La lettre de Caml
numéro 7

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

automne-hiver 1997

Édito

Nous décrivons dans ce nouveau numéro de La Lettre de Caml *une structure de dictionnaire pour des chaînes de caractères, fondée sur l'utilisation d'arbres binaires compactés. On écrit les trois fonctions de base (recherche, insertion et suppression), et on explique comment créer une petite bibliothèque* Caml*, à l'interface très réduite, qui permette d'utiliser facilement cette nouvelle structure de données.*
Dans une deuxième partie, on montre comment construire une représentation des séries formelles. On programme les opérations habituelles, y compris la composition de deux séries formelles. On en profite pour rappeler comment Caml *nous permet de définir de nouveaux opérateurs infixes.*
Notre représentation des séries formelles utilise la bibliothèque `num`*, et on obtient donc des résultats formels tout à fait intéressants. Je laisse au lecteur le soin de comparer les performances des algorithmes présentés ici à celles des algorithmes implémentés dans Maple.*

Table des matières

Liste des programmes

Table des figures

Arbres compactés de recherche de chaînes de caractères

Présentation du problème

On cherche à implémenter une structure de dictionnaire sur les chaînes de caractères, c'est-à-dire une structure de données qui permette les trois opérations fondamentales suivantes : la recherche d'une chaîne, l'insertion et la suppression d'une chaîne.
Nous choisissons ici une représentation arborescente, à base d'arbres binaires, ce qui conduit naturellement à représenter une chaîne par la suite de ses bits (dans le codage ASCII par exemple). Cela posé, on peut bien entendu construire un arbre de recherche guidé sur les bits successifs de la chaîne cherchée : à la lecture d'un bit 0 on descend à gauche, à la lecture d'un bit 1 on descend à droite. Plusieurs problèmes apparaissent : l'arbre obtenu a une profondeur linéaire en la taille de la plus longue chaîne ; il faut un moyen d'indiquer que deux chaînes dont l'une est préfixe de l'autre figurent dans le même arbre...
Nous définirons donc une structure plus compacte d'arbre binaire de recherche mieux adaptée à notre problème.

Écriture binaires d'une chaîne

Commençons par le début, et par le plus simple : l'écriture binaire des chaînes de caractères.
Pour des raisons qui apparaîtront plus loin, nous convenons d'ajouter un caractère ASCII(0) à la fin de chaque chaîne, ce qui nous conduit à écrire les fonctions du programme 1.

Programme 1 écriture binaire des chaînes de caractères

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

let bits_of_char c =
    let c' = int_of_char c
    in
    it_list (fun l i -> (c' lsr i) land 1 :: l)
        [] (intervalle 0 7) ;;

let bits_of_string s =
    list_it
        (fun i l -> bits_of_char s.[i] @ l)
        (intervalle 0 (string_length s - 1))
        [0;0;0;0;0;0;0;0] ;;
```

La compaction des arbres

Dans la suite, nous travaillerons — pour la description des algorithmes — sur des suites de 0 et 1, terminées par la suite $0^8 = (0,0,0,0,0,0,0,0)$ que nous noterons indifféremment 0^8 ou ω. Cette convention, qui suppose bien sûr que le caractère ASCII(0) ne figure pas dans les chaînes utilisées, permet de s'assurer qu'aucun mot n'est préfixe d'un autre, ce qui résout un premier problème à peu de frais.
Dessinons l'arbre correspondant aux quatre chaînes $A = \omega$, $B = 00001001\omega$, $C = 00000111\omega$ et $D = 00001010\omega$. On obtient l'arbre de la figure 1 page suivante.
Rappelons que nous sommes convenus de descendre à gauche sur le bit nul.

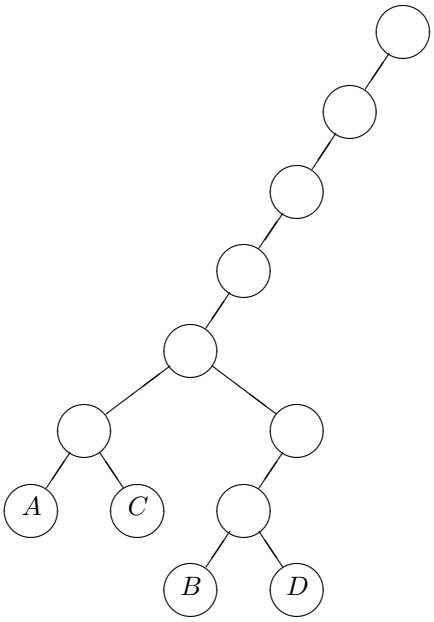

FIG. 1: l'arbre non compacté pour les mots A, B, C et D

Les quatre premiers bits ne discriminent pas nos quatre mots : il serait donc plus économique de ne pas en tenir compte. De même, ayant lu le préfixe 00001, les deux seuls mots encore concernés, à savoir B et D, ne sont pas discriminés par le bit suivant, qui est nul pour tous les deux.
On arrive finalement à l'arbre compacté de la figure 2.

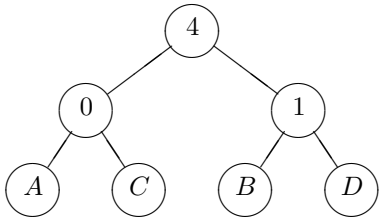

FIG. 2: l'arbre compacté pour les mots A, B, C et D

Si par exemple on recherche le mot $B = 00001001\omega$ dans cet arbre, on commence par biffer les quatre premiers bits, obtenant 1001ω. Le premier bit est égal à 1 : on descend à droite. On biffe alors le bit suivant, obtenant 01ω, et on termine la descente à gauche : on a bien trouvé l'emplacement du mot B.
Le type CAML utilisé pour nos arbres de recherche sera donc :

```
type arbre = Feuille of string | Nœud of int * arbre * arbre ;;
```

Quelques remarques générales

Observons tout d'abord qu'un arbre de recherche compacté est univoquement déterminé par la donnée de l'ensemble des chaînes qu'il représente : l'ordre d'insertion de ces chaînes ne change pas l'arbre final.

En outre le nombre de feuilles de l'arbre est égal au nombre N de ces chaînes, et donc, puisqu'il est compact, le nombre de ses nœuds internes est égal à $N-1$. La profondeur p de l'arbre vérifie alors (comme pour tout arbre binaire) : $1 + \lfloor \lg(N-1) \rfloor \leqslant p \leqslant N-1$.

La recherche

Nous sommes maintenant en mesure de programmer la recherche d'une chaîne dans un arbre.
Nous commençons par une fonction `skip : int -> 'a list -> 'a list` telle que `skip k l` renvoie la liste `l` privée de ses `k` premiers éléments. Comme cette fonction ne procède à aucune vérification, elle est susceptible de déclencher une exception (en l'occurrence `Failure "tl"`).
Notre fonction de recherche, `recherche : arbre -> string -> bool`, renverra `true` ou `false` selon que la chaîne figure ou non dans l'arbre. Elle procède comme on peut s'y attendre, en biffant le nombre requis de bits avant de diriger la descente dans l'arbre selon la valeur du bit suivant. On utilise un rattrapage d'exception pour le cas où la liste des bits d'une chaîne (absente de l'arbre) serait *épuisée* en cours d'algorithme.
Tout cela fait l'objet du programme 2.

Programme 2 la recherche dans les arbres compactés

```
let rec skip k l =
    if k = 0 then l
    else skip (k - 1) (tl l) ;;

let recherche a s =
    let rec aux l = function
        | Feuille s' -> s = s'
        | Nœud(k,g,d)
            ->  try
                    let t :: q = skip k l
                    in
                    aux q (if t = 0 then g else d)
                with _ -> false
    in
    aux (bits_of_string s) a ;;
```

La suppression

La suppression d'une chaîne est à peine plus compliquée. Signalons toutefois que notre structure doit toujours contenir au moins une chaîne — nous commencerons toujours par un arbre `Feuille ""` contenant le seul mot vide — et que nous devrons donc déclencher une exception si on tentait de supprimer la dernière chaîne d'un arbre.
Nous choisissons d'autre part d'accepter la suppression d'une chaîne absente : nous entendons par là que nous renvoyons l'arbre inchangé si on demande d'en ôter une chaîne qui n'y figure pas.
Le programme 3 page ci-contre montre comment on peut programmer cette suppression. Qu'il nous suffise de dire qu'il faut avant de descendre, que ce soit à gauche ou à droite, regarder si la chaîne cible n'est pas la feuille sous-arbre gauche ou droit. Si tel est le cas, on remplace le nœud père par celui des fils qui reste, mais en procédant à un décalage du nombre de bits à biffer.
Un dessin permet de bien comprendre : construisons, dans la figure 3 page suivante, l'arbre compacté correspondant aux mots $A = \omega$, $B = 00001001\omega$, $C = 00000111\omega$, $D = 00001010000\omega$

Programme 3 la suppression dans les arbres compactés

```
let rec suppression a s =
    let rec aux l = function
        | Feuille s' when s <> s' -> Feuille s'
        | Feuille _ -> failwith "Arbre vide !"
        | Nœud(k,Feuille s',d) when s' = s
            ->  (   match d with
                        | Feuille _ -> d
                        | Nœud(k',g',d') -> Nœud(k + k' + 1,g',d')  )
        | Nœud(k,g,Feuille s') when s' = s
            ->  (   match g with
                        | Feuille _ -> g
                        | Nœud(k',g',d') -> Nœud(k + k' + 1,g',d')  )
        | Nœud(k,g,d)
            -> try
                    match skip k l with
                        t :: q ->   if t = 0 then Nœud(k,(aux q g),d)
                                    else Nœud(k,g,(aux q d))
                with _ -> Nœud(k,g,d)
    in
    aux (bits_of_string s) a ;;
```

et $E = 00001010001\omega$.

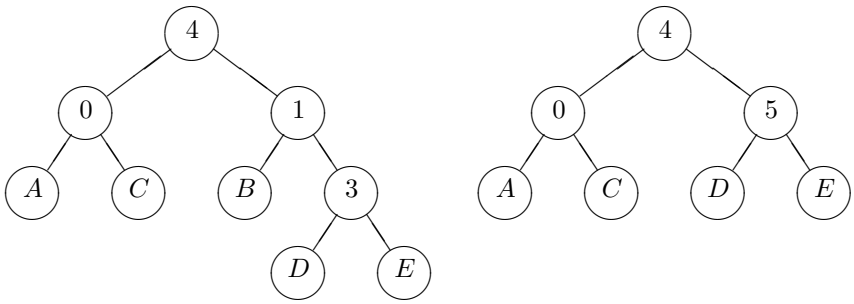

FIG. 3: l'arbre pour les mots A, B, C, D et E, avant et après suppression du mot B

Si nous supprimons le mot B, il faut s'en apercevoir quand on considère le nœud étiquetté par 1. On remplace alors ce nœud par `Nœud(5,D,E)` puisque les deux mots D et E ont à ce niveau un préfixe commun de $1+1+3=5$ bits (ne pas oublier le bit 1 qui a permis de descendre à droite).

L'insertion

Pour réaliser l'insertion d'une nouvelle chaîne, il faut tout d'abord descendre dans l'arbre jusqu'à la feuille qui a le plus long préfixe commun avec notre chaîne, afin de déterminer l'emplacement de l'insertion effective.
On écrit (cf. le programme 4 page suivante) une fonction `trouve_place : string -> arbre -> string` qui ressemble comme deux gouttes d'eau à la recherche, et qui renvoie la chaîne de la feuille à laquelle on accède dans la recherche d'une chaîne à insérer. Dans le cas où la chaîne à insérer est épuisée, on décide de descendre systématiquement à gauche, de toutes façons cela n'a plus

d'importance, la discrimination sera faite avant... c'est ce qui explique le choix qui est fait dans le rattrapage d'exception.
La petite fonction `discrimine : 'a list -> 'a list -> int` sera appelée avec un troisième argument nul. Elle renvoie un triplet (i, x, y) où i est le nombre d'éléments communs en tête des deux listes arguments, x et y étant les premiers éléments distincts de ces listes.

Programme 4 quelques fonctions auxiliaires de l'insertion

```
let trouve_place s a =
    let rec descend = function
        | Feuille s' -> s'
        | Nœud(_,g,_) -> descend g
    in
    let rec aux l = function
        | Feuille s' -> s'
        | Nœud(k,g,d)
            ->  try
                    match skip k l with
                        t :: q -> aux q (if t = 0 then g else d)
                with _ -> descend g
    in
    aux (bits_of_string s) a ;;

let rec discrimine l1 l2 i =
    match l1,l2 with
        t1 :: q1,t2 :: q2
            ->  if t1 <> t2 then i,t1,t2
                else discrimine q1 q2 (i + 1) ;;
```

L'insertion se résume simplement, arrivé à un `Nœud(k,g,d)` où `k` est plus grand que ce qui reste du préfixe commun à la chaîne à insérer et à la chaîne qu'a renvoyée `trouve_place`, à un dédoublement de ce nœud. Quand en revanche on a `k` plus petit, il suffit de descendre dans l'arbre, du bon côté bien sûr.

Programme 5 l'insertion dans un arbre compacté de recherche

```
let insertion a s =
    let s' = trouve_place s a
    in
    if s = s' then a
    else
    let l,l' = (bits_of_string s),(bits_of_string s')
    in
    let i,c,_ = discrimine l l' 0
    in
    let rec aux j l = function
        | Feuille s' as a
            ->  if c = 0 then Nœud(j,Feuille s,a)
                else Nœud(j,a,Feuille s)
        | Nœud(k,g,d) as a
            ->  if j > k then
                    let t :: q = skip k l
                    in
                    if t = 0 then Nœud(k,(aux (j - k - 1) q g),d)
                    else Nœud(k,g,(aux (j - k - 1) q d))
                else if j < k then
                    if c = 0 then Nœud(j,Feuille s,Nœud(k-j-1,g,d))
                    else Nœud(j,Nœud(k-j-1,g,d),Feuille s)
                else (* j = k *)
                    failwith "Erreur irrécupérable"
    in
    aux i l a ;;
```

Une petite bibliothèque

On écrit tout d'abord le petit fichier d'interface : c'est le programme 6.

Programme 6 le fichier d'interface `arbres.compacts.mli`

```
1 type arbre_de_recherche =
2     {
3         chercher : string -> bool ;
4         insérer : string -> unit ;
5         supprimer : string -> unit
6     } ;;
7
8 value nouvel_arbre : unit -> arbre_de_recherche ;;
```

On trouvera, dans les programmes 7 page ci-contre, 8 page 12 et 9 page 13, le listing complet du fichier `.ml` qui définit notre bibliothèque.
Voici les invocations qui permettent alors de compiler l'interface puis la bibliothèque elle-même.

```
compile "arbres.compacts.mli" ;;
#open "arbres.compacts" ;;
compile "arbres.compacts.ml" ;;
```

Ont été ainsi créés les fichiers `arbres.compacts.zi` puis `arbres.compacts.zo`.
Voici pour conclure le début de ce que pourrait être une session qui utilise cette bibliothèque :

```
##open "arbres.compacts" ;;
#load_object "arbres.compacts" ;;
- : unit = ()
#nouvel_arbre ;;
- : unit -> arbre_de_recherche = <fun>
#let a = nouvel_arbre () ;;
a : arbre_de_recherche =
 {chercher = <fun>; insérer = <fun>; supprimer = <fun>}
#do_list a.insérer [ "CA" ; "CAML" ; "CAMELIA" ; "CAMEL" ; "CHAMEAU" ; "CHAMELLE" ; "CHAMELIER" ] ;;
- : unit = ()
```

Programme 7 le début du fichier `.ml` complet

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

let bits_of_char c =
    let c' = int_of_char c
    in
    it_list (fun l i -> (c' lsr i) land 1 :: l)
        [] (intervalle 0 7) ;;

let bits_of_string s =
    list_it
        (fun i l -> bits_of_char s.[i] @ l)
        (intervalle 0 (string_length s - 1))
        [0;0;0;0;0;0;0;0] ;;

type arbre = Feuille of string | Nœud of int * arbre * arbre ;;

let rec skip k l =
    if k = 0 then l
    else skip (k - 1) (tl l) ;;

let recherche a s =
    let rec aux l = function
        | Feuille s' -> s = s'
        | Nœud(k,g,d)
            ->  try
                    let t :: q = skip k l
                    in
                    aux q (if t = 0 then g else d)
                with _ -> false
    in
    aux (bits_of_string s) a ;;

let rec suppression a s =
    let rec aux l = function
        | Feuille s' when s <> s' -> Feuille s'
        | Feuille _ -> failwith "Arbre vide !"
        | Nœud(k,Feuille s',d) when s' = s
            ->  (   match d with
                        | Feuille _ -> d
                        | Nœud(k',g',d') -> Nœud(k + k' + 1,g',d')  )
        | Nœud(k,g,Feuille s') when s' = s
            ->  (   match g with
                        | Feuille _ -> g
                        | Nœud(k',g',d') -> Nœud(k + k' + 1,g',d')  )
        | Nœud(k,g,d)
            -> try
                    match skip k l with
                        t :: q ->   if t = 0 then Nœud(k,(aux q g),d)
                                    else Nœud(k,g,(aux q d))
                with _ -> Nœud(k,g,d)
    in
    aux (bits_of_string s) a ;;
```

Programme 8 la deuxième partie du fichier `.ml` complet

```
55 let trouve_place s a =
56     let rec descend = function
57         | Feuille s' -> s'
58         | Nœud(_,g,_) -> descend g
59     in
60     let rec aux l = function
61         | Feuille s' -> s'
62         | Nœud(k,g,d)
63             ->  try
64                     match skip k l with
65                         t :: q -> aux q (if t = 0 then g else d)
66                 with _ -> descend g
67     in
68     aux (bits_of_string s) a ;;
69
70 let rec discrimine l1 l2 i =
71     match l1,l2 with
72         t1 :: q1,t2 :: q2
73             ->  if t1 <> t2 then i,t1,t2
74                 else discrimine q1 q2 (i + 1) ;;
75
76 let insertion a s =
77     let s' = trouve_place s a
78     in
79     if s = s' then a
80     else
81     let l,l' = (bits_of_string s),(bits_of_string s')
82     in
83     let i,c,_ = discrimine l l' 0
84     in
85     let rec aux j l = function
86         | Feuille s' as a
87             ->  if c = 0 then Nœud(j,Feuille s,a)
88                 else Nœud(j,a,Feuille s)
89         | Nœud(k,g,d) as a
90             ->  if j > k then
91                     let t :: q = skip k l
92                     in
93                     if t = 0 then Nœud(k,(aux (j - k - 1) q g),d)
94                     else Nœud(k,g,(aux (j - k - 1) q d))
95                 else if j < k then
96                     if c = 0 then Nœud(j,Feuille s,Nœud(k-j-1,g,d))
97                     else Nœud(j,Nœud(k-j-1,g,d),Feuille s)
98                 else (* j = k *)
99                     failwith "Erreur irrécupérable"
100    in
101    aux i l a ;;
```

Programme 9 la fin du fichier `.ml` complet

```
102 (*****************************************
103  ce type est défini dans le fichier .mli,
104  sa définition ne doit pas être répétée !
105
106 type arbre_de_recherche =
107     {
108         chercher : string -> bool ;
109         insérer : string -> unit ;
110         supprimer : string -> unit } ;;
111
112 *****************************************)
113
114 let nouvel_arbre () =
115     let a = ref (Feuille "")
116     in
117     {
118         chercher = (function s -> recherche !a s) ;
119         insérer = (function s -> a := insertion !a s) ;
120         supprimer = (function s -> a := suppression !a s)
121     } ;;
```

Séries formelles

Quelques rappels mathématiques

Les quatre opérations

Nous considérerons ici des séries formelles à coefficients dans $\mathbb{Q}$; leur ensemble est traditionnellement noté $\mathbb{Q}[[X]]$.
Une série formelle est simplement une application de $\mathbb{N}$ dans $\mathbb{Q}$. Nous noterons une série formelle de la façon suivante : $A(X) = \sum_{k=0}^{+\infty} a_k X^k$, pour signifier que A est l'application qui à tout entier k associe le rationnel a_k.
Pour toute série formelle non nulle A on appelle *valuation* de A et on note $v(A)$ le plus petit entier k tel que $a_k \neq 0$.
On identifie habituellement tout rationnel r avec la série formelle A définie par $a_0 = r$ et $a_k = 0$ si $k \geqslant 1$. Plus généralement on identifie un polynôme avec une série formelle.
On définit l'addition de deux séries formelles et le produit d'une série formelle par une constante rationnelle en posant :

$$\sum_{k=0}^{+\infty} a_k X^k + \sum_{k=0}^{+\infty} b_k X^k = \sum_{k=0}^{+\infty} (a_k + b_k) X^k; \qquad \lambda . \sum_{k=0}^{+\infty} a_k X^k = \sum_{k=0}^{+\infty} \lambda a_k X^k.$$

On aura bien compris qu'on ne se pose pas le moindre problème de convergence de série...
Le produit de deux séries formelles se définit grâce au produit de Cauchy : en conservant les conventions de notation précédentes, le produit de deux séries formelles A et B est la série formelle C définie par la relation

$$\forall n \in \mathbb{N}, \quad c_n = \sum_{k=0}^{n} a_k b_{n-k}.$$

Notons que ce produit est commutatif. Notons aussi que *modulo* l'identification de $\mathbb{Q}$ à une partie de $\mathbb{Q}[[X]]$ le produit d'une constante par une série formelle n'est qu'un cas particulier du produit général de deux séries formelles.
On montre alors sans difficulté (nous le verrons d'ailleurs lus loin) que si une série formelle A est de valuation nulle (c'est-à-dire si $a_0 \neq 0$) il existe une série formelle unique B — qu'on appelle l'inverse de A — telle que $A \times B$ est la série formelle qu'on a identifiée à la constante 1.
L'espace $\mathbb{Q}[[X]]$ muni des opérations précédentes a naturellement une structure de $\mathbb{Q}$-algèbre commutative et associative.
Plus délicate à définir proprement est la composition de deux séries formelles.

La composition des séries formelles

On vérifie facilement que pour deux séries formelles A et B quelconques on a toujours $v(A \times B) = v(A) + v(B)$, et, par conséquent, que $v(A^n) = nv(A)$ pour tout entier $n \geqslant 1$.
On définit, pour tout entier naturel k, une opération de *troncature au rang k*, notée τ_k, application linéaire de $\mathbb{Q}[[X]]$ dans l'espace $\mathbb{Q}[X]$ des polynômes à coefficients rationnels, en décidant que si $B = \tau_k(A) = \sum_{j=1}^{k} b_j X^j$, avec $b_i = a_i$ quand $i \leqslant k$. On a bien sûr $\tau_k(A) = 0$ si $v(A) \geqslant k + 1$.

En outre on vérifie aisément que pour tout entier k, et toutes séries formelles A et B, on a $\tau_k(A \times B) = \tau_k(A) \times \tau_k(B)$.

Soit alors B une série formelle de valuation au moins égale à 1, de sorte que $v(B^n) \geqslant n$ pour tout $n \in \mathbb{N}$ et A une série formelle quelconque. On veut définir la composée C, notée $A(B)$ ou $A \circ B$ voire $A(B(X))$, des deux séries formelles.

Pour cela il suffit de définir pour tout entier $n \geqslant 0$ le polynôme $C_n = \tau_n(C)$, pourvu qu'on puisse assurer pour $k < n$ que $\tau_k(C_n) = C_k$.

Comme $v(B) \geqslant 1$ par hypothèse, on vérifie facilement qu'il suffit de poser $C_n = \tau_n(\tau_n(A) \circ \tau_n(B))$, où bien sûr $\circ$ désigne la composition des polynômes. Nous verrons plus bas une formule explicite pour les coefficients de la composée de deux séries formelles.

Une représentation des séries formelles en Caml

On reprend ici les définitions de Pierre Weis dans *La lettre de Caml* numéro 2. Le lecteur est invité à relire attentivement l'article de Pierre qui explique le fonctionnement de l'évaluation paresseuse.

Le type correspondant est défini ainsi :

```
type 'a glaçon =
    | Gelé of unit -> 'a
    | Connu of 'a ;;

type série_formelle = { Constante : num ; mutable Reste : série_formelle glaçon } ;;
```

Commençons, dans le programme 10, par définir quelques constantes du type `num` qui seront utiles dans la suite.

Programme 10 quelques définitions de constantes

```
#open "num" ;;

let moins_un = num_of_int (-1) and moins_deux = num_of_int (-2) ;;

let [zéro;un;deux;trois;quatre;cinq;six;sept;huit;neuf;dix]
    = map num_of_int [0;1;2;3;4;5;6;7;8;9;10] ;;

let [moins_un;moins_deux;moins_trois;moins_quatre;moins_cinq;
     moins_six;moins_sept;moins_huit;moins_neuf;moins_dix]
    = map num_of_int [-1;-2;-3;-4;-5;-6;-7;-8;-9;-10] ;;

let un_demi = un // deux and moins_un_demi = moins_un // deux ;;
```

On écrit facilement les fonctions des programmes 11 page suivante et 12 page 17. Les commentaires inclus dans les sources décrivent leur usage.

Rappelons simplement que la série génératrice exponentielle associée à une suite (a_n) est la série formelle $\displaystyle\sum_{n=0}^{+\infty} \frac{a_n}{n!} X^n$.

Programme 11 les fonctions de base sur les séries formelles (début)

```
let reste_SF s = match s.Reste with
    | Gelé r -> let a = r() in s.Reste <- Connu a ; a
    | Connu a -> a ;;


(*----------------------------------------------------------------------------
  crée_SF_de : (num -> num) -> série_formelle
    crée_SF_de f renvoie la série formelle dont les coefficients sont les f(n) *)

let crée_SF_de f =
    let rec crée n =
        {   Constante = f n ;
            Reste = Gelé (function () -> crée (n +/ un))
        }
    in
    crée zéro ;;
(*----------------------------------------------------------------------------*)

(*----------------------------------------------------------------------------
  SF_de_poly : num list -> série_formelle
    SF_de_poly [a0 ; ... ; aN ] renvoie la série formelle qu'on imagine ! *)

let SF_de_poly l =
    let rec crée = function
        | [] -> { Constante = zéro ; Reste = Gelé (function () -> crée []) }
        | a :: q -> { Constante = a ; Reste = Gelé (function () -> crée q) }
    in
    crée l ;;
(*----------------------------------------------------------------------------*)

(*----------------------------------------------------------------------------
  crée_SF_expo_de : (num -> num) -> série_formelle
    crée_SF_expo_de f renvoie la série génératrice exponentielle de la suite des f(n) *)

let crée_SF_expo_de f =
    let rec crée n nn =
        {   Constante = (f n) // nn ;
            Reste = Gelé (function () -> crée (n +/ un) (nn */ (n +/ un)))
        }
    in
    crée zéro un ;;
(*----------------------------------------------------------------------------*)

(*----------------------------------------------------------------------------
  liste_des_coefficients : série_formelle -> int -> num list
    liste_des_coefficients s n renvoie la liste des n premiers coefficients
    de la série formelle s *)

let rec liste_des_coefficients s = function
    | 0 -> []
    | n -> s.Constante :: (liste_des_coefficients (reste_SF s) (n-1)) ;;
(*----------------------------------------------------------------------------*)
```

Programme 12 les fonctions de base sur les séries formelles (fin)

```
(*----------------------------------------------------------------------------
  zéro_SF : unit -> série_formelle
    renvoie la série formelle nulle *)

let rec zéro_SF () = { Constante = zéro ; Reste = Gelé zéro_SF } ;;
(*----------------------------------------------------------------------------*)

(*----------------------------------------------------------------------------
  zn_SF : int -> série_formelle
    xn_SF n renvoie la série formelle égale à z^n *)

let rec zn_SF = function
    | 0 -> { Constante = un ; Reste = Gelé zéro_SF }
    | n -> { Constante = zéro ; Reste = Gelé (function () -> zn_SF (n-1)) } ;;
(*----------------------------------------------------------------------------*)

(*----------------------------------------------------------------------------
  évalue_SF : série_formelle -> num -> int -> num
    évalue_SF s x n renvoie la valeur en x de la série s tronquée à n termes k0 *)

let rec évalue_SF s x n =
    if n = 0 then s.Constante
    else s.Constante +/ x */ (évalue_SF (reste_SF s) x (n-1)) ;;
(*----------------------------------------------------------------------------*)
```

Les calculs sur les séries formelles

Opérations élémentaires

On écrit sans la moindre difficulté les opérations de base de l'espace vectoriel $\mathbb{Q}[[X]]$: l'addition, la soustraction, et la multiplication par un scalaire ; voir le programme 13 page suivante.

Dérivation et intégration

On écrit, sur le modèle de Pierre Weis, dans le programme 14 page 19, les fonctions de dérivation et d'intégration des séries formelles.

La multiplication

L'application de la formule *à la Cauchy* conduit très directement au programme 15 page 19. Si deux séries formelles $S(X) = \sum_{n=0}^{+\infty} s_n X^n$ et $T(X) = \sum_{n=0}^{+\infty} t_n X^n$ ont pour produit la série formelle $U(X) = \sum_{n=0}^{+\infty} u_n X^n$, on calcule u_n en posant $u_n = \sum_{k=0}^{n} s_k t_{n-k}$. C'est pourquoi le calcul de u_n utilise tous les coefficients s_0, s_1, …, s_n et t_0, t_1, …, t_n.
La fonction auxiliaire `produit_de_Cauchy` calcule d'un joli coup de `it_list2` le coefficient u_n. Bien entendu, il faut prendre soin de fournir en arguments les listes $[s_0; \ldots; s_n]$ et $[t_0; \ldots; t_n]$ à la fonction `multiplie_aux`.

Programme 13 les opérations de base sur les séries formelles

```
(*----------------------------------------------------------------------------
  addition_SF : série_formelle -> série_formelle -> série_formelle *)

let rec addition_SF s t =
    {   Constante = s.Constante +/ t.Constante ;
        Reste = Gelé (function () -> addition_SF (reste_SF s) (reste_SF t)) } ;;
(*----------------------------------------------------------------------------*)

(*----------------------------------------------------------------------------
  soustraction_SF : série_formelle -> série_formelle -> série_formelle *)

let rec soustraction_SF s t =
    {   Constante = s.Constante -/ t.Constante ;
        Reste = Gelé (function () -> soustraction_SF (reste_SF s) (reste_SF t)) } ;;
(*----------------------------------------------------------------------------*)

(*----------------------------------------------------------------------------
  multiplication_SF_num : série_formelle -> num -> série_formelle *)

let rec multiplication_SF_num s n =
    {   Constante = s.Constante */ n ;
        Reste = Gelé (function () -> multiplication_SF_num (reste_SF s) n)
    } ;;
(*----------------------------------------------------------------------------*)

(*----------------------------------------------------------------------------
  opposé_SF : série_formelle-> série_formelle *)

let opposé_SF s = multiplication_SF_num s moins_un ;;
(*----------------------------------------------------------------------------*)
```

Programme 14 intégration et dérivation des séries formelles

```
(*-----------------------------------------------------------------------------
  intégration_SF : série_formelle -> num -> série_formelle
    intégration_SF s k0 renvoie la série primitive de terme constant k0 *)

let rec intégration_SF s k0 =
    {   Constante = k0 ;
        Reste = Gelé (function() -> intègre_SF_depuis_un_certain_rang s un)
    }
and intègre_SF_depuis_un_certain_rang s n =
    {   Constante = s.Constante // n ;
        Reste = Gelé (function () -> intègre_SF_depuis_un_certain_rang (reste_SF s) (n +/ un))
    } ;;
(*-----------------------------------------------------------------------------*)

(*-----------------------------------------------------------------------------
  dérivation_SF : série_formelle -> série_formelle
    dérivation_SF s renvoie la série dérivée *)

let dérivation_SF s =
    let rec dérivation_aux s n =
        {   Constante = s.Constante */ n ;
            Reste = Gelé (function () -> dérivation_aux (reste_SF s) (n +/ un))
        }
    in
    dérivation_aux (reste_SF s) un ;;
(*-----------------------------------------------------------------------------*)
```

Programme 15 multiplication de deux séries formelles

```
(*-----------------------------------------------------------------------------
  multiplication_SF : série_formelle -> série_formelle -> série_formelle *)

let multiplication_SF s t =
    let produit_de_Cauchy a b =
        let b' = rev b
        in
        it_list2 (fun t x y -> t +/ x */ y) zéro a b'
    in
    let rec multiplie_aux s t sl tl =
        let sl' = s.Constante :: sl
        and tl' = t.Constante :: tl
        in
        {   Constante = produit_de_Cauchy sl' tl' ;
            Reste = Gelé (function () -> multiplie_aux (reste_SF s) (reste_SF t) sl' tl')
        }
    in
    multiplie_aux s t [] [] ;;
(*-----------------------------------------------------------------------------*)
```

Le problème de la division

Nous écrirons ici deux méthodes très différentes pour la division de deux séries formelles.
Si on veut diviser une série formelle S par une série formelle T, il faut imposer que $v(T) \leqslant v(S)$, sans quoi la division est impossible. On simplifie alors par $X^{v(T)}$ les deux membres du quotient et on est rapporté au problème suivant.
Étant donné $S(X) = \sum_{n=0}^{+\infty} s_n X^n$ et $T(X) = \sum_{n=0}^{+\infty} t_n X^n$ avec la condition $t_0 \neq 0$, on cherche la série quotient $U(X) = \sum_{n=0}^{+\infty} u_n X^n$ définie par $S = UT$.
On obtient facilement les formules donnant les coefficients u_n en inversant la formule du produit de Cauchy :

$$\forall n \geqslant 1, \quad u_n = \frac{1}{t_0}\Big(s_n - \sum_{k=0}^{n-1} u_k t_{n-k}\Big).$$

On obtient ainsi le programme 16.

Programme 16 division de deux séries formelles, première méthode

```
(*-----------------------------------------------------------------------------
  division_SF : série_formelle -> série_formelle -> série_formelle *)

let rec division_SF s t =
    let rec Cauchy_inverse a b t0 sn =
        let b' = tl (rev b)
        in
        (sn -/ (it_list2 (fun s x y -> s +/ x */ y) zéro a b')) // t0
    in
    let rec divise_aux s t ul tl t0 =
        let tl' = t.Constante :: tl
        in
        let u_n = Cauchy_inverse ul tl' t0 s.Constante
        in
        {
            Constante = u_n ;
            Reste = Gelé (function ()
                          -> divise_aux
                                  (reste_SF s) (reste_SF t)
                                  (u_n :: ul) tl' t0)
        }
    in
    if t.Constante =/ zéro then
        if s.Constante =/ zéro then division_SF (reste_SF s) (reste_SF t)
        else failwith "Division impossible : valuation du numérateur inférieure
                       à celle du dénominateur"
    else
        divise_aux s t [] [] t.Constante ;;
(*-----------------------------------------------------------------------------*)
```

Il existe une autre façon de procéder : on sait en effet écrire le développement en série formelle de $I(X) = \dfrac{1}{a+X} = \displaystyle\sum_{k=0}^{+\infty} \frac{(-1)^k}{a^{k+1}} X^k$.
Il suffit alors de poser $a = t_0$ et, en utilisant la composition des séries formelles que nous n'avons pas encore écrite, de remarquer que $U = \dfrac{S}{T} = S \times (I \circ (T - t_0))$. Cela donnerait le programme 17.

Programme 17 division des séries formelles par composition

```
(*--------------------------------------------------------------------------------
  un_sur_a_plus_z_SF : num -> série_formelle
    un_sur_a_plus_z_SF a renvoie la série 1 / (a + z) *)

let un_sur_a_plus_z_SF a =
    if a =/ zéro then failwith "1/z n'a pas de développement en série formelle" ;
    let a' = zéro -/ a
    in
    let rec aux coeff =
        {   Constante = coeff ;
            Reste = Gelé (function () -> aux (coeff // a'))
        }
    in
    aux (un // a) ;;

(*--------------------------------------------------------------------------------*)

(*--------------------------------------------------------------------------------
  division_SF : série_formelle -> série_formelle -> série_formelle *)

let rec division_SF s t =
    if t.Constante =/ zéro then
        if s.Constante =/ zéro then division_SF (reste_SF s) (reste_SF t)
        else failwith "Division impossible : valuation du numérateur inférieure
                       à celle du dénominateur"
    else multiplication_SF
            s (composition_SF
                (un_sur_a_plus_z_SF t.Constante)
                { Constante = zéro ; Reste = Gelé (function () -> reste_SF t) }) ;;

(*--------------------------------------------------------------------------------*)
```

La composition des séries formelles : un problème difficile

Notons qu'un autre exemple typique de l'utilisation de la composition des séries formelles est celui de l'élévation à une puissance scalaire d'une série formelle.
En effet, on connaît le développement en série formelle de $(1+X)^a$, et on en déduit aisément le calcul de $S(z)^a$ sous réserve bien sûr que s_0^a soit calculable par la bibliothèque `num`.
On obtient ainsi le programme 18.

Programme 18 une autre utilisation de la composition, l'exponentiation des séries formelles

```
(*--------------------------------------------------------------------------
  un_plus_z_puissance_a_SF : num -> série_formelle
    un_plus_z_puissance_a_SF a renvoie la série (1 + z)^a *)

let un_plus_z_puissance_a_SF a =
    let rec aux coef k =
        let k' = k +/ un
        in
        {   Constante = coef ;
            Reste = Gelé (function () -> aux (coef */ (a -/ k) // k') k')
        }
    in
    {   Constante = un ;
        Reste = Gelé (function () -> aux a un)
    } ;;
(*--------------------------------------------------------------------------*)

(*--------------------------------------------------------------------------
  puissance_SF_num : série_formelle -> num -> série_formelle *)

let puissance_SF_num s n =
    let a = s.Constante
    in
    try
    multiplication_SF_num
        (composition_SF
            (un_plus_z_puissance_a_SF n)
            {
                Constante = zéro ;
                Reste = Gelé(function () -> reste_SF (multiplication_SF_num s (un // a)))
            })
        (if a =/ un then un else a **/ n)
    with _ -> failwith "Exponentiation impossible" ;;

(*--------------------------------------------------------------------------*)
```

Le plus naturel est de faire le calcul direct de la composée de deux séries formelles $U = \sum_{n=0}^{+\infty} u_n X^n = S \circ T = \left(\sum_{n=0}^{+\infty} s_n X^n\right) \circ \left(\sum_{n=0}^{+\infty} t_n X^n\right)$, en supposant bien sûr que $t_0 = 0$ sans quoi le calcul n'a pas de sens.
En effet, on écrit :

$$\forall n \geqslant 1, \quad u_n = \sum_{k=1}^{n} s_k \times \sum_{p_1+\cdots+p_k=n} t_{p_1} \ldots t_{p_k},$$

la deuxième sommation s'entendant sur tous les k-uplets d'entiers naturels non nuls de somme égale à n.
On traduit ces résultats en Caml, obtenant le programme 19 page suivante.

L'impression des séries formelles

Nous écrivons maintenant, sur la base précieuse de ce qu'avait rédigé Pierre Weis, les fonctions qui permettent d'afficher les séries formelles (bien sûr tronquées à un certain rang) comme des développements limités. Le lecteur est invité à relire la *Lettre de Caml* numéro 4 avant de se plonger dans le programme 20 page 25.

Faciliter l'écriture des séries formelles à l'utilisateur

De la même façon que la bibliothèque `num` définit de nouveaux opérateurs tous suffixés par `/`, comme `+/`, `=/`, etc, nous définissons des opérateurs pour les fonctions les plus usuelles de notre bibliothèque sur les séries formelles, que nous choisissons de systématiquement suffixer par `@`.
On obtient la liste des définitions du programme 21 page 25.
Il est alors facile d'écrire quelques exemples de séries formelles comme il est fait dans le programme 22 page 26.

La bibliothèque des séries formelles

Nous sommes maintenant en mesure d'écrire, dans le programme 23 page 27, l'interface de notre bibliothèque de calcul sur les séries formelles.
Nous avons choisi d'écrire une fonction `installe_impression : int -> unit` qui installe les procédures d'impression en fixant le rang de la troncature choisi.
Pour terminer, on trouvera dans le programme 24 page 28 une petite session Caml qui utilise cette nouvelle bibliothèque.

Programme 19 un calcul de la composée de deux séries formelles

```
(*-------------------------------------------------------------------------
  composition_SF : série_formelle -> série_formelle -> série_formelle
    composition_SF a b évalue a(b(z)), mais suppose donc b de valuation au moins 1 *)

let composition_SF s t =
    let rec intervalle i j =
        if i > j then []
        else i :: (intervalle (i + 1) j)
    in
    let rec k_somme k s =   (* k_somme 3 6 renvoie [[4; 1; 1]; [3; 2; 1]; [3; 1; 2]; ... *)
        if s = 0 then []
        else if k = 1 then [ [ s ] ]
        else it_list
                (fun ll i -> (map (function l -> i :: l) (k_somme (k - 1) (s - i))) @ ll)
                [] (intervalle 1 s)
    in
    let coeff av bv =
        let n = vect_length av - 1
        in
        let sbk l = it_list (fun x i -> x */ bv.(n - i)) un l
        in
        it_list
            (fun s k -> s +/ av.(n-k) */ (it_list
                                            (fun x l -> x +/ (sbk l))
                                            zéro
                                            (k_somme k n)))
            zéro
            (intervalle 1 n)
    in
    let rec aux al bl s t =
        let al' = s.Constante :: al
        and bl' = t.Constante :: bl
        in
        {   Constante = coeff (vect_of_list al') (vect_of_list bl') ;
            Reste = Gelé(function () -> aux al' bl' (reste_SF s) (reste_SF t))
        }
    in
    if t.Constante <>/ zéro then failwith "La composée a(b(z)) n'existe que si v(b) >= 1" ;
    {   Constante = s.Constante ;
        Reste = Gelé(function () -> aux [ s.Constante ] [ t.Constante ]
                                        (reste_SF s) (reste_SF t))
    } ;;

(*-------------------------------------------------------------------------*)
```

Programme 20 l'impression des séries formelles

```
#open "format" ;;

let print_num n = print_string (string_of_num n) ;;

let print_variable = function
    | 0 -> false
    | 1 -> print_string " z" ; true
    | n -> print_string " z^" ; print_int n ; true ;;

let print_term plus degré s =
    let c = s.Constante in
    if c =/ zéro then false else
    if c =/ un then begin print_string plus ; print_variable degré end else
    if c =/ moins_un
        then begin print_string "- " ; print_variable degré end
        else
        begin
            if c >=/ zéro then print_string plus else print_string "- " ;
            print_num (abs_num c) ;
            print_variable degré
        end ;;

let rec print_SF s until =
    open_hovbox 1;
    let c = s.Constante
    in
    if until == 0 then print_num c else
    let rest = ref s
    in
    let nul = ref true
    in
    if not (c =/ zéro) then (print_num c ; print_space() ; nul := false) ;
    for i = 1 to until do
        rest := reste_SF !rest;
        let delim = if !nul then "" else "+ "
        in
        if print_term delim i !rest then ( nul := false ; print_space())
    done ;
    if not !nul then print_string "+ " ;
    print_string "O(z^"; print_int (succ until) ;
    print_string ")" ;
    close_box() ;;

let print_par_défaut s = print_SF s 11 ;;
install_printer "print_par_défaut" ;;
```

Programme 21 les opérateurs sur les séries formelles

```
let prefix +@ = addition_SF
and prefix -@ = soustraction_SF
and prefix *@ = multiplication_SF
and prefix /@ = division_SF
and prefix @@ = composition_SF
and prefix ^@ = puissance_SF_num
and prefix !@ = SF_de_poly
and prefix %@ n s = multiplication_SF_num s n ;;
```

Programme 22 quelques exemples de séries formelles classiques

```
(* L'exemple de Pierre Weis : *)

let rec sinus =
    {   Constante = zéro ;
        Reste = Gelé (function ()
                        -> intègre_SF_depuis_un_certain_rang cosinus un) }
and cosinus =
    {   Constante = un ;
        Reste = Gelé (function ()
                        -> intègre_SF_depuis_un_certain_rang (opposé_SF sinus) un) } ;;

(* Fonctions classiques *)

let rec sinus_h = { Constante = zéro ;
                    Reste = Gelé (function () -> intègre_SF_depuis_un_certain_rang cosinus_h un) }
and cosinus_h = {   Constante = un ;
                    Reste = Gelé (function () -> intègre_SF_depuis_un_certain_rang sinus_h un) } ;;

let tangente = sinus /@ cosinus ;;
let tangente_h = sinus_h /@ cosinus_h ;;
let arctangente   = intégration_SF ((!@ [un]) /@ (!@ [un;zéro;deux])) zéro ;;
let arctangente_h = intégration_SF ((!@ [un]) /@ (!@ [un;zéro;moins_deux])) zéro ;;
let exponentielle = crée_SF_expo_de (function _ -> un) ;;
let ln_un_plus_z = intégration_SF (un_plus_z_puissance_a_SF moins_un) zéro ;;
let arcsinus = intégration_SF ((!@ [un;zéro;moins_deux]) ^@ moins_un_demi) zéro ;;
let arcsinus_h = intégration_SF ((!@ [un;zéro;deux]) ^@ moins_un_demi) zéro ;;

let catalan = ((!@ [un]) -@ ((!@ [un;moins_quatre]) ^@ un_demi))
               /@ (!@ [zéro;deux]) ;;
```

Programme 23 l'interface de la bibliothèque sur les séries formelles

```
 1 #open "num" ;;
 2
 3 type série_formelle ;;
 4
 5 value   prefix +@ : série_formelle -> série_formelle -> série_formelle
 6 and     prefix -@ : série_formelle -> série_formelle -> série_formelle
 7 and     prefix *@ : série_formelle -> série_formelle -> série_formelle
 8 and     prefix /@ : série_formelle -> série_formelle -> série_formelle
 9 and     prefix @@ : série_formelle -> série_formelle -> série_formelle
10 and     prefix ^@ : série_formelle -> num -> série_formelle
11 and     prefix!@ : num list -> série_formelle
12 and     prefix %@ : num -> série_formelle -> série_formelle
13
14 and
15     zéro    : num
16 and un      : num       and moins_un        : num
17 and deux    : num       and moins_deux      : num
18 and trois   : num       and moins_trois     : num
19 and quatre  : num       and moins_quatre    : num
20 and cinq    : num       and moins_cinq      : num
21 and six     : num       and moins_six       : num
22 and sept    : num       and moins_sept      : num
23 and huit    : num       and moins_huit      : num
24 and neuf    : num       and moins_neuf      : num
25 and dix     : num       and moins_dix       : num
26 and un_demi : num       and moins_un_demi   : num
27 and
28     print_SF            : série_formelle -> int -> unit
29 and print_par_défaut    : série_formelle -> unit
30 and installe_impression : unit -> unit
31 and
32     crée_SF_de      : (num -> num) -> série_formelle
33 and crée_SF_expo_de : (num -> num) -> série_formelle
34 and intégration_SF  : série_formelle -> num -> série_formelle
35 and dérivation_SF   : série_formelle -> série_formelle
36 and
37     sinus           : série_formelle
38 and cosinus         : série_formelle
39 and sinus_h         : série_formelle
40 and cosinus_h       : série_formelle
41 and tangente        : série_formelle
42 and tangente_h      : série_formelle
43 and arctangente     : série_formelle
44 and arctangente_h   : série_formelle
45 and exponentielle   : série_formelle
46 and ln_un_plus_z    : série_formelle
47 and arcsinus        : série_formelle
48 and arcsinus_h      : série_formelle
49 and catalan         : série_formelle ;;
```

Programme 24 une session CAML

```
>       Caml Light version 0.73/Mac.3

##open "sf" ;;
#load_object "sf" ;;
- : unit = ()
#catalan ;;
- : série_formelle = <abstr>
#installe_impression () ;;
- : unit = ()
#catalan ;;
- : série_formelle =
 1 +  z + 2 z^2 + 5 z^3 + 14 z^4 + 42 z^5 + 132 z^6 + 429 z^7 + 1430 z^8
  + 4862 z^9 + 16796 z^10 + 58786 z^11 + O(z^12)
#let a = !@ [ un ; un ] and b = !@ [ un ; deux ; trois ] ;;
a : série_formelle = 1 +  z + O(z^12)
b : série_formelle = 1 + 2 z + 3 z^2 + O(z^12)
#a /@ b ;;
- : série_formelle =
 1 -  z -  z^2 + 5 z^3 - 7 z^4 -  z^5 + 23 z^6 - 43 z^7 + 17 z^8 + 95 z^9
  - 241 z^10 + 197 z^11 + O(z^12)
#print_SF (a /@ b) 17 ;;
1 -  z -  z^2 + 5 z^3 - 7 z^4 -  z^5 + 23 z^6 - 43 z^7 + 17 z^8 + 95 z^9
 - 241 z^10 + 197 z^11 + 329 z^12 - 1249 z^13 + 1511 z^14 + 725 z^15
 - 5983 z^16 + 9791 z^17 + O(z^18)- : unit = ()
#let s = (tangente @@ sinus) -@ (sinus @@ tangente) ;;
s : série_formelle = 1/30 z^7 + 29/756 z^9 + 1913/75600 z^11 + O(z^12)
#print_SF tangente 19 ;;
 z + 1/3 z^3 + 2/15 z^5 + 17/315 z^7 + 62/2835 z^9 + 1382/155925 z^11
 + 21844/6081075 z^13 + 929569/638512875 z^15 + 6404582/10854718875 z^17
 + 443861162/1856156927625 z^19 + O(z^20)- : unit = ()
#
```