Une majorité des exercices de cette feuille ont été écrit par Simon Halfon pour l'année scolaire 2015/2016

1 Rappels du précédent TP

Dans le précédent TP, on a vu comment se servir de la ligne de comande avec BASH. Cela vous permet d'intéragir directement avec le système d'exploitation en lui donnant des commandes. C'est par ce biais qu'il est possible notamment de modifier l'*arborescence de fichiers* (créer des dossiers/fichiers par exemple). Vous devriez normalement être un minimum à l'aise avec les commandes suivantes : *ls, cd, man, mkdir, ocamlc*. Cependant, la ligne de commande n'est pas très pratique pour écrire des programme, et notamment du code OCAML. Pour cela, en adoptant un des principe Unix[1]

> Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".

on va préférer utiliser un éditeur de texte. Dans le TP précédent, je vous ai fait découvrir deux éditeurs de texte à savoir EMACS et VIM. Peu importe l'éditeur que vous choisissez, l'objectif c'est que vous soyez rapide et efficace avec cet outil. De plus, vous pourrez le réutilisez pour éditer des fichiers dans différent langages comment python, C, x86, LaTeX pour en mentionner quelqu'uns.

Enfin, on a vu comment compiler un programme *hello world* en OCAML puis l'éxécuter. J'ai oublié cependant de mentionner dans le TP l'option `o` qui permet de renommer l'éxécutable produit. Ainsi avec la commande `$ ocamlc helloworld.ml -o helloworld`

OCAMLC va produire un fichier éxécutable `helloworld` qui pourra être éxécuté par

`$ ./helloworld`

2 Programmer en Ocaml

Maintenant que vous avez préparé votre environnement de travail, il va être temps de coder en Ocaml! Cette feuille de TP regroupe un certains nombre d'exercices qui vont globalement dans l'ordre croissant de difficulté. L'objectif de ce TP va être de vous préparer à faire le mini-projet qui seras mis en ligne **le 23 septembre**. Le premier exercice vise principalement les gens qui ont déjà fait du Caml-Light en vous expliquant les principales différences entre Caml-Light et Ocaml. Pour ceux qui n'ont aucune expérience d'Ocaml, je vous invite dans un premier temps à aller regarder les ressources que j'ai mentionné dans le précédent TP et que vous pouvez retrouver sur ma page web www.lsv.ens-cachan.fr/~fthire.

Bonnes pratiques

Il est *primordial* en programmation d'avoir recours à des bonnes pratiques. Dites-vous qu'en général un programme est beaucoup plus souvent lu qu'il n'est modifié. Ainsi, il faut qu'en le lisant, le code soit clair et compréhensible. Ce critère sera pris en compte dans l'évaluation de vos projet lors de ce cours. Voici quelques critères à prendre en compte

- Documenter[2] et commenter votre code ! (commenter $\neq$ documenter)
- Les commentaires, les noms de fonction ainsi que ceux des variables doivent être en anglais!
- Donner des noms explicites (et en général plutôt court) à vos variables. Les noms de variables et a forciori de fonctions doivent être en miniscule, les mots séparés par des *underscore* (spécifique à Ocaml)
- N'oubliez pas d'indenter votre code. L'éditeur que vous avez choisi devrait pouvoir le faire automatiquement pour vous.

[1] https://en.wikipedia.org/wiki/Unix_philosophy
[2] http://caml.inria.fr/pub/docs/manual-ocaml/ocamldoc.html

- La compilation ne doit pas indiquer de *warning*.
- Utilisez un maximum les idiomes du langage (En Ocaml, éviter les `while` par exemple)
- Découper votre code en module

Exercice - 1

Librairies Ocaml

listes, tableaux, programmation récursive, while, for

Certains d'entre vous ont peut être appris en CPGE le langage "Caml-Light". Comme son nom l'indique, il s'agit d'une version allégée de Ocaml. Malheureusement, ce langage vous donne quelques mauvaises habitudes:

- Les librairies `List` et `Array` y sont "ouvertes" par défaut
- Les fonctions de la librairie `Array` sont renommées avec le préfixe "vect" et en utilisant une syntaxe qui n'existe pas en Ocaml: par exemple `make_vect`

Dans Ocaml, il n'y a pas de fonction `hd`. Par contre, il y a une librairie `List` qui contient une fonction `hd`. Pour l'utiliser dans votre code, vous pouvez:

- Soit utiliser `List.hd`
- Soit ajouter la ligne `open List` au début de votre fichier, puis utiliser `hd`.

Une documentation des librairies Ocaml (fonctions de la librairie et leur description) est disponible sur le site de l'INRIA. En pratique, tapez "Ocaml Lib" dans un moteur de recherche (Lib = nom de la librairie). Vous serez ammenés à utiliser les librairies suivantes: `List`, `Array`, `Printf`, `Scanf`, `String`, ... Vous pouvez aussi consulter la librairie `Pervasives` qui contient les fonctions de bases d'Ocaml (elle n'a pas besoin d'être ouverte !).
Question 1 : Ecrivez une fonction qui transforme une liste en un tableau.[3]
Question 2 : Si ce n'est pas déjà le cas, réécrivez la sans boucle `for` ou `while`.
Question 3 : Ecrivez une fonction qui transforme un tableau en liste.
Question 4 : Si ce n'est pas déjà le cas, réécrivez la sans boucle `for` ou `while`.
Question 5 : Ecrivez une fonction

```
make_matrix: int -> int -> int array array
```

qui crée une matrice de tailles données en entrée remplie de 0 (interdiction d'utiliser `Array.make_matrix`).
Question 6 : Créez une matrice de taille 4×4 et mettez à jour la case $(2, 1)$. Afficher la matrice.

Exercice - 2 *Soyez dynamique !*

---

programmation récursive, while, fonctions récursives terminales

Question 1 : Programmez une fonction récursive **fibo** de type `int -> int` qui calcule le $n^{\text{ième}}$ terme de la suite de Fibonacci. Avec la commande `time`, observer le temps qu'il faut pour calculer le $43^{\text{ième}}$ terme.
Question 2 : Programmez une nouvelle fonction **fibo** qui cette fois utilise une boucle **while**. Observer la différence de temps de calcul avec la commande `time`.
Question 3 : En utilisant une fonction récursive auxiliaire de type `int -> int -> int -> int` , implémentez **fibo** qui soit aussi rapide que celle programmée précédemment. Votre fonction récursive auxiliaire devrait être normalement une fonction *récursive terminale*. C'est à dire que la dernière instruction éxécutée par la machine est l'appel récursif (https://en.wikipedia.org/wiki/Tail_call).

[3] `[||]` construit un tableau vide de type α **array**

Exercice - 3 *Folding@home*

list, fold

Réimplémenter quelques fonctions de la librairie standard du module **List** en utilisant seulement la fonction `fold_left`. Vous n'avez pas le droit d'utiliser le `match` ni le `let rec` pour cet exercice.

Exercice - 4 *Exceptions*

Question 1 : Sans utiliser la librairie Array, écrivez une fonction qui cherche si un élément est dans un tableau.
Question 2 : Réécrivez votre fonction sans utiliser de récursion, ni de boucle `while`. La fonction ne doit pas pour autant parcourir inutilement le tableau (si vous ne voyez pas comment faire, regardez le titre de l'exercice).
Question 3 : Modifiez votre fonction pour qu'elle renvoie l'indice de première occurence de l'élément dans le tableau.
Question 4 : Implémentez une fonction de recherche dans un arbre dont les noeuds ne sont pas étiquetés et les feuilles sont étiquetées par des entiers.
Question 5 : Réécrire cette fonction en utilisant une exception.
Question 6 : En utilisant la fonction `time` suivante, comparez les temps d'exécution des deux fonctions sur arbres binaires complets de grande profondeur dont toutes les feuilles sont étiquetées par 0. Qu'observez vous ? Comment l'expliquez vous ?

```
let time f x =
  let t = Unix.gettimeofday () in
  let res = f x in
  Printf.printf "Execution time: %fs\n" (Unix.gettimeofday () -. t ;
  res ;;
```

Exercice - 5 *Polymorphisme*

arbres, types algébriques, curryfication

Question 1 : Ecrivez le type d'un arbre d'arité non fixée dont les noeuds sont étiquetés par une valeur de type polymorphe.
Question 2 : Ecrivez une fonction qui réalise le parcours en profondeur d'un tel arbre en appliquant une fonction abstraite de type `'a -> unit` où `'a` est le type des valeurs aux noeuds.
Question 3 : Même chose avec un parcours en largeur (vous pouvez utiliser la librairie `Queue`).
Question 4 : Réécrivez la fonction précédente sans utilisez la librairie `Queue`.
Question 5 : Ecrire deux fonctions:

```
f: 'a * 'b -> 'b * 'a
g: 'b * 'a -> 'a * 'b
```

telles que $f \circ g$ et $g \circ f$ soient l'identité. On dit que les types `'a * 'b` et `'b * 'a` sont isomorphes.
Question 6 :
Question 7 : Montrer que les types

```
('a * 'b) -> 'c
'a -> ('b -> 'c)
```

sont isomorphes, en écrivant les fonctions associées. C'est ce que l'on appelle la *curryfication*.
Question 8 : Les types `'a` et `'a * 'a` sont-ils isomorphes ?
Question 9 : Et les types `int` et `int * int` ? (vous sentez un piège là, non ?)

Exercice - 6 *Réalisation d'une calculatrice à l'ancienne*

---

modularité, parsing

Dans cet exercice, nous abordons le problème du *parsing*: un compilateur reçoit en argument un fichier, qui n'est autre qu'une grosse chaine de caractère. La première étape est donc de retrouver la structure *arborescente* du programme à l'origine. Vous ne connaissez pas encore les outils théoriques utilisés pour faire du *parsing*, vous verrez cela au cours de Langages Formels au second semestre (et oui, les automates que vous avez vu en prépa, en fait, ça sert à quelque chose !). On va donc travailler avec des expressions très simples, et faire du parsing "à la main". Vous allez réaliser un parseur d'expressions mathématiques écrites en *notation polonaise inverse* (https://fr.wikipedia.org/wiki/Notation_polonaise_inverse). Il faut prendre en compte les opérations entières `+`, `-`, `*`, `/`.

Cet exercice est aussi l'occasion de parler de la compilation séparée. Pour un projet de taille importante, il est fondamental d'organiser son code. Il convient notamment de séparer le code en plusieurs fichier `.ml`. Cela présente plusieurs avantages:

- On peut ne recompiler qu'une partie du projet
- Un module peut être réutilisé
- Clarté du code

Il faut alors "lier" les fichiers. C'est la même chose que pour une librairie (type `List`): on ajoute la ligne `open Module` en début de fichier (ou on écrit `Module.f` à chaque utilisation d'une fonction `f` du module). Pour compiler le tout, il suffit de mettre les fichiers `.ml` dans le bon ordre.
Question 1 : Récupérer les sources de la calculatrice (demandez moi la clé USB) et utilisez la commande `$ ocamlc -o Calc graphics.cma parser.ml calculator.ml interface.ml` pour compiler le tout.

Imaginons que le fichier `parser.ml` ne soit pas de vous, et que vous soyez sûr de ne pas le modifier. Il est alors inutile de le recompiler à chaque fois. Vous pouvez le compiler une seule fois indépendément avec `$ ocamlc -o parser.ml` puis utiliser le *fichier objet* (.cmo) généré avec `$ ocamlc -o Calc graphics.cma parser.cmo calculator.ml interface.ml` . C'est la même chose pour les librairies Ocaml: dans ce cas vous ne disposez même pas du fichier `.ml` à recompiler, vous ne pouvez que lui fournir le fichier objet. Cette fois ci cependant, l'extension est `.cma` pour indiquer au compilateur d'aller chercher le fichier dans les librairies standards et pas dans le dossier courant. Tous ces aspects module sont l'une des force d'Ocaml (*e.g.* les signatures, les foncteurs, etc). Vous les verrez en détail dans le cours de programmation 2.

Vous vous dites sans doute que ça commence à être pénible de taper des lignes aussi longue pour compiler, et notre petit projet ne contient que trois fichiers ! Pour cette raison il existe un outil particulièrement pratique: le `Makefile`. Ce n'est pas l'objet de ce TP de comprendre en détail comment cela fonctionne, pour l'instant, retenez simplement que pour compiler il vous suffit de taper `$ make` , et cela crée (ou met à jour) l'exécutable du projet, que j'ai nommé `Calc`. Pour lancer la calculette, tapez donc `$ ./Calc` .

Question 2 : Faites le, et jouez un peu avec la calculette. Vous allez vite vous rendre compte de ce qui n'est pas codé.

Les fichiers à compléter sont `parser.ml` et `calculator.ml`, mais il se pourrait que vous ayez besoin d'aller consulter `interface.ml` pour savoir comment doivent se comporter les fonctions `parse` et `compute` (notamment vis à vis des exceptions).
Question 3 : écrire le type `arith` qui vous permettra de représenter les expressions arithmétiques décrites ci dessus

Question 4 : coder une fonction `split : string -> char -> string list` telle que `split s c` retourne une liste de string $s_1, \ldots, s_n$ et tel que

$$s_1 c s_2 c \ldots c s_n = s$$

Question 5 : écrire la fonction `parse`, qui produit la représentation de type `arith` d'une expression arithmétique sous forme de chaine de caractère, ou produit l'exception `Parsing_error` si la chaine reçue n'est pas correcte. Il suffit pour cela de créer une pile puis de parcourir la chaine de gauche à droite, si l'on voit un nombre on l'empile, si l'on voit un opérateur, on dépile deux expressions de la pile et on empile l'expression obtenue en appliquant l'opérateur aux deux expressions dépilées.
Question 6 : écrire la fonction (triviale) qui prend une expression arithmétique et calcule sa valeur entière. Notez que l'on aurait pu court-circuiter le type `arith`, il est très simple de modifier la fonction `parse` pour qu'elle effectue ce calcul directement.

Exercice - 7 *Mini Librairie Listes circulaires doublement chainées*

---

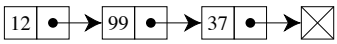
références, types mutables, listes circulaires

Plutôt que d'inclure quelques rappels très incomplets sur les records, je vous invite à consulter l'un des très bon tutoriels que l'on peut trouver sur Internet (par exemple sur https://realworldocaml.org). En particulier, renseignez vous sur les champs mutables. L'exemple le plus simple de type produit, auquel vous êtes habitué, est le type référence. Essayez cette ligne en Ocaml:

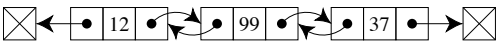
```
let x = ref 0 in x ;;
```

Le point d'exclamation "!x" qui permet d'accéder à la valeur contenu dans "x" est en fait un raccourci syntaxique pour "x.content".

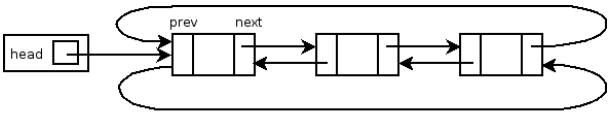
En mémoire, une liste est stockée comme plusieurs *cellules* "éparpillée" dans la mémoire. Chaque cellule contient une valeur d'élément, et l'adresse mémoire de la cellule suivante.



Lorsque Ocaml manipule une liste, il ne connait en réalité que l'adresse de la première cellule. C'est pour cela qu'il est coûteux d'accéder au dernier élément: il faut "sauter" de cellule en cellule, chaque cellule dévoilant l'adresse de la suivante. De plus, connaissant l'adresse mémoire d'une cellule, il est impossible de retrouver l'adresse de la cellule précédente. Il est donc parfois d'utiliser une structure de donnée plus complexe: les listes doublement chainées.



On va en fait utiliser une troisième structure: les listes doublement chainées circulaires.



Ecriture de la librairie:
Question 1 : On commence par définir les types polymorphes `'a cell` pour les cellules et `'a circ_list` pour les listes. Pour pouvoir traiter le cas de la liste vide, il va falloir utiliser le type `option`.
Question 2 : Définir `nil` la liste vide, puis la liste singleton contenant l'entier 1.
Question 3 : Définir les fonctions:

- `hd: 'a circ_list -> 'a`

  qui retourne le premier élément d'une liste.

- `last: 'a circ_list -> 'a`

  qui retourne le dernier élément d'une liste.

- `nth: int -> 'a circ_list -> 'a`

  qui retourne le nieme élément d'une liste.

- `cell: 'a circ_list -> 'a cell`

  qui retourne la première cellule d'une liste.

- `length: 'a circ_list -> int`

  Vous aurez sans doute besoin de la primitive == de Ocaml.

- `list_of_circ_list: 'a circ_list -> 'a list`

  qui retourne la liste (du type list de Ocaml) correspondant à une liste circulaire. Cela vous permettra d'afficher vos listes de manière lisible, mais il est interdit de l'utiliser pour les fonctions suivantes (seulement pour afficher vos exemples).

- `egal: 'a circ_list -> 'a circ_list -> bool`

  qui teste si deux listes circulaires représentent la même liste.

- `circ_perm: 'a circ_list -> 'a circ_list -> bool`

  qui teste si une liste circulaire est une permutation circulaire de l'autre.

- `circ_perm__memory: 'a circ_list -> 'a circ_list -> bool`

  qui teste si une liste est physiquement une permutation circulaire de l'autre. C'est par exemple le cas pour les listes ℓ_1 et ℓ_2 si ℓ_1 pointe vers une cellule `cell` et ℓ_2 vers `cell.next`.

- `copy: 'a circ_list -> 'a circ_list`

  qui copie une liste circulaire. Attention, la liste retournée doit être indépendante de la liste donnée en argument: modifier la première liste ne doit pas changer la seconde ! (penseze aux vecteurs).

- `insert: 'a -> 'a circ_list -> 'a circ_list`

  qui insère un élément en tête de liste. Cette fois, pas de copie ! La liste retournée doit être la même. Par exemple, votre fonction doit vérifier ceci (si old_list n'est pas vide):

  ```
  let new_list = insert 3 old_list in
  let c = cell new_list in
  c.next == cell old_list
  ```

  doit renvoyer "true". Est-ce aussi le cas avec:

  ```
  let new_list = insert 3 old_list in
  let c = cell old_list in
  c.prev == cell new_list
  ```

  (ici "next" et "prev" sont deux champs du type "'a cell").

- La réciproque de la fonction `list_of_circ_list`

  `circ_list_of_list: 'a list -> 'a circ_list`

  qui pourrait s'appeller `create`. Elle vous permettra de créer vos exemples sans avoir à en écrire des tartines à la main.

- `append: 'a circ_list -> 'a circ_list -> 'a circ_list`

  qui effectue la concaténation de deux listes. Ici aussi, la fonction doit être destructive, c'est à dire qu'elle altère ses arguments.

- `filter: ('a -> bool) -> 'a circ_list -> 'a circ_list`

  qui sélectionne les élément d'une liste qui vérifie le prédicat donné en argument. La liste en argument NE doit PAS être modifiée.

- En déduire la fonction

  `suppr_elt: 'a -> 'a circ_list -> 'a circ_list`

  qui supprime les occurences d'un élément.

- `map: ('a -> 'b) -> 'a circ_list -> 'b circ_list`

Exercice - 8 *Streaming*

---

listes infinis, call-by-value

Á votre avis, quelle est la différence importante entre ces deux programmes ? Essayez de les éxécuter chacun séparemment pour observer la différence (dans le terminal, le raccourci (Ctrl-C) permet d'interrompre un processus).

```
let foo = while true do () done;
```

et

```
let bar () = while true do () done;
```

?

Question 1 : En utilisant cette remarque, essayez de définir un type pour des *streams* qui sont des listes infinies.

Question 2 : Implémenter quelques fonctions sur les streams:

- `val gen l : 'a list -> 'a stream`

  qui génère un stream où la liste l est répétée en boucle

- `val circ n : int -> 'a stream`

  qui génère un stream où l'entier n est répété à l'infini.

- `val sub m l : int -> 'a stream -> 'a list`

  Qui prend un stream l et qui retourne ses m premiers éléments.

Exercice - 9 *Blooming*

Dans cet exercice, on va implémenter une structure de données qui s'appelle *filtre de Bloom*[4]. Cette structure de donnée permet de tester de façon très efficace l'appartenance d'un objet dans un ensemble. Cependant, ce test est un test probabiliste qui peut donner des *faux-positifs*, c'est à dire qu'il y a une probabilité pour qu'il répondre *oui* alors que l'objet n'y est pas. Par contre, s'il répond non, c'est sûr à tout les coups que l'objet n'est pas présent.

L'algorithme gère un tableau de bits en mémoire. Au départ, le tableau de bits est initialisé à 0. Quand on ajoute un élément à notre ensemble, l'algorithme va mettre k (k sera déterminé plus tard) bits du tableau à 1.

[4] https://en.wikipedia.org/wiki/Bloom_filter

Quand on voudra tester si l'élément appartient à notre ensemble, il suffira de tester si k bits ont été mis à 1.

Pour déterminer quels bits sont mis à 1, on va utiliser une *fonction de hashage*. Une fonction de hashage prendre en entrée un objet de type α et retourne un `int`. La valeur retournée peut-être vu comme une empreinte de l'objet initial. Évidemment, si cette fonction de hashage est implémentée comme suit :

$$f(x) = 1$$

alors tous les objets sont identifiés de la même façon et la fonction de hashage est inutile. On préferera des fonctions compliquées qui garantissent deux entiers différents pour deux objets très similaire. Par exemple en utilisant la fonction `Hashtbl.hash` de la librairie standard, vous pouvez observez les entiers générés pour les chaînes "a" et "aa" (ou "a" et "b").

Question 1 : A partir du nombre de mots estimés dans l'ensemble ainsi qu'une probabilité p de faux-positif, l'article wikipédia permet de calculer m, la taille du tableau de bits ainsi que k le nombre de fonctions de hashage.

$$m = -\frac{n \times \log p}{(\log 2)^2}$$

$$k = \frac{m}{n} \times \log 2$$

Implémenter ces formules en Ocaml (n'oubliez pas que m et k seront des entiers).

Pour représenter un tableau de bit, on ne va pas utiliser le type `int array` ce qui serait terriblement peu efficace. Par contre, on peut astucieusement utiliser un `int` Ocaml pour représenter 31 bits (le type `int` d'Ocaml est stocké sur 31 bits[5]). 31 bits c'est un peu embêtant à manipuler, et donc je vous demander d'utiliser la librairie `Int32`.

Question 2 : Donner un type `bloom` pour la structure de donnée du filtre de Bloom. Ainsi que deux fonctions :

- ```
  val get : bloom -> int -> bool
  ```

  qui retourne vrai si le $n$ième bit a été mis à 1 et faux sinon.

- ```
  val set : bloom -> int -> unit
  ```

  qui met à vrai le $n^{\text{ième}}$ bit.

Question 3 : Implémenter maintenant les deux fonctions suivantes :

- ```
  val add : bloom -> 'a -> unit
  ```

  qui ajoute un élément au filtre de bloom

  ```
  val check : bloom -> 'a -> bool
  ```

  qui teste si un mot appartient au filtre de bloom.

Question 4 : Télécharger le fichier `dictionary.txt` à l'adresse suivante : http://www.lsv.ens-cachan.fr/~fthire/teaching/2016-2017/programmation-1/1/blooming/dictionary.txt. Tester le filtre de Bloom avec ce fichier. Pour connaître le nombre de mots dans le fichier, vous pouvez utiliser la commande `wc -l`.

Question 5 : (bonus) Comparer avec la commande `time` l'efficacité du filtre vis à vis d'un algorithme de recherche classique comme la recherche dichotomique, ou bien tout simplement avec la commande BASH `grep`.

[5] http://stackoverflow.com/questions/3773985/why-is-an-int-in-ocaml-only-31-bits
  ```