TP3

November 22, 2021

1 Programmation fonctionnelle

1.1 TP 3 : Programmer avec des arbres

1.1.1 Etienne Lozes - 2019

1.2 Exercice 1 : Fractions

1. Définissez un type `fraction` ayant deux champs `num` et `den` entiers.
2. Définissez une fonction `fraction:int->int->fraction` qui construit une fraction à partir d'un numérateur et un dénominateur. La fonction commence par simplifier par le pgcd et se ramène à un dénominateur ≥ 0. Pour calculer le pgcd, vous devrez reprogrammer la fonction en utilisant l'algorithme d'Euclide
3. Définissez une fonction `fraction_to_string` qui renvoie une représentation de la fraction, en affichant un entier lorsque c'est possible. Par exemple, `fraction_to_string (fraction 3 2)` affichera `3//2`, mais `fraction_to_string (fraction 3 1)` affichera `3`.
4. Associez la fonction `fraction` au symbole infixe `//`, et installez un pretty-printer. Quand c'est fini, tapez `6//(-4)` au toplevel et vérifiez qu'il s'affiche bien -3//2.

1.3 Exercice 2 : Arbres binaires d'expression

On considère une version légèrement simplifiée du type des arbres binaires d'expression défini en cours (on a seulement enlevé la soustraction et la division).

```
type arbre =
  | Feuille of feuille
  | Noeud of operateur * arbre * arbre

and feuille =
  | Const of int
  | Var of string

and operateur = PLUS | MULT
```

1. Définissez des opérateurs unaires préfixe `(?!) : string->arbre` et `(??) : int->arbre` et des opérateurs binaires infixes `++` et `**` de type `arbre->arbre->arbre` pour pouvoir définir un arbre sans écrire un seul constructeur: par exemple, `?! "x" ++ ?? 2 ** ?! "y"` sera l'arbre de l'expression $x + 2 \times y$, autrement dit `Noeud(PLUS, Feuille(Var "x"), Noeud(MULT, Feuille(Const 2), Feuille(Var "y")))`.

2. Définissez une fonction `string_of_arbre` différente de celle vue en cours, qui ne met des parenthèses que quand elles sont nécessaires.

Indication: Vous pourrez, par exemple, utiliser un fonction auxiliaire `str_of arbre est_facteur` dont le paramètre booléen `dans_prod` est vrai si le sous-arbre considéré est le facteur d'un produit.

3. Définissez une fonction `deriv : string -> arbre -> arbre` telle que `deriv x e` renvoie la dérivée formelle de l'expression `e` par rapport à `x`

1.4 Exercice 3 : Concaténation de listes en temps constant

Nous avons vu en cours que la concaténation de deux listes OCaml (chaînées, et non mutables) n'était pas une opération en temps constant, mais en temps proportionnel en la longueur de la première des deux listes à concaténer. Nous allons maintenant étudier une idée à première vue prometteuse: une structure de données permettant de concaténer deux listes en temps constant.

```
type 'a clist = Empty | Single of 'a | Cat of 'a clist * 'a clist
```

1. Écrivez une fontion `list_of_clist :'a clist -> 'a list` qui convertit une clist en une liste OCaml standard.
2. Écrivez une fonction `hd:'a clist -> 'a option` qui renvoie la première valeur de la liste lorsqu'elle existe., `None` sinon. Vous chercherez une solution de complexité en temps linéaire.
3. Écrivez une fonction `tl:'a clist -> 'a clist option` qui renvoie la queue de la liste. Vous chercherez une solution de complexité en temps linéaire.

1.5 Exercice 4: Pas de temps à perdre avec Swann

La table des occurrences d'un texte T est la fonction qui à un mot m de T associe la liste des positions où m apparait dans T. Par exemple, la table des occurrences du texte "le chien course le chat", présentée sous forme de liste associative, est

```
[("le",[0;3]); ("chien",[1]); ("course",[2]); ("chat",[4])]
```

On peut imaginer d'exploiter cette table pour faire une analyse littéraire d'un texte. Votre mission pour cet exercice sera donc de calculer de manière efficace cette table pour le roman "Du côté de chez Swann" de Marcel Proust.

1.5.1 Étape 1 : le type `table_occur`

On rappelle les types polymorphes `arbre` et `trie` vus en cours:

```
type ('n,'a) arbre = Noeud of 'n * (('a * ('n, 'a) arbre) list)
type 'a trie = ('a option, char) arbre
```

Recopiez la définition du type `arbre` et définissez le type `table_occur` comme suit

```
type table_occur = (int list, char) arbre
```

Nous allons donc représenter une table des occurrences par une trie très similaire à celle du cours, à ceci près que désormais toute clé a une valeur, la valeur par défaut étant la liste vide (au lieu de `None` pour les tries vues en cours).

Définissez 1. une variable `table_vide` qui contient une table des occurrences vide, 2. une fonction `fils : char -> table_occur -> table_occur` telle que `fils x tbl` renvoie le premier fils éti-

queté par x de l'arbre tbl s'il existe, ou sinon la table vide. Cette fonction est très proche de celle du cours, il vous faut seulement tester en plus si le fils `x` existe (regardez la fonction `List.mem_assoc` vue en cours). 3. la fonction `assoc : string -> table_occur -> int list` telle que `assoc m tbl` renvoie la liste des occurrences de m. C'est la même fonction que celle vue en cours à ceci près qu'elle ne renvoie pas une `'a option` mais une `int list`.

Vous pouvez tester votre code avec

```
let () = assert(assoc "a" table_vide  = [])
let tbl1 = Noeud([], ['a', Noeud([1],[])])
let () = assert(fils 'a' tbl1  = Noeud([1],[]))
let () = assert(assoc "a" tbl1  = [1])
let () = assert(assoc "ab" tbl1 = [])
```

4. Définissez une fonction `assoc_list_of_table_occur : table_occur -> (string * int list) list` qui convertit une table des occurrences en une liste associative `(string * int list) list`, comme illustré au début de l'exercice.

```
let () = assert(assoc_list_of_table_occur tbl1 = [("a",[1])]))
```

1.5.2 Étape 2 : construction de la table

1. Définissez une fonction `ajoute_occurrence : table_occur -> string -> int -> table_occur` telle que `ajoute_occurrence tbl m i` renvoie une nouvelle table dans laquelle a été ajoutée à la table `tbl` l'occurrence `i` du mot `m`. Vérifiez que `_assoc_list_of_table_occur (ajoute_occurrence table_vide "a" 1)` renvoie bien `("a",[1])`.
2. Définissez une fonction `build_table : string list -> table_occur` qui prend une suite de mots et qui renvoie la table des occurrences associée. Par exemple `build_table ["le"; "chien"; "course"; "le"; "chat"]` renvoie une table qui, lorsqu'on la transforme en liste associative, correspond à celle donnée en exemple au début de l'exercice.

1.5.3 Étape 3 : l'analyse littéraire

Cette dernière étape permet de vérifier que votre code est efficace et éventuellement de partir à la recherche du temps perdu en calculs inutiles.

Télécharger la liste des mots de "Du côté de chez Swan". Sauvez le fichier téléchargé dans votre répertoire de travail pour ce TP, et compilez-le. Enfin, lancez emacs dans le même répertoire.

```
cd mon/repertoire/de/travail/ocaml/tp3/
wget http://i3s.unice.fr/~elozes/enseignementPF/swann.ml
ocamlc -c swann.ml
emacs occur.ml &
```

Depuis Emacs, créez un fichier `occur.ml` dans lequel vous taperez

```
#load "swann.cmo"
open Swann
let _ = swann
```

Evaluez votre code dans le toplevel (C-x C-e, éventuellement plusieurs fois). Vous devriez voir s'afficher

```
- : string list =
["Longtemps"; "je"; "me"; "suis"; "couché"; "de"; "bonne"; "heure"; ...]
```

Rajoutez dans la suite du fichier `occur.ml` vos fonctions précédentes. Utilisez ces fonctions pour produire une liste associative qui représente la table des occurrences de “Du côté de chez Swann”. Est-ce bien plus rapide que si vous aviez utilisé la fonction `build_table_naif` ci-dessous?

```
let build_table_naif l =
  let rec add_index s i = function
    | [] -> [(s,[i])]
    | (s2,li)::l when s=s2 -> (s2,i::li)::l
    | p::l -> p::add_index s i l
  in
  let rec f i res = function
    | [] -> res
    | s::l -> f (i+1) (add_index s i res) l
  in f 0 [] l
```