université de BORDEAUX	**ANNÉE UNIVERSITAIRE:** 2019/2020 **PARCOURS:** Licence Informatique 2e année UE 4TINA01U **ÉPREUVE:** Tp Noté de Programmation fonctionnelle **DATE:** jeudi 12 Décembre 2019 15:30 – 16:50 **DURÉE:** 1h20	**Collège Sciences et Technologies**

Vous pouvez utiliser les fonctions du module `List`

Consignes

1. Depuis `Moodle`, téléchargez et décompressez l'archive `NOM-PRENOM.zip` contenant le squelette du tp.
2. Renommez le répertoire `NOM-PRENOM` en remplaçant `NOM` et `PRENOM` par vos nom et prénom.
   Exemple : `TURING-ALAN`.
3. Placez-vous dans le répertoire ainsi renommé.

   Ce répertoire contient le fichier `graph.ml` que vous aurez à compléter. Il contient des exemples de graphes et la liste des fonctions que vous devez écrire.
4. Indiquez à nouveau vos nom et prénom et votre numéro de groupe dans le fichier `graph.ml`.
5. Pensez à sauver régulièrement votre travail.
6. À la fin du tp,
   a. Déposez le fichier `graph.ml` à l'endroit prévu sur Moodle.
   b. Créez une archive `<NOM>-<PRENOM>.zip` à partir de votre répertoire et envoyez là par mail à votre chargé de CI.

Graphes

Un graphe est une modélisation d'un ensemble d'objets reliés entre eux. Les objets sont appelés **sommets**, et les liens entre les objets sont appelés **arêtes**. On considère des graphes **orientés** : chaque arête relie un premier sommet (appelé origine de l'arête) à un second sommet (appelé extrémité de l'arête).

On utilise des entiers pour numéroter les sommets d'un graphe : si il contient `n` sommets, on numérote ceux-ci de `1` à `n`. Cette notation nous permet de représenter une arête par un couple d'entiers : l'arête allant du sommet numéro `i` vers le sommet numéro `j` est représentée par le couple `(i,j)`. On représente donc un graphe par deux éléments : son *nombre de sommets* et son *ensemble d'arêtes*. Ainsi, on utilise le type *OCaml* suivant :

```
type graph = G of int * (int * int) list ;;
```

Un graphe est donc représenté par un élément `G(n,l)` où `n` est un entier (type `int`) représentant le nombre de sommets dans le graphe et `l` est une liste (de type `(int * int) list`) représentant l'ensemble des arêtes dans le graphe. Par exemple, considérons, le code OCaml suivant :

```
# let exemple_01 = G(8,[(1,2);(2,4);(2,5);(3,1);
(5,2);(5,7);(6,7);(7,6);(7,7)]);;
val exemple_01 : graph =
G (8,
 [(1, 2); (2, 4); (2, 5); (3, 1); (5, 2); (5, 7); (6, 7); (7, 6); (7, 7)])
```

La variable `exemple_01` de type `graph` représente un graphe à 8 sommets (numérotés de 1 à 8) dont on donne la représentation graphique dans la Figure 1 ci-dessous :

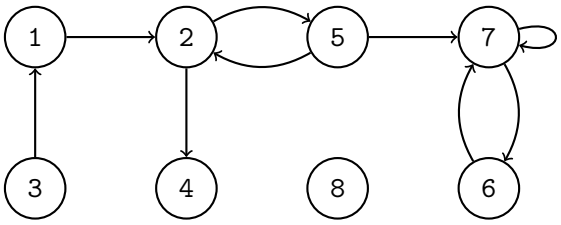


FIG. 1: Graphe représenté par la variable `exemple_01`

D'autres exemples sont donnés dans le fichier `graph.ml`

Remarque

Les éléments de type `graph` ne sont pas tous la représentation correcte d'un graphe. Par exemple, considérons la variable suivante :

```
# let bad = G(2,[(1,2);(42,2)]);;
```

La variable `bad` n'est pas une représentation correcte car on déclare un graphe à 2 sommets mais la liste d'arêtes dépend du sommet numéro 42 (qui ne fait pas partie du graphe).

Un autre point est qu'on se sert de listes pour représenter des ensembles d'arêtes. La conséquence est que la représentation d'un graphe donne un ordre sur les arêtes et peut mentionner la même arête plusieurs fois. Cela n'impacte pas le graphe représenté. Par exemple, considérons les éléments suivants :

```
# let same_01 = G(3,[(1,2);(2;3)]);;
# let same_02 = G(3,[(2;3);(1,2)]);;
# let same_03 = G(3,[(1,2);(1,2);(2,3);(1,2);(2;3)]);;
```

Les trois variables ci-dessus sont trois représentations du *même* graphe décrit graphiquement ci-dessous :

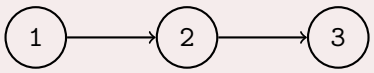


FIG. 2: Graphe représenté par les variables `same_01`, `same_02` et `same_03`

Exercice 1 *Exemples de graphes*

1. Définir une variable `exemple_02` de type `graph` qui représente le graphe donné dans la Figure 3 suivante :

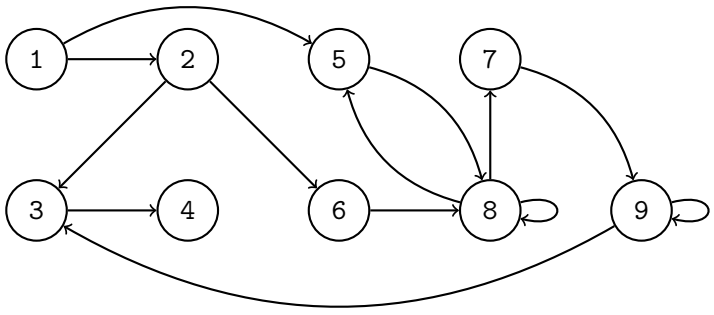
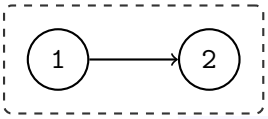
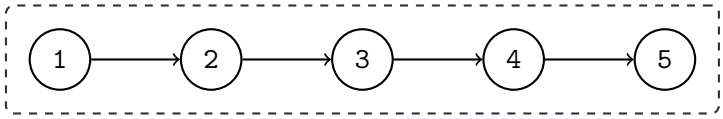


FIG. 3: Graphe devant être représenté par la variable `exemple_02`

2. Écrire une fonction `gen_path` de type `int -> graph` qui prend un entier `n` en entrée et retourne le graphe à `n` sommets qui est constitué d'un unique chemin reliant le sommet `1` au sommet `n`. On représente ce graphe pour `n = 2` et `n = 5` dans la Figure 5 ci-dessous :



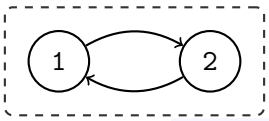
Graphe retourné par `gen_path 2`



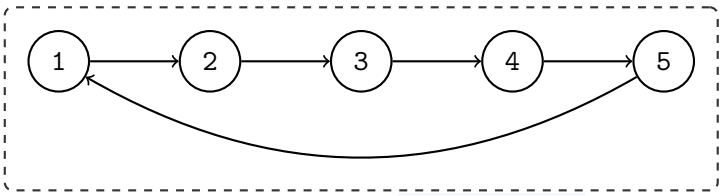
Graphe retourné par `gen_path 5`

FIG. 4: Chemins de taille 1 et 4

3. Écrire une fonction `gen_cycle` de type `int -> graph` qui prend un entier `n` en entrée et retourne le graphe cyclique de taille `n`. On représente ce graphe pour `n = 2` et `n = 5` dans la Figure 4 ci-dessous :



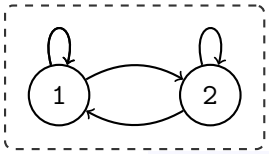
Graphe retourné par `gen_cycle 2`



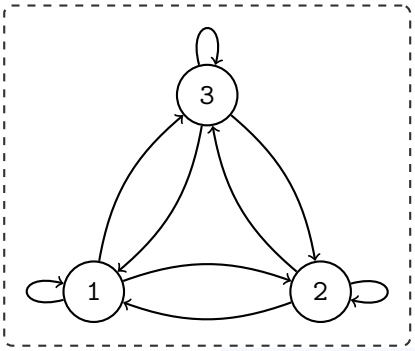
Graphe retourné par `gen_cycle 5`

FIG. 5: Cycles de taille 2 et 5

4. Écrire une fonction `gen_complete` de type `int -> graph` qui prend un entier `n` en entrée et retourne le graphe complet à `n` sommets. Celui-ci contient toutes les arêtes possibles entre ses sommets. On représente ce graphe pour `n = 2` et `n = 3` dans la Figure 6 ci-dessous :



Graphe retourné par `gen_complete 2`



Graphe retourné par `gen_complete 3`

FIG. 6: Graphes complets de taille 2 et 3

Exercice 2 *Fonctions sur les graphes*

1. Écrire une fonction qui teste si une variable de type `graph` est la représentation correcte d'un graphe (voir la remarque au début du sujet). Il faut donc vérifier que la liste d'arêtes ne mentionne que des sommets qui sont valides (c'est-à-dire que leur numéro doit être strictement positif et inférieur ou égal au nombre de sommets déclaré).

```
val is_correct : graph -> bool = <fun>
# is_correct  exemple_01;;
- : bool = true
# is_correct  (G(2,[(1,2);(1,42)]));;
- : bool = false
# is_correct  (G(2,[(1,2);(1,-4)]));;
- : bool = false
```

2. Écrire une fonction qui, étant donné un graphe `g`, teste si pour toute arête allant d'un sommet `i` vers un sommet `j`, il existe aussi une arête dans le sens inverse (c'est-à-dire allant du sommet `j` vers le sommet `i`).

```
val is_inverse : graph -> bool = <fun>
```

```
# is_inverse  exemple_01;;
- : bool = false
# is_inverse (gen_complete 3);;
- : bool = true
```

3. Écrire une fonction qui "simplifie" la représentation d'un graphe. Étant donné un élément de type `graph` la fonction doit retourner un autre élément de type `graph` représentant le même graphe mais dont la liste d'arêtes ne mentionne chaque arête qu'une seule fois.

```
val simplify : graph -> graphe = <fun>
```

```
# simplify  (G(3,[(1,3);(2,1);(2,1);(3,1);(1,3);(1,3);(2,1)]));;
- : graph = G(3,[(3,1);(1,3);(2,1)])
```

4. Écrire une fonction qui retourne le nombre d'arêtes d'un graphe.

```
val number_of_edges : graph -> int = <fun>
```

5. Écrire une fonction qui prend en entrée un graphe `g` et un sommet `i` et retourne la liste d'adjacence de `i`. C'est-à-dire la liste de tous les sommets `j` tels que le graphe `g` contient l'arête `(i,j)`.

```
val list_adj : graph -> int -> int list = <fun>
```

```
# list_adj exemple_01 7;;
- : int list = [6;7]
# list_adj exemple_01 5;;
- : int list = [2;7]
```

6. Un *chemin* entre deux sommets `i` et `j` est une suite d'arêtes qui relie `i` à `j`. Sa longueur est le nombre d'arêtes sur celui-ci. Par exemple dans le graphe `exemple_01`, on a le chemin $1 \to 2 \to 5 \to 7$ de longueur 3 du sommet 1 vers le sommet 7. On a aussi le chemin $1 \to 2 \to 5 \to 2 \to 5 \to 7$ de longueur 5 qui va aussi du sommet 1 vers le sommet 7.
Écrire une fonction qui prend en entrée un graphe, deux sommets `i` et `j`, et en entier `n` (dans cet ordre). La fonction devra retourner `true` si il existe un chemin de *longueur inférieure ou égale* à `n` entre `s` et `t` et `false` sinon. On pourra utiliser une récursion sur le nombre `n` ainsi que la fonction `list_adj` de la question précédente.

```
val bounded_path : graph -> int -> int -> int -> bool = <fun>
```

```
# bounded_path exemple_01 1 7 2;;
- : bool false
# bounded_path exemple_01 1 7 3;;
- : bool true
```

7. Écrire une fonction qui prend en entrée un graphe ainsi que deux sommets `s` et `t`. La fonction devra retourner `true` si il existe un chemin de *longueur quelconque* entre `s` et `t` et `false` sinon.

```
val exists_path : graph -> int -> int -> bool = <fun>
```

```
# exists_path exemple_01 1 7;;
- : bool true
# exists_path exemple_01 6 1;;
- : bool false
```

FIN