Université de Paris
M1 Informatique

Programmation Fonctionnelle Avancée
Année 2022-2023

Projet

Un logiciel de calcul formel

`Sujet version 1`

1 Introduction

Vous souhaitez aider votre petite sœur ou votre petit frère à passer l'épreuve de mathématiques du bac, et en particulier l'exercice d'étude de fonction. Mais traiter à la main ce genre d'exercice, c'est galère (c'est quoi déjà la dérivée de la tangente ? ou le cosinus d'une somme ?). Vous allez réaliser en OCaml un logiciel de calcul formel, capable de simplifier une expression algébrique, de la dériver, d'intégrer certaines formules simples, etc. Vous pourrez vous inspirer des logiciels existants tels que Maple ou Mathematica (logiciels propriétaires) ou encore Maxima (logiciel libre, version graphique disponible sous le nom `wxmaxima`).

2 Expressions algebriques

Nous vous fournissons un fichier `syntax.ml` proposant le type suivant pour les expressions algébriques:

```
type op0 = Pi | E
type op1 = Sqrt | Exp | Log | Sin | Cos | Tan | ASin | ACos | ATan | UMinus
type op2 = Plus | Mult | Minus | Div | Expo

type nums = int

type expr =
  | Num of nums
  | Var of string
  | App0 of op0
  | App1 of op1 * expr
  | App2 of op2 * expr * expr
```

Par exemple, $x + \pi\sqrt{3}$ sera représenté par:

```
App2 (Plus, Var "x", App2 (Mult, App0 Pi, App1 (Sqrt, Num 3)))
```

Cet exemple peut aussi être définie via `Syntax.Light.(Var "x" + pi * sqrt (Num 3))` dans le code OCaml, grâce aux quelques fonctions auxiliaires du module `Syntax.Light` aidant la saisie des différents opérateurs.

Une fonction `Syntax.to_string` propose également un premier afficheur simple d'expressions, sans chercher à minimiser le nombre de parenthèses grâce aux priorités des opérations. Sur l'expression précédente, `Syntax.to_string` donne ainsi la chaîne `"(x+(pi*sqrt(3)))"` .

Nous vous fournissons également des fichiers `lexer.mll` et `parser.mly` permettant le "parsing" d'une expression algébrique depuis une chaîne de caractères ou depuis un fichier. Ces deux fichiers `lexer.mll` et `parser.mly` ne sont pas en syntaxe OCaml, ils doivent être "pré-processés" respectivement par les outils `ocamllex` et `menhir`. Nous vous recommandons d'utiliser l'outil `dune` pour compiler votre projet, il pourra gérer automatiquement les appels à `ocamllex` et `menhir`. Ensuite, dans votre projet `(Parser.expr Lexer.token (Lexing.from_string s))` permettra de transformer une chaîne `s` telle que `"x+pi*sqrt(3)"` en l'expression algébrique correspondante de type `expr`. Et utiliser `Lexing.from_channel` au lieu de `Lexing.from_string` ci-dessous permet de lire un fichier ou l'entrée standard, voir par exemple le fichier `calc.ml` fourni.

La syntaxe acceptée lors de ce "parsing" fourni est simple:

- Entiers naturels
- Identifiants seuls donnant des variables (à part les cas particuliers des deux constantes `pi` et `e` donnant des `Op0`)
- Opérateurs unaires préfixes écrits sous la forme `op(...)`, `lorsque op est parmi sqrt exp log sin cos tan asin acos atan.`
- `Opérateurs infixes + * - / ^.`

Ce type `expr` peut éventuellement être remanié selon vos besoins, à condition d'adapter le "parsing" en conséquence. En particulier, le type `expr` utilise pour l'instant le type `int` pour représenter les constantes. Ceci n'est pas forcément idéal pour mener des calculs formels exacts, les opérations usuelles sur `int` peuvent conduire silencieusement à des débordements, et mener à des égalités fausses en mathématiques. Par exemple en OCaml l'entier `2*max_int+2` est égal à 0, alors que `max_int` est strictement positif ! Vous pouvez chercher à réaliser des opérations "sures" sur `int`, qui échouent en cas de débordement, ou mieux encore passer à une autre présentation des nombres (en taille arbitraire). Pour cela, vous pouvez par exemple utiliser la bibliothèque externe `zarith`.

Par ailleurs le type actuel permet d'exprimer des fractions ou des nombres à virgule, mais de manière indirecte, en utilisant la division comme pour `123/456` ou la puissance `10^(-3)` au lieu de 0.001. Mais vous pouvez remanier le type des constantes pour mieux gérer ces cas.

3 Commandes à réaliser

Le type `cmd` donne une représentation des commandes que l'on souhaite pouvoir réaliser sur nos expressions algébriques:

```
type cmd =
  | Eval of expr
  | Subst of expr*string*expr
  | Simpl of expr
  | Derive of expr*string
  | Integ of expr*string*expr*expr  (* Facultatif *)
  | Plot of expr*string             (* Facultatif *)
```

Le code `(Parser.command Lexer.token (Lexing.from_string s))` permet de lire une commande depuis une chaîne de caractère `s`.

Voici le descriptif des commandes à réaliser, les deux dernières étant facultatives (`Integ` et `Plot`):

- `eval(e)` : calcule une valeur approchée de l'expression `e` en utilisant des calculs sur les nombres flottants. L'expression `e` ne doit pas contenir de variables, sinon l'évaluation échoue (cf. la commande `subst` ci dessous).
- `subst(e,x,e')` : ici `x` doit être une variable, et alors dans l'expression `e`, chaque occurrence de `x` est remplacée par l'expression `e'`.
- `simpl(e)` : produit une version simplifiée (mais toujours exacte) de l'expression `e`. Il n'y a pas de définition précise ou unique de ce qu'on entend ici par forme simplifiée, à vous d'appliquer au mieux les règles de simplifications usuelles, par exemple `x-x=0`, `log(exp(x))=x`, `sqrt(8)=2*sqrt(2)`, `sin(pi/3)=sqrt(3)/2`, etc. Plus globalement on pourra s'efforcer de développer les expressions pour faire apparaître des termes qui s'annulent ou au contraire se regroupent.
- `derive(e,x)` : calcule la dérivée de `e` vis-à-vis de la variable `x`. On parle ici de dérivée formelle (ou exacte). Cette commande doit pouvoir fonctionner sur toutes les expressions constituées à partir des opérateurs listés précédemment.
- (Facultatif) `integ(e,x,a,b)` : cacule l'intégrale de l'expression `e` lorsque la variable `x` va des expressions `a` à `b`. On cherche de nouveau une réponse exacte. Cette fois-ci il n'y a pas de méthode générale comme pour la dérivée, on vous demande de savoir reconnaître et traiter un certain nombre de cas classiques, tels que ceux qu'on retrouve dans les formulaires de type Bac.
- (Facultatif) `plot(e,x)` : dessine la courbe correspondant à l'expression `e` vue comme fonction à une variable `x`. Cette commande échoue si `e` contient d'autres variables que `x`. Par défaut, cet affichage se fait dans la zone $[-5..5;-5..5]$, mais on peut proposer des variantes comme `plot(e,x,a,b,c,d)` pour laisser à l'utilisateur le choix des bornes $[a..b;c..d]$.

La description de ces commandes (en particulier `simpl`) est volontairement succinte, et de plus certaines commandes comme `integ` n'ont pas forcément de solution générale. A vous de traiter le plus de situations possibles, en utilisant des techniques de niveau Bac à Bac+2.

Autres idées possibles de travail réalisable :

- Utiliser des techniques de "Hash-consing" pour éviter de simplifier de multiples fois les mêmes sous-expressions.
- Signaler les contraintes rencontrées lors des manipulations, par exemple dire que l'on a supposé $x \neq 0$ avant de simplifier x/x en 1.
- Un mode verbeux qui explique à l'utilisateur quelles règles de calculs ont été utilisées (comme sur une vraie copie de bac où on doit justifier le cheminement au lieu de juste donner le résultat final).
- Réalisation d'un tableau de variation et/ou de signe pour une fonction.
- Calculs de solutions d'équations, soit de manière exacte quand cela est possible, soit de manière approchée sinon (méthode de Newton par exemple).
- Calculs de limites.

4 Interfaces

Plusieurs choix sont possibes pour l'interface utilisateur de votre programme. Il n'est pas indispensable de proposer une interface graphique, mais cela reste possible si vous le souhaitez.

- Par exemple, vous pouvez simplement réaliser un programme en ligne de commande, proposant une boucle interactive: l'utilisateur tape une ligne d'expression algébrique commençant par une commande, et votre programme affiche une version simplifiée de cette expression, dans la syntaxe d'origine. Attention à ce que cet affichage soit correct tout en étant aussi léger que possible (ni trop de parenthèses, ni trop peu).
- Lorsque qu'un tracé de fonction est demandé (commande `plot`), vous pouvez par exemple utiliser le module `Graphics` fourni avec OCaml, ou bien chercher à fabriquer une image (via la bibliothèque `camlimage`).
- Une autre idée d'interface est de proposer un joli rendu des formules. Vous pouvez par exemple chercher à produire un fichier de sortie LaTeX, ou bien du MathML.
- Si vous souhaitez réaliser une interface graphique, vous pouvez combiner l'idée d'une boucle interactive et celle de MathML afin de se rapprocher de programmes réalistes comme Maple, Mathematica ou WxMaxima. Pour cela, vous pouvez par exemple utiliser les bibliothèques `lablgtk2` et `gtkmathview`. Voir le paquet `liblablgtkmathview-ocaml-dev` sur Debian. Attention une installation manuelle sera sans doute nécessaire sur d'autres systèmes (pas de paquet `opam` en particulier, voir directement les sources sur https://github.com/AbiWord/gtkmathview).
- On peut enfin chercher à réaliser une application embarquée dans un navigateur web, grâce à la bibliothèque `js_of_ocaml`. Attention par contre, dans ce cas certaines bibliothèques externes mentionnées précédemment ne seront probablement pas disponibles (parties internes en C).

5 Rendu de projet

Le code fourni est disponible sur le gitlab de l'UFR d'informatique:
http://gaufre.informatique.univ-paris-diderot.fr/letouzey/pfa-2023

Vous devez effectuer un "fork" de cet embryon de projet, lui donner une visibilité **privée**, et nous donner accès à votre projet (utilisateurs `@letouzey` et `@geofroy`).

La date limite de rendu du projet est le 17 mai, sera suivi d'une soutenance de projet.

Vous devrez compléter le fichier `projet/README.md` en y indiquant ce que vous avez réalisé durant votre projet, et comment s'en servir. Nous vous recommandons de conserver l'infrastructure fournie pour ce projet (`Makefile` lançant `dune`), mais en cas de changement vous devrez l'indiquer dans `projet/README.md`.

6 Autres informations

Binômes. Ce projet est à réaliser par groupe de deux maximum. Nous contacter si vous ne trouvez pas de binômes. Attention, même s'il s'agit d'un travail de groupe, chacun devra parfaitement connaître l'ensemble du code réalisé. Lors des soutenances les questions et les notations pourront être individualisées.

Conseils méthodologiques. L'accent devra être mis sur la *lisibilité* et la *clarté* du code produit. En particulier:

- Indentez systématiquement et évitez les lignes de plus de 80 colonnes.
- Choisissez des noms de fonctions et de variables parlants.
- Utilisez les commentaires à bon escient : ni trop, ni trop peu. Un commentaire doit apporter quelque chose: organisation, point délicat, justification ...
- Si une fonction dépasse un écran de long, il est temps de songer à subdiviser en sous-fonctions, chacunes s'occupant d'une chose à la fois.
- OCaml permet très facilement la création de types de données personnalisés, ne vous en privez pas...
- Le copier-coller de code est à proscrire. A la place, mieux vaut prendre le temps de regrouper les choses similaires via des fonctions génériques.

Bibliothèques externes OCaml permet facilement l'utilisation de bibliothèques supplémentaires, par exemple pour le graphisme, et ce sujet en suggère un certain nombre. Si vous désirez utiliser d'autres bibliothèques non mentionnées ici, vous devez nous contacter au préalable et nous demander l'autorisation.

7 Historique

Version 1: version initiale