Calcul formel

DURAND Ulysse

Les expressions ou morceaux de code Ocaml seront en gras, les chaines de caractére seront soulignées.

Le but est de faire ce qui peut s'apparenter à un logiciel de calcul formel en caml, il doit pouvoir évaluer des fonctions, les dériver, et pouvoir afficher leurs expressions en LaTeX.
Un objectif est aussi de pouvoir renseigner les expressions des fonctions à traiter en LaTeX. On devrait à la fin avoir ce résultat :

```
let fonctiona = "\frac{x+y}{ln(x-y)}";;
let fonctionb = derive "x" fonctiona;;
print_float (evalueb [|3. ; 1.|]);;print_new_line() ;
print_string (affiche fonctionb);;print_new_line() ;
(*DOIT AFFICHER*)
-1.67935843
\frac{1}{ln(x-y)}-\frac{x+y}{(x-y)(ln(x-y))^2}
```

1 Représentation des fonctions

Nous traiterons d'expressions algébriques représentées par des arbres d'expression.

l'arbre suivant représente l'expression algébrique $\frac{x+3}{-y}$:

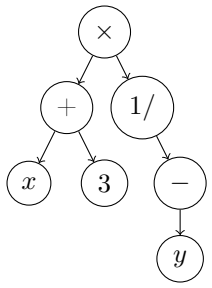


Figure 1: Exemple d'arbre d'expression

On discernera deux types de feuilles : les variables et les constantes Et les noeuds sont de la forme (une operation, des fils)
d'où une telle implémentation en Caml :

```
type operation = {nbvar : int ;affichage : string ; evaluation : float array->float
    ; derive : deriv}
and expression =
    C of float |
    V of int |
```

```
    F of operation * (expression array)
and deriv = NonDeriv | Deriv of ((expression array)->(expression -> expression) ->
    expression);;
```

Les variables sont caractérisées par un entier ($x_i, i \in \mathbb{N}$),
nous allons détailler les caractéristiques **affichage**, **evaluation** et **derive** des opérations et le type **deriv**.

2 affichage

Construisons une fonction **affiche**: **expression -> string**, telle que si a est l'arbre d'expression Figure1, alors **affiche a** retourne `\frac{x+3}{-y}`
Pour expliciter comment afficher une expression, nous allons, à chaque opération, associer un code dans un langage inventé (nous allons l'appeler langage d'affichage d'expression), ce code étant sous forme d'une chaîne de caractères, il décrit la forme de l'affichage d'une opération.

2.1 le langage d'affichage d'expression

Pour comprendre la construction de ce langage inventé, voici des exemples. Avec l'opération plus, on veut que **affichage (F(plus,[|a ; b|])) = (affichage a)^"+"^(affichage b)**.
Alors voici le contenu de plus.affichage : %|0;%+%|1;%, on dit alors de retourner (affichage du fils 0) + (affichage du fils 1).

Pour l'opération det, on veut que **affichage (F(det,[|x0,x1,x2|])) = "det("^(affichage x0)^","^(affichage x1)^","^(affichage x2)^")"**.
Alors voici, avec **det.nbvar=3**, le contenu de det.affichage : det(%,|a;%) (, le délilmiteur, et a comme all pour all variables)

On peut imaginer une opérations qui doit s'afficher ainsi : ope(x3;x4;x5;...;x10) alors on aura **ope.affichage = "ope%;|3-11;%"**
ou bien on peut imaginer l'opération s'affichant ainsi : ope(x0;x1;...;x9 / x15,x16,x17,x20,x21,...,x(n-1)) où n est ope.nbvar, alors voici le contenu de ope.affichage : ope(%;|-10;% / %,|15-18;20-;%)

2.2 Compilation de ce langage

Comment faire comprend à Caml ce langage ?
Nous allons utiliser un type d'automate particulier qui nous permettra de reconnaitre un tel langage, c'est ce qui sera appelé ici un automate deterministe qui écrit avec mémoire. Il écrit, c'est à dire qu'il reconnaît des mots mais renvoit aussi une chaine de caractéres. On peut donc associer à un tel automate une fonction (ce sera $\eta^\star(i, mem_i, \cdot)$, explications dans la section 3)

Cet automate particulier sera associé à une fonction **evaluelatex** prenant en paramétres le nombre de variables de l'opération, une fonction rendv qui elle prend un entier i en entrée et qui lui associe le rendu de la variable i, et une chaîne de caractères : notre code dans notre langage pour notre expression. **evaluelatex** retournera l'affichage voulu pour notre opération. Voici le typage de **evaluelatex** : **int -> (int->char list) -> char list -> char list**
Par exemple, **evaluelatex ope.nbvar rendv "ope(%;|-3%)" = "ope("^(rendv 0)^"(rendv 1)"^(rendv 2)^")"** (nous confondrons ici string et char list par souci de simplicitée.)
Les détails de l'automate et de sa fonction, **evaluelatex**, associée, seront explicités dans la section 3.

3 De quoi faire la fonction evaluelatex

3.1 Automates utiles

En caml, voilà une manière d'implémenter les automates :

```
type ('q, 'sig) automate = ('q -> bool)*('q -> bool)*(q'*sig'*('q list));;
(*('q,'ssig) automate = (i,f,delta) correspond a un automate ('q,'sig,i,f,delta)*)
```

3.1.1 Automate déterministe qui écrit

Definition 3.1 (Automate déterministe qui écrit)**.** Ce type d'automate permet de transformer un mot en un autre.

$$\mathcal{A} = (Q, \Sigma_1, \Sigma_2, i, F, \delta, \eta)$$
$$i \in Q, F \in \mathcal{P}(Q)$$

$$\delta : Q \times \Sigma_1 \to Q$$
$$\eta : Q \times \Sigma_1 \to (\Sigma_2)^\star$$

$$\begin{aligned}\delta^\star : Q \times \Sigma_1^\star &\to Q\\ (q, \epsilon) &\mapsto q\\ (q, l.m) &\mapsto \delta^\star(\delta(q,l), m), \text{ avec } l \in \Sigma_1\end{aligned}$$

$$\begin{aligned}\eta^\star : Q \times \Sigma_1^\star &\to (\Sigma_2)^\star\\ (q, \epsilon) &\mapsto \epsilon'\\ (q, l.m) &\mapsto \eta(q,l).\eta^\star(\delta(q,l), m), \text{ avec } l \in \Sigma_1\end{aligned}$$

Implementation en Caml :

```
type ('q, 'sig1, 'sig2) automatequiecrit = ('q*('sig2 list), 'sig1 ) automate;;
```

('q,'sig1,'sig2) automatequiecrit = (i,f,g) correspond à un automate $('q,'sig1,'sig2, i, f, delta, eta)$. Supposons g sous la forme $g((q,m),l) = (q', p :: m)$, alors $delta(q,l) = q'$ et $eta(q,l) = p$

3.1.2 Automate qui écrit avec une mémoire

textvf

Definition 3.2 (Automate qui écrit avec mémoire)**.** Ce type d'automate n'a pas de d'intérêt théorique vu que la mémoire est finie, mais a un intérêt pratique.

$$\mathcal{A} = (Q, \Sigma_1, \Sigma_2, S_m, i, mem_i, F, \delta, \eta)$$
$$Q, \Sigma_1, \Sigma_2, i, F \text{ restent inchangés}$$
$$mem_i \text{est la mémoire initiale}$$

$$\delta : Q \times S_m \times \Sigma_1 \to Q \times S_m$$
$$\eta : Q \times S_m \times \Sigma_1 \to (\Sigma_2)^\star$$

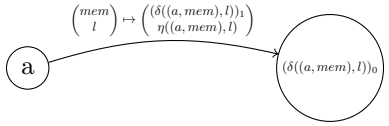
$$\begin{aligned}
\delta^\star : Q \times S_m \times \Sigma_1^\star &\to Q \\
(q, M, \epsilon) &\mapsto q \\
(q, M, l.m) &\mapsto \delta^\star(\delta(q, M, l), m), \text{ avec } l \in \Sigma \\
\eta^\star : Q \times S_m \times \Sigma_1^\star &\to (\Sigma_2)^\star \\
(q, M, \epsilon) &\mapsto \epsilon' \\
(q, M, l.m) &\mapsto \eta(q, M, l).\eta^\star(\delta(q, M, l), m), \text{ avec } l \in \Sigma_1
\end{aligned}$$

Il s'agit en fait d'un automate qui écrit avec $Q' = Q \times S_m$
Implementation en Ocaml :

```
type ('q, 'sm, 'sig1, 'sig2) automatequiecritavecmemoire = ('q*'sm,'sig1,'sig2)
    automatequiecrit
```

('q,'sm,'sig1,sig2) automatequiecritavecmemoire : (i,f,g) correspond à un automate qui ecrit avec mémoire $('q, sig1, sig2, sm, i, f, delta, eta)$. Supposons g sous la forme $g((q, m, mem), l) = ((q', mem'), p@m)$, alors $delta((q, mem), l) = (q', mem')$ et $eta((q', mem'), l) = p$

Pour les représenter graphiquement, nous ferons comme les automates, mais avec des annotation sur les arêtes différentes :



3.2 Et dans notre cas

Voici l'automate associé à la fonction **evaluelatex n renv s** (correspond à $\delta^\star(i, mem_i, s)$ pour l'automate suivant):
avec n un entier et $N = \{$"0","1","2","3","4","5","6","7","8","9"$\}$

Si A est cet automate qui écrit, **n = 6**, **rendv a** retourne x_a
fung défini ainsi en Caml :

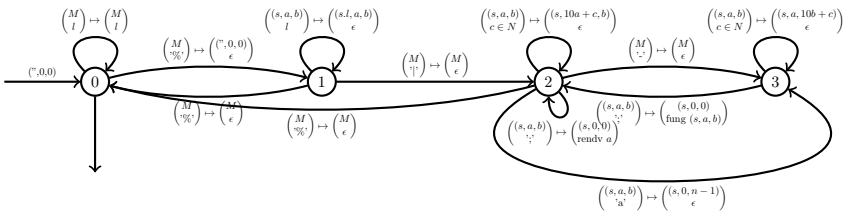


Figure 2: Automate de **evaluelatex**

```
let rec fung (s,a,b) =
    if a > b then [] else
    if a = b then rendv a
    else (rendv a)@s@(fung (s,a+1,b)) in
```

Par exemple, **fung (s,0,3) = (rendv 0)^s^(rendv 1)^s^(rendv 2)^s^(rendv 3)**

$\eta^\star(0,(",0,0),$ `\frac{{%|0;%+%|15;%}^{(%*|1-4;%)}}{%+|a;%}` $) =$
`\frac{{x_0+x_15}^{x_1*x_2*x_3}}{x_0+x_1+x_2+x_3+x_4+x_5}`

Voila finalement la fonction **affiche**que l'on voulait, en Caml :

```
let rec affiche f = match f with
    |C(x) -> string_of_float x
    |V(i) -> "{x_{"^(string_of_int i)^"}}"
    |F(g,va) -> evaluelatex (g.nbvar) (fun a -> List.rev (explode (affiche va.(a))))
        (g.affichage);;
```

4 Evaluation

Construisions une fonction evalue qui prend en entrée une expression et un point en lequel l'évaluer : **expression -> float array -> float**, telle que si a est l'arbre d'expression Figure1, v le point en lequel évaluer a, alors **evalue a [|2.;3.|]** retourne **-1.6666**
L'évaluation est alors plutôt simple, se faisant par induction.
Pour une constante, on retourne la constante,
pour la variable x_i on retourne **v.(i)**,
et pour une opération sur plusieurs expressions, on retourne la fonction d'évaluation de l'opération appliquée à l'évaluation de chaque fils.

d'où une telle implémentation en Caml :

```
let rec evalue f v = match f with
    |C(x) -> x
    |V(i) -> v.(i)
    |F(g,fa) -> g.evaluation (Array.map (fun unef -> evalue unef v) fa );;
```

5 Dérivation

Pour renseigner pour une opération si elle est dérivable et comment évaluer sa dérivée, nous allons utiliser le type dérivation défini en première page : soit l'opération n'est pas dérivable, soit elle l'est et alors on donne une expression de sa dérivée.
Pour l'expression de sa dérivée nous aurons besoin d'utiliser la fonction **derive** à l'intérieur, qui sera alors renseignée dans Deriv (c'est le **expression -> expression**). Le **expression array** correspond aux fils de la fonction qui auront leur rôle dans l'expression de la dérivée.

Alors on a **Derive** prenant en paramètres **ar** et **d**, **ar** étant les fils de l'opération et **d** étant la fonction dérivée.

Un exemple sur l'opération plus,
plus.derive = Deriv(fun ar d -> F(plus,[|d ar.(0);d ar.(1)|])), on retrouve $(u+v)' = u' + v'$
Un deuxième exemple, sur l'opération de multiplication,
fois.derive = Deriv(fun ar d -> F(plus,[| F(fois,[|d ar.(0);ar.(1)|]) ; F(fois,[|ar.(0);d ar.(1)|]) |])),
on retrouve $(u*v)' = u' * v + u * v'$

Pour ce qui est de la fonction **derive**, il y a plusieurs fonctions de dérivation que l'on pourra renseigner dans le type Derive de notre opération, ce sont les dérivées par rapport aux différentes variables.
C'est pourquoi nous allons prendre en arguments un entier **k** et une expression **f** pour retourner l'expression de la dérivée selon la k-ième variable de **f**.
Si **f** est la variable x_i, alors on retourne $\delta_{k,i}$.
Si **f** est une constante, alors on retourne le flottant **0**.
Si **f** est une opération **g** dont les fils sont **va** et que **g.derive** est **NonDeriv**, on renvoit une erreur mais si c'est **Deriv(laf)** alors tout est renseigné dans **laf**, il n'y a qu'a retourner **laf va (derive k)**
Voici l'expression de **derive** en Caml :

```
let rec derive k f = match f with
        |C(x)->C(0.);
        |V(i)-> if i=k then C(1.) else C(0.);
        |F(g,va) -> match g.derive with
                |NonDeriv->failwith "Fonction non derivable !";
                |Deriv(laf)->laf va (derive k);;
```

6 Parser

Le plus dur reste à faire : transformer une expression **entree** sous forme LaTeXen une expression sous forme d'un arbre d'expression. L'idée est de faire du pattern matching, mais sur une chaîne de caractères. En effet, on voudrait une application recursive **parselatex** telle que par exemple
parselatex x+y retourne **F(plus,[|parselatex x ; parselatex y |])**
On reconnait un pattern matching.
Pour se faire, pourquoi ne pas réutiliser notre langage défini section 2.1, on aurait alors un pattern %|0;%+%|1;%.

L'idée est la suivante : pour chaque pattern **p**, utiliser une fonction associée à un automate que nous expliciterons plus loin **automate_de_pattern p** qui retournerait un automate qui écrit reconnaissant **entree** et ressortant les chaines de caractères à la place des x_i habituels.

Pour le pattern du plus et pour banane+pomme comme valeur pour **entree**, l'automate du pattern de plus reconnaitrait **entree** et ressortirait un tableau **[|"banane","pomme"|]** auquel il ne resterait qu'à associer **F(plus,[|parselatex "banane" ; parselatex "pomme"|])**
La difficultée réside en la construction d'un tel automate.
Un détail : plusieurs pattern pourraient être reconnus, alors on utilisera seulement la sortie de l'automate du premier de ces pattern dans notre liste de pattern. Ils seront ainsi priorisés.

La tache sera découpée en deux étapes, il faut construire l'automate à partir d'un pattern dans notre langage, puis appliquer cette construction à du latex comme décrit précedement. Construisons alors les fonctions **automate_de_pattern** et **parselatex**.

6.1 automate_de_pattern

Comme dit précedemment, **automate_de_pattern** sera la fonction associée à un automate qui ecrit ressortant l'automate voulu.
Ce sera un automate qui écrit mais avec Σ' étant l'enemble des automates déterministes, la concaténation sur cet alphabet étant un opérateur **concat_automate** que nous définirons bientôt.

List of Figures