La lettre de Caml numéro 6

Laurent Chéno
54, rue Saint-Maur
75011 Paris
Tél. 01 48 05 16 04
Fax 01 48 07 80 18
email : laurent.cheno@hol.fr

été 1997

Édito

Dans cette nouvelle lettre, nous envisageons de différents points de vue les décompositions en sommes d'entiers d'un entier naturel n, et obtenons de nombreux résultats combinatoires intéressants.
Ensuite, nous nous intéressons — sans prétentions — aux bases du λ-calcul, en insistant sur l'intérêt pratique de la notation de de Bruijn des λ-termes. Si vous témoignez de votre intérêt pour la matière, nous pourrons prolonger notre étude dans des Lettres ultérieures.

Table des matières

Liste des programmes

Partitions d'un entier

Un faux départ : décompositions d'un entier en sommes d'entiers

Définition et dénombrement

Un entier naturel n peut être décomposé de différentes façons en somme d'entiers naturels non nuls. Par exemple, $4 = 3+1 = 1+3 = 1+1+1+1 = 1+1+2 = 2+1+1 = 1+2+1 = 2+2$, ce qui fait 8 décompositions, si l'on tient compte de l'ordre des termes.

Théorème 1 *L'entier $n \geqslant 1$ peut être décomposé de 2^{n-1} façons différentes en somme d'entiers naturels non nuls, l'ordre dans lequel on somme les termes étant pris en compte.*

✧ Donnons une preuve combinatoire de ce résultat.
Pour un entier $n \geqslant 1$ fixé, nous considérons n boules blanches alignées de gauche à droite. Pour représenter une décomposition $n = x_1 + \cdots + x_k$, nous colorions en noir les boules d'indices x_1, $x_1 + x_2$, ... Par exemple, à la décomposition $5 = 1+2+2$ nous associons le schéma $\bullet \circ \bullet \circ \bullet$, et à la décomposition $5 = 2+3$ le schéma $\circ \bullet \circ \circ \bullet$.
Notons que la dernière boule est toujours coloriée en noir.
De cette façon, nous avons une correspondance biunivoque entre l'ensemble des décompositions de l'entier n et l'ensemble des parties (les boules en noir) de l'ensemble des $n-1$ premières boules, d'où le résultat espéré. ✥

Théorème 2 *Notons d_n le nombre de décompositions de l'entier n, et convenons que $d_0 = 1$. Alors la série génératrice des d_n s'écrit*

$$D(z) = \sum_{n=0}^{+\infty} d_n z^n = \frac{1-z}{1-2z}.$$

✧ Il suffirait bien entendu d'utiliser le théorème précédent. Mais nous donnons ici une preuve directe, qui en fait correspond à l'idée du programme CAML de génération des décompositions qu'on trouvera plus bas.
En effet, toute décomposition d'un entier $n \geqslant 1$ a pour premier terme un entier k de l'intervalle $[1, \ldots, n]$, et il reste ensuite à dénombrer les décompositions de la différence $n-k$. C'est dire que

$$\forall n \geqslant 1, d_n = \sum_{k=1}^{n} d_{n-k} = \sum_{k=0}^{n-1} d_k.$$

On reconnaît le produit de Cauchy : $(1-z)^{-1} \times D(z)$ et plus précisément (à cause du décalage d'indice), on peut écrire :

$$1 + \frac{z}{1-z} D(z) = D(z) \text{ d'où : } D(z) = \frac{1-z}{1-2z}.$$

✥

Programme 1 Décomposition d'un entier (version avec référence)

```
let rec décompose = function
    | 0 -> []
    | 1 -> [[1]]
    | n ->  let res = ref []
            in
            for k = 1 to n - 1 do
                do_list (function l -> res := (k :: l) :: !res)
                        (décompose (n - k))
            done ;
            [n] :: !res ;;
```

Programme 2 Décomposition d'un entier (version avec `append`)

```
let rec décompose n =
    let rec décompose_rec n accu = function
        | 0 ->  accu
        | k ->  let ll = décompose (n - k)
                in
                décompose_rec n ((map (function l -> k :: l) ll) @ accu) (k - 1)
    in
    match n with
        | 0 -> []
        | 1 -> [ [1] ]
        | n -> décompose_rec n [ [n] ] (n - 1) ;;
```

Programmation en Caml

Une première façon consiste naturellement à écrire le programme 1.
Mais on peut regretter l'écriture non fonctionnelle : on utilise une référence.
On transforme la boucle en appel récursif et on obtient la version totalement fonctionnelle du programme 2.
On peut cette fois regretter l'appel à l'opérateur de concaténation de listes, qui augmente le coût de façon rédhibitoire. On obtient finalement le programme 3 page ci-contre.

Partitions d'un entier

Définition

En fait, les partitions au sens qui nous intéressera ici ne répondent pas exactement à la définition précédente, puisqu'on conviendra de ne pas tenir compte de l'ordre des termes de la somme. On décide une fois pour toutes d'écrire les termes de la somme en ordre décroissant.
Il n'y a ainsi plus par exemple que 5 décompositions de l'entier 4 :

$$4 = 3 + 1 = 2 + 2 = 2 + 1 + 1 = 1 + 1 + 1 + 1.$$

Il sera plus facile de noter ces décompositions comme des mots sur l'alphabet $\mathbb{N}^\star$, dont les lettres seront rangées en ordre décroissant. On écrira par exemple :

$$4^1 = 3^1.1^1 = 2^2 = 2^1.1^2 = 1^4.$$

Programme 3 Décomposition d'un entier

```
let rec décompose n =
    let rec gonfle_accu k ll accu =
        match ll with
            | [] -> accu
            | t :: q -> gonfle_accu k q ((k :: t) :: accu)
    in
    let rec décompose_rec n accu = function
        | 0 ->  accu
        | k ->  let ll = décompose (n - k)
                in
                décompose_rec n (gonfle_accu k ll accu) (k - 1)
    in
    match n with
        | 0 -> []
        | 1 -> [ [1] ]
        | n -> décompose_rec n [ [n] ] (n - 1) ;;
```

Pour un tel mot λ, on notera $\lambda \vdash n$ si λ représente une décomposition de n. Ainsi : $2^2.1 \vdash 5$ et $3.2.1^3 \vdash 8$.
Plus généralement, si $\lambda = a_1^{k_1} a_2^{k_2} \dots a_p^{k_p}$, avec $a_1 > a_2 > \dots > a_p$, on aura $\lambda \vdash \sum_{\ell=1}^{p} k_\ell a_\ell$.

Le nombre total de lettres de l'alphabet $\mathbb{N}^\star$ qui constitue le mot λ, et qui vaut donc $\sum_{\ell=1}^{p} k_\ell$, s'appelle la *longueur* de λ et sera noté ici $|\lambda|$.

Fonction génératrice

Soit p_n le nombre de partitions de l'entier n, avec la convention $p_0 = 1$.
On dispose alors (il suffit de développer...) du

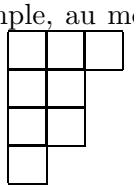
Théorème 3 *La fonction génératrice du nombre de partitions s'écrit*

$$\sum_{n=0}^{+\infty} p_n z^n = 1 \Big/ \Big(\prod_{k=1}^{+\infty} (1 - z^k) \Big).$$

Hélas, il est bien clair que cette équation ne nous avance guère. Nous ferons mieux plus loin.

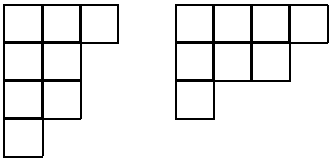
Une représentation graphique des partitions

À une partition de n représentée par le mot $\lambda = a_1^{k_1} a_2^{k_2} \dots a_p^{k_p}$, nous associerons la figure composée de n cases organisées en $|\lambda|$ lignes, les k_1 premières comportant a_1 cases, les k_2 suivantes en comportant a_2, ..., les k_p dernières en comportant enfin a_p.
Par exemple, au mot $3.2^2.1$, qui représente une partition de l'entier 8, sera associé le tableau suivant :



Inversement tout tableau de n cases organisées en lignes alignées à gauche et de taille décroissante

correspond à une partition de l'entier n.
On peut d'autre part associer à tout tableau de ce genre son *transposé*, c'est-à-dire plus géométriquement son symétrique par rapport à la deuxième bissectrice des axes :



Dans l'exemple ci-dessus, nous avons deux partitions de 8 : $3.2^2.1 \vdash 8$ et $4.3.1 \vdash 8$. Nous conviendrons de noter $\lambda^\star$ le mot représenté par le tableau transposé du tableau qui représente le mot λ.
On a bien sûr $\lambda \vdash n \Longleftrightarrow \lambda^\star \vdash n$ et $(\lambda^\star)^\star = \lambda$.
On trouvera dans le programme 4 les instructions CAML qui permettent l'affichage sous forme de ces tableaux des partitions d'un entier.

Programme 4 Impression des partitions d'un entier

```
(* la partition 3.2.2.1  (de somme 8)            *)
(* est représentée par la liste [ 3 ; 2 ; 2 ; 1 ] *)

#open "format" ;;

(* l'impression sous forme de tableau *)
let print_partition p =
    let rec make_bullets = function
        | 0 -> ""
        | n -> "• " ^ (make_bullets (n - 1))
    in
    let print_n_stars n =
        ( print_string (make_bullets n) ; print_break(0,0) )
    in
    open_vbox 0 ;
    do_list print_n_stars p ;
    close_box () ;;

install_printer "print_partition" ;;
```

Le listing 5 page ci-contre montre sur un exemple l'effet du programme précédent.
Plus compliqué est de décrire le mot $\lambda^\star$ à partir du mot λ.
Pour cela, je propose le programme purement fonctionnel 6 page suivante qui mérite à mon avis le détour...

Génération des partitions

Nous nous intéressons maintenant à la génération des partitions d'un entier n : nous avons choisi ici d'utiliser une fonction récursive auxiliaire, qui attend en arguments un entier à décomposer et la valeur maximale des entiers dont nous avons le droit de nous servir pour écrire la décomposition. Ainsi, pour obtenir toutes les partitions de n en entiers au plus égaux à k, il suffit d'ajouter, pour j décrivant l'intervalle d'entiers $[1, k]$, l'entier j en tête de chaque partition de $n-j$ en entiers au plus égaux à j. On est ainsi certain d'écrire les partitions en ordre décroissant.
Tout cela fait l'objet du programme 7 page 10.

Programme 5 Exemple d'impression de partitions simples

```
#let exemple = [ 3 ; 2 ; 2 ; 1 ] ;;
exemple ;; (transpose exemple) ;; (transpose (transpose exemple)) ;;

exemple : int list = • • •
                     • •
                     • •
                     •

#- : int list = • • •
                • •
                • •
                •

#- : int list = • • • •
                • • •
                •

#- : int list = • • •
                • •
                • •
                •
```

Programme 6 La transposition d'une partition

```
(* la partition 3.2.2.1  |-  8                    *)
(* est représentée par la liste [ 3 ; 2 ; 2 ; 1 ] *)

let somme = it_list (prefix +) 0 ;;

let transpose p =
    let nb_de_plus_grands k liste =
            it_list (fun n x -> if x >= k then n + 1 else n) 0 liste
    in
    let rec intervalle i j =
        if i > j then []
        else i :: (intervalle (i+1) j)
    in
    map (function x -> nb_de_plus_grands x p) (intervalle 1 (hd p)) ;;
```

Programme 7 Génération des partitions d'un entier

```
let rec intervalle i j =
    if i > j then []
    else i :: (intervalle (i+1) j) ;;

let partitions n =
    let cons a b = a :: b
    in
    let rec aux n k =
        if k > n then aux n n
        else if k = 0 && n = 0 then [[]]    (* fin normale de récursion *)
        else if k = 0 then []               (* pas de solution ici !    *)
        else it_list (fun l j -> map (cons j) (aux (n-j) j) @ l)
                     []
                     (intervalle 1 k)
    in
    aux n n ;;
```

Nombres pentagonaux et récurrence sur le nombre de partitions

Considérons le petit tableau suivant :

k	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
$E_k = \dfrac{k(3k-1)}{2}$	57	40	26	15	7	2	0	1	5	12	22	35	51
$\dfrac{k(3k+1)}{2}$	51	35	22	12	5	1	0	2	7	15	26	40	57

La figure 1 page suivante justifie que ces nombres sont appelés *nombres pentagonaux* depuis Euler. On notera $E_k = \dfrac{k(3k-1)}{2}$, et on obtiendra l'ensemble des nombres pentagonaux quand k décrit $\mathbb{Z}$.

Nous noterons dans la suite Λ_n l'ensemble des mots λ qui décrivent les partitions de l'entier n **sans répétition** : nous entendrons par là que nous ne considérerons que les mots de la forme $a_1^1 a_2^1 \dots a_p^1$ avec $a_1 > a_2 > \dots > a_p$.

Le programme 8, qui est une modification facile du programme 7, réalise la génération des partitions sans répétition d'un entier n.

Programme 8 Génération des partitions sans répétition

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

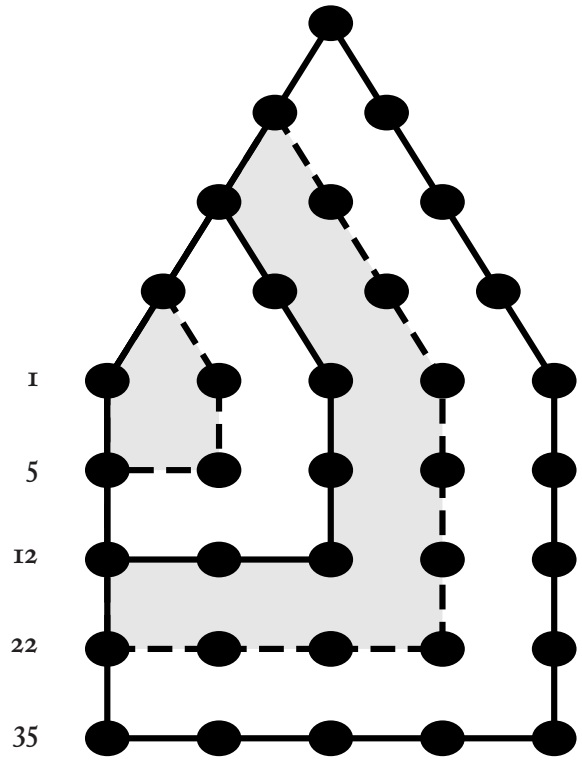


Figure 1: Nombres pentagonaux

Nous adoptons également la convention américaine pour noter les cardinaux, et $\sharp\Lambda_n$ désignera donc le nombre de partitions de n.
Nous notons Λ_n^0 (*resp.* Λ_n^1) l'ensemble des partitions **sans répétition** de l'entier n qui sont de longueur paire (*resp.* impaire).
Ainsi : $\Lambda_n^0 = \{\lambda \in \Lambda_n, |\lambda| \text{ est pair}\}$ et $\Lambda_n^1 = \{\lambda \in \Lambda_n, |\lambda| \text{ est impair}\}$
Avec ces notations nous pouvons énoncer le

Théorème 4 *Si n n'est pas un nombre pentagonal, alors $\sharp\Lambda_n^0 = \sharp\Lambda_n^1$.*
Sinon $\left|\sharp\Lambda_n^0 - \sharp\Lambda_n^1\right| = 1$ et, plus précisément, si n s'écrit $k(3k-1)/2$ avec $k \in \mathbb{Z}$, alors :

- *si k est pair, $\sharp\Lambda_n^0 = 1 + \sharp\Lambda_n^1$;*
- *si k est impair, $\sharp\Lambda_n^1 = 1 + \sharp\Lambda_n^0$.*

Ainsi, par exemple :

$$\begin{array}{rcl@{\qquad}rcl}
\Lambda_5^0 &=& \{4.1, 3.2\} & \sharp\Lambda_5^0 &=& 3 = 1+2 \\
\Lambda_5^1 &=& \{5\} & \sharp\Lambda_5^1 &=& 2 \\
\Lambda_8^0 &=& \{7.1, 6.2, 5.3\} & \sharp\Lambda_8^0 &=& 3 \\
\Lambda_8^1 &=& \{8, 5.2.1, 4.3.1\} & \sharp\Lambda_8^1 &=& 3 \\
\Lambda_{12}^0 &=& \{11.1, 10.2, 9.3, 8.4, 6.3.2.1, 5.4.2.1\} & \sharp\Lambda_{12}^0 &=& 6 \\
\Lambda_{12}^1 &=& \{12, 9.2.1, 8.3.1, 7.4.1, 7.3.2, 6.5.1, 6.4.2\} & \sharp\Lambda_{12}^1 &=& 7 = 1+6
\end{array}$$

Base et flanc d'une partition sans répétition

Avant de commencer la démonstration, nous aurons besoin d'une définition à propos des tableaux qui décrivent nos partitions. Comme nos partitions sont supposées sans répétition, chaque ligne contiendra un nombre de cases **strictement** supérieur à la suivante.
Pour chacune des partitions (sans répétition — ce qui sera sous-entendu jusqu'à la prochaine sous-section), on dessine le tableau correspondant. Sa *base* est constituée de la dernière — et plus petite — ligne. Son *flanc* regroupe le nombre maximal de cases à partir de la dernière de la première ligne en descendant à 45 degrés vers la gauche. La figure 2 page 14, dans son premier schéma, explicite cette définition.
Le programme 9 permet de calculer la taille de la base et du flanc d'une partition supposée sans répétition.

Programme 9 Base et flanc d'une partition

```
let rec last = function
    | [] -> failwith "last"
    | [a] -> a
    | t :: q -> last q ;;

let taille_base = last ;;

let rec taille_flanc = function
    | [] -> 0
    | a :: b :: q when a = b + 1 -> 1 + (taille_flanc (b :: q))
    | _ -> 1 ;;
```

La ligne médiane de la même figure 2 page 14 montre deux tableaux tels que

— la base et le flanc se coupent ;

— la longueur de la base dépasse d'au plus une unité celle du flanc.

Remarquons que si k est le nombre de lignes d'un tel tableau,

— dans le cas où base et flanc ont la même taille, le tableau contient très exactement $k\,\dfrac{k+(k+(k-1))}{2} = \dfrac{k(3k-1)}{2}$ cases;

— et sinon, $k\,\dfrac{k+1+(k+1+(k-1))}{2} = \dfrac{k(3k+1)}{2}$.

C'est dire qu'il y a dans ce cas un nombre pentagonal de cases. Je laisse le lecteur vérifier que c'est bien là le seul cas.
Pour toute autre partition (sans répétition), on se trouve dans le cas des derniers schémas de la figure 2 page 14. On peut alors ou bien déplacer les cases du flanc pour former une nouvelle base, ou bien au contraire flanquer la partition des cases de la base.
Le programme 10 page ci-contre décide de la manipulation qu'on peut opérer sur une partition sans répétition donnée : rien si on est dans le cas n pentagonal, ou bien un échange base/flanc sinon.
Enfin, le programme 11 page suivante procède aux manipulations proprement dites.

Programme 10 Choix de la manipulation à effectuer

```
let intersection_BF l = (taille_flanc l) = (list_length l) ;;

type manipulation = BF | FB | NOP ;;

let que_faire l =
    let tf = taille_flanc l
    and tb = taille_base  l
    and n  = list_length  l
    in
    if (n = tf) && ( (tb = tf) || (tb = 1 + tf)) then NOP
    else if tf >= tb then BF
    else FB ;;
```

Programme 11 Manipulation des partitions sans répétition

```
let rec équeute = function (* détruit le dernier élément d'une liste *)
    | [] -> failwith "équeute"
    | [a] -> []
    | t :: q -> t :: (équeute q) ;;

let rec map_borné f k = function
    | [] -> []
    | t :: q when k > 0 -> (f t) :: (map_borné f (k - 1) q)
    | l -> l ;;

let manipule l = match que_faire l with
    | NOP -> l
    | BF -> équeute (map_borné succ (taille_base l) l)
    | FB -> (map_borné pred (taille_flanc l) l) @ [ taille_flanc l ] ;;
```

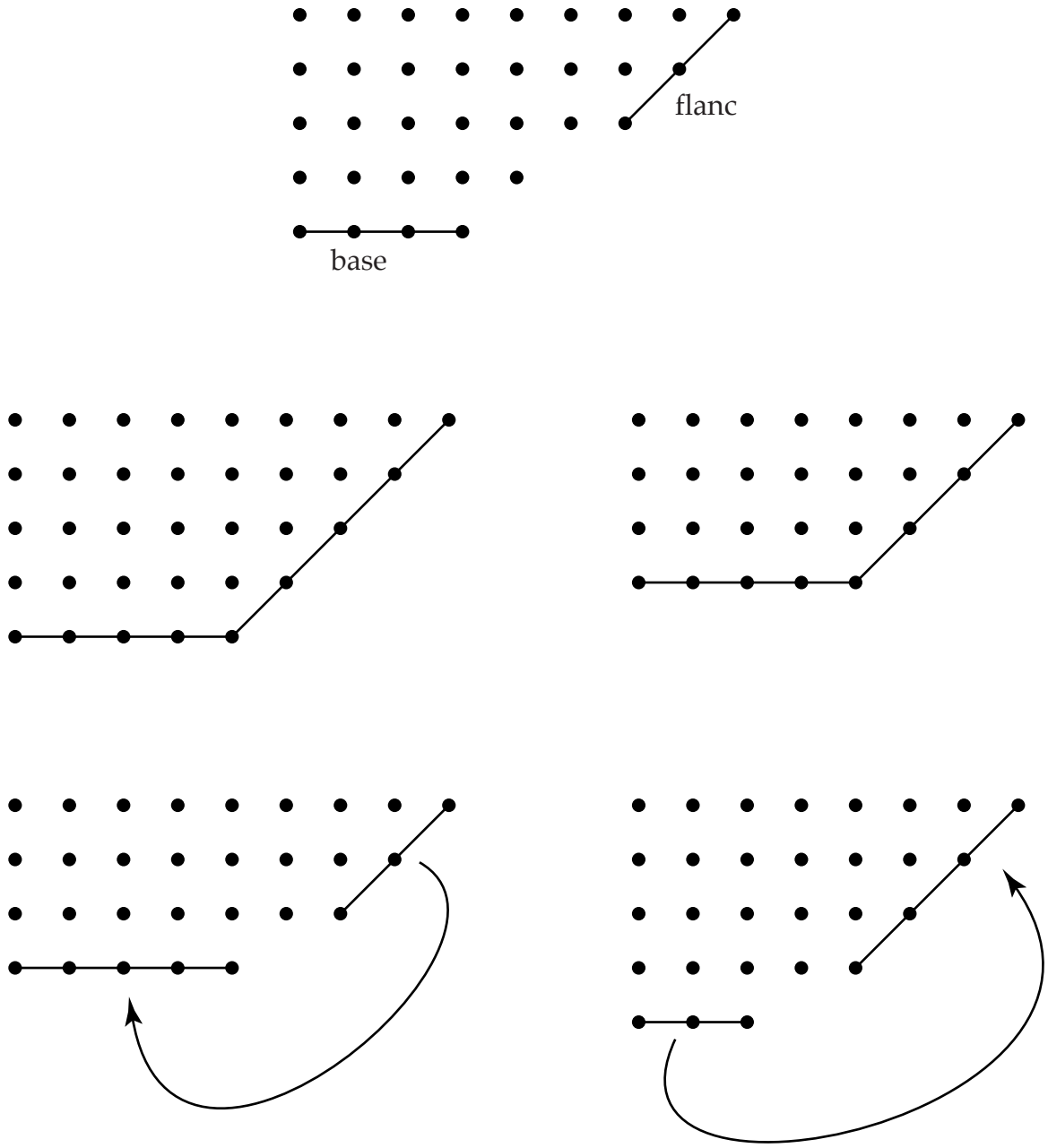


Figure 2: Base et flanc d'un tableau, et classification des tableaux

Démonstration du théorème 4 page 11

Les manipulations précédentes permettent d'apparier deux à deux les partitions d'un entier n, sauf — dans le cas où n est pentagonal — pour **la** partition associée à `NOP`.
On remarque alors que deux partitions ainsi appariées ont l'une un nombre pair et l'autre un nombre impair de lignes.
Le résultat en découle aussitôt.

Application : une récurrence sur le nombre de partitions

On déduit immédiatement du théorème 4 page 11 la fonction génératrice de $u_n = \sharp\Lambda_n^0 - \sharp\Lambda_n^1$ qui s'écrit $\displaystyle\sum_{n=1}^{+\infty} u_n z^n = \sum_{k\in\mathbb{Z}} (-1)^k z^{k(3k-1)/2}$.
Mais, dans le développement de $\displaystyle\prod_{n=1}^{+\infty}(1-z^n)$, z^m apparaît pour chaque partition sans répétition de m en $m = m_1 + \cdots + m_p$ avec un coefficient égal à $(-1)^p$. C'est dire qu'on obtient au total comme coefficient de z^m l'entier $u_m = \sharp\Lambda_m^0 - \sharp\Lambda_m^1$. Bref, on peut énoncer le

Théorème 5

$$\prod_{n=1}^{+\infty}(1-z^n) = \sum_{k\in\mathbb{Z}} (-1)^k z^{k(3k-1)/2} = 1 + \sum_{k=1}^{+\infty} (-1)^k \left(z^{k(3k-1)/2} + z^{k(3k+1)/2} \right).$$

On est maintenant en mesure d'écrire le

Théorème 6 *Les entiers p_n, qui dénombrent les partitions d'un entier, vérifient la récurrence :*

$$\begin{aligned} \forall n \geqslant 1, \qquad p_n &= \sum_{k>0} (-1)^{k-1} (p_{n-k(3k-1)/2} + p_{n-k(3k+1)/2}) \\ &= (p_{n-1} + p_{n-2}) - (p_{n-5} + p_{n-7}) + (p_{n-12} + p_{n-15}) - (p_{n-22} + p_{n-26}) + \cdots \end{aligned}$$

✧ D'après le théorème 3 page 7, on a

$$\prod_{n=1}^{+\infty} p_n z^n \times \left(1 + \sum_{k=1}^{+\infty} (-1)^k \left(z^{k(3k-1)/2} + z^{k(3k+1)/2} \right) \right) = 1.$$

Développant, on identifie le coefficient de z^n dans le membre gauche de l'équation précédente à 0 pour obtenir le résultat escompté. ✦

On peut facilement écrire une fonction récursive qui calcule ces nombres p_n, ce qui fait l'objet du programme 12 page suivante.
De la même façon qu'on utilise l'option `remember` en MAPLE, on peut écrire une fonction moins naïve, comme celle du programme 13 page 17.
On obtient les premiers p_n que voici :

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
p_n	1	1	2	3	5	7	11	15	22	30	42	56	77	101	135	176	231	297	385	490

Programme 12 Une version naïve du calcul des p_n

```
exception Stop ;;

let even n = (n mod 2) = 0
and odd  n = (n mod 2) = 1 ;;

let rec nb_de_partitions_naïf n =
    if n < 0 then 0
    else if n < 2 then 1
    else    let résultat = ref 0
            and k = ref 1
            in
            try
            while true do
                let k1 = (!k * (3 * !k - 1)) / 2
                and k2 = (!k * (3 * !k + 1)) / 2
                in
                if k1 > n then raise Stop ;
                if even(!k) then
                    résultat := !résultat
                        - (nb_de_partitions_naïf (n - k1))
                        - (nb_de_partitions_naïf (n - k2))
                else
                    résultat := !résultat
                        + (nb_de_partitions_naïf (n - k1))
                        + (nb_de_partitions_naïf (n - k2)) ;
                incr k
            done ; 0
            with Stop -> !résultat ;;
```

Programme 13 Une version plus raisonnable du calcul des p_n

```
exception Stop ;;

let even n = (n mod 2) = 0
and odd  n = (n mod 2) = 1 ;;

let nb_de_partitions =
    let mémoire = ref [ (0,1) ; (1,1) ; (2,2) ]
    in
    let rec f n =
        if n < 0 then 0
        else try assoc n !mémoire with Not_found
                ->  let résultat = ref 0
                    and k = ref 1
                    in
                    try
                    while true do
                        let k1 = !k * (3 * !k - 1) / 2
                        and k2 = !k * (3 * !k + 1) / 2
                        in
                        if k1 > n then raise Stop ;
                        if even(!k) then
                            résultat := !résultat - (f (n - k1)) - (f (n - k2))
                        else
                            résultat := !résultat + (f (n - k1)) + (f (n - k2)) ;
                        incr k
                    done ; 0
                    with Stop
                        ->  (   mémoire := (n,!résultat) :: !mémoire ;
                                !résultat   )
    in f ;;
```

La correspondance de Robinson-Schensted-Knuth

Tableaux de Young

Nous avons associé plus haut à une partition $\lambda = a_1^{k_1} \dots a_p^{k_p}$ un tableau de cases vides organisées en $|\lambda| = \sum k_i$ lignes.
Nous considérons maintenant ce même genre de tableaux de cases que cette fois nous remplissons avec les entiers de 1 à n en imposant la condition suivante : quand on lit une ligne (de gauche à droite) ou une colonne (de haut en bas) quelconque du tableau, on rencontre des entiers rangés en ordre croissant. C'est dire en particulier que la case le plus en haut à gauche contient toujours l'entier 1. Un tel tableau de cases pleines s'appelle un *tableau de Young*, et j'appelle *forme* de ce tableau de Young le tableau de cases vides associé.
Pour une partition λ fixée de n, et donc une forme de tableaux de Young fixée, on peut bien entendu utiliser plusieurs solutions pour remplir le tableau de Young, et nous noterons ainsi $Y_n(\lambda)$ le nombre de tableaux de Young dont la forme est le tableau associé à la partition λ.
Par exemple, je laisse le soin au lecteur de vérifier que $Y_4(3.1) = 3$, les tableaux de Young correspondants étant les suivants :

1	2	3
4		

1	2	4
3		

1	3	4
2		

On peut par exemple démontrer le

Théorème 7 *Pour tout entier naturel $k \geqslant 1$ on dispose de l'égalité suivante, qui fait intervenir le nombre de Catalan C_k :*

$$Y_{2k}(k^2) = C_k = \frac{1}{k+1}\binom{2k}{k}.$$

La démonstration de ce théorème est — de façon étonnante — plutôt difficile. Je laisse le lecteur intéressé par les tableaux de Young lire avec soin la section 5.1.4 (pages 48–67) du volume 3, *Sorting and Searching*, de *The Art of Computer Programming*, par Donald E. Knuth.

L'algorithme de Robinson, Schensted et Knuth (RSK)

L'idée est d'associer à une permutation σ de S_n un couple (D, E) de deux tableaux de Young de même forme (à n cases, bien sûr) de telle sorte que l'application obtenue soit bijective.
Avant de démontrer cette propriété, d'en découvrir d'autres, et d'obtenir de jolis corollaires, décrivons l'algorithme RSK.
On va créer de façon incrémentale les deux tableaux de Young (D, E). On part de deux tableaux vides, et on ajoute au fur et à mesure une case à chacun d'eux : on remarquera, c'est important, que tout le long de l'algorithme les deux tableaux D et E ont la même forme. Cependant, alors que E reste un vrai tableau de Young en permanence, il n'en est pas de même de D.
Mais prenons plutôt un exemple, celui de la permutation $\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 2 & 5 & 3 & 1 \end{pmatrix}$.
On obtient successivement les tableaux de la figure 3 page suivante.

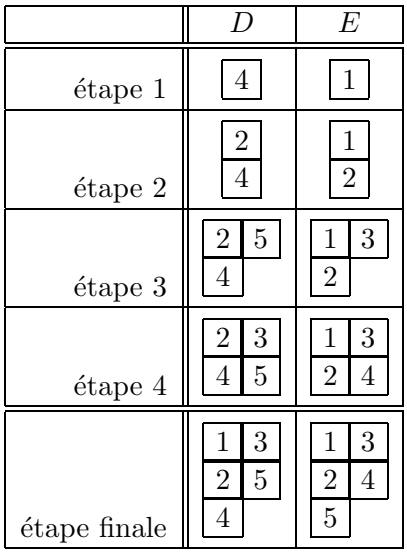


Figure 3: Les étapes successives de l'algorithme RSK sur un exemple

Voici une description formelle de l'algorithme RSK :

1. on commence par initialiser nos deux tableaux à vide ;
2. pour k variant de 1 à n on réalise les deux opérations suivantes :
   (a) on procède à l'insertion de $\sigma(k)$ dans la première ligne de D, conformément à la méthode décrite ci-après ;
   (b) on crée une nouvelle case vide dans E, de telle sorte que E ait la même forme que D, et on y écrit l'entier k.

Tout le mystère réside dans la procèdure d'insertion (dans D) d'un nouvel entier.
Voici quelle est la procédure d'insertion de l'entier x dans la ligne i de D :

— si x est plus grand que le dernier élément de la ligne i (ou si cette ligne est vide), on l'ajoute simplement en bout de ligne (dans une nouvelle case, bien sûr) ;

— sinon, si a est le premier terme de la ligne i de D qui soit plus grand que x, on *éjecte* a de cette ligne, en remplaçant a par x, et en insérant — de façon récursive — a dans la ligne $i+1$ de D.

On se convainc facilement que l'algorithme RSK conserve la monotonie en lignes et colonnes des deux tableaux (c'est un invariant de boucle). Il est peut-être un peu plus délicat de comprendre pourquoi si à une permutation σ on associe le couple (D, E), on associe à la permutation inverse σ^{-1} le couple (E, D).
Il me semble qu'il vaut mieux pratiquer l'algorithme soi-même sur plusieurs exemples pour s'efforcer à imaginer la méthode permettant de reconstituer la permutation à partir du couple final (D, E), que rédiger *in extenso* une démonstration que la correspondance réalisée est bijective.
Qu'il suffise de dire qu'on procède à rebours, pour k variant de n à 1, en écrivant les différents couples (D, E) intermédiaires. On observe que $\sigma(k)$ est finalement toujours écrit dans une case de la première ligne, et on n'a plus qu'à reconstituer la succession de *sauts* qui ont été réalisés.

Prenons un exemple : quelle est la permutation associée au couple

$$D = \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 & 5 \\ \hline \end{array}\begin{array}{|c|}\hline 4 \\ \hline \end{array} \qquad E = \begin{array}{|c|c|}\hline 1 & 3 \\ \hline 2 & 4 \\ \hline \end{array}\begin{array}{|c|}\hline 5 \\ \hline \end{array} \quad ?$$

Cherchons $\sigma(5)$.
On repère la position dans E du chiffre 5 : 4 a été éjecté dans la dernière ligne, créant la nouvelle case. Mais 4 a été éjecté par le dernier entier plus petit de la ligne précédente, c'est-à-dire ici par 3, et 3 lui-même a été éjecté par le dernier entier plus petit de la première ligne, c'est-à-dire par 2. Comme on est arrivé à la première ligne, on est sûr que $\sigma(5) = 2$, et on peut reconstituer les tableaux de l'étape précédente :

$$D = \begin{array}{|c|c|}\hline 1 & 3 \\ \hline 4 & 5 \\ \hline \end{array} \qquad E = \begin{array}{|c|c|}\hline 1 & 3 \\ \hline 2 & 4 \\ \hline \end{array}.$$

On recommence, pour trouver $\sigma(4)$. La position du 4 dans E nous dit de nous intéresser à l'histoire de 5 dans E. Or 5 a été éjecté par 3. Ainsi $\sigma(4) = 3$.
Je laisse le lecteur continuer… on obtient la permutation $\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 5 & 3 & 2 \end{pmatrix}$.

Programmation de l'algorithme RSK et de l'algorithme inverse

Le programme 14 page ci-contre implémente l'algorithme RSK en Caml.
Un tableau de Young est représenté par une `int list list`, très naturellement, et une permutation par un `int vect` : nous choisissons d'éviter tout souci d'indice en écrivant des permutations de l'ensemble $\{0, 1, \ldots, n-2, n-1\}$.
En lignes 25–27, nous appliquons successivement les insertions dans les tableaux D et E des entiers k et de leurs images $\sigma(k)$.
La fonction `substitue`, définie dans les lignes 4–11, prend en arguments k, la valeur x à insérer dans D, et les lignes `dl` et `el` de D et E où on doit réaliser l'insertion. Elle fait son travail, et renvoie, outre les nouvelles lignes obtenues pour D et E, ou bien la valeur `None` si on a terminé (l'insertion s'est donc faite — pour D — en fin de ligne), ou bien la valeur `Some(a)` si il y a eu éjection de la valeur `a`.
Ces valeurs `None` et `Some of 'a` définissent le type prédéfini `'a option`.
Il ne reste plus qu'à répercuter l'éjection éventuelle : c'est le rôle de la fonction `insertion`, qui est définie en lignes 13–23.
Remarquons que les lignes 11 et 23 n'ont d'autre rôle qu'éviter à Caml de nous annoncer des filtrages non exhaustifs : en réalité (sauf erreur de programmation) nous savons que D et E ont la même forme, et ces cas de filtrage sont inutiles.
Le programme 15 page 22 implémente la transformation inverse de l'algorithme RSK en Caml.
On reprend tout d'abord (en lignes 1–9) les fonctions `last` qui renvoie le dernier élément d'une liste et `équeute` qui renvoie la liste argument privée de son dernier élément.
En lignes 12–13, on transforme les tableaux D et E en vecteurs, d'un usage plus aisé ici. Le programme consiste simplement en une boucle à rebours (ligne 43) de calcul de chacun des $\sigma(k)$, et on renvoie la permutation ainsi obtenue.
On commence par appliquer la fonction `cherche`, définie en lignes 18–27, qui renvoie le couple `(i,a)` constitué du numéro de la ligne où se trouve dans E l'entier k et de la valeur correspondante dans le tableau D. En outre, en ligne 23, `cherche` se charge tout seul d'effacer les cases

Programme 14 Algorithme RSK

```
let RSK_direct sigma =
    let n = vect_length sigma
    in
    let rec substitue k a dl el = match dl,el with
        | [],[] -> [a],[k],None
        | (td::qd),(te::qe) when td < a
            ->  let qd',qe',suite = substitue k a qd qe
                in
                (td::qd'),(te::qe'),suite
        | (td::qd),_ -> (a::qd),el,Some(td)
        | _ -> failwith "bug"
    in
    let rec insertion k a_k (d,e) =
        match d,e with
            | [],[] -> ([[a_k]],[[k]])
            | (dl :: dq),(el :: eq)
                -> ( match substitue k a_k dl el with
                        | dl',el',None -> (dl'::dq),(el'::eq)
                        | dl',el',Some(a)
                            ->  let dq',eq' = insertion k a (dq,eq)
                                in
                                (dl'::dq'),(el'::eq') )
            | _ ->  failwith "bug"
    in
    it_list (fun (d,e) k -> insertion k sigma.(k) (d,e))
            ([],[])
            (intervalle 0 (n-1)) ;;
```

Programme 15 Algorithme RSK-inverse

```
let rec last = function
    | [] -> failwith "last"
    | [a] -> a
    | t :: q -> last q ;;

let rec équeute = function (* détruit le dernier élément d'une liste *)
    | [] -> failwith "équeute"
    | [a] -> []
    | t :: q -> t :: (équeute q) ;;

let RSK_inverse (d,e) =
    let dv = vect_of_list d
    and ev = vect_of_list e
    and n = it_list (fun x l -> x + (list_length l)) 0 d
    in
    let sigma = make_vect n 0
    in
    let cherche k =
        let rec aux i =
            if last ev.(i) = k then
                let a = last dv.(i)
                in
                dv.(i) <- équeute dv.(i) ; ev.(i) <- équeute ev.(i) ;
                (i,a)
            else aux (i+1)
        in
        aux 0
    in
    let rec substitue l a = match l with
        | [ x ] -> [a],x
        | x :: y :: q when y > a -> (a::y::q),x
        | x :: q -> let q',a = substitue q a in (x::q'),a
        | _ -> failwith "bug"
    in
    let rec remonte i a = match i,(substitue dv.(i) a) with
        | 0,(l,a) -> dv.(i) <- l ; a
        | _,(l,a) -> dv.(i) <- l ; remonte (i-1) a
    in
    let calcule k = match cherche k with
        | 0,a -> sigma.(k) <- a
        | i,a -> sigma.(k) <- remonte (i-1) a
    in
    for k = n-1 downto 0 do calcule k done ;
    sigma ;;
```

correspondantes. La récursion est écrite sans test d'arrêt : on a intérêt à donner à notre fonction `RSK_inverse` de vrais tableaux de Young !
Notons au passage qu'à ce moment le traitement du tableau E est achevé.
Il s'agit alors de remonter la valeur `a` obtenue, sauf bien sûr si on est à la première ligne : c'est ce que distinguent les lignes 40–41.
La remontée (lignes 35–37) se fait bien sûr récursivement. Elle utilise la fonction `substitue` qui prend en arguments la ligne du tableau D à considérer et la valeur à remonter, et renvoie la ligne modifiée et la nouvelle valeur à remonter.

Conséquences

On déduit immédiatement de ce que la correspondance RSK est bijective le

Théorème 8 *On a la relation :* $\displaystyle\sum_{\lambda\vdash n}(Y_n(\lambda))^2 = n!$.

On peut aussi observer que d'après les propriétés de la correspondance RSK que nous avons mentionnées, les involutions (c'est-à-dire les permutations σ telles que $\sigma\circ\sigma$ est l'identité) correspondent aux couples (D, E) tels que $D = E$.
Si donc ι_n désigne le nombre d'involutions de S_n, on a

$$\iota_n = \sum_{\lambda\vdash n} Y_n(\lambda).$$

Signalons au passage la récurrence suivante sur les ι_n:

Théorème 9 *Avec la convention $\iota_0 = 1$, on a, pour tout entier naturel $n \geqslant 2$:*

$$\iota_n = \iota_{n-1} + (n-1)\iota_{n-2}.$$

✧ Si on pense à la décomposition en cycles des permutations, on caractérise les involutions comme étant les permutations qui se décomposent en cycles de longueur au plus égale à 2. Autrement dit, les involutions sont les permutations qui sont produit de transpositions de supports disjoints.
La récurrence s'interprète alors de la façon suivante : ou bien $\sigma(n) = n$ et on réalise une involution des $n-1$ premiers éléments, ou bien on choisit $\sigma(n)$ parmi les $n-1$ premiers entiers, et on réalise une involution des $n-2$ restants. ✥

Le programme 16 calcule les nombres ι_n.

Programme 16 Calcul des ι_n

```
let iota n =
    let rec aux a b k =
        if k = n then a
        else aux b (b + (k + 1) * a) (k + 1)
    in
    aux 1 1 0 ;;
```

On peut même obtenir la série génératrice exponentielle de cette suite :

Théorème 10 *La série génératrice exponentielle des* (ι_n) *vaut :*

$$I(z) = \sum_{n=0}^{+\infty} \frac{\iota_n}{n!} z^n = e^{z+z^2/2}.$$

✧ La relation de récurrence ci-dessus peut s'écrire aussi

$$n\frac{\iota_n}{n!} = \frac{\iota_{n-1}}{(n-1)!} + \frac{\iota_{n-2}}{(n-2)!},$$

que l'on traduit aussitôt par $I'(z) = (1+z)I(z)$.
On observe que $I(0) = \iota_0 = 1$, et on résout l'équation différentielle. C'est tout. ✥

Le plus amusant est de faire le lien avec l'étude précédente pour écrire le

Théorème 11 *On a :* $\forall n \geqslant 1, \qquad \sqrt{n!} \leqslant \iota_n \leqslant \sqrt{p_n n!}.$

✧ La minoration est claire :

$$n! = \sum_{\lambda \vdash n} Y_n(\lambda)^2 \leqslant \left(\sum_{\lambda \vdash n} Y_n(\lambda) \right)^2 = \iota_n^2.$$

La majoration découle de l'inégalité de Cauchy-Schwarz appliquée dans l'espace de dimension $p_n = \sharp\{\lambda, \lambda \vdash n\}$ au produit scalaire du vecteur $(1, \ldots, 1)$ et du vecteur $(Y_n(\lambda_1), \ldots, Y_n(\lambda_{p_n}))$, où λ_i est une numérotation des partitions de l'entier n. ✥

Un peu de lambda-calcul

λ-termes

Les λ-termes, ou termes du λ-calculs, sont définis de la façon suivante.
On considère un ensemble $\mathcal{V}$ dénombrable de *variables*. On définit alors l'ensemble Λ des λ-termes de façon inductive :

— toute variable $a \in \mathcal{V}$ est un λ-terme ;

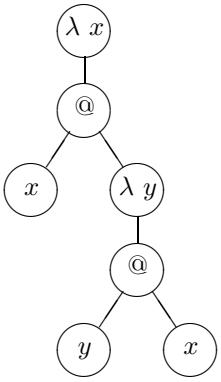
— si $a \in \mathcal{V}$ est une variable, et si t est un λ-terme, alors $(\lambda\ a.t)$ est un λ-terme ;

— si t_1 et t_2 sont deux λ-termes, alors $(t_1\ t_2)$ est un λ-terme, appelé application de t_1 à t_2.

Intuitivement, un λ-terme tel que $\lambda\ x.(y\ x)$ est à rapprocher du CAML `function x -> (y x)`.
De même, l'application identique sera-t-elle représentée par le λ-terme $\lambda\ x.x$.
La définition précédente correspond exactement à la déclaration de type CAML:

```
type lambda_terme =
    | Variable of string
    | Application of lambda_terme * lambda_terme
    | Abstraction of string * lambda_terme ;;
```

On associera à chaque λ-terme un arbre dont les feuilles porteront les variables, et dont les nœuds se divisent en deux genres : les nœuds d'applications, qui auront deux fils, et les nœuds λ, qui contiennent le nom de la variable associée à λ, et un seul fils.
Par exemple, au terme $(\lambda\ x.(x\ (\lambda\ y.(y\ x))))$ est associé l'arbre suivant :



Variables libres et liées

Considérations théoriques

Soit t un terme, représenté par un arbre a. On appelle occurrence du terme t tout sous-arbre de a, qu'on identifiera par un mot sur $\mathbb{N}^\star$ de la façon suivante : on numérote de gauche à droite les arètes issues d'un même nœud. Le mot associé à une occurrence u sera simplement la suite des numéros lus lors du parcours depuis la racine de l'arbre jusqu'à l'occurrence u.
Par exemple, les occurrences du terme précédent, $(\lambda\ x.(x\ (\lambda\ y.(y\ x))))$, seront désignées par les mots ε, 1, 11, 12, 121, 1211, 1212, et correspondent aux sous-termes suivants respectifs $(\lambda\ x.(x\ (\lambda\ y.(y\ x))))$, $(x\ (\lambda\ y.(y\ x)))$, x, $(\lambda\ y.(y\ x))$, $(y\ x)$, y et x.

Plus précisément :

— le terme $t = a$, où a est une variable, n'a qu'une occurrence, ε, qui correspond au terme a lui-même ;

— le terme $t = \lambda\ a.s$ a pour occurrences ε, qui correspond à t tout entier ; 1, qui correspond à s ; et, si u est une occurrence de s correspondant au sous-terme w, $1u$ qui correspond justement à w ;

— le terme $t = (t_1\ t_2)$ a pour occurrences ε, qui correspond à t tout entier ; 1, qui correspond à t_1 ; 2, qui correspond à t_2 ; et, si u est une occurrence de t_1 (*resp.* t_2) qui correspond au sous-terme w, $1u$ (*resp.* $2u$) qui correspond justement à s.

Dans la suite, on notera $\mathcal{O}(t)$ l'ensemble des occurrences d'un terme t, et, si $u \in \mathcal{O}(t)$, t/u le sous-terme de t correspondant à u.
On a donc, pour une variable a et des λ-termes t, t_1 et t_2 : $\mathcal{O}(a) = \{\varepsilon\}$, $t/\varepsilon = t$, $\mathcal{O}(\lambda\ a.t) = \{\varepsilon\} \cup 1.\mathcal{O}(t)$ et $(\lambda\ a.t)/(1.u) = t/u$ si $u \in \mathcal{O}(t)$, et enfin $\mathcal{O}(t_1\ t_2) = \{\varepsilon\} \cup 1.\mathcal{O}(t_1) \cup 2.\mathcal{O}(t_2)$ et $(t_1\ t_2)/(1.u) = t_1/u$ si $u \in \mathcal{O}(t_1)$ et $(t_1\ t_2)/(2.u) = t_2/u$ si $u \in \mathcal{O}(t_2)$.
Soit maintenant t un terme et $u \in \mathcal{O}(t)$ une occurrence de t telle que t/u soit une variable a. On dit plus simplement que u est une occurrence de la variable a dans t. Remontons alors dans l'arbre de u jusqu'à la racine. Deux cas peuvent se présenter :

— on ne rencontre, lors de cette remontée vers la racine, aucun nœud $\lambda\ a$, on dit alors que u est une occurrence *libre* de la variable a dans t ;

— sinon, on dit que u est une occurrence *liée* de la variable a dans t. La première occurrence de $\lambda\ a$ rencontrée lors de cette remontée est appelée le *lieur* de l'occurrence u de a. Le nombre d'occurrences de $\lambda\ b$ avec $b \neq a$ qu'il a fallu traverser s'appelle la *hauteur de liaison* de u.

Par exemple, dans le cas du terme $(\lambda\ x.(x\ (\lambda\ y.(y\ x))))$, l'occurrence 11 de x est liée par ε, et sa hauteur de liaison est nulle. De même, l'occurrence 1212 de x est liée par ε, mais sa hauteur de liaison vaut 1. Enfin, l'occurrence 1211 de y est liée par 12, et sa hauteur de liaison est nulle.
Une **variable** a est dite *libre* dans un terme t si elle a au moins une occurrence libre. L'ensemble des variables libres, noté $var(t)$ est donc défini inductivement par :

$$var(a) = \{a\}, \quad var(\lambda\ a.t) = var(t) \setminus \{a\}, \quad var(t_1\ t_2) = var(t_1) \cup var(t_2).$$

Un terme qui ne possède pas de variable libre est dit *fermé*, ou *clos*. On dit encore qu'il s'agit d'un *combinateur*.

Programmation

Le programme 17 explicite deux fonctions sur les λ-termes :

— `lt_occurrences : string -> lambda_terme -> int list list * int list list`

prend en arguments une variable v et un λ-terme t. Elle renvoie un couple `(free,bound)` de listes : `free` est la liste des occurrences *libres* de v dans t, alors que `bound` est la liste des occurrences *liées* de v dans t. Une occurrence est représentée par une liste d'entiers (à prendre parmi 1 et 2), tout naturellement ;

— `lt_libres : lambda_terme -> (int list * string) list` prend en argument un λ-terme t et renvoie une liste de couples (v, u), où u est une occurence *libre* de la variable v dans t. `lt_libres` décrit bien sûr l'ensemble $var(t)$.

Programme 17 Occurrences de variables dans un λ-terme

```
let cons a b = a :: b ;;

let lt_occurrences x t =
    let ajoute c = map (cons c)
    in
    let rec aux est_liée = function
        | Variable(a) when a = x -> if est_liée then [[]],[] else [],[[]]
        | Variable(_) -> [],[]
        | Application(t1,t2)
            ->  let free1,bound1 = aux est_liée t1
                and free2,bound2 = aux est_liée t2
                in
                (ajoute 1 free1) @ (ajoute 2 free2) ,
                (ajoute 1 bound1) @ (ajoute 2 bound2)
        | Abstraction(a,t)
            ->  let free,bound = aux (est_liée || x = a) t
                in
                (ajoute 1 free),(ajoute 1 bound)
    in
    aux false t ;;

let lt_libres t =
    let ajoute c = map (function (o,v) -> (c::o,v))
    in
    let rec aux liées = function
        | Variable(a) -> if mem a liées then [] else [[],a]
        | Application(t1,t2) -> (ajoute 1 (aux liées t1)) @ (ajoute 2 (aux liées t2))
        | Abstraction(x,t) -> ajoute 1 (aux (x :: liées) t)
    in
    aux [] t ;;
```

Notations de de Bruijn

Considérations théoriques

Soit t un terme fermé, c'est-à-dire sans variable libre. On va remplacer chaque variable par sa hauteur de liaison. Par exemple, on remplace $(\lambda\ x.(x\ (\lambda\ y.(y\ x))))$ par $(\lambda(0(\lambda(0\ 1))))$.
La figure 4 illustre cette transformation.

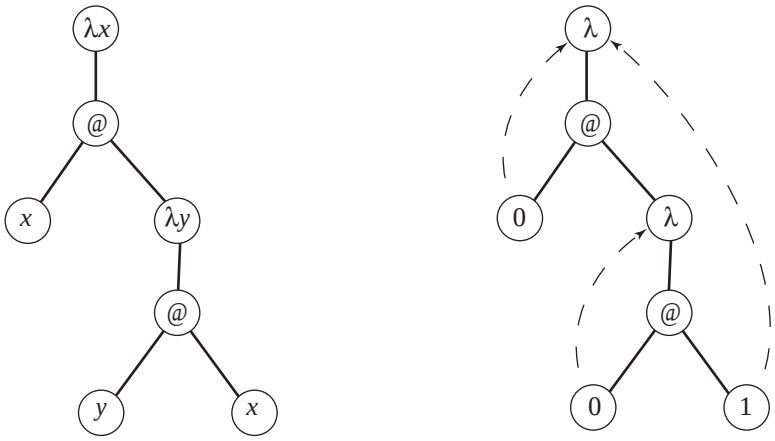


Figure 4: La notation de de Bruijn illustrée par un exemple

Mais que faire avec les occurrences libres de variables ?
On définit la *profondeur d'abstraction* d'une occurrence u d'une variable a dans un terme t comme étant le nombre de $\lambda\ x$ qu'on traverse en remontant dans l'arbre de u jusqu'à la racine.
Rappelons qu'on a dit que l'ensemble $\mathcal{V}$ des variables est dénombrable : choisissons une fois pour toutes une numérotation $\mathcal{V} = \{v_0, v_1, v_2, \ldots\}$.
On convient d'associer à l'occurrence u de profondeur d'abstraction p d'une variable v_i l'entier $i + p$.
Voici quelques exemples :

λ-terme	$v_0\ v_1$	$\lambda\ x.(v_0\ x)$	$\lambda\ x.((\lambda\ y.(y\ v_0))\ x)$	$\lambda\ x.(v_2\ (x\ v_1))$
terme de de Bruijn	0 1	$\lambda(1\ 0)$	$\lambda((\lambda(0\ 2)\ 0)$	$\lambda(3\ (0\ 2))$

On obtient ainsi des *termes de de Bruijn* qui sont des mots sur l'alphabet $\{\lambda, (,)\} \cup \mathbb{N}$ et qu'on peut définir en Caml par la déclaration de type suivante :

```
type terme_de_de_Bruijn =
    | Index of int
    | Appl of terme_de_de_Bruijn * terme_de_de_Bruijn
    | Lambda of terme_de_de_Bruijn ;;
```

Programmation

D'une façon analogue à ce qu'on a fait dans le programme 17 page 27, on peut écrire le programme 18 qui explicite deux fonctions sur les termes de de Bruijn :

— `db_occurrences : string -> terme_de_de_Bruijn -> int list list` attend en arguments une variable v et un terme de de Bruijn t et renvoie la liste des occurrences *libres* de v dans t, (on ne s'inquiète pas des variables *liées* puisqu'ici elles ne sont plus nommées) ;

— `db_libres : terme_de_de_Bruijn -> (int list * string) list` attend en argument un terme de de Bruijn t et renvoie la liste des couples (v, u) où u est une occurrence libre de la variable v.

On notera que ces deux fonctions supposent qu'on dispose de la liste de toutes les variables du monde (je veux parler de l'ensemble $\mathcal{V}$) dans la variable globale CAML nommée `variables`.
On a appelé `p` la profondeur d'abstraction courante dans chacune de ces fonctions.

Programme 18 Occurrences de variables dans un terme de de Bruijn

```
let cons a b = a :: b ;;

let db_occurrences x t =
    let ajoute c = map (cons c)
    in
    let rec aux p = function
        | Index(i) when i < p -> []
        | Index(i) -> if variables.(i-p) = x then [[]] else []
        | Appl(t1,t2) -> (ajoute 1 (aux p t1)) @ (ajoute 2 (aux p t2))
        | Lambda(t) -> ajoute 1 (aux (p + 1) t)
    in
    aux 0 t ;;

let db_libres t =
    let ajoute c = map (function (o,v) -> (c::o,v))
    in
    let rec aux p = function
        | Index(i) -> if i < p then [] else [[],variables.(i-p)]
        | Appl(t1,t2) -> (ajoute 1 (aux p t1)) @ (ajoute 2 (aux p t2))
        | Lambda(t) -> ajoute 1 (aux (p + 1) t)
    in
    aux 0 t ;;
```

Le problème du passage d'une notation à l'autre

Il convient d'écrire les fonctions de conversion des λ-termes en notation de de Bruijn.
Le programme 19 page suivante décrit quelques fonctions utiles dans la suite :

— `vect_index v a` renvoie le premier indice `i` tel que `v.(i) = a`, ou déclenche l'exception `Not_found` ;

— `var_index` et `index_var` sont synonymes, et renvoient le numéro d'une variable dans $\mathcal{V}$;

— `index` est l'analogue de `vect_index` pour les listes ;

— `nth i l` renvoie le `i`-ème élément de la liste `l`.

Programme 19 Quelques fonctions auxiliaires

```
let vect_index v a =
    let i = ref 0
    in
    try
        while v.(!i) <> a do incr i done ;
        !i
    with _ -> raise Not_found ;;

let var_index a = vect_index variables a ;;
let index_var = var_index ;;

let index l a =
    let rec aux i = function
        | [] -> raise Not_found
        | t :: _ when t = a -> i
        | _ :: q -> aux (i+1) q
    in
    aux 0 l ;;

let rec nth k l = match k,l with
    | _,[] -> raise Not_found
    | 0,t::_ -> t
    | _,_::q -> nth (k - 1) q ;;
```

La conversion vers la notation de de Bruijn ne pose pas de réelle difficulté : il aura suffi, de passer en argument à la fonction récursive `aux` un paramètre `liées` où on empile successivement les variables liées par des λ. Dans le cas d'une variable, on regardera si elle fait partie des variables liées et dans ce cas on renverra l'index dans `liées`, sinon, c'est qu'il s'agit d'une variable libre, et on n'a plus qu'à remarquer que la profondeur d'abstraction est égale à la taille de la variable `liées`.

Cela donne la première fonction, `db_of_lt`, du programme 20 page ci-contre.

La conversion inverse pose davantage de problème, puisqu'elle n'est pas univoque : on peut choisir arbitrairement le nom des variables liées. On devra pourtant éviter les collisions dans le cas de λ emboîtés, et on fera attention à ne pas interférer avec les éventuelles variables libres.

La solution proposée dans le programme 20 consiste à calculer tout d'abord la liste des variables libres. La fonction récursive `aux` admet en paramètres la profondeur courante d'abstraction, utile pour discriminer variables libres et liées, une pile des variables liées (qu'on a générées au fur et à mesure), et enfin un entier `k` qui représente l'index dans $\mathcal{V}$ de la prochaine variable muette/liée disponible.

Si donc on rencontre une variable liée, on récupère son nom dans la pile `liées`, si on rencontre une variable libre, on utilise la profondeur courante d'abstraction `p` pour trouver son nom.

Dans le cas d'une λ-expression, il s'agit de trouver un nom pour la nouvelle variable liée. On essaie *a priori* `variables.(k)`, sauf si elle figure dans l'ensemble des variables libres, pour éviter toute collision. Il suffit alors d'appeler récursivement `aux`, en incrémentant la profondeur d'abstraction et `p` et sans oublier d'empiler dans `liées` la nouvelle variable.

Le listing 21 page suivante montre le résultat sur quelques exemples. On observera l'éventuel renommage des seules variables liées.

Programme 20 Les conversions λ-termes–termes de de Bruijn

```
let db_of_lt t =
    let rec aux liées = function
        | Variable(x) -> Index( try index liées x
                                with Not_found -> (list_length liées) + (var_index x)   )
        | Application(t1,t2) -> Appl(aux liées t1 , aux liées t2)
        | Abstraction (x,t) -> Lambda(aux (x :: liées) t)
    in
    aux [] t ;;

let lt_of_db t =
    let libres = map snd (db_libres t)
    in
    let rec aux p liées k = function
        | Index(i) when i < p -> Variable(nth i liées)
        | Index(i) -> Variable(variables.(i - p))
        | Appl(t1,t2) -> Application(aux p liées k t1 , aux p liées k t2)
        | Lambda(t)
            ->  if mem variables.(k) libres then aux p liées (k + 1) (Lambda t)
                else
                    let x = variables.(k)
                    in
                    Abstraction(x,aux (p + 1) (x :: liées) (k + 1) t)
    in
    aux 0 [] 0 t ;;
```

Programme 21 Quelques exemples de conversion

```
#let foo t = print_lambda_terme t ; print_terme_de_de_Bruijn (db_of_lt t) ;
             print_newline() ; lt_of_db (db_of_lt t) ;;
foo : lambda_terme -> lambda_terme = <fun>

#foo exemple1 ;;
x -> w -> (y -> y) w x
(\ (\ (((\ 0) 0) 1)))
- : lambda_terme = a -> b -> (c -> c) b a

#foo exemple2 ;;
x -> y -> (x -> x) y x
(\ (\ (((\ 0) 0) 1)))
- : lambda_terme = a -> b -> (c -> c) b a

#foo exemple3 ;;
x -> y -> x x y
(\ (\ ((1 1) 0)))
- : lambda_terme = a -> b -> a a b

#foo exemple4 ;;
x -> y -> z x y
(\ (\ ((27 1) 0)))
- : lambda_terme = a -> b -> z a b

#foo exemple5 ;;
x -> z (z -> x z)
(\ (26 (\ (1 0))))
- : lambda_terme = a -> z (b -> a b)
```

Une notation plus parlante

Je préfère personnellement pour les λ-termes la notation *à la Caml*, qui consiste à écrire un λ-terme comme $(\lambda\ x.t)$ sous la forme $x \to t$. En outre, on utilisera les règles de parenthésage de Caml, et en particulier, conviendrons de noter simplement `x y z` le terme $(x\ y)\ z$.
Le programme 22 page ci-contre se charge de l'entrée de tels termes : il se compose d'un lexeur et d'un parseur de λ-termes.
Le programme 23 page 34 se charge lui du bel affichage des λ-termes.
En revanche, pour les termes de de Bruijn, je conserve la notation standard, et leur bel affichage est assuré par le programme 24 page 35.

Programme 22 Un lexeur et un parseur pour les λ-termes

```
type lambda_terme =
    | Variable of string
    | Application of lambda_terme * lambda_terme
    | Abstraction of string * lambda_terme ;;

type lexème = ParG | ParD | Flèche | Identificateur of string ;;

let string_of_char_list l =
    let s = make_string (list_length l) `.`
    in
    let rec construit i = function
        | [] -> s
        | t :: q -> s.[i] <- t ; construit (i+1) q
    in
    construit 0 l;;

let mange flot c =
    let rec accumule () =
        try let c,_ = stream_get flot
            in
            if c == ` ` || c == `)` then []
            else ( stream_next flot ; c :: (accumule ()) )
        with Parse_failure -> []
    in
    string_of_char_list (c :: accumule ()) ;;

let rec fin_ligne flot = match flot with
    | [< ''\n` >] -> [< flot >]
    | [< 'c >] -> fin_ligne flot ;;

let rec lexeur flot = match flot with
    | [< '(` ` | `\r` | `\n`) >] -> lexeur flot
    | [< ''%` >] -> lexeur (fin_ligne flot)
    | [< ''(` >] -> ParG :: (lexeur flot)
    | [< '')` >] -> ParD :: (lexeur flot)
    | [< ''-` >]
        -> (   match flot with
                    | [< ''>` >] -> Flèche :: (lexeur flot)
                    | [< >] -> let u = Identificateur(mange flot `-`)
                               in
                               u :: (lexeur flot)  )
    | [< 'c >] -> let u = Identificateur(mange flot c)
                  in
                  u :: (lexeur flot)
    | [< >] -> [] ;;

let rec parse_terme lex_list = let r,_ = parse_T lex_list in r
and parse_T lex_list = match lex_list with
    | (Identificateur v) :: Flèche :: q
        -> let t,q = parse_T q in Abstraction(v,t),q
    | _ -> parse_E lex_list
and parse_E lex_list =
    let f,q = parse_F lex_list in parse_E' f q
and parse_E' f = function
    | (ParD :: _) as lex_list -> f,lex_list
    | [] -> f,[]
    | lex_list -> let f',q = parse_F lex_list in parse_E' (Application(f,f')) q
and parse_F = function
    | (Identificateur v) :: q -> Variable(v),q
    | ParG :: q -> let t,q = parse_T q in match q with
                                            | ParD :: q -> t,q
                                            | _ -> failwith "Erreur de syntaxe" ;;

let parseur chaîne = parse_terme (lexeur (stream_of_string chaîne)) ;;
```

Programme 23 Un bel affichage des λ-termes

```
#open "format" ;;

let print_lambda_terme e =
    let ps = print_string
    and pc = print_char
    and space = print_space
    and open() = open_hovbox 3
    and close = close_box
    in
    let rec print e p =
        open() ;
        if p then pc `(` ;
        ( match e with
            | Variable(v) -> ps v
            | Abstraction(x,u)
                -> ps x ; space() ; ps "->" ; space() ; print u false
            | Application(g,d)
                ->  let pg,pd = match g,d with
                        | Variable(_),Abstraction(_) -> false,true
                        | Variable(_),_ -> false,false
                        | Application(_),Variable(_) -> false,false
                        | Abstraction(_),Variable(_) -> true,false
                        | _ -> true,true
                    in
                    print g pg ; space() ; print d pd
        ) ;
        if p then pc `)` ;
        close()
    in
    print e false ; print_newline () ;;

install_printer "print_lambda_terme" ;;
```

L'α-équivalence

Considérations théoriques

Deux termes t_1 et t_2 seront dits α-équivalents, et on notera $t_1 \equiv_\alpha t_2$ si les termes de de Bruijn associés sont égaux. Intuitivement, c'est dire qu'ils sont égaux à un renommage près des variables liées.
Ainsi, par exemple : $x \to y \to x\ x\ y \equiv_\alpha a \to y \to a\ a\ y$ et $x \to y \to z\ x\ y \equiv_\alpha y \to x \to z\ y\ x$.
Ou encore :

$$(x \to (w \to ((y \to y)w)x)) \equiv_\alpha (x \to (y \to ((x \to x)y)x)),$$

puisque le terme de de Bruijn associé est $(\lambda\ (\lambda\ (((\lambda 0)\ 0)\ 1)))$.

Programmation

Il est facile de programmer l'α-équivalence en notation de de Bruijn : c'est simplement l'égalité ! Pour les λ-termes, le plus simple (et de très loin) consiste à passer d'abord en notation de de Bruijn... tout cela est fait dans le programme 25 page suivante.

Programme 24 Un bel affichage des termes de de Bruijn

```
type terme_de_de_Bruijn =
    | Index of int
    | Appl of terme_de_de_Bruijn * terme_de_de_Bruijn
    | Lambda of terme_de_de_Bruijn ;;

let print_terme_de_de_Bruijn e =
    let ps = print_string
    and pi = print_int
    and pc = print_char
    and space = print_space
    and open() = open_hovbox 3
    and close = close_box
    in
    let rec print e =
        open() ;
        ( match e with
            | Index(i) -> pi i
            | Lambda(u)
                -> ps "(\\" ; space() ; print u ; pc `)`
            | Appl(g,d)
                ->  pc `(` ; print g ; space() ; print d ; pc `)`
        ) ;
        close()
    in
    print e ;;

install_printer "print_terme_de_de_Bruijn" ;;
```

Programme 25 L'α-équivalence des termes

```
let rec db_alpha_équivalence t1 t2 = match t1,t2 with
    | Index(i1),Index(i2) -> i1 = i2
    | Appl(t11,t12),Appl(t21,t22) -> (db_alpha_équivalence t11 t21)
                                        && (db_alpha_équivalence t12 t22)
    | Lambda(t1),Lambda(t2) -> db_alpha_équivalence t1 t2
    | _ -> false ;;

let lt_alpha_équivalence t1 t2 =
    db_alpha_équivalence (db_of_lt t1) (db_of_lt t2) ;;
```

La β-réduction

Considérations théoriques

Pour l'instant, notre théorie du λ-calcul ne sait pas ... calculer. C'est le but de la β-réduction de nous faire franchir ce saut conceptuel, et on verra que nous pourrons utiliser plusieurs stratégies de calcul.
L'idée de base est la suivante : si on *applique* un λ-terme de la forme $(\lambda\ x.u)$ à un λ-terme t, c'est-à-dire qu'on considère le λ-terme $(\lambda\ x.u)\ t$, on aimerait que le résultat du calcul soit le terme u où toute occurrence de x est remplacée par t. Bien sûr, tout n'est pas si simple...
Si par exemple on veut calculer `(x -> y -> x y) y` on veut obtenir `z -> y z` et non pas `y -> y y`. Ou encore on veut que le calcul de `(x -> y -> z x y) (z -> y z)` aboutisse à `a -> z (b -> y b) a`, ou bien sûr tout autre λ-terme qui lui soit α-équivalent. C'est dire qu'une simple substitution ne fait pas l'affaire, et qu'on se doit d'une définition plus précise.
Soit $(\lambda\ x'.u')$ un terme α-équivalent à $(\lambda\ x.u)$ tel que x' ni aucune variable liée de u' ne soit libre dans t, et soit t' un terme α-équivalent à t tel qu'aucune variable liée de t' ne soit libre dans u. (Notons que cela est possible car les variables libres de u sont celles de u', et les variables libres de t celles de t').
Soit alors U le λ-terme obtenu en substituant dans u' à toute occurrence de x' liée au premier λ dans $(\lambda\ x'.u')$ le terme t'. On dit qu'une β-réduction fait passer de $(\lambda\ x.u)\ t$ à U, et on notera $(\lambda\ x.u)\ t \models_\beta U$.
Ces précautions étant prises, on vérifie que la β-réduction est compatible avec l'α-équivalence, c'est-à-dire que si $t_1 \models_\beta t_2$ et si $t_1 \equiv_\alpha t'_1$ et $t_2 \equiv_\alpha t'_2$ on a encore $t'_1 \models_\beta t'_2$.
Cela revient à dire qu'on pourrait écrire plus simplement la β-réduction en notation de de Bruijn. Ce sera justement ainsi qu'on l'implémentera en Caml plus bas.
Redonnons au passage les exemples ci-dessus en notation de de Bruijn :

$$\begin{array}{rcl} ((\lambda\ (\lambda\ (1\ 0)))\ 24) & \models_\beta & \lambda\ (25\ 0)) \\ ((\lambda\ (\lambda\ ((27\ 1)\ 0)))\ (\lambda\ (25\ 0))) & \models_\beta & (\lambda\ ((26\ (\lambda\ (26\ 0)))\ 0)) \end{array}$$

(j'ai numéroté les variables $a, \ldots, z$ dans l'ordre alphabétique ($a = 0, \ldots, z = 25$).
Une fois donnée cette définition de la β-réduction pour les termes de la forme $(\lambda\ x.u)\ t$, on généralise en écrivant que :

- si $t \models_\beta t'$ alors $(t\ u) \models_\beta (t'\ u)$ et $(u\ t) \models_\beta (u\ t')$;
- si $t \models_\beta t'$ alors $(\lambda\ x.t) \models_\beta (\lambda\ x.t')$.

Cette définition n'est pas un algorithme : il n'y a plus déterminisme, et rien n'indique par quelle β-réduction commencer si on a le choix.
Par exemple on a tout aussi bien

$$\begin{array}{rcl} (\lambda\ x.(x\ x))\ ((\lambda\ y.y)\ z) & \models_\beta & ((\lambda\ y.y)\ z)\ ((\lambda\ y.y)\ z) \\ (\lambda\ x.(x\ x))\ ((\lambda\ y.y)\ z) & \models_\beta & (\lambda\ x.(x\ x))\ z \end{array}$$

Notons toutefois que chacun des deux termes obtenus se β-réduit en $(z\ z)$, ce qui n'est pas étonnant, comme nous allons bientôt le voir.

On note ensuite $\models^\star_\beta$ la *clôture transitive* de la relation $\models_\beta$.
Elle est définie ainsi : pour tous termes t_1 et t_2 on a $t_1 \models^\star_\beta t_2$ si $t_1 = t_2$ ou si $t_1 \models_\beta t_2$ ou encore s'il existe un nombre fini de β-réductions faisant passer de t_1 à t_2 :

$$\exists (u_1, \ldots, u_p) \in \Lambda^p, t_1 \models_\beta u_1 \models_\beta \cdots \models_\beta u_p \models_\beta t_2.$$

On démontre alors le

Théorème 12 (Church–Rosser) *La relation $\models_\beta$ est confluente, c'est-à-dire que si $t \models^\star_\beta u$ et $t \models^\star_\beta v$ alors il existe un λ-terme t' tel que $u \models^\star_\beta t'$ et $v \models^\star_\beta t'$.*

On définit un terme *normal* comme étant un terme sur lequel on ne peut procéder à aucune β-réduction : c'est en quelque sorte un résultat qu'on ne peut simplifier (l'image est adéquate puisque la β-réduction contracte les λ-termes en profondeur d'abstraction).
Un terme t est *normalisable* si il existe un terme normal n tel que $t \models^\star_\beta n$.
Notons qu'en conséquence immédiate du théorème de Church–Rosser, si t est normalisable, il existe un **unique** terme normal n tel que $t \models^\star_\beta n$.
Calculer, en λ-calcul, c'est donc, quand c'est possible déterminer ce terme normal n à partir d'un terme t.
Plusieurs remarques s'imposent :

— tout terme n'est pas normalisable : c'est le cas typique du célèbre $\omega = (\lambda\, x.(x\, x))\ (\lambda\, x.(x\, x))$ qui vérifie $\omega \models_\beta \omega$, et qui mène donc à un calcul infini ;

— on montre que le problème de savoir si un terme est normalisable est indécidable ;

— même si un terme t est normalisable, il peut exister une chaîne infinie de β-réductions $t \models_\beta t_1 \models_\beta t_2 \models_\beta \cdots \models_\beta t_k \models_\beta \cdots$ (on dit que t n'est pas fortement normalisable).

La dernière de ces remarques montre que la *stratégie* de normalisation, c'est-à-dire du choix de la β-réduction à appliquer à chaque étape, est cruciale : une mauvaise stratégie peut empêcher d'achever le calcul.
Par exemple considérons le terme $t = ((\lambda\, x.\lambda\, y.x)\ x)\ \omega$, qui est une abréviation de

$$((\lambda\, x.\lambda\, y.x)\ x)\ ((\lambda\, x.(x\, x))\ (\lambda\, x.(x\, x))).$$

Si on commence par β-réduire à gauche, on obtient successivement :

$$t \models_\beta ((\lambda\, y.x)\ \omega) \models_\beta x,$$

alors que si on commence par s'attaquer à la β-réduction de ω on tombe dans une boucle infinie...
Heureusement, on montre qu'il existe (au moins) une *bonne stratégie* (c'est-à-dire qui conduit pour tout terme normalisable à un terme normal en un nombre fini d'étapes) qui en outre est des plus simples.

Théorème 13 *Si t est un terme normalisable, la stratégie consistant à opérer à chaque étape la première β-réduction possible dans la lecture naturelle (de gauche à droite) du λ-terme en cours conduit à un terme normal en un nombre fini d'étapes.*

Programmation

Le programme 26 s'intéresse au cœur du problème de la β-réduction puisqu'il se limite au cas d'un terme de de Bruijn de la forme $(\lambda\ u)\ t$.
La fonction récursive auxiliaire `aux` utilisera en paramètre la profondeur courante d'abstraction, pour discriminer en particulier (en ligne 14) la variable à laquelle on substituera le terme t. On conservera telles quelles (ligne 15) les autres variables liées, mais on devra faire attention que la β-réduction supprime un niveau d'abstraction et que cela induit une décrémentation des index des variables libres du terme u (ligne 16).
La substitution elle-même est du ressort de la fonction `décalage`, en lignes 2–9, qui doit réaliser un décalage des index des variables liées de t cette fois. On utilisera là encore une fonction auxiliaire récursive `aux_décale` qui utilise un paramètre `p` de profondeur d'abstraction dans t pour discriminer les variables libres des variables liées : les variables liées ne sont pas modifiées (ligne 6), les variables libres subissent le décalage (ligne 7).

Programme 26 Le cœur de la β-réduction

```
 1 let bêta (Lambda u) t =
 2     let décalage d t =
 3         let rec aux_décale p = function
 4             | Appl(t1,t2) -> Appl(aux_décale p t1,aux_décale p t2)
 5             | Lambda(t) -> Lambda(aux_décale (p + 1) t)
 6             | Index(i) when i < p -> Index(i)
 7             | Index(i) -> Index(i + d)
 8         in
 9         aux_décale 0 t
10     in
11     let rec aux p = function
12         | Appl(u1,u2) -> Appl(aux p u1,aux p u2)
13         | Lambda(v) -> Lambda(aux (p + 1) v)
14         | Index(i) when i = p -> décalage p t
15         | Index(i) when i < p -> Index(i)
16         | Index(i) -> Index(i - 1)
17     in
18     aux 0 u ;;
```

Je laisse au lecteur sceptique à propos de la notation de de Bruijn le soin d'écrire la même fonction en travaillant directement sur les λ-termes, je ne vois pas de meilleure façon de le convaincre...
Il n'y a plus qu'à se rappeler quelle stratégie nous voulons utiliser pour la normalisation éventuelle des λ-termes : nous avons dit qu'on commencerait par la β-réduction la plus à gauche dans l'ordre naturel de lecture du λ-terme. La fonction `bêta_un` du programme 27 page suivante se charge de l'application de cette stratégie : elle effectue la première β-réduction possible, ou déclenche l'exception `Stop` si aucune β-réduction ne s'applique. Il n'y a plus qu'à profiter de la gestion des exceptions de CAML pour écrire la β-réduction complète, en n'oubliant pas que la fonction écrite boucle sur des termes non normalisables.
Le listing 28 page ci-contre montre le résultat sur un premier exemple banal, puis ce qui arrive si on tente la β-réduction du terme $\omega = (\lambda\ x.(x\ x))(\lambda\ x.(x\ x))$ qui n'est pas normalisable : le programme tombe dans une boucle infinie, et d'ailleurs une première β-réduction renvoie ω tel quel... on avait déjà dit que $\omega \models_\beta \omega$.
Le dernier exemple est celui que nous avions introduit quand nous avons discuté du choix de la

Programme 27 La β-réduction

```
exception Stop ;;

let rec bêta_un = function
    | Index(_) -> raise Stop
    | Lambda(t) -> Lambda(bêta_un t)
    | Appl(Lambda(u),t) -> bêta (Lambda u) t
    | Appl(t1,t2) -> try Appl(bêta_un t1,t2) with Stop -> Appl(t1,bêta_un t2) ;;

let rec bêta_réduction t =
    try bêta_réduction (bêta_un t)
    with Stop -> t ;;

let db_bêta_réduction = bêta_réduction
and lt_bêta_réduction t = lt_of_db (bêta_réduction (db_of_lt t)) ;;
```

stratégie de β-réduction.

Programme 28 Exemples de β-réductions

```
#let e = parseur "(x -> x x) ((y -> x y) a)" ;;
e : lambda_terme = (x -> x x) ((y -> x y) a)

#lt_bêta_réduction e ;;
- : lambda_terme = (x a) (x a)

#let omega = parseur "(x -> x x) (x -> x x)" ;;
omega : lambda_terme = (x -> x x) (x -> x x)

#lt_bêta_réduction omega ;;
Interruption.

#(** d'ailleurs : **)
let omega' = db_of_lt omega ;;
omega' : terme_de_de_Bruijn = ((\ (0 0)) (\ (0 0)))

#bêta_un omega' ;;
- : terme_de_de_Bruijn = ((\ (0 0)) (\ (0 0)))

#let t = parseur "(x -> y -> x) x ((x -> x x) (x -> x x))" ;;
t : lambda_terme = ((x -> y -> x) x) ((x -> x x) (x -> x x))

#lt_bêta_réduction t ;;
- : lambda_terme = x
```