :: suite -> ([],suite)
| _ -> let (a,suite) = arbre_of_expr(liste) in
       let (f,reste) = forêt_of_expr(suite) in
       (a::f, reste)
;;
```

`arbre_of_expr(liste)` vérifie que le début de `liste` est l'expression d'un arbre et renvoie cet arbre et le reste de la liste. De même, `forêt_of_expr(liste)` extrait la forêt représentée par le début de `liste` et terminée par une parenthèse fermante, et renvoie cette forêt et le reste de la liste sans la parenthèse fermante. La liste complète représente bien un arbre si et seulement si `arbre_of_expr(liste)` termine sans erreur et si le reste de la liste est vide.

Exercice 7-15

```
(* imprime une formule en plaçant des parenthèses si *)
(* nécessaire. "p" est la priorité de l'opérateur    *)
(* précédent ou suivant.                             *)
let rec imprime_avec_parenthèses p f = match f with
| Valeur(_)    -> imprime(f)
| Op1(_)       -> imprime(f)
| Op2(_,q,_,_) -> if q < p then begin
                    print_char `(`; imprime(f); print_char `)`
                  end
                  else imprime(f)

(* imprime une formule sans placer de parenthèses autour *)
and imprime(f) = match f with
| Valeur(v)      -> print_string(v)
| Op1(nom,g)     -> print_string(nom);
                    print_char `(`;
                    imprime(g);
                    print_char `)`
| Op2(nom,p,g,h) -> imprime_avec_parenthèses p g;
                    print_char ` `;
                    print_string(nom);
                    print_char ` `;
                    imprime_avec_parenthèses p h
;;
```

Exercice 7-16

Soit $h(n)$ la hauteur maximale d'un arbre α-équilibré de taille inférieure ou égale à n. On a la relation :

$$h(n) \leqslant 1 + h(\lfloor \alpha n \rfloor)$$

et donc par récurrence,

$$h(n) \leqslant k + h(\lfloor \alpha^k n \rfloor).$$

En prenant $k = \lceil -\ln(n)/\ln(\alpha) \rceil$ on obtient $h(n) = O(\ln n)$ et l'inégalité inverse est vraie pour tout arbre binaire de taille n, qu'il soit équilibré ou non.

Exercice 7-17
Pour $h \in \mathbf{N}$ soit $m(h)$ le plus petit nombre de nœuds d'un arbre de hauteur h équilibré à k près. On a pour $h \geqslant 2$:

$$m(h) = 1 + m(h-1) + \min(m(h-1), m(h-2), \dots, m(h-k-1)).$$

Par conséquent la fonction m est croissante sur $\mathbf{N}^*$ et $m(0) = 1$, $m(1) = 2$, donc elle est croissante sur $\mathbf{N}$. On en déduit :

$$m(h) = 1+m(h-1)+m(h-k-1) \iff 1+m(h) = (1+m(h-1))+(1+m(h-k-1)).$$

Soit α l'unique racine positive de l' équation $x^{k+1} = x^k + 1$ $(1 < \alpha < 2)$. Par récurrence on a $m(h) \geqslant c\alpha^h$ pour une certaine constante c donc :

$$h \leqslant \frac{\ln(m(h)) - \ln c}{\ln \alpha}.$$

Pour un arbre binaire équilibré à k près et à n nœuds on a alors :

$$h \leqslant \frac{\ln(n) - \ln c}{\ln \alpha}.$$

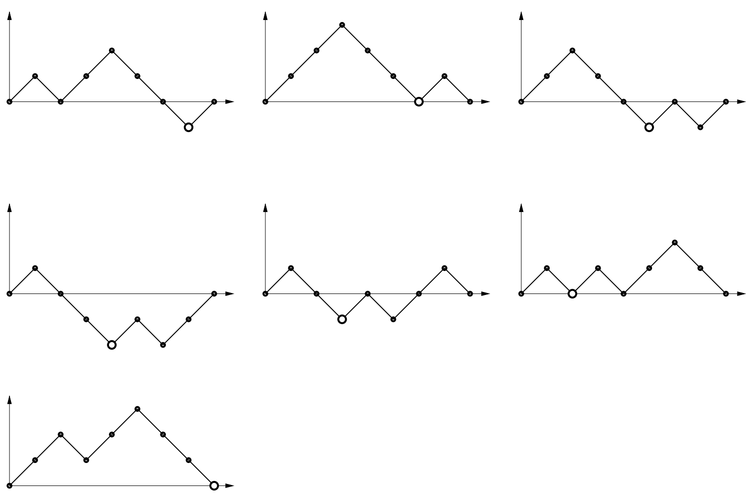
Exercice 7-18

1. La transformation « fils gauche - frère droit » met en correspondance les arbres généraux ordonnés à n nœuds et les arbres binaires à n nœuds dont la branche droite est vide. Donc $G_n = B_{n-1}$.

2. C'est immédiat par récurrence sur n. L'exercice **7-14** fournit par ailleurs un algorithme de calcul de l'arbre représenté par une liste admissible de parenthèses.

3. On note $f(i)$ la différence entre le nombre de parenthèses ouvrantes et le nombre de parenthèses fermantes dans $[u_0, \dots, u_{i-1}]$. La figure ci-dessous présente les graphes des fonctions f associées aux itérées par T de la suite

$$u = [\ (\ )\ (\ (\ )\ )\ )\ (\ ].$$



Soit i_m le premier entier $i \in [\![2, 2n]\!]$ où f atteint son minimum : T a pour effet de diminuer i_m d'une unité modulo $2n-1$ donc il existe un unique entier k tel que $T^k(u)$ corresponde à $i_m = 2n$, c'est-à-dire tel que $T^k(u)$ soit admissible.

4. Il y a C_{2n-1}^n suites u constituées de n parenthèses ouvrantes et n parenthèses fermantes la première étant ouvrante, et ces suites se répartissent en $C_{2n-1}^n/(2n-1)$ groupes de $2n-1$ suites déduites les unes des autres par l'action de T. Chaque groupe contient une et une seule suite admissible, donc le nombre de suites admissibles est :

$$\frac{C_{2n-1}^n}{2n-1} = \frac{1}{n} C_{2n-2}^{n-1}$$

et l'on a :
$$G_n = \frac{1}{n} C_{2n-2}^{n-1}, \qquad B_n = \frac{1}{n+1} C_{2n}^n.$$

Exercice 7-19

1. Soit h la hauteur de a et x un nœud de profondeur h. S'il existe un nœud y de profondeur $h' < h-1$ ayant une branche vide, alors on peut « déplacer » le nœud x pour le mettre à la place de cette branche vide. L'arbre a' obtenu est un arbre binaire à n nœuds tel que $L_e(a') = L_e(a) + h' - h + 1 < L_e(a)$ donc la longueur du chemin externe de a n'est pas minimale. Ceci prouve qu'une condition nécessaire pour que $L_e(a)$ soit minimale est que tous les nœuds de profondeur inférieure ou égale à $h-2$ ont deux fils, ce qui équivaut à dire qu'il y a 2^k nœuds de profondeur k pour tout $k \in [\![0, h-1]\!]$.

Réciproquement, supposons cette dernière condition remplie et notons p le nombre de nœuds de a de profondeur h. Alors $n = p + 2^h - 1$ avec $1 \leqslant p \leqslant 2^h$ d'où $h = \lfloor \log_2 n \rfloor$, $p = n + 1 - 2^h$ et :

$$L_e(a) = 2p(h+1) + (2^h - p)h = (n+1)h + 2p,$$

quantité indépendante de l'arbre considéré, et donc minimale. L'inégalité $L_e(a) \geqslant (n+1)\lfloor \log_2 n \rfloor$ pour un arbre binaire quelconque à n nœuds est alors immédiate.

2. On représente les comparaisons effectuées par $\mathcal{A}$ lors du tri d'une permutation σ par des questions de la forme : « est-ce que $\sigma(i) \geqslant \sigma(j)$? » où i et j sont deux entiers « choisis » par $\mathcal{A}$ en fonction des réponses obtenues pour les questions précédentes (l'inégalité dans la question pourrait être stricte, mais c'est une distinction sans importance puisqu'on trie des listes d'éléments distincts).

On construit alors un arbre de décision a associé à $\mathcal{A}$ de la manière suivante : la racine de a est étiquetée par la première comparaison effectuée par $\mathcal{A}$ sur une permutation σ. Cette première comparaison est indépendante de σ puisque $\mathcal{A}$ est un algorithme de tri par comparaisons. Les branches de l'arbre de décision sont définies récursivement : la branche gauche est l'arbre de décision associé au tri de toutes les permutations pour lesquelles la réponse à la première question est oui, la branche droite est l'arbre de décision associé au tri des permutations ayant répondu non à la première question. Ces branches sont étiquetées par les questions posées par $\mathcal{A}$ pour le tri des permutations considérées, première question exceptée.

L'arbre a obtenu est un arbre ayant une branche vide pour chaque permutation σ de $[\![1,n]\!]$: cette branche vide est issue de la dernière question que $\mathcal{A}$ a posé lors du tri de σ, et deux permutations distinctes ne peuvent aboutir sur la même branche vide sinon elles seraient indiscernables par $\mathcal{A}$ et $\mathcal{A}$ ne serait pas un algorithme de tri. Il peut y avoir d'autres branches vides correspondant aux questions pour lesquelles toutes les permutations à qui la question a été posée ont répondu de la même manière, soit a' l'arbre déduit de a en supprimant ces questions inutiles. Le nombre de comparaisons effectuées par $\mathcal{A}$ pour trier une permutation σ est égal à la profondeur de la branche vide associée à σ dans a', donc la complexité moyenne de $\mathcal{A}$ est au moins égale au quotient par $n!$ de la longueur du chemin externe de a'. Comme a' a exactement $n!$ branches vides, il a au moins $n! - 1$ nœuds et, d'après la première question, $L_e(a') \geqslant n!\lfloor \log_2(n!-1)\rfloor$.

Exercice 8-1

On examine tous les nœuds en ordre infixe et on vérifie au fur et à mesure que les étiquettes forment une suite croissante.

```
(* vérifie que les étiquettes de a sont en ordre croissant *)
(* et supérieures à m. Retourne la plus grande étiquette   *)
(* trouvée ou m pour un arbre vide                         *)
let rec vérifie compare a m = match a with
| B_vide -> m
| B_noeud(e,g,d) ->
   let g1 = vérifie compare g m in
   let d1 = vérifie compare d e in
   if compare e g1 = PLUSPETIT then failwith "arbre mal formé"
   else d1
;;

let est_de_recherche compare a = match a with
| B_vide -> true
| _      -> try let _ = vérifie compare a (minimum a) in true
            with Failure "arbre mal formé" -> false
;;
```

Exercice 8-2

On peut trier une liste en construisant un arbre binaire de recherche associé et en le parcourant en ordre infixe. Comme le parcours infixe d'un arbre a une complexité linéaire en la taille de l'arbre, on obtient ainsi un algorithme de tri par comparaisons de complexité $T_n + O(n)$, d'où le résultat compte tenu de la complexité intrinsèque d'un tri par comparaisons.

Exercice 8-3

```
(* place les éléments de v.(a..b) dans un arbre de recherche *)
let rec construit v a b =
  if a > b then B_vide
  else if a = b then B_noeud(v.(a),B_vide,B_vide)
  else let c = (a+b)/2 in
       B_noeud(v.(c), construit v a (c-1), construit v (c+1) b)
;;

let arbre_of_vect v = construit v 0 (vect_length(v)-1);;
```

Si $T(n)$ est le temps d'exécution de `arbre_of_vect` pour un sous-vecteur de longueur n alors on a :

$$T(n) \leqslant T\left(\left\lfloor\frac{n-1}{2}\right\rfloor\right) + T\left(\left\lceil\frac{n-1}{2}\right\rceil\right) + K,$$

d'où l'on déduit $T(n) \leqslant \max(T(1), K)n$ par récurrence sur n. Ainsi, $T(n) = O(n)$ et même $T(n) = \Theta(n)$ car chaque élément du sous-vecteur est placé dans l'arbre. Le cas d'une liste chaînée triée peut se traiter de même, mais il faut parcourir la liste pour repérer l'élément médian et ceci doit être fait à chaque niveau de récursion, ce qui donne une complexité $\Theta(n \ln n)$. Il est préférable de recopier la liste dans un vecteur avant de la mettre en arbre.

Exercice 8-4

Par récurrence sur n, on montre que l'arbre obtenu par insertion aux feuilles est identique à celui obtenu par insertion à la racine en inversant l'ordre d'insertion des éléments.

Exercice 8-5

Pour fusionner un arbre a de racine r et un arbre b on découpe b en arbres majoré et minoré par r et on fusionne ces sous-arbres avec les branches de b.

```
let rec fusion compare a b = match a with
| B_vide -> b
| B_noeud(e,g,d) ->
    let (g',d') = découpe compare e b in
    B_noeud(e, fusion compare g g', fusion compare d d')
;;
```

Complexité : soient n_a la taille de a et h_b la hauteur de b. `fusion` est appelé exactement une fois pour chaque nœud et chaque branche vide de a, et le temps de découpage du deuxième arbre est $O(h_b)$ (car les arbres découpés dans b ont une hauteur inférieure ou égale à celle de b). Le temps de fusion dans le pire des cas est donc $O(n_a h_b)$. Ce temps peut être atteint si n_a est négligeable devant h_b et si la hauteur des arbres découpés ne diminue que d'une quantité bornée à chaque découpage.

On pourrait symétriser les rôles de a et b en permutant les arbres à chaque appel récursif, mais l'analyse semble délicate et expérimentalement les arbres ainsi obtenus sont très déséquilibrés. Une solution plus sûre est de parcourir a et b en ordre infixe et de reconstituer un arbre binaire de recherche à partir de la fusion des listes associées.

Exercice 8-6
Dans `quicksort` chaque élément à partir du deuxième est comparé à la tête de la liste, puis le tri se poursuit récursivement sur les sous-listes u et v obtenues. Dans l'insertion aux feuilles le premier élément inséré est la tête de liste et il est placé à la racine de l'arbre en construction. Tous les autres éléments seront comparés au premier puis insérés dans la branche gauche ou droite en fonction du résultat de la comparaison. Donc les éléments insérés dans la branche gauche sont exactement les éléments de la liste u, et ils sont insérés dans l'ordre défini par u, ce qui établit le résultat par récurrence.

Complexité moyenne de `quicksort` : le nombre de comparaisons à effectuer pour insérer un élément dans un arbre binaire de recherche est égal à la profondeur de la position d'insertion moins 1, donc le nombre total de comparaisons effectuées est égal à la somme des profondeurs de tous les nœuds de l'arbre final diminuée de n. Comme la longueur moyenne du chemin interne d'un arbre binaire construit aléatoirement est équivalente à $2n\ln(n)$ (cf. section **7-4**), on en déduit que le nombre moyen de comparaisons effectuées est $\Theta(n \ln n)$.

Exercice 9-1
Oui avant simplification, non après. Par exemple avec $e = \mathtt{CL}[(1,x);(1,y);(1,z)]$, $e' = y$ et $f = \mathtt{CL}[(-1,x)]$ on a $e < e'$ donc $\mathtt{CL}[(1,e);(1,f)] < \mathtt{CL}[(1,e');(1,f)]$, mais après simplification, $\mathtt{CL}[(1,y);(1,z)] > \mathtt{CL}[(-1,x);(1,y)]$. On construit un contre-exemple similaire pour le produit, et en ce qui concerne la substitution, $\mathtt{subs}(y=-x,e) > \mathtt{subs}(y=-x,e')$ après simplification.

Exercice 9-2
Recherche d'une sous-expression :

```
let rec figure e f =
  (compare e f = EQUIV)
  or
  (match e,f with
   | CL(e1), CL(f1) -> contient e1 f1
   | PG(e1), PG(f1) -> contient e1 f1
   | _              -> false)
  or
  (match e with
   | Fct(_,e1)      -> figure e1 f
   | CL(l)          -> figure_liste l f
   | PG(l)          -> figure_liste l f
   | _              -> false)

and figure_liste l f = match l with
| []                -> false
| (_,e)::suite      -> (figure e f) or (figure_liste suite f)
```

```
and contient l l' = match l,l' with
| _,       []        -> true
| [],      _         -> false
| (a::s), (b::s')  -> if b = a then contient s s'
                        else (b > a) &  (contient s l')
;;
```

`figure_liste` parcourt la liste `l` jusqu'à trouver un terme dans lequel figure `f` ou avoir épuisé la liste, et `contient` dit si une liste en contient une autre, ces listes étant supposées triées par ordre croissant.

L'algorithme de substitution suit le même schéma de parcours que `figure`, mais au lieu de s'arrêter sur la première substitution réussie, on doit continuer à parcourir le reste de la formule pour remplacer les autres occurrences éventuelles de `f` par `g`. On effectue une substitution simple, c'est-à-dire que la formule obtenue après substitution n'est pas réexaminée pour chercher de nouvelles substitutions possibles. En effet, une substitution récursive peut ne pas terminer :

$$\texttt{substitue}\ x\ x\ (x^x) : x \longrightarrow x^x \longrightarrow (x^x)^{(x^x)} \longrightarrow ((x^x)^{(x^x)})^{((x^x)^{(x^x)})} \longrightarrow \ldots.$$

```
(* compte-rendu d'une tentative de substitution *)
type 'a compte_rendu = Succès of 'a | Echec;;

(* remplace les occurrences dans e de f par g *)
let rec substitue e f g =

  if compare e f = EQUIV then g

  else let res = match e,f with
   | CL(e1), CL(f1) ->
     (match supprime e1 f1 with
      | Succès(h1) -> Succès(CL((1.0,g)::(subs_liste h1 f g)))
      | Echec      -> Echec)
   | PG(e1), PG(f1) ->
     (match supprime e1 f1 with
      | Succès(h1) -> Succès(PG((1.0,g)::(subs_liste h1 f g)))
      | Echec      -> Echec)
  | _ -> Echec

  in match res with
   | Succès(h) -> h
   | Echec -> match e with
   | Fct(n,e1) -> Fct(n, substitue e1 f g)
   | CL(l)     -> CL(subs_liste l f g)
   | PG(l)     -> PG(subs_liste l f g)
   | _         -> e
```

```
(* idem pour une liste *)
and subs_liste l f g = match l with
   | []           -> []
   | (c,e)::suite -> (c,substitue e f g)::(subs_liste suite f g)

(* supprime les éléments de l' dans l si l' est inclus dans l *)
and supprime l l' = match l,l' with
   | _, []  -> Succès(l)
   | [], _  -> Echec
   | (a::s), (b::s') ->
            if b = a then supprime s s'
       else if b > a then match supprime s l' with
                          | Succès(l'') -> Succès(a::l'')
                          | Echec       -> Echec
       else Echec
;;
```

Exercice 9-3

On note $T(n)$ le nombre maximal de nœuds que peut contenir la dérivée d'une expression de n nœuds. La dérivation de $f_1 \circ \ldots \circ f_{n-1}(x)$ montre que $T(n) \geqslant n^2/2$. Supposons qu'il existe un nombre $K \geqslant 1$ tel que $T(k) \leqslant Kk^2$ pour tout k strictement inférieur à n et considérons une formule e à n nœuds ($n \geqslant 2$).

Si $e = f(u)$ alors $e' = u'f'(u)$ comporte au plus $K(n-1)^2 + a(n-1) + b$ nœuds où a est le nombre d'occurrences de x dans $f'(x)$ et b le nombre de nœuds de $f'(x)$ autres que ces occurrences. En considérant que a et b sont bornés, on peut supposer que $a \leqslant 2K$ et $b \leqslant K$ quitte à changer K, et donc le nombre de nœuds de $u'f'(u)$ est majoré par Kn^2.

Si $e = a_1e_1 + \ldots + a_pe_p$ alors $e' = a_1e'_1 + \ldots a_pe'_p$. Soient $n_1, \ldots, n_p$ les nombres de nœuds de $e_1, \ldots, e_p$: $n_1 + \ldots + n_p = n - 1$ et le nombre de nœuds de e' est majoré par :

$$1 + K(n_1^2 + \ldots + n_p^2) \leqslant 1 + K(n_1 + \ldots + n_p)^2 \leqslant Kn^2.$$

Si $e = e_1^{a_1} \ldots e_p^{a_p}$ où e_i contient n_i nœuds alors $e' = \sum a_ie'_ie_i^{-1}e$, donc le nombre de nœuds de e' est majoré par :

$$\begin{aligned} 1 + p + K(n_1^2 + \ldots + n_p^2) + (p+1)(n_1 + \ldots + n_p) \\ \leqslant K((n-p)^2 + p - 1) + n(p+1) \\ \leqslant Kn^2 + \underbrace{Kp^2 + (K - n(2K-1))p + n - K}_{\varphi(p)} \end{aligned}$$

On a $n_1^2 + \ldots + n_p^2 \leqslant (n-p)^2 + p - 1$ car le p-uplet $(n_1, \ldots, n_p)$ varie dans l'ensemble convexe d'équations $n_1 + \ldots + n_p = n - 1$, $n_i \geqslant 1$, donc sa distance à l'origine est maximale quand ce p-uplet est un point extrémal. Par ailleurs :

$$\varphi(1) = K(1-n) + n(2-K), \qquad \varphi(n-1) = (1-K)n^2 + K(n-1)$$

et ces quantités sont négatives si $K \geqslant 2$ et $n \geqslant 1$. Comme φ est une fonction convexe, on a aussi $\varphi(p) \leqslant 0$ pour $K \geqslant 2$, $p \in [1, n-1]$. L'hypothèse $T(n) \leqslant Kn^2$ est donc récurrente et par conséquent établie. La complexité spatiale est proportionnelle au nombre de nœuds créés (même en tenant compte de la pile de récursion puisque sa hauteur est celle de e) donc est $O(n^2)$. De même, le temps de création d'un nœud étant borné, et la dérivation se résumant à la création de nœuds et à quelques calculs en nombre borné par nœud, la complexité temporelle est elle aussi $O(n^2)$.

Exercice 9-4

Il s'agit en fait d'un problème de parcours de graphe où l'on ne veut passer qu'une seule fois par chaque nœud. Une solution naïve est d'ajouter à la représentation des nœuds un indicateur mutable signalant que le nœud a déjà été traité :

```
type 'a résultat = Inconnu | Calculé of 'a;;

type 'a mexpression = {
  exp         : 'a expression;
  mutable res : 'a mexpression résultat
}

and 'a expression =
| MConst of 'a
| MVar of string
| MFct of string * ('a mexpression)
| MCL of ('a * 'a mexpression) list
| MPG of ('a * 'a mexpression) list
;;
```

Un nœud `n` de type `mexpression` est constitué de deux champs : `n.exp` qui décrit la structure algébrique de `n` et `n.res` qui contient le résultat du calcul en cours si l'on est déjà passé par `n`, ou `Inconnu` sinon. La dérivation s'écrit alors :

```
let fder f u = let r = match f with
| "exp" -> MFct("exp",u)             (* exp'(u) = exp(u)  *)
| "ln"  -> MPG [(-1.0,u)]            (* ln'(u)  = u^-1    *)
| "cos" -> MCL [(-1.0,{exp=MFct("sin",u); res=Inconnu})]
                                     (* cos'(u) = -sin(u) *)
| "sin" -> MFct("cos",u)             (* sin'(u) = cos(u)  *)
  .......                        (* autres dérivées usuelles *)
| _     -> MFct(f^"'",u)             (* f'(u)   = f'(u)   *)
in {exp=r; res=Inconnu}
;;

let rec dérive x m = match m.res with
| Calculé(r) -> r
| Inconnu -> let e = match m.exp with
  | MConst(_)  -> MConst(0.0)
  | MVar(v)    -> if v = x then MConst(1.0) else MConst(0.0)
```

```
    | MFct(f,u)  -> MPG [(1.0,dérive x u); (1.0,fder f u)]
    | MCL(liste) -> MCL(map (dériveCL x) liste)
    | MPG(liste) -> MCL(map (dérivePG liste x) liste)
    in
    let m' = {exp=e; res=Inconnu}
    in m.res <- Calculé(m'); m'

and dériveCL x (a,e) = (a,dérive x e)
and dérivePG liste x (a,e) =
  (a, {exp=MPG( (1.0,dérive x e) :: (-1.0,e) :: liste );
       res=Inconnu})
;;
```

Cette structure de données semble « fonctionner », c'est-à-dire que les branches partagées ne sont dérivées qu'une fois et le partage est conservé dans la dérivée, mais elle a un grave défaut : `dérive x m` modifie physiquement `m` et il faut réinitialiser tous les champs `res` des nœuds de `m` à `Inconnu` avant de calculer une dérivée par rapport à une autre variable, *cette réinitialisation affectant non seulement* `m`, *mais aussi la dérivée que l'on vient de calculer*. Autrement dit, le résultat d'un calcul peut être modifié sans qu'on en soit conscient lors d'un calcul ultérieur !

Une solution moins dangereuse consiste à ranger les dérivées que l'on calcule avec la variable de dérivation dans une table indépendante et à consulter cette table à chaque fois que l'on veut dériver une expression. La technique des *tables de hachage* permet un accès aux formules mémorisées en un temps moyen constant, c'est paraît-il cette technique qui est utilisée dans le logiciel MAPLE.

Exercice 9-5
Soit T_n la taille maximale d'un arbre tenant dans n mots mémoire. On a la relation de récurrence :

$$T_1 = T_2 = 1, \qquad T_n = 1 + \max(pT_{n-p-1}, \quad 1 \leqslant p \leqslant n-2).$$

En particulier pour p fixé, $T_n \geqslant pT_{n-p-1}$ donc T_n croît au moins comme $p^{n/(p+1)}$. L'étude de la fonction $p \longmapsto p^{1/(p+1)}$ montre qu'elle passe par un maximum pour $p = p_0 \approx 3.59$ où p_0 est la racine de l'équation $x \ln x = x + 1$ et on montre (difficilement) que $T_n = O(p_0^{n/(p_0+1)})$ donc T_n a une croissance exponentielle. En particulier $T_{100} > 10^{12}$, c'est-à-dire que l'on peut « tasser » un arbre bien choisi de mille milliards de nœuds dans 100 mots mémoire !

Exercice 9-6

```
let rec simplifie e = match e with
| Const(_) -> e
| Var(_)   -> e

| Fct(f,u) -> begin
    match simplifie(u) with
    | Const(c) -> feval f c
```

```
      | u'        -> Fct(f,u')
      end

  | CL(liste) -> begin
      let l1 = map simp_couple liste in     (* simplifie les termes *)
      let l2 = développeCL(l1)       in     (* distributivité       *)
      let l3 = tri(l2)               in     (* tri et regroupement  *)
      let l4 = constantesCL 0.0 l3   in     (* regr. les constantes *)
      match l4 with
      | []        -> Const(0.0)
      | [(1.0,x)] -> x
      | _         -> CL(l4)
      end

  | PG(liste) -> begin
      let l1 = map simp_couple liste in  (* simplifie les facteurs  *)
      let l2 = développePG(l1)       in  (* distributivité          *)
      let l3 = tri(l2)               in  (* tri et regroupement     *)
      let (c,f) = constantesPG 1.0 l3    (* regr. les constantes    *)
      in match (c,f) with
      | (_,[])          -> Const(c)
      | (0.0,_)         -> Const(0.0)
      | (1.0,[1.0,g])   -> g
      | (1.0,g)         -> PG(g)
      | (_,[1.0,CL(l)]) -> CL(distribue c l)
      | (_,[1.0,g])     -> CL [c,g]
      | _               -> CL [c,PG(f)]
      end

  and simp_couple(a,e) = (a,simplifie e)

  (* éclate les CL. imbriquées *)
  and développeCL(liste) = match liste with
  | []               -> []
  | (a,CL(l))::suite -> (distribue a l) @ (développeCL suite)
  | x::suite         -> x :: (développeCL suite)

  (* éclate les produits imbriqués *)
  and développePG(liste) = match liste with
  | [] -> []
  | (a,PG(l))::suite ->
       (distribue a l) @ (développePG suite)
  | (a,CL([b,PG(l)]))::suite ->
       (a,Const(b)) :: (distribue a l) @ (développePG suite)
  | (a,CL([b,x]))::suite ->
       (a,Const(b)) :: (a,x) :: (développePG suite)
  | x::suite ->
       x :: (développePG suite)
```

```
(* distribue un coefficient ou exposant *)
and distribue c liste = match liste with
| []           -> []
| (a,x)::suite -> (c*.a,x)::(distribue c suite)

(* tri rapide, regroupement et élimination des termes nuls *)
and tri(liste) = match liste with
| []      -> []
| [(a,x)] -> if a = 0.0 then [] else [(a,x)]
| (a,x)::suite ->
     let (b,l1,l2) = sépare (a,[],[]) x suite in
     if b = 0.0 then (tri l1) @ (tri l2)
                else (tri l1) @ [(b,x)] @ (tri l2)

and sépare (a,l1,l2) x liste = match liste with
| []           -> (a,l1,l2)
| (b,y)::suite -> if y < x then sépare (a,(b,y)::l1,l2) x suite
             else if y > x then sépare (a,l1,(b,y)::l2) x suite
             else sépare(a+.b,l1,l2) x suite

(* additionne les constantes *)
and constantesCL c liste = match liste with
| (a,Const(b))::suite -> constantesCL (a*.b +.c) suite
| _ -> if c = 0.0 then liste else (1.0,Const(c))::liste

(* multiplie les constantes *)
and constantesPG c liste = match liste with
| (a,Const(b))::suite -> constantesPG (c *. b**.a) suite
| _ -> (c,liste)

(* évaluation d'une fonction *)
and feval f u =  match f with
| "exp" -> Const(exp u)
| "ln"  -> Const(log u)
| "cos" -> Const(cos u)
| "sin" -> Const(sin u)
  .......                        (* autres fonctions usuelles *)
| _     -> Fct(f,Const(u))       (* fonction non évaluable    *)
;;
```

Remarque : on trie les listes des combinaisons linéaires et produits généralisés par l'algorithme du tri rapide, modifié pour regrouper en même temps les sous-expressions égales au pivot. La complexité de ce tri est $O(n \ln n)$ *en moyenne* seulement. Par ailleurs, on utilise la comparaison polymorphe `<` qui classe les constantes avant tous les autres types d'expressions, ceci parce que le constructeur `Const` est déclaré en premier dans la définition du type `expression`. On est ainsi assuré que tous les termes constants figureront en début de liste après classement, ce qui simplifie le regroupement des constantes. Cette simplification est probablement « non portable » et devra être revue si l'on change de compilateur.

Exercice 9-7
D'après l'exercice **9-3**, si e est une expression à n nœuds alors $\partial e/\partial x$ a $O(n^2)$ nœuds donc la simplification de $\partial e/\partial x$ prend un temps $O(n^4 \ln n)$. Si l'on simplifie au fur et à mesure les sous-expressions que l'on dérive, en notant $T(e)$ le temps de calcul et simplification de $\partial e/\partial x$, on a :

$$\begin{aligned} T(e) &\leqslant \alpha \quad \text{lorsque } e \text{ est une constante ou une variable,} \\ T(f(u)) &\leqslant T(u) + \beta n^2 \ln n, \\ T(a_1 e_1 + \ldots + a_p e_p) &\leqslant T(e_1) + \ldots + T(e_p) + \beta n^2 \ln n, \\ T(e_1^{a_1} \ldots e_p^{a_p}) &\leqslant T(e_1) + \ldots + T(e_p) + \beta n^2 \ln n. \end{aligned}$$

On en déduit que $T(e) = O(n^3 \ln n)$. Il *semble* donc préférable de simplifier à chaque étape de la dérivation plutôt que d'attendre la fin, la complexité ayant un meilleur majorant. Mais il s'agit de majorants seulement et il n'est pas évident qu'ils soient optimaux. Compte tenu du partage important de sous-expressions lors de la dérivation, on peut aussi envisager de dériver, simplifier et mémoriser chaque sous-expression, tout cela en même temps.

Exercice 10-1

1. Si $|u| \leqslant |x|$ alors la relation $uv = xy$ implique que u est un préfixe de x donc il existe $z \in A^*$ tel que $x = uz$. On a alors $uv = uzy$ d'où $v = zy$. Le cas $|u| > |x|$ se traite de même.

2. Si x et y ont mêmes longueurs on a immédiatement $x = y$. Sinon, supposons $|x| > |y|$: puisque $xy = yx$ il existe z tel que $x = yz$ et $x = zy$ donc y et z commutent et on conclut par récurrence sur la longueur du plus grand des deux mots.

Exercice 10-2
$L_1 = A^*\mathtt{a}A^*\mathtt{v}A^*\mathtt{i}A^*\mathtt{o}A^*\mathtt{n}A^*$.
$L_2 = \mathtt{n}'^* + \mathtt{o}'^*\mathtt{n}\mathtt{n}'^* + \mathtt{i}'^*\mathtt{o}\mathtt{o}'^*\mathtt{n}\mathtt{n}'^* + \mathtt{v}'^*\mathtt{i}\mathtt{i}'^*\mathtt{o}\mathtt{o}'^*\mathtt{n}\mathtt{n}'^* + \mathtt{a}'^*\mathtt{v}\mathtt{v}'^*\mathtt{i}\mathtt{i}'^*\mathtt{o}\mathtt{o}'^*\mathtt{n}\mathtt{n}'^*$ où x' désigne toutes les lettres sauf x. Un élève m'a signalé l'expression plus simple :

$$L_2 = \mathtt{a}'^*\mathtt{v}'^*\mathtt{i}'^*\mathtt{o}'^*\mathtt{n}'^*$$

qui se déduit de la mienne par factorisations à droite successives. Montrons la validité de cette dernière expression : soit $u = u_a u_v u_i u_o u_n$ avec $u_x \in x'^*$. Si u contient les lettres a, v, i, o dans cet ordre alors ces lettres appartiennent respectivement aux facteurs $u_v u_i u_o u_n$, $u_i u_o u_n$,$u_o u_n$ et u_n donc il ne peut y avoir de n dans u après le o. Réciproquement, si u ne contient pas les lettres a, v, i, o, n dans cet ordre, alors on note u_a le plus long préfixe de u sans a, m_a le suffixe tel que $u = u_a m_a$, puis u_v le plus long préfixe de m_a sans v, m_v le suffixe correspondant, et ainsi de suite. On a donc $u = u_a u_v u_i u_o u_n m_n$ avec $u_x \in x'^*$ et si $m_n \neq \varepsilon$ alors m_a, m_v, m_i, m_o et m_n sont tous différents de ε donc commencent par les lettres a, v, i, o et n, ce qui est contraire à l'hypothèse faite sur u.

Exercice 10-3

Soit X l'ensemble des chiffres autres que 1 et 3 et L le langage cherché. On a :

$$L = (X + 3)^*(1^*1X(X + 3)^*)^*1^*.$$

En effet, si u est un mot quelconque, on peut découper u en facteurs séparés par des suites de 1, c'est-à-dire $u = u_1 1^{n_1} \ldots u_p 1^{n_p}$ où les u_i ne contiennent pas le chiffre 1, $u_i \neq \varepsilon$ pour $i \geqslant 2$ et $n_1 \geqslant 1$, $\ldots n_{p-1} \geqslant 1$, $n_p \geqslant 0$. Alors $u \in L$ si et seulement si $u_2, \ldots, u_p$ ne commencent pas par 3, soit $u_1 \in (X+3)^*$ et $u_i \in X(X+3)^*$ pour $i \geqslant 2$. En découpant un mot suivant les suites de 3, on obtient une autre expression :

$$L = 3^*((X + 1)^*X33^*)^*(X + 1)^*.$$

Exercice 10-4

1. $0^*1^*2^*3^*4^*5^*6^*7^*8^*9^*$.
2. Le langage des suites arithmétiques de raison nulle est : $0^* + 1^* + \ldots + 9^*$. Les suites arithmétiques de raison non nulle sont en nombre fini (puisque les termes sont bornés), il suffit de les lister toutes.

Exercice 10-5

On note a, b, c, d, e les caractères `(`, `)`, `{`, `}` et `*` et a', b', c', d', e' les ensembles complémentaires dans l'ensemble de tous les caractères. Alors le langage des commentaires est défini par :

$$L = aeL_1eb + cd'^*d$$

où L_1 est le langage des mots ne comportant pas la séquence `*)`. On a, comme à l'exercice **10-3** :

$$L_1 = e^*(e'^*(b + e)'^*bb^*)^*e'^*.$$

Exercice 10-6

1. Soit X solution de $X = KX+L$. On a donc $X = K(KX+L)+L = K^2X+(K+\varepsilon)L$ puis de proche en proche :

$$X = K^nX + (K^{n-1} + \ldots + K + \varepsilon)L$$

pour tout entier $n \geqslant 1$. Comme K ne contient pas le mot vide, la longueur d'un mot de K^nX est au moins égale à n, donc tout mot de X appartient à l'un des langages $(K^{n-1} + \ldots + K + \varepsilon)L$ pour n assez grand. Ceci prouve que $X \subset K^*L$. Réciproquement, tout mot de K^*L appartient à un langage de la forme $(K^{n-1} + \ldots + K + \varepsilon)L$ pour n convenable, donc appartient aussi à X, d'où $X = K^*L$. Enfin, le langage K^*L est effectivement solution de l'équation : $KK^*L + L = (KK^* + \varepsilon)L = K^*L$.

Remarque : si K contient le mot vide alors K^*L est encore solution de l'équation, mais il n'y a plus unicité car tout langage contenant K^*L est aussi solution.

2. Le système :

$$\begin{cases} X = KX + LY + M \\ Y = K'X + L'Y + M' \end{cases}$$

peut être résolu par substitution. La première équation donne :

$$X = K^*(LY + M) = K^*LY + K^*M$$

que l'on reporte dans la deuxième :

$$Y = (K'K^*L + L')Y + K'M + M'$$

et $K'K^*L + L'$ ne contient pas ε, donc on peut calculer Y puis X. Les expressions obtenues pour Y et X sont clairement régulières.

Exercice 10-7

En appliquant les règles : $M(N + P) = MN + MP$, $(N + P)M = NM + PM$ et $(M + N)^* = (M^*N)^*M^*$, on peut éliminer tous les $+$ ou les remonter au niveau d'imbrication zéro dans une expression régulière définissant L. On applique alors les règles : $\varnothing^* = \varepsilon^* = \varepsilon$, $\varnothing M = M\varnothing = \varnothing$ et $\varepsilon M = M = M\varepsilon$ pour éliminer les $\varnothing$ et ε internes. Comme $\varepsilon \notin L$, il ne reste plus de ε au niveau zéro. On obtient ainsi une expression régulière pour L sous forme d'une somme d'expressions dont une au moins n'est pas $\varnothing$ car $L \neq \varnothing$, donc on peut éliminer tous les $\varnothing$ de cette somme.

Exercice 10-8

Soit M un langage défini par une expression régulière de longueur $n \geqslant 1$. On montre par récurrence sur n que si M est inclus dans un L_p alors M est fini.

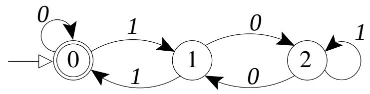
Pour $n = 1$, $M = \varepsilon$ ou $M = \varnothing$ ou M est réduit à une lettre, donc est fini. Pour $n \geqslant 1$, M est de l'une des formes :

- $M = M_1 + M_2$: M_1 et M_2 sont inclus dans le même L_p que M et sont définis par des expressions plus courtes, donc sont finis.

- $M = M_1^*$: si $M_1 = \varnothing$ alors $M = \varepsilon$ est fini. Sinon, si $u \in M_1$, alors $|u|_b - |u|_a = p$ car $u \in M \subset L_p$ et il en est de même pour u^2 ce qui implique $p = 0$. Mais le seul mot de $L_0 = L$ dont le carré soit aussi dans L_0 est le mot vide, donc M_1 est réduit à ε et $M = M_1^* = \varepsilon$ est fini.

- $M = M_1M_2$: si M_1 ou M_2 est vide alors M est vide. Sinon, soient $u \in M_1$ et $v \in M_2$. Comme $uv \in L_p$ on a $|u|_b - |u|_a = p - |v|_b + |v|_a = q$ et u est constitué d'une suite de a suivie d'une suite de b, donc $u \in L_q$. Comme l'entier q est indépendant de u d'après son expression en fonction de v, on a en fait $M_1 \subset L_q$ et donc M_1 est fini par hypothèse de récurrence. Le même raisonnement s'applique à M_2 et donc M est aussi fini.

Exercice 11-1
On construit un automate simulant l'évaluation de $\sum a_k 2^k \bmod 3$ par l'algorithme de HÖRNER :



Exercice 11-2
On interprète les mots sur $\{0, 1\}$ comme les représentations binaires de nombres entiers naturels avec bit de poids faible en tête. Notons $f(x)$ (resp. $g(x), h(x)$) le mot émis par l'automate lorsqu'il part de l'état 0 (resp. $1, 2$) et lit le nombre x, complété par suffisamment de zéros pour rejoindre l'état final (ce nombre est bien défini puisque l'automate émet 0 lorsqu'il lit un zéro à partir de l'état 0). On a :

$$\begin{array}{lll} f(0) = 0, & f(2x) = 2f(x), & f(2x+1) = 2g(x)+1, \\ g(0) = 1, & g(2x) = 2f(x)+1, & g(2x+1) = 2h(x), \\ h(0) = 2, & h(2x) = 2g(x), & h(2x+1) = 2h(x)+1. \end{array}$$

Voici les premières valeurs prises par f, g, h :

x	0	1	2	3	4	5	6	7	8	9	10
$f(x)$	0	3	6	9	12	15	18	21	24	27	30
$g(x)$	1	4	7	10	13	16	19	22	25	28	31
$h(x)$	2	5	8	11	14	17	20	23	26	29	32

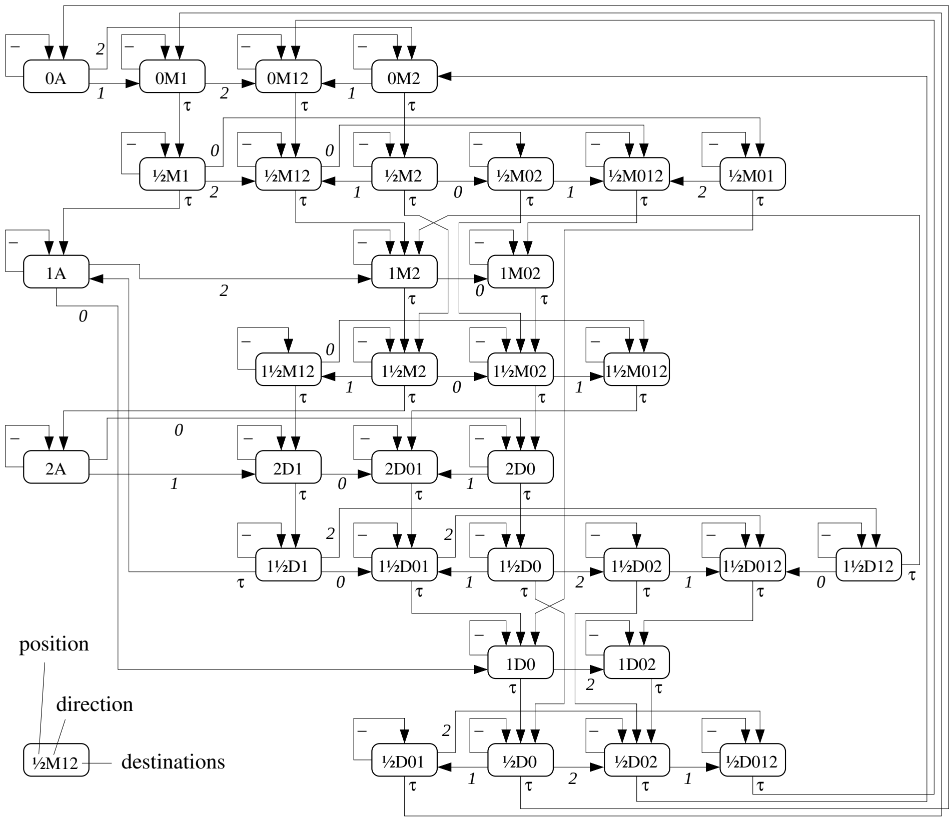
On conjecture alors que $f(x) = 3x$, $g(x) = 3x+1$ et $h(x) = 3x+2$, et l'on établit facilement la validité de cette conjecture par récurrence sur x. En conclusion, l'automate séquentiel prend un nombre x écrit en binaire avec les bits de poids faible en tête et complété par suffisamment de zéros, et émet le nombre $3x$.

Exercice 11-3
L'automate $A \times B$ permet de faire fonctionner en parallèle les automates A et B sur un même mot en prenant comme états initiaux les couples constitués d'un état initial pour A et un état initial pour B. Les états finals sont choisis en fonction du langage à reconnaître : pour $L_A \cap L_B$ un état de $A \times B$ est final si et seulement si ses composantes sont des états finals dans A et B. De même pour $L_A \cup L_B$ et $L_A \setminus L_B$.

Exercice 11-4
On définit comme états de l'ascenseur l'agrégation des informations suivantes :

- sa position (nombre entier pour un étage, nombre demi entier entre deux étages) ;
- sa direction (montée, descente, arrêt) ;
- quels étages ont été demandés et non encore servis.

L'alphabet d'entrée est constitué de trois lettres $0, 1, 2$ représentant chacune un étage d'appel (indifféremment à l'intérieur ou à l'extérieur de la cabine), et d'une quatrième lettre, τ, représentant le temps qui passe et qui sera émise à intervalles réguliers pour forcer l'ascenseur à changer d'état lorsqu'il n'y a pas d'appels.



Exercice 11-5

Première étape : calcul des clôtures par ε-transitions.

On représente les ensembles d'états par des vecteurs de booléens et il s'agit de construire une matrice `m` telle que `m.(i).(j)` vaut `true` si et seulement si l'on peut passer de l'état `i` à l'état `j` par ε-transitions. Dans un premier temps on initialise `m` avec toutes les ε-transitions connues, puis on applique l'algorithme de Warshall pour trouver les états accessibles par ε-transitions généralisées.

```
let clôture(n,liste) =
(* n = nombre d'états, liste = liste des e-transitions *)
(* retourne la matrice d'incidence m                  *)

  (* création et initialisation de la matrice d'incidence *)
  let m = make_matrix n n false in
  for i = 0 to n-1 do m.(i).(i) <- true done;
  do_list (fun (x,y) -> m.(x).(y) <- true) liste;
```

```
    (* calcule la clôture par l'algorithme de Warshall *)
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        for k = 0 to n-1 do
          m.(j).(k) <- m.(j).(k) or (m.(j).(i) & m.(i).(k))
    done done done;

   m (* résultat *)
;;
```

On prouve la correction de `clôture` avec l'invariant de boucle :

> *à l'entrée de la boucle* `for i`*, pour tous* `j` *et* `k`*,* `m.(j).(k)` *vaut* `true` *si et seulement s'il existe un chemin par ε-transitions menant de l'état* `j` *à l'état* `k` *et passant par des états intermédiaires de numéro strictement inférieur à* `i`.

Complexité : $O(n^3)$ de manière évidente (si `liste` ne comporte pas de répétitions et donc est de taille inférieure ou égale à n^2).

Deuxième étape : calcul des transitions sur lettre.
A l'aide de la matrice `m` obtenue précédemment et de la liste des transitions sur lettre fournie en argument, on construit la matrice `t` de transition généralisée de l'automate, telle que `t.(i).(a)` est l'ensemble des états accessibles à partir de l'état `i` par transition sur la lettre `a` puis ε-transitions.

```
 let transitions_g(n,p,liste,m) =
 (* n = nombre d'états, p = nombre de lettres    *)
 (* liste = liste des transitions sur lettre     *)
 (* m = matrice d'incidence                      *)
 (* renvoie la matrice de transition généralisée *)

   let t = make_matrix n p [||] in
   for i = 0 to n-1 do for a = 0 to p-1 do
     t.(i).(a) <- make_vect n false
   done done;

   (* initialise avec les transitions fournies *)
   do_list (fun (x,a,i) ->
              for y = 0 to n-1 do
                if m.(i).(y) then t.(x).(a).(y) <- true
              done)
           liste;

   t (* résultat *)
;;
```

La complexité de cette étape est $O(pn^3)$ en considérant que `liste` contient au plus pn^2 triplets (c'est-à-dire est sans répétitions).

Troisième étape : calcul des ensembles d'états accessibles depuis les états initiaux.
On applique la méthode vue en cours : ne générer que les ensembles d'états nécessaires. Pour cela on part de l'ensemble des états initiaux (sous forme de vecteur de booléens) et on essaye toutes les transitions jusqu'à ce qu'il n'y ait plus de nouvel état créé. De façon à garder un temps d'accès constant, les états déjà créés sont conservés dans un vecteur de longueur variable dont la taille est doublée à chaque fois qu'il menace de déborder (ce doublement garantit un temps d'insertion moyen constant).

```
type 'a mémoire = {
   mutable v : 'a vect; (* vecteur sur-dimensionné    *)
   mutable l : int};;   (* nombre d'éléments présents *)

(* cherche "état" dans la mémoire "mem" et l'insère au besoin *)
(* "mem" est modifiée sur place, renvoie l'indice de "état".  *)
let numéro(mem,état) =

  let i = ref(0) in
  while (!i < mem.l) & (mem.v.(!i) <> état) do incr i done;
  if !i < mem.l then !i (* état déjà enregistré *)

  else begin

    (* état non trouvé, il faut l'insérer *)
    if mem.l = vect_length(mem.v) then begin
      (* plus de place, agrandit le tampon *)
      let w = make_vect (2*mem.l) état in
      for j = 0 to mem.l - 1 do w.(j) <- mem.v.(j) done;
      mem.v <- w
    end;

    (* maintenant il y a assez de place, insère "état" *)
    mem.v.(mem.l) <- état;
    mem.l <- mem.l + 1;
    !i (* indice de "état" *)

  end
;;
```

Il n'y a plus qu'à calculer les sous-ensembles obtenus à partir du sous-ensemble initial et à construire la nouvelle table de transition :

```
(* n = nombre d'états (autom. indéterministe)*)
(* p = nombre de lettres                     *)
(* initiaux = liste des états initiaux       *)
(* finals   = liste des états finals         *)
(* m = matrice d'incidence                   *)
(* t = matrice de transition généralisée     *)
```

```
(* renvoie le nombre d'états, la liste des   *)
(* transitions et la liste des états finals  *)
(* du nouvel automate                        *)

let construit(n,p,initiaux,finals,m,t) =

  (* transforme "initiaux" en vecteur de booléens *)
  (* et complète par epsilon-transitions         *)
  let e = make_vect n false in
  let f(x) = for i = 0 to n-1 do e.(i) <- e.(i) or m.(x).(i) done
  in do_list f initiaux;

  (* initialise la mémoire et la nouvelle liste   *)
  (* de transitions                               *)
  let mem = {v = [|e|]; l=1} and trans = ref([]) in

  (* boucle sur tous les ensembles d'états générés *)
  let num = ref(0) in while !num < mem.l do

    (* traitement de l'ensemble de numéro !num *)
    let e = mem.v.(!num) in
    for l = 0 to p-1 do
      (* calcule les états accessibles avec la lettre "l" *)
      let e' = make_vect n false in
      for i = 0 to n-1 do if e.(i) then for j = 0 to n-1 do
        e'.(j) <- e'.(j) or t.(i).(l).(j)
      done done;
      let n' = numéro(mem,e') in trans := (!num,l,n') :: !trans
    done;
    incr num

  done;

  (* terminé, il reste à déterminer les états finals *)
  let f = ref([]) in
  for i = 0 to mem.l-1 do
    if exists (fun x -> mem.v.(i).(x)) finals then f := i :: !f
  done;
  (mem.l, !trans, !f) (* renvoie les résultats promis *)
;;

(* déterminisation *)
let déterminise(n,p,teps,tlettre,initiaux,finals) =
  let m = clôture(n,teps)                            in
  let t = transitions_g(n,p,tlettre,m)               in
  let (n',t',f') = construit(n,p,initiaux,finals,m,t) in
  (n',t',0,f')
;;
```

```
(* exemple du cours : a -> 0, b -> 1 *)
déterminise( 5,                   (* n *)
             2,                   (* p *)
             [],                  (* teps *)
             [(0,0,1); (0,0,4);
              (1,1,0); (1,1,2);
              (2,0,1); (2,0,3);
              (3,0,2); (3,0,4);
              (4,1,3); (4,1,0)],  (* tlettre  *)
             [0],                 (* initiaux *)
             [0]                  (* finals   *)
);;
- : int * (int * int * int) list * int * int list =
5,                                (* n'    *)
[4, 1, 3;   4, 0, 4;              (* 1234  *)
 3, 1, 2;   3, 0, 4;              (* 023   *)
 2, 1, 2;   2, 0, 2;              (* rebut *)
 1, 1, 3;   1, 0, 2;              (* 14    *)
 0, 1, 2;   0, 0, 1],             (* 0     *)
0,                                (* init. *)
[3; 0]                            (* final *)
```

Complexité : la recherche et l'insertion éventuelle d'un nouvel état dans la mémoire prennent un temps $O(N)$ où N est le nombre d'états déjà enregistrés en supposant que les comparaisons entre états prennent un temps constant. La constitution d'une transition prend un temps $O(n^2)$ compte non tenu de la recherche de l'état créé dans la mémoire. Le temps total de création du nouvel automate en négligeant les durées d'initialisation est donc $O(n^3 + Nn^2 + N^2)$ où N est le nombre total d'états créés. On pourrait éliminer la partie N^2 due aux recherches dans la mémoire en utilisant une mémoire plus rapide (par exemple un arbre binaire de recherche). Notons que la technique consistant à doubler la taille du vecteur mémoire donne un temps moyen d'insertion constant, mais que cela n'a pas d'effet sur la complexité totale puisque les recherches ont une complexité linéaire.

Exercice 11-6

1. Soit $u \in A^*$: si $u \in L_{\mathcal{A}}$ alors il existe une suite de transitions indexée par u faisant passer $\mathcal{A}$ d'un état initial à un état final et par définition les états appartenant à cette suite sont accessibles et coaccessibles donc appartiennent à $\mathcal{A}'$. La même suite de transitions fait passer $\mathcal{A}'$ d'un état initial à un état final donc $u \in L_{\mathcal{A}'}$. La réciproque est immédiate.

2. Calcul de l'ensemble des états accessibles : pour chaque état q de $\mathcal{A}$ on détermine l'ensemble S_q de tous les états accessibles à partir de q avec une ε-transition ou une transition sur lettre (ensemble des successeurs de q dans $\mathcal{A}$). L'ensemble E des états accessibles est alors obtenu par l'algorithme suivant :

   $E \longleftarrow \varnothing$; $F \longleftarrow \{$*états initiaux de* $\mathcal{A}\}$.
   tant que $F \neq \varnothing$ *faire :* $E \longleftarrow E \cup F$; $F \longleftarrow \left(\bigcup_{q\in F} S_q\right) \setminus E$ *fin tant que.*
   retourner E.

L'ensemble E retourné ne contient manifestement que des états accessibles et si q est un état accessible alors on montre que q est placé dans E par récurrence sur la longueur d'un chemin dans $\mathcal{A}$ reliant un état initial à q. L'ensemble des états coaccessibles s'obtient de même à partir de l'ensemble des états finals de $\mathcal{A}$ et des ensembles de prédécesseurs d'un état.

3. $\mathcal{A}$ est déterministe si et seulement si les langages d'entrée des états de $\mathcal{A}$ sont deux à deux disjoints.

4. $\mathcal{B}$ et $\mathcal{C}$ reconnaissent le langage $\widetilde{L_{\mathcal{A}}}$ donc $\mathcal{D}$ et $\mathcal{E}$ reconnaissent le langage $L_{\mathcal{A}}$. En particulier $\mathcal{E}$ est équivalent à $\mathcal{A}$.

L'algorithme de déterminisation par la méthode des sous-ensembles sans état rebut ne produit que des états accessibles donc tous les états de $\mathcal{C}$ sont accessibles, et par conséquent tous les états de $\mathcal{D}$ sont coaccessibles. Les états de $\mathcal{E}$ sont des ensembles d'états de $\mathcal{D}$ non vides (car on déterminise sans état rebut), ils sont eux aussi tous coaccessibles.

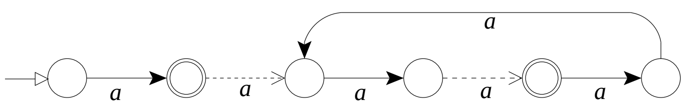
$\mathcal{C}$ étant déterministe à états accessibles, les langages d'entrée des états de $\mathcal{C}$ sont deux à deux disjoints non vides donc les langages de sortie des états de $\mathcal{D}$ sont eux aussi deux à deux disjoints et non vides. Alors si $\{q_1, \dots, q_n\}$ est un état de $\mathcal{E}$, son langage de sortie dans $\mathcal{E}$ est la réunion des langages de sortie dans $\mathcal{D}$ des états $q_1, \dots, q_n$, elle est distincte de toute autre réunion associée à un sous-ensemble différent de $\{q_1, \dots, q_n\}$.

5. Soient $q_1, \dots, q_n$ les états de $\mathcal{E}$ et, pour chaque i, u_i un mot de $L_{\mathcal{A}}$ menant $\mathcal{E}$ de son état initial à q_i. Soit $\mathcal{E}'$ un automate déterministe quelconque reconnaissant $L_{\mathcal{A}}$ et q'_i l'état de $\mathcal{E}'$ atteint à partir de son état initial par la suite de transitions indexée par u_i. Les langages de sortie des q_i dans $\mathcal{E}$ sont deux à deux distincts donc il en est de même pour les langages de sortie des q'_i dans $\mathcal{E}'$. En particulier les états q'_i sont deux à deux distincts et $\mathcal{E}'$ a au moins autant d'états que $\mathcal{E}$.

Ainsi $\mathcal{E}$ est un automate déterministe équivalent à $\mathcal{A}$, minimal en nombre d'états. On peut démontrer en prolongeant le raisonnement précédent que si $\mathcal{E}'$ a autant d'états que $\mathcal{E}$ alors $\mathcal{E}$ et $\mathcal{E}'$ sont isomorphes par la correspondance $q_i \longmapsto q'_i$.

Exercice 11-7

Si $L = F + G(a^n)^*$ alors L est régulier. Si L est fini, $L = L + \varnothing(a^0)^*$. Si L est infini, soit $\mathcal{A}$ un automate déterministe reconnaissant L. Étant déterministe, $\mathcal{A}$ a la forme suivante :



On note n la longueur de la boucle, F le langage menant de l'état initial à un état final avant la boucle et G le langage menant de l'état initial à un état final intérieur à la boucle sans emprunter l'arc retour. Alors $L = F + G(a^n)^*$.

Exercice 11-8

Soit $\mathcal{A}$ un automate fini reconnaissant L. Si x est un état de $\mathcal{A}$, on note I_x le langage menant de l'état initial de $\mathcal{A}$ à x et F_x le langage menant de x à un état final de $\mathcal{A}$. Alors I_x et F_x sont réguliers (thm. de KLEENE) et $\sqrt{L}$ est la réunion pour tous les états x de $\mathcal{A}$ des langages réguliers $I_x \cap F_x$.

La réciproque de cette propriété est fausse, c'est-à-dire que si $\sqrt{L}$ est régulier, il n'en va pas nécessairement de même pour L. Par exemple si $L = \{u^2 \text{ tq } u \in A^*\}$ où A possède au moins deux lettres, alors $\sqrt{L} = A^*$ est régulier mais L ne l'est pas car il admet une infinité de langages résiduels : $(ab^n)^{-1}L \ni ab^n \notin (ab^p)^{-1}L$ pour $n \neq p$.

Exercice 11-9

On procède par récurrence sur la longueur d'une expression régulière définissant un langage L.

- Si $L = \varnothing$, $L = \varepsilon$ ou L est une lettre, le résultat est évident.
- Si $L = M + N$ alors pour $u \in A^*$ on a $u^{-1}L = u^{-1}M + u^{-1}N$ donc M et N ayant un nombre fini de résiduels, il en va de même pour L.
- Si $L = MN$, notons $M_1, \dots, M_p$ les résiduels de M et $N_1, \dots, N_q$ ceux de N. Alors tout résiduel de MN est la réunion d'un M_iN et de certains N_j, donc L a au plus $p2^q$ résiduels (plus précisément, si $u \in A^*$ alors $u^{-1}L = (u^{-1}M)N + \sum\limits_{\substack{u=vw\\ v\in M}} w^{-1}N$).
- Si $L = M^*$, alors les résiduels de L sont des réunions de M_iM^* où $M_1, \dots, M_p$ sont les résiduels de M : $u^{-1}M^* = \sum\limits_{\substack{u=vw\\ v\in M^*}} (w^{-1}M)M^*$.

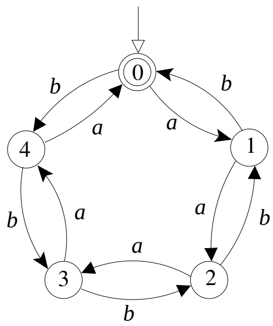
Exercice 11-10

1.

```
let vérifie(v) =
  let n = vect_length(v) and i = ref(0) and j = ref(0) in
  while (!i  < n) & (v.(!i) = `a`) do i := !i+1 done;
  j := !i;
  while (!j  < n) & (v.(!j) = `b`) do j := !j+1 done;
  (!i*2 = n) & (!j = n)
;;
```

2. l'ordinateur ne reconnaît pas le langage des parenthèses emboîtées, mais seulement le langage des parenthèses emboîtées de longueur bornée par sa capacité mémoire et par le plus grand entier représentable en machine.

Exercice 11-11

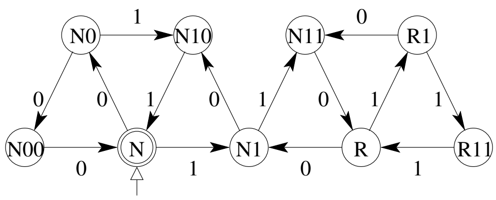


L'automate ci-contre reconnaît le langage L_5 qui est donc un langage régulier (le numéro d'un état correspond à la différence $|x|_a - |x|_b \bmod 5$, où x est le préfixe déjà lu du mot à analyser). Par contre, pour tous $n, p \in \mathbf{N}$ avec $n \neq p$ on a $b^n \in (a^n)^{-1}L$ et $b^n \notin (a^p)^{-1}L$ donc les résiduels $(a^n)^{-1}L$ sont deux à deux distincts ce qui prouve l'irrégularité de L.

Exercice 11-12

1. Si $n, p \in \mathbf{N}$ alors parmi les résiduels $(a^n b)^{-1}L$, seul $(a^p b)^{-1}L$ contient c^{p+1}, donc ces résiduels sont distincts et en nombre infini.

2. On construit un automate « additionneur » qui effectue l'addition en binaire des nombres m et n et vérifie au fur et à mesure que le chiffre de p correspondant est le bon. Les états de l'automate ci-dessous sont notés N (pas de retenue, on attend un chiffre de m), Nx (pas de retenue, on a lu le chiffre x de m et on attend celui de n), Nxy (pas de retenue, on a lu les chiffres x de m et y de n) et de même pour R, Rx et Rxy lorsqu'il y a une retenue en attente.



Exercice 11-13

On montre que L vérifie le lemme de l'étoile avec $n = 3$. Soit $u = p\tilde{p}q$ avec $p \in A^+$ et $q \in A^*$. On décompose u sous la forme $u = xyz$ avec $|xy| \leqslant 3$, $|y| \geqslant 1$, et $xy^kz \in L$ pour tout $k \in \mathbf{N}$ de la manière suivante :

- Si $|p| = 1$ alors on choisit $x = p\tilde{p}$, $y =$ la première lettre de q (qui existe puisque $|u| \geqslant 3$) et $z =$ le reste de q. Donc $xy^kz = p\tilde{p}y^kz \in L$ pour tout $k \in \mathbf{N}$.

- Si $|p| \geqslant 2$ alors on choisit $x = \varepsilon$, $y =$ la première lettre de p et $z =$ le reste de u. On a $p = yr$ avec $r \in A^+$ donc $xy^0z = r\tilde{r}yq \in L$, $xy^1z = u \in L$ et si $k \geqslant 2$ alors $xy^kz = y^kr\tilde{r}yq \in L$ car y est une lettre.

Ainsi, L vérifie le lemme de l'étoile. Pourtant L n'est pas régulier, car si a, b sont deux lettres distinctes alors les résiduels $L_n = (ab^n)^{-1}L$ sont deux à deux distincts quand n décrit $\mathbf{N}$: L_n est le seul de ces résiduels à contenir le mot b^na.

Exercice 11-14

On montre par récurrence sur la taille de $\mathcal{E}$ que l'évaluation de $\mathcal{E}'$ retourne :

0 si et seulement si $L = \varnothing$,
1 si et seulement si $L = \varepsilon$,
2 si et seulement si L est fini différent de $\varnothing$ et ε,
∞ si et seulement si L est infini.

Solutions des problèmes

Tri par distribution

Question 1-a
$a_0 = a \bmod 2$.

Question 1-b
$\lfloor a/2 \rfloor = a_1 + 2a_2 + 4a_3 + \ldots + 2^{n-2}a_{n-1}$.

Question 1-c
```
let rec bit_r p a = if p = 0 then a mod 2 else bit_r (p-1) (a/2);;

let bit_i p a =
  let x = ref(a) in
  for i = 0 to p-1 do x := !x/2 done;
  !x mod 2
;;
```

Remarque : il existe des méthodes de calcul du p-ème bit de a en temps constant, par exemple : `let bit p a = (a lsr p) mod 2;;`

Question 2-a

permute(v):
  $i \longleftarrow 0$; $j \longleftarrow 0$
  pour $k = 0, 1, \ldots, n-1$ *faire :*
    si $v.(k)$ *est pair alors* $v.(i) \longleftarrow v.(k)$*;* $i \longleftarrow i+1$
        sinon $w.(j) \longleftarrow v.(k)$*;* $j \longleftarrow j+1$
  fin pour
  pour $k = 0, 1, \ldots, j-1$ *faire* $v.(i+k) \longleftarrow w.(k)$ *fin pour*
fin

La validité se démontre à l'aide de l'invariant de boucle suivant : *à l'entrée dans la boucle, on a* $i + j = k$ *et les sous-vecteurs* $v.(0\ldots i-1)$ *et* $w.(0\ldots j-1)$ *constituent une partition du sous-vecteur initial* $v.(0\ldots k-1)$ *telle que tous les éléments pairs sont dans* v *et tous les impairs dans* w*, l'ordre initial des éléments étant conservé dans chaque groupe.*

Question 2-b
```
let permute v =
  let n = vect_length(v) in
  let w = make_vect n 0  in
  let i = ref(0) and j = ref(0) in
  for k = 0 to n-1 do
    if v.(k) mod 2 = 0 then begin v.(!i) <- v.(k); i := !i+1 end
                       else begin w.(!j) <- v.(k); j := !j+1 end;
  done;
  for k = 0 to !j-1 do v.(!i+k) <- w.(k) done
;;
```

Question 2-c
$a + 2b$.

Question 2-d
Remplacer le test : `if v.(k) mod 2 = 0` par : `if (bit p v.(k)) = 0`.

Question 3-b
```
let tri v =
  let n = vect_length(v) in
  let M = ref(v.(0)) in for i = 1 to n-1 do M := max !M v.(i)    done;
  let K = ref(0)     in while !M > 0     do K := !K+1; M := !M/2 done;
  for p = 0 to !K-1 do permute p v done
;;
```

Question 3-c
Si la suite $(v_0 \bmod 2^p, \ldots, v_{n-1} \bmod 2^p)$ est croissante, alors les sous-suites de $(v_0, \ldots, v_{n-1})$ déterminées par `permute p v` sont elles aussi croissantes « modulo 2^p » puisque `permute` conserve l'ordre relatif des éléments. On place en tête de v les éléments ayant un bit de rang p nul, c'est-à-dire ceux compris entre 0 et $2^p - 1$ modulo 2^{p+1}, puis les éléments compris entre 2^p et $2^{p+1} - 1$ modulo 2^{p+1} donc la suite $(v_0 \bmod 2^{p+1}, \ldots, v_{n-1} \bmod 2^{p+1})$ est bien croissante.

Question 3-d
La complexité est $O(n)$ ou plus précisément $O(n \ln M)$.

Interpolation de LAGRANGE et multiplication rapide

Question 1
On utilise l'algorithme de HÖRNER vu en cours :

> *Valeur*(P, z) *:* calcule la valeur du polynôme P au point z.
> $n \longleftarrow$ *longueur*(P)
> $r \longleftarrow 0$
> *pour* $i = n - 1 \ldots 0$ *faire :* $r \longleftarrow r * z + P.(i)$
> *retourner* r
>
> *fin*

Validité : à la sortie de la boucle on a l'invariant : $r = a_i + \ldots + a_{n-1} z^{n-i-1}$.
Complexité : n multiplications et n additions, soit $T(n) = 2n$.

Question 2
On itère le calcul de $P(x_i)$ pour chaque i :

Liste_valeurs(P, X*) :* calcule la liste des valeurs $P(x_i)$.
  $n \longleftarrow$ *longueur*(X)
  $Y \longleftarrow$ *vecteur de longueur* n
  pour $i = 0 \ldots n-1$ *faire :* $Y.(i) \longleftarrow$ *Valeur*$(P, X.(i))$
  retourner Y
fin

Complexité : n appels à *Valeur*, soit $T(n) = 2n^2$.

Question 3
Fonctions auxiliaires de calcul sur les polynômes :

Combinaison(P, Q, a, b*) :* calcule $aP + bQ$
  $p \longleftarrow$ *longueur*(P)
  $q \longleftarrow$ *longueur*(Q)
  $r \longleftarrow \max(p, q)$
  $R \longleftarrow [|0; \ldots; 0|]$ (r fois zéro)
  pour $i = 0 \ldots p-1$ *faire :* $R.(i) \longleftarrow a * P.(i)$
  pour $i = 0 \ldots q-1$ *faire :* $R.(i) \longleftarrow R.(i) + b * Q.(i)$
  retourner R
fin

Complexité : $T(p, q) = p + 2q$.

Produit(P, a, b*) :* calcule $P(z) * (az + b)$.
  $n \longleftarrow$ *longueur*(P)
  $R \longleftarrow$ *vecteur de longueur* $n+1$
  pour $i = 1 \ldots n-1$ *faire :* $R.(i) \longleftarrow b * P.(i) + a * P.(i-1)$
  $R.(0) \longleftarrow b * P.(0)$
  $R.(n) \longleftarrow a * P.(n-1)$
  retourner R
fin

Complexité : $T(n) = 3n - 1$.

Formule de LAGRANGE
On calcule pour chaque i le polynôme L_i et on cumule les polynômes $y_i L_i$ dans le résultat :

Lagrange(X, Y*) :*
calcule le polynôme P tel que $P\{X\} = Y$ par la formule de LAGRANGE
  $n \longleftarrow$ *longueur*(X)
  $P \longleftarrow [|0; \ldots; 0|]$ (n fois zéro)
  pour $i = 0 \ldots n-1$ *faire :*
    $L \longleftarrow [|Y.(i)|]$ (polynôme constant)
    pour $j = 0 \ldots n-1$ *si* $j \neq i$ *faire :*

```
                a ⟵ 1/(X.(i) − X.(j))
                b ⟵ −X.(i) ∗ a
                L ⟵ Produit(L, a, b)
            fin pour j
            P ⟵ Combinaison(P, L, 1, 1)
        fin pour i
        retourner P
    fin
```

Complexité : dans la boucle *pour* j il est fait 4 opérations pour calculer a et b et $3k-1$ opérations pour calculer *Produit*(L, a, b) où k est la longueur courante de L. Cette longueur prend les valeurs $1, 2, \dots, n$ donc le nombre total d'opérations pour calculer $y_i L_i$ est :

$$\sum_{k=1}^{n}(3k+3) = \tfrac{3}{2}n^2 + \tfrac{9}{2}n.$$

L'addition de P et $y_i L_i$ coûte $3n$ opérations (n suffiraient si l'on définit une fonction d'addition sans coefficients). La complexité totale est donc :

$$T(n) = \tfrac{3}{2}n^3 + \tfrac{15}{2}n^2 = O(n^3).$$

Formule de NEWTON
On calcule récursivement P et U :

```
    Newton_aux(X, Y, i) :
    calcule le polynôme P pour x_0, ..., x_i et le polynôme V(z) = (z−x_0)...(z−x_i).
        si i = 0 alors retourner ([|Y.(0)|], [| − X.(0); 1|])
        sinon :
            (Q, U) ⟵ Newton_aux(X, Y, i − 1)
            c ⟵ (Y.(i) − Valeur(Q, X.(i)))/Valeur(U, X.(i))
            P ⟵ Combinaison(Q, U, 1, c)
            V ⟵ Produit(U, 1, −X.(i))
            retourner (P, V)
        fin sinon
    fin

    Newton(X, Y) :
        n ⟵ longueur(X)
        (P, V) ⟵ Newton_aux(X, Y, n − 1)
        retourner P
    fin
```

Complexité : soit $T_{aux}(i)$ le temps nécessaire au calcul de *Newton_aux*(X, Y, i). On a la relation de récurrence :

$$\begin{cases} T_{aux}(0) = 1; \\ T_{aux}(i) = T_{aux}(i-1) + 10i + 9. \end{cases}$$

(en comptant $3i + 2$ opérations pour la combinaison de Q et U). On en déduit pour $n \geqslant 1$:

$$T(n) = T_{\mathrm{aux}}(n-1) = 1 + \sum_{i=0}^{n-1}(10i + 9) = 5n^2 + 4n + 1 = O(n^2).$$

Formule d'AITKEN

La formule donnée contient deux appels récursifs, ce qui conduit à calculer plusieurs fois les polynômes intermédiaires et à une complexité totale exponentielle. Il faut donc mémoriser les calculs effectués :

```
Aitken(X, Y) :
calcule tous les polynômes P_{a,b} et renvoie le dernier.
    n ⟵ longueur(X)
    table ⟵ tableau n × n de polynômes vides
    pour i = 0...n − 1 faire : table.(i).(i) ⟵ Y.(i)
    pour k = 1...n − 1 faire :
        pour i = 0...n − k − 1 faire :
            a ⟵ 1/(X.(i) − X.(i + k))
            b ⟵ −X.(i + k) * a
            c ⟵ −a
            d ⟵ X.(i) * a
            P ⟵ Produit(table.(i).(i + k − 1), a, b)
            Q ⟵ Produit(table.(i + 1).(i + k), c, d)
            table.(i).(i + k) ⟵ Combinaison(P, Q, 1, 1)
        fin pour i
    fin pour k
    retourner T.(0).(n − 1)
fin
```

Complexité : le calcul de *table*.$(i).(i+k)$ requiert $9k + 7$ opérations (car les polynômes *table*.$(i).(i+k-1)$ et *table*.$(i+1).(i+k)$ sont de longueur k). On obtient :

$$T(n) = \sum_{k=1}^{n-1}\sum_{i=0}^{n-k-1}(9k+7) = \sum_{k=1}^{n-1}(n-k)(9k+7) = \tfrac{3}{2}n^3 + \tfrac{7}{2}n^2 - 5n = O(n^3).$$

Conclusion de cette question : la formule de NEWTON est la plus performante des trois pour le calcul des coefficients du polynôme d'interpolation. La formule d'AITKEN est intéressante pour le calcul numérique de $P(x)$ (dans ce cas sa complexité devient $O(n^2)$ car les appels à *Produit* et *Combinaison* sont remplacés par un nombre constant d'opérations complexes). La formule de LAGRANGE peut aussi être utilisée pour un calcul numérique, mais elle présente l'inconvénient par rapport aux autres de ne pas être évolutive, c'est-à-dire que les calculs faits ne peuvent pas être réutilisés si l'on ajoute des points d'interpolation.

Question 4-a

Soit $x = x_k$: si k est pair alors $x^{n/2} = 1$ et $P_0(x) = y_k$ par définition de P_0. Si k est impair alors $x^{n/2} = -1$ et $P_1(xe^{-2i\pi/n}) = P_1(x_{k-1}) = y_k$ par définition de P_1. Donc le polynôme proposé interpole bien Y pour la liste X et son degré est au plus $n - 1$ car ceux de P_0 et P_1 sont majorés par $n/2 - 1$.

Question 4-b

Interpole(Y) :
calcule le polynôme d'interpolation de Y aux racines de l'unité.
    $n \longleftarrow$ *longueur*(Y)
    si $n = 1$ *retourner* $[|Y.(0)|]$
    sinon :
        diviser Y *en deux parties* Y_0, Y_1 *comme défini dans l'énoncé*
        $P_0 \longleftarrow$ *Interpole*(Y_0)
        $P_1 \longleftarrow$ *Interpole*(Y_1)

        calcul du polynôme $P_1(xe^{-2i\pi/n})$
        $z \longleftarrow 1$
        pour $k = 0 \ldots n/2 - 1$ *faire :*
            $P_1.(k) \longleftarrow P_1.(k) * z$
            $z \longleftarrow z * e^{-2i\pi/n}$
        fin pour k

        combine P_0 et P_1
        $R \longleftarrow$ *vecteur de longueur* n
        pour $k = 0 \ldots n/2 - 1$ *faire :*
            $R.(k) \longleftarrow (P_0.(k) + P_1.(k))/2$
            $R.(k + n/2) \longleftarrow (P_0.(k) - P_1.(k))/2$
        fin pour k
        retourner R
    fin sinon
fin

Question 4-c

Si l'on excepte les appels récursifs, *Interpole* effectue un nombre d'opérations complexes linéaire en n. On a donc l'équation $T(2^\alpha) = 2T(2^{\alpha-1}) + a2^\alpha + b$ que l'on résout en :

$$\frac{T(2^\alpha)}{2^\alpha} = \frac{T(2^{\alpha-1})}{2^{\alpha-1}} + a + b2^{-\alpha},$$
$$\frac{T(2^\alpha)}{2^\alpha} = \frac{T(1)}{1} + (\alpha - 1)a + b(\tfrac{1}{2} - 2^{-\alpha}),$$
$$T(2^\alpha) \sim a\alpha 2^\alpha = O(n \ln n) \text{ pour } n \to \infty.$$

Question 4-d

Le principe est le même : on divise P en deux polynômes « entrelacés » :

$$\begin{aligned} P(z) &= a_0 + a_1 z + \ldots + a_{n-1}z^{n-1} \\ &= (a_0 + a_2 z^2 + \ldots + a_{n-2}z^{n-2}) + z(a_1 + a_3 z^2 + \ldots + a_{n-1}z^{n-2}) \\ &= P_0(z^2) + zP_1(z^2) \end{aligned}$$

où P_0, P_1 sont deux polynômes de longueur $n/2$. Pour calculer la liste $(P(e^{2ik\pi/n}))$, il suffit de connaître les listes $(P_0(e^{4ik\pi/n}))$ et $(P_1(e^{4ik\pi/n}))$, c'est-à-dire les listes des valeurs de P_0 et P_1 aux racines $n/2$-èmes de l'unité. A partir de ces deux listes, on obtient une valeur $P(e^{2ik\pi/n})$ en un temps constant (deux opérations) donc le temps total de calcul suit une relation de récurrence similaire à celle établie pour l'algorithme d'interpolation.

Question 5
On calcule les listes des valeurs de P et Q pour les racines $2n$-èmes de l'unité en $O(n \ln n)$ opérations, on multiplie terme à terme ces deux listes en $O(n)$ opérations, ce qui donne la liste des valeurs du produit PQ pour les racines $2n$-èmes de l'unité. Cette liste permet de reconstituer PQ par interpolation en $O(n \ln n)$ opérations.

Le choix des racines de l'unité permet donc de multiplier deux polynômes en $O(n \ln n)$ opérations, au lieu de n^2 multiplications et $n(n-1)$ additions complexes pour la multiplication par distributivité et $\Theta(n^{\log_2(3)})$ pour la multiplication récursive de KARATSUBA. Cependant cet algorithme, appelé *multiplication par transformation de* FOURIER *rapide*, nécessite d'effectuer des calculs en nombres complexes, donc approchés. On obtient alors une *approximation* du produit. S'il s'agit de polynômes à coefficients entiers, sachant que le résultat est à coefficients entiers on peut arrondir le résultat approché : KNUTH démontre que l'incertitude sur les coefficients de PQ est de l'ordre de $(n \ln n)\varepsilon$ où ε est l'incertitude sur une opération. Donc, pour $n \ln n \ll 10^{18}$, il suffit de faire les calculs « avec une vingtaine de chiffres après la virgule » pour être sûr d'avoir un arrondi correct. Par ailleurs, le gain en vitesse n'est réel que pour de très grandes valeurs de n compte tenu des complications de l'algorithme et de la nécessité de garantir la précision des calculs intermédiaires. La multiplication par transformation de FOURIER rapide est utilisée pour les calculs sur des très grands nombres (plusieurs milliers de chiffres).

Plus longue sous-séquence commune

Question 1-a
On parcourt simultanément A et C en « retirant » un élément de C à chaque fois qu'il apparaît dans A.

```
let rec sous_séquence (a,c) =
  if c = [] then true
  else if a = [] then false
  else if (hd a) <> (hd c) then sous_séquence (tl a, c)
                           else sous_séquence (tl a, tl c)
;;
```

Validité : lorsque C ou A est vide alors `sous_séquence` renvoie immédiatement le résultat correct. Supposons C et A non vides : si C est une sous-séquence de A alors il existe une suite d'indices $0 \leqslant i_0 < i_1 < \ldots < i_{q-1} < n$ tels que $c_k = a_{i_k}$ pour tout k. Donc, si $c_0 \neq a_0$, alors $i_0 \geqslant 1$ et C est une sous-séquence de $(a_1, \ldots, a_{n-1})$. Lorsque $c_0 = a_0$ on a $(c_1, \ldots, c_{q-1}) = (a_{i_1}, \ldots, a_{i_{q-1}})$ qui est une sous-séquence de $(a_1, \ldots, a_{n-1})$ car $i_1 \geqslant 1$.

Réciproquement, si $c_0 = a_0$ et si $(c_1, \ldots, c_{q-1})$ est une sous-séquence de $(a_1, \ldots, a_{n-1})$ alors il existe des indices $1 \leqslant i_1 < \ldots < i_{q-1} < n$ tels que $c_k = a_{i_k}$ pour $k \geqslant 1$, et cette relation a encore lieu pour $k = 0$ en posant $i_0 = 0$, donc C est une sous-séquence de A. Enfin, si $c_0 \neq a_0$ et C est une sous-séquence de $(a_1, \ldots, a_{n-1})$ alors C est évidement une sous-séquence de A.

Ceci prouve que, si l'appel `sous_séquence a c` termine, alors il renvoie le bon résultat. Et il y a terminaison car la longueur de l'argument `a` diminue d'une unité à chaque appel récursif.

Question 1-b

On reprend l'algorithme précédent en accumulant les indices des éléments de A non appariés à ceux de C. La construction de la liste des indices n'est pas terminée lorsque $C = ()$ car il faut constituer la liste des indices des éléments restant dans A.

```
let rec indices_à_supprimer(a,c,i) =
  if a = [] then []
  else if (c = []) or ((hd a) <> (hd c))
    then i :: (indices_à_supprimer(tl a, c, i+1))
    else indices_à_supprimer(tl a, tl c, i+1)
;;

(* utilisation *)
indices_à_supprimer(a,c,0);;
```

Question 2-a

Si $a_{i-1} = b_{j-1}$: si $C = (c_0, \ldots, c_{q-1})$ est une PLSC à A_{i-1} et B_{j-1} alors la liste $C' = (c_0, \ldots, c_{q-1}, a_{i-1})$ est une sous-séquence commune à A_i et B_j. Ceci prouve que $\ell_{i,j} \geqslant 1 + \ell_{i-1,j-1}$. Si $C = (c_0, \ldots, c_{q-1})$ est une PLSC à A_i et B_j avec $q \geqslant 2$ alors $C' = (c_0, \ldots, c_{q-2})$ est une sous-séquence commune à A_{i-1} et B_{j-1}, quelle que soit la valeur de c_{q-1}. Ceci prouve que $\ell_{i-1,j-1} \geqslant \ell_{i,j} - 1$ dans le cas $\ell_{i,j} \geqslant 2$ et l'inégalité est également vraie si $\ell_{i,j} = 1$.

Si $a_{i-1} \neq b_{j-1}$: toute PLSC à A_{i-1} et B_j est une sous-séquence commune à A_i et B_j donc $\ell_{i,j} \geqslant \ell_{i-1,j}$. De même, $\ell_{i,j} \geqslant \ell_{i,j-1}$ et donc $\ell_{i,j} \geqslant \max(\ell_{i,j-1}, \ell_{i-1,j})$. Inversement, si $C = (c_0, \ldots, c_{q-1})$ est une PLSC à A_i et B_j, alors $c_{q-1} \neq a_{i-1}$ ou $c_{q-1} \neq b_{j-1}$ donc C est une sous-séquence commune à A_{i-1} et B_j ou à A_i et B_{j-1}. Dans le premier cas on obtient : $\ell_{i,j} \leqslant \ell_{i-1,j}$ et dans le second cas : $\ell_{i,j} \leqslant \ell_{i,j-1}$ d'où, dans tous les cas : $\ell_{i,j} \leqslant \max(\ell_{i,j-1}, \ell_{i-1,j})$.

Question 2-b

On choisit ici de représenter A et B par des vecteurs pour accéder en temps constant à a_{i-1} et b_{j-1} :

```
let matrice_plsc(a,b) =
  let n = vect_length(a) and p = vect_length(b) in
  let l = make_matrix (n+1) (p+1) 0 in
  for i = 1 to n do
    for j = 1 to p do
      (* invariant de boucle : ici, tous les coefficients     *)
      (* l.(u).(v) sont corrects si u < i ou (u = i et v < j) *)
      if a.(i-1) = b.(j-1)
        then l.(i).(j) <- 1 + l.(i-1).(j-1)
        else l.(i).(j) <- max  l.(i-1).(j)  l.(i).(j-1)
    done
  done;
  l
;;
```

Validité : à la suite de la création de `l` on a $\texttt{l}.(0).(j) = \texttt{l}.(i).(0) = 0$ pour tous i et j. Ceci montre que la propriété énoncée en commentaire est vérifiée lors de la première entrée dans la boucle interne ($i = j = 1$). L'invariance de cette propriété résulte de ce que l'on applique les formules du **2a** et de ce que les valeurs $\texttt{l}.(i-1).(j-1)$, $\texttt{l}.(i-1).(j)$ et $\texttt{l}.(i).(j-1)$ ont été correctement calculées auparavant.

Complexité : le temps de création de `l` est proportionnel à sa taille, soit $O(np)$. Le temps de calcul d'un coefficient $\texttt{l}.(i).(j)$ est borné (un accès à `a`, un accès à `b`, une comparaison, une addition ou un calcul de `max` et une affectation à un élément de tableau, plus quelques soustractions pour calculer $i - 1$ et $j - 1$) et chaque coefficient de `l` est calculé au plus une fois, donc le temps de remplissage de `l` est aussi $O(np)$.

Remarque : si A et B sont représentées par des listes chaînées et non des vecteurs, on peut encore produire la matrice `l` en $O(np)$ opérations car ces listes sont parcourues séquentiellement :

```
let matrice_plsc(a,b) =
  let n = list_length(a) and p = list_length(b) in
  let l = make_matrix (n+1) (p+1) 0 in
  let a' = ref(a) in
  for i = 1 to n do
    let b' = ref(b) in
    for j = 1 to p do
      (* invariant de boucle : ici, tous les coefficients     *)
      (* l.(u).(v) sont corrects si u < i ou (u = i et v < j) *)
      if hd(!a') = hd(!b')
        then l.(i).(j) <- 1 + l.(i-1).(j-1)
        else l.(i).(j) <- max  l.(i-1).(j)  l.(i).(j-1);
      b' := tl(!b')
    done;
    a' := tl(!a')
  done;
  l
;;
```

Question 2-c
$\ell = \mathtt{l}.(n).(p)$ est la longueur d'une PLSC à A et B. Si $a_{n-1} = b_{p-1}$ alors d'après **2a** il existe une PLSC à A et B obtenue en plaçant a_{n-1} au bout d'une PLSC à A_{n-1} et B_{p-1}. Si $a_{n-1} \neq b_{p-1}$, alors on cherche une PLSC à A_{n-1} et B_p ou à A_n et B_{p-1} suivant que $\mathtt{l}.(n-1).(p) > \mathtt{l}.(n).(p-1)$ ou non. On traite ici le cas d'une représentation de A et B par des vecteurs :

```
let plsc(a,b,l) =
  let n = vect_length(a) and p = vect_length(b) in
  let q = l.(n).(p) in
  let c = make_vect q a.(0) in
  let i = ref(n) and j = ref(p) and k = ref(q) in
  while !k > 0 do
    if a.(!i-1) = b.(!j-1)
      then begin
        c.(!k-1) <- a.(!i-1);
        i := !i - 1;
        j := !j - 1;
        k := !k - 1
      end
    else if l.(!i-1).(!j) > l.(!i).(!j-1)
      then i := !i-1
      else j := !j-1
  done;
  c
;;
```

Question 3-a
On crée deux vecteurs booléens, E_A et E_B, indexés par les éléments de E (convertis en nombres entiers au besoin) tels que :

- $E_A.(x) =$ `true` si est seulement si x apparaît dans A ;
- $E_B.(x) =$ `true` si est seulement si x apparaît dans B.

Ensuite on parcourt A en extrayant les éléments x tels que $E_B.(x)$ vaut `true` et on parcourt B en extrayant les éléments x tels que $E_A.(x)$ vaut `true`. Ceci donne les listes A' et B'.

```
(* On suppose les éléments de E sont numérotés de 0 à r-1 *)

let crée_tableau(a) =
  let e_a = make_vect r false in
  for i = 0 to vect_length(a) - 1 do
    e_a.(numéro(a.(i))) <- true
  done;
  e_a
;;
```

```
let extrait(a,e_b) =
  let a' = make_vect (vect_length a) a.(0) in
  let k = ref 0 in
  for i = 0 to vect_length(a) - 1 do
    if e_b.(numéro(a.(i))) then begin
      a'.(!k) <- a.(i);
      k := !k + 1
    end
  done;
  sub_vect a' !k
;;
```

Question 3-b

On passe en revue les listes A^t et B^t comme pour une fusion, en notant les indices des éléments communs à A et B. On constitue alors A' et B' à partir de ces notes.

```
let intersection(a,b,at,bt) =

  let n = vect_length(a) and p = vect_length(b) in
  let dans_b = make_vect n false
  and dans_a = make_vect p false in
  let i = ref 0 and j = ref 0 in

  while (!i < n) & (!j < p) do

    if a.(at.(!i)) = b.(bt.(!j))
      then begin
      (* élément commun *)
        let x = a.(at.(!i)) in
        (* note tous les éléments de a égaux à cet élément *)
        while (!i < n) & (a.(at.(!i)) = x) do
          dans_b.(at.(!i)) <- true;
          i := !i+1
        done;
        (* note tous les éléments de b égaux à cet élément *)
        while (!j < p) & (b.(bt.(!j)) = x) do
          dans_a.(bt.(!j)) <- true;
          j := !j+1
        done
      end

    else if a.(at.(!i)) < b.(bt.(!j))
      then i := !i + 1
      else j := !j + 1

  done;
  (dans_a, dans_b)
;;
```

```
let extrait(a,dans_b) =
  let a' = make_vect (vect_length a) a.(0) in
  let k = ref 0 in
  for i = 0 to vect_length(a) - 1 do
    if dans_b.(i) then begin
      a'.(!k) <- a.(i);
      k := !k + 1
    end
  done;
  sub_vect a' !k
;;
```

Question 4
Premier cas : soit r le cardinal de E. La création de `e_a` se fait en $O(r)$ opérations, le remplissage en $O(n)$ opérations et l'extraction de `a'` à partir de `a` et `e_b` en $O(n)$ opérations. La complexité du pré-traitement est donc $O(n + p + r)$.

Second cas : le calcul de `at` à partir de `a` se fait en $O(n \ln(n))$ opérations si l'on emploie un algorithme de tri par fusion (le quicksort a aussi cette complexité, mais en moyenne seulement) ; le calcul des vecteurs `dans_a` et `dans_b` a la complexité d'une fusion : $O(n+p)$ et l'extraction de `a'` à partir de `a` et `dans_b` se fait en $O(n)$ opérations. La complexité du pré-traitement est donc $O(n \ln(n) + p \ln(p))$.

En supposant $n = p$ et r petit devant n dans **3a**, on a donc une complexité de pré-traitement $O(n)$ ou $O(n \ln(n))$ ce qui est négligeable devant la complexité de calcul de la PLSC : $O(n^2)$. On peut donc toujours procéder au pré-traitement, ce n'est asymptotiquement pas pénalisant. Cependant, l'utilité de ce pré-traitement est discutable : dans le cas **3a**, si E est petit et A, B sont grandes, alors il est probable que presque tous les éléments de E figurent au moins une fois dans chacune des listes, et le pré-traitement ne diminue pas notablement les longueurs des listes à traiter. Dans le cas **3b**, si A et B sont aléatoires, indépendantes, et de longueur petite devant card(E) alors elles sont probablement « presque disjointes » et le pré-traitement peut être efficace. Par contre, si A et B ne sont pas indépendantes (en particulier s'il s'agit de versions successives d'un fichier), il est probable que le pré-traitement ne permettra de retirer que quelques éléments seulement.

Enfin, si n et p sont « petits », le temps de pré-traitement ne peut plus être considéré comme négligeable... En conclusion, je déconseille ce pré-traitement.

Complément

Le système UNIX possède une commande `diff` listant les différences entre deux fichiers texte suivant un algorithme inspiré de la recherche d'une PLSC : la distance entre deux fichiers A et B considérés comme des listes de lignes est le nombre d'éléments de A et de B ne faisant pas partie d'une PLSC à A et B. Elle est notée $d(A, B)$. On détermine cette distance en comparant A et B *par les deux bouts* : on progresse alternativement d'une unité de distance à partir des débuts de A et B, puis d'une unité à partir des fins de A et B, jusqu'à un point milieu, c'est-à-dire un couple (i, j) tel que $d(A_i, B_j) = \lfloor d(A, B)/2 \rfloor$. Le nombre de couples

(i,j) examinés au cours de cette recherche est majoré par $(n+p)d(A,B)$ *et il n'est pas nécessaire de construire une matrice de taille* np *pour cela*, il suffit de disposer de $O(d(A,B))$ positions mémoire.

On peut alors obtenir récursivement une PLSC à A et B en concaténant une PLSC à A_i et B_j avec une PLSC à $A \setminus A_i$ et $B \setminus B_j$. Le temps de calcul d'une PLSC suivant cet algorithme (découvert indépendamment par EW. MYERS (1986) et E. UKKONEN (1985)) est $O(nd(A,B))$ pour $n = p$. Pour plus de renseignements, consulter l'aide Emacs sur la commande `diff` et le code source (en C) de GNU-diff.

Arbres de priorité équilibrés

Question 1
Par récurrence sur $H(a)$.

Question 2

```
let rec insertion(a,e) = match a with
| B_vide         -> B_noeud(e,B_vide,B_vide)
| B_noeud(r,g,d) -> if e >= r then B_noeud(e, insertion(d,r), g)
                               else B_noeud(r, insertion(d,e), g)
;;
```

Correction : `insertion`(a,e) renvoie un APE contenant la seule étiquette e lorsque a est vide. Si `insertion`(a,e) renvoie un résultat correct pour tout arbre de hauteur strictement inférieure à h, alors pour un arbre $a = \mathtt{B\_noeud}(r,g,d)$ de hauteur h on insère la plus petite des étiquettes e ou r dans la branche droite, ce qui fournit un nouvel APE d' correct par hypothèse de récurrence, puis on renvoie $a' = \mathtt{B\_noeud}(\max(e,r), d', g)$. Ceci est un arbre de priorité car $\max(e,r)$ est la plus grande étiquette figurant dans a' et d', g sont des APE. De plus, $N(d') = N(d) + 1$ est compris entre $N(g)$ et $N(g)+1$, donc les deux branches ont même nombre de nœuds à une unité près, celle de gauche (d') étant la plus lourde en cas d'inégalité.

Complexité : $O(H(a))$.

Question 3

```
let rec transforme(a1) = match a1 with

| B_vide -> a1
| B_noeud(e,B_vide,B_vide) -> a1
| B_noeud(e,g,d) ->

  (* compare les racines de g et d si elles existent *)
  let g_plus_grand_que_d =
    match (g,d) with
    | (B_noeud(x,_,_), B_noeud(y,_,_)) -> x >= y
    | _ -> true (* une des branches est vide, c'est donc d *)
  in

  (* compare e au plus grand fils *)
  if g_plus_grand_que_d then match g with
     | B_vide -> a1
     | B_noeud(e',g',d') ->
        if e >= e' then a1
        else B_noeud(e', transforme(B_noeud(e,g',d')), d)
  else match d with
     | B_vide -> a1
     | B_noeud(e',g',d') ->
        if e >= e' then a1
        else B_noeud(e', g, transforme(B_noeud(e,g',d')))
;;
```

Correction : par récurrence sur la hauteur de a_1 comme à la question précédente.
Complexité : $O(H(a))$.

Question 4-a
Par récurrence sur $H(a)$.

Question 4-b

```
let rec retire_gauche(a) = match a with
| B_vide                   -> failwith "arbre vide"
| B_noeud(r,B_vide,B_vide) -> (B_vide,r)
| B_noeud(r,g,d)           -> let (g',e) = retire_gauche(g) in
                              (B_noeud(r,d,g'),e)
;;

let extraction(a) = let (a',e) = retire_gauche(a) in
match a' with
| B_vide         -> (a',e)
| B_noeud(r,g,d) -> (transforme(B_noeud(e,g,d)), r)
;;
```

Complexité : $O(H(a))$.

Question 5
Le temps de constitution de l'APE vérifie la relation de récurrence :

$$T_n = T_{n-1} + O(\ln(n)).$$

Donc $T_n = O(\ln 1 + \ln 2 + \ldots + \ln n) = O(n \ln n)$. Le temps d'extraction de tous les noeuds de l'APE vérifie la même relation, donc est aussi $O(n \ln n)$ et le temps de constitution de la liste finale, non compris les extractions dans l'APE, est $O(n)$ car on effectue des insertions en tête de liste. Le temps total du tri est donc $O(n \ln n)$.

Compilation d'une expression

Question 1
L'ordre des opérations est, compte tenu de la priorité de $*$ sur $+$ et des parenthèses : $-$, $-$, $*$, $+$. Les opérandes de chaque soustraction doivent être empilés juste avant l'opération de soustraction et la multiplication doit être effectuée immédiatement après les soustractions. Les libertés dans le choix du code compilé portent donc sur le fait que chaque opération peut être codée de manière directe ou inverse. Ceci fait au total 16 possibilités de compilation (64 si l'on considère que $+$ et $*$ sont commutatives). Le code suivant n'utilise que trois niveaux de pile :

```
empile_v b; empile_v c; binaire −; empile_v d; empile_v e;
binaire −; binaire ∗; empile_v a; binaire_inv +.
```

Ce code est optimal puisque la multiplication doit immédiatement suivre la deuxième soustraction et donc il faut au moins trois opérandes dans la pile avant d'effectuer la deuxième soustraction.

Question 2
Si $f = op_1(g)$ alors $N(f) = N(g)$.

Si $f = g\ op_2\ h$, il y a deux manières de compiler f :

- < calcul de g >; < calcul de h >; `binaire` op_2.
- < calcul de h >; < calcul de g >; `binaire_inv` op_2.

Le nombre de niveaux de pile utilisé est $\max(N(g), 1 + N(h))$ dans le premier cas et $\max(N(h), 1 + N(g))$ dans le second cas. On a donc :

$$\begin{aligned} N(f) &= \min(\max(N(g), 1 + N(h)), \max(N(h), 1 + N(g))) \\ &= \begin{cases} N(h) & \text{si } N(g) < N(h) \\ N(g) & \text{si } N(g) > N(h) \\ 1 + N(g) & \text{si } N(g) = N(h). \end{cases} \end{aligned}$$

Question 3-a
Algorithme :

- *une constante* c *est compilée en* `empile_c` c *;*
- *une variable* v *est compilée en* `empile_v` v *;*
- *pour compiler* $op_1(g)$*, compiler* g *et insérer* `unaire` op_1 *en queue du code obtenu ;*
- *pour compiler* $g\ op_2\ h$*, compiler séparément* g *et* h*, soient* $\overline{g}$ *et* $\overline{h}$ *les codes obtenus, puis comparer* $N(g)$ *et* $N(h)$*.*

  Si $N(g) < N(h)$ *alors retourner le code :* $\overline{h}$ *;* $\overline{g}$ *;* `binaire_inv` op_2*,*
  sinon retourner le code : $\overline{g}$ *;* $\overline{h}$ *;* `binaire` op_2*.*

Le calcul de $N(g)$ et $N(h)$ dans le dernier cas peut être effectué par parcours des formules g et h en appliquant les relations obtenues à la question précédente. Le programme ci-dessous évite le double parcours en retournant à la fois le code compilé et le nombre de niveaux de pile utilisés par ce code.

```
type 'a formule =
| Const of 'a
| Var   of string
| Op1   of string * ('a formule)
| Op2   of string * ('a formule) * ('a formule)
;;

type 'a instruction =
| Empile_c    of 'a
| Empile_v    of string
| Unaire      of string
| Binaire     of string
| Binaire_inv of string
;;

let rec compile(f) = (* retourne le code de f et N(f) *)
match f with
| Const(c)     -> ([Empile_c c], 1)
| Var(v)       -> ([Empile_v v], 1)
| Op1(op1,g)   -> let (code,n) = compile(g) in (code @ [Unaire op1], n)
| Op2(op2,g,h) ->
   let (codeg,ng) = compile(g)
   and (codeh,nh) = compile(h) in
   if      ng > nh then (codeg @ codeh @ [Binaire op2],     ng)
   else if ng < nh then (codeh @ codeg @ [Binaire_inv op2], nh)
   else                 (codeg @ codeh @ [Binaire op2],     1+ng)
;;
```

Question 3-b
On peut transformer la formule à compiler en un arbre d'instructions par un premier parcours en profondeur, chaque nœud étant remplacé par l'instruction correspondante, puis mettre ces instructions en liste par un deuxième parcours en profondeur en insérant les instructions en tête d'une liste initialement vide.

Question 4
Soit $k(n)$ le nombre minimal de nœuds d'une formule non calculable avec n niveaux de pile. Pour $n \geqslant 2$, une telle formule est de la forme $f = g \ op_2 \ h$ avec $N(g) \leqslant n$, $N(h) \leqslant n$ (sinon f n'est pas minimale) et donc $N(g) > n-1$, $N(h) > n-1$ (sinon $N(f) \leqslant n$). On en déduit $k(n) = 2k(n-1)+1$ et $k(1) = 3$ d'où $k(n) = 2^{n+1}-1$. En particulier, $k(8) = 511$.

Recherche d'une chaîne de caractères dans un texte

Question 1-a

```
let position(t,m) =
  let i = ref 0 and j = ref 0 in
  while (!j < vect_length(m)) & (!i + !j < vect_length(t)) do
    if m.(!j) = t.(!i + !j)
    then j := !j + 1
    else begin i := !i + 1; j := 0 end
  done;
  if !j = vect_length(m) then !i else failwith "non trouvé"
;;
```

Remarque : on peut remplacer le test : `!i + !j < vect_length(t)` par le test : `!i < vect_length(t) - vect_length(m)` cette dernière expression étant calculée en dehors de la boucle.

Complexité : i peut aller jusqu'à $|t|-|m|-1$ et pour chaque valeur de i, j peut aller de 0 à $|m|-1$ (cas où c'est la dernière comparaison qui échoue, par exemple $m = \text{aaaab}$ et $t = \text{aaa}\ldots\text{aaa}$) et il est fait un nombre constant d'opérations pour chaque couple (i,j) donc le nombre total d'opérations dans le pire des cas est $O((|t|-|m|)|m|) = O(|t||m|)$.

Question 1-b
Oui. Lorsqu'une comparaison échoue on a $m_j \neq t_{i+j}$ et la première lettre de m ne peut pas être retrouvée dans t avant la position $i+j$ puisque les positions précédentes contiennent les lettres $m_1, \ldots, m_{j-1}$. On peut donc remplacer l'instruction `i := !i + 1` par `i := !i + !j + 1` dans le programme précédent. De toutes façons, chaque lettre de t ne sera examinée qu'au plus deux fois (que l'algorithme soit modifié ou non), donc la complexité de la recherche devient $O(|t|)$.

Remarque : il n'est pas nécessaire que toutes les lettres de m soient distinctes pour obtenir cette situation, il suffit en fait que la première lettre de m soit distincte de toutes les autres.

Question 1-c

Pour $k, i \in \mathbf{N}$ on note $N(k,i)$ le nombre de mots m de longueur ℓ pour lesquels on effectue k comparaisons de lettres dans $(*)$ avant de conclure à la présence de m dans t ou de passer à la position suivante $i+1$. Alors :

$$\overline{\mu}(t,\ell) = \frac{1}{|A|^\ell} \sum_{i=0}^{|t|-1} \sum_{k=0}^{\ell} kN(k,i).$$

On a :

$$\begin{aligned}
&N(k,i) = 1 && \text{si } k = \ell,\\
&N(k,i) = (|A|-1)|A|^{\ell-k} && \text{si } k < \ell \text{ et } i+k < |t|,\\
&N(k,i) = |A|^{\ell-k} && \text{si } k < \ell \text{ et } i+k = |t|,\\
&N(k,i) = 0 && \text{sinon.}
\end{aligned}$$

D'où :

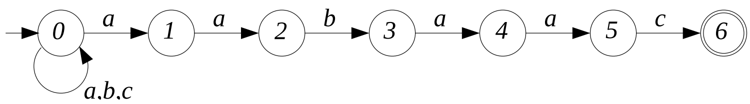
$$\begin{aligned}
\overline{\mu}(t,\ell) &\leqslant \sum_{i=0}^{|t|-1} \Big(\sum_{k=0}^{\ell-1} k(|A|-1)|A|^{-k} + \ell|A|^{-\ell} \Big)\\
&\leqslant |t|(|A|-1) \Big(\sum_{k=0}^{\infty} k|A|^{-k} \Big) = \frac{|A|}{|A|-1}|t|.
\end{aligned}$$

Question 1-d

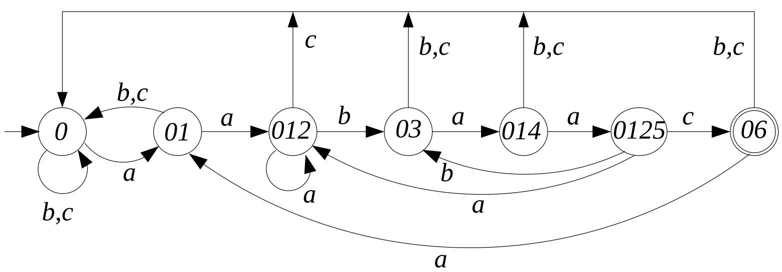
Par un calcul similaire on obtient $\tilde{\mu}(\ell, m) \leqslant \dfrac{|A|}{|A|-1}|m|$.

Question 2-a

automate non déterministe :



après déterminisation :



Question 2-b

On convient de représenter $\mathcal{A}_m$ par une matrice de transition a telle que $a.(i).(j)$ donne le numéro de l'état succédant à l'état i lorsqu'on lit la lettre j, et par un vecteur de booléens f tel que $f.(i) = \texttt{true}$ si et seulement si l'état i est un état final. L'état initial de $\mathcal{A}_m$ est par convention l'état 0 (i, j sont des entiers, l'automate est supposé déterministe complet).

```
let position a f m texte =
  let état = ref 0     (* numéro de l'état courant *)
  and i    = ref 0 in  (* indice dans le texte     *)
  while not(f.(!état)) & (!i < vect_length(texte)) do
    état := a.(!état).(texte.(!i));
    i := !i + 1
  done;
  if f.(!état) then !i - vect_length(m) - 1 else failwith "non trouvé"
;;
```

La complexité de `position` est $O(|t|)$ puisque les accès à la matrice `a` et au vecteur `f` se font en temps constant.

Question 2-c

On montre par récurrence sur $|t|$ que $\mathcal{A}_m$ passe de l'état ε à l'état $S(m,t)$ en lisant le texte t. Donc un mot t est reconnu par $\mathcal{A}_m$ si et seulement si $S(m,t) = m$, soit si et seulement si m est un suffixe de t, soit enfin si et seulement si $t \in A^*m$.

Question 2-d

Le premier cas est évident. Dans le deuxième cas, ua n'est pas un préfixe de m donc il faut retirer au moins une lettre à ua pour obtenir un préfixe de m, et par conséquent $S(m, ua) = S(m, u'a)$. Notons $v = S(m, u')$ et $w = S(m, u'a)$: il faut démontrer que $S(m, va) = w$. Pour cela on remarque que $S(m, va)$ est un suffixe de $u'a$ en même temps préfixe de m donc, par définition de w, $S(m, va)$ est un suffixe de w. Si $w = \varepsilon$ on obtient l'égalité cherchée et sinon, $w = w'a$ et w' est un suffixe de u' en même temps préfixe de m donc c'est un suffixe de v et $w = w'a$ est un suffixe de va en même temps préfixe de m, donc enfin w est un suffixe de $S(m, va)$, ce qui achève la démonstration.

Question 2-e

calcul de la première ligne
Pour chaque lettre $a \neq m.(0)$ *:* $M.(0).(a) \longleftarrow 0$.
Pour $a = m.(0)$ *:* $M.(0).(a) \longleftarrow 1$.

calcul de proche en proche
$v \longleftarrow 0$.
Pour $u = 1, \dots, |m| - 1$ *:*
  pour chaque lettre $a \neq m.(u)$ *:* $M.(u).(a) \longleftarrow M.(v).(a)$.
  pour $a = m.(u)$ *:* $M.(u).(a) \longleftarrow u + 1$*;* $v \longleftarrow M.(v).(a)$.
Fin pour.

dernière ligne
Pour chaque lettre a *:* $M.(|m|).(a) \longleftarrow M.(v).(a)$.

Complexité : $O(|m||A|) = O(|m|)$.

Exemple :

m	u	v	a	b	c
a	0		1	0	0
a	1	0	2	0	0
b	2	1	2	3	0
a	3	0	4	0	0
a	4	1	5	0	0
c	5	2	2	3	6
	6	0	1	0	0

Question 2-f
Comme tout automate déterministe reconnaissant $\mathcal{L}_m$ a au moins autant d'états que $\mathcal{L}_m$ a de résiduels, il suffit de prouver que $\mathcal{L}_m$ a au moins $|m|+1$ résiduels distincts. Soient u et uv deux préfixes de m ($m = uvw$) avec $v \neq \varepsilon$. Alors le langage résiduel de uv contient w ce qui n'est pas le cas du langage résiduel de u. Donc les langages résiduels des préfixes de m sont deux à deux distincts et il y en a $|m|+1$.

Question 3-b
Comme $k-p < k$, on a $p \geqslant 1$ donc le mot m est décalé vers la fin de t. Il faut seulement justifier que si $p > 1$ on ne risque pas de manquer des positions d'apparition de m dans t : si on décale m de q positions avec $q < p$ alors la lettre a_{k-q} est mise en correspondance avec b_{i+k} donc il y a non concordance.

La complexité dans le pire des cas est la même qu'au **1-a** car on peut seulement affirmer que $k \geqslant 0$ et $p \geqslant 1$ donc on peut avoir à comparer chaque lettre de m avec chaque lettre de t (sauf pour les dernières lettres de t).

Question 3-c
De façon à éviter de recalculer l'entier p à chaque comparaison négative, on précalcule la fonction de décalage dans une matrice `mat` indexée par les positions dans m et les lettres de A :

```
(* p est la taille de l'alphabet, retourne  *)
(* la matrice de décalage associée au mot m *)
let matrice p m =
  let n   = vect_length(m)    in
  let mat = make_matrix n p 1 in

    for i = 1 to n-1 do
      for a = 0 to p-1 do mat.(i).(a) <- mat.(i-1).(a) + 1 done;
      mat.(i).(m.(i-1)) <- 1
   done;

   mat
;;
```

La validité de `matrice` est évidente, sa complexité est $O(|m||A|) = O(|m|)$.

```
let position mat m texte =
  let n    = vect_length(m)         in
  let imax = vect_length(texte) - n in
  let i = ref 0 and j = ref(n-1)    in

  while (!i <= imax) & (!j >= 0) do
    if m.(!j) = texte.(!i + !j) then j := !j-1
    else begin i := !i + mat.(!j).(texte.(!i + !j)); j := n-1 end
  done;

  if !j < 0 then !i else failwith "non trouvé"
;;
```

Question 3-d

Oui. Au lieu de chercher la dernière occurrence de b_{i+k} dans $a_1 \dots a_{k-1}$, on peut chercher la dernière occurrence du mot $b_{i+k}a_{k+1} \dots a_{|m|-1}$ dans $a_1 \dots a_{k-1}$ et décaler m en conséquence. Ceci garantit que chaque lettre de t ne sera examinée qu'une fois dans le pire des cas, et $1/|m|$ fois en général. La construction de la table de décalage est cependant plus compliquée.

Solutions des travaux pratiques

Chemins dans $\mathbf{Z}^2$

1. Conversion mot → liste de points

```
(* avance (x,y) d'un pas dans la direction dir *)
let avance dir (x,y) = match dir with
  | N -> (x,     y + 1)
  | S -> (x,     y - 1)
  | E -> (x + 1, y    )
  | W -> (x - 1, y    )
;;

(* liste des points associés au mot "chemin" avec l'origine z *)
let rec points z chemin = match chemin with
  | [] -> [z]
  | dir::suite -> z :: points (avance dir z) suite
;;
```

2. Détection et élimination des boucles

```
(* détecte les points multiples (algorithme en O(n^2)) *)
let rec multiples points = match points with
  | []       -> false
  | p::suite -> (mem p suite) or (multiples suite)
;;

(* liste d'association (point, prédécesseur) *)
let rec prédécesseurs points = match points with
  | u::v::suite -> (v,u) :: prédécesseurs (v::suite)
  | _           -> []
;;

(* élimine les boucles *)
let sans_boucle points =
  let z0   = hd(points)           in
  let pred = prédécesseurs(points) in
  let l    = ref([hd(rev points)]) in
  while hd(!l) <> z0 do l := (assoc (hd !l) pred) :: !l done;
  !l
;;
```

3. Remplissage

```
(* ordonnées extrêmes *)
let rec extrêmes points = match points with
  | []      -> failwith "liste vide"
  | [(x,y)] -> (y,y)
  | (x,y)::suite -> let (y1,y2) = extrêmes(suite)
                    in (min y1 y, max y2 y)
;;

(* intersections avec une horizontale *)
let rec intersecte y0 points = match points with
  | (x1,y1)::(x2,y2)::suite ->
      if (min y1 y2 = y0) & (max y1 y2 > y0)
      then x1 :: intersecte y0 ((x2,y2)::suite)
      else intersecte y0 ((x2,y2)::suite)
  | _ -> []
;;

(* remplissage *)
let remplit points =
  let (ymin,ymax) = extrêmes(points) in
  for y = ymin to ymax-1 do
    let l = ref(sort__sort (prefix <=) (intersecte y points)) in
    while !l <> [] do match !l with
      | x1::x2::suite -> noircit (x1,y) (x2,y+1); l := suite
      | _ -> failwith "cas impossible"
    done
  done
;;
```

Files d'attente et suite de HAMMING

1. Implémentation des files d'attente

```
(* création, inspection *)
let nouv_file() = {avant = []; arrière = []};;
let est_vide f  = (f.avant = []) & (f.arrière = []);;
let longueur f  = list_length(f.avant) + list_length(f.arrière);;
```

```
(* insertion, suppression *)
let ajoute f x = f.arrière <- x :: f.arrière;;
let retire f = match f.avant with
 | x::suite -> f.avant <- suite; x
 | []       -> match rev(f.arrière) with
               | x::suite -> f.avant <- suite; f.arrière <- []; x
               | []       -> failwith "file vide"
;;
let premier f = let x = retire f in f.avant <- x :: f.avant; x;;
```

2. La suite de Hamming

a)

```
(* initialisation *)
let h2 = nouv_file();;  ajoute h2 2;;
let h3 = nouv_file();;  ajoute h3 3;;
let h5 = nouv_file();;  ajoute h5 5;;

(* extrait l'entier de Hamming suivant *)
let suivant() =
  let x2 = premier(h2)
  and x3 = premier(h3)
  and x5 = premier(h5) in
  let x  = min (min x2 x3) x5 in
  if x = x2 then (let _ = retire h2 in ());
  if x = x3 then (let _ = retire h3 in ());
  if x = x5 then (let _ = retire h5 in ());
  ajoute h2 (2*x);
  ajoute h3 (3*x);
  ajoute h5 (5*x);
  x
;;
```

b)

```
(* extraction optimisée *)
let suivant() =
  let x2 = premier(h2)
  and x3 = premier(h3)
  and x5 = premier(h5) in
  let x  = min (min x2 x3) x5 in
  if x = x2 then (let _ = retire h2 in ());
  if x = x3 then (let _ = retire h3 in ());
  if x = x5 then (let _ = retire h5 in ());
  ajoute h5 (5*x);
  if x mod 5 > 0 then begin
     ajoute h3 (3*x);
     if x mod 3 > 0 then ajoute h2 (2*x)
  end;
  x
;;
```

Validité. On montre par récurrence sur n la propriété suivante : *soit* H_n *le* n*-ème entier de* HAMMING *supérieur ou égal à* 2. *Alors à l'issue du* n*-ème appel à* `suivant`, *le nombre retourné est* H_n *et les files* h_2, h_3, h_5 *contiennent ensemble tous les entiers de* HAMMING *compris entre* H_{n+1} *et* $5H_n$ *de la forme* $2H_k$ *ou* $3H_k$ *ou* $5H_k$ *pour* $k \leqslant n$, *chaque entier étant présent dans une seule des files.*

Le cas $n = 1$ est immédiat. Supposons la propriété vraie au rang n et examinons la situation à l'issue du $(n+1)$-ème appel à `suivant` : le nombre x retourné est H_{n+1} car c'est le plus petit des éléments présents dans h_2, h_3, h_5 à l'issue de l'appel précédent. x étant retiré de toutes les files qui le contiennent (il y en a fait une seule), tous les entiers restant dans h_2, h_3, h_5 sont supérieurs strictement à H_{n+1}. On sait déjà que h_2, h_3 et h_5 contiennent ensemble tous les entiers de HAMMING supérieurs strictement à H_{n+1} de la forme $2H_k$ ou $3H_k$ ou $5H_k$ avec $k \leqslant n$, il reste à étudier le cas des multiples de H_{n+1}.

Étude de $5H_{n+1}$ **:** $5H_{n+1}$ est inséré dans h_5 et c'est un nouvel élément car avant cette insertion h_2, h_3 et h_5 ne contenaient que des nombres inférieurs ou égaux à $5H_n$.

Étude de $3H_{n+1}$ **:** si l'entier H_{n+1} est divisible par 5, $H_{n+1} = 5H_p$, alors $3H_{n+1} = 5(3H_p) = 5H_q$ et $q < n+1$ car $H_q = \frac{3}{5}H_{n+1} < H_{n+1}$ donc $3H_{n+1}$ est présent dans une et une seule des files h_2, h_3, h_5 avant l'extraction de H_{n+1} et il n'est pas réinséré lors de cette extraction.

Si H_{n+1} n'est pas divisible par 5 alors $3H_{n+1}$ n'appartient à aucune des files h_2, h_3, h_5 avant l'insertion effectuée lors de l'extraction de H_{n+1}. En effet, on ne peut avoir $3H_{n+1} = 5H_k$ puisque H_{n+1} n'est pas divisible par 5 et on n'a pas non plus $3H_{n+1} = 3H_k$ ou $3H_{n+1} = 2H_k$ avec $k \leqslant n$ car ces relations impliquent $H_k \geqslant H_{n+1} > H_n$. Donc $3H_{n+1}$ est bien présent dans une et une seule des files h_2, h_3, h_5 à l'issue du $(n+1)$-ème appel à `suivant`.

Étude de $2H_{n+1}$ **:** en distinguant les cas « H_{n+1} divisible par 5 », « H_{n+1} non divisible par 5 mais divisible par 3 » et « H_{n+1} divisible ni par 5 ni par 3 », on démontre de même que $2H_{n+1}$ est présent dans une et une seule des files h_2, h_3, h_5 à l'issue du $(n+1)$-ème appel à `suivant`. Ceci achève la démonstration de validité de `suivant`. ■

c)

```
(* initialisation *)
let h2 = nouv_file();;    ajoute h2 (1,0,0);;
let h3 = nouv_file();;    ajoute h3 (0,1,0);;
let h5 = nouv_file();;    ajoute h5 (0,0,1);;

(* compare deux triplets de Hamming *)
let ln2 = log(2.0) and ln3 = log(3.0) and ln5 = log(5.0);;
let minimum (a,b,c as x) (a',b',c' as y) =
  let t = float_of_int(a-a')*.ln2
       +. float_of_int(b-b')*.ln3
       +. float_of_int(c-c')*.ln5 in
  if t <= 0. then x else y
;;
```

```
(* extrait le triplet de Hamming suivant *)
let suivant() =
  let x2 = premier(h2)
  and x3 = premier(h3)
  and x5 = premier(h5) in
  let (a,b,c as x)  = minimum (minimum x2 x3) x5 in
  if x = x2 then (let _ = retire h2 in ());
  if x = x3 then (let _ = retire h3 in ());
  if x = x5 then (let _ = retire h5 in ());
  ajoute h5 (a,b,c+1);
  if c = 0 then begin
     ajoute h3 (a,b+1,c);
     if b = 0 then ajoute h2 (a+1,b,c)
  end;
  x
;;
```

d) Le programme suivant affiche la somme ℓ_n des longueurs des trois files h_2, h_3 et h_5 après extraction du n-ème entier de HAMMING :

```
(* taille des files pour n0, 2n0, 4n0, .., n1 *)
let étude n0 n1 =

  h2.avant <- [1,0,0]; h2.arrière <- [];
  h3.avant <- [1,0,0]; h3.arrière <- [];
  h5.avant <- [1,0,0]; h5.arrière <- [];

  let borne = ref n0 in
  for n = 2 to n1 do
    let _ = suivant() in
    if n = !borne then begin
      let l = longueur h2 + longueur h3 + longueur h5 in
      printf__printf "n = %6d   l = %6d\n" n l;
      borne := !borne * 2
    end
  done
;;
#étude 1000 100000;;
n =   1000   l =    256
n =   2000   l =    402
n =   4000   l =    638
n =   8000   l =   1007
n =  16000   l =   1596
n =  32000   l =   2526
n =  64000   l =   4002
- : unit = ()
```

On constate $\ell_{8n} \approx 4\ell_n$ ce qui conduit à conjecturer que $\ell_n \approx \lambda n^{2/3}$ pour une certaine constante λ. Cette conjecture est exacte, on trouvera une démonstration de ce fait et une étude de divers algorithmes permettant de calculer les n premiers entiers de Hamming, ou seulement le n-ème, dans les deux documents suivants :

```
http://pauillac.inria.fr/~quercia/papers/hamming.ps.gz
http://pauillac.inria.fr/~quercia/papers/hamming-new.ps.gz
```

Recherche de contradictions par la méthode des consensus

3. Programmation de l'étape 1

```
(* compare deux clauses *)
let implique (p1,n1) (p2,n2) =
  (subtract p1 p2 = []) & (subtract n1 n2 = []);;

(* teste si une clause est impliquée par un système *)
let rec present c syst = match syst with
  | []       -> false
  | x::suite -> (implique x c) or (present c suite);;

(* ajoute une clause au système en éliminant *)
(* les clauses qui en découlent              *)
let rec ajoute c syst = match syst with
  | []       -> [c]
  | a::suite -> if implique c a
                then if implique a c then syst else ajoute c suite
                else if implique a c then syst else a :: (ajoute c suite)
;;

(* simplifie un système *)
let simplifie s =
  let rec simpl s1 s2 = match s2 with
    | []       -> s1
    | c::suite -> simpl (ajoute c s1) suite
  in simpl [] s
;;
```

4. Programmation des étapes 2 et 3

```
(* ajoute au système le consensus éventuel de deux clauses *)
let ajoute_cons (p1,n1) (p2,n2) syst =
  match (intersect p1 n2, intersect p2 n1) with
    | ([x],[]) ->
         ajoute (union (except x p1) p2, union n1 (except x n2)) syst
    | ([],[x]) ->
         ajoute (union p1 (except x p2), union (except x n1) n2) syst
    | _ -> syst
;;

(* ajoute à l tous les consensus entre c et une clause du système *)
let rec ajoute_consensus c syst l = match syst with
  | [] -> l
  | c1::suite -> ajoute_consensus c suite (ajoute_cons c c1 l)
;;

(* clôt un système par consensus *)
let rec cloture s =
  let rec consensus s1 s2 = match s1 with
    | [] -> s2
    | c::suite -> consensus suite (ajoute_consensus c s s2)
  in
  let t = simplifie(consensus s s) in
  if (subtract s t = []) & (subtract t s = []) then s else cloture t
;;
```

6. Affichage des étapes d'une contradiction

```
(* "ajoute", "simplifie", "ajoute_consensus", "cloture" sont inchangées *)

let prefix =>   a b = ([b],  [a],   H)   (* a => b         *)
and prefix =>~  a b = ([],   [a;b], H)   (* a => non b     *)
and prefix ~=>  a b = ([a;b],[],    H)   (* non a => b     *)
and prefix ~=>~ a b = ([a],  [b],   H)   (* non a => non b *)
;;

let string_of_clause(p,n,_) =
  (if n = [] then "" else (concat "." n)^" => ") ^
  (if p = [] then "Faux" else concat "+" p)
;;

let print_clause(c) = format__print_string(string_of_clause c);;
install_printer "print_clause";;

(* compare deux clauses *)
let implique (p1,n1,_) (p2,n2,_) =
   (subtract p1 p2 = []) & (subtract n1 n2 = []);;
```

```
(* ajoute au système le consensus éventuel de deux clauses *)
let ajoute_cons (p1,n1,_ as c1) (p2,n2,_ as c2) syst =
  match (intersect p1 n2, intersect p2 n1) with
  | ([x],[]) ->
     ajoute (union (except x p1) p2, union n1 (except x n2), D(c1,c2)) syst
  | ([],[x]) ->
     ajoute (union p1 (except x p2), union (except x n1) n2, D(c1,c2)) syst
  | _ -> syst
;;

(* explique les étapes d'une déduction *)
let explique clause =
  let vus = ref [] in
  let rec expl (p,n,r as c) =
  if not(mem c !vus) then match r with
  | H -> print_string "J'ai l'hypothèse : ";
         print_string (string_of_clause c);
         print_newline();
         vus := c :: !vus
  | D((p1,n1,r1 as c1),(p2,n2,r2 as c2)) ->
         if r1 = H then (expl c2; expl c1) else (expl c1; expl c2);
         print_string "De ";  print_string(string_of_clause c1);
         print_string " et "; print_string(string_of_clause c2);
         print_string " je déduis : "; print_string(string_of_clause c);
         print_newline();
         vus := c :: !vus
  in expl clause
;;
```

Modélisation d'un tableur

1. Tri topologique

```
let dépendances(t) =
  for i = 0 to t.n-1 do for j = 0 to t.p-1 do
    t.suc.(i).(j) <- []
  done done;
  for i = 0 to t.n-1 do for j = 0 to t.p-1 do match t.f.(i).(j) with
  | Rien         -> t.dep.(i).(j) <- 0
  | Somme(liste) ->
    t.dep.(i).(j) <- list_length(liste);
    do_list (fun (k,l) -> t.suc.(k).(l) <- (i,j)::t.suc.(k).(l)) liste
  done done
;;
```

```
let rec place t (i,j) = t.lt <- (i,j)::t.lt; décompte t t.suc.(i).(j)
and décompte t liste = match liste with
  | []          -> ()
  | (i,j)::suite -> t.dep.(i).(j) <- t.dep.(i).(j) - 1;
                    if t.dep.(i).(j) = 0 then place t (i,j);
                    décompte t suite
;;

let tri_topo(t) =
  dépendances(t);
  t.lt <- [];
  for i = 0 to t.n-1 do for j = 0 to t.p-1 do
    if t.f.(i).(j) = Rien then place t (i,j)
  done done;
  t.lt <- rev(t.lt);
  if list_length(t.lt) < t.n*t.p then failwith "tri impossible"
;;
```

2. Calcul du tableau

```
let calc t (i,j) = match t.f.(i).(j) with
| Rien -> ()
| Somme(liste) ->
  t.v.(i).(j) <- 0;
  do_list (fun (k,l) -> t.v.(i).(j) <- t.v.(i).(j) + t.v.(k).(l)) liste
;;
let calcule t = do_list (calc t) t.lt;;
```

3. Complément

On ajoute dans la structure `tableau` une nouvelle matrice contenant pour chaque cellule la liste des cellules qui en dépendent, liste triée par ordre topologique. Cette liste s'obtient par fusion des listes des cellules dépendant des successeurs de la cellule considérée. On est ainsi ramené au problème du calcul d'un tableau en ordre topologique inverse, il suffit de conduire les calculs dans l'ordre donné par `rev(t.lt)`.

Analyse syntaxique

2. Analyse lexicale

```
(* découpe une chaine en lexèmes *)
let lexèmes s =

  let i   = ref 0           (* indice pour s             *)
  and res = ref []          (* liste des lexèmes trouvés *)
  and l   = string_length s in
```

```
  while !i < l do match s.[!i] with
    (* chiffre -> Nombre *)
    | `0`..`9` ->
          let x = ref 0 in
          while (!i < l) & (s.[!i] >= `0`) & (s.[!i] <= `9`) do
            x := !x*10 + int_of_char s.[!i] - int_of_char `0`;
            i := !i + 1
          done;
          res := Nombre(!x) :: !res

    (* caractère alphabétique -> Ident *)
    | `a`..`z` ->
          let i0 = !i in
          while (!i < l) & (s.[!i] >= `a`) & (s.[!i] <= `z`) do
            i := !i+1
          done;
          res := (match sub_string s i0 (!i - i0) with
                    | "let"  -> Let
                    | "in"   -> In
                    | "fun"  -> Fun
                    | "if"   -> If
                    | "then" -> Then
                    | "else" -> Else
                    | x      -> Ident(x)) :: !res

    (* opérateurs, parenthèses et symboles *)
    | `%` -> res := Opr(1,`%`) :: !res; i := !i+1
    | `+` -> res := Opr(2,`+`) :: !res; i := !i+1
    | `*` -> res := Opr(3,`*`) :: !res; i := !i+1
    | `/` -> res := Opr(3,`/`) :: !res; i := !i+1
    | `(` -> res := ParO :: !res;       i := !i+1
    | `)` -> res := ParF :: !res;       i := !i+1
    | `=` -> res := Egal :: !res;       i := !i+1

    (* reconnaît - et -> *)
    | `-` -> if (!i+1 < l) & (s.[!i+1] = `>`)
             then begin res := Flèche :: !res;     i := !i+2 end
             else begin res := Opr(2,`-`) :: !res; i := !i+1 end

    (* ignore les espaces *)
    | ` ` | `\t` | `\n` -> i := !i+1

    (* refuse les autres caractères *)
    | _ -> failwith ("caractère illégal : " ^ (make_string 1 s.[!i]))

  done;
  rev !res
;;
```

3. Analyse syntaxique

```
(* empile un lexème ou une expression et *)
(* simplifie la pile autant que possible *)
let rec empile e pile =

 (* priorité de e *)
  let p = match e with
    | L(In) | L(Then) | L(Else) | L(ParF) -> 0
    | L(Opr(p,_)) -> p
    | _           -> 5 (* > priorité maximum *)
  in

  (* conversion des nombres et identificateurs *)
  let e = match e with
    | L(Nombre x) -> E(Const x)
    | L(Ident  x) -> E(Var   x)
    | _ -> e
  in

  match e, pile with
  (* application de fonction *)
  |  (E x), (E f)::suite -> empile (E(Appl(f,x))) suite

  (* parenthèses *)
  | (L ParF), (E a)::(L ParO)::suite -> empile (E a) suite

  (* opération prioritaire en instance *)
  | _, (E b)::(L(Opr(q,o)))::(E a)::suite
   when p <= q -> empile e (empile (E(Bin(a,o,b))) suite)

  (* définition de fonction *)
  | _, (E a)::(L Flèche)::(E(Var x))::(L Fun)::suite
   when p = 0 -> empile e (empile (E(Fct(x,a))) suite)

  (* liaison locale *)
  | _, (E b)::(L In)::(E a)::(L Egal)::(E(Var x))::(L Let)::suite
   when p = 0 -> empile e (empile (E(Letin(x,a,b))) suite)

  (* test *)
  | _, (E c)::(L Else)::(E b)::(L Then)::(E a)::(L If)::suite
   when p = 0 -> empile e (empile (E(Test(a,b,c))) suite)

  (* pas de simplification possible *)
  | _, _ -> e::pile
;;
```

4. Évaluation

```
(* évaluation *)
let rec valeur env expr = match expr with

  | Const c  -> Int c
  | Var   x  -> assoc x env
  | Fct(x,e) -> Cloture(env,expr)

  | Bin(e1,op,e2) ->
      let v1 = match valeur env e1 with
        | Int x -> x | _ -> failwith "valeur non entière" in
      let v2 = match valeur env e2 with
        | Int x -> x | _ -> failwith "valeur non entière" in
      let v = match op with
        | `+` -> v1 + v2
        | `-` -> v1 - v2
        | `*` -> v1 * v2
        | `/` -> v1 / v2
        | `%` -> v1 mod v2
        | _   -> failwith "opérateur inconnu"
      in Int v

  | Appl(f,e) ->
      let env1,x,code = match valeur env f with
        | Cloture(env1,Fct(x,code)) -> env1,x,code
        | _ -> failwith "valeur non fontionnelle"
      in valeur ((x,valeur env e) :: env1) code

  | Test(e1,e2,e3) ->
      let v = match valeur env e1 with
        | Int x -> x | _ -> failwith "valeur non entière"
      in valeur env (if v = 0 then e2 else e3)

  | Letin(x,e1,e2) ->

      (* attention : si e1 s'évalue en une fonction, il faut    *)
      (* introduire une liaison pour x dans la cloture de cette *)
      (* fonction pour le cas où la fonction serait récursive   *)

      let v = match valeur env e1 with
        | Int(c)             -> Int(c)
        | Cloture(env2,code) -> let rec y = Cloture((x,y)::env2,code) in y
      in valeur ((x, v) :: env) e2
;;
```

Annexes

Bibliographie

[AHO] AHO, ULLMAN
Concepts fondamentaux de l'informatique
Dunod, 1993

[ALBERT] Ouvrage collectif, coordination L. ALBERT
Cours et exercices d'informatique
Vuibert, 1998

[CORM] CORMEN, LEISERSON, RIVEST
Introduction à l'algorithmique
Dunod, 1993

[FROID] FROIDEVAUX, GAUDEL, SORIA
Types de données et algorithmes
Ediscience, 1994

[KNUTH] KNUTH
The art of computer programming
Addison-Wesley, 1973

[LEROY] LEROY, WEIS
Le langage Caml
InterÉditions, 1993

[MEHL] MEHLHORN
Data Structures and Algorithms
Springer-Verlag, 1984

[SEDG] SEDGEWICK
Algorithmes en langage C
Interéditions, 1991

[STERN] STERN
Fondements mathématiques de l'informatique
Ediscience, 1994

Aide mémoire de CAML

Déclarations et instructions

commentaires	`(* ... *)`
définition d'une valeur	`let` *v* `=` *expression*
récursive	`let rec` *v* `= ...`
locale	`let` *v* `= ... in` *expression*
	expression `where` *v* `= ...`
définitions parallèles	`let` *v* `= ... and` *w* `= ...`
successives	`let` *v* `= ... in let` *w* `= ...`
variable modifiable	`let` *v* `= ref(`*expression*`)`
valeur d'une référence	`!`*v*
modification d'une référence	*v* `:= ...`
fonction sans argument	`let` *f*`() = ...`
fonction à un argument	`let` *f x* `= ...`
fonction à n arguments	`let` *f x1* `..` *xn* `= ...`
expression conditionnelle	`if ... then` *expr-vrai* `else` *expr-faux*
choix multiple	`match` *valeur* `with`
	`|` *motif-1* `->` *expression-1*
	`|` *motif-2* `->` *expression-2*
	`...`
	`|` *motif-n* `->` *expression-n*
	`| _ ->` *expression-par-défaut*
ne rien faire	`()`
calculs en séquence	`begin ... end`
boucle croissante	`for` *i* `=` *début* `to` *fin* `do ... done`
boucle décroissante	`for` *i* `=` *début* `downto` *fin* `do ... done`
boucle conditionnelle	`while` *condition* `do ... done`
déclencher une erreur	`failwith "`*message*`"`

Expressions booléennes

vrai, faux	`true false`
et, ou, non	`& or not`
comparaison	`< <= = <> >= >`
booléen $\longmapsto$ chaîne	`string_of_bool`
chaîne $\longmapsto$ booléen	`bool_of_string`

Expressions entières

opérations arithmétiques	`+ - * /`
modulo	`mod`

valeur absolue	`abs`
entier précédent, suivant	`pred succ`
minimum et maximum	`min a b,  max a b`
opérations bit à bit	`land lor lxor lnot`
décalage de bits	`lsl lsr asr`
entier $\longmapsto$ chaîne	`string_of_int`
chaîne $\longmapsto$ entier	`int_of_string`
entier aléatoire entre 0 et $n-1$	`random__int(n)`

Expressions réelles

opérations arithmétiques	`+.  -.  *.  /.`
puissance	`**` ou `**.`
minimum et maximum	`min a b,  max a b`
fonctions mathématiques	`abs_float exp log sqrt sin cos tan` `sinh cosh tanh asin acos atan atan2`
réel $\longmapsto$ chaîne	`string_of_float`
réel $\longmapsto$ entier	`int_of_float`
chaîne $\longmapsto$ réel	`float_of_string`
entier $\longmapsto$ réel	`float_of_int`
réel aléatoire entre 0 et a	`random__float(a)`

Expressions rationnelles

utiliser les rationnels	`#open "num"`
opérations arithmétiques	`+/  -/  */  //  **/` `minus_num quo_num mod_num square_num`
comparaison	`</  <=/  =/  <>/  >=/  >/`
minimum et maximum	`min_num a b,  max_num a b`
valeur absolue	`abs_num`
numérateur, dénominateur	`numerator_num denominator_num`
simplifier une fraction	`normalize_num`
simplification automatique	`arith_status__set_normalize_ratio true`
partie entière	`floor_num round_num ceiling_num`
rationnel $\longmapsto$ chaîne	`string_of_num`
rationnel $\longmapsto$ entier	`int_of_num`
rationnel $\longmapsto$ réel	`float_of_num`
chaîne $\longmapsto$ rationnel	`num_of_string`
entier $\longmapsto$ rationnel	`num_of_int`
réel $\longmapsto$ rationnel	`num_of_float`

Listes

liste	`[x; y; z; ...]`
liste vide	`[ ]`
tête et queue	`hd tl,  x :: suite`
longueur d'une liste	`list_length`
concaténation	`@`

image miroir .... `rev`
appliquer une fonction .... `map fonction liste`
itérer un traitement .... `do_list traitement liste`
test d'appartenance .... `mem élément liste`
test de présence .... `exists prédicat liste`
`for_all prédicat liste`
recherche d'un élément .... `index élément liste`
opérations ensemblistes .... `union intersect subtract`
tri .... `sort__sort ordre liste`
association .... `assoc b [(a,x); (b,y); (c,z); ...] = y`
itérer une opération .... `it_list op a [x; y; z]`
`= op (op (op a x) y) z`
`list_it op [x; y; z] a`
`= op x (op y (op z a))`

Vecteurs
vecteur .... `[|x; y; z; ...|]`
vecteur vide .... `[| |]`
i-ème élement .... `v.(i)`
modification .... `v.(i) <- qqch`
longueur d'un vecteur .... `vect_length`
création .... `make_vect longueur valeur`
création d'une matrice .... `make_matrix n p valeur`
extraction .... `sub_vect vecteur début longueur`
concaténation .... `concat_vect`
copie .... `copy_vect`
appliquer une fonction .... `map_vect fonction vecteur`
itérer un traitement .... `do_vect traitement vecteur`
vecteur $\longmapsto$ liste .... `list_of_vect`
liste $\longmapsto$ vecteur .... `vect_of_list`

Chaînes de caractères
caractère .... `‘x‘`
chaîne de caractères .... `"xyz..."`
i-ème caractère .... `chaîne.[i]`
modification .... `chaîne.[i] <- qqch`
longueur d'une chaîne .... `string_length`
création .... `make_string longueur caractère`
caractère $\longmapsto$ chaîne .... `make_string 1 caractère`
extraction .... `sub_string chaîne début longueur`
concaténation .... `ch1 ^ ch2`, `concat [ch1; ch2; ch3; ...]`

Graphisme
utiliser le graphisme .... `#open "graphics"`
initialiser la fenêtre graphique .... `open_graph ""`
refermer la fenêtre .... `close_graph()`

effacer la fenêtre	`clear_graph()`
position du crayon	`current_point()`
changer la couleur du crayon	`set_color` *couleur*
couleurs	`black white red green blue yellow cyan magenta, rgb` *r g b*
changer l'épaisseur du crayon	`set_line_width` *épaisseur*
tracer un point	`plot` *x y*
déplacer, crayon levé	`moveto` *x y*
crayon baissé	`lineto` *x y*
tracer un cercle	`draw_circle` *x y rayon*
écrire un texte	`draw_string "`*texte*`"`
peindre un rectangle	`fill_rect` *x y largeur hauteur*
un polygone	`fill_poly [| (`*x0,y0*`); (`*x1,y1*`); ... |]`
un disque	`fill_circle` *x y rayon*
attendre un évènement	`read_key()` `wait_next_event [`*ev1*`; `*ev2*`; ...]`

Entrées-sorties au terminal

impression de valeurs	`print_int print_float num__print_num print_char print_string print_endline`
changer de ligne	`print_newline()`
impression formattée	`printf__printf` *format valeurs*
lecture de valeurs	`read_int read_float read_line`

Entrées-sorties dans un fichier

ouverture en lecture	`let` *canal* `= open_in "`*nom*`"`
en écriture	`let` *canal* `= open_out "`*nom*`"`
lecture	`input_char input_line input_byte input_value`
écriture	`output_char output_string output_byte output_value flush`
fermeture	`close_in close_out`

Commande de l'interpréteur

tracer une fonction	`trace "`*fonction*`"`
ne plus tracer	`untrace "`*fonction*`"`
utiliser une fonction d'impression	`install_printer "`*fonction*`"`
ne plus l'utiliser	`remove_printer "`*fonction*`"`
charger un fichier source	`load "`*nom*`", include "`*nom*`"`
charger un module compilé	`load_object "`*nom*`"`
nom complètement qualifié	*module__nom*
utiliser les noms courts d'un module	`#open "`*module*`"`
ne plus les utiliser	`#close "`*module*`"`
ajouter un répertoire de recherche	`directory "`*chemin*`"`
quitter l'interpréteur	`quit()`

Index