université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

PFA

L3 Info, L3 Mag Info

TP 5

Présentation

Le but du TP est de continuer à se familiariser avec la notion de foncteur. On se concentre cette fois sur l'utilisateur de foncteurs déjà écrits. Le TP est composé de deux exercices chacun situé dans un sous-répertoire (`exo1` et `exo2`). Il est fortement conseillé de se placer dans le répertoire contenant ces deux sous-répertoire. Chacun des exercices à la même structure :

- un fichier `main.ml` contenant le code OCaml que vous devez compléter
- un fichier `dune` contenant les instructions de compilation

La commande `dune build` construit les deux exécutables `exo1/main.exe` et `exo2/main.exe`

1 Statistiques sur les mots

On souhaite effectuer des statistiques sur les mots d'un texte. On utilisera à titre d'exemple le fichier `jungle2.txt`. Ce fichier contient le texte du livre « Le second Livre de la Jungle » [1]. Le texte a été normalisé pour ne contenir aucune ponctuation et uniquement des mots en majuscule contenant des caractères de `A` à `Z` (pas d'accents ou autres symboles). Le texte est constitué de lignes de mots, séparés par des espaces, sans ligne vide.

Les fonctions utiles pour cet exercice sont :

- `open_in : string -> in_channel` qui permet d'ouvrir en lecture un fichier dont le nom est donné. La valeur renvoyée est un descripteur de fichier. Si l'ouverture est impossible (fichier inexistant, problème de permission, ...) la fonction lève une exception.
- `close_in : in_channel -> unit` ferme le descripteur de fichier et libère les ressources associées.
- `In_channel.input_line : in_channel -> string option` la fonction renvoie `None` si on est arrivé en fin de fichier. Sinon, elle lit la prochaine ligne du fichier et renvoie `Some s` où `s` est la chaîne de caractères correspondant à cette ligne, sans `\n` final.
- `String.split_on_char : char -> string -> string list` telle que `String.split_on_char c s` sépare la chaîne `s` selon le caractère `c` et renvoie la liste des chaînes ainsi obtenues.
- Les fonctions du module `Map`. Ce dernier permet d'obtenir des dictionnaires dont les clés sont celles d'un type implémentant l'interface `OrderedType :`

```
1  module type OrderedType = sig
2    type t
3    val compare : t -> t -> int
4  end
```

La documentation de ce module est disponible à l'adresse suivante :

`https://v2.ocaml.org/api/Map.Make.html`

On s'intéressera particulièrement aux fonctions `find`, `add`, `fold` et `iter` de ce foncteur.

1. On souhaite d'abord compter le nombre d'occurrences de chaque mot du texte. Donner l'algorithme de haut niveau permettant de faire ce calcul. Vous donnerez en particulier :
   - les structures de données à utiliser
   - les opérations à faire

   On peut, pour cette question faire une réflexion collective dans le groupe de TD comme cela a été fait en cours. On ne s'intéresse pas dans un premier temps à la façon d'implémenter cela en OCaml.

1. libre de droits et récupéré sur `https://www.gutenberg.org/`

Réponse: On peut procéder de la façon suivante :
— ouvrir le fichier en lecture
— parcourir l'ensemble de ses lignes en maintenant un dictionnaire associant des mots à des entiers
— pour chaque ligne, la transformer en liste de mots
— pour chaque mot rechercher le nombre d'occurrences de ce mot dans le dictionnaire (si le mot n'a pas encore été vu, on prend 0)
— à jouter 1 au nombre d'occurrences et mettre à jour le dictionnaire
— (en fin de parcours, avant de renvoyer le dictionnaire, penser à fermer le fichier)

2. Implémenter l'algorithme en question et le tester sur le fichier `jungle2.txt`

Réponse: Cf `exo1/main.ml`

3. Tester le programme en écrivant une fonction qui affiche pour chaque mot son nombre d'occurrences. Expliquer pourquoi les mots sont listés par ordre alphabétique.

Réponse: Comme vu en cours, la structure de données sous-jacente des `Map`s d'OCaml est un arbre binaire de recherche, la fonction `iter` parcours l'arbre par ordre croissant des clé.

4. On souhaite maintenant afficher pour chaque nombre d'occurrences n la liste des mots apparaissant exactement n fois (en séparant les mots par des virgules). Par exemple, pour le fichier `jungle2.txt` on souhaite que l'affichage soit :

```
...
166: WHO,THEN,ARE
169: DO
173: SO
177: WHAT,ME
...
5530: THE
```

indiquant que les mots « who », « then » et « are » apparaissent chacun 166 fois, et « the » 5530 fois. En supposant que l'on dispose du dictionnaire associant chaque mot à sa fréquence (calculé en question 2), donner l'algorithme permettant de calculer le résultat demandé.

Réponse: On applique l'algorithme suivant :
— parcourir chaque couple (w, c) du dictionnaire (w est un mot et c son nombre d'occurrences), en maintenant un dictionnaire dont les clés sont des entiers
— rechercher dans le dictionnaire maintenu la liste des mots apparaissant c fois. Si c'est la première fois que l'on rencontre c, renvoyer la liste vide.
— ajouter w à la liste de mots et mettre à jour le dictionnaire

5. implémenter l'algorithme et le tester. Pour l'affichage, la fonction

```
String.concat : string -> string list -> string
```

permet de concaténer les chaînes de la liste données avec un séparateur donné. Par exemple

```
String.concat "-" ["a"; "b"; "c"]
```

renvoie `"a-b-c"`

6. Pourquoi pour chaque fréquence la liste de mot s'affiche en ordre inverse ?
7. Pourquoi a-t-on la garantie qu'une liste de mots ne contient jamais de doublon ?

Réponse: Pour le code du corrigé, on parcourt le dictionnaire mots avec `iter`, donc par ordre croissant de mots. Comme c'est un dictionnaire, les clés sont uniques et on sait donc que chaque mot apparait exactement une fois dans le dictionnaire et donc dans les listes construites. Pour construire la liste, on place les mots trouvés en début de liste au fur et à mesure du parcours, donc les mots du dictionnaire visités en dernier se trouvent en début de liste.

2 Du *Snowboard* à Alanis Morissette

Cette exercice est conceptuellement plus complexe, mais il y a peu de code à écrire. Il sert à illustrer comment les modules et foncteurs d'OCaml permettent d'obtenir d'instancier des algorithmes complexes sur des cas particuliers.

On dispose du graphe de wikipedia *simple english* (une version simplifiée de Wikipedia à destination des enfants et des personnes apprenant l'anglais). Ce graphe est constitué de deux fichiers. Un fichier `simple-pages.txt` qui donne sur des lignes un identifiant unique de page et le nom de cette page :

```
1
April
2
August
6
Art
8
A
9
Air
12
Autonomous_communities_of_Spain
13
Alan_Turing
14
Alanis_Morissette
...
```

Ainsi, la page d'identifiant 13 est

`https://simple.wikipedia.org/wiki/Alan_Turing`

Un autre fichier, `simple-links.txt` donne pour chaque identifiant de page la liste des identifiants des pages pointées :

```
...
132666 64169 520
...
```

indiquant que la page 132666 :

`https://simple.wikipedia.org/wiki/Antioxidant`

contient deux liens vers les pages 64169 et 520 :

`https://simple.wikipedia.org/wiki/Redox`
`https://simple.wikipedia.org/wiki/Molecule`

On souhaite connaître le chemin le plus court entre 2 pages. Pour cela un algorithme de graphe bien connu est celui du plus court chemin de Dijkstra. Il est fourni par la bibliothèque `ocamlgraph` que l'on se propose d'utiliser.

2.1 ocamlgraph

La bibliothèque `ocamlgraph`[2] est une bibliothèque fournissant de nombreux algorithmes sur les graphes (plus courts chemins, partitionnement, parcours, flot, ...). Son interface repose sur les foncteurs. Un utilisateur de la bibliothèque doit donner un module ayant la signature suivante :

```
1  module type G :
2  sig
3    type t
4    (** le type des graphes *)
5
6    (** un sous module V représentant les sommets et opérations associées *)
7    module V: sig
8      type t
9      val compare : t -> t -> int
10     val hash : t -> int
11     val equal : t -> t -> bool
12   end
13
14   (** un sous-module E représentant les arrêtes et opérations associées *)
15   module E: sig
16     type t
17     type label (* type d'étiquette d'une arrête *)
18     val label : t -> label
19     val src : t -> V.t
20     val dst : t -> V.t
21     val create : V.t -> label -> V.t -> t
22   end
23
24   (** [iter_vertex f g] appelle la fonction [f] tour a tour
25       sur tous les sommets du graphe [g]*)
26   val iter_vertex : (V.t -> unit) -> t -> unit
27
28   (** [fold_vertex f g] calcule [(f vn .... (f v2 (f v1 acc)))]
29       où les [vi] sont les sommets du graphe. *)
30   val fold_vertex : (V.t -> 'a -> 'a) -> t -> 'a -> 'a
31
32   (** [iter_succ f g v] appelle [f] tour à tour sur tous
33       les voisins de [v] *)
34   val iter_succ : (V.t -> unit) -> t -> V.t -> unit
35
36   (** [iter_succ_e f g v] appelle [f] tour à tour sur toutes les arrêtes
37       de [(v,w)] où [w] est un voisin de [v] *)
38   val iter_succ_e : (E.t -> unit) -> t -> V.t -> unit
39
40   (** [fold_edges_e f g acc]  calcule [(f em .... (f e2 (f e1 acc)))]
41       où les [ei] sont les arrêtes du graphe.
42   *)
43   val fold_edges_e : (E.t -> 'a -> 'a) -> t -> 'a -> 'a
44
45   (** [nb_vertex g] renvoie le nombre de sommets *)
46   val nb_vertex : t -> int
```

Cette signature représente les opérations minimales nécessaires sur les graphes pour implémenter les divers algorithmes.

Pour l'algorithme de Dijkstra en particulier on a en plus besoin d'un module `Weight` représentant les

2. Si vous n'êtes pas au PUIO, installer avec `opam install ocamlgraph`

poids des arrêtes

```
module type WEIGHT : sig
type edge (* type des arrêtes, doit être égal à G.E.t ci-dessus *)
type t (* type des poids,
          généralement un type numérique comme int ou float*)

(** récupérer le poids d'une arrête *)
val weight : edge -> t

(** comparer 2 poids *)
val compare : t -> t -> int

(** ajouter 2 poids *)
val add : t -> t -> t

(** élément neutre pour [add] (utilisé pour initialiser) *)
val zero : t
```

Une fois ces modules fournis, l'utilisateur peu simplement faire :

```
module D = Graph.Path.Dijkstra(MyGraph)(MyWeigth)

let path, len = D.shortest_path graph v1 v2
```

où `MyGraph` et `MyWeight` sont deux modules implémentant les signatures ci-dessus. La fonction `shortest_path` renvoie le chemin de poids minimal sous forme de la liste des arrêtes à traverser pour aller de `v1` à `v2` ainsi que le poids du chemin.

1. Lire les fichiers `wiki.ml` et `main.ml` du répertoire `exo2` (on pourra commencer par le fichier `main.ml` pour comprendre la finalité du programme). Repérer dans `wiki.ml` les définitions des sous-modules `V` et `E` ainsi que du type `t` représentant un graphe, puis répondre aux questions suivantes :
   — par quel type représentant un sommet ?
   — par quel type représentant une arrête ?
   — comment sont stockés l'ensemble des sommets dans le graphe ?
   — comment sont stockés l'ensemble des arrêtes dans le graphe ?

   Réponse:
   — les sommets sont représentés par des entiers (`V.t = int`)
   — les arrêtes sont représentées par des paires d'entiers (`E.t = int * int`)
   — l'ensemble des sommets est un `IntSet` un ensemble OCaml d'entiers obtenu par le foncteur `Set.Make`
   — l'ensemble des arrêtes est un dictionnaire `IntMap` dont les clés sont des entiers et les valeurs la liste des voisins du sommets représentant la clé.

2. compléter les fonctions `iter_*` et `fold_*` pour qu'elles correspondent à leur spécification.
3. tester votre programme
4. reformuler le module `Weight` du fichier `wiki.ml` pour ne pas copier les définitions de `t compare add` et `zero`, déja présentes dans le module `Int` de la bibliothèque standard.