TP 28

Compilation OCaml
Parcours de graphes

1 Compilation OCaml

On ne l'a pas encore utilisé de la sorte, mais OCaml est avant tout un langage compilé (tout comme C).

Exemple. Voici un exemple de fichier `.ml` :

```
let rec fact n =
  match n with
  | 0 -> 1
  | _ -> n * fact (n - 1)

let () =
  Printf.printf "La factorielle de %d vaut %d.\n" 5 (fact 5);
  exit 0
```

Le fichier se termine par un « `let` » de type `unit` : on peut voir cette dernière définition comme l'équivalent la fonction `main` d'un programme C (et `exit` permet de fournir le code de retour du programme). On compile ce fichier dans un terminal avec la commande `ocamlc` ou `ocamlopt` [1]

```
# ocamlc -o factorial factorial.ml
# ./factorial
La factorielle de 5 vaut 120.
```

```
# ocamlopt -o factorial factorial.ml
# ./factorial
La factorielle de 5 vaut 120.
```

Arguments en ligne de commandes

Il est aussi assez aisé de récupérer des arguments passés en ligne de commandes en utilisant le tableau `Sys.argv`.

1. pour nous, les différences sont minimes entre ces deux commandes : la première renvoie un exécutable en *byte-code* (c'est-à-dire en code bas-niveau mais qui a encore besoin d'un interpréteur pour être lancé), la seconde compile en code natif.

```
let rec fact n =
  match n with
  | 0 -> 1
  | _ -> n * fact (n - 1)

let () =
  if Array.length Sys.argv <> 2
  then (
    Printf.printf "La fonction prend un argument\n";
    exit 1)
  else (
    let n = int_of_string Sys.argv.(1) in
    Printf.printf "La factorielle de %d vaut %d.\n" n (fact n);
    exit 0)
```

```
ocamlc -o factorial factorial_arg.ml
# ./factorial
La fonction prend un argument
# ./factorial 5
La factorielle de 5 vaut 120.
```

Séparer les fichiers

On peut bien entendu séparer le code dans plusieurs fichiers.

- Fichier `factorial.ml` :

```
let rec fact n =
  match n with
  | 0 -> 1
  | _ -> n * fact (n - 1)
```

- Fichier principal `main.ml` (le nom `main` est ici complètement arbritraire) :

```
let () =
  let n = 5 in
  Printf.printf "fact(%d) = %d\n" n (Factorial.fact n);
  exit 0
```

On remarque que l'appel à la fonction `fact` qui est définie dans le premier fichier se fait avec le préfixe `Factorial.` (remarquez bien la majuscule qui se rajoute au nom du fichier).
Il est possible d'éviter ce préfixe (par exemple si on doit beaucoup utiliser les fonctions d'un module) avec la commande `open` :

- Fichier principal (seconde version) :

```
open Factorial

let () =
  let n = 5 in
  Printf.printf "fact(%d) = %d\n" n (fact n);
  exit 0
```

On compilera alors les deux fichiers en même temps :

```
# ocamlc -o factorial factorial.ml main.ml
```

```
# ocamlopt -o factorial factorial.ml main.ml
```

L'ordre de passage des fichiers au compilateur est important (il y a un DAG sous-jacent!) : le fichier `factorial.ml` doit être compilé avant le fichier `main.ml` car ce dernier utilise une fonction du premier.

Avec une interface restreinte

Par défaut, toutes les fonctions définies dans un fichier `.ml` seront accessibles si on l'utilise depuis un autre fichier. On peut limiter ce comportement et définir précisément quels sont les types et fonctions accessibles depuis l'extérieur en construisant l'*interface* du module. C'est le rôle du fichier `.mli`.

Exemple. Dans la suite, je vous propose d'écrire des parcours de graphes avec la structure de données « *sac* ». Celle-ci est construite dans le fichier `sac.ml` et son interface est fournie dans le fichier `sac.mli`.

La compilation nécessite alors une étape supplémentaire : il faut compiler l'interface auparavant avec l'option `-c` (un fichier `.cmi` est alors créé) :

```
# ocamlc -c sac.mli
```

```
# ocamlopt -c sac.mli
```

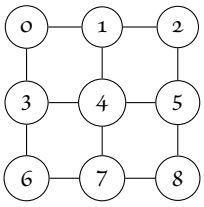
11 Whatever-first search

On souhaite ici écrire un programme qui implémente le parcours de graphes général qui a introduit le cours sur les parcours.
Pour cela, on fournit donc la structure de données de *sac* qui permet essentiellement, une fois un sac vide créé, d'ajouter des objets dedans et de retirer un objet aléatoirement [2]

1. Écrire une fonction `parcours`, prenant un argument un graphe et un sommet de ce graphe, qui réalise ce parcours. Cette fonction devra renvoyer la liste des sommets visités, dans l'ordre où ils ont été visités.

   ```
   parcours : graph -> vertex -> vertex list
   ```

2. Écrire une fonction `graphe_carre` qui construit un graphe « carré » de taille n. Par exemple, pour $n = 3$, on doit obtenir :


   ```
   graphe_carre : int -> graph
   ```

2. *aléatoire* est ici un bien grand mot… Si vous lancer plusieurs fois votre programme, vous devez obtenir exactement les mêmes résultats. Il en est ici car j'ai initialisé la graine de mon générateur aléatoire avec une valeur fixe. Pour changer ce comportement, il faut aller modifier la ligne `Random.init 861` dans le fichier `sac.ml`. Pour obtenir des résultats différents à chaque essai, on pourra la remplacer par `Random.self_init ()`.

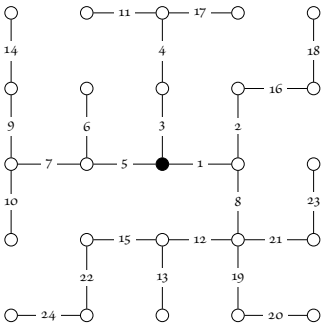
Les fichiers `affichage.ml` et `affichage.mli` fournissent une interface pour affiche dynamiquement dans le terminal. Pour réaliser cet affichage, j'ai fait appel au module `Unix` (donc, ça fonctionnera bien sous Linux ; normalement, ça devrait tourner sous Windows avec WSL ; pour le reste, je ne garantis rien). Ce module n'étant pas standard, il faudra rajouter une petite précision à la compilation :

```
# ocamlc -o ... unix.cma ... affichage.ml ... parcours.ml
```

```
# ocamlopt -o ... unix.cma ... affichage.ml ... parcours.ml
```

3. Utiliser cette interface pour visualiser votre parcours.

On souhaiterait maintenant être en mesure de reconstruire l'*arbre de parcours*. Par exemple, pour $n = 5$ en partant du nœud central, on construierait cet arbre :



4. Modifier votre parcours pour **être en mesure** de reconstruire cet arbre [3].

III Depth-first search

Pour utiliser une pile, vous pouvez utiliser le module `Stack` [4] (il est dans la bibiothèque standard, donc il n'y a rien à ajouter à la compilation). Je vous laisse aller découvrir dans la documentation l'interface de ce module.

5. Écrire un parcours en profondeur à l'aide d'une pile.

   ```
   dfs : graph -> vertex -> vertex list
   ```

   Le tester !

6. Écrire un parcours en profondeur récursif.

   ```
   dfs_rec : graph -> vertex -> vertex list
   ```

   Le tester !

IV Breadth-first search

Pour utiliser une file, vous pouvez utiliser le module `Queue`... (la suite du blabla est identique).

7. Écrire un parcours en largeur à l'aide d'une file.

   ```
   dfs : graph -> vertex -> vertex list
   ```

   Le tester !

3. si vous connaissez un outil permettant de produire effectivement ce graphe, faites-vous plaisir...
4. vous pouvez aussi en réimplenter une...