

TP 4

0 Environnement de travail

0.1 Session de secours

- Les sessions de secours étant recrées tous les jours, il est conseillé de travailler de la façon suivante :
- télécharger l'archive ZIP contenant les fichiers à compléter et la décompresser. Cette dernière contient un unique répertoire `pfa_l3_info_tpxx` où `xx` est le numéro du TP.
 - ouvrir un terminal et se placer dans le répertoire en question `cd chemin/vers/le/repertoire`
 - exécuter le script d'initialisation `./init_tp.sh`
 - fermer le terminal et en ouvrir un nouveau
 - travailler (voir la section 0.2)
 - sauvegarder régulièrement et au moins une fois en fin de séance les fichiers du TP (soit en ligne sur un espace personnel¹, soit en les stockant sur une clé USB)

0.2 Utilisation de Visual Studio Code

Une fois configuré correctement (par le script d'initialisation), VSCode est un éditeur confortable pour du code OCaml (moins lourd que Netbeans ou Eclipse en particulier). Voici l'ensemble minimal des commandes pour les TPs :

- Création d'un nouveau Fichier : **CTRL-N** (ou « *File → New file* ». Attention, le fichier nouvellement créé n'a pas de nom. Il convient alors de l'enregistrer (**CTRL-S**) ou *Save...*) en lui donnant un nom se terminant par `.ml`
- Évaluation d'une expression OCaml : il est possible de surligner (**SHIFT+↑ / ↓** ou en utilisant la souris) une portion de code OCaml puis de l'envoyer dans un interpréteur avec **SHIFT-ENTER**. Attention, il convient d'évaluer les définitions dans l'ordre.

De façon générale, les archives de TP contiennent déjà un ou plusieurs fichier à compléter, il est rare de devoir créer des fichiers.

Présentation

Le but du TP est de se familiariser avec la notion de module, de foncteurs et de compilation séparée. Le TP est basé sur une archive à télécharger sur la page du cours. Dans cette archive on trouve les fichiers :

tree.ml : ce dernier contient la définition du type des arbres binaires de recherche et toutes les fonctions associées. **Il faut compléter ce fichier.**

tree.mli : ce dernier contient les signatures des types de modules et du foncteur **Make**. **Il faut compléter ce fichier.**

main.ml : ce fichier contient le programme principal, c'est dans celui-ci qu'il faudra écrire vos tests. **Il faut compléter ce fichier.**

dune-project, dune, .ocamlformat : fichiers de configuration du projet. **Il ne faut pas modifier ces fichiers.**

On peut compiler le programme avec

```
$ dune build
```

comme vu en cours. Une façon agréable de travailler est la suivante :

1. a priori seul le HTTP et HTTPS sortant est autorisé à l'heure actuelle, il est donc possible que l'utilisation de `git` via SSH ne fonctionne pas

- ouvrir un terminal et se placer dans le répertoire du TP. Ce terminal servira à lancer l'exécutable :
\$./main.exe
- ouvrir le répertoire du TP avec VSCode. Par exemple, dans le terminal précédemment ouvert lancer l'éditeur avec
\$ code .
- ouvrir un terminal **dans VSCode** avec le raccourci **Ctrl-Shift-`** ou avec le menu Terminal → New Terminal.
- dans le terminal VSCode, lancer la commande **dune build -w**. Cette dernière va reconstruire le fichier **main.exe** à chaque changement dans vos sources. Si tout se passe bien, le message :

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
Success, waiting for filesystem changes...
```

s'affiche. En cas de problèmes un message d'erreur s'affiche en rouge. Il est alors possible de faire **Ctrl-Click** sur le numéro de ligne pour positionner le curseur sur l'erreur.

```
PROBLEMS  Open file in editor (ctrl + click)  SOLE  TERMINAL  PORTS
File "main.ml", line 1, characters 14-17:
1 | let f x = x + 1.0
   ^^^
Error: This expression has type float but an expression was expected of type
      int
Had errors, waiting for filesystem changes...
```

Attention, tant que la commande **dune build -w** est en cours d'exécution, tout autre invocation de **dune** (par exemple dans un autre terminal) échouera (en disant qu'il y a déjà une instance entrain de construire le projet).

Attention on suppose dans la suite que le TP est fait sous Linux, de préférence sur les machines du PUIO ou sur une machine configurée comme indiqué sur la page de cours. Si l'une de ces commande ne fonctionne pas sur une machine du PUIO (problème de version de **dune**, problème avec l'extension VSCode qui ne trouve pas des composants, ...), merci de vérifier que vous avez effectué scrupuleusement toutes les étapes de configuration du TP 1.

1 Prise en main du foncteur Make

Dans cette section on se familiarise avec le foncteur **Make** au moyen de quelques petits problèmes algorithmiques.

1. Prendre quelques minutes pour lire le fichier **tree.mli**. Noter en particulier la différence avec les signatures du cours. Le type de module pour les éléments est **CompPrint** ou l'on demande pour un type **t** à la fois une fonction de comparaison et une fonction d'affichage.
2. Dans le fichier **main.ml** créer un module **IntSet** contenant des ensembles d'entiers. Vous pouvez utiliser come argument du foncteur **Tree.Make** le module **Int** de la bibliothèque standard, qui contient les fonctions **compare** et **to_string**. Créez un ensemble de quelques entiers dans une variable globale **iset**.
3. Dans le fichier **tree.ml**, implémenter la fonction **to_string : t -> string** du foncteur **Make**. On souhaite que l'affichage se fasse comme suit :
 - si l'ensemble est vide, renvoyer "{ }"
 - si l'ensemble est un singleton $\{v_1\}$, renvoyer "{ s1 }", où **s1** est la chaîne de caractère correspondant à l'élément v_1
 - sinon pour un ensemble $\{v_1, \dots, v_n\}$ (avec $n \geq 2$), renvoyer "{ s1; s2; ...; sn }" où les **si** sont les chaînes représentant les v_i .

Indication la fonction est plus simple à écrire si l'on reconvertit d'abord l'arbre en liste au moyen de **elements**. La fonction doit être réursive terminale. Si on utilise des concaténation de chaînes (comme dans le corrigé) cette fonction est quadratique en la taille de la chaîne finale. On revisitera cela après avoir vu les effets de bords et le type **Buffer.t**.

4. Tester la fonction **IntSet.to_string** dans **main.exe**
5. Écrire (dans **main.ml**) une fonction **subset_without_duplicates : string list -> IntSet.t**. Cette dernière prend en argument la liste des lignes d'un fichier au format suivant :

```
17
1000
150
403
--
3002
2131
3002
--
17
12
--
18
2323
18
--
```

c'est à dire des paquets d'entiers séparés par des `--`. La fonction renvoie l'ensemble de tous les entiers appartenant à un paquet sans doublon. Dans l'exemple ci-dessus, la fonction renvoie l'ensemble constitué des entiers du premier et du troisième paquet (l'entier 17, bien qu'apparaissant 2 fois dans le fichier apparait dans deux paquets différents, il faut donc le garder). Tester votre fonction sur les fichiers **intset1.txt** et **intset2.txt** fournis. La fonction utilitaire **read_file** fournie au début du fichier **main.ml** peut être utilisée pour lire facilement un fichier. Pour le fichier **intset2.txt**, vous pouvez vérifier par exemple que la valeur 9801 n'apparaît pas dans votre résultat (alors qu'elle est dans un des paquets).

6. Créer un module **Chr** de signature **CompPrint**. Pour ce module, le type **t** est le type **char** d'OCaml, la fonction **compare** est **Char.compare** et la fonction **to_string** renvoie pour le caractère *c* la chaîne `"'s'"` où *s* est la version échappée de *c*, obtenue via **Char.escaped**. (Ainsi pour le caractère de code 10, on obtiendra bien la chaîne `"'n'"` qui une fois affichée produira `"n"`). Utiliser ce module pour créer un module **ChrSet** des ensembles de caractères.
7. Écrire une fonction **charset : string -> CharSet.t** qui renvoie l'ensemble des caractères d'une chaîne donnée en argument. On pourra utiliser la fonction

String.fold_left ('a -> char -> 'a) -> string -> 'a

similaire à celle des listes, mais qui parcourt les caractères d'une chaîne.

8. Écrire une fonction **only_one_row : string list -> (int * string) list**. Cette dernière prend en argument des listes de mots (contenant des majuscules, minuscules ou d'autres caractères tels que `'`). La fonction renvoie la liste des couples (i, w) où *i* vaut entre 1 et 3 et *w* est un mot de la liste initiale qui peut s'écrire uniquement en utilisant les lettres de la ligne *i* d'un clavier QWERTY (on ignore l'utilisation de **Shift**). En d'autre termes, renvoyez la liste des mots constitués uniquement des lettres `"qwertyuiop"` ou uniquement `"asdfghjkl"` ou uniquement `"zxcvbnm"`. Testez sur le fichier **english.txt**.

2 Amélioration de l'interface **tree.mli**

L'interface des ensembles n'est pas tout à fait satisfaisante. Un premier problème est qu'il manque des fonctionnalités telles que **remove_min : t -> elt * t** et **remove_max : t -> elt * t**. Un autre pro-

blème est que les itérateurs **iter** et **fold** sont utiles lorsque l'on veut parcourir la totalité de la collection mais ne permettent pas de s'arrêter prématurément.

1. rajouter la fonction **remove_max** qui renvoie le couple de l'élément le plus grand de l'arbre et l'arbre privé de ce dernier.
 - si l'arbre donné en argument est vide, votre fonction peut lever une exception (par exemple avec **failwith**)
 - la fonction interne **remove_max_elt** existe déjà dans le fichier, mais elle prend en argument l'opération **join**. Il suffit de lui passer le bon paramètre et d'exporter la fonction ainsi obtenue.Rappel : pour exporter la fonction il faut ajouter son type dans la signature **S**, à la fois dans le fichier **tree.ml** et dans le fichier **tree.mli**. Sous VSCode, on peut basculer d'un fichier **.ml** au fichier **.mli** et vice versa avec le raccourci **Alt-o**.
2. écrire sur le même modèle la fonction **remove_min** qui renvoie le plus petit élément et l'arbre privé de ce dernier
3. tester les deux fonctions précédentes dans le fichier **main.ml**

Les fonctions **fold** ou **iter** ne permettent pas de s'arrêter dès que possible. On souhaite donc que la signature **S** propose une interface *d'itérateurs* :

```
1  ...
2  type iterator
3  val mk_iterator : t -> iterator
4  val next : iterator -> (elt * iterator) option
```

Le type **iterator** est abstrait. On peut juste construire un itérateur à partir d'un arbre avec **mk_iterator**. Cet itérateur « pointe » initialement à une position fictive placée avant la plus petite valeur de l'arbre. Étant donnée un itérateur **it**, la fonction **next** renvoie soit la valeur **None** s'il n'y a plus de valeur, soit **Some (v, itt)** où **v** est la prochaine valeur dans l'arbre par ordre croissant et **itt** est l'itérateur pointant vers la valeur suivante, si elle existe.

4. ajouter et implémenter l'interface d'itérateur dans la signature de **S**.
5. vous servir de cette interface pour écrire une fonction

smallest_such_that : (int -> bool) -> IntSet.t -> int option

qui renvoie le plus petit entier d'un ensemble vérifiant le prédicat donné en argument. La fonction renvoie **None** si un tel entier n'existe pas et **Some (i)** sinon, où **i** est l'entier recherché.