Préparation aux oraux Centrale/Mines-Ponts n°2

Calculatrice

L'objectif de ce TP est de réaliser une calculatrice. On s'intéressera à plusieurs aspects de cette réalisation : la manipulation de polynômes, puis la gestion d'expressions mathématiques, et enfin la représentation graphique de fonctions.

Les questions sont annotées de différents pictogrammes pour signifier ce qui est attendu de l'élève :
- ✍ signifie une question à rédiger sur le compte-rendu (il sera rendu en même temps que le code) ;
- 🗣 signifie une question (à préparer et) à présenter à l'examinateur ;
- 💻 signifie une question où du code devra être écrit.

L'évaluation des compétences de l'étudiant se basera principalement sur les critères suivants :
- la qualité et la clarté du code ;
- la qualité et la clarté du compte-rendu ;
- la qualité des interactions avec l'examinateur (en particulier sur les connaissances du cours) ;
- l'avancement dans le sujet.

QUELQUES REMARQUES . . .

Ce sujet est un *vrai* Mines-Ponts que j'ai pu passer avec Victor Aribaud et Nathan Roos en 2023. Il a été légèrement modifié (questions Q4, Q20 et Q21 en particulier).

Il se divise en trois parties relativement indépendantes :
Section I. Manipulations de polynômes en SQL (~45 min)
Section II. Réalisation d'un *parser* d'expression mathématiques (~1h30)
Section III. Création d'une calculatrice graphique (~45 min).

Dans le cadre de la préparation aux oraux, ce sujet est assez bien pour plusieurs raisons :
- on révise SQL en faisant quelques manipulations avec des opérations peu courantes ;
- on réalise un *parser* dans un cadre assez guidé ;
- tout le sujet peut être fait en récursif (et on préfère le récursif : c'est OCaml quand même !) ;
- on se place dans un cadre auquel on n'est pas habitué (utilisation du module `Graphics` par exemple) ;
- on peut gagner beaucoup de temps en regardant la documentation sur *Zeal* (l'outil pour voir la documentation : il est sur les machines du lycée, et sur les machines de TP à Centrale et à Mines-Ponts) ;
- il est intéressent à faire (on touche à plusieurs aspects du programme d'info, dans un but assez précis : réaliser une calculatrice).

À la fin des différentes parties de ce TP, vous trouverez des cadres comme celui-ci. Ils permettent de voir les erreurs « classiques » (celles qui sont remontées pendant le TP du vendredi 07 juin en tout cas).

I. Polynômes en SQL.

Dans le fichier base de données `polynomes.db`, vous trouverez une seule table `monome` contenant trois colonnes : `facteur`, `exposant` et `poly`. L'utilisation de l'interpréteur SQLite3 est détaillé dans l'Annexe A. Afin de représenter un polynômes P, on le décompose comme somme de monômes $a \times X^k$. Chaque ligne de la table contient un monôme, où a est la valeur de `facteur`, et k est la valeur

de `exposant`. La valeur de `poly`, un entier, permet de déterminer quel polynôme P est représenté, avec un identifiant commun.

Q1. Quelle est la représentation des polynômes X et $X^2 - 1$ dans la table ?

RÉPONSE.

POLY	EXPOSANT	FACTEUR
1	1	1
2	0	-1
2	2	1

Q2. Lister les polynômes présents dans la table `monome`.

RÉPONSE. Les polynômes sont $X^3 - 2X + 1$ et $8X^4 + 0{,}5\ X^2 + 3{,}14$.

Remarquons que la représentation décrite ci-dessus n'est pas unique : nous pouvons représenter $2X$ comme $2 \times X$ (un unique monôme) ou comme $X + X$ (somme de deux monômes). On dira que la représentation d'un polynôme est *acceptable* s'il n'y a pas plusieurs monômes de même exposant. Cette représentation acceptable est unique.

Q3. Écrire une requête SQL permettant de trouver les polynômes dont la représentation n'est pas acceptable. Proposer une modification de la structure de données imposant que la représentation d'un polynôme soit acceptable.

RÉPONSE.

```sql
select poly from monome group by poly
having count(distinct(exposant)) <> count(exposant);
```

Là, on ne demande pas une modification sous la forme d'une requête SQL. On peut modifier en ajoutant une *clé primaire* sur le couple (poly, exposant). Ceci permet l'unicité, pour chaque polynôme, du seul monôme avec un certain exposant.

Q4. Écrire une requête SQL permettant de calculer $P + Q$ où P et Q sont deux polynômes représenté dans la table `monome`. La requête retournera deux colonnes `facteur` et `exposant` représentant le polynôme $P + Q$ de manière acceptable.

RÉPONSE. Cette question est un échec pédagogique. Soit les réponses sont trop « simples » soit trop peu généralisable (somme $P + P$ par exemple). J'annule cette question.

Q5. Comme pour Q4, écrire une requête SQL permettant de calculer $P \times Q$ où P et Q sont deux polynômes représentés dans la table `monome`.

RÉPONSE. On veut réaliser un *produit de Cauchy* en SQL. Si P et Q sont deux polynômes que l'on peut écrire comme $P = \sum_{i\in I} a_i X^i$ et $Q = \sum_{j\in J} b_j X^j$, alors

$$P \times Q = \sum_{k\in K} c_k X^k \quad \text{avec } c_k = \sum_{i+j=k} a_i b_j.$$

```sql
select a.exposant + b.exposant as k, sum(a.facteur * b.facteur)
from monome a join monome b on a.poly = 1 and b.poly = 2 group by k;
```

QUELQUES REMARQUES ...

On peut perdre beaucoup de temps dans cette partie si on n'est pas efficace en SQL. Et le temps en TP, c'est précieux. En TP, cette partie a pu durer plus d'1h30, mais ce n'est qu'un quart du sujet !

Déjà, il est important de ne pas perdre de temps sur des bêtises : comment lancer SQLite3, comme fonctionne `GROUP BY`, quelle différence entre `WHERE` et `HAVING`, pourquoi ma requête SQL ne veut pas « partir » (il y a un « ; » à la fin des requêtes SQL), *etc.*

Pour certaines de ces questions, il suffit de lire le sujet en entier. **Lisez le sujet !**

Il faut connaître quelques mots en SQL (les fonctions d'agrégation) : `SUM`, `DISTINCT`, `COUNT` par exemple. Ces fonctions fonctionnent avec `GROUP BY`. On peut même les composer.

Les jointures (`JOIN ... ON`), c'est bien. Ça marche même sur la même table. Ça permet de simplifier grandement les requêtes SQL.

II. Expressions mathématiques en OCaml.

Dans cette partie, on cherche à effectuer des opérations sur des expressions mathématiques (évaluation, simplification, dérivation), puis à convertir une chaîne de caractères en une expression (et inversement). On définit le type `expression` comme dans Code 1.

```
type expression =
| Var of char
| Number of int
| Plus of expression * expression
| Mult of expression * expression
| Neg of expression
| Inv of expression
```

Code 1. Représentation en OCaml d'une expression mathématique.

II.1. Représentation d'une expression.

Q6. Discuter de comment représenter une expression mathématique à l'aide d'une expression de type `expression`. On s'intéressera particulièrement à comment représenter une *différence* ou un *quotient*. Représenter les expressions ci-dessous en OCaml :

$$\frac{x^3+x}{x^2}, \qquad 2x-1, \qquad \text{et} \qquad x^2+y^2-1.$$

RÉPONSE. Dans cette question, on s'intéresse à comment est représentée une expression mathématique en OCaml, avec deux opérations en particulier :
- a/b sera représenté par `Mult`$(a,$ `Inv` $b)$ par exemple ;
- $a-b$ sera représenté par `Plus`$(a,$ `Neg` $b)$ par exemple.

On peut également s'intéresser à la représentation de $a \times b \times c$. On le représente en OCaml avec l'expression `Mult`$(a,$ `Mult`$(b,c))$ ou `Mult`$($`Mult`$(a,b),c)$? De même avec l'addition.

Représentation des formules :

```
Mult(
  Plus(
```

```
    Mult(Var 'x', Mult(Var 'x', Var 'x')), (* x^3 *)
    Var 'x'
  ),
  Mult(Var 'x', Var 'x') (* x^2 *)
)

Plus(Mult(Number 2, Var 'x'), Neg(Number 1))

Plus(
  Mult(Var 'x', Var 'x'),
  Plus(
    Mult(Var 'y', Var 'y'),
    Neg(Number 1)
  )
)
```

Q7. Étant donnée une variable, calculer la dérivée d'une expression par rapport à cette variable. On ne simplifiera pas les expressions retournées, c'est l'objet de Q9.

RÉPONSE. On implémente les règles de dérivation usuelles. On pourra tester son comportement sur les expressions définies avant.

```
let rec derive (v: char) = function
| Var y when y = v -> Number 1
| Var _ -> Number 0
| Number _ -> Number 0
| Plus(a, b) -> Plus(derive v a, derive v b)
| Mult(a, b) -> Plus(Mult(derive v a, b), Mult(a, derive v b))
| Neg a -> Neg(derive v a)
| Inv a -> Mult(Neg(derive v a), Inv(Mult(a,a)))
```

Q8. On souhaite évaluer une expression. On souhaite donc assigner à une variable un flottant (sa valeur). Quelle représentation utiliser pour assigner à chaque variable leur valeur ? Cette représentation n'a pas besoin d'être efficace : le nombre de variables est faible, et en général moins de 2. Écrire une fonction d'évaluation d'une expression.

RÉPONSE. Pour l'évaluation d'une expression, on peut utiliser plusieurs structures de données :

- une table de hachage ;
- un arbre binaire de recherche étiqueté sur les noms de variables ;
- un arbre rouge-noir étiqueté sur les noms de variables ;
- ou tout simplement une liste d'association (une liste de couples (v, x) où v est une variable et x sa valeur).

On préfère cette dernière solution pour plusieurs points (les autres sont également très bien, mais il faut le justifier avec l'examinateur, qui pourra vous conseillera une représentation plus simple pour éviter de vous faire perdre trop de temps) : elle est simple, on peut faire du récursif avec elle (c'est *un peu* plus facile qu'avec une table de hachage), et comme le nombre de variables est assez faible, elle ne prend pas tant que ça d'espace mémoire. De plus, la fonction `List.assoc` existe, et elle est très utile (même si elle peut être codée assez facilement).

```
type assignment = (char * float) list
```

```
let rec evaluate (a: assignment) = function
| Var y -> List.assoc y a
| Number i -> float_of_int i
| Plus(u,v) -> (evaluate a u) +. (evaluate a v)
| Mult(u,v) -> (evaluate a u) *. (evaluate a v)
| Neg u -> -.(evaluate a u)
| Inv u -> 1. /. (evaluate a u)
```

Q9. On souhaite simplifier une expression, comme celles obtenues à la suite de Q7. Par exemple, on peut simplifier la multiplication par zéro, par un, l'addition d'un zéro, *etc.* Proposer des règles de simplification d'une expression et les implémenter.

RÉPONSE. On réalise une fonction qui réalise une étape de simplification, puis on itère l'application de cette fonction jusqu'à l'obtention d'un point fixe. On n'a pas besoin d'implémenter *autant* de règles de simplifications.

```
let rec simplify_step = function
| Mult(Number 0, x) | Mult(x, Number 0) -> Number 0
| Mult(Number 1, x) | Mult(x, Number 1) -> x
| Inv(Inv x) -> x
| Neg(Neg x) -> x
| Plus(Number 0, x) | Plus(x, Number 0) -> x
| Inv(Number 1) -> Number 1
| Plus(Number i, Number j) -> Number(i+j)
| Neg(Number i) -> Number (-i)
| Mult(Number i, Number j) -> Number(i*j)
| Mult(Mult(a,b), c) -> Mult(a, Mult(b,c))
| Plus(Plus(a,b), c) -> Plus(a, Plus(b,c))

| Plus(a,b) -> Plus(simplify_step a, simplify_step b)
| Mult(a,b) -> Mult(simplify_step a, simplify_step b)
| Inv a -> Inv(simplify_step a)
| Neg a -> Neg(simplify_step a)
| x -> x

let rec simplify (e: expression): expression =
  let e' = simplify_step e in
  if e' <> e then simplify e'
  else e
```

QUELQUES REMARQUES ...

Dans cette sous-section, ça va : on manipule des arbres syntaxiques comme d'habitude, on fait du récursif, *etc.* La partie discussion ne pose pas trop de problèmes, mais connaître un peu de la librairie standard OCaml permet d'aider l'implémentation suite à cette discussion. Par exemple, en Q8, on pourrait facilement discuter l'utilisation d'un arbre rouge-noir pour une structure de données efficace, mais connaître le module `Hashtbl` ou la fonction `List.assoc` peut grandement guider ce choix. Connaître toute les fonctions du module `List`, non. Mais savoir qu'il existe quelques fonctions pour utiliser une liste d'association pour représenter une liaison clé-valeur, c'est très utile.

Également, en Q9, on réalise des simplifications. Pour cela, c'est facile de faire une fonction qui réalise une étape de simplification, et une autre qui applique cette fonction étape jusqu'un point fixe. Le faire dans une seule fonction est bien plus complexe et, on l'occurrence, pas plus utile.

Faire des fonctions simples, c'est important. D'ailleurs, le mot clé `function` est très utile pour réduire le code (un peu plus) : il y a beaucoup de *pattern matching*.

II.2. Transformations entre chaînes de caractères et expressions.

Dans cette sous-section, on s'intéresse à la transformation entre chaîne de caractère et expression.

On rappelle qu'une expression *bien parenthésée* est une séquence de symboles mathématiques où chaque ouverture de parenthèse est correctement appariée avec une fermeture correspondante. En d'autres termes, chaque parenthèse ouvrante doit être suivie d'une parenthèse fermante dans un ordre correct, et chaque paire de parenthèses doit contenir une expression valide entre elles.

✍ **Q10.** Donner une grammaire non contextuelle $\mathcal{G}$ reconnaissant les expressions mathématiques bien parenthésées.

RÉPONSE. Dans l'objectif de simplifier le *parser*, on utilise la grammaire $\mathcal{G} = (V, \Gamma, \Pi, S)$ avec l'alphabet de terminaux $\Gamma = \{(,), \times, -, +, /, \mathtt{a}, \dots, \mathtt{z}, 0, \dots, 9\}$, pour ensemble alphabet de symboles non terminaux $V = \{S, N, V\}$, et pour règles de productions Π :

$$S \to (\ S \times S\ ) \mid (\ S + S\ ) \mid (\ -\ S\ ) \mid (\ 1\ /\ S\ ) \mid (\ N\ ) \mid (\ V\ )$$
$$N \to 0\ N \mid \cdots \mid 9\ N \mid 0 \mid \cdots \mid 9$$
$$V \to \mathtt{a} \mid \cdots \mid \mathtt{z}$$

Cette grammaire a pour avantage d'être non ambigüe. Ceci simplifiera l'implémentation en Q15. Il est important de préciser que `( x - y )` n'est pas dans le langage de la grammaire $\mathcal{G}$.

💻 **Q11.** Implémenter, en OCaml, une fonction `vers_chaine : expression -> string` transformant une expression en chaîne de caractère bien parenthésée la représentant.

RÉPONSE.

```
let rec vers_chaine = function
| Number i -> "(" ^ string_of_int i ^ ")"
| Var x -> "(" ^ String.make 1 x ^ ")"
| Plus(a,b) -> "(" ^ vers_chaine a ^ "+" ^ vers_chaine b ^ ")"
| Mult(a,b) -> "(" ^ vers_chaine a ^ "*" ^ vers_chaine b ^ ")"
| Inv(a) -> "(1/" ^ vers_chaine a ^ ")"
| Neg(a) -> "(-" ^ vers_chaine a ^")"
```

💻 **Q12.** Écrire une fonction `decoupe : string -> char list` qui transforme une chaîne de caractère en la liste des caractères qui la compose.

RÉPONSE. Avec une fonction comme ça, on peut comprendre que c'est important de connaître un peu la librairie standard (surtout le module `List`).

```
let rec decoupe (s: string): char list =
  let n = String.length s in
  List.init n (fun i -> s.[i]);;
```

Q13. Écrire une fonction `parse_numbers : char list -> expression * (char list)` qui, sur une entrée s, retourne le couple (e, r) où $s = e \cdot r$ et e est l'expression représentant le plus long préfixe non-vide de s ne contenant que des entiers de 0 à 9 (avec le variant `Number`, *c.f.* Code 1).

RÉPONSE.

```
let is_digit (c: char) = c >= '0' && c <= '9'

let rec extract_numbers = function
| x :: q when is_digit x ->
  let (u,v) = extract_numbers q in
  (x :: u, v)
| l -> ([], l)

let get_digit (c: char): int = int_of_char c - int_of_char '0'

let rec parse_numbers (l: char list): expression * (char list) =
  let (u, v) = extract_numbers l in
  let num = List.fold_left (fun acc x ->
    10 * acc + (get_digit x)) 0 u
  in (Number num, v);;
```

Q14. Expliquer brièvement pourquoi une expression bien parenthésée a, au plus, un préfixe bien parenthésé.

RÉPONSE. Dans notre cas (et vue la grammaire choisie), on ne peut pas avoir deux préfixes bien parenthésés. On n'a pas la de règle de la forme $S \to SS$ ce qui pourrait permettre d'avoir plus d'un préfixe bien parenthésé. Pour trouver ce préfixe, il suffit de commencer à la parenthèse ouvrante et de continuer jusqu'à la parenthèse fermante associée (qu'on peut trouver en comptant le nombre de parenthèse ouvrantes et le nombre de parenthèses fermantes sur un préfixe de la chaîne : si ces deux nombres sont égaux, on l'a trouvé).

Q15. Écrire une fonction `parse_expr : char list -> expression * (char list)` qui, sur une entrée s, retourne le couple (e, r) où $s = e \cdot r$ et e est l'unique préfixe de l'expression représentée dans la chaîne de caractères en entrée.

RÉPONSE.

```
let format_prefix (p: char list): char list =
  List.rev (List.tl (List.rev (List.tl p)))

let extract_prefix (l: char list): (char list) * (char list) =
  let rec aux (diff: int) (p: char list) = function
  | _ when diff < 0 -> failwith "Mal parenthésé"
  | l when diff = 0 && p <> [] -> (List.rev p, l)
  | '(' :: q -> aux (diff + 1) ('(' :: p) q
  | ')' :: q -> aux (diff - 1) (')' :: p) q
  |  x  :: q -> aux diff (x :: p) q
```

```
  | [] -> ([], [])

  in aux 0 [] l

let is_var (c: char) = c >= 'a' && c <= 'z'
```

Q16. Écrire une fonction `vers_expression : string -> expression` transformant la chaîne de caractères en l'expression quelle représente.

QUELQUES REMARQUES ...

Partie bien plus longue : réalise un *parser* d'expressions mathématiques. Le sujet est relativement bien guidé, sauf pour Q15, qui prend bien plus de temps, mais votre examinateur pourra vous aider à trouver ce qui pose problème. La méthode est simple : prendre son temps pour réfléchir avant de commencer à coder : comment est-ce que je procède globalement, quelle fonctions je peux avoir besoin, *etc.* L'examinateur aura plus envie de vous aider si vous avez clairement réfléchis au problème ; il ne vous aidera pas autant si vous êtes sous une montagne de code, de fonctions entremêlées... Ça reste un oral : l'objectif n'est pas *que* le code, c'est aussi les interactions avec l'examinateur.

Également, savoir se debugger, c'est un *must-have*. Si la fonction ne produit pas le bon résultat, il faut savoir mettre des « `print` » partout, et essayer de comprendre ce qui ne va pas. Par exemple, faire une fonction qui affiche une liste de caractère comme un seul mot, ça semble très utile.

III. Calculatrice graphique en OCaml.

L'objectif de cette partie est la représentation de courbes sur une fenêtre. Pour cela, nous utiliserons OCaml ainsi que le module `Graphics`. Les commandes permettant d'inclure le module `Graphics` lors de la compilation/exécution du code OCaml sont données en Annexe B. Pour afficher une fenêtre, on utilise le code ci-dessous.

```
open Graphics;

let _ = open_graph ""
```

Dans les fichiers associés à ce TP, vous trouverez la documentation du module `Graphics`. Elle pourra vous être utile dans la suite de ce TP.

Q17. Remplir la fenêtre en jaune en utilisant *uniquement* les fonctions `open_graph`, `size_x`, `size_y` et `plot`. Vous pourrez regarder la documentation de ces fonctions. On procédera, par exemple, en utilisant deux boucles `for`.

On suppose que l'on veuille « colorier » un point de la fenêtre ouverte précédemment aux coordonnées $(x, y) \in [x_1, x_2] \times [y_1, y_2]$. L'écran sera donc représenté par ses « limites » : x_1–x_2 pour l'axe x, et y_1–y_2 pour l'axe y. Pour avoir les coordonnées du pixel à modifier, on utilisera la formule :

$$\text{pix}(x, y) = \left(\texttt{size\_x()} \times \left\lfloor \frac{x - x_1}{x_2 - x_1} \right\rfloor, \texttt{size\_y()} \times \left\lfloor \frac{y - y_1}{y_2 - y_1} \right\rfloor \right)$$

Q18. On souhaite afficher le graphe d'une fonction représentée par son expression mathématique (de type `expression`). On supposera que la dérivée de cette fonction n'est pas trop importante sur la fenêtre considérée. Écrire, en OCaml, une fonction dont la signature est `dessine : float -> float -> float -> float -> expression -> unit` qui, lorsqu'elle est appelée de la forme `dessine x1 x2 y1 y2 e` trace la courbe de l'expression `e` sur la fenêtre avec les limites définies par les paramètres `x1`, `x2`, `y1`, `y2`.

Q19. On souhaite à présent afficher une courbe paramétrée. Écrire une fonction `dessine_param` dont les arguments sont, dans l'ordre `t1`, `t2`, `dt`, `x1`, `x2`, `y1`, `y2`, `ex`, et `ey`. Les arguments `x1`, `x2`, `y1` et `y2` définissent les limites de la fenêtre. Les arguments `t1` et `t2` définissent les limites de la courbe paramétrée, et `dt` sa résolution (vue comme le pas d'incrémentation). Les expressions `ex` et `ey` sont pour représenter les deux fonctions $x(t)$ et $y(t)$.

Par la suite, on ne donne pas les arguments des fonctions à coder. En effet, ils sont généralement les mêmes.

Q20. Écrire une fonction qui dessinera les axes x et y passant par l'origine $(0, 0)$. On vérifiera que chacun des axes soit dans les limites de l'écran.

Q21. Écrire une fonction `dessine_tang` qui dessine la droite tangente à une fonction en un point x. On pourra utiliser la fonction codée en Q7.

QUELQUES REMARQUES . . .

On finit calmement le TP. La seule *vraie* difficulté, c'est lire l'Annexe B et comprendre comment on compile du OCaml avec le module `Graphics`. Sinon, il suffit de réutiliser les fonctions définies avant (Q7 et Q8 notamment). Pour l'examinateur (et pour vous aussi), c'est assez facile de vérifier que vos fonctions . . . fonctionnent.

Également, on teste votre capacité à vous adapter : vous n'êtes pas habitués à utiliser `Graphics`, ou à faire de l'affichage à l'écran (sans MatplotLib en tout cas). Et, vu que la documentation est donnée, vous n'avez qu'à la lire, et comprendre ce qu'il faut faire.

Annexe A. Utilisation de SQLite3.

Pour cet exercice, on vous fournit un fichier `polynomes.db` qui stocke une base de données SQLite3. Pour pouvoir exécuter des commandes sur ce fichier on peut lancer la commande `sqlite3 polynomes.db` depuis le dossier où se trouve ce fichier. Cette commande lance un interpréteur SQLite3. On peut par exemple taper la commande suivante pour avoir le nombre d'enregistrements dans la table `monome` :

```
SELECT COUNT(*) FROM monome ;
```

Pour avoir un affichage plus lisible avec SQLite3, on pourra utiliser les deux commandes suivantes (à copier directement dans le shell SQLite3) :

```
.header on
.mode column
```

Annexe B. Compilation OCaml avec `Graphics`.

Pour charger ce module, on pourra utiliser la commande

```
ocaml -I /home/candidat/.opam/default/lib/graphics graphics.cma
```

pour exécuter, et écrire la ligne `#load "graphics.cma";;` (ou `open Graphics;;`) dans un fichier d'extension `.ml`.

REMARQUE. La commande sera à adapter en fonction du nom d'utilisateur. On remplacera, dans la commande, le « `candidat` » par le nom d'utilisateur de la session Linux actuelle.