La lettre de Caml numéro 3

Laurent Chéno
54, rue Saint-Maur
75011 Paris
Tél. (1) 48 05 16 04
Fax (1) 48 07 80 18
email : `cheno@micronet.fr`

mars 1996

Édito

Je remercie Alain Bèges pour l'implémentation en Caml qu'il nous propose d'un algorithme de mise sous forme normale disjonctive (ou conjonctive) d'une proposition logique.
De même, Pierre Weis m'envoie un texte fort intéressant sur l'arithmétique exacte en Caml, signé Valérie Ménissier-Morain, et que vous trouverez bien sûr dans cette lettre.
Denis Monasse m'a fait observer que le traitement des expressions arithmétiques qui avait été proposé dans les lettres précédentes ne gérait pas convenablement les opérateurs habituels : c'est ainsi que toute opération était considérée comme associative à droite, ce qui posait problème pour l'évaluation de `2-3-5` *par exemple. On trouvera ici une solution correcte.*
En outre, je dois présenter mes excuses pour des erreurs de frappe dans un des fichiers de la Lettre numéro 1, au sujet du chargement d'un fichier image. La correction sera opérée sur le serveur de l'INRIA dès que cette Lettre numéro 3 sera disponible : chargez donc à nouveau la version qui vous convient de `bmp2clg.ml`*, qui est le fichier en cause.*
Bruno Petazzoni m'a signalé, en relisant les numéros précédents de cette Lettre, que Raymond Seroul, qui avait déjà publié son excellent Le petit livre de TEX, *a sorti récemment un nouvel ouvrage,* math-info Informatique pour mathématiciens, *où il présente un algorithme intéressant de génération des permutations. Je me suis fait un devoir d'en présenter une traduction en Caml.*

Table des matières

Forme normale dis/con-jonctive (merci à A. Bèges)

Mise sous forme normale

Je reprends ici les programmes d'Alain Bèges qui nous propose d'écrire les formes normales conjonctive et disjonctive des propositions logiques.
Voici pour commencer quelques fonctions utilitaires simples qui seront utiles dans la suite, et qui constituent le programme 1.

Programme 1 Fonctions utiles

```
let rec point_fixe tr x0 =
    let rec aux x xx = if x = xx then x else aux xx (tr xx)
    in
    aux x0 (tr x0) ;;

let set_of_list =
    let rec aux accu = function
        | [] -> accu
        | h :: t -> if mem h accu then aux accu t else aux (h :: accu) t
    in
    aux [] ;;

let remove_p prédicat =
    let rec aux accu = function
        | [] -> accu
        | h :: t -> if prédicat h then (aux accu t) else aux (h :: accu) t
    in aux [] ;;
```

Ensuite, il est temps de définir le type des propositions logiques (les variables seront représentées par `Var n`, où `n` représente un entier). Alain Bèges nous propose de commencer par gérer les `Implique` et `Équivalent` éventuels (on pourrait de même travailler avec des `Oux` et `Etx`). Plus intéressant est son choix d'*intérioriser* les négations : il s'agit de leur faire descendre (grâce aux lois de Morgan) les arbres d'expression. Inversement, on *extériorise* la conjonction et la disjonction. Tout cela fait l'objet du programme 2 page suivante.
Il nous propose alors d'écrire sous forme normale nos propositions, c'est le programme 3 page 5.
Il serait agréable de disposer de conjonctions (ou de disjonctions) n-aires. On va donc écrire les arguments de ces con/dis-jonctions sous forme de listes. Mais, en réalité, nous n'aurons pas encore là les formes normales espérées. Il convient en effet de simplifier l'expression obtenue. Diverses simplifications peuvent se présenter.
Le cas de la double-négation s'évacue facilement. Mais il faut aussi repérer des expressions du genre $a \vee \neg a$ ou $a \wedge \neg a$. Enfin il se peut qu'apparaisse inutilement deux fois la même variable dans une con/dis-jonction, puis, pour terminer, qu'il n'y ait tout simplement plus aucun argument! Toutes ces simplifications sont traitées dans le programme 4 page 5.
La génération des formes normales n'est plus qu'une formalité (cf. programme 5 page 6)!

Programme 2 Préliminaires à la mise sous forme normale

```
type proposition = Var of int
                 | Non of proposition
                 | Implique of proposition * proposition
                 | Equivalent of proposition * proposition
                 | Ou of proposition * proposition
                 | Et of proposition * proposition ;;

let éliminer_implique =
    let rec étape = function
        | Var(n)            -> Var(n)
        | Non(t)            -> Non(étape t)
        | Implique(t1,t2)   -> Ou(Non(étape t1),étape t2)
        | Equivalent(t1,t2) -> Et( Ou(Non(étape t1),étape t2) ,
                                   Ou(Non(étape t2),étape t1) )
        | Ou(t1,t2)         -> Ou(étape t1,étape t2)
        | Et(t1,t2)         -> Et(étape t1,étape t2)
    in point_fixe étape ;;

let intérioriser_négation =
    let rec étape = function
        | Non(Et(t1,t2)) -> Ou(Non(étape t1),Non(étape t2))
        | Non(Ou(t1,t2)) -> Et(Non(étape t1),Non(étape t2))
        | Var(n) -> Var(n)
        | Non(t) -> Non(étape t)
        | Ou(t1,t2) -> Ou(étape t1,étape t2)
        | Et(t1,t2) -> Et(étape t1,étape t2)
    in point_fixe étape ;;

let exterioriser_conjonction =
    let rec étape = function
        | Ou(f,Et(g,h)) -> Et(Ou(étape f,étape g),Ou(étape f,étape h))
        | Ou(Et(g,h),f) -> Et(Ou(étape g,étape f),Ou(étape h,étape f))
        | Var(n) -> Var(n)
        | Non(t) -> Non(étape t)
        | Ou(t1,t2) -> Ou(étape t1,étape t2)
        | Et(t1,t2) -> Et(étape t1,étape t2)
    in point_fixe étape ;;

let exterioriser_disjonction =
    let rec étape = function
        | Et(f,Ou(g,h)) -> Ou(Et(étape f,étape g),Et(étape f,étape h))
        | Et(Ou(g,h),f) -> Ou(Et(étape g,étape f),Et(étape h,étape f))
        | Var(n) -> Var(n)
        | Non(t) -> Non(étape t)
        | Ou(t1,t2) -> Ou(étape t1,étape t2)
        | Et(t1,t2) -> Et(étape t1,étape t2)
    in point_fixe étape ;;
```

Programme 3 Formes normales intermédiaires

```
let fnc_vers_ll p =
    let aplatir_et =
        let rec aux l = function
            | Et(a,b) -> (aux l a) @ (aux l b)
            | a -> a :: l
        in aux []
    and aplatir_ou =
        let rec aux l = function
            | Ou(a,b) -> (aux l a) @ (aux l b)
            | a -> a :: l
        in aux []
    in map aplatir_ou (aplatir_et p) ;;

let fnd_vers_ll p =
    let aplatir_et =
        let rec aux l = function
            | Et(a,b) -> (aux l a) @ (aux l b)
            | a -> a :: l
        in aux []
    and aplatir_ou =
        let rec aux l = function
            | Ou(a,b) -> (aux l a) @ (aux l b)
            | a -> a :: l
        in aux []
    in map aplatir_et (aplatir_ou p) ;;
```

Programme 4 Simplifications des expressions logiques

```
let éliminer_double_négation =
    let élim =
        let rec étape = function
            | Var(n) -> Var(n)
            | Non(Non(a)) -> (étape a)
            | Non(a) -> Non(étape a)
        in point_fixe étape
    in map (map élim) ;;

let éliminer_a_non_a =
    let inutile liste =
        let négation = function
            | Var(n) -> exists (function x -> x = Non(Var(n))) liste
            | Non(Var(n)) -> exists (function x -> x = Var(n)) liste
        in exists négation liste
    in map (function x -> if (inutile x) then [] else x) ;;

let éliminer_variable_inutile = map set_of_list ;;

let éliminer_vide = remove_p (function [] -> true | _ -> false) ;;
```

Notons que les deux fonctions `formeC` et `formeD` renvoient des listes de listes de propositions.
Par exemple `formeD` doit renvoyer un résultat qui représente une expression du genre $(a \wedge \neg b \wedge c) \vee (a \wedge b \wedge c)$. Il s'agira ici d'une liste à 2 éléments, le premier étant une liste des 3 propositions a, $\neg b$ et c, le second une liste des 3 propositions a, b, c.

Programme 5 Formes normales conjonctives ou disjonctives

```
let formeC t =
     (éliminer_vide
        (éliminer_a_non_a
            (éliminer_variable_inutile
                (éliminer_double_négation
                    (fnc_vers_ll
                        (fnc t)))))) ;;

let formeD t =
     (éliminer_vide
        (éliminer_a_non_a
            (éliminer_variable_inutile
                (éliminer_double_négation
                    (fnd_vers_ll
                        (fnd t)))))) ;;
```

Un parseur pour lire des expressions logiques

Passons rapidement sur le lexeur, qu'on trouvera dans le programme 6.

Programme 6 Analyse lexicale des expressions logiques

```
type lexème = Entier of int | Conjonction | Disjonction | Implication | Équivalence
                | Négation | ParenthèseGauche | ParenthèseDroite ;;

let int_of_digit c = (int_of_char c) - (int_of_char ‘0‘) ;;

let rec MangeEntier flot accu = match flot with
    | [< ’(‘0‘..‘9‘ as c) >] -> MangeEntier flot (10*accu+(int_of_digit c))
    | [< >] -> Entier(accu) ;;

let rec lexeur flot = match flot with
    | [< ’(‘ ‘ | ‘\r‘ | ‘\t‘ | ‘\n‘) >] -> lexeur flot
    | [< ’‘^‘ >] -> [< ’Conjonction ; (lexeur flot) >]
    | [< ’‘|‘ >] -> [< ’Disjonction ; (lexeur flot) >]
    | [< ’‘=‘;’‘>‘ >] -> [< ’Implication ; (lexeur flot) >]
    | [< ’‘<‘;’‘=‘;’‘>‘ >] -> [< ’Équivalence ; (lexeur flot) >]
    | [< ’‘-‘ >] -> [< ’Négation ; (lexeur flot) >]
    | [< ’‘(‘ >] -> [< ’ParenthèseGauche ; (lexeur flot) >]
    | [< ’‘)‘ >] -> [< ’ParenthèseDroite ; (lexeur flot) >]
    | [< ’(‘0‘..‘9‘ as c) >]
            -> [< ’(MangeEntier flot (int_of_digit c)) ; (lexeur flot) >]
    | [< >] -> [< >] ;;
```

Il s'agit ici de respecter les priorités habituelles: la conjonction l'emporte sur la disjonction. La négation est de plus haute priorité, l'implication ou l'équivalence sont de priorité intermédiaire entre la négation et la conjonction.
Suivant les exemples vus précédemment, on obtient la grammaire non ambigüe suivante :

```
E ::= F ou E | F
F ::= G et F | G
G ::= H => H | H <=> H | H
H ::= non I | I
I ::= ( E ) | entier
```

qui se factorise ainsi :

```
E  ::= F E'
E' ::= ou E | ∅
F  ::= G F'
F' ::= et F | ∅
G  ::= H G'
G' ::= => H | <=> H | ∅
H  ::= non I | I
I ::= ( E ) | entier
```

De même qu'on l'avait fait pour les expressions arithmétiques, on peut écrire le parseur correspondant, ce que fait le programme 7 page suivante.

Un formatteur pour écrire des propositions logiques sous forme normale

Nous allons ici donner un exemple d'utilisation de la bibliothèque standard `format`, qui affiche de façon agréable les formes normales conjonctives et disjonctives que nous avons obtenues.
Nous dirons à Caml d'utiliser nos fonctions de formatage grâce à la fonction `install_printer`. Il faut donc pouvoir distinguer les formes normales disjonctives des conjonctives, par leur type: c'est pourquoi nous définissons un nouveau type et modifions les fonctions `formeD` et `formeC` en conséquence (programme 8 page suivante).
Il reste à écrire puis installer une fonction `print_forme_normale`.
Tout cela est plus fastidieux que difficile (programmes 9 page 9 et 10 page 10).

Programme 7 Analyse syntaxique des expressions logiques

```
exception Syntax_error ;;

let rec parseur_E flot = match flot with
    | [< parseur_F f ; parseur_E' e' >]
            -> match e' with
                | [< 'Ou(_,e) >] -> Ou(f,e)
                | [< >] -> f
    | [< >] -> raise Syntax_error
and parseur_E' flot = match flot with
    | [< 'Disjonction ; parseur_E e >] -> [< 'Ou(Var(0),e) >]
    | [< >] -> [< >]
and parseur_F flot = match flot with
    | [< parseur_G g ; parseur_F' f' >]
            -> match f' with
                | [< 'Et(_,e) >] -> Et(g,e)
                | [< >] -> g
    | [< >] -> raise Syntax_error
and parseur_F' flot = match flot with
    | [< 'Conjonction ; parseur_F f >] -> [< 'Et(Var(0),f) >]
    | [< >] -> [< >]
and parseur_G flot = match flot with
    | [< parseur_H h ; parseur_G' g' >]
            -> match g' with
                | [< 'Implique(_,e) >] -> Implique(h,e)
                | [< 'Equivalent(_,e) >] -> Equivalent(h,e)
                | [< >] -> h
    | [< >] -> raise Syntax_error
and parseur_G' flot = match flot with
    | [< 'Implication ; parseur_G g >] -> [< 'Implique(Var(0),g) >]
    | [< 'Équivalence ; parseur_G g >] -> [< 'Equivalent(Var(0),g) >]
    | [< >] -> [< >]
and parseur_H flot = match flot with
    | [< 'Négation ; parseur_I i >] -> Non(i)
    | [< parseur_I i >] -> i
    | [< >] -> raise Syntax_error
and parseur_I flot = match flot with
    | [< 'ParenthèseGauche ; parseur_E e ; 'ParenthèseDroite >] -> e
    | [< 'Entier(n) >] -> Var(n)
    | [< >] -> raise Syntax_error ;;

let parseur s = parseur_E (lexeur (stream_of_string s)) ;;
```

Programme 8 Distinction des formes disjonctives et conjonctives

```
type forme_normale = FNC of proposition list list
                   | FND of proposition list list ;;

let forme_normale_conjonctive s = FNC(formeC (parseur s))
and forme_normale_disjonctive s = FND(formeD (parseur s)) ;;
```

Programme 9 Fonctions auxiliaires du formatteur

```
#open "format" ;;

let print_variable = function
    | Var(n)       -> open_hbox () ;
                      print_char (char_of_int (n-1+(int_of_char `a`))) ;
                      close_box ()
    | Non(Var (n)) -> open_hbox () ;
                      print_char `-` ;
                      print_char (char_of_int (n-1+(int_of_char `a`))) ;
                      close_box () ;;

let print_disjonction l =
    let rec aux = function
        | [] -> ()
        | tête :: queue -> print_space () ;
                           print_char `|` ;
                           print_space () ;
                           print_variable tête ;
                           aux queue
    in
    open_hovbox 3 ;
    match l with
        | [] -> ()
        | [x] -> print_variable x
        | tête :: queue ->
                print_char `(` ;
                print_variable tête ; aux queue ;
                print_char `)` ;
    close_box () ;;

let print_conjonction l =
    let rec aux = function
        | [] -> ()
        | tête :: queue -> print_space () ;
                           print_char `^` ;
                           print_space () ;
                           print_variable tête ;
                           aux queue
    in
    open_hovbox 3 ;
    match l with
        | [] -> ()
        | [x] -> print_variable x
        | tête :: queue ->
                print_char `(` ;
                print_variable tête ; aux queue ;
                print_char `)` ;
    close_box () ;;
```

Programme 10 Installation du formatteur des formes normales

```
let print_fnc ll =
    let rec aux = function
        | [] -> ()
        | tête :: queue -> print_space () ;
                           print_char `^` ;
                           print_space () ;
                           print_disjonction tête ;
                           aux queue
    in
    open_hvbox 2 ;
    match ll with
        | [] -> ()
        | [ l ] -> print_conjonction l
        | tête :: queue -> print_break(2,0) ;
                           print_disjonction tête ;
                           aux queue ;
    close_box () ;;

let print_fnd ll =
    let rec aux = function
        | [] -> ()
        | tête :: queue -> print_space () ;
                           print_char `|` ;
                           print_space () ;
                           print_conjonction tête ;
                           aux queue
    in
    open_hvbox 2 ;
    match ll with
        | [] -> ()
        | [ l ] -> print_conjonction l
        | tête :: queue -> print_break(2,0) ;
                           print_conjonction tête ;
                           aux queue ;
    close_box () ;;

let print_forme_normale = function
    | FNC ll -> print_fnc ll
    | FND ll -> print_fnd ll ;;

install_printer "print_forme_normale" ;;
```

Le résultat est d'une utilisation facile. Une session Caml fournira par exemple :

```
>       Caml Light version 0.71/mac

#include "formes_normales.ml";;
[...]
#let exemple = "(1|2|3)^((1=>2)|(2|-3))^-((1=>2)^(2|-3))" ;;
exemple : string = "(1|2|3)^((1=>2)|(2|-3))^-((1=>2)^(2|-3))"
#forme_normale_disjonctive exemple ;;
- : forme_normale =   (-b ^ -a ^ c) | (-b ^ -c ^ a)
#forme_normale_conjonctive exemple ;;
- : forme_normale =
   (c | -b) ^ (c | a) ^ -b ^ (-b | a) ^ (-c | b | -a) ^ (c | b | a)
```

De l'utilisation de l'arithmétique exacte en Caml, par Valérie Ménissier-Morain

On dispose dans chacune des implémentations de Caml d'une arithmétique rationnelle exacte. Nous présentons ici deux exemples d'utilisation de cette arithmétique. Le premier exemple met l'expressivité symbolique de Caml et son arithmétique rationnelle exacte au service de la démonstration de formules mathématiques. Le second exemple montre comment calculer des décimales de π avec Caml en employant la formule utilisée pour battre le record du monde.

Démonstration automatique de formules trigonométriques

Nous considérons ici un exemple où l'exactitude des résultats est obligatoire: il s'agit de *prouver* les formules définissant $\pi/4$ avec des combinaisons linéaires à coefficients rationnels d'arctangentes. Pour que la preuve soit valide, il ne doit y avoir aucune erreur d'arrondi pendant le calcul.
On testera le programme sur les formules classiques suivantes pour $\frac{\pi}{4}$:

$$\begin{aligned} \frac{\pi}{4} &= 4 \ \arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right) \\ \frac{\pi}{4} &= 12 \ \arctan\left(\frac{1}{18}\right) + 8 \ \arctan\left(\frac{1}{57}\right) - 5 \ \arctan\left(\frac{1}{239}\right). \end{aligned}$$

Classiquement, on démontre une telle formule en utilisant successivement deux arguments: le premier consiste, par des majorations ou minorations grossières à prouver que l'expression avec des arctan se trouve dans le quart supérieur droit du cercle trigonométrique ce qui permet dans un deuxième temps de se contenter de démontrer que la tangente du second arc est bien égale à 1. C'est cette seconde partie de la preuve qui est bien souvent la plus longue et la plus pénible, que l'on se propose d'automatiser avec la bibliothèque d'arithmétique rationnelle exacte.
On doit déterminer la valeur numérique de formules du type:

$$\tan\left(\sum_{i=1}^{n} a_i \ \arctan r_i\right) \text{ avec } a_i \in \mathbb{Z}, r_i \in \mathbb{Q} \text{ et } n \in \mathbb{N}^*.$$

On définit le type de données Caml qui décrit ces expressions:

```
type term = Arctan of num | Mult of num * term | Add of term * term;;
```

Puis on définit récursivement la fonction tan sur de tels termes en utilisant les égalités suivantes:

$$\begin{aligned}
\tan(0) &= 0 \\
\tan(-a) &= -\tan(a) \\
\tan(a+b) &= \frac{\tan(a)+\tan(b)}{1-\tan(a)\ \times\ \tan(b)} \\
\tan(\arctan(x)) &= x \quad \text{si } 0 \leqslant x < \frac{\pi}{2}
\end{aligned}$$

et particulièrement une conséquence de la troisième égalité:

$$\tan(2\times x) = \frac{2\times\tan(x)}{1-\tan(x)^2}.$$

Dans le cas de la multiplication d'une expression par un nombre, on calcule la fonction tan par décomposition en base 2 de ce nombre.
On en déduit le code suivant pour Caml V3.1:

```
#standard arith false;;

let rec tan = function
  Arctan x -> x
| Mult (0, x) -> 0
| Mult (1, x) -> tan x
| Mult (-1, x) -> -(tan x)
| Mult (n, x) -> (* Calcul de tan(n*x) par décomposition de n en base 2 *)
    let k = floor (n/2) in
    if n = 2*k then let tan_a = tan (Mult (k, x)) in
                    (2*tan_a)/(1-tan_a*tan_a)
    else let tan_a = tan (Mult (k, x)) and tan_b = tan x in
         let tan_c = (tan_a + tan_b)/(1-tan_a*tan_b) in
         (tan_a+tan_c)/(1-tan_a*tan_c)
| Add (a, b) -> let tan_a = tan a and tan_b = tan b in
                (tan_a+tan_b)/(1-tan_a*tan_b);;
```

Puisque notre arithmétique est exacte, on peut maintenant facilement *prouver* les lemmes fastidieux pour les formules précédentes:

```
#tan (Add (Mult (4, Arctan (1/5)), Mult (-1, Arctan (1/239))));;
1: num

#tan (Add (Mult (12, Arctan (1/18)),
          Add (Mult (8, Arctan (1/57)), Mult (-5, Arctan (1/239)))));;
1: num
```

Approximations rationnelles de π

Plus classiquement, voyons une utilisation intensive de cette bibliothèque avec le calcul d'approximations rationnelles de π. On utilise une formule due à Dimitri et Gregory Chudnovsky [4]:

$$\frac{1}{\pi} = \sum_{n=0}^{\infty} (-1)^n \frac{12(6n)!}{(n!)^3(3n)!} \frac{13591409 + 545140134n}{(640320^3)^{n+\frac{1}{2}}}.$$

Cette formule est une conséquence de la théorie des équations modulaires initiée par Srinivasa Ramanujan [7, 5] et développée plus récemment par Jonathan et Peter Borwein [1, 2, 3]).

On implémente cette formule avec le type `num` (le code source pour les types `big_int` et `nat` est décrit dans l'annexe A de [6]).

```
(* nombre de chiffres de l'élément x de type num,
   en base 2^32 si x est un grand entier *)
let num_digits_of_num x = num_of_int (num_digits_big_int (big_int_of_num x));;

(* la fonction de test pour la boucle *)
let test (x, y, z, t) =
  (num_digits_of_num x)+(num_digits_of_num y)+z > (num_digits_of_num t);;

(* la racine carrée de 640320 représentée comme un rationnel
   à une précision de digits chiffres décimaux *)
let sqrt640320 digits =
  let pow = 10**digits in
  (num_of_big_int (sqrt_big_int (big_int_of_num (640320*pow*pow))), pow);;

(* approximation rationnelle de π à une précision de digits chiffres décimaux *)
let approx_pi digits =
  let prod = ref 12
  and sum = ref 13591409
  and D = ref 640320
  and N = ref (12*13591409)
  and sn = ref 0
  and binom = ref 1
  and pown3 = ref 0
  and (sqrt, pow) = sqrt640320 (digits-2)
  and pow3 = 640320**3 in
  let sizeB = succ (num_digits_of_num pow) in
  while test (!prod, !sum, sizeB, !D) do
    prod := -8*(!sn+1)*(!sn+3)*(!sn+5)*!prod;
    sum := 545140134+!sum;
    pown3 := !binom+!pown3;
    D := !pown3*pow3*!D;
    N := !pown3*pow3*!N+!prod*!sum;
    sn := !sn+6;
    binom := !sn+!binom
  done;
  sqrt*!D/(!N*pow);;
```

nombre de décimales	quantité mesurée	num	big_int	nat
1000	temps d'exécution	0.55s	0.49s	0.36s
	temps de glanage de cellules	0s	0s	0s
10000	temps d'exécution	46.61s	45.39s	34.98s
	temps de glanage de cellules	12.89s	16.00s	0s
20000	temps d'exécution	182.86s	179.48s	147.83s
	temps de glanage de cellules	59.32s	77.82s	0s
	nombre de lignes de code source	28	39	125

Figure 1: Temps d'exécution et de glanage de cellules.

On peut maintenant imprimer, par exemple, les 500 premières décimales de π sous forme de table en utilisant la bibliothèque de formatage correspondant:

```
#load_lib_file "format_numbers";;
/home/margaux/formel2/caml/V3.1/lib/format_numbers.lo loaded
() : unit

#print_string (beautiful_string approx_num_fix (#500, (approx_pi 500)));;


+3.14 15926  53589 79323  84626 43383  27950 28841  97169 39937
51058 20974  94459 23078  16406 28620  89986 28034  82534 21170
67982 14808  65132 82306  64709 38446  09550 58223  17253 59408
12848 11174  50284 10270  19385 21105  55964 46229  48954 93038
19644 28810  97566 59334  46128 47564  82337 86783  16527 12019

09145 64856  69234 60348  61045 43266  48213 39360  72602 49141
27372 45870  06606 31558  81748 81520  92096 28292  54091 71536
43678 92590  36001 13305  30548 82046  65213 84146  95194 15116
09433 05727  03657 59591  95309 21861  17381 93261  17931 05118
54807 44623  79962 74956  73518 85752  72489 12279  38183 01194

913
() : unit
```

La figure 1 compare les temps d'exécution, de glanage de cellules (sur une DecStation 5000/200 sous Ultrix 4.1) et la longueur du code source pour les trois implémentations en travaillant respectivement avec le type `num`, `big_int` et `nat`.
Les temps d'écriture et de mise au point du code source pour les implémentations avec le type `num` et le type `big_int` son comparables. Mais il n'en va pas de même pour la version utilisant le type `nat`, version qui a nécessité beaucoup plus d'efforts et de temps avant d'être correcte. D'un point de vue algorithmique, la version avec les `nat` nécessite la justification de la taille maximale de chaque variable et il faut prouver que le nombre d'itérations effectuées correspond effectivement à la précision demandée par l'utilisateur. L'utilisation du type `nat` conduit l'implémenteur à optimiser toujours plus l'algorithme utilisé pour rentabiliser l'effort mathématique. Il est par conséquent possible que les

différences de temps entre la version avec le type `nat` et les deux autres soient dues en partie à des optimisations non-obligatoires de l'implémentation avec des `nat`.
Il est intéressant de noter que l'utilisation du type `num` à la place des types sous-jacents n'est pas nécessairement moins efficace. En ce qui concerne l'algorithme considéré ici, à chaque itération on doit calculer le produit de plusieurs petits entiers par un grand entier et la multiplication est effectuée aussi longtemps que possible avec des petits entiers, ce qui exige une moindre allocation que le calcul en grands entiers. Typiquement les temps d'exécution sont comparables (à 2% près environ) alors que les temps de glanage de cellules sont inférieurs d'un tiers environ, et le temps total est inférieur d'environ 5% avec le type `num`.

Le code pour Caml Light

Pour utiliser la bibliothèque de grands nombres de Caml Light sous Unix, il faut le lancer avec

```
camllight camlnum
```

puis ouvrir le module `num` par :

```
#open"num";;
```

La bibliothèque d'arithmétique exacte est beaucoup moins intégrée, tant au niveau des opérations que des constantes, à Caml Light qu'à Caml. On peut cependant diminuer cet inconfort en définissant un imprimeur pour le type `num` :

```
#open "format";;
let print_num n = print_string (string_of_num n);;
install_printer "print_num";;
```

ce qui nous permet d'avoir un affichage des nombres de type `num` tel que

```
num_of_string "1/2";;
- : num = 1/2
```

alors que sans cela on obtient

```
num_of_string "1/2";;
- : num = Ratio <abstr>
```

De même nous redéfinissons les symboles arithmétiques habituels pour qu'ils opèrent sur le type `num` :

```
let prefix + = prefix +/
and prefix - = prefix -/
and prefix * = prefix */
and prefix / = prefix //;;
```

Nous donnons ici le code pour pour chacun des deux exemples:

```
#open "num";;
type term = Arctan of num | Mult of num * term | Add of term * term;;

let tan_aux a b = (a+b)/((num_of_int 1)-a*b);;
```

```
let rec tan = function
  Arctan x -> x
| Mult (n, x) ->
   if n = num_of_int 0 then num_of_int 0 else
   if n = num_of_int 1 then tan x else
   if n = num_of_int (-1) then minus_num (tan x) else
   let k = quo_num n (num_of_int 2) in
       if n = (num_of_int 2)*k
          then let tan_a = tan (Mult (k, x)) in tan_aux tan_a tan_a
          else let tan_a = tan (Mult (k, x)) and tan_b = tan x in
               let tan_c = tan_aux tan_a tan_b in tan_aux tan_a tan_c
| Add (a, b) -> let tan_a = tan a and tan_b = tan b in tan_aux tan_a tan_b;;

(* Exemple *)
tan (Add (Mult (num_of_int 4, Arctan (num_of_string "1/5")),
          Mult (num_of_int (-1), Arctan (num_of_string "1/239"))));;

tan (Add (Mult (num_of_int 12, Arctan (num_of_string "1/18")),
     Add (Mult (num_of_int 8, Arctan (num_of_string "1/57")),
          Mult (num_of_int (-5), Arctan (num_of_string "1/239")))));;
```

et pour le deuxième exemple

```
#open "big_int";;
#open "num";;
let num_digits_of_num x = num_digits_big_int (big_int_of_num x);;

let test (x, y, z, t) =
  add_int (num_digits_of_num x) (add_int (num_digits_of_num y) z) > (num_digits_of_num t);;

let sqrt640320 digits =
  let pow = power_num (num_of_int 10) digits in
  let sqr = (num_of_int 640320)*(square_num pow) in
  (num_of_big_int (sqrt_big_int (big_int_of_num sqr)), pow);;

let approx_pi digits =
  let prod = ref (num_of_int 12)
  and sum = ref (num_of_int 13591409)
  and D = ref (num_of_int 640320)
  and N = ref (num_of_int (mult_int 12 13591409))
  and sn = ref 0
  and binom = ref (num_of_int 1)
  and pown3 = ref (num_of_int 0)
  and (sqrt, pow) = sqrt640320 (digits-(num_of_int 2))
  and pow3 = power_num (num_of_int 640320) (num_of_int 3) in
  let sizeB = succ (num_digits_of_num pow) in
  while test (!prod, !sum, sizeB, !D) do
    prod := (num_of_int (-8)) * (num_of_int (add_int !sn 1)) *
            (num_of_int (add_int !sn 3)) * (num_of_int (add_int !sn 5)) * !prod;
    sum := (num_of_int 545140134) + !sum;
    pown3 := !binom + !pown3;
    D := !pown3 * pow3 * !D;
    N := !pown3 * pow3 * !N + !prod * !sum;
```

```
    sn := add_int !sn 6;
    binom := (num_of_int !sn) + !binom
  done;
  (sqrt * !D)/(!N * pow);;

approx_num_fix 500 (approx_pi (num_of_int 500));;
```

La fonction `beautiful_string` qui nous a permis de tabuler les 500 premières décimales de π n'a pas été portée en Caml Light.

Une version fonctionnelle de l'inverse d'une permutation, par B. Petazzoni

Dans le numéro 2 de la Lettre de Caml, Laurent étudie les permutations; pour inverser aisément une permutation, il est amené à la représenter par un vecteur. Je propose ici une méthode totalement fonctionnelle.
Deux idées, à la base : tout d'abord, réaliser un tri revient à déterminer la permutation inverse d'une permutation donnée. D'autre part, on connaît l'interprétation matricielle de la méthode des transformations élémentaires: on multiplie à gauche simultanément une matrice A et la matrice identité d'ordre n par des matrices de dilatations et transvections ; lorsque A a été transformée en la matrice identité d'ordre n, cette dernière a été transformée en la matrice inverse de A.
Nous allons donc considérer une permutation f de $[1, n]$ comme une application de `'int` vers `'int`. Pour déterminer son inverse, nous composons f et id à gauche simultanément par des transpositions. Le reste est facile à comprendre.

Permutations (d'après R. Seroul, merci à B. Petazzoni)

Dans son livre, pages 199–204, [9], Raymond Seroul (auquel on devait déjà [8]) décrit un algorithme astucieux dû à Johnson pour la génération des permutations, qui date de 1963.

L'algorithme de Johnson de génération des permutations

Johnson représente une permutation par des entiers surmontés de girouettes, comme, par exemple :

$$(1, \leftarrow), (3, \leftarrow), (5, \leftarrow), (7, \rightarrow), (6, \leftarrow), (4, \rightarrow), (2, \rightarrow).$$

On dira sur cet exemple que 1 voit *dehors*, 2 voit *dehors*, 3 voit 1, 4 voit 2, 5 voit 3, 6 voit 7 et 7 voit 6. Un entier sera décrété *mobile* s'il voit un entier plus petit que lui, et pas dehors. (Il y a là, je pense, une erreur de frappe dans [9].) Ainsi sont ici mobiles les entiers 3, 5, 7, 4, et eux seulement.

Programme 11 Un calcul fonctionnel de l'inverse d'une permutation

```
(* deux définitions utiles *)
let id = function x -> x;;
let compose f g = function x-> f(g x);;

(* la transposition de p et q *)
let tau p q x = if x = p then q else if x = q then p else x ;;

(* la composition à gauche par icelle *)
let by_tau p q h = compose (tau p q) h ;;

(* tout est prêt, allons-y !  *)
let inverse f n =
   let rec inv_rec f g n =
     if n = 0 then g else
     if n = f(n) then inv_rec f g (n-1)
                 else inv_rec (by_tau n (f n) f) (by_tau n (f n) g) (n-1)
   in
   inv_rec f id n ;;

(* on fait un essai pour voir *)
let f = function 1 -> 3 | 2 -> 5 | 3 -> 4 | 4 -> 1 | 5 -> 2 | 6 -> 6 | x -> x ;;

for k=1 to 6 do print_int(f(inverse f 6 k)) done ;;
```

Description

L'algorithme de Johnson balaie alors toutes les permutations en suivant le schéma suivant :

1. on démarre avec la permutation $(1, \leftarrow), (2, \leftarrow), \ldots, (n, \leftarrow)$;
2. on affiche la permutation courante ;
3. on recherche le plus grand entier mobile m — s'il n'y a pas d'entier mobile, on termine ici ;
4. on échange m et l'entier que voit m, sans modifier leurs girouettes respectives ;
5. on inverse la direction des girouettes de tous les entiers $k > m$;
6. on reprend à l'étape 2.

Cet algorithme se programme sans difficulté réelle en Caml, et fait l'objet des programmes 12 page ci-contre, et 13 page 20, plus spécialement chargé de l'affichage du résultat.

Élucidation

Je laisse à votre sagacité le soin de faire la preuve de cet algorithme : ce serait bien sûr un exercice assez peu trivial, surtout si l'on ne lit pas les quelques

Programme 12 Génération des permutations par l'algorithme de Johnson

```
exception Fin ;;

type sens = Gauche | Droite ;;

let valeur_de (_,n) = n
and sens_de (s,_) = s ;;

let voit table i =
    match sens_de table.(i) with
        | Gauche -> i-1
        | Droite -> i+1 ;;

let est_mobile table i =
    try valeur_de table.(voit table i) < valeur_de table.(i)
    with Invalid_argument(_) -> false ;;

let inverse table i =
    match table.(i) with
        | Gauche,p -> table.(i) <- Droite,p
        | Droite,p -> table.(i) <- Gauche,p ;;

let initialise n =
    let v = make_vect n (Gauche,0)
    in
    for i = 0 to n-1 do v.(i) <- (Gauche,i+1) done ;
    v ;;

let avance table n =
    let rec cherche_max_mobile i indice_du_max =
        if i = n then
            if indice_du_max = (-1) then raise Fin else indice_du_max
        else
            if est_mobile table i && (indice_du_max = (-1) ||
                valeur_de table.(i) > (valeur_de table.(indice_du_max)) )
            then cherche_max_mobile (i+1) i
            else cherche_max_mobile (i+1) indice_du_max
    in
    let indice_du_max = cherche_max_mobile 0 (-1)
    in
    let le_max = table.(indice_du_max)
    and indice_vu = voit table indice_du_max
    in
    let vu = table.(indice_vu)
    in
    table.(indice_du_max) <- vu ; table.(indice_vu) <- le_max ;
    for i = 0 to n-1 do
        if (valeur_de table.(i)) > (valeur_de le_max) then inverse table i
    done ;;
```

Programme 13 Affichage de la liste des permutations

```
let affiche_permutation table n =
    for i = 0 to n-1 do
        print_int (valeur_de table.(i)) ;
        print_char ‘ ‘
    done ;
    print_newline () ;;

let affiche_les_permutations n =
    let table = initialise n
    in
    try
        while true do
            affiche_permutation table n ;
            avance table n
        done
    with Fin -> () ;;
```

lignes qui viennent (ou, mieux, [9]) et qui devraient vous aider à *élucider* le fonctionnement de l'algorithme de Johnson.

Si on observe un peu son déroulement sur quelques exemples simples, $n = 2$, $n = 3$ ou $n = 4$, on voit apparaître des régularités qui aident à comprendre l'ordre dans lequel sont affichées les permutations, qui n'est pas du tout l'ordre lexicographique. J'oserais dire qu'il s'agit d'un ordre qui peut faire penser à une structure fractale : si on observe le déplacement de n, on s'aperçoit qu'il décrit un zigzag qui traverse la même structure (pour $n - 1$) qui subit récursivement une transformation analogue.

Mais tout cela est bien flou, aussi je vous propose d'observer sur la figure suivante, numéro 2, le résultat pour $n = 4$, en suivant des yeux le parcours du chiffre 4. Je regroupe volontairement les permutations en $3! = 6$ groupes de 4 ; regardez, et vous comprendrez…

1 2 3 4	4 1 3 2	3 1 2 4	4 3 2 1	2 3 1 4	4 2 1 3
1 2 4 3	1 4 3 2	3 1 4 2	3 4 2 1	2 3 4 1	2 4 1 3
1 4 2 3	1 3 4 2	3 4 1 2	3 2 4 1	2 4 3 1	2 1 4 3
4 1 2 3	1 3 2 4	4 3 1 2	3 2 1 4	4 2 3 1	2 1 3 4
(1 2 3)	(1 3 2)	(3 1 2)	(3 2 1)	(2 3 1)	(2 1 3)

1 2 3	3 2 1
1 3 2	2 3 1
3 1 2	2 1 3

Figure 2: Élucidation de l'algorithme de Johnson

Application au calcul direct d'une permutation

Une fois qu'on a compris comment fonctionne l'algorithme de Johnson, on s'aperçoit que l'on peut programmer directement le calcul de la k-ième permutation dans la série de Johnson correspondant à un n fixé.
En effet, le numéro du bloc de n permutations auquel appartient celle qui nous intéresse est clairement $1+\lfloor\frac{k-1}{n}\rfloor$, et le numéro de la ligne dans ce bloc est bien sûr $1+(k-1)$ `mod` n. Dans ce bloc, et sur cette ligne, n est sur la diagonale (quand le bloc est de numéro pair) ou l'antidiagonale (quand le bloc est de numéro impair). Voilà donc placé l'entier n. Et, on l'aura deviné, un appel récursif place l'entier $n-1$, puis le suivant...
On a donc ainsi un algorithme (traduit en Caml dans le programme 14) qui écrit directement la k-ième permutation de S_n dans la série de Johnson. On vérifie sans difficulté qu'elle tourne au pire en $O(n^2)$, le placement de chaque entier nécessitant au plus un parcours du tableau entier.

Programme 14 Calcul d'une permutation particulière

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

let permutation n k =
    let table = make_vect n 0
    in
    let rec installe d p saut =
        if table.(d) = 0 then
            if saut = 0 then table.(d) <- p
            else installe (d+1) p (saut-1)
        else installe (d+1) p saut
    in
    let rec place p k =
        let bloc = 1 + (k-1)/p
        and ligne = 1 + (k-1) mod p
        in
        installe 0 p (if impair bloc then p-ligne else ligne-1) ;
        if p>1 then place (p-1) bloc
    in
    place n k ;
    table ;;
```

Arithmétique et associativité des opérateurs (merci à D. Monasse)

Le parseur qui avait été écrit dans la lettre numéro 2 ne gérait pas convenablement l'associativité des opérations arithmétiques élémentaires : s'il est raisonnable de considérer l'élévation à la puissance comme associant à droite (`2^3^4` doit désigner 2^{3^4}), en revanche la soustraction et la division doivent associer à gauche : `2-3-4` représente $(2-3)-4$, et non pas $2-(3-4)$.

Le problème est plus ardu qu'il n'y paraît au premier abord : je vous invite à chercher une solution par vous-même, avant de lire la solution suivante.
Je ne fournis ici que le code du parseur lui-même, mais vous trouverez dans les sources Caml l'intégralité du code ainsi corrigé.

Programme 15 Un parseur qui gère bien l’associativité des opérations

```
exception Syntax_error ;;

let rec parseur_E pile flot =
    let e = parseur_F [] flot
    in
    match flot with
        | [< ’Plus >] -> parseur_E ((e,Plus) :: pile) flot
        | [< ’Moins >] -> parseur_E ((e,Moins) :: pile) flot
        | [< >] -> construit_E pile e
and construit_E pile e = match pile with
    | [] -> e
    | (f,op) :: queue -> match op with
                            | Plus -> Somme(construit_E queue f,e)
                            | Moins -> Différence(construit_E queue f,e)
and parseur_F pile flot =
    let e = parseur_G flot
    in
    match flot with
        | [< ’Multiplie >] -> parseur_F ((e,Multiplie) :: pile) flot
        | [< ’Divise >] -> parseur_F ((e,Divise) :: pile) flot
        | [< >] -> construit_F pile e
and construit_F pile e = match pile with
    | [] -> e
    | (f,op) :: queue -> match op with
                            | Multiplie -> Produit(construit_F queue f,e)
                            | Divise -> Quotient(construit_F queue f,e)
and parseur_G flot = match flot with
    | [< parseur_H h ; parseur_G’ g’ >] -> match g’ with
                                            | [< ’Élévation(_,e) >] -> Élévation(h,e)
                                            | [< >] -> h
    | [< >] -> raise Syntax_error
and parseur_G’ flot = match flot with
    | [< ’Puissance ; parseur_G g >] -> [< ’Élévation(N(0),g) >]
    | [< >] -> [< >]
and parseur_H flot = match flot with
    | [< ’Moins ; parseur_I i >] -> Opposé(i)
    | [< parseur_I i >] -> i
    | [< >] -> raise Syntax_error
and parseur_I flot = match flot with
    | [< ’ParenthèseGauche ; (parseur_E []) e ; ’ParenthèseDroite >] -> e
    | [< ’Entier(n) >] -> N(n)
    | [< >] -> raise Syntax_error ;;

let parseur s = parseur_E [] (lexeur (stream_of_string s)) ;;
```

Références

[1] J. Borwein and P. Borwein. *Pi and the AGM, A study in Analytic Number Theory and Computational Complexity.* Canadian Mathematical Society series of monographs and advanced texts. Wiley-Interscience, 1987.

[2] J. Borwein and P. Borwein. *Ramanujan revisited.* Academic Press, San Diego, CA, 1988.

[3] J. Borwein and P. Borwein. Class number three ramanujan type series for $1/\pi$. *Journal of Computational and Applied Mathematics*, (46): 281–290, 1993.

[4] D. Chudnovski and G. Chudnovski. *Ramanujan revisited.* Academic Press, San Diego, CA, 1988.

[5] G. Hardy. *Ramanujan's Collected Papers.* Chelsea, New York, 1962.

[6] V. Ménissier-Morain. The caml numbers reference manual. Technical Report 141, INRIA, juillet 1992.

[7] S. Ramanujan. Modular equations and approximations to π. *Quart. Journal of Math. Oxford*, (45): 350–372, 1914.

[8] Raymond Seroul. *Le petit livre de TEX.* InterÉditions, Paris, 1989.

[9] Raymond Seroul. *Math-info, informatique pour mathématiciens.* InterÉditions, Paris, 1995.